ABSTRACT

A Model of Clocked Electric Field Inputs for Molecular Quantum-dot Cellular Automata

Jackson Alan Henry, M.S.E.C.E.

Mentor: Enrique P. Blair, Ph.D.

Quantum-dot cellular automata (QCA) is a low-power, high-speed, beyond-CMOS approach to general-purpose computing [1]. Elementary devices called "cells" are implemented using mixed-valence molecules with redox centers having a few quantum dots. These support three distinct localized electronic states labeled "0", "1", and "Null". Cells can be clocked to either the "Null" state or an active ("0" or "1") state using the vertical component of an applied electric field. Clocking provides power gain for restoring weakened signals and allows synchronous control of QCA circuits.

In this paper, clocked molecular QCA circuits are simulated in the presence of an applied input field, using the intercellular Hartree-Fock approximation [2]. Input circuits and down-stream circuits function in the presence of the input field and unwanted field fringing from electrodes. This emphasizes that widely-available fabrication techniques may be used to form electrodes for writing bits to molecular QCA circuits.

A Model of Clocked Electric Field Inputs for Molecular Quantum-dot Cellular Automata

by

Jackson Alan Henry, B.S.E.C.E.

A Thesis

Approved by the Department of Electrical and Computer Engineering

---

Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

Enrique P. Blair, Ph.D., Chairperson

Tomas Cerny, Ph.D.

Scott Koziol, Ph.D.

Accepted by the Graduate School
December 2019

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First I would like to thank Dr. Blair for all of his guidance and his help throughout my entire time at Baylor.

Besides my advisor, I would like to thank the rest of my thesis committee for letting my defense be an enjoyable moment. I gratefully acknowledge their time and valuable feedback on this thesis

My sincere thanks goes to my wonderful labmates: Nishat, Joe and Shenghyang, for all the of our thought provoking discussions we had in the last two years.

I would like to thank my family for all their support and love while completing my degree.

And, I have saved this last word of acknowledgment for Elizabeth. You have kept me solid and focused. Thank you for all of your love and encouragement no matter what is happening. I love you, Elizabeth.

DEDICATION

To all those who made this possible

CHAPTER ONE

Introduction and Motivation

Introduction

The design and operation of transistors at the physical limits of scaling has led to significant power dissipation [5], the real and very present limit to progress. Present fabrication technologies support device densities of over 10^9 cm^{-2}, but full device utilization will dissipate levels of heat that can destroy the chip. While design techniques such as multi-core processors and dark silicon mitigate actual levels of dissipation, the ability to sink this heat dissipation has limited commercial computer clock speeds to 3-4 GHz since 2003. Additionally, power dissipation contributes directly to the significant and growing percentage of global electrical power consumed by information and computing technologies, calculated at 10% in 2010 and estimated to reach 30-50% in 2030 [6]. Clearly, to achieve energy-efficient, next-generation computing technologies, solutions beyond CMOS must be considered.

To answer the concerns raised by power dissipation in CMOS, a low-power, general-purpose, digital computational paradigm known as quantum-dot cellular automata (QCA) was developed [1]. In particular, a molecular implementation of QCA promises nanometer-scale devices [7, 8, 9], THz-or-faster switching speeds [10], and room-temperature operation. While QCA devices have been fabricated [11] and the concept of information processing using QCA circuits has been established [12, 13], technical challenges persist in the realization of molecular QCA.

The discussion begins with Section 1, which provides a brief overview of the QCA concept and emphasizes the molecular implementation. The concept of clocking in molecular QCA circuits is presented. The conclusion to Section 1 sets the context for my work by briefly overviewing the technical challenges which must be overcome

if computation using molecular QCA is to be achieved, including: molecule design; molecular QCA circuit layout; bit inputs to the QCA molecules; and bit readout from molecular QCA circuits. In particular, this work proposes a solution for bit inputs to clocked molecular QCA circuits using electrodes which may be fabricated using mature nanolithographic processes. Chapter 2 describes the model used for describing a QCA circuit immersed in an electric field $\vec{E}(\vec{r})$ which provides both bit inputs and clocking. Simulation results of bits selected on clocked molecular QCA circuits are presented in Chapter 3. This solution for bit write-in is important because it may enable the experimental demonstration of controlled switching of QCA molecules, which in turn, will support the development and testing bit read-out technologies. This discussion is concluded in Chapter 4 by underscoring the fact that novel solutions to one technical challenge in realizing molecular QCA can help enable solutions to other challenges.

Overview of QCA

In QCA, the elementary device is a *cell*, a structure with a set of quantum dots which provide charge localization sites for a few mobile charges. The configuration of charge on these dots encodes a bit, and device switching occurs via the quantum tunneling of charge between the dots.

Figure 1.1 depicts three states of a cell with six dots and two mobile electrons. The two states, labeled "0" and "1" are designated active states, and the third state is designated as "Null," conveying no information. The cell can be clocked to the Null state by applying a positive voltage sufficient to attract the electrons to the central null dots (dots 2 and 5). We refer to dots 1, 3, 4, and 6 as active dots. For this cell, it is a design feature that direct tunneling between "0" and "1" is suppressed: the transition between either of the active states requires an intermediate transition to the "Null" state.

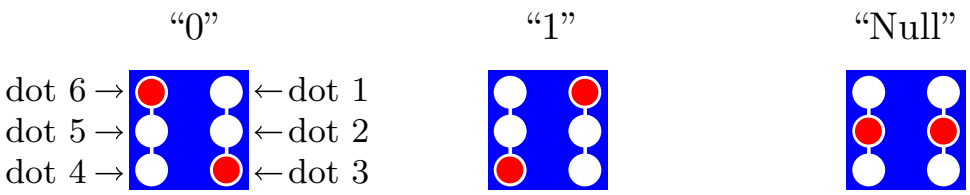

Figure 1.1: Charge-localized states of a six-dot QCA cell. Two mobile electrons (red discs) provide three states on a system of six quantum dots (white discs). The thin, white connecting lines between dots indicate tunneling paths. Thus, a transition between the "0" and "1" states requires an intermediate transition to the "Null" state.

When arranged on a substrate, neighboring cells are networked locally via the electrostatic field, enabling general-purpose computation. Basic QCA circuits are shown in Figure 1.2. Cells arranged in a row tend to align via simple Coulomb repulsion, and diagonal coupling can be used to achieve a bit inversion. The natural logic gate in QCA is the majority gate, for which three inputs have each an equal influence over a device cell. The bit in the majority on the inputs appears on the device cell and is copied to the output. Any one of the three inputs can function as a control bit, making the majority gate function as a programmable two-input AND/OR gate between the other two inputs. These devices provide a logically-complete set, from which more complex devices may be formed. QCA circuits such as adders and a Simple-12 processor have been designed [12, 13].

Molecular QCA

There are various implementations for QCA. The earliest implementation of QCA used metallic dots patterned on an insulating substrate [11, 14]. Here, tunnel junctions allow inter-dot charge transfer. Later, semiconductor quantum dots were used [15, 16]. Also, cells have been written on a hydrogen-passivated silicon surface using a scanning tunneling microscope (STM) tip: individual H atoms were removed, exposing single dangling bonds, each of which functions as a dot [17]. By virtue of nanometer-scale dimensions, these atomic-scale QCA cells have bit energies of the

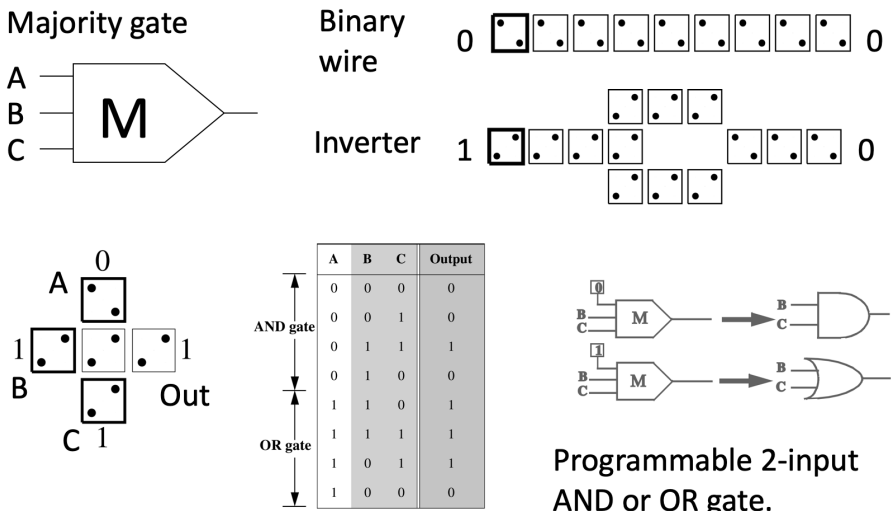

Figure 1.2: Basic QCA devices. In the upper right cells arranged in a row function as a binary wire, as cells align through Coulomb repulsion. Just below this, diagonal interactions between cells provide a bit inversion. On the left is a majority gate, which has three inputs, A, B, and C, which vote on the state of the central device cell. $M(A, B, C)$ is the bit in the majority among the inputs, which appears on the device cell and gets copied to the ouput. One of inputs may be used as a control bit to program the gate to function as a programmable, two-input AND/OR gate between the other two inputs.

electron-volt scale and are robust at room temperature. Molecular QCA also support room-temperature operation and are the focus of this thesis. Here, mixed-valence molecules function as QCA cells, and redox centers on those molecules provide dots [8, 9]. Molecular QCA also may support THz-scale or better device operation [10].

The zwitterionic nido carborane molecule ($Fc^+FcC_2B_9^-$) was designed specifically for use as a QCA molecule [18]. $Fc^+FcC_2B_9^-$ is a self-doping molecule with a net-zero charge which provides three dots, as shown in Figure 1.3. In $Fc^+FcC_2B_9^-$, the mobile charge is a single hole. Two such molecules can be paired to function as a six-dot QCA cell like the one depicted in Figure 1.1.

A key advantage of molecular QCA is the small length scale of molecular cells. This length scale enables ultra-high device densities at 10^{14} cm^{-2} for tight-packed, 1-nm cells. The molecular QCA length scale also provides bit energies robust at room temperature. Consider a pair of two-dot QCA molecules of length a, separated by

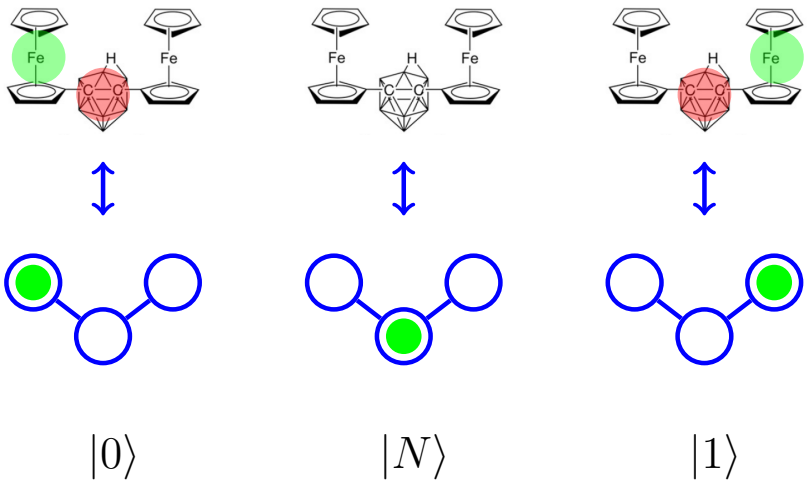

Figure 1.3: A zwitterionic nido carborane ($Fc^{+}FcC_2B_9^{-}$) molecule is designed to function as a three-dot QCA cell. Two iron centers and one central carborane cage provide one quantum dot each. Three charge configurations of $Fc^{+}FcC_2B_9^{-}$ are depicted in the top row of this figure. When one mobile hole (translucent green disc) occupies the either iron center, it uncovers one electron (translucent red disc) on the central (null) dot. These are the active states "0" ($|0\rangle$) or "1" ($|1\rangle$). In the "Null" state ($|N\rangle$), the hole occupies the carborane cage, masking the fixed electron. The states of three-dot molecule are shown schematically in the bottom row of this graphic (the fixed electron is not shown).

distance a, as shown in Fig. 1.4. We define the kink energy, E_k, as the cost of a bit flip. This is the difference in electrostatic energy between the kinked configuration of two cells [Figure 1.4(b)] and their relaxed (favored) configuration [Figure 1.4(a)]. It can be shown that the kink energy depends on the length a of the cells and physical constants ϵ_0 (the permittivity of free space), q (the fundamental charge):

$$E_k = \frac{q^2}{4\pi\epsilon_0 a}\left(1 - \frac{1}{\sqrt{2}}\right) . \tag{1.1}$$

The kink energy also defines the bit energy for the double-dot QCA cell. Equation (1.1) also holds for three-dot QCA cells.

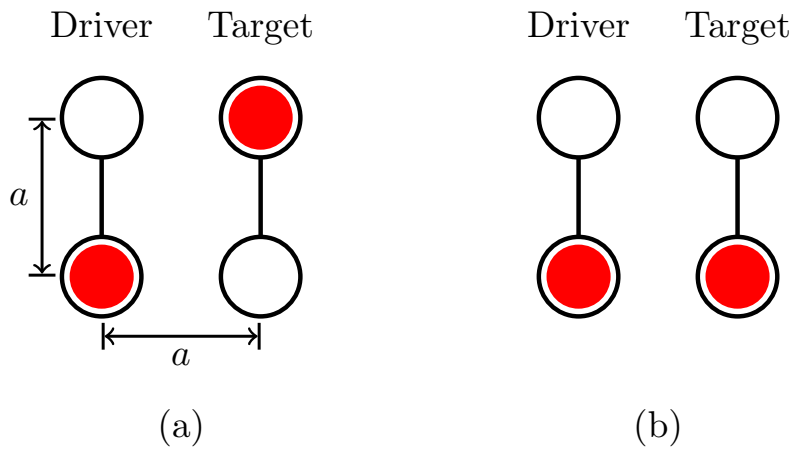

Figure 1.4: The bit energy for a molecular two-dot cells is the kink energy, the difference in electrostatic energies of the two-cell favored configuration [subfigure (a)] and the kinked configuration [subfigure (b)], given that the driver is fixed as shown.

Clocked Molecular QCA

Molecular six-dot cells may be clocked using an externally-applied electric field [19], as depicted in Figure 1.5. Here, the six-dot cell is attached to the substrate by the null dots, so that the active dots are elevated above the substrate. A voltage applied to a conducting slab buried beneath the molecule results in an electrostatic field with a vertical component that affects the state of the cell. In this depiction, the mobile charge is a pair of electrons, so that a negative voltage applied to the slab repels the electrons from the null dots, forcing the cell to take an active state determined by interaction with neighboring cells. A positive clocking voltage, on the other hand, establishes an electric field which attracts the mobile electrons to the null dots, and the cell takes the null state regardless of neighbor interactions. Since direct tunneling between active states is suppressed, clocked molecular QCA support latching. Once the cell is clocked to an active state X, it is latched in X until the clock is lowered: a bit flip $X \rightarrow \overline{X}$ first requires a reset to the null state, which is prevented by the clock.

Arrays of independently-charged conductors can be used to create an inhomogeneous electric at the device layer, as shown in Figure 1.6. Some domains of the field

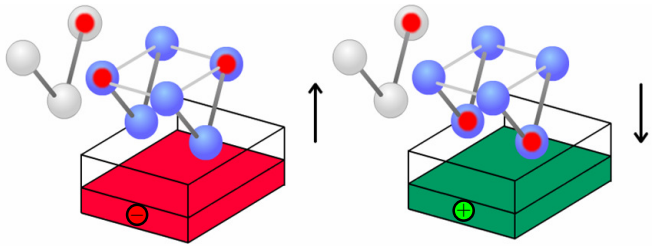

Figure 1.5: A molecular six-dot cell is clocked using an externally-applied electric field. Light-blue spheres represent the quantum dots of a six-dot cell, and a red sphere represents a single mobile electron. The cell is adsorbed onto the substrate at the central ("null") dots such that the active dots used to represent "0" and "1" are elevated above the substrate. An electric field with a vertical component can be applied to the molecule using a buried conducting slab. A negative voltage establishes a field that repels the mobile electrons from the null dots. With this bias, the cell is forced to the active state favored by neighbor interactions. A positive voltage, on the other hand, attracts the electrons to the null dots so that the cell takes the null state regardless of the states of neighboring molecules.

activate cells; other regions clock cells to the null state. Calculations will take place in the transition region between active and null domains. It is worth noting that the clocking conductors may be much larger than the molecules themselves. QCA molecules are of the 1-nm scale, and it would be onerous if not intractable to wire each individual molecule, a task that this clocking scheme avoids.

Figure 1.7 demonstrates how active domains from the electric field can be used to drive bit packets through circuitry. Here, we show the length scale of several cells (each cell is a 1-nm-by-1-nm square) to see an active domain propagating along a binary wire. The strong clocking field at the center of the active domain has latched cells in a given state. Cells at the leading edge of the active domain are activated to the state favored by interactions with their latched neighbors to their immediate left (more internal to the active domain). Computation—in this case, a bit copy—occurs at the leading edge of the moving active domain. At the trailing edge of the active domain, cells are released to the null state in preparation for another computation. Since the action of the binary wire is clocked rather than ballistic, we refer to this

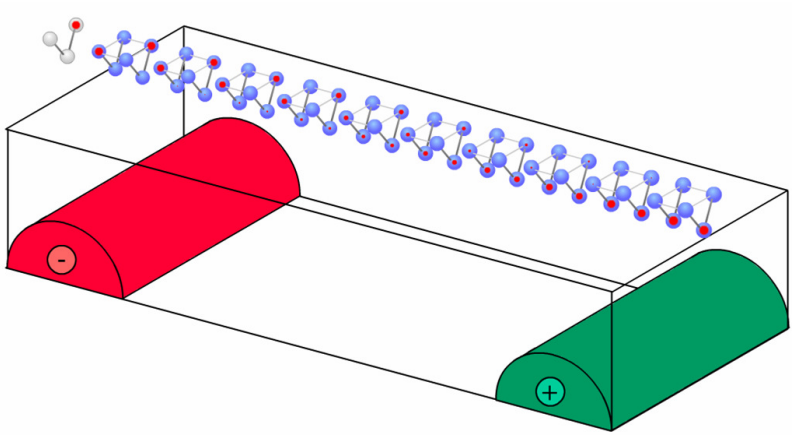

Figure 1.6: Multiple indpendently-charged conductors can establish an electric field with an inhomogeneous vertical component at the QCA device plane. In some domains of this field, cells will be driven to active states, whereas in other regions, cells will be driven to the null state. Calculations and erasures will occur in the transitions between active and null domains.

as a shift register. Computations may be more complex than a mere copy, however: logic gates such as AND, OR, NOT, and XOR have been simulated.

Realizing Molecular QCA Computation

Technical challenges remain in the path to realizing molecular QCA. We briefly outline these challenges here, and discuss the state of solutions to these challenges. Technical challenges include: molecule design, circuit layout, bit write-in for molecular QCA circuits, and bit read-out for molecular QCA circuits.

Molecule Design

The design of QCA molecules is an ongoing endeavor involving physicists, synthetic chemists, and physical chemists. New molecules are being designed and studied as QCA candidates [20, 18]. Models are being developed to predict the performance of molecules as QCA devices. Such predictive models must treat relevant quantum phenomena for particular QCA candidates such as electron localization and the self-

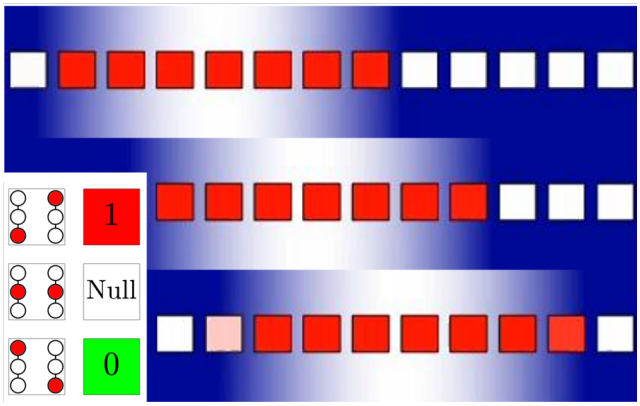

Figure 1.7: An active domain in the clocking field E_z carries a "1" bit packet through a binary wire. Three time-ordered snapshots are shown of an active domain (white region of background gradient representing E_z) sweeping rightward along a binary wire (row of squares). The face color of each cells is coded to indicate its state, as shown in the inset). As the active domain propagates rightward, the cells at the leading (right-most) edge of the active domain transition from "Null" to an active state determined by interaction with neighboring, latched cells deeper within the active domain. Cells at the trailing (left-most) edge of the active domain are released to the Null state. Here, a "1" bit was clocked from other cells to the left of the segment shown, and the output propagates rightward off the image. The action here is not ballistic, but rather is synchronous. Therefore, this is not just a binary wire, but also a shift register.

trapping of charge [21, 22, 23], vibration-coupled electron transfer [10], as well as environmentally-driven quantum decoherence [24, 25], power dissipation, and disentanglement [26]. Candidate molecules may be functionalized with a ligand to facilitate adsorption on a substrate.

Some helpful advances have come from this work. The development of net-neutral, self-doping QCA candidate molecules [18] eliminate the need for additional counterions associated with each cell. Theoretical work suggests that environmental interactions strengthen molecular QCA bits [24]. Models of field-driven, vibration-coupled electron transfer suggest QCA can switch at THz or faster speeds [10].

Controlled switching has not yet been observed in any molecule. Thus, efforts on designing, synthesizing, and testing individual molecuar QCA species will be efforts well-spent. Models of circuit-level behaviors for particular molecular species may also be developed.

Circuit Layout

The layout of molecular circuitry is another technical challenge. Homogenous monolayers of devices, while the most simple to achieve, perform no useful calculation. Therefore, the preferential layout of molecules in well-defined, specific arrangements is necessary.

For circuit layout, self-assembled techniques are promising. Researchers have demonstrated exquisite control over processes to form 2D and 3D structures from DNA [27, 28]. In particular, DNA tiles may be used as molecular circuit boards [29]. Here, DNA tiles will be programmed with specific sites to conjugate with a docking ligand built into the QCA molecule. It may be possible to control the location of DNA tiles on a substrate using lithographic techniques.

One challenge with DNA tiles is localized negative charges due to phosphate groups on the DNA itself. Stray charge has been shown to adversely affect the performance of QCA devices [30, 31]. Therefore, tiles formed from peptide nucleic acid (PNA) [32] may better serve as molecular QCA circuit boards, since PNA has no such stray charge.

Bit Write-in for Molecular QCA Circuits

The nanometer scale of molecular QCA makes bit input more challenging than in conventional devices. It is neither possible nor desirable to form contacts that address individual molecules, because lithographically-feasible devices are an order of magnitude larger than molecular QCA. Optical control of molecular QCA is challenging because optical wavelengths are much longer than the length scale of desirable QCA molecules, and optical interactions are weak at this scale. Additionally, beam sizes are macroscopic, so they can only illuminate large groups of molecular QCA.

Some solutions to the bit input problem have been proposed. One system proposed complementary pairs of fixed-state molecules as bit sources to a shift register

[3] as seen in Figure 1.8. The clock is used to selectively couple one of the sources to a binary wire. In addition to the molecular species required to implement the clocked molecular QCA, this solution likely requires the use of a second molecular species to provide the fixed input sources. Another concept modeled by Pulimeno, *et al*, was an input electric field applied to a single molecule at the end of a row of clocked, three-dot bisferrocene molecules [4]. While this eliminates the need for fixed-state molecules, nanoelectrodes are required to generate a field with single-molecule specificity. This can be seen in Figure 1.9.

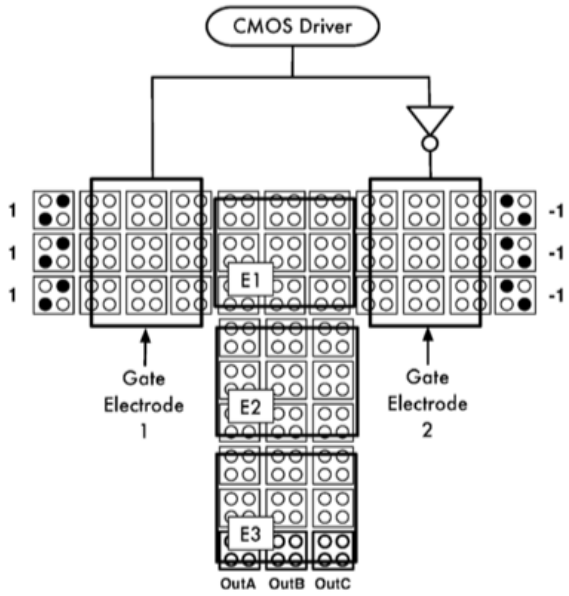

Figure 1.8: This figure shows using fixed-state molecules as inputs to molecular QCA circuits [3]. This method requires additional molecular species.

Bit Read-out from Molecular QCA Circuits

One promising approach to the detection of the state of QCA molecules is the use of single-electron devices, such as single-electron transistors (SETs) [33] and single-electron boxes (SEB) [34]. In particular, SETs have demonstrated sensitivity to single-electron tunneling events over distances less than one nanometer [35]. SETs and SEBs require cryogenic temperatures for operation. The readout of molecular

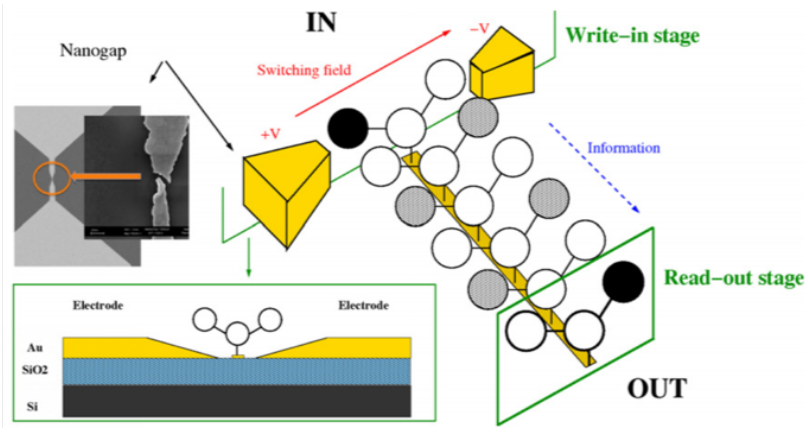

Figure 1.9: This figure shows nanoelectrodes used to generate a field with single molecule specificity [4]. While this eliminates fixed-state molecules, nanoelectrodes are difficult to fabricate.

QCA states using SETs or SEBs has not yet been achieved. This will be enabled by the development of a method for controllably switching QCA molecules which is amenable to SET- or SEB-based read-out.

The focus of this thesis is to present models of a solution for bit write-in to clocked molecular QCA circuits which requires neither fixed-state QCA molecules nor fields with single-molecule specificity. This extends an earlier work that proposed an input system for asynchronous molecular QCA circuits, in which lithographically-formed electrodes establish an input field $\vec{E}_{in}$ to select the state of multiple unclocked two-dot molecules [36]. The selected input bit packet may then be transmitted via a binary wire to QCA circuitry for processing. It was shown that single-molecule specificity is not required of $\vec{E}_{in}$ for the input circuits to work; however, if the interaction of $\vec{E}_{in}$ with the binary wire portion can be suppressed, the circuits will be more robust.

CHAPTER TWO

Model

Molecular QCA is a promising step for beyond-CMOS based computing, but one of the technical challenges is bit write-in. Here, I present a model for bit write-in using an applied electric field for input. In the same way that electric fields are used for clocking QCA circuits, this model will use an electric field to drive a state for QCA cells in an input section. Similar to the work done using input electric fields for two-dot QCA cells [37], this work work uses clocked QCA Cells with either three dots or six dots. This model has the ability to simulate QCA circuits at both a nano-electrode limit as well as a large-electrode limit shown in Figure 2.1. The nano-electrode limit is an idealized version of this input model where the input electric field can be set on a specific set of QCA cells. In subfigure (a) of Figure 2.1 cells 1-3 function as the input section and cells 4-8 function as the binary wire. The large-electrode limit is a more lithographically feasible version where entire QCA circuits are immersed in the input electric field.

A clocked molecular QCA input circuit is depicted in Figure 2.2. Here, we apply a constant field $\vec{E}(\vec{r}) = \vec{E}_{in} + \vec{E}_{clk}$. Input field $\vec{E}_{in}(\vec{r})$ selects bits on cells 1-3, and the field $\vec{E}_{clk}(\vec{r})$ for clockings the entire molecular circuit. In the nano-electrode limit, the input field $\vec{E}_{in}$ is idealized to

$$\vec{E}_{in}(\vec{r}) = \begin{cases} E_y \hat{y}, & \text{in the volume between electrodes,} \\ 0, & \text{elsewhere,} \end{cases} \tag{2.1}$$

with

$$E_y = \frac{v_{in}}{d} \ . \tag{2.2}$$

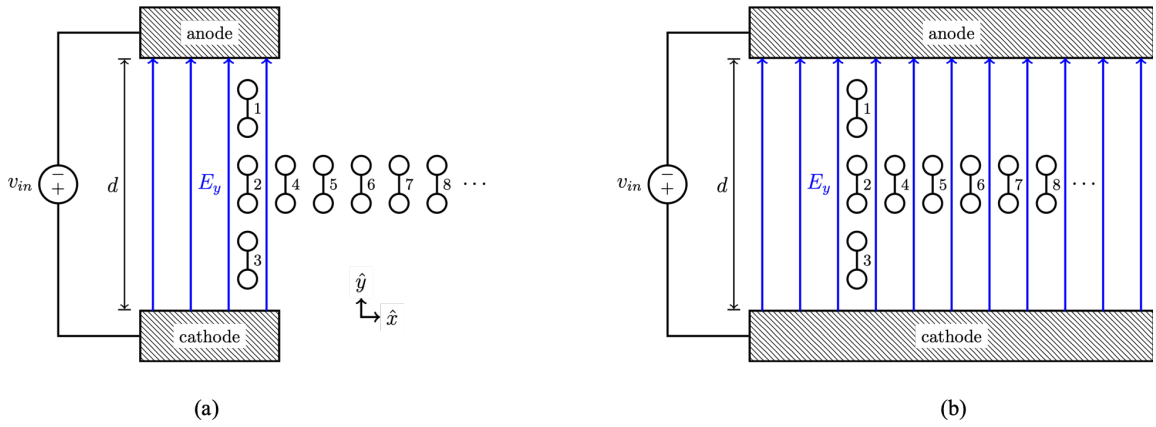

Figure 2.1: The nano-electrode limit [subfigure (a)] is an idealized version of the input model where an electric field can select specific bits. The large-electrode limit [subfigure (b)] has the input electric field completely immerse the entire QCA circuit. While, this is less ideal, it is more lithographically feasbile. The large-electrode limit does not take into account fringing fields or QCA circuits that might leave the bounds of the electrodes. This gives a worst-case scenario for these QCA circuits.

Thus, present results are calculated within a limit where interaction between $\vec{E}_{in}$ and the shift register (cells 4-8) may be suppressed, i.e. the nano-electrode limit. The clock $\vec{E}_{clk}(\vec{r})$ is assumed to be uniform: $\vec{E}_{clk}(\vec{r}) = E_z\hat{z}$ over all space. The rest of the cells shift the selected input bit to computational QCA circuits. With this approach, a single molecular species can provide both input devices and computational QCA circuits.

A single three-dot molecule like $Fc^{+}FcC_2B_9^{-}$ is modeled as a quantum three-state system. Figure 2.3 shows the localized states of a single mobile electron for this molecule. These states form a basis $\mathcal{B} = \{|0\rangle, |N\rangle, |1\rangle\}$ for the molecule's electronic quantum state. Here, $\vec{a}$ is a displacement vector from dot 1 to dot 0 of the cell, and $\vec{h}$ is a displacement from dot zero to a line passing through both dots 0 and 1 perpendicular to $\vec{a}$. All charges are treated as point charges.

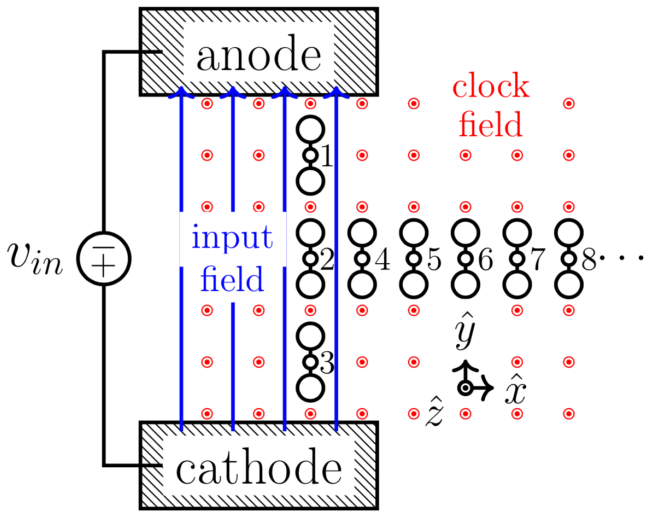

Figure 2.2: An input voltage v_{in} is applied to electrodes to select a QCA input bit. Each three-dot QCA molecules is schematically drawn as system of three quantum dots (black circles). The middle dot is drawn smaller than the other two to indicate that it sits on the substrate ($z = 0$), while the other dots are elevated above the substrate in the $+\hat{z}$ direction. Each cell is assigned the number next to its middle dot. The applied v_{in} establishes an electric field $\vec{E} = E_y\hat{y}$, which immerses the column of molecules aligned in the $\pm\hat{y}$ direction (cells 1-3). Thus, E_y will select the state of molecules 1-3, which are activated when a negative clock (the z-component, $E_z\hat{z}$) is applied. The row of molecules aligned in the $\pm\hat{x}$ direction (cells 4-8) will function as a shift register and can transmit the input bit to other QCA circuitry.

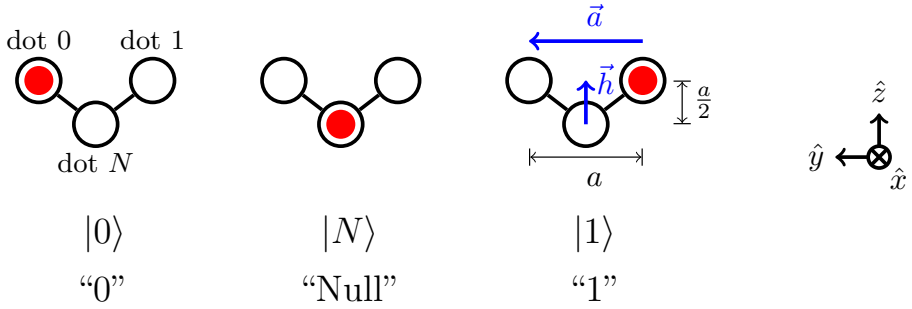

Figure 2.3: The localized states $\{|0\rangle, |N\rangle, |1\rangle$ of a mobile electron provide a basis state for a three-dot QCA molecule. The red disc represents one mobile electron. Neutralizing charge is not shown. Solid lines show tunneling paths between dots, so that tunneling occurs between dots 0 and N or dots 1 and N only (direct tunneling between dots 0 and 1 is suppressed). The orientation vectors indicate how the molecules are arranged in Figure 2.2: the dot 0 is on the substrate, and dots 0 and 1 are elevated above the substrate by $h = a/2$. The vector $\vec{a}$ points in the $+\hat{y}$-direction, and the vector $\hat{h}$ points in the $+\hat{z}$ direction for all cells.

The Hamiltonian for a single cell immersed in electrostatic electric field $\vec{E}$ may be written as

$$\hat{H} = \hat{H}_o + \hat{H}_E + \hat{H}_{\text{int}} \tag{2.3}$$

with

$$\begin{aligned}
\hat{H}_o &= -\gamma \left(\hat{P}_{1,N} + \hat{P}_{N,1} + \hat{P}_{0,N} + \hat{P}_{N,0} \right), \\
\hat{H}_E &= -\frac{q_e \vec{E} \cdot \vec{a}}{2} \left(\hat{P}_{1,1} - \hat{P}_{0,0} \right) - q_e \vec{E} \cdot \vec{h} \hat{P}_{N,N}, \text{ and} \qquad (2.4) \\
\hat{H}_{\text{int}} &= \sum_{\substack{j \in \{0,N,1\} \\ l}} U_j^{(l)} \hat{P}_{j,j}. \qquad (2.5)
\end{aligned}$$

Inter-dot electron transfers are described by $\hat{H}_o$, in which γ is the hopping energy and $\hat{P}_{j,k} \equiv |j\rangle \langle k|$ are transition operators. The projection operators $\hat{P}_{j,j} \equiv |j\rangle \langle j|$ are used to form $\hat{H}_E$, which includes the effects of the applied field. The first term of $\hat{H}_E$ describes the field-driven bias between the active states $|0\rangle$ and $|1\rangle$, and the second term describes the field-driven bias between $|N\rangle$ and the active states. Finally, $\hat{H}_{\text{int}}$ describes the interaction between the cell and any neighbors, each labeled with an integer $l \in \{1, 2, \ldots\}$. The electrostatic potential energy $U_j^{(l)}$ is calculated by setting the target cell to state $|j\rangle$ and then evaluating the electrostatic energy of interaction between the charges from neighbor l with the charges in the target cell. $U_j^{(l)}$ depends on the state of the l-th neighbor cell.

The result of this equation can be seen visually in Figures 2.4 and 2.5. In Figure 2.4 there is a driver QCA cell (green electron) and a respondent QCA cell (red electron). We set the state of the driver cell to simulate a neighboring cell, which can be used as an input to a circuit. We can manually set the polarization and activation of a driver QCA cell and they do not respond to the interaction of other QCA cells or electric fields. Both graphs show the polarization of a respondent cell interacting

with either a driver cell or an input electric field. The y-axis shows the clock field strength which causes the respondent cell to be driven between the $|N\rangle$ state and the active states. This clock field strength changes the second term in Equation 2.4. The first term in Equation 2.4 can be seen visually in Figure 2.5 on the x-axis. As the input field changes this causes the respondent cell to be driven between $|0\rangle$ and $|1\rangle$. Both Figures show a high responsiveness at the center of the graph which shows that QCA cells tend to be in the active states in the presence of any amount of clock field. Because the figures look similar, this show that either neighbor interactions and the clock or an applied input and the clock determine the state of a cell.

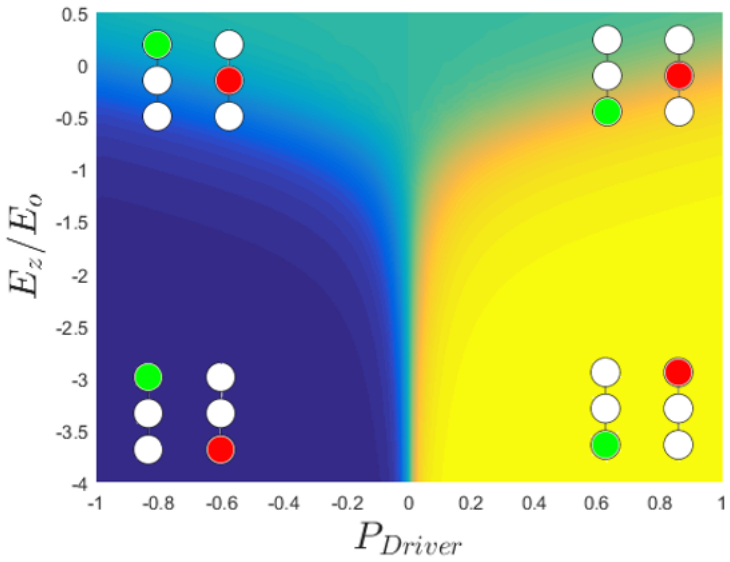

Figure 2.4: A target cell's state is determined by both a blocking electric field and neighbor interactions. This graph shows the polarization of a cell (red electron) in the presence of a driver cell (green electron) and a clocking electric field. As the driver cell's polarization sweeps from -1 to 1 the respondent cell polarizes appropriately. When the clock field is near 0, the respondent cell stays in the inactive, null state.

The ground state of a circuit of M cells maps to the desired calculational result. To calculate the ground state, an intercellular Hartree approximation (ICHA) is made [2]. This involves making an initial guess for the ground state of each cell in the M-cell circuit, and then iterating through the circuit, relaxing each cell to its ground state.

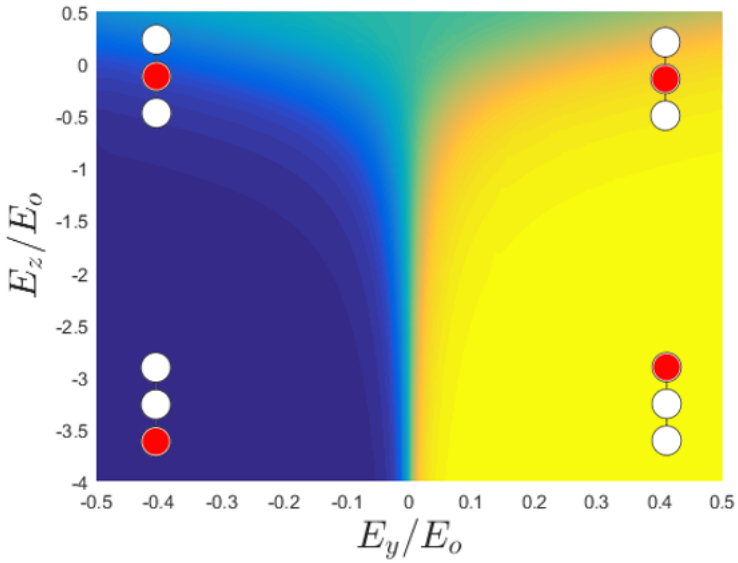

Figure 2.5: The target cell's response is determined by the clock and input components of the electric field, E_z and E_y, respectively. This graph shows the polarization of a cell in the presence of an input electric field and a clocking electric field. As the input electric field sweeps from $-0.5E_o$ to $0.5E_o$ the respondent cell polarizes appropriately. When the clock field is near 0, the respondent cell stays in the inactive, null state. This graph should look similar to Figure 2.4, which verifies the respondent cell operation in the presence input electric fields.

This process is repeated until a self-consistent configuration is achieved for the circuit. ICHA is a useful approximation because calculating in the full Hilbert space increases calculation time exponentially. For a composite quantum system of N 3-state cells the full Hamiltonian must be represented as a matrix that is 3^Nx3^N. There are not many computers that can usefully calculate a system with more than about 8 cells and since these circuits use many more, this approximation is incredibly useful.

This model uses a time-dependent calculation that relaxes the circuit to its ground state. During each time step setting the clock field strength and the input field strength are set, then the ICHA calculation is repeated till the ground state is found.

ICHA does not use the full Hilbert space, therefore some intercellular quantum correlations are discarded [38]. Occasionally this causes our simulation of the circuits in the to fail because the ICHA is an approximation. However, this may be

alleviated using supercells. A supercell groups together cells and has them iterate to self-consistency before continuing on to the rest of the circuit. The main use case for supercells is to alleviate the 'race condition' for majority gates, which can be seen in Figure 2.6. Because the top and bottom inputs wires have to 'turn the corner' to get to the input of the majority gate, the middle bit can arrive at the majority gate before the other two inputs. This give more weight to the middle bit than the outer two bits and can cause improper operation.

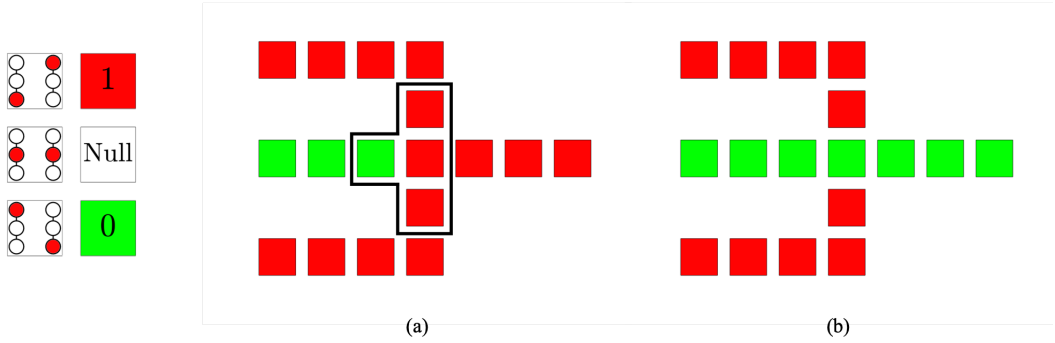

Figure 2.6: This figure shows an example of the 'race condition' and the need for a supercell. In subfigure (b), the middle bit has reached the majority gate faster than the top and bottom bits. Because of this, the middle bit has been given more weight and the majority gate does not have a proper operation. Subfigure (a) shows a black box around the majority gate which allows for these cells to relax at the same time. This causes the ICHA to calculate a properly operating majority gate.

Computational Implementation of the Model

QCA Circuit Layout Tool Front-end

While there have been some tools created to simulate clocked QCA circuits [39], a new tool was needed to simulate clocked molecular QCA circuits using electric fields for bit write-in. The front end of this the QCA Layout Tool was most useful during circuit building. While the front end has more uses, the most useful was laying out many QCA cells in a specific pattern. Linear algebra is the natural language of Quantum Mechanics, so MATLAB was an obvious choice for building this application.

The front end of this application was built with the MATLAB integrated designer GUIDE. Figure 2.7 shows the application during use. The main portion of the application is the layout window where QCA Cells are added to the work space. This is how the QCA Layout Tool was used primarily although there is functionality for running fairly simple simulations with basic signals.

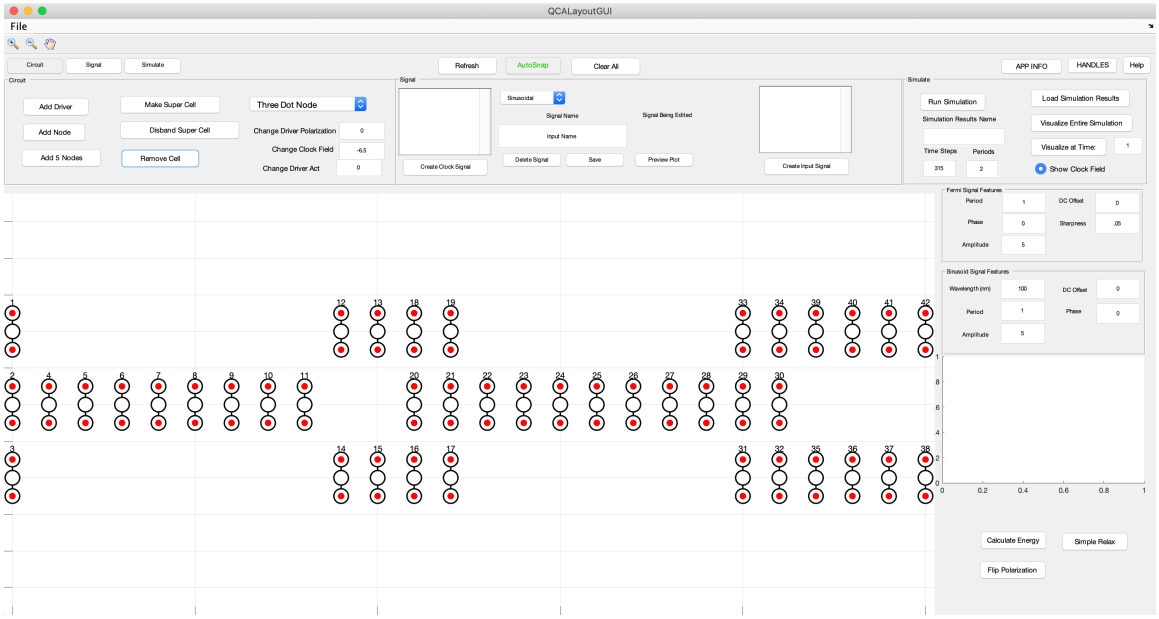

Figure 2.7: This application is used to layout QCA circuits and run simulations. The front end has the ability to create circuits, signals and simulations.

QCA Circuit Layout Tool Class Structure

While many of the basic functionality of this tool can be accessed from the front end, a majority of the more powerful functions are accessed through custom scripts. The class structure is shown in Figure 2.8. The main class is the QCACircuit class which is a custom list or container for the QCACell class and QCASuperCell class. The main functions for the QCACircuit class are the functions for relaxing the circuit to its ground state and the function for sending input and clock signals to the circuit as time progresses. The ThreeDotCell and SixDotCell classes inherit from the QCACell class because while similar each use slightly different functions for math

and for drawing. The QCASuperCell class is a special container class for QCACell class which forces certain cells to work relax to the ground state in the same instance. While this tends to increase calculation time, the results align more with expectations. Finally the signal class is an abstract class that can be used for any periodic signal. Most often it is used for the clocking electric field and the the input electric field, however it can also be used to set the polarization of a driver QCA Cell.

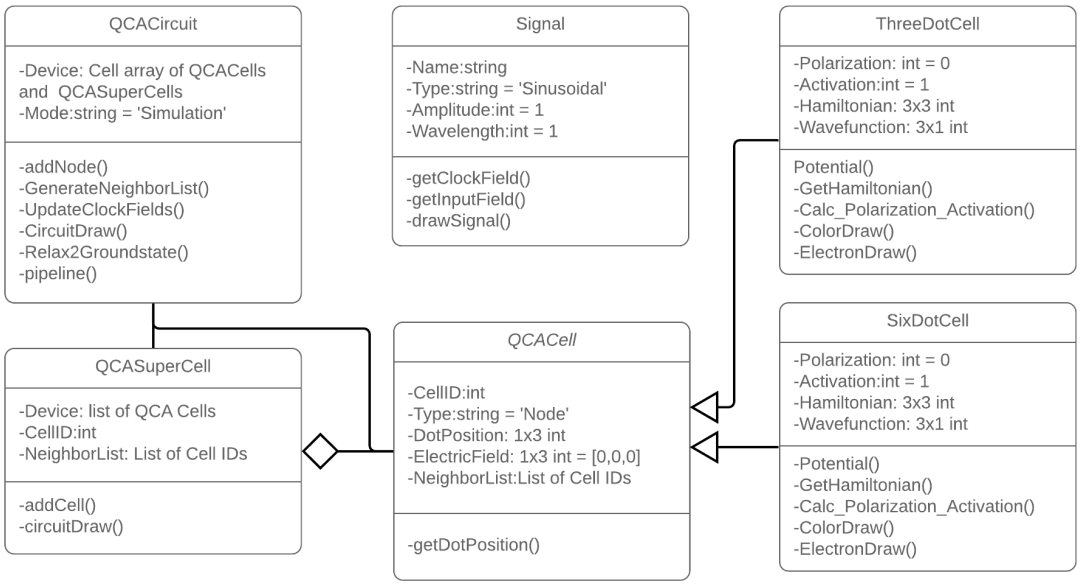

Figure 2.8: The main class in this structure is the QCACircuit class, which is a special list class that holds QCACells, and QCASuperCells. Signal classes are abstract signals that can be used for the clock field, input field, or driver polarization signals.

CHAPTER THREE

Results

The goal of this work is to demonstrate the validity of electric-field inputs for clocked molecular QCA circuits. The first step is to verify the functionality of clocked molecular inputs. This is performed at the nano-electrode limit first, then the next step in validation for this model is to include six-dot molecular logic. Validation is confirmed because these results replicate previous results of clocked molecular QCA circuit simulations. Finally using six-dot logic, an engineering study was performed using this simulator. This study investigates how clock wavelength affects the performance of clocked molecular QCA logic at the nano-electrode limit.

Demonstration of the Clocked Molecular QCA Inputs

A simulation of a clocked QCA input system of Figure 2.2 is shown in Figure 3.1. Specifying a bit on the input cells is a matter of specifying the sign of E_y of an appropriate magnitude and then applying a clock voltage $\vec{E}_{clk} = E_z\hat{z}$. The field is specified in units of E_o, the input field strength required to induce a kink between two molecules:

$$E_o = \left| \frac{E_k}{q_e a} \right| \ . \tag{3.1}$$

The kink energy, E_k, is defined in Equation 1.1. E_o is a useful unit of measurement because it scales based off of the kink energy and the molecule's characteristic length. By scaling the clock and input fields by E_o, these results are consistent across molecules with different physical parameters.

Here, an input field $E_y < E_o$ is sufficient to select an input bit, and an applied clock $E_z = -1.5E_o$ activates the system of cells. In this case, cells 1 and 3 could be eliminated; however, they are necessary in the case where $\vec{E}_{in} = E_y\hat{y}$ is uniform and is applied to all cells (molecules 1-8) (see Figure 3.5). Cells 1 and 3 also may be helpful when fringing field effects are considered. These results were calculated using an ICHA approach with only nearest-neighbor and next-nearest neighbor interactions.

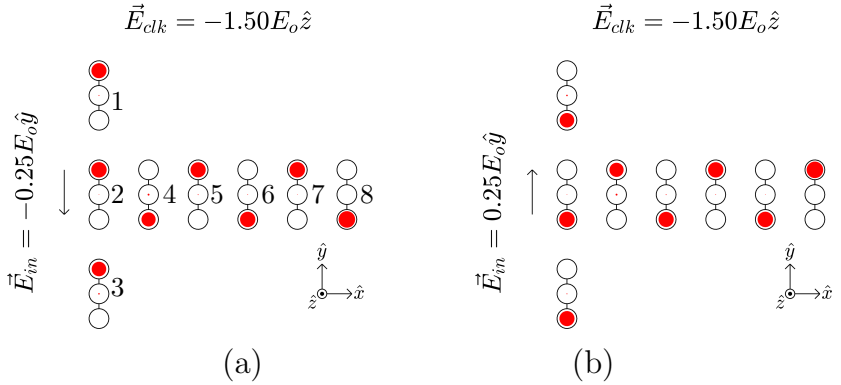

Figure 3.1: An applied electric field $\vec{E} = \vec{E}_{in} + \vec{E}_{clk}$ selects an input state and activates the circuit. The input field $\vec{E}_{in} = E_y\hat{y}$ is applied across cells 1-3 only. Subfigure (a): for an input field $E_y < 0$, cells in this column take the $|0\rangle$ state. Subfigure (b): when $E_y > 0$, cells in this column take the $|1\rangle$ state. In each case, the binary wire formed by cells 4-8 couples to the input cells (cells 1-3), and the bit propagates rightward from the input to other QCA circuits beyond cell 8. The radius of the red dot (electron) in each quantum dot is proportional to the weight or probability that the electron exists in that location.

The result of Figure 3.1 is extended to show the operation of a majority gate is shown in Figure 3.2 with 3 separate bit input supplied to the input chains. While this demonstrates *in silico* the viability of clocked majority logic using three-dot molecules, a more realistic simulation of the electrode/molecular-circuit input system requires further development of the model's software implementation. A larger circuit footprint would be required to accommodate large input electrodes, and the majority logic will likely require isolation from fringing fields due to input electrodes. This will require longer shift registers between inputs and the majority gate. A larger circuit with more molecules will likely require a non-uniform, time-dependent clock $\vec{E}_{clk}(\vec{r}, t)$.

While Figure 3.2 shows three-dot logic with an input system, this is only the beginning. This figure did not use a full time stepped clocked field: only an instantaneous ground state relaxation using an nano-electrode limit for each input section similar to 2.2. This shows promise, however in the next section, six-dot logic is shown using this input model and full time dependent, clocked circuit.

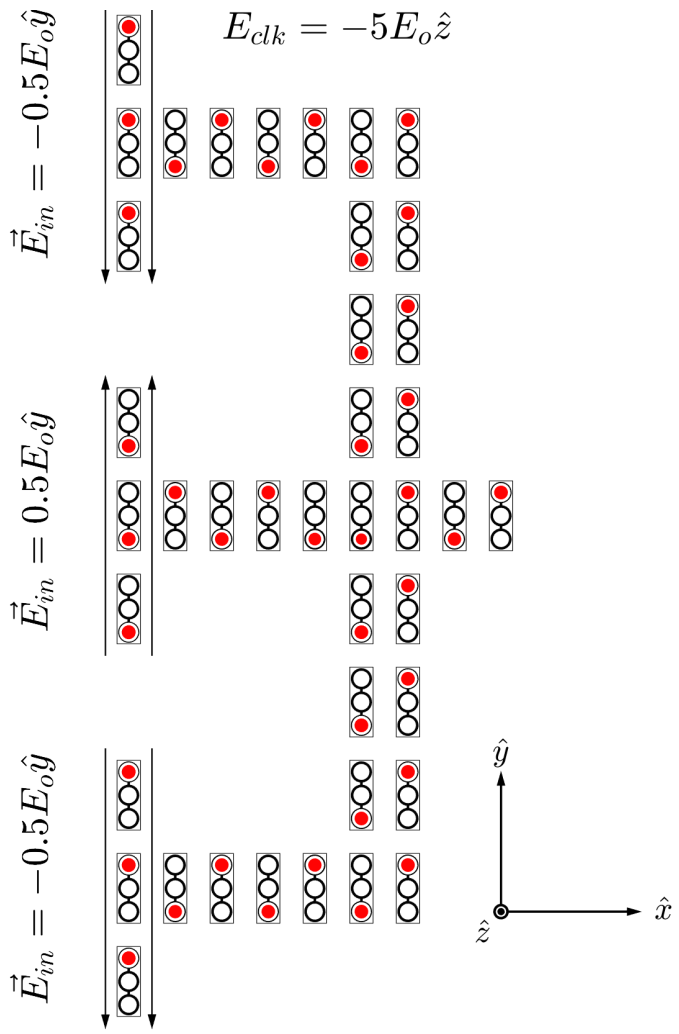

Figure 3.2: A clocked molecular majority gate is a simple extension of the input circuit from Figure 3.1. Here, two like bits dominate one unlike bit in a majority circuit.

Demonstration of Clocked Inputs and Six-dot Molecular QCA Logic

This input model was added to six-dot logic to validate the functionality of the simulator platform. To pass the validation test, the simulator must reproduce previous clocked molecular six-dot QCA logic simulations. Figure 3.3 shows a binary wire, inverter, fan-in, and fan-out circuit using six-dot logic with this input model for bit write-in. In Figure 3.3 each of the circuits operate as intended. This shows that this simulator can correctly simulate six-dot logic while using an input field to write bits to clocked six-dot molecular QCA circuits.

For these simulations 5 nodes were used for the input section instead of the previously used 3 nodes. This was done to give more strength to the input section. The wires leading off of the input section do have some back interaction which can cause disruption in the input section while latching. Once latched, disrupted input nodes can cause the rest of the logic to be disrupted.

Study of Circuit Performance

I studied circuit performance under varied field conditions. The input model and six-dot logic were both validated so, some engineering questions were explored: Which circuits work best with input and clock fields, and how can the clock and input fields be designed to optimize performance?

The first study was to use this simulator to study how the circuits function as the clock wavelength varies. As the clock wavelength varies, the gradient of the clock, which could affect performance as the transition from the active domain to the null domain widens. To study the range of clock field wavelengths that support proper operation for each circuit, simulations were tested for circuits at the nano-electrode limit. Figure 3.4 shows each of these ranges.

The next step for this study is to perform the study using three-dot logic, first at the nano-electrode limit, then at the large electrode limit. It is important to test three-

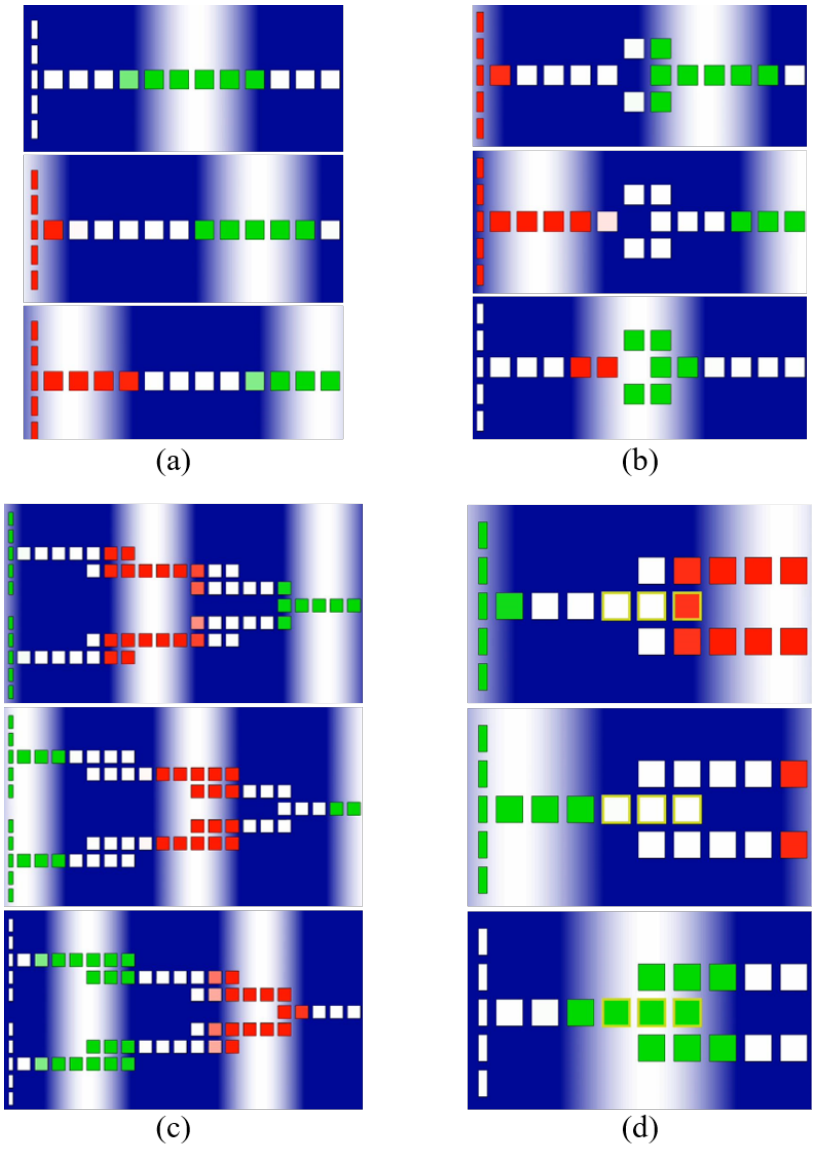

Figure 3.3: In each subfigure, an active domain in the clocking field E_z carries a bit packet through the circuit. The input is selected by an input field E_y at the large electrode limit. Subfigure (a) shows a binary wire. Subfigure (b) shows an inverter. Subfigure (c) shows a fan-in. Subfigure (d) shows a fan-out.

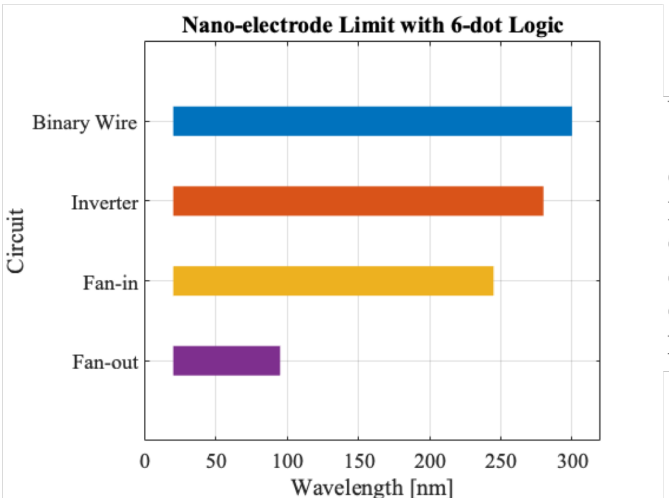

Figure 3.4: This graph shows the working range of clock wavelength for each circuit in the ideal case (nano-electrode limit).

dot QCA cells for a couple of reasons. First, It is anticipated that three-dot logic will be more susceptible to disruption by the input fields because they interact with the field like a dipole, and six-dot cells interact with the field like a quadrupole, having a much weaker moment. Second, six-dot and three-dot QCA cells both use a three state basis, this means that six-dot cells discard many potential kinks. By doing this, many potential kinks have been eliminated. As result we might not accurately predict where three-dot logic fails. Finally molecular design is one of the main technical challenges in realizing molecular QCA, and three-dot logic would eliminate the need for multiple molecular species.

As a preliminary calculation, to show this disruption Figure 3.5 is a calculation of a circuit with three-dot logic in the large-electrode limit, where there is a uniform input field over all cells. It is important that the input field is able to select an input bit, while not introducing kinks into the circuit. While the nano-electrode limit in Figure 3.1 does select a bit, Figure 3.5 shows the boundary for bit selection at the large-electrode limit. This figure shows that weak input fields are sufficient to select an input while not introducing any kinks to the circuit. This is the beginning of the next study and the next steps for this work. This result is similar to the results using

input electric field for two-dot QCA cells [37]. It is encouraging that this preliminary calculation mimics the results found from a two state system.

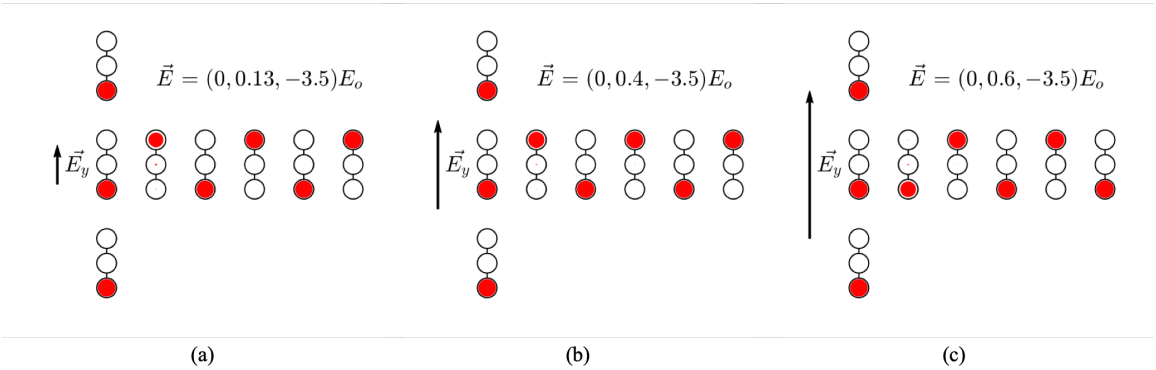

Figure 3.5: Weak fields ($E_y < 0.5E_o$) are sufficient to select an input and still support binary wire states shown in subfigure(a). Strong fields ($E_y \geq 0.5E_o$) introduce kinks shown in subfigure (b). The radius of the red dot (electron) in each quantum dot is proportional to the weight or probability that the electron exists in that location.

CHAPTER FOUR

Conclusion and Future Work

In conclusion a model for bit write-in for Molecular QCA was presented. This model uses both a clocking electric field and an input electric field to propagate bit packets as well as write bits onto QCA circuits. First it was important to show that bit write-in is feasible with electric fields. This was done by using a nano-electrode limit and can be seen in Figure 3.1. I performed this calculation and had it published before this thesis, however this thesis has extended beyond the work of that paper. The next step was to add six-dot molecular QCA logic to show that this model can successful write bits to six-dot molecular QCA circuits. In order to test how robust these circuits are the wavelength of the clocking field was studied. Figure 3.4 shows all the QCA circuits at the nano-electrode limit.

A much more formal and rigorous study of three-dot cells would be important. The reason six-dot cells were studied in this was for calculation verification. The six-dot cell devices are easier on calculation because it still uses a three state basis, effectively discarding many possible kinks that might arise in three-dot cell circuits. While six-dot QCA circuits have been simulated before, none have used this model for bit write-in. It is anticipated that three-dot logic will be more susceptible to disruption by the input fields because they interact with the field like a dipole, and six-dot cells interact with the field like a quadrupole, having a much weaker moment. If research in six-dot cells were to continue for this model, there would be a need for two molecule species. Since molecular design is one of the main technical challenges in realizing molecular QCA, three-dot logic would disregard the need for multiple molecular species.

Molecular QCA is a promising option to move beyond CMOS computing. This thesis showed a model for bit write-in on clocked QCA circuits, which is one of the technical challenges in realizing molecular QCA. These results show promise in this model for bit write-in to molecular QCA circuits

APPENDIX

APPENDIX

MatLab Code

```
function AlignHoriz()
%aligning selected cells horizontally



myCircuit = getappdata(gcf,'myCircuit');


yvals=[];
xvals=[];


i=1;
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on')
                Sel=Sel+1;


            end


        end
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)
                yvals(end+1) = myCircuit.Device{i}.Device{j}.
                   CenterPosition(2);
```

```
                xvals(end+1) = myCircuit.Device{i}.Device{j}.
                    CenterPosition(1);
            end
        end

    else

        if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
            yvals(end+1) = myCircuit.Device{i}.CenterPosition
                (2);
            xvals(end+1) = myCircuit.Device{i}.CenterPosition
                (1);
        end
    end
end

yavg = mean(yvals);
xmin = min(xvals);

x=0;
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
                Selected,'on')
                 Sel=Sel+1;
```

```
            end

        end
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)
                myCircuit.Device{i}.Device{j}.CenterPosition(2)
                    = yavg;
                myCircuit.Device{i}.Device{j}.CenterPosition(1)
                    = xmin+x;

                x=x+1;
            end
        end

    else

        if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
            myCircuit.Device{i}.CenterPosition(2) = yavg;
            myCircuit.Device{i}.CenterPosition(1) = xmin+x;

            x=x+1;
        end
    end
end

myCircuit = myCircuit.CircuitDraw(gca);

setappdata(gcf,'myCircuit',myCircuit);
```

```
end
```

Listing A.1. AlignHoriz Function

```
function AlignVert()
%Aligning selected cells vertically



myCircuit = getappdata(gcf,'myCircuit');

%list of all xvals and yvals
yvals=[];
xvals=[];



%fill the vectors with all the selected cells
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on')
                Sel=Sel+1;


            end


        end
```

```
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)
                yvals(end+1) = myCircuit.Device{i}.Device{j}.
                    CenterPosition(2);
                xvals(end+1) = myCircuit.Device{i}.Device{j}.
                    CenterPosition(1);
            end
        end

    else

        if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
            yvals(end+1) = myCircuit.Device{i}.CenterPosition
                (2);
            xvals(end+1) = myCircuit.Device{i}.CenterPosition
                (1);
        end
    end
end

ymin = min(yvals);  %lowest point of the column that will be
   created
xavg = mean(xvals); %the xvalue where the column is located

y=0;
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
```

```
        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on')
                Sel=Sel+1;

            end

        end
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)
                myCircuit.Device{i}.Device{j}.CenterPosition(1)
                    = xavg;
                myCircuit.Device{i}.Device{j}.CenterPosition(2)
                    = ymin+y;%shift selected cells by 2 upwards

                y=y+2;
            end
        end

    else

        if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
            myCircuit.Device{i}.CenterPosition(1) = xavg;
            myCircuit.Device{i}.CenterPosition(2) = ymin+y;

            y=y+2;
        end
    end
```

```
end

myCircuit = myCircuit.CircuitDraw(gca);

setappdata(gcf,'myCircuit',myCircuit);




end
```

Listing A.2. AlignVert Function

```
function AutoSnap(handles)
% This function allows the opportunity for the user to
   alternate snap to
% grid functionality on and off.  The value attained from
   handles.autoSnap
% switches the circuit's SnapToGrid property on or off
   depending on its
% current state.

    myCircuit = getappdata(gcf,'myCircuit');

snap = get(handles.autoSnap,'Value');%use this for the switch



switch snap
    case  1 %clear, draw, set app data

        myCircuit.SnapToGrid='on';
```

```
        myCircuit = myCircuit.CircuitDraw(0,gca);
        set(handles.autoSnap,'BackgroundColor',[.8 .8 .8],...
            'ForegroundColor',[.1 .8 0]);




    case  0 %switch to sim mode, clear, draw, set app data
        myCircuit.SnapToGrid='off';
        set(handles.autoSnap,'BackgroundColor',[1 1 1],'
           ForegroundColor',[0 0 0]);

end



setappdata(gcf,'myCircuit',myCircuit);
end
```

Listing A.3. AutoSnap Function

```
function CalculateEnergyCallback(handles)
%This function will use the simulation .mat files to create a
   video with
%the help of the PipelineVisualization function

%vizAtCertainTimeButton(handles)

myCircuit = getappdata(gcf,'myCircuit');

%time = 0;
```

```
time = get(handles.vizAtCertainTimeEditBox,'String');
myCircuit = myCircuit.GenerateNeighborList();
%myCircuit = myCircuit.Relax2GroundState(time);


myCircuit = myCircuit.CircuitDraw(time, gca);


all_energy = myCircuit.calculateEnergy(time);
circuit_energy = sum(all_energy)

end
```

Listing A.4. CalculateEnergyCallback Function

```
function ChangeCellPos(handles)
%Change a single cell's position.
myCircuit = getappdata(gcf,'myCircuit');

newPos = str2num(get(handles.chngPos,'String'));



selCells=0; %which cells are selected.  We want it to be 1

%these two will only become useful if selCells == 1
pick=0;
SCpick=0;
```

```
if length(newPos) == 2 %we only want to move one cell by
   position

            for i=1:length(myCircuit.Device)

                if isa(myCircuit.Device{i},'QCASuperCell') %
                   check for device being selected

                    for j=1:length(myCircuit.Device{i}.Device)
                        if strcmp(myCircuit.Device{i}.Device{j
                           }.SelectBox.Selected,'on')

                            selCells=selCells+1;
                            pick = i;
                            SCpick = j;
                        end
                    end
                else
                    if strcmp(myCircuit.Device{i}.SelectBox.
                       Selected,'on')
                        selCells=selCells+1;
                        pick = i;
                    end
                end
            end
```

```
            if selCells == 1  %there is only 1 cell selected,
               now we know which one to change its position
                if isa(myCircuit.Device{pick},'QCASuperCell')
                    myCircuit.Device{pick}.Device{SCpick}.
                       CenterPosition(1:2) = newPos;



                else
                    myCircuit.Device{pick}.CenterPosition(1:2)
                       = newPos;
                end
            end

            myCircuit = myCircuit.CircuitDraw(gca);




end
setappdata(gcf,'myCircuit',myCircuit);
end
```

Listing A.5. ChangeCellPos Function

```
function ChangeClockField(handles)
%This function will change the clock field of the circuit,
   namely the
%electric field in the z direction.  Once the clock field is
   strong enough
```

```
%in the negative z direction, this will allow nodes to get
   relaxed to a
%state of -1 or 1.
%
    myCircuit = getappdata(gcf,'myCircuit');

    newclockfield = str2num(get(handles.chngClock,'String'));

        for i=1:length(myCircuit.Device)
            if isa (myCircuit.Device{i},'QCASuperCell') %if any
                of the cells in a supercell are selected, the
               whole thing will be deleted
                for j=1:length(myCircuit.Device{i}.Device)


                    myCircuit.Device{i}.Device{j}.ElectricField
                        = [0 0 newclockfield];



                end

            else %any cell can be deleted also

                myCircuit.Device{i}.ElectricField = [0  0
                   newclockfield];

            end
```

```
        end
%          myCircuit = myCircuit.CircuitDraw(handles.
   LayoutWindow);
        myCircuit.Device;

        setappdata(gcf,'myCircuit',myCircuit);
        Simulate(handles);

end
```

Listing A.6. ChangeClockField Function

```
function ChangeInputField(handles)
%Aligning all nodes in a column to be the same polarization,
   they will be
%locked in place following the phyics of the electric field.
myCircuit = getappdata(gcf,'myCircuit');



% strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')



inputfield = str2num(get(handles.changeInputField,'String'));
clockSignalsList = getappdata(gcf,'clockSignalsList');

for k=1:length(clockSignalsList)
    for i=1:length(myCircuit.Device)
        if isa (myCircuit.Device{i},'QCASuperCell') %if any of
           the cells in a supercell are selected, the whole
           thing will be deleted
```

```
            for j=1:length(myCircuit.Device{i}.Device)
                if strcmp(myCircuit.Device{i}.Device{j}.
                   SelectBox.Selected,'on')

                    myCircuit.Device{i}.Device{j}.ElectricField
                       (2) = inputfield;

                end


            end
        else %any cell can be deleted also
            if strcmp(myCircuit.Device{i}.SelectBox.Selected,'
               on')
                myCircuit.Device{i}.ElectricField(2) =
                   inputfield;
                myCircuit.Device{i}.ElectricField(2);
            end
        end

    end
end
setappdata(gcf,'myCircuit',myCircuit);


Simulate(handles);
```

```
end
```

Listing A.7. ChangeInputField Function

```
function ChangePol(handles, varargin)
%Change the position of any single driver or node.
myCircuit = getappdata(gcf,'myCircuit');



pol = str2num(get(handles.chngPol,'String'));
act = str2num(get(handles.chngAct,'String'));



if isempty(pol)
    pol = -1;
else
    if pol<-1
        pol=-1;
    elseif pol>1
        pol=1;
    end
    set(handles.chngPol, 'String', num2str(pol));
end



if nargin > 1
    polarization = varargin{1};

    switch polarization
        case 1
```

```
            pol = 1;
        case -1
            pol = -1;
    end
end

    for i=1:length(myCircuit.Device)

        if isa(myCircuit.Device{i},'QCASuperCell')

            for j=1:length(myCircuit.Device{i}.Device)
                if strcmp(myCircuit.Device{i}.Device{j}.Type,'
                   Driver') && (strcmp(myCircuit.Device{i}.
                   Device{j}.SelectBox.Selected,'on'))%check
                   for device being on

                    myCircuit.Device{i}.Device{j}.Polarization=
                       pol;
                    myCircuit.Device{i}.Device{j}.Activation=
                       act;

                    %if it's on, change polarization to
                       whatever the user inputs

                end
            end
        else
```

```
            if strcmp(myCircuit.Device{i}.Type,'Driver') && (
               strcmp(myCircuit.Device{i}.SelectBox.Selected,'
               on'))%check for device being on
                myCircuit.Device{i}.Polarization=pol;
                myCircuit.Device{i}.Activation=act;

            end
        end
    end



    myCircuit = myCircuit.CircuitDraw(0, gca);


    setappdata(gcf,'myCircuit',myCircuit);

end
```

Listing A.8. ChangePol Function

```
function ClearAll(handles)
%UNTITLED11 Summary of this function goes here
%   Detailed explanation goes here

f=gcf;

a=gca;
myCircuit = getappdata(f,'myCircuit');
clockSignalsList = getappdata(f,'clockSignalsList');
```

```
cla;%clear the axes

plot(handles.plotAxes,0,0);

copies = getappdata(f,'Copies');

copies={};

clockSignalsList = {};
handles.signalList.String = '';
handles.signalList.Value = 1;
handles.signalEditor.String = '' ;
handles.signalEditType.String = '';
handles.signalType.Value = 1;
        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'off';
        handles.electrodePanel.Visible = 'off';

setappdata(f,'clockSignalsList',clockSignalsList);


myCircuit.Device{1}={};%empty first node

myCircuit.Device=myCircuit.Device{1};%the empty first device

myCircuit.Mode = 'Simulation';
myCircuit = myCircuit.CircuitDraw(gca);
```

```
setappdata(f,'myCircuit',myCircuit);
setappdata(f,'Copies',copies);




end
```

Listing A.9. ClearAll Function

```
function ClickFunctionality(handles,eventdata,Select)
%All the different types of clicks are documented here, along
   with the
%specific functionalities attributed to each one.


switch Select

    case 'normal' %left click
        %we don't want anything else for left click

    case 'extend' %shift click or scroll click
        RectangleSelect()

    case 'alternate' %right click
        RightClickThings() %but this is redundant

        %it is also redundant
```

```
    case 'open' %double click
        %not sure what would work for this yet...

end
```

Listing A.10. ClickFunctionality Function

```
function CopyCells()
%Make a copy of all selected cells.
myCircuit = getappdata(gcf,'myCircuit');

%we want to clear the current data
copyParts = getappdata(gcf,'Copies');

copyParts={}; %empty the current copies




%which ones are to be copied.  Put them into the copyParts cell
    array
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)

            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on')
                Sel = Sel+1;
            end
```

```
        end
        if Sel
            copyParts{end+1} = myCircuit.Device{i};
        end

    else
            if strcmp(myCircuit.Device{i}.SelectBox.Selected,'
               on')
                copyParts{end+1} = myCircuit.Device{i};
            end
    end
end



%put the copied cells into the current fig for pasting later
setappdata(gcf,'Copies',copyParts);
setappdata(gcf,'myCircuit',myCircuit);









end
```

Listing A.11. CopyCells Function

```
function CreateSignal(handles, signalCategory)
%This function creates the signal. It can be sinusoidal,
   electrode, fermi,
```

```
%or custom, and will be saved as such.  The user can decide
   which of the 4
%they want to have their signal to be, and assign the
   corresponding values
%before creating


if ~isempty(handles.signalName.String) %the signal must get a
   name in order to be created
    clockSignalsList = getappdata(gcf,'clockSignalsList');
    inputSignalsList = getappdata(gcf,'inputSignalsList');


    clockNames = cellstr(get(handles.signalList,'String'));
    inputNames = cellstr(get(handles.inputSignalList,'String'))
       ;


    contents = cellstr(get(handles.signalType,'String'));
    sigType = contents{get(handles.signalType,'Value')};
    transitions = cellstr(get(handles.transitionType,'String'))
       ;
    transType = transitions{get(handles.transitionType,'Value')
       };



    % sigType = handlesButton.signalType.String;
```

```
mySignal = Signal();


switch sigType %assign the values depending on what signal
   type is selected
    case 'Sinusoidal'
        mySignal.Amplitude = str2num(handles.changeAmp.
           String);
        mySignal.Wavelength = str2num(handles.changeWave.
           String);
        mySignal.Period = str2num(handles.changePeriod.
           String);
        mySignal.Phase = str2num(handles.changePhase.String
           );
        mySignal.Type = sigType;



    case 'Fermi'
        mySignal.Amplitude = str2num(handles.changeAmpFermi
           .String);
        mySignal.Sharpness = str2num(handles.
           changeSharpnessFermi.String);
        mySignal.Period = str2num(handles.changePeriodFermi
           .String);
        mySignal.Phase = str2num(handles.changePhaseFermi.
           String);
        mySignal.MeanValue = str2num(handles.
           changeSharpnessFermi.String);
        mySignal.Type = sigType;
```

```
    case 'Custom'



        mySignal.Type = sigType;



    case 'Electrode'
        mySignal.Type = sigType;
        mySignal.InputField = str2num(handles.
           changeInputField.String);
end



copy=0;
switch signalCategory
    case 'clockSignal'
        for i=1:length(clockNames)
            if strcmp(clockNames{i},handles.signalName.
               String)
                copy=copy+1;
            end
        end

        if copy~=0
            newName = strcat(handles.signalName.String,'(
               copy)');
```

```
            mySignal.Name = newName;
            handles.signalList.String{end+1,1} = newName;
        else
            mySignal.Name = handles.signalName.String;
            handles.signalList.String{end+1,1} = handles.
               signalName.String;
        end

        clockSignalsList{end+1} = mySignal;
        setappdata(gcf,'clockSignalsList',clockSignalsList)
           ;

    case 'inputSignal'
        for i=1:length(inputNames)
            if strcmp(inputNames{i},handles.signalName.
               String)
                copy=copy+1;
            end
        end

        if copy~=0
            newName = strcat(handles.signalName.String,'(
               copy)');
            mySignal.Name = newName;
            handles.inputSignalList.String{end+1,1} =
               newName;
        else
            mySignal.Name = handles.signalName.String;
```

```
                handles.inputSignalList.String{end+1,1} =
                    handles.signalName.String;
            end

            inputSignalsList{end+1} = mySignal;
            setappdata(gcf,'inputSignalsList',inputSignalsList)
                ;

        otherwise
            disp('Not currently implemented')
    end









    handles.signalName.String = 'Input Name';
    handles.signalName.ForegroundColor = 'black';
    handles.signalName.Value = 1;

else

    handles.signalName.String = 'NAME NEEDED';
    fprintf('INPUT NAME NEEDED\n')
    %         handles.signalName.ForegroundColor = 'red';
```

```
    handles.signalName.String = 'Input Name';

    handles.signalName.ForegroundColor = 'red';
end

end
```

Listing A.12. CreateSignal Function

```
function CreateSimulationFile(handles)
%The simulation results .mat file will be created here by
   running numerous
%simulations and storing them into a .mat file witha  default
   name or one
%assigned by the user in an edit box


myCircuit = getappdata(gcf,'myCircuit');
clockSignalsList = getappdata(gcf,'clockSignalsList');
inputSignalsList = getappdata(gcf,'inputSignalsList');
myCircuit = myCircuit.GenerateNeighborList();

name = num2str(handles.nameSim.String);
nt = num2str(handles.numberOfTimeSteps.String);
numOfPeriods = num2str(handles.numberOfPeriods.String);


% f=gcf;
```

```
%
% f.Pointer = 'watch';



if length(clockSignalsList)==1 %pipeline will run

%     if isempty(name)
%         %myCircuit = myCircuit.pipeline(SignalsList);
%         myCircuit.pipeline(clockSignalsList, 'Filename', name
   ,'inputSignalsList', inputSignalsList, 'TimeSteps', nt);
%
%     else
%         myCircuit = myCircuit.pipeline(clockSignalsList,name)
   ;
%     end


    myCircuit.pipeline(clockSignalsList, 'Filename', name,'
       inputSignalsList', inputSignalsList, 'TimeSteps', nt, '
       numOfPeriods', numOfPeriods);

elseif length(clockSignalsList) > 1

    %no functionality for multiple Signals yet kinda broken rn

    if isempty(name)
        myCircuit = myCircuit.pipeline(clockSignalsList);
```

```
    else
        myCircuit = myCircuit.pipeline(clockSignalsList,name);

    end



%     d = dialog('Position',[300 300 250 150],'Name','Clear All
   ');
%     txt = uicontrol('Parent',d,...
%         'Style','text',...
%         'Position',[20 80 210 40],...
%         'String','Clear all? (nothing will be saved)');
%
%
%     btn1 = uicontrol('Parent',d,...
%         'Position',[49 40 70 25],...
%         'String','Yes',...
%         'Callback',@Yes);
%
%     btn2 = uicontrol('Parent',d,...
%         'Position',[139 40 70 25],...
%         'String','Close',...
%         'Callback',@No);



end
```

```
myCircuit = myCircuit.CircuitDraw(gca);


setappdata(gcf,'myCircuit',myCircuit);
setappdata(gcf,'clockSignalsList',clockSignalsList);



% f.Pointer = 'arrow';




    function Yes(source,callbackdata)

        delete(d);
        ClearAll(handles);

    end

    function No(source,callbackdata)

        delete(d);
    end




end
```

Listing A.13. CreateSimulationFile Function

```
function DeleteSignal(handles)
```

```
%This function will delete the current signal selected in the
   signal list
%on the GUI.

contents = cellstr(get(handles.signalList,'String'));

clockSignalsList = getappdata(gcf,'clockSignalsList');

if ~isempty(contents) %locating the selected signal to delete

    handles.signalEditor.String = '';
    handles.signalEditType.String = '';

    sigName = contents{get(handles.signalList,'Value')};

    newSigs = {};
    contents = {};

    delete = handles.signalList.Value; %this tells us which
       signal is selected

    clockSignalsList{delete} = {}; %erase that signal within
       the signal list
```

```
    for i=1:length(clockSignalsList) %replacing the old signals
        list with the nonempty signals for both the gui and the
        appdata list
        if ~isempty(clockSignalsList{i})

            newSigs{end+1} = clockSignalsList{i};
            contents{end+1,1} = clockSignalsList{i}.Name;
        end
    end

    handles.signalList.String = contents;

    clockSignalsList = newSigs;
    handles.signalList.Value = 1;

    plot(handles.plotAxes,0,0); %clear the plotting axis


    handles.signalName.String = 'Input Name';

    setappdata(gcf,'clockSignalsList',clockSignalsList);

end
end
```

Listing A.14. DeleteSignal Function

```
function DisbandSuperCell()
%If a member of a a supercell is selected and then this
   function is called,
```

```
%then the supercell will be disbanded and all constituent parts
    will be
%individual nodes again.
myCircuit = getappdata(gcf,'myCircuit');


newCircuit = QCACircuit;

for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')%if it's a
       supercell

        findselect=[];


        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on') %supercell that is selected
                findselect(end+1)=1;%are any of the cells in
                   the SC selected?
            else
                findselect(end+1)=0;
            end
        end


        if sum(findselect)>0 %if any of them are selected, send
            each cell out of the supercell
```

```
            for j=1:length(myCircuit.Device{i}.Device)
                newCircuit= newCircuit.addNode(myCircuit.Device
                   {i}.Device{j});
            end

        else
            newCircuit = newCircuit.addNode(myCircuit.Device{i
               });
        end


    else
        newCircuit = newCircuit.addNode(myCircuit.Device{i});%
           if it's not a supercell
    end
end

snapping = myCircuit.SnapToGrid; %keep the snap to grid trait


myCircuit=newCircuit; %old = new

myCircuit.SnapToGrid = snapping;



myCircuit=myCircuit.CircuitDraw(0,gca);
setappdata(gcf,'myCircuit',myCircuit);
```

```
end
```

Listing A.15. DisbandSuperCell Function

```
function DragDrop()
%Drag and drop functionality for single cells, supercells, and
   groups of
%cells.



dragging = [];
orPos = [];
myOrPos=[];

% FOR THE GUI::: This function will allow drag and drop
   capability during

f=gcf;
a=gca;



myCircuit = getappdata(f,'myCircuit');







%assigning callbacks for each of the script functions the
   button functions
```

```
f.WindowButtonUpFcn=@dropObject; %dropping the object being
   dragged

f.WindowButtonMotionFcn=@moveObject; %moving that object




%when dragging multiple cells, we want to keep a list of those
   cells along
%with their positions
selectedcells=[];
selectedPos=[];



for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)

            if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on')
                Sel=Sel+1;
            end
        end
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)
```

```
                selectedcells(end+1)=myCircuit.Device{i}.Device
                    {j}.CellID;

                myCircuit.Device{i}.Device{j}.SelectBox.
                    ButtonDownFcn = @dragObject;

                selectedPos(:,end+1)=myCircuit.Device{i}.Device
                    {j}.CenterPosition;
            end

        end

    else
        if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')

            selectedcells(end+1)=myCircuit.Device{i}.CellID;  %
                we want to store the cell ID's and the center
                positions
            selectedPos(:,end+1)=myCircuit.Device{i}.
                CenterPosition;

            myCircuit.Device{i}.SelectBox.ButtonDownFcn =
                @dragObject; %every cell is draggable even in a
                group
        end
    end
end
```

```
selectedcells;
selectedPos;
patchList = {};
for a=1:length(selectedcells)
    patchList{a} = {}; %we want a list of patches to drag for
       each of the selected cells
end


cellList = myCircuit.getCellArray(selectedcells); %these are
   the actual cells being dragged


loopit = length(cellList); %loop iteration


%making all the ghost patches to move as we drag the cells
for i=1:loopit
    if (selectedcells(i)-floor(selectedcells(i)))==0 %normal
       node

        patchList{i} = patch;
        patchList{i}.XData=[cellList{i}.CenterPosition(1)-.25;
           cellList{i}.CenterPosition(1)+.25;cellList{i}.
           CenterPosition(1)+.25;cellList{i}.CenterPosition(1)
           -.25];
        patchList{i}.YData=[cellList{i}.CenterPosition(2)-.75;
           cellList{i}.CenterPosition(2)-.75;cellList{i}.
```

```
            CenterPosition(2)+.75;cellList{i}.CenterPosition(2)
            +.75];
        patchList{i}.FaceColor=[1 1 1];
        patchList{i}.FaceAlpha = .02; %they are white patches
            that will be dragged.  Same size as any cell

        patchList{i}.ButtonDownFcn = @dragObject;

    else %supercell

        patchList{i} = patch;
        patchList{i}.XData=[cellList{i}.CenterPosition(1)-.25;
            cellList{i}.CenterPosition(1)+.25;cellList{i}.
            CenterPosition(1)+.25;cellList{i}.CenterPosition(1)
            -.25];
        patchList{i}.YData=[cellList{i}.CenterPosition(2)-.75;
            cellList{i}.CenterPosition(2)-.75;cellList{i}.
            CenterPosition(2)+.75;cellList{i}.CenterPosition(2)
            +.75];
        patchList{i}.EdgeColor=cellList{i}.SelectBox.EdgeColor;
        patchList{i}.FaceAlpha = .02;
        patchList{i}.LineWidth = 3;
        patchList{i}.ButtonDownFcn = @dragObject;
    end

end
```

```
% The job of the first function is to initiate the dragging
   process
% by getting the current point of the mouse upon clicking
    function dragObject(hObject,eventdata)
        dragging = hObject;
        orPos = get(gca,'CurrentPoint') ; %will help use
           dictate how much the position changes
        myOrPos = orPos; %will be used later to determine which
            cell was clicked to be dragged



        %finding which of the selected cells was the indivdual
           cell
        %dragged
        myOrPos=myOrPos(1,:);
        myOrPos(3)=0;
        selectedPos(3,:)=0;
        orPosDiffs = selectedPos - myOrPos';

        orPosDiffs = orPosDiffs.^2;
        orPosDiffs = sum(orPosDiffs,1).^(.5);

        [val indx] = min(orPosDiffs);

    end
```

```
% This function will "drop" the object by using the difference
   of the
% final and intial point will be added to the x and y data of
   the
% patch's original position.  Note that in get() we use gca
   instead
% of gcf since the units of the patch's x data and y data are
   in coordinates
% instead of pixels (much better)
    function dropObject(hObject,eventdata)
        if ~isempty(dragging)

            newPos = get(gca,'CurrentPoint');
            posDiff = newPos - orPos;%change in position


            mainPosDiff = newPos - myOrPos;
            mainPosDiff = mainPosDiff(1,:);
            mainPosDiff(3)=0;

            set(dragging,'XData',get(dragging,'XData') +
               posDiff(1,1));%the patch will be at this
               position now
            set(dragging,'YData',get(dragging,'YData') +
               posDiff(1,2));
            dragging = [];
```

```
for i=1:length(myCircuit.Device) %moving each cell
   or supercell by the position difference
    sel=0;
    if isa(myCircuit.Device{i},'QCASuperCell')

        for j=1:length(myCircuit.Device{i}.Device)
            if strcmp(myCircuit.Device{i}.Device{j
               }.SelectBox.Selected,'on')
                sel=sel+1;
            end
        end

        if sel

            for j=1:length(myCircuit.Device{i}.
               Device)
                myCircuit.Device{i}.Device{j}.
                   CenterPosition = myCircuit.
                   Device{i}.Device{j}.
                   CenterPosition + mainPosDiff;
            end

        end

    else %not a super cell
```

```
        if strcmp(myCircuit.Device{i}.SelectBox.
           Selected,'on')
            myCircuit.Device{i}.CenterPosition =
               myCircuit.Device{i}.CenterPosition +
                mainPosDiff;
            myCircuit.Device{i}.CenterPosition;

        end

    end
end

patchList={}; %empty the patch list since we don't
   need any of them anymore



myCircuit = myCircuit.CircuitDraw(0,gca); %redraw,
   Select() will get called automatically upon
   being redrawn



setappdata(gcf,'myCircuit',myCircuit);
```

```
        end  %end drag super cell

    end




% This is using the change in position to make the drag and
   drop look
% seamless, instead of jumping from place to place
    function moveObject(hObject,eventdata)
        if ~isempty(dragging)

            newPos = get(gca,'CurrentPoint'); %getting the
               coordinates
            posDiff = newPos - orPos;
            orPos = newPos;


            for i=1:length(patchList)  %moving each of the
               ghost patches so they are dragged as a group
                        patchList{i}.XData = patchList{i}.XData
                            + posDiff(1,1);
                        patchList{i}.YData = patchList{i}.YData
                            + posDiff(1,2);
            end
```

```
        end

    end
end
```

Listing A.16. DragDrop Function

```
function DrawElectrodes()
%FUNCTION SHOULD BE OBSOLETE

            SignalsList = getappdata(gcf,'SignalsList');

            for i=1:length(SignalsList)

                if strcmp(SignalsList{i}.IsDrawn,'on')


                    if ~isempty(SignalsList{i}.Height)

                        SignalsList{i} = SignalsList{i}.
                            drawElectrode();
                    else

                    end

                end
            end
            setappdata(gcf,'SignalsList',SignalsList);
```

```
end
```

Listing A.17. DrawElectrodes Function

```
function d = DrawThreeDot(CenterPosition)
CenterPosition=[1 2];
c1 = circle(CenterPosition(1), CenterPosition(2), 0.5, [1 1 1])
   ;
c2 = circle(CenterPosition(1), CenterPosition(2)+2, 0.5, [1 1
   1]);
c3 = circle(CenterPosition(1), CenterPosition(2)+4, 0.5, [1 1
   1]);

%draw electron



e1 = circle(CenterPosition(1), CenterPosition(2)+4, 0.7*0.5, [1
    0 0]);



hold on;

p1 = plot([CenterPosition(1),CenterPosition(1)], [
   CenterPosition(2)+0.5, CenterPosition(2)+1.5] ,'b');
p2 = plot([CenterPosition(1),CenterPosition(1)],[CenterPosition
   (2)+2.5, CenterPosition(2)+3.5],'b');
```

```
r1 = rectangle('Position',[CenterPosition(1)-1, CenterPosition
   (2)-1, 2, 6]);

hold off;

%axis([0 4 0 10])

set(gca , 'FontName', 'Times', 'FontSize', 20); % format the
   current axes
grid on;

xlim ([0 4]); % adjust the x- limits of the axis
ylim ([0 10]); % adjust the y- limits of the axis

% xlabel ('$x$(m)', 'Interpreter', 'latex') % add an x- label
% ylabel ('$y$(m)', 'Interpreter', 'latex') % add a y- label

title('Three-Dot Cell')

axis equal
end
```

Listing A.18. DrawThreeDot Function

```
function ElectrodeDrawer(handles)
%This function will draw the selected signal on the circuit
   axis as long as
%the selected signal is an electrode.

SignalsList = getappdata(gcf,'SignalsList');
```

```
contents = cellstr(get(handles.signalList,'String'));

if ~isempty(contents)
    sigName = contents{get(handles.signalList,'Value')};

    for i=1:length(SignalsList)
        SignalsList{i}.Name;
        SignalsList{i}.Type;
        if strcmp(SignalsList{i}.Name,sigName) && strcmp(
           SignalsList{i}.Type,'Electrode')

            SignalsList{i}.IsDrawn = 'on';
            mySignal = SignalsList{i};

            if ~isempty(mySignal.Height)
                mySignal = mySignal.drawElectrode(); %the
                    signal has all of the required traits for
                    drawing an electrode

            else
                [center height width field] = GetBoxTraits(
                    handles); %we need to attain the center
                    position, height, width, and Efield
                mySignal = mySignal.drawElectrode(center ,
                    height ,width ,field);
```

```
                SignalsList{i} = mySignal;
            end

        end

    end

end


setappdata(gcf,'SignalsList',SignalsList); %must set app data
   for SignalsList


myCircuit = getappdata(gcf,'myCircuit');
myCircuit = myCircuit.CircuitDraw(gca);



setappdata(gcf,'myCircuit',myCircuit);


% ChangeInputField(handlesButton);


end
```

Listing A.19. ElectrodeDrawer Function

```
function ElectrodeEraser(handles)
```

```
%This function will erase the selected electrode(s). The signal
    will still be stored still,
%not deleted even though it is not drawn

SignalsList = getappdata(gcf,'SignalsList');
myCircuit = getappdata(gcf,'myCircuit');



for i=1:length(SignalsList)

    if strcmp(SignalsList{i}.TopPatch.Selected,'on') || strcmp(
       SignalsList{i}.BottomPatch.Selected,'on')
        SignalsList{i}.IsDrawn = 'off';
    end

end

setappdata(gcf,'SignalsList',SignalsList);

myCircuit = myCircuit.CircuitDraw(gca);




setappdata(gcf,'myCircuit',myCircuit);

end
```

Listing A.20. ElectrodeEraser Function

```
function f = FermiTransition( x, XF, Xtemp )
```

```
%FermiTransition Returns a fermi transition given a tranision
   point (XF)
%and a sharpness factor (XTemp)
%   Detailed explanation goes here

    f = 1 ./ (1 + exp( (x-XF)/Xtemp ));



end
```

Listing A.21. FermiTransition Function

```
function FlipPol(handles, varargin)
%Change the position of any single driver or node.
myCircuit = getappdata(gcf,'myCircuit');




idx = 1;


for i=1:length(myCircuit.Device)

    if isa(myCircuit.Device{i},'QCASuperCell')

        for j=1:length(myCircuit.Device{i}.Device)
            if (strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
               Selected,'on'))%check for device being on
```

```
                myCircuit.Device{i}.Device{j}.Polarization = -1
                   *myCircuit.Device{i}.Device{j}.Polarization;
            end
        end
    else
        if (strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
           )%check for device being on
            myCircuit.Device{i}.Polarization = -1*myCircuit.
               Device{i}.Polarization;
        end
    end
end









myCircuit = myCircuit.CircuitDraw(0, gca);



setappdata(gcf,'myCircuit',myCircuit);




end
```

Listing A.22. FlipPol Function

```
function GenerateSimulationVideo(handles)
```

```
%This function will use the simulation .mat files to create a
   video with
%the help of the PipelineVisualization function



Sim = getappdata(gcf,'SimResults');
path = getappdata(gcf, 'SimResultsPath');



%[Sim path]= uigetfile('*.mat'); %path gets sent into Pipeline
   in order to change the path, that way we can put the video
   file anywhere

% f=gcf;



% f.Pointer = 'watch';



if Sim
    PipelineVisualization(Sim,gca, path);

else
    [Sim, path]= uigetfile('*.mat'); %path gets sent into
       Pipeline in order to change the path, that way we can
       put the video file anywhere
    PipelineVisualization(Sim,gca,path);
end
```

```
myCircuit = getappdata(gcf,'myCircuit');
myCircuit = myCircuit.CircuitDraw(gca);

setappdata(gcf,'myCircuit',myCircuit);



% f.Pointer = 'arrow';
% f.Pointer;
end
```

Listing A.23. GenerateSimulationVideo Function

```
function [center height width field] = GetBoxTraits(handles)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
myCircuit = getappdata(gcf,'myCircuit');

center = [0 0 0];
height = 1.5;
width = .5;

field  = str2num(handles.changeInputField.String);

x=[];
y=[];

for i=1:length(myCircuit.Device)%set clock field for all cells
    if isa(myCircuit.Device{i},'QCASuperCell')
```

```
    for j=1:length(myCircuit.Device{i}.Device)



        if strcmp(myCircuit.Device{i}.Device{j}.SelectBox.
           Selected,'on')
            x(end+1)=myCircuit.Device{i}.Device{j}.
               CenterPosition(1);
            y(end+1)=myCircuit.Device{i}.Device{j}.
               CenterPosition(2);
            myCircuit.Device{i}.Device{j}.ElectricField  =
               [0 str2num(handles.changeInputField.String)
               0 ];
        end
    end


else



    if strcmp(myCircuit.Device{i}.SelectBox.Selected,'on')
        x(end+1)=myCircuit.Device{i}.CenterPosition(1);
        y(end+1)=myCircuit.Device{i}.CenterPosition(2);
        myCircuit.Device{i}.ElectricField  = [0 str2num(
           handles.changeInputField.String) 0 ];
    end
    myCircuit.Device{i}.ElectricField
end
```

```
end



x;
y;

%relax and redraw

xdiff=0;
ydiff=0;

% must draw the circuit before we can draw the arrows denoting
   the electric
% field lines
% myCircuit=myCircuit.CircuitDraw(gca);

if ~isempty(x)
    if (length(x) > 1)
        xdiff = (max(x)-min(x));
        ydiff = (max(y)-min(y));

        midy = (max(y)+min(y))/2;
        midx = (max(x)+min(x))/2;

%         mySignal = mySignal.drawElectrode([midx, midy, 0],
   ydiff, xdiff,inputfield);
        center = [midx midy 0];
        height = ydiff;
```

```
        width = xdiff;

    elseif length(x) == 1
        xdiff = 0;
        ydiff = (max(y)-min(y));
%          mySignal = mySignal.drawElectrode([x(1), y(1), 0],
   ydiff,xdiff,inputfield);

        center = [x(1) y(1) 0];
        height = ydiff;
        width = xdiff;
    end

%     setappdata(gcf,handles.signalEditor.String,mySignal);
end

setappdata(gcf,'myCircuit',myCircuit);

end
```

Listing A.24. GetBoxTraits Function

```
function GetSignalPropsGUI(handles)
%This is the precursor to editing a signal and saving it.  When
    the user
%selects a signal from the signal list in the gui, its name and
    type will
%be displayed, and the signal if it is Fermi, custom/piecewise,
    or sinusoidal, will be plotted
```

```
contents = cellstr(get(handles.signalList,'String')); %get the
   signal list from handles



if contents{1}~=""
    sigName = contents{get(handles.signalList,'Value')};

    clockSignalsList = getappdata(gcf,'clockSignalsList'); %get
        the signal list from the app data that already exists



    if ~isempty(clockSignalsList)

        for i=1:length(clockSignalsList)
            clockSignalsList{i}.Name;
            if strcmp(sigName,clockSignalsList{i}.Name)
                mySignal = clockSignalsList{i};
                pick = i;
            end
        end

        handles.signalEditor.String = sigName;
        handles.signalEditType.String = mySignal.Type;
        handles.signalName.String = sigName;

        clockSignalsList{pick} = mySignal;
```

```
Type = mySignal.Type;

SignalTypePanelSwitch(handles,Type); %make the right
   panel pop up

switch Type

    case 'Sinusoidal'
        handles.signalType.Value = 1;



        handles.changeAmp.String = num2str(mySignal.
           Amplitude); %display the properties in the
           edit boxes
        handles.changeWave.String = num2str(mySignal.
           Wavelength);
        handles.changePeriod.String = num2str(mySignal.
           Period);
        handles.changePhase.String = num2str(mySignal.
           Phase);
        handles.changeMeanValue.String = num2str(
           mySignal.MeanValue);


    case 'Fermi'
        handles.signalType.Value = 2;
```

```
            handles.changeAmpFermi.String = num2str(
               mySignal.Amplitude); %display the properties
                in the edit boxes
            handles.changeWaveFermi.String = num2str(
               mySignal.Wavelength);
            handles.changePeriodFermi.String = num2str(
               mySignal.Period);
            handles.changePhaseFermi.String = num2str(
               mySignal.Phase);
            handles.changeMeanValueFermi.String = num2str(
               mySignal.MeanValue);
            handles.changeSharpnessFermi.String = num2str(
               mySignal.Sharpness);


        case 'Custom'
            handles.signalType.Value = 3;
            %nothing yet, but it will also get plotted once
                custom
            %becomes a functionality

        case 'Electrode'
            handles.signalType.Value = 4;

            handles.changeInputField.String = num2str(
               mySignal.InputField);

    end
```

```
        PlotSignal(handles, mySignal);



        setappdata(gcf,'clockSignalsList',clockSignalsList);
    end
else
    disp('No Signals')
end


end
```

Listing A.25. GetSignalPropsGUI Function

```
%WE WANT SNAP TO GRID (correctly)
%panning
%drag and drop
%dumb it down a little
%check this out in GUI
%GUIDE: snap; drag/drop; axes


f=figure;



drag_drop(f);




%drag and drop
```

Listing A.26. GuiDemo Function

```
function LoadCircuit(handles)
%   A circuit/signal file can be selected from the dialog box
    upon calling
%   this function.  Once the circuit is loaded, the user can
    interact with
%   and change that circuit and the signals with it as well.

Path = getappdata(gcf,'Path');

myCircuit = getappdata(gcf,'myCircuit'); %going to replace this
     current circuit
clockSignalsList = getappdata(gcf,'clockSignalsList'); %same
    with this SignalsList



%We use paths to move between the folders where we use the gui
    and where we
%place the .mat files.

home = Path.home;

handles.signalList.String='';

[newFile pathname] = uigetfile('*.mat');



%if the user cancels, then nothing goes wrong
```

```
if ~newFile
    cd(home);
else

    if ~strcmp(pathname(1:end-1),home)
        cd(pathname);
        copyfile(newFile,home);
        cd(home);



    end

    loader=load(newFile);

    %replacing the old circuit and signals list with the loaded
        data
    myCircuit = loader.Circuit;
    clockSignalsList = loader.clockSignalsList;

    for i=1:length(clockSignalsList)
        handles.signalList.String{end+1,1} = clockSignalsList{i
           }.Name;
    end

    setappdata(gcf,'clockSignalsList',clockSignalsList);
    myCircuit = myCircuit.CircuitDraw(0,gca);
```

```
    %the file stays in the folder we put it originally
    if ~strcmp(pathname(1:end-1),home)
        delete(newFile);
    end


    setappdata(gcf,'myCircuit',myCircuit);

end



end
```

Listing A.27. LoadCircuit Function

```
function LoadSimResults(handles)
%This function will use the simulation .mat files to create a
   video with
%the help of the PipelineVisualization function

[Sim, path]= uigetfile('*.mat'); %path gets sent into Pipeline
   in order to change the path, that way we can put the video
   file anywhere

% f=gcf;


% f.Pointer = 'watch';
```

```
if Sim
    disp('Simulation Loaded.')
end

load(Sim);

myCircuit = obj;



setappdata(gcf,'myCircuit',myCircuit);
setappdata(gcf,'SimResults', Sim);
setappdata(gcf,'SimResultsPath', path);

myCircuit = getappdata(gcf,'myCircuit');
%cla;
myCircuit = myCircuit.CircuitDraw(0, gca);

setappdata(gcf,'myCircuit',myCircuit);



% f.Pointer = 'arrow';
% f.Pointer;
end
```

Listing A.28. LoadSimResults Function

```
function MajorPanelSwitch(handles, panel)
%This function is for the three main panels (circuit, signal,
   simulation).
```

```
%Handles dictates which panel will be visible depending on what
    toggle
%button is selected in the gui.

switch panel
    case 'circuit'
        handles.circuitPanel.Value = 1;
        handles.circuitButtonGroup.Visible = 'on';

        handles.signalPanel.Value = 0;
        handles.signalButtonGroup.Visible = 'off';

        handles.simulatePanel.Value = 0;
        handles.simulateButtonGroup.Visible = 'off';


    case 'signal'
        handles.circuitPanel.Value = 0;
        handles.circuitButtonGroup.Visible = 'off';

        handles.signalPanel.Value = 1;
        handles.signalButtonGroup.Visible = 'on';

        handles.simulatePanel.Value = 0;
        handles.simulateButtonGroup.Visible = 'off';


    case 'simulate'
```

```
        handles.circuitPanel.Value = 0;
        handles.circuitButtonGroup.Visible = 'off';

        handles.signalPanel.Value = 0;
        handles.signalButtonGroup.Visible = 'off';

        handles.simulatePanel.Value = 1;
        handles.simulateButtonGroup.Visible = 'on';

end
```

Listing A.29. MajorPanelSwitch Function

```
function MakeSuperCellGUI()
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
myCircuit = getappdata(gcf,'myCircuit');



%which elements of the circuit will be made into a supercell
SCParts=[];



%find out which ones will be put into the new supercell. Cannot
%add current supercell members to another supercell
for i=1:length(myCircuit.Device)
    if  ~isa(myCircuit.Device{i},'QCASuperCell') && strcmp(
       myCircuit.Device{i}.SelectBox.Selected,'on') && strcmp(
       myCircuit.Device{i}.Type,'Node')
```

```
        SCParts(end+1)=i; %need to call the cells to become
            part of the super cell
    end
end



%filling supercell with all the devices
if length(SCParts)>1
    SuperCell = QCASuperCell();

    for i=1:length(SCParts)

        SuperCell = SuperCell.addCell(myCircuit.Device{SCParts(
            i)});
        myCircuit.Device{SCParts(i)}={};%emptying the circuit
            if that part was selected

    end

    %emptying the old circuit into the new circuit
    newCircuit={};
    for i=1:length(myCircuit.Device)
        if ~isempty(myCircuit.Device{i})
            newCircuit{end+1} = myCircuit.Device{i};
        end
    end

    %the new circuit is now the current circuit
```

```
    myCircuit.Device = newCircuit;

    %add the supercell onto the end of the new circuit
    myCircuit=myCircuit.addNode(SuperCell);

    %          myCircuit.GetCellIDs(myCircuit.Device)
    myCircuit.Device;


    %          setappdata(gcf,'myCircuit',myCircuit);
    %          myCircuit=myCircuit.CircuitDraw(gca);
end


myCircuit.Device;


myCircuit=myCircuit.CircuitDraw(0,gca);




setappdata(gcf,'myCircuit',myCircuit);




end
```

Listing A.30. MakeSuperCellGUI Function

```
function NewCircuit(handles)
%The axes are cleared for a new circuit to be made.
f=gcf;

a=gca;
myCircuit = getappdata(f,'myCircuit');
clockSignalsList = getappdata(f,'clockSignalsList');
cla;%clear the axes
plot(handles.plotAxes,0,0);



copies = getappdata(f,'Copies');

copies={};

clockSignalsList = {}; %reset all the visual aspects of the gui
handles.signalList.String = '';
handles.signalList.Value = 1;
handles.signalEditor.String = '' ;
handles.signalEditType.String = '';

handles.signalType.Value = 1;
        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'off';
        handles.electrodePanel.Visible = 'off';
```

```
setappdata(f,'clockSignalsList',clockSignalsList); %must set
   app data before CircuitDraw() happens



myCircuit.Device{1}={};%empty first node

myCircuit.Device=myCircuit.Device{1};%the empty first device

myCircuit.Mode = 'Simulation';
myCircuit = myCircuit.CircuitDraw(gca);



setappdata(f,'myCircuit',myCircuit);
setappdata(f,'Copies',copies);

end
```

Listing A.31. NewCircuit Function

```
function PasteCells()
%Paste the cells that were just copied.



    %get both the circuit and the copied cells from the current
        fig
    myCircuit = getappdata(gcf,'myCircuit');

    copies = getappdata(gcf,'Copies');
```

```
xvals=[];
%collect all the x values
for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        for j=1:length(myCircuit.Device{i}.Device)
            xvals(end+1) = myCircuit.Device{i}.Device{j}.
               CenterPosition(1);

        end
    else

        xvals(end+1) = myCircuit.Device{i}.CenterPosition(1);
    end


end


%find the max x val so we know where to draw the copied cells
xmax = max(xvals);

% if xmax == 0
%     xmax = 1;
% end

%shift all the copied cells by the same amount
```

```
for i=1:length(copies)
    if isa(copies{i},'QCASuperCell')
        for j=1:length(copies{i}.Device)
            copies{i}.Device{j}.CenterPosition(1) = copies{i}.
               Device{j}.CenterPosition(1)+xmax+1;



        end
    else

        copies{i}.CenterPosition(1) = copies{i}.CenterPosition
           (1)+xmax+1;




    end


end


    %add the copied cells to the circuit

    for i=1:length(copies)

       myCircuit = myCircuit.addNode(copies{i}); %put them all
          in the circuit
```

```
    end



    %draw
    myCircuit = myCircuit.CircuitDraw(gca);



    %reset the center position of the copied cells so the next
       time we
    %paste it doesn't add too much to the center position of
       the cells
for i=1:length(copies)
    if isa(copies{i},'QCASuperCell')
        for j=1:length(copies{i}.Device)
            copies{i}.Device{j}.CenterPosition(1) = copies{i}.
               Device{j}.CenterPosition(1)-xmax-1;



        end
    else

        copies{i}.CenterPosition(1) = copies{i}.CenterPosition
           (1)-xmax-1;




    end
```

```
end

    %set it all back into the figure
    setappdata(gcf,'myCircuit',myCircuit);

    setappdata(gcf,'Copies',copies);

end
```

Listing A.32. PasteCells Function

```
function f = PeriodicFermi( x, xperiod, sharpness )
%PeriodicFermi constructs a periodic function using Fermi
   Transitions.
%   Detailed explanation goes here

% t1 = xperiod/4;
% t2 = 3*xperiod/4;
%
% xl = mod(x-t1, xperiod);
%
% f = FermiTransition(xl, t1, sharpness) - FermiTransition(xl,
   t2, sharpness);



t1 = xperiod/4;
t2 = 3*xperiod/4;
% xl = mod(x-t1, xperiod);
```

```
f = FermiTransition(x, t1, sharpness) - FermiTransition(x, t2,
    sharpness);




end
```

Listing A.33. PeriodicFermi Function

```
function PipelineVisualization( simresults, targetaxis, path,
    varargin )
%This function takes in a file and visualizes the simulation.
%This should be used in conjuction with circuit function
    pipeline()
%   draw the circuit at each time step.
%   things we care about for this: The signal info to construct
%   the gradient, and the cells positions, pol and act at each
    time step


home = pwd;
cd(home);
path;


f=gcf;
f.Units = 'pixels';


% w8bar = waitbar(0,'Please wait...');
% w8bar.Position = w8bar.Position + 50;


pause(.5);
```

```
switch nargin

    case 3
        vfilename = 'CircuitVideo.mp4';
        downsamplerate = 2;



    case 4
        vfilename = varargin{1};
        downsamplerate = 2;

    case 5
        downsamplerate = varargin{2};
        vfilename = varargin{1};

    otherwise
        disp('other')

end

cd(path);
load(simresults);
cd(home);
myCircuit = obj;

%get all of the xpositions
xit = 1;
```

```
% waitbar(0, w8bar , 'Processing Simulation');


for i=1:length(obj.Device)
%     waitbar(i/length(obj.Device), w8bar , 'Retrieving Circuit
    and Signal data...');


    if isa(obj.Device{i},'QCASuperCell')
        for j=1:length(obj.Device{i}.Device)
            center = obj.Device{i}.Device{j}.CenterPosition;
            xpos(xit) = center(1);
            ypos(xit) = center(2);
            xit = xit + 1;
        end
    else
        center = obj.Device{i}.CenterPosition;
        xpos(xit) = center(1);
        ypos(xit) = center(2);
        xit = xit + 1;
    end
end



xmax = max(xpos);
xmin = min(xpos);
```

```
ymax = max(ypos);
ymin = min(ypos);



x = xmin:xmax;
nx = 125;
xq = linspace(xmin-1, xmax+1, nx);
yq = linspace(ymin-2, ymax+2, nt);

if (length(clockSignalsList) == 1)
    clockSignal = clockSignalsList{1};
else
    error('Too many signals')
end
tperiod = clockSignal.Period*numOfPeriods;
time_array = linspace(1,tperiod,nt);

tp = mod(time_array, tperiod);
% xp = mod(xq, signal.Period);

% get the clock field
% efields_mat = cell2mat(efields(:,:));
% zfields = efields_mat(:,3:3:end);
% zmax = max(max(zfields));
% zmin = min(min(zfields));



%reconstruct signal
```

```
% for t = 1:size(pols,1)
%     waitbar( t / size(pols,1) , w8bar , 'Reconstructing
   Signal...');
%
%     for idx = 1:nx
%
%         efplots_temp = signal.getClockField([xp(idx),0,0], tp
   (t));
%         efplots(t, idx) = efplots_temp(3);
%
%     end
% end









%
Frame(nt) = struct('cdata',[],'colormap',[]);
v = VideoWriter(vfilename,'MPEG-4');
open(v);

myCircuit.Simulating = 'on';

% f.Pointer = 'watch';

f=gcf;
```

```
% waitbar(0, w8bar , 'Writing to Video File...');

maxheight=f.Position(4);
maxwidth=f.Position(3);

sizePol = size(pols,1);




% epsilon_0 = 8.854E-12;
% a=1e-9;%[m]
% q=1;%[eV]
% Eo = q^2*(1.602e-19)/(4*pi*epsilon_0*a)*(1-1/sqrt(2));
%
% inputfield = 0.85*Eo;
%
% centerpos = [0,0,0];
% amp = 2*inputfield;
% period = 400;
% phase = period/4;
% sharpness = 3;
% mu = amp/2;
%time_array = linspace(1, period, nt);
%tp = mod(time_array, period);
```

```
for t = downsample(1:sizePol,downsamplerate);



%     cla;

    %ef = efplots(t,:);
%     efz = zfields(t,:);
%
%     interps = interp1(x,efz,xq,'pchip','extrap');
%     Eplot = repmat(ef,[nt,1]);
%     pcolor(xq' * ones(1, nx), ones(nx, 1)* yq, Eplot');
%     colormap cool;
%     shading interp;
%     colorbar;
%     caxis([zmin zmax])


    clockSignal.drawSignal([xmin-1,xmax+1], [ymin-2.5, ymax
       +2.5], tp(t));

    myCircuit = myCircuit.CircuitDraw(t, targetaxis, [pols(t,:)
       ; acts(t,:)]);



    %%%%%%%arrowmod = amp * PeriodicFermi(mod(centerpos(1) -
       tp(t) - phase , period), period, sharpness) + mv;
```

```
%arrowtext = strcat('E_y=',num2str(arrowmod/Eo),'E_o');
%%%%%%%textborder(xmin-3, 0, '$E_{y}$', [0,0,0],[1,1,1], '
   FontSize', 28, 'Interpreter', 'latex')
%rectangle('Position',[xmin-3.1 -0.5 1.3 1],'Curvature
   ',0.2,'FaceColor',[1,1,1]);
%text(xmin-3, 0, '$E_{in}$', 'Color', [0,0,0], 'FontSize',
   28, ...
%    'Interpreter', 'latex')


%%%%%%%arrow([xmin-1 -10*arrowmod], [xmin-1 10*arrowmod],'
   Width',2,'EdgeColor',[1,1,1],'FaceColor',[0,0,0]);

drawnow
axis equal
axis off
%save it
%gcf
%gca
%r = getrect(gcf)
%Frame(t) = getframe(gcf,[0 0 maxwidth*.65 maxheight*.5]);
%Frame(t) = getframe(gca);
Frame(t) = getframe(gcf);


writeVideo(v,Frame(t));
disp(['t: ' num2str(t)])
```

```
end
```

```
myCircuit.Simulating = 'off';
caxis('auto');

a=gca;

a.Box = 'off';
a.YLimMode = 'auto';

colorbar('delete');
close(v);
disp('Complete!')

end
```

Listing A.34. PipelineVisualization Function

```
function PlotSignal(handles, mySignal)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

type = mySignal.Type;
name = mySignal.Name;
```

```
switch type

    case 'Sinusoidal'



        A = str2num(handles.changeAmp.String); %plotting
        L = str2num(handles.changeWave.String);
        T = str2num(handles.changePeriod.String);
        b = str2num(handles.changePhase.String);
        x=-2:.01:12;


        y = ( cos((2*pi*(x/L - 1/T) )+ b) )*A;

        plot(handles.plotAxes,x,y);
        handles.signalDisplayBox.String = name;


    case 'Fermi'


        Amp = str2num(handles.changeAmpFermi.String);
        T = str2num(handles.changePeriodFermi.String);
        b = str2num(handles.changePhaseFermi.String);
        K = str2num(handles.changeSharpnessFermi.String);
        M = str2num(handles.changeMeanValueFermi.String);
```

```
        x=-2:.01:12;


        y = Amp * PeriodicFermi(mod(x  - b , T), T, K) + M;

        plot(handles.plotAxes,x,y);
        handles.signalDisplayBox.String = name;

    case 'Custom'
        %nothing yet, but it will also get plotted once custom
        %becomes a functionality

    case 'Electrode'
        handles.changeInputField.String = num2str(mySignal.
           InputField);

end


end
```

Listing A.35. PlotSignal Function

```
classdef QCACell
    %QCACell Container class for any QCA Cell
    %   Detailed explanation goes here
    %   Actually add a explaination of properties
    %   Also mention the Parent-Child interatction.
```

```
properties
    CellID = 0; % Unique Cell identifier
    Type = 'Node';% Driver/Node
    CenterPosition = [0, 0, 0]; % poisiton of the cell
       center

    DotPosition = []; % [*CharacteristicLength] positions
       of dots
                      %      relative to cell center
    CharacteristicLength = 1; % [nm]


    Gamma = 0.05; % [eV] from Henry, Blair 2017: 0 < gamma
       < 0.3*Ek
                  % for now, 0 <--> 0.1265 for
                     characteristic length
                  % 1nm


    ElectricField = [0, 0, 0]; %Electric Field [V/nm]

    NeighborList = []; %this Cell's Neighbors


end
```

```
methods
    function obj = QCACell( varargin )
        % obj = QCACell( varargin )
        %
        % Detailed



        switch nargin
            case 0

            case 1 % Q = QCACell( [x,y,z] )
                if( isnumeric(varargin{1}) )
                   if(size(varargin{1}) == [1, 3])
                    obj.CenterPosition = varargin{1};
                   else
                        error('Incorrect data input size.')
                   end
                else
                   error('Incorrect data input type.')
                end

            otherwise
                error('Invalid number of inputs for QCACell
                   ');



        end % END: Switch nargin
```

```
    end

    function obj = set.Type(obj,value)
        if (~isequal(value, 'Driver') && ~isequal(value,'
           Node'))
            error('Invalid Type. Must be Driver or Node')
        else
            obj.Type = value;
        end
    end

    function obj = set.ElectricField(obj,value)
        if (isequal(size(obj.ElectricField), size(value)))
            obj.ElectricField = value;
        else
            error('Electric field must by [x,y,z]')
        end
    end

    function p = getCharacteristicLength(obj)
        p = obj.CharacteristicLength;
    end

    function obj = translateCell(obj,TranslationVect)
        % Q = Q.translateCell( TranslationVect )
        %
        %       Q is an implicit first arguement
```

```
        %
        %       Q is not modified outside of this method
           unless we
        %       reassign it (i.e. we need "Q = Q.
           translateCell(...)")
        obj.Position = obj.CenterPosition + TranslationVect
           ;



    end

    function TrueDotPosition = getDotPosition( obj )
        % R = Q.getDotPosition returns the absolute
           coordinates of the
        % dots of cell Q
        %
        % If Q has n dots, then R is a n-by-3 matrix, with
           the kth row
        % representing the xyz-triple for th kth dot

        %number of rows in obj.DotPosition
        n = size(obj.DotPosition, 1);

        %position of cell dots relative to axes position
        TrueDotPosition = ones(n,1) * obj.CenterPosition +
           ...
                obj.CharacteristicLength*obj.DotPosition;
```

```
        end




    end

end
```

Listing A.36. QCACell Class

```
classdef QCACircuit
    % make a class "qca circuit" that contains cells and knows
       which ones are
    % neighbors. Should be able to individually save file
       without corrupting
    % the original program.

    %%
    properties
        Device = {}; % QCA CELL ARRAY
        RefinedDevice = {};
        GroundState = [];
        Mode='Simulation';
        SnapToGrid = 'off';
        Simulating = 'off';
    end
    %%
    methods
        %%
```

```
function obj = QCACircuit( varargin ) % constructor
    class

     if nargin > 0
         placeholder = obj.Device;
         obj.Device = {placeholder varargin};

     else
         % Nothing happens

     end

end
%%
function obj = addNode( obj, newcell )
     n_old = length(obj.Device);
     ids=[];

     if length(obj.Device)
         ids = obj.GetCellIDs(obj.Device);
     end

     obj.Device{n_old+1} = newcell;
     obj.Device{n_old+1}.CellID = length(obj.Device);

     if length(ids)
         for i=1:n_old  %ensuring that CellIDs cannot be
             repeated
```

```
            compare = (obj.Device{n_old+1}.CellID==
               floor(ids));
            maxID = floor(max(ids));
            if sum(compare)>0
                obj.Device{n_old+1}.CellID = maxID +1;
            end


        end
    end
    compare = (obj.Device{n_old+1}.CellID==ids);
    ids = obj.GetCellIDs(obj.Device);

    newIDs = obj.GetCellIDs(obj.Device);
    if isa(newcell, 'QCASuperCell')
        newcell = obj.Device{n_old+1}; %call just
           recently added supercell, newcell


        for x = 1:length(newcell.Device) % edit each
           subcell's CellID to reflect the supernode's
           integer
            newcell.Device{x}.CellID = newcell.Device{x
               }.CellID + newcell.CellID;
        end
        obj.Device{n_old+1} = newcell;
```

```
        end



    end
    %%
    function obj = GenerateNeighborList( obj )
        %this function steps through each cell and assigns
           the neighborList for each

        %All CellID Array (including subcells)
        cellIDArray = [];
        cellpositions=[];



        node = 1;
        for node=1:length(obj.Device)
            if(isa(obj.Device{node}, 'QCASuperCell'))
                for subnode = 1:length(obj.Device{node}.
                   Device)
                    cellpositions(end+1,:) = obj.Device{
                       node}.Device{subnode}.CenterPosition
                       ;


                end


            else
                cellpositions(end+1,:) = obj.Device{node}.
                   CenterPosition;
```

```
    end

end

cellIDArray = obj.GetCellIDs(obj.Device);
cellpositions = cellpositions';

cellIDToplevelnodes = floor(cellIDArray);


%now go through list of CellID's to find neighbors


cellposit = 1;
for idx=1:length(obj.Device)

    leng = length(obj.Device);


    if(isa(obj.Device{idx}, 'QCASuperCell')) %if
       supernode overwrite supernode ID with first
       subnode
        superCellID = obj.Device{idx}.CellID;

        len=length(obj.Device{idx}.Device);
```

```
                for subnode = 1:length(obj.Device{idx}.
                    Device)

                    subnode;
                    c = obj.Device{idx}.Device{subnode}.
                        CellID;
                    cellpositions(:,idx+subnode-1);
                    %shift and find magnitudes  idx+subnode
                        -1
                    shifted = cellpositions - repmat(
                        cellpositions(:,cellposit),1,length(
                        cellIDArray));
                    shifted = shifted.^2;
                    shifted = sum(shifted,1);
                    shifted = shifted.^(.5);


                    %give me the cellid's of the node
                        within a certain
                    %limit. Limit depends on object type
                    switch class(obj.Device{idx}.Device{
                        subnode})
                        case 'ThreeDotCell'
                            neighbors = cellIDArray(shifted
                                < 6.1 & shifted > 0.1); %or
                                2.25
                        case 'SixDotCell'
```

```
                        neighbors = cellIDArray(shifted
                            < 3.1 & shifted > 0.1); %or
                            2.25
                    end

                    obj.Device{idx}.Device{subnode}.
                       NeighborList = neighbors;
                    cellposit = cellposit+1;
                end

            else
                %shift and find magnitudes

                shifted = cellpositions - repmat(
                   cellpositions(:,cellposit),1,length(
                   cellIDArray));
                shifted = shifted.^2;
                shifted = sum(shifted,1);
                shifted = shifted.^(.5);

                %give me the cellid's of the node within a
                   certain limit
                id = obj.Device{idx}.CellID;

                switch class(obj.Device{idx})
                    case 'ThreeDotCell'
                        neighbors = cellIDArray(shifted <
                           3.1 & shifted > 0.1); %or 2.25
```

```
                    case 'SixDotCell'
                        neighbors = cellIDArray(shifted <
                            3.1 & shifted > 0.1); %or 2.25
                end


                %                   disp(['id: ' num2str(
                    id) ' neighbors: ' num2str(neighbors)])
                obj.Device{idx}.NeighborList = neighbors;
                cellposit = cellposit+1;

            end




        end


    end
    %%
    function obj = CircuitDraw(obj,time,targetaxis,varargin
        )
        %cla;


        hold on
        CellIndex = length(obj.Device);
```

```
        snapmode = obj.SnapToGrid;


        if length(varargin) == 1
            polact = varargin{1};
            pols = polact(1,:);
            acts = polact(2,:);


            it = 1;
            % format for iterating through circuit
            for idx=1:length(obj.Device)
                if isa(obj.Device{idx},'QCASuperCell')
                    for sub=1:length(obj.Device{idx}.Device
                       )
                        %assign pol and act
                        obj.Device{idx}.Device{sub}.
                           Polarization = pols(it);
                        obj.Device{idx}.Device{sub}.
                           Activation = acts(it);
                        it = it + 1;
                    end


                else
                    %assign pol and act
                    obj.Device{idx}.Polarization = pols(it)
                       ;
                    obj.Device{idx}.Activation = acts(it);
                    it = it + 1;
                end
```

```
    end


elseif length(varargin) > 1
    error('Too many arguments')


end
%normal functionality

switch snapmode %snapping to grid mode
    case 'off' %do nothing extra

    case 'on' %begin snapping each cell to the grid
       , skipping every .5
        obj = obj.Snap2Grid();

end %end snap to grid


obj = obj.AntiOverlap();

if strcmp(obj.Simulating,'off')
    cla;

end
```

```
for CellIndex = 1:length(obj.Device)
    if( isa(obj.Device{CellIndex}, 'QCASuperCell')
       )

        %check to see if there is a color for the
           SC
        if strcmp(obj.Device{CellIndex}.BoxColor,''
           )

            %We make a cell array of all colors
               that have been
            %used
            colors=0;
            for j=1:length(obj.Device)
                if isa(obj.Device{j},'QCASuperCell'
                   ) && ~isempty(obj.Device{j}.
                   BoxColor) && j~= CellIndex
                    colors =  colors+1;
                end

            end

            if colors>0
                id = floor(obj.Device{CellIndex}.
                   Device{1}.CellID);
```

```
                color(1)= abs(sin(.4*id*now/100000-
                    id));
                color(3)= abs(sin(colors*id-(id^2))
                    *abs(cos(id)));
                color(2)= abs(cos(colors*id + id*(
                    id-1)*now/100000));

                obj.Device{CellIndex}.BoxColor=
                    color;
            else
                obj.Device{CellIndex}.BoxColor=[
                    rand rand rand]; %the color will
                     remain the same for the same
                    super cell
            end

        else
            %don't make a new color
        end

        for subnode = 1:length(obj.Device{CellIndex
            }.Device)

            %obj.Device{CellIndex}.Device{subnode}
                = obj.Device{CellIndex}.Device{
                subnode}.ElectronDraw(time,
                targetaxis);
```

```
                        obj.Device{CellIndex}.Device{subnode} =
                            obj.Device{CellIndex}.Device{
                           subnode}.ColorDraw(targetaxis);
                        obj.Device{CellIndex}.Device{subnode} =
                            obj.Device{CellIndex}.Device{
                           subnode}.BoxDraw();
                        obj.Device{CellIndex}.Device{subnode}.
                           SelectBox.Selected = 'off';
                        obj.Device{CellIndex}.Device{subnode}.
                           SelectBox.EdgeColor = obj.Device{
                           CellIndex}.BoxColor; %turns on and
                           off the supercell color
                        obj.Device{CellIndex}.Device{subnode}.
                           SelectBox.LineWidth = 3;
                        Select(obj.Device{CellIndex}.Device{
                           subnode}.SelectBox);
                    end
                else

%                       if isa(obj.Device{CellIndex},'SixDotCell
    ')
%                           obj.Device{CellIndex} = obj.Device{
    CellIndex}.ColorDraw(time, targetaxis);
%                       else
%                           obj.Device{CellIndex} = obj.Device{
    CellIndex}.ElectronDraw(time,targetaxis);
%                       end
```

```
                if CellIndex == 2 || CellIndex == 4 ||
                   CellIndex == 6
                    obj.Device{CellIndex} = obj.Device{
                       CellIndex}.ColorDraw(time,
                       targetaxis);
                else
                    obj.Device{CellIndex} = obj.Device{
                       CellIndex}.ElectronDraw(time,
                       targetaxis);
                end
                %obj.Device{CellIndex} = obj.Device{
                   CellIndex}.ColorDraw(time, targetaxis);
                %obj.Device{CellIndex} = obj.Device{
                   CellIndex}.ElectronDraw(time,targetaxis)
                   ;
                obj.Device{CellIndex} = obj.Device{
                   CellIndex}.BoxDraw();
                obj.Device{CellIndex}.SelectBox.Selected =
                   'off';
                %obj.Device{CellIndex}.SelectBox.FaceAlpha
                   = .01;

                Select(obj.Device{CellIndex}.SelectBox);

            end

        end
```

```
        RightClickThings();    %uicontextmenu available upon
            drawing

        hold off
        grid on

        %DrawElectrodes();

        axis equal

    end
    %%
    function obj =  AntiOverlap(obj)
        %              clc;
        IDList = obj.GetCellIDs(obj.Device);
        cellList = obj.getCellArray(IDList);




        diffs=[];

        for i=1:length(cellList)
            for j=1:length(cellList)%j=length(cellList):-1:
               i+1
                if i~=j
```

```
                diffs(1,end+1) = cellList{i}.
                    CenterPosition(1) - cellList{j}.
                    CenterPosition(1);
                diffs(2,end) = cellList{i}.
                    CenterPosition(2) - cellList{j}.
                    CenterPosition(2);
                diffs(3,end) = cellList{i}.CellID;


            end
        end
    end



    diffs;
    sizeof = size(diffs);

    OL=[];
    overlap = 0;
    for i=1:sizeof(2)
        if abs(diffs(1,i)) < .499 && abs(diffs(2,i)) <
            1.499
             overlap = overlap +1;
             OL(end+1) = diffs(3,i);


        end
    end
```

```
            for i=1:length(obj.Device)
                if isa(obj.Device{i},'QCASuperCell')

                    for j=1:length(obj.Device{i}.Device)

                        if sum(obj.Device{i}.Device{j}.CellID
                           == OL) > 0
                            obj.Device{i}.Device{j}.Overlapping
                                = 'on';

                        else
                            obj.Device{i}.Device{j}.Overlapping
                                = 'off';

                        end




                    end

                else

                    if sum(obj.Device{i}.CellID == OL) > 0

                        obj.Device{i}.Overlapping = 'on';

                    else
```

```
                obj.Device{i}.Overlapping = 'off';

            end
            obj.Device{i}.CellID;
            obj.Device{i}.Overlapping;
        end
    end
end
%%
function sref = subsref(obj,s) %reference this based on
    CellId
    % obj(index) is the same as obj.Device(index)

    switch s(1).type
        case '.'
            sref = builtin('subsref',obj,s);
        case '()'

            if length(s) < 2
                s.type = '{}';
                sref = builtin('subsref',obj.Device,s);
                return
            else
                s(1).type = '{}';
                sref = builtin('subsref',obj.Device,s);
            end
        case '{}'
```

```
                if length(s) < 2
                    sref = builtin('subsref',obj.Device,s);
                    return
                else
                    sref = builtin('subsref',obj,s);
                end
            end
        end

        %%
        function obj = subasgn(obj,s,val)%assign this based on
            CellId
            if isempty(s) && isa(val,'QCACircuit')
                obj = QCACircuit(val.Device,val.Description);
            end
            switch s(1).type
                case '.'
                    obj = builtin('subsasgn',obj,s,val);
                case '()'

                case '{}'
                    if length(s)<2
                        if isa(val,'QCACircuit')
                            error('Error: Invalid Indexing')
                        elseif isa(val,'double')
                            % Redefine the struct s to make the
                                call: obj.Device(i)
```

```
                    snew = substruct('.','Device','()',
                        s(1).subs(:));
                    obj = subsasgn(obj,snew,val);
                end
            end
        end
    end


    %%
    function all_energy = calculateEnergy( obj, time,
        varargin )

        % use varargin to add in Clock Field Biases



        for nodeIdx=1:length(obj.Device)
            if isa(obj.Device{nodeIdx},'QCASuperCell')

                for subnodeIdx=1:length(obj.Device{nodeIdx
                    }.Device)
                    sub_nl = obj.Device{nodeIdx}.Device{
                        subnodeIdx}.NeighborList;
                    sub_nl_obj = obj.getCellArray(sub_nl);
                    subnode = obj.Device{nodeIdx}.Device{
                        subnodeIdx};
```

```
                    sub_objDotPosition = subnode.
                       getDotPosition();
                    sub_mobileCharges = subnode.
                       getMobileCharge(time);
                    subnode_energy = zeros(length(sub_nl_
                       obj),1);

                    for sub_neighborIdx = 1:length(sub_nl_
                       obj)
                        for x = 1:length(sub_objDotPosition
                           )
                            potential_energy = sub_nl_obj{
                               sub_neighborIdx}.Potential(
                               sub_objDotPosition(x,:),
                               time );

                            subnode_energy(sub_neighborIdx)
                                = subnode_energy(sub_
                               neighborIdx) + potential_
                               energy*sub_mobileCharges(x);
                        end
                    end

                    all_subnode_energy(subnodeIdx) = sum(
                       subnode_energy);

                end %end subnode
```

```
            %all_subnode_energy = sum(subnode_energy);
            all_energy(nodeIdx) = sum(all_subnode_
               energy);

        else %start non-supercell
            nl = obj.Device{nodeIdx}.NeighborList;
            nl_obj = obj.getCellArray(nl);
            node = obj.Device{nodeIdx};

            objDotPosition = node.getDotPosition();
            mobileCharges = node.getMobileCharge(time);
            node_energy = zeros(length(nl_obj),1);

            for neighborIdx = 1:length(nl_obj)
                for x = 1:length(objDotPosition)
                    potential_energy = nl_obj{
                       neighborIdx}.Potential(
                       objDotPosition(x,:), time );

                    node_energy(neighborIdx) = node_
                       energy(neighborIdx) + potential_
                       energy*mobileCharges(x);
                end
            end

            all_energy(nodeIdx) = sum(node_energy);
```

```
                end %end non-supercell
            end %end stepping through circuit




%             all_energy = zeros(length(obj.Device),1);
%             for nodeIdx = 1:length(obj.Device)
%
%                 nl = obj.Device{nodeIdx}.NeighborList;
%                 nl_obj = obj.getCellArray(nl);
%
%                 node = obj.Device{nodeIdx};
%
%                 objDotpotential = zeros(size(node.DotPosition
   ,1),1);
%                 objDotPosition = node.getDotPosition();
%                 mobileCharges = node.getMobileCharge(time);
%                 node_energy = zeros(length(nl_obj),1);
%
%                 for neighborIdx = 1:length(nl_obj)
%                     for x = 1:length(objDotPosition)
%                         %V_neighbors(x,:) = obj2.Potential(
   objDotPosition(x,:), time );
%                         potential_energy = nl_obj{neighborIdx
   }.Potential( objDotPosition(x,:), time );
%
%                         node_energy(neighborIdx) = node_
   energy(neighborIdx) + potential_energy*mobileCharges(x);
```

```
%                       end
%
%
%                   end
%
%                   all_energy(nodeIdx) = sum(node_energy);
%
%
%
%               end
%
%
%           end

        end

        %%
        function obj = Relax2GroundState(obj, time, varargin)

            %Iterate to Self consistency

            if length(varargin) == 1
                disp(num2str(varargin{1}))
            end

            NewCircuitPols = ones(1,length(obj.Device));
            convergance Tolerance = 1;
            sub = 1;
```

```
chi = 0.6;
it=1;
oldmit = 5;
while (convergenceTolerance > 0.1 ) %&& it < 600
    if(it > 500)

        newmit = fix(it/100);
        if newmit > oldmit
            chi = chi - 0.1 % or multiply by 0.9
        end

        if chi <= 0
            chi = 0.1
        end
        oldmit = newmit;
    end


    OldCircuitPols = NewCircuitPols;

    idx = 1;

    while idx <= length(obj.Device)

        if( isa(obj.Device{idx}, 'QCASuperCell') )

            NewPols = ones(1,length(obj.Device{idx
               }.Device));
```

```
subnodeTolerance = 1;
super = 1;



while (subnodeTolerance > 0.00001)
    OldPols = NewPols;

    %supernode = floor(obj.Device{idx}.
       Device{1}.CellID)



    L=length(obj.Device);
    for subnode = 1:length(obj.Device{
       idx}.Device)

        if( strcmp(obj.Device{idx}.
           Device{subnode}.Type, '
           Driver') )
            %don't relax
        else

            id = obj.Device{idx}.Device
               {subnode}.CellID;
            nl = obj.Device{idx}.Device
               {subnode}.NeighborList;
            pol = obj.Device{idx}.
               Device{subnode}.
               Polarization;
```

```
%disp(['id: ', num2str(id)
    ,' nl: ', num2str(nl)
    ,' pol: ', num2str(pol)
    ])

if length(varargin) == 1
    obj.Device{idx}.Device{
        subnode}.
        ElectricField(3) =
        varargin{1};
end

if ~isempty(nl)

    %get Neighbor Objects

    nl_obj = obj.
        getCellArray(nl);

    %get hamiltonian for
        current cell
    hamiltonian = obj.
        Device{idx}.Device{
        subnode}.
        GetHamiltonian(nl_
        obj, time);
```

```
obj.Device{idx}.Device{
    subnode}.Hamiltonian
     = hamiltonian;



[V, EE] = eig(
    hamiltonian);
newpsi = V(:,1);

normpsi = (1-chi)*obj.
    Device{idx}.Device{
    subnode}.
    Wavefunction + chi*
    newpsi;
normpsi = normalize_psi
    _1D(normpsi');



%calculate polarization
obj.Device{idx}.Device{
    subnode} = obj.
    Device{idx}.Device{
    subnode}.Calc_
    Polarization_
    Activation(normpsi')
    ;
```

```
                                NewPols(subnode) = obj.
                                    Device{idx}.Device{
                                    subnode}.
                                    Polarization;



                                % disp(['id: ', num2str
                                    (id), ' pol: ',
                                    num2str(pol)]) %, '
                                    nl: ', num2str(nl)
                            else
                                disp('potential room
                                    for thinking')%else
                                    nl is empty
                            end

                        end

                    end

                    deltaPols = abs(OldPols) - abs(
                        NewPols);
                    subnodeTolerance = max(abs(
                        deltaPols));
                    super = super + 1;
                end

                idx=idx+1;
```

```
else
    %obj.Device{idx} = obj.Device{idx}.Calc
       _Polarization_Activation();
    id = obj.Device{idx}.CellID;
    nl = obj.Device{idx}.NeighborList;
    pol = obj.Device{idx}.Polarization;
    if length(varargin) == 1
        obj.Device{idx}.ElectricField(3) =
           varargin{1};
    end

    if ~isempty(nl)



        %get Neighbor Objects
        nl_obj = obj.getCellArray(nl);



        %get hamiltonian for current cell
        hamiltonian = obj.Device{idx}.
           GetHamiltonian(nl_obj,time);
        obj.Device{idx}.Hamiltonian =
           hamiltonian;
```

```
%get the new groundstate, average
   and normalize
%the current groundstate and the
   new
%groundstate then calculate pol
   with that psi
[V, EE] = eig(hamiltonian);
newpsi = V(:,1);



normpsi = (1-chi)*obj.Device{idx}.
   Wavefunction + chi*newpsi;
normpsi = normalize_psi_1D(normpsi
   ');



%calculate polarization
obj.Device{idx} = obj.Device{idx}.
   Calc_Polarization_Activation(
   normpsi');

if(isa(obj.Device{idx}, '
   QCASuperCell'))
    NewCircuitPols(idx) = 0;
else
```

```
                        NewCircuitPols(idx) = obj.
                            Device{idx}.getPolarization(
                            time);
                    end

                    %disp(['id: ', num2str(id), ' pol:
                        ', num2str(pol)  ' nl: ',
                        num2str(nl)])

                end
                idx = idx+1;
            end



        end
        %                   fprintf('\n');
        deltaCircuitPols = abs(OldCircuitPols) - abs(
            NewCircuitPols);
        [convergance Tolerance, cellindex] = max(abs(
            deltaCircuitPols));

        sub=sub+1;
        it=it+1;
    end
    %disp(['it:', num2str(it)]);
end
```

```
        %%
        function obj = Relax2GroundState_mobilecharge_serial(
           obj, time, varargin)




            %optional changes
            args = varargin(1:end);
            while length(args) >= 2
                prop = args{1};
                val = args{2};
                args = args(3:end);
                switch prop
                    case 'randomizedRelaxation'
                        randFlag = val;
                        %disp(['randomized' num2str(1)])
                    otherwise
                        error(['QCACircuit.Relax2GroundState_
                           mobilecharge_serial ', prop, ' is an
                            invalid property specifier.']);
                end
            end



%               if length(varargin) == 1
%                   disp(num2str(varargin{1}))
%               end
```

```
            NewMobileCharges = zeros(6,length(obj.Device));
            convergenceTolerance = 1;

            chi = 0.6;
            it=1;
            oldmit = 5;


            %set up randomization
            r = 1:length(obj.Device);
            if randFlag
                r = rand(1,length(obj.Device));
            end
            circuit_idx = 1:length(obj.Device);
            [~,s] = sort(r);
            circuit_idx_random = circuit_idx(s);


%             old = obj;
%             old_energy = sum(old.calculateEnergy(time));

            while (convergenceTolerance > 0.01 && it < 1000) %&
               & it < 600
```

```
if(it > 500)

    newmit = fix(it/100);
    if newmit > oldmit
        chi = chi * 0.9; % or multiply by 0.9
    end

    if chi <= 0
        chi = 0.1;
    end
    oldmit = newmit;
end

%%%%%%%%%%%%%%%%%%%%SUBNODE%%%%%%%%%%%%%%%%%%
   %%%%%%%%%%%%

 %deal with super cells first
for idx = 1:length(obj.Device)
    if isa(obj.Device{circuit_idx_random(idx)},
        'QCASuperCell')
       subnodeNewMobileCharges = zeros(6,length
          (obj.Device));
```

```
subnodeTolerance = 1;
subit = 1;
suboldmit = 5;
subchi = 0.6;

while (subnodeTolerance > 0.01)

    if(subit > 500)

        subnewmit = fix(subit/100);
        if subnewmit > suboldmit
            subchi = subchi * 0.9 % or
               multiply by 0.9
        end

        if subchi <= 0
            subchi = 0.1;
        end
        suboldmit = subnewmit;
    end

    subnodeOldMobileCharges =
       subnodeNewMobileCharges;

    %get all hamiltonians
    for sub = 1:length(obj.Device{
       circuit_idx_random(idx)}.Device)
```

```
            nl = obj.Device{circuit_idx_
                random(idx)}.Device{sub}.
                NeighborList;
            if ~isempty(nl)
                %get Neighbor Objects
                nl_obj = obj.getCellArray(nl
                    );

                %get hamiltonian for current
                     cell
                hamiltonian = obj.Device{
                    circuit_idx_random(idx)}.
                    Device{sub}.
                    GetHamiltonian(nl_obj,
                    time);
                obj.Device{circuit_idx_
                    random(idx)}.Device{sub}.
                    Hamiltonian = hamiltonian
                    ;
            end

        end


        %get all groundstates
        sub_groundstates = cell(1,length(obj
            .Device{circuit_idx_random(idx)}.
            Device));
```

```
                    for sub = 1:length(obj.Device{
                       circuit_idx_random(idx)}.Device)
                        hamiltonian = obj.Device{circuit
                           _idx_random(idx)}.Device{sub
                           }.Hamiltonian;
                        wavefunction = obj.Device{
                           circuit_idx_random(idx)}.
                           Device{sub}.Wavefunction;
                        %get the new groundstate,
                           average and normalize
                        %the current groundstate and the
                            new
                        %groundstate then calculate pol
                           with that psi
                        [V, EE] = eig(hamiltonian);
                        newpsi = V(:,1);

                        normpsi = (1-subchi)*
                           wavefunction + subchi*newpsi;
                        normpsi = normalize_psi_1D(
                           normpsi');
                        sub_groundstates{sub} = normpsi;
                    end

                    %update pols and acts and then get
                       all the charges
                    for sub = 1:length(obj.Device{
                       circuit_idx_random(idx)}.Device)
```

```
                    obj.Device{circuit_idx_random(
                        idx)}.Device{sub} = obj.
                        Device{circuit_idx_random(idx
                        )}.Device{sub}.Calc_
                        Polarization_Activation(sub_
                        groundstates{sub}');




                    if(isa(obj.Device{circuit_idx_
                        random(idx)}.Device{sub},'
                        SixDotCell'))
                         subnodeNewMobileCharges(:,
                            sub) = obj.Device{circuit
                            _idx_random(idx)}.Device{
                            sub}.getMobileCharge(time
                            );
                    else
                         subnodeNewMobileCharges(1:3,
                            sub) = obj.Device{circuit
                            _idx_random(idx)}.Device{
                            sub}.getMobileCharge(time
                            );
                    end



                end
```

```
                subnodedeltaMobileCharges =
                   subnodeOldMobileCharges -
                   subnodeNewMobileCharges;

                [subnodeTolerance, subnodecellindex]
                    = max(max(abs(
                   subnodedeltaMobileCharges)));
                subit = subit+1   ;
            end

            %disp(['++++++++++++++++++++++subit:',
               num2str(subit)]);


        end
    end


%%%%%%%%%%%%%%%%%SUBNODE%%%%%%%%%%%%%%%%%%%%
   %%%%%%%%%%%
```

```
                OldMobileCharges = NewMobileCharges;

                %get all hamiltonians
                for idx = 1:length(obj.Device)
                    if ~isa(obj.Device{circuit_idx_random(idx)
                       }, 'QCASuperCell')

                        nl = obj.Device{circuit_idx_random(idx)
                           }.NeighborList;
                        if ~isempty(nl)
                            %get Neighbor Objects
                            nl_obj = obj.getCellArray(nl);

                            %get hamiltonian for current cell
                            hamiltonian = obj.Device{circuit_
                               idx_random(idx)}.GetHamiltonian(
                               nl_obj,time);
                            obj.Device{circuit_idx_random(idx)
                               }.Hamiltonian = hamiltonian;
                        else
                            hamiltonian = obj.Device{circuit_
                               idx_random(idx)}.GetHamiltonian
                               ({}, time);
                            obj.Device{circuit_idx_random(idx)
                               }.Hamiltonian = hamiltonian;
                        end
                    end
                end
```

```
%get all groundstates
groundstate = cell(1,length(obj.Device));
for idx = 1:length(obj.Device)
    if ~isa(obj.Device{circuit_idx_random(idx)
       }, 'QCASuperCell')

        hamiltonian = obj.Device{circuit_idx_
           random(idx)}.Hamiltonian;
        wavefunction = obj.Device{circuit_idx_
           random(idx)}.Wavefunction;
        %get the new groundstate, average and
           normalize
        %the current groundstate and the new
        %groundstate then calculate pol with
           that psi
        [V, EE] = eig(hamiltonian);
        newpsi = V(:,1);

        normpsi = (1-chi)*wavefunction + chi*
           newpsi;
        normpsi = normalize_psi_1D(normpsi');
        groundstate{circuit_idx_random(idx)} =
           normpsi;
    end
end
```

```
%update pols and acts and then get all the
   charges
for idx = 1:length(obj.Device)

    if ~isa(obj.Device{idx}, 'QCASuperCell')
        obj.Device{idx} = obj.Device{idx}.Calc_
           Polarization_Activation(groundstate{
           idx}');

        if(isa(obj.Device{idx},'SixDotCell'))
            NewMobileCharges(:,idx) = obj.
               Device{idx}.getMobileCharge(time
               );
        else
            NewMobileCharges(1:3,idx) = obj.
               Device{idx}.getMobileCharge(time
               );
        end

    else
        NewMobileCharges(:,idx) =
           [0;0;0;0;0;0];

    end


end
```

```
        %%%%%%%%%%%%%%%%%%%%SUBNODE%%%%%%%%%%%%%%%%%%%%
            %%%%%%%%%%
        %or here
        %%%%%%%%%%%%%%%%%%%%SUBNODE%%%%%%%%%%%%%%%%%%%%
            %%%%%%%%%%









        deltaMobileCharges = OldMobileCharges -
            NewMobileCharges;

        [convergance Tolerance, cellindex] = max(max(abs
            (deltaMobileCharges)));
        %                   disp(num2str(
            convergance Tolerance));

        it=it+1;









end
```

```
%              new = obj;
%              new_energy = sum(new.calculateEnergy(time));
%              if(new_energy > old_energy)
%                      disp(['**********************************
   ****************************************: ' num2str(new_
   energy-old_energy) ' : ' num2str(converganceTolerance)])
%              end
             %disp(['-------------it:', num2str(it)]);




        end



        %%
        function obj = Relax2GroundState_serial(obj, time,
           varargin)

            %Iterate to Self consistency
            inverseflag = 0;
            if length(varargin) == 1
                if varargin{1} == 'inverse'
                    inverseflag = 1;
```

```
        end
        disp(num2str(varargin{1}))
    end

    NewCircuitPols = ones(1,length(obj.Device));
    convergance Tolerance = 1;
    sub = 1;
    chi = 0.4;
    it=1;
    while (convergance Tolerance > 0.1)
        OldCircuitPols = NewCircuitPols;

        %get all hamiltonians
        for idx = 1:length(obj.Device)
            nl = obj.Device{idx}.NeighborList;
            nl_obj = obj.getCellArray(nl);

            hamiltonian = obj.Device{idx}.
               GetHamiltonian(nl_obj,time);
            obj.Device{idx}.Hamiltonian = hamiltonian;
        end

        %get all the groundstates
        for idx = 1:length(obj.Device)
            hamiltonian = obj.Device{idx}.Hamiltonian;
            wavefunction = obj.Device{idx}.Wavefunction
               ;
```

```
        if inverseflag
            newpsi = invitr(hamiltonian,0.001,10);
        else
            [V, EE] = eig(hamiltonian);
            newpsi = V(:,1);
        end


        normpsi = (1-chi)*wavefunction + chi*newpsi
           ;
        normpsi = normalize_psi_1D(normpsi');
        groundstate{idx} = normpsi;
    end

    %update pols and acts
    for idx = 1:length(obj.Device)
        obj.Device{idx} = obj.Device{idx}.Calc_
           Polarization_Activation(groundstate{idx
           }');
        NewCircuitPols(idx) = obj.Device{idx}.
           getPolarization(time);
    end


    deltaCircuitPols = abs(OldCircuitPols) - abs(
       NewCircuitPols);
    [converganceTolerance maxid] = max(abs(
       deltaCircuitPols));
```

```
            sub=sub+1;
            it=it+1;

            disp(['---- Iteration: ', num2str(it), ' -
               MaxID: ', num2str(maxid), ' -
               convergenceTolerance: ', num2str(
               convergenceTolerance), ' ----']);
            if (it == 500)
                break
            end
        end

    end
    %%
    function obj = pipeline(obj, clockSignalsList, varargin
       )



        home = pwd;

        %default values
        file = 'simResults.mat';
        nt=315;
        inputSignalsList = {};
        numOfPeriods = 1;
        parallelFlag = 0;
```

```
%optional changes
args = varargin(1:end);
while length(args) >= 2
    prop = args{1};
    val = args{2};
    args = args(3:end);
    switch prop
        case 'Filename'
            file = val;

        case 'TimeSteps'
            if ischar(val)
                nt = str2num(val);
            else
                nt = val;
            end

        case 'inputSignalsList'
            inputSignalsList = val;

        case 'numOfPeriods'
            numOfPeriods = val;

        case 'randomizedRelaxation'
            randomizedFlag = val;
```

```
            if randomizedFlag == 1

                disp('USING RANDOMIZED RELAXATION
                   ORDER')

            end
        case 'mobileCharge'
            mobileChargeFlag = val;

        case 'Parallel'
            parallelFlag = val;

            if parallelFlag == 1
                %disp('parpool')
                maxThreads = min([feature('NumCores
                   '),4 ]);
                maxNumCompThreads(maxThreads);
                delete(gcp('nocreate'));
                parpool('local',maxThreads);
                %parallel.pool.Constant

            end


        otherwise
            error(['QCACircuit.pipeline ', prop, '
               is an invalid property specifier.'])
               ;
```

```
    end
end



obj.Simulating = 'on';




% find the longest period, create time steps

maxPeriod = clockSignalsList{1}.Period;
for signalidx = 1:length(clockSignalsList)
    if(clockSignalsList{signalidx}.Period >
       maxPeriod )
        maxPeriod = clockSignalsList{signalidx}.
           Period;
    end
end

tperiod = maxPeriod*numOfPeriods; %numOfPeriods;
time_array = linspace(0,tperiod,nt);
tc = mod(time_array, tperiod);




m = matfile(file, 'Writable', true);
```

```
save(file, 'clockSignalsList', '-v7.3');
save(file, 'inputSignalsList', '-append');
save(file, 'numOfPeriods', '-append');
save(file, 'obj', '-append');

m.pols = [];%zeros(nt,length(obj.Device));
m.acts = [];%zeros(nt,length(obj.Device));
m.efields = {};%zeros(nt,length(obj.Device));
m.nt = nt;

pols = [];
acts = [];
efields = {};



for t = 1:nt %time step

    percentage = t/nt;
    %waitbar(percentage); annoying right now...
    disp(['t: ', num2str(t)]);

    obj = obj.UpdateClockFields(tc(t),
       clockSignalsList, inputSignalsList);



    %relax2Groundstate
```

```
if parallelFlag == 0 && mobileChargeFlag == 0
    obj = obj.Relax2GroundState(tc(t)); %tp(t)
elseif parallelFlag == 0 && mobileChargeFlag ==
    1
    %obj = obj.Relax2GroundState_randomized(tc(
       t)); %tp(t)
    if randomizedFlag == 1 && t ~= 1
        obj = obj.Relax2GroundState_
           mobilecharge_serial(tc(t), '
           randomizedRelaxation', 1); %tp(t)
    else

        obj = obj.Relax2GroundState_
           mobilecharge_serial(tc(t), '
           randomizedRelaxation', 0); %tp(t)
        %obj = obj.Relax2GroundState_
           mobilecharge(tc(t)); %tp(t)




    end
elseif parallelFlag == 1
    obj = Relax2GroundState_parallel(obj, tc(t)
       ); %tp(t)
elseif parallelFlag == 2
    obj = obj.Relax2GroundState_serial(tc(t));
       %, 'inverse' tp(t)
end
```

```
%data output
it = 1;
for idx=1:length(obj.Device)

    if isa(obj.Device{idx},'QCASuperCell')

        for sub=1:length(obj.Device{idx}.Device
           )


            pols(t,it) = obj.Device{idx}.Device
               {sub}.getPolarization(tc(t));
            acts(t,it) = obj.Device{idx}.Device
               {sub}.Activation;
            efields{t,it} = obj.Device{idx}.
               Device{sub}.ElectricField;

            it = it + 1;
        end


    else


        pols(t,it) = obj.Device{idx}.
           getPolarization(tc(t));
```

```
                acts(t,it) = obj.Device{idx}.Activation
                    ;
                efields{t,it} = obj.Device{idx}.
                    ElectricField;
                it = it + 1;
            end
        end



end %time step loop
%delete(wb);



m.pols = pols;
m.acts = acts;
m.efields = efields;



disp('Complete!')

obj.Simulating = 'off';



delete(gcp('nocreate')); % only delete parallel
    pool if one was created.
```

```
        cd(home);
    end
    %%
    function obj = UpdateClockFields(obj, time,
       clockSignalList, inputSignalList)

        if( iscell(clockSignalList) && isa(clockSignalList
           {1}, 'Signal') && iscell(inputSignalList) && isa
           (inputSignalList{1}, 'Signal') ) %still need to
           check if all clock signals are a Signal()

            % then assign clock fields
            CircuitIdx = 1;
            while CircuitIdx <= length(obj.Device)
                if( isa(obj.Device{CircuitIdx}, '
                   QCASuperCell') )
                    for subnode = 1:length(obj.Device{
                       CircuitIdx}.Device)
                        efield = obj.Device{CircuitIdx}.
                           Device{subnode}.ElectricField; %
                            for this node, set the z-efield
                            to zero
                        %efield(3) = 0;
                        efield = [efield(1), 0, 0];

                        for signalidx = 1:length(
                           clockSignalList)
```

```
                        %step through each signal and
                            accumulate
                        %the efield
                        efield = efield +
                            clockSignalList{signalidx}.
                            getClockField(obj.Device{
                            CircuitIdx}.Device{subnode}.
                            CenterPosition, mod(time,
                            clockSignalList{signalidx}.
                            Period )); %changes E Field.
                        efield = efield +
                            inputSignalList{signalidx}.
                            getInputField(obj.Device{
                            CircuitIdx}.Device{subnode}.
                            CenterPosition, mod(time,
                            inputSignalList{signalidx}.
                            Period )); %changes E Field.
                        obj.Device{CircuitIdx}.Device{
                            subnode}.ElectricField =
                            efield;

                    end %signalList

                end
                CircuitIdx = CircuitIdx+1;
            else
```

```
                    efield = obj.Device{CircuitIdx}.
                       ElectricField; % for this node, set
                       the z-efield to zero
                    %efield(3) = 0;
                    efield = [efield(1), 0, 0];

                    for signalidx = 1:length(
                       clockSignalList)
                        %step through each signal and
                           accumulate
                        %the efield
                        efield = efield + clockSignalList{
                           signalidx}.getClockField(obj.
                           Device{CircuitIdx}.
                           CenterPosition, mod(time,
                           clockSignalList{signalidx}.
                           Period )); %changes E Field.

                        %if (CircuitIdx < 4)
                            efield = efield +
                               inputSignalList{signalidx}.
                               getInputField(obj.Device{
                               CircuitIdx}.CenterPosition,
                               mod(time, inputSignalList{
                               signalidx}.Period )); %
                               changes E Field.
                        %end
```

```
                        obj.Device{CircuitIdx}.
                            ElectricField = efield;

                    end% signalList

                    CircuitIdx = CircuitIdx+1;
                end

            end




        else
            error('Input must be Cell Array of Signals')
        end % if is a cell array of signals (kinda works as
            a check)

    end

    %%
    function cell_obj = getCellArray(obj, CellIDArray)
        %this function returns an array of QCACell objects
           given a list
        %of IDs

        cell_obj = {};
```

```
        for i=1:length(obj.Device)
            if isa(obj.Device{i},'QCASuperCell')
                for j=1:length(obj.Device{i}.Device)
                    for k=1:length(CellIDArray)
                        if CellIDArray(k) == obj.Device{i}.
                           Device{j}.CellID
                            cell_obj{end+1} = obj.Device{i
                               }.Device{j};
                        end
                    end
                end

            else

                for k=1:length(CellIDArray)
                    if CellIDArray(k) == obj.Device{i}.
                       CellID
                        cell_obj{end+1} = obj.Device{i};
                    end
                end

            end
        end

    end
    %%
    function CellIds = GetCellIDs(obj,cells)
        %returns just the CellIDs given a list of objects.
```

```
        CellIds=[];

        idx = 1;

        while idx <= length(cells)
            if isa(cells{idx}, 'QCASuperCell')

                for sub = 1:length(cells{idx}.Device)
                    CellIds(end+1) = cells{idx}.Device{sub
                        }.CellID;

                end

                idx = idx + 1;
            else
                CellIds(end+1) = cells{idx}.CellID;
                idx = idx + 1;
            end

        end

    end
    %%
    function obj = Snap2Grid(obj)
        coord = {};
```

```
        for i=1:length(obj.Device)  %fill cell with all
           center positions
            if isa(obj.Device{i},'QCASuperCell')


                for j=1:length(obj.Device{i}.Device)
                    coord{end+1} = obj.Device{i}.Device{j}.
                       CenterPosition;

                end


            else
                coord{end+1} = obj.Device{i}.CenterPosition
                   ;
            end

        end


        for i=1:length(coord)


            diffx=coord{i}(1)-floor(coord{i}(1)); %range of
                0 to 1 for rounding to 0, .5, or 1
               relatively speaking
            diffy=coord{i}(2)-floor(coord{i}(2)); %this is
               priming the snap to grid functionality
```

```
        %determining how each x,y will be rounded to
           floor, .5 or
        %up to the next integer
        if diffx<.25
            coord{i}(1)=floor(coord{i}(1));
        end
        if diffx>=.25 && diffx<=.75
            coord{i}(1)=floor(coord{i}(1))+.5;
        end
        if diffx>.75
            coord{i}(1)=floor(coord{i}(1))+1;
        end



        if diffy<.25
            coord{i}(2)=floor(coord{i}(2));
        end
        if diffy>=.25 && diffy<=.75
            coord{i}(2)=floor(coord{i}(2))+.5;
        end
        if diffy>.75
            coord{i}(2)=floor(coord{i}(2))+1;
        end
    end



    it=1;
```

```
            for i=1:length(obj.Device) %replace the cell center
                positions with the new snapped center positions
                if isa(obj.Device{i},'QCASuperCell')
                    for j=1:length(obj.Device{i}.Device)

                        obj.Device{i}.Device{j}.CenterPosition
                           = coord{it};
                        it=it+1;
                    end
                else
                    obj.Device{i}.CenterPosition = coord{it};
                    it = it+1;
                end

            end
        end


    end

end




% format for iterating through circuit
% for i=1:length(obj.Device)
%     if isa(obj.Device{i},'QCASuperCell')
```

```
%           Sel=0;
%           for j=1:length(obj.Device{i}.Device)
%               nl = obj.GenerateNeighborList();
%
%               %do a thing
%
%
%           end
%           if Sel
%
%           end
%
%       else
%           %do a thing
%
%       end
% end
```

Listing A.37. QCACircuit Class

```
function QCAHelp()
% Circuit
% The user may add/remove node (red) and driver(green) cells,
   along with selecting multiple
% nodes to create a super cell, identified by the outline of a
   common unique color.  Driver
% cells cannot be in super cells.  Cells can be selected in
   groups or alone, along with
% being dragged and dropped to any position. When dragging and
   dropping, the user can elect
```

```
% to have snap to to grid functionality on or off.  The user
   may also use the arrow keys
% to 'nudge' cell(s) in any direction.
%
%
% Signal
% The user may create a signal which is one of four types:
   sinusoidal, custom,
% fermi, and electrode.  The former 3 will be plotted on the
   signal axes in the signal panel if
% a signal of one of those 3 types is selected in the signal
   list box at the far left of the panel.
% Editing a signal consists of selecting it from the list, thus
    opening up the signal
% and putting all the relevant information into the boxes for
   the
% corresponding signal type.  The signal's properties can be
   changed,
% including but not limited to (if applicable) its type,
   wavelength,
% amplitude, etc.
%
%
% Hot Keys
% Ctrl+h = align selected cells horizontally
% Ctrl+u = align selected cells vertically
% Ctrl+m = make super cell
```

```
% Ctrl+l = disband supercell (click any member of the super
   cell then ctrl+l)
% Ctrl+f = add a node cell (red)
% Ctrl+d = add a driver cell (green)
% Ctrl+b = rectangle select (see mouse functionality)
% Ctrl+back/del = delete selected cell(s)
% Ctrl+g = snap to grid
% Ctrl+e = reset cells to P=0,A=1
% Ctrl+t = refresh (redraw)
% Ctrl+a = select all cells
% Ctrl+q = deselect all cells (or simply use the Shift button
   alone)
% Ctrl+n = New circuit (clears the axes and signals list)
% Ctrl+o = Open a previously saved circuit
% Ctrl+s = Save circuit
% Ctrl+. = make selected driver(s) P=1
% Ctrl+, = make selected driver(s) P=-1
%
%
% Click Functionality
% Left click allows the user to select any button or cell
% Right click opens a context menu with known functionality
% Scroll (middle) click allows the user to use rectangle select
    (the arrow will become a cross)
%
%
% Simulations
```

```
% In order to Simulate, the user must first create a circuit
   and a signal.
% Once both are created, the user can select the Simulation
   button in the Simulation
% panel.  There is an option to name the simulation below that
   button, but if no
% name is entered, the simulation will automatically be named '
   simResults'.  Upon
% completion of that simulation, the user can elect to
   visualize that simulation by
% pressing the Visualize Simulation button.  Then a .mp4 file
   will be created that the user
% can view upon completion of the visualization function.

Circuit = 'The user may add/remove node (red) and driver(green)
    cells, along with selecting multiple nodes to create a
   super cell, identified by the outline of a common unique
   color.  Driver cells cannot be in super cells.  Cells can be
    selected in groups or alone, along with being dragged and
   dropped to any position. When dragging and dropping, the
   user can electto have snap to to grid functionality on or
   off.  The user may also use the arrow keys to ''nudge'' cell
   (s) in any direction.';

Signal  = 'The user may create a signal which is one of four
   types: sinusoidal, custom, fermi, and electrode.  The former
    3 will be plotted on the signal axes in the signal panel if
    a signal of one of those 3 types is selected in the signal
```

```
   list box at the far left of the panel. Editing a signal
   consists of selecting it from the list, thus opening up the
   signal and putting all the relevant information into the
   boxes for the corresponding signal type.  The signal''s
   properties can be changed, including but not limited to (if
   applicable) its type, wavelength, amplitude, etc.';

h='Ctrl+h = align selected cells horizontally';
u='Ctrl+u = align selected cells vertically';
m='Ctrl+m = make super cell';
l='Ctrl+l = disband supercell (click any member of the super
   cell then ctrl+l)';
f='Ctrl+f = add a node cell (red)';
d='Ctrl+d = add a driver cell (green)';
b='Ctrl+b = rectangle select (see mouse functionality)';
back='Ctrl+back/del = delete selected cell(s)';
g='Ctrl+g = snap to grid';
e='Ctrl+e = reset cells to P=0,A=1';
t='Ctrl+t = refresh (redraw)';
a='Ctrl+a = select all cells';
q='Ctrl+q = deselect all cells (or simply use the Shift button
   alone)';
n='Ctrl+n = New circuit (clears the axes and signals list)';
o='Ctrl+o = Open a previously saved circuit';
s='Ctrl+s = Save circuit';
pd='Ctrl+. = make selected driver(s) P=1';
com='Ctrl+, = make selected driver(s) P=-1';
```

```
Click   = 'Left click allows the user to select any button or
   cell\nRight click opens a context menu with known
   functionality\nScroll (middle) click allows the user to use
   rectangle select (the arrow will become a cross)\n';
Sims    = 'In order to Simulate, the user must first create a
   circuit and a signal.  Once both are created, the user can
   select the Simulation button in the Simulation panel.  There
    is an option to name the simulation below that button, but
   if no name is entered, the simulation will automatically be
   named ''simResults''.  Upon completion of that simulation,
   the user can elect to visualize that simulation by pressing
   the Visualize Simulation button.  Then a .mp4 file will be
   created that the user can view upon completion of the
   visualization function.';

msgbox({'Circuit';Circuit;'Signal';'';
    Signal;'Hot Keys';h;u;m;l;f;d;b;back;g;e;t;a;q;n;o;s;pd;com
       ;'';
    'Click Functionality';Click;'';
    'Simulations';Sims;'';},'Help','helpd');

end
```

Listing A.38. QCAHelp Function

```
function QCALayoutAddDriver()
%This allows the user to add a driver.  There is a similar
   button for adding
%a node.
```

```
myCircuit = getappdata(gcf, 'myCircuit');
nodeType = getappdata(gcf, 'nodeType');

% xloc=[];



if(isempty(myCircuit.Device))
    newXlocation = 0;
    newYlocation = 0;
else
    xs=[];
    ys=[];

    for i=1:length(myCircuit.Device)
        if isa(myCircuit.Device{i},'QCASuperCell')
            for j=1:length(myCircuit.Device{i}.Device)

                xs(end+1)=myCircuit.Device{i}.Device{j}.
                   CenterPosition(1);
                ys(end+1)=myCircuit.Device{i}.Device{j}.
                   CenterPosition(2);
            end
        else
            xs(end+1)=myCircuit.Device{i}.CenterPosition(1);
            ys(end+1)=myCircuit.Device{i}.CenterPosition(2);
        end
```

```
    end




end


%create node
switch nodeType
     case 'Three Dot Node'
        if(isempty(myCircuit.Device))
            newXlocation = 0;
            newYlocation = 0;
        else
            newXlocation = max(xs)+1;
            newYlocation = min(ys);
        end

        node = ThreeDotCell([newXlocation newYlocation 0]);
        node.CenterPosition = [newXlocation newYlocation 0];

    case 'Six Dot Node'
        if(isempty(myCircuit.Device))
            newXlocation = 0;
            newYlocation = 0;
        else
            newXlocation = max(xs)+2;
            newYlocation = min(ys);
```

```
        end

        node = SixDotCell([newXlocation newYlocation 0]);
        node.CenterPosition = [newXlocation newYlocation 0];

    otherwise
        error('Node Type was set to an Invalid Type');
end

node.Type = 'Driver';
node.Polarization = 1;

% add node to circuit
myCircuit = myCircuit.addNode(node);



% circuitDraw
mode = myCircuit.Mode;


myCircuit = myCircuit.CircuitDraw(0,gca);
```

```
setappdata(gcf,'myCircuit',myCircuit);




%axis tight
axis equal
end
```

Listing A.39. QCALayoutAddDriver Function

```
function QCALayoutAddNode()
%This allows the user to add a node.  There is a similar button
    for adding
%a driver.  See also the Add 5 Nodes button.

myCircuit = getappdata(gcf, 'myCircuit');
nodeType = getappdata(gcf, 'nodeType');

% xloc=[];



if(isempty(myCircuit.Device))
    newXlocation = 0;
    newYlocation = 0;
else
    xs=[];
    ys=[];
    for i=1:length(myCircuit.Device)
```

```
        if isa(myCircuit.Device{i},'QCASuperCell')
            for j=1:length(myCircuit.Device{i}.Device)

                xs(end+1)=myCircuit.Device{i}.Device{j}.
                   CenterPosition(1);
                ys(end+1)=myCircuit.Device{i}.Device{j}.
                   CenterPosition(2);
            end
        else
            xs(end+1)=myCircuit.Device{i}.CenterPosition(1);
            ys(end+1)=myCircuit.Device{i}.CenterPosition(2);
        end


    end







end

%create node
switch nodeType
     case 'Three Dot Node'

        if(isempty(myCircuit.Device))
            newXlocation = 0;
```

```
            newYlocation = 0;
        else
            newXlocation = max(xs)+1;
            newYlocation = min(ys);
        end

        node = ThreeDotCell([newXlocation newYlocation 0]);
        node.CenterPosition = [newXlocation newYlocation 0];

    case 'Six Dot Node'
        if(isempty(myCircuit.Device))
            newXlocation = 0;
            newYlocation = 0;
        else
            newXlocation = max(xs)+2;
            newYlocation = min(ys);
        end

        node = SixDotCell([newXlocation newYlocation 0]);
        node.CenterPosition = [newXlocation newYlocation 0];

    otherwise
        error('Node Type was set to an Invalid Type');
end

% add node to circuit
myCircuit = myCircuit.addNode(node);
```

```
myCircuit = myCircuit.CircuitDraw(0,gca);
%          handles.layoutchange.Value=0;



setappdata(gcf,'myCircuit',myCircuit);



%axis tight
axis equal
```

Listing A.40. QCALayoutAddNode Function

```
function varargout = QCALayoutGUI(varargin)
% QCALAYOUTGUI MATLAB code for QCALayoutGUI.fig
%      QCALAYOUTGUI, by itself, creates a new QCALAYOUTGUI or
   raises the existing
%      singleton*.
%
%      H = QCALAYOUTGUI returns the handle to a new
   QCALAYOUTGUI or the handle to
%      the existing singleton*.
%
%      QCALAYOUTGUI('CALLBACK',hObject,eventData,handlesButton
   ,...) calls the local
```

```
%      function named CALLBACK in QCALAYOUTGUI.M with the given
    input arguments.
%
%      QCALAYOUTGUI('Property','Value',...) creates a new
   QCALAYOUTGUI or raises the
%      existing singleton*.  Starting from the left, property
   value pairs are
%      applied to the GUI before QCALayoutGUI_OpeningFcn gets
   called.  An
%      unrecognized property name or invalid value makes
   property application
%      stop.  All inputs are passed to QCALayoutGUI_OpeningFcn
   via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI
   allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help
   QCALayoutGUI

% Last Modified by GUIDE v2.5 27-Aug-2019 14:39:38

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
```

```
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @QCALayoutGUI_OpeningFcn,
                      ...
                   'gui_OutputFcn',  @QCALayoutGUI_OutputFcn,
                      ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin
       {:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT



% --- Executes just before QCALayoutGUI is made visible.
function QCALayoutGUI_OpeningFcn(hObject, eventdata, handles,
   varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of
   MATLAB
```

```
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)
% varargin   command line arguments to QCALayoutGUI (see
   VARARGIN)

% Choose default command line output for QCALayoutGUI
handles.output = hObject;

% Update handlesButton structure
guidata(hObject, handles)

% set(handlesButton.figure1,'Name','QCA Layout Demo');
axes(handles.MainAxes);

myCircuit = QCACircuit();
myCircuit.CircuitDraw(gca);
RightClickThings();
nodeTypepopupmenu_Callback(hObject, eventdata, handles);



clockSignalsList = {};
inputSignalsList = {};



setappdata(gcf,'clockSignalsList',clockSignalsList);
setappdata(gcf,'inputSignalsList',inputSignalsList);

setappdata(gcf, 'myCircuit', myCircuit);
```

```
Path.home = pwd;

Path.circ = './Circuits folder';

setappdata(gcf,'Path',Path);




% UIWAIT makes QCALayoutGUI wait for user response (see
    UIRESUME)
% uiwait(handlesButton.figure1);


% --- Outputs from this function are returned to the command
    line.
function varargout = QCALayoutGUI_OutputFcn(hObject, eventdata,
     handles)
% varargout  cell array for returning output args (see
    VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handlesButton    structure with handlesButton and user data (
    see GUIDATA)

% Get default command line output from handlesButton structure
```

```
varargout{1} = handles.output;



% --- Executes on button press in addNodeButton.
function addNodeButton_Callback(hObject, eventdata, handles)
% hObject    handle to addNodeButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% add a new node
QCALayoutAddNode();




% --- Executes when user attempts to close figure1.
function figure1_CloseRequestFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hint: delete(hObject) closes the figure

delete(hObject);
```

```
% --------------------------------------------
function FileMenus_Callback(hObject, eventdata, handles)
% hObject    handle to FileMenus (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
% --------------------------------------------


function SaveMenu_Callback(hObject, eventdata, handles)
% hObject    handle to SaveMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%saving both the circuit and the signal to a .mat file
SaveCircuit();

% --------------------------------------------
function OpenMenu_Callback(hObject, eventdata, handles)
% hObject    handle to OpenMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)
```

```
%open a previously made circuit
LoadCircuit(handles);



% --------------------------------------------
function NewMenu_Callback(hObject, eventdata, handles)
% hObject    handle to NewMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%get rid of the current circuit and clear its data from the
   appdata
NewCircuit(handles);




function chngPol_Callback(hObject, eventdata, handles)
% hObject    handle to chngPol (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of chngPol as
   text
%        str2double(get(hObject,'String')) returns contents of
   chngPol as a double

%change the polarization of any driver cell(s)
ChangePol(handles);



% --- Executes during object creation, after setting all
   properties.
function chngPol_CreateFcn(hObject, eventdata, handles)
% hObject    handle to chngPol (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on button press in makeSC.
function makeSC_Callback(hObject, eventdata, handles)
% hObject    handle to makeSC (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of makeSC

%make the group of selected cells a supercell. Drivers cannot
   be in
%supercells
MakeSuperCellGUI();



% --- Executes on button press in disbandsupercell.
function disbandsupercell_Callback(hObject, eventdata, handles)
% hObject    handle to disbandsupercell (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%if any member of a supercell is selected, the SC can be
   disbanded
DisbandSuperCell();
```

```
%may be depreciated
% --- Executes on button press in simbutton.
function simbutton_Callback(hObject, eventdata, handles)
% hObject    handle to simbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%currently the simulate button, but it may be deleted later
   since it it not
%the same as the simulation panel functionality
Simulate(handles);



% --- Executes on button press in removeNode.
function removeNode_Callback(hObject, eventdata, handles)
% hObject    handle to removeNode (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%Remove the node or nodes selected
RemoveNode();
```

```
% --- Executes on button press in addDriver.
function addDriver_Callback(hObject, eventdata, handles)
% hObject    handle to addDriver (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%add a driver to the circuit and axis
QCALayoutAddDriver();



% --- Executes on button press in add5Cells.
function add5Cells_Callback(hObject, eventdata, handles)
% hObject    handle to add5Cells (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)
% Add5Cells(handlesButton);

%add 5 nodes instead of just 1 (we could desire certain shapes
   of
%additions, something to think about)
for i=1:5
    QCALayoutAddNode();
end
```

```
% --- Executes on button press in resetButton.
function resetButton_Callback(hObject, eventdata, handles)
% hObject    handle to resetButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%set all nodes to a polarization of 0
ResetCells();



%OBSOLETE
function chngClock_Callback(hObject, eventdata, handles)
% hObject    handle to chngClock (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of chngClock as
    text
%        str2double(get(hObject,'String')) returns contents of
   chngClock as a double
ChangeClockField(handles);
```

```
% --- Executes during object creation, after setting all
   properties.
function chngClock_CreateFcn(hObject, eventdata, handles)
% hObject    handle to chngClock (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




% --- Executes on button press in autoSnap.
function autoSnap_Callback(hObject, eventdata, handles)
% hObject    handle to autoSnap (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)
```

```
% Hint: get(hObject,'Value') returns toggle state of autoSnap

%this is the switch function for turning snap2grid
   functionality on and off
AutoSnap(handles);



% --- Executes on button press in refresh.
function refresh_Callback(hObject, eventdata, handles)
% hObject    handle to refresh (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%for debugging, we redraw the circuit
myCircuit = getappdata(gcf,'myCircuit');

% f=gcf;
% f.Pointer = 'arrow';

myCircuit = myCircuit.CircuitDraw(0,gca);

setappdata(gcf,'myCircuit',myCircuit);
```

```
% --- Executes on button press in circuitPanel.
function circuitPanel_Callback(hObject, eventdata, handles)
% hObject    handle to circuitPanel (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
   circuitPanel
% PanelSwitch(handlesButton);

%switching between the three main panels (circuit, signal,
   simulation)
%MajorPanelSwitch(handles,'circuit');



% --- Executes on button press in signalPanel.
function signalPanel_Callback(hObject, eventdata, handles)
% hObject    handle to signalPanel (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
   signalPanel
```

```
% PanelSwitch(handlesButton);

%switching between the three main panels (circuit, signal,
   simulation)
%MajorPanelSwitch(handles,'signal');




% --- Executes on button press in simulatePanel.
function simulatePanel_Callback(hObject, eventdata, handles)
% hObject    handle to simulatePanel (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
   simulatePanel

%switching between the three main panels (circuit, signal,
   simulation)
%MajorPanelSwitch(handles,'simulate');




function changeWave_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to changeWave (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of changeWave
   as text
%        str2double(get(hObject,'String')) returns contents of
   changeWave as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeWave_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changeWave (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function changeAmp_Callback(hObject, eventdata, handles)
% hObject    handle to changeAmp (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of changeAmp as
    text
%        str2double(get(hObject,'String')) returns contents of
   changeAmp as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeAmp_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changeAmp (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end



function changePeriod_Callback(hObject, eventdata, handles)
% hObject    handle to changePeriod (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of changePeriod
    as text
%        str2double(get(hObject,'String')) returns contents of
   changePeriod as a double
RePlotSignal(handles)

function changePhase_Callback(hObject, eventdata, handles)
% hObject    handle to changePhase (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of changePhase
   as text
```

```
%        str2double(get(hObject,'String')) returns contents of
   changePhase as a double
RePlotSignal(handles)



% --- Executes during object creation, after setting all
   properties.
function changePhase_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changePhase (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function changeMeanValue_Callback(hObject, eventdata, handles)
% hObject    handle to changeMeanValue (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of
   changeMeanValue as text
%        str2double(get(hObject,'String')) returns contents of
   changeMeanValue as a double


% --- Executes during object creation, after setting all
   properties.
function changeMeanValue_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changeMeanValue (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on selection change in signalType.
function signalType_Callback(hObject, eventdata, handles)
% hObject    handle to signalType (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
   signalType contents as cell array
%        contents{get(hObject,'Value')} returns selected item
   from signalType

%switch between the different signal type panels within the
   major signal
%panel

signalTypes = cellstr(get(handles.signalType,'String')); %get
   the list of signal types from handles

sigType = signalTypes{get(handles.signalType,'Value')}; %find
   which one is selected

SignalTypePanelSwitch(handles, sigType);

if ~isempty(handles.signalEditor.String) %changing the signal
   type and replotting if a signal is being edited
    clockSignalsList = getappdata(gcf,'clockSignalsList');

    for i=1:length(clockSignalsList)
```

```
        if strcmp(clockSignalsList{i}.Name,handles.signalEditor
           .String)
            clockSignalsList{i}.Type = sigType;
        end
    end
    setappdata(gcf,'clockSignalsList',clockSignalsList);

    RePlotSignal(handles);
end




% --- Executes during object creation, after setting all
   properties.
function signalType_CreateFcn(hObject, eventdata, handles)
% hObject    handle to signalType (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: popupmenu controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end



% --- Executes during object creation, after setting all
   properties.
function changePeriod_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changePeriod (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




% --- Executes on button press in createSignal.
function createSignal_Callback(hObject, eventdata, handles)
% hObject    handle to createSignal (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%create a signal of a designated type
CreateSignal(handles, 'clockSignal');



function signalName_Callback(hObject, eventdata, handles)
% hObject    handle to signalName (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of signalName
   as text
%        str2double(get(hObject,'String')) returns contents of
   signalName as a double



% --- Executes during object creation, after setting all
   properties.
function signalName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to signalName (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on selection change in signalList.
%THIS IS THE MENU LIST THAT HOLDS ALL THE SIGNALS
function signalList_Callback(hObject, eventdata, handles)
% hObject    handle to signalList (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
   signalList contents as cell array
%        contents{get(hObject,'Value')} returns selected item
   from signalList
```

```
%attain signal properties by selecting that signal in the gui
   listbox

GetSignalPropsGUI(handles);



% --- Executes during object creation, after setting all
   properties.
function signalList_CreateFcn(hObject, eventdata, handles)
% hObject    handle to signalList (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: listbox controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in deleteSignal.
function deleteSignal_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to deleteSignal (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%Delete a signal from the gui listbox and from the appdata
DeleteSignal(handles);



% --- Executes on button press in saveSignal.
function saveSignal_Callback(hObject, eventdata, handles)
% hObject    handle to saveSignal (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%save the current signal that is being edited
SaveEditedSignal(handles);



% --- Executes on selection change in transitionType.
function transitionType_Callback(hObject, eventdata, handles)
% hObject    handle to transitionType (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
   transitionType contents as cell array
%        contents{get(hObject,'Value')} returns selected item
   from transitionType




% --- Executes during object creation, after setting all
   properties.
function transitionType_CreateFcn(hObject, eventdata, handles)
% hObject    handle to transitionType (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called

% Hint: popupmenu controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end



function changeInputField_Callback(hObject, eventdata, handles)
% hObject    handle to changeInputField (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changeInputField as text
%        str2double(get(hObject,'String')) returns contents of
   changeInputField as a double
ChangeInputField(handles)




% --- Executes during object creation, after setting all
   properties.
function changeInputField_CreateFcn(hObject, eventdata, handles
   )
% hObject    handle to changeInputField (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    empty - handlesButton not created until
   after all CreateFcns called
```

```
% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on key press with focus on figure1 or any of its
    controls.
function figure1_WindowKeyPressFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  structure with the following fields (see MATLAB.UI
   .FIGURE)
% Key: name of the key that was pressed, in lower case
% Character: character interpretation of the key(s) that was
   pressed
% Modifier: name(s) of the modifier key(s) (i.e., control,
   shift) pressed
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%all the hotkeys are here
eventdata.Key;
eventdata.Modifier;

HotKeysFuncList(handles,eventdata);
```

```
function figure1_WindowButtonDownFcn(hObject, eventdata,
   handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

f=gcf;
f.SelectionType;

Select = f.SelectionType;

ClickFunctionality(handles,eventdata,Select);



% --- Executes on button press in drawElectrode.
function drawElectrode_Callback(hObject, eventdata, handles)
% hObject    handle to drawElectrode (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%drawing the electrodes
ElectrodeDrawer(handles);
```

```
% --- Executes on button press in eraseElectrodes.
function eraseElectrodes_Callback(hObject, eventdata, handles)
% hObject    handle to eraseElectrodes (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%erasing the selected electrodes
ElectrodeEraser(handles);



% --- Executes on button press in clearAll.
function clearAll_Callback(hObject, eventdata, handles)
% hObject    handle to clearAll (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%a dialog box opens, if the use selects yes then the figure
   gets cleared
choosedialog(handles)
```

```
% --- Executes on button press in createSimulation.
function createSimulation_Callback(hObject, eventdata, handles)
% hObject    handle to createSimulation (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%creating the simulation file that we will make the video from
% f=gcf;
% f.Pointer = 'watch';
CreateSimulationFile(handles);
% f.Pointer = 'arrow';



% --- Executes on button press in visualizeSim.
function visualizeSim_Callback(hObject, eventdata, handles)
% hObject    handle to visualizeSim (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%calling the pipelinevisualization function to create the mp4
   video file
% f=gcf;
% f.Pointer = 'watch';
```

```
GenerateSimulationVideo(handles);
% f.Pointer = 'arrow';

function nameSim_Callback(hObject, eventdata, handles)
% hObject    handle to nameSim (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of nameSim as
   text
%        str2double(get(hObject,'String')) returns contents of
   nameSim as a double


% --- Executes during object creation, after setting all
   properties.
function nameSim_CreateFcn(hObject, eventdata, handles)
% hObject    handle to nameSim (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




% --- Executes on button press in handlesButton.
function handlesButton_Callback(hObject, eventdata, handles)
% hObject    handle to handlesButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handlesButton    structure with handlesButton and user data (
    see GUIDATA)

%attaining the handles, not a necessary function and it will be
    gone
%eventually
handles
gca;



%WILL DELETE LATER. IT JUST MAKES IT EASIER TO GET DATA
% --- Executes on button press in getAppInfo.
function getAppInfo_Callback(hObject, eventdata, handles)
% hObject    handle to getAppInfo (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handlesButton    structure with handlesButton and user data (
   see GUIDATA)

%as said above, only getting the app data for debugging
   purposes
AppInfo = getappdata(gcf)
circuit = AppInfo.myCircuit
signals = AppInfo.clockSignalsList



function changeWaveFermi_Callback(hObject, eventdata, handles)
% hObject    handle to changeWaveFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changeWaveFermi as text
%        str2double(get(hObject,'String')) returns contents of
   changeWaveFermi as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeWaveFermi_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changeWaveFermi (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function changePhaseFermi_Callback(hObject, eventdata, handles)
% hObject    handle to changePhaseFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changePhaseFermi as text
%        str2double(get(hObject,'String')) returns contents of
   changePhaseFermi as a double
RePlotSignal(handles)
```

```
% --- Executes during object creation, after setting all
   properties.
function changePhaseFermi_CreateFcn(hObject, eventdata, handles
   )
% hObject    handle to changePhaseFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function changeAmpFermi_Callback(hObject, eventdata, handles)
% hObject    handle to changeAmpFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changeAmpFermi as text
```

```
%        str2double(get(hObject,'String')) returns contents of
   changeAmpFermi as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeAmpFermi_CreateFcn(hObject, eventdata, handles)
% hObject    handle to changeAmpFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function changePeriodFermi_Callback(hObject, eventdata, handles
   )
% hObject    handle to changePeriodFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changePeriodFermi as text
%        str2double(get(hObject,'String')) returns contents of
   changePeriodFermi as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changePeriodFermi_CreateFcn(hObject, eventdata,
   handles)
% hObject    handle to changePeriodFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function changeMeanValueFermi_Callback(hObject, eventdata,
   handles)
% hObject    handle to changeMeanValueFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changeMeanValueFermi as text
%        str2double(get(hObject,'String')) returns contents of
   changeMeanValueFermi as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeMeanValueFermi_CreateFcn(hObject, eventdata,
   handles)
% hObject    handle to changeMeanValueFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end



function changeSharpnessFermi_Callback(hObject, eventdata,
   handles)
% hObject    handle to changeSharpnessFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   changeSharpnessFermi as text
%        str2double(get(hObject,'String')) returns contents of
   changeSharpnessFermi as a double
RePlotSignal(handles)

% --- Executes during object creation, after setting all
   properties.
function changeSharpnessFermi_CreateFcn(hObject, eventdata,
   handles)
% hObject    handle to changeSharpnessFermi (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called
```

```
% Hint: edit controls usually have a white background on
    Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in helpButton.
function helpButton_Callback(hObject, eventdata, handles)
% hObject    handle to helpButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handles    structure with handles and user data (see GUIDATA)
help QCAHelp
QCAHelp()


% --- Executes on selection change in nodeTypepopupmenu.
function nodeTypepopupmenu_Callback(hObject, eventdata, handles
    )
% hObject    handle to nodeTypepopupmenu (see GCBO)
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: contents = cellstr(get(hObject,'String')) returns
   nodeTypepopupmenu contents as cell array
%        contents{get(hObject,'Value')} returns selected item
   from nodeTypepopupmenu
nodeTypes = cellstr(get(handles.nodeTypepopupmenu,'String')); %
   get the list of signal types from handles
nodeType = nodeTypes{get(handles.nodeTypepopupmenu,'Value')}; %
   find which one is selected

setappdata(gcf,'nodeType', nodeType);



% --- Executes during object creation, after setting all
   properties.
function nodeTypepopupmenu_CreateFcn(hObject, eventdata,
   handles)
% hObject    handle to nodeTypepopupmenu (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: popupmenu controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end



% --- Executes on button press in loadSimResults.
function loadSimResults_Callback(hObject, eventdata, handles)
% hObject    handle to loadSimResults (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
LoadSimResults();



% --- Executes on button press in vizAtCertainTimeButton.
function vizAtCertainTimeButton_Callback(hObject, eventdata,
   handles)
% hObject    handle to vizAtCertainTimeButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
vizAtCertainTimeButton(handles);
DragDrop();

function vizAtCertainTimeEditBox_Callback(hObject, eventdata,
   handles)
% hObject    handle to vizAtCertainTimeEditBox (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of
   vizAtCertainTimeEditBox as text
%        str2double(get(hObject,'String')) returns contents of
   vizAtCertainTimeEditBox as a double



% --- Executes during object creation, after setting all
   properties.
function vizAtCertainTimeEditBox_CreateFcn(hObject, eventdata,
   handles)
% hObject    handle to vizAtCertainTimeEditBox (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function numberOfTimeSteps_Callback(hObject, eventdata, handles
    )
% hObject    handle to numberOfTimeSteps (see GCBO)
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
    numberOfTimeSteps as text
%        str2double(get(hObject,'String')) returns contents of
    numberOfTimeSteps as a double




% --- Executes during object creation, after setting all
    properties.
function numberOfTimeSteps_CreateFcn(hObject, eventdata,
    handles)
% hObject    handle to numberOfTimeSteps (see GCBO)
% eventdata  reserved - to be defined in a future version of
    MATLAB
% handles    empty - handles not created until after all
    CreateFcns called

% Hint: edit controls usually have a white background on
    Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function numberOfPeriods_Callback(hObject, eventdata, handles)
% hObject    handle to numberOfPeriods (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
   numberOfPeriods as text
%        str2double(get(hObject,'String')) returns contents of
   numberOfPeriods as a double


% --- Executes during object creation, after setting all
   properties.
function numberOfPeriods_CreateFcn(hObject, eventdata, handles)
% hObject    handle to numberOfPeriods (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called
```

```
% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in inputSignalList.
function inputSignalList_Callback(hObject, eventdata, handles)
% hObject    handle to inputSignalList (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns
   inputSignalList contents as cell array
%        contents{get(hObject,'Value')} returns selected item
   from inputSignalList


% --- Executes during object creation, after setting all
   properties.
function inputSignalList_CreateFcn(hObject, eventdata, handles)
% hObject    handle to inputSignalList (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
```

```
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: listbox controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in inputSignalButton.
function inputSignalButton_Callback(hObject, eventdata, handles
   )
% hObject    handle to inputSignalButton (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
CreateSignal(handles, 'inputSignal');



% --- Executes on button press in circuit_energy_button.
function circuit_energy_button_Callback(hObject, eventdata,
   handles)
% hObject    handle to circuit_energy_button (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
CalculateEnergyCallback(handles);



% --- Executes on button press in flip_pol_button.
function flip_pol_button_Callback(hObject, eventdata, handles)
% hObject    handle to flip_pol_button (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
FlipPol(handles);



% --- Executes on button press in Relax_button.
function Relax_button_Callback(hObject, eventdata, handles)
% hObject    handle to Relax_button (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
Simulate(handles);









function chngAct_Callback(hObject, eventdata, handles)
% hObject    handle to chngAct (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of chngAct as
   text
%        str2double(get(hObject,'String')) returns contents of
   chngAct as a double



% --- Executes during object creation, after setting all
   properties.
function chngAct_CreateFcn(hObject, eventdata, handles)
% hObject    handle to chngAct (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    empty - handles not created until after all
   CreateFcns called

% Hint: edit controls usually have a white background on
   Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
   defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

Listing A.41. QCALayoutGUI Function

```
classdef QCASuperCell
```

```
%SuperCell Holds a list of cells
%   This is to help with the race condition that happens
   with the
%   Hartree Fock approximation

properties
    Device = {}; %what QCACells are in
    CellID = 0;  %unique ID
    NeighborList = []; %this SuperCell's Neighbors
    BoxColor='';

end

methods
    function obj = SuperCell( varargin ) % constructor
       class

        if nargin > 0
            placeholder = obj.Device;
            obj.Device = {placeholder varargin};

        else
            % Nothing happens

        end

    end
```

```
        function obj = addCell( obj, newcell )
            n_old = length(obj.Device);
            obj.Device{n_old+1} = newcell;

            obj.Device{n_old+1}.CellID = obj.CellID + length(
               obj.Device)/100;

        end

        function obj = CircuitDraw(obj, targetAxes)
            hold on
            for CellIndex = 1:length(obj.Device)
                obj.Device{CellIndex} = obj.Device{CellIndex}.
                   ColorDraw();
            end
            hold off
        end

    end

end
```

Listing A.42. QCASuperCell Class

```
function RePlotSignal(handles)
%This function gets called when the user edits one of the edit
   boxes and hits enter for a
%signal (fermi, sine, or possibly custom).  It will replace the
    old data
```

```
%in the signal with the new input, along with plotting the
   signal in its
%new form.

clockSignalsList = getappdata(gcf,'clockSignalsList');

sigName = handles.signalEditor.String;



if ~isempty(sigName) %check to see if there is a signal being
   edited

    for i=1:length(clockSignalsList)
        clockSignalsList{i}.Name;
        if strcmp(sigName,clockSignalsList{i}.Name)
            mySignal = clockSignalsList{i}; %find that signal
               in the list
            pick = i;
        end
    end

    sigType = mySignal.Type;

    types = cellstr(get(handles.signalType,'String'));
    newSigType = types{get(handles.signalType,'Value')};

    if strcmp(sigType,newSigType)
        mySignal.Type = newSigType;
```

```
end


switch sigType %old info is going into the signal

    case 'Sinusoidal'

        mySignal.Amplitude = str2num(handles.changeAmp.
           String);
        mySignal.Wavelength = str2num(handles.changeWave.
           String);
        mySignal.Period = str2num(handles.changePeriod.
           String);
        mySignal.Phase = str2num(handles.changePhase.String
           );
        mySignal.MeanValue = str2num(handles.
           changeMeanValue.String);



    case 'Custom'
        %still nothin' so far


    case 'Fermi'
        mySignal.Amplitude = str2num(handles.changeAmpFermi
           .String);
```

```
            mySignal.Sharpness = str2num(handles.
               changeSharpnessFermi.String);
            mySignal.Period = str2num(handles.changePeriodFermi
               .String);
            mySignal.Phase = str2num(handles.changePhaseFermi.
               String);
            mySignal.MeanValue = str2num(handles.
               changeMeanValueFermi.String);



        case 'Electrode'
            mySignal.InputField = str2num(handles.
               changeInputField.String);
    end

    clockSignalsList{pick} = mySignal;

    PlotSignal(handles,mySignal); %replotting



end

setappdata(gcf,'clockSignalsList',clockSignalsList);

end
```

Listing A.43. RePlotSignal Function

```
function RectangleSelect()
```

```
% This function allows the user to click and drag on the axis
   to draw a
% rectangle that, if overlapping with any center points of a
   cell, will
% select them and allow them to be dragged and dropped.
myCircuit = getappdata(gcf,'myCircuit');

r = getrect(gca);



for i=1:length(myCircuit.Device)
    if isa(myCircuit.Device{i},'QCASuperCell')
        Sel=0;
        for j=1:length(myCircuit.Device{i}.Device)

            %find out if any of the cells within the SC are
               selected by the
            %box
            if myCircuit.Device{i}.Device{j}.CenterPosition(1)
               >= r(1) && myCircuit.Device{i}.Device{j}.
               CenterPosition(2) >= r(2) && myCircuit.Device{i
               }.Device{j}.CenterPosition(1) <= (r(1)+r(3))  &&
                myCircuit.Device{i}.Device{j}.CenterPosition(2)
                <= (r(2)+r(4))
                Sel = Sel +1;


            end
```

```
        end
        %select all of them
        if Sel
            for j=1:length(myCircuit.Device{i}.Device)

                myCircuit.Device{i}.Device{j}.SelectBox.
                   Selected = 'on';
            end
        end

    else
            if myCircuit.Device{i}.CenterPosition(1) >= r(1) &&
                myCircuit.Device{i}.CenterPosition(2) >= r(2) &
               & myCircuit.Device{i}.CenterPosition(1) <= (r(1)
               +r(3))  && myCircuit.Device{i}.CenterPosition(2)
                <= (r(2)+r(4))
                myCircuit.Device{i}.SelectBox.Selected = 'on';

            end

    end
end

%setappdata then call dragdrop, otherwise clicking will merit
   nothing
setappdata(gcf,'myCircuit',myCircuit);
```

```
DragDrop();
end
```

Listing A.44. RectangleSelect Function

```
function RemoveNode()
%Any and all selected cells will be removed.  If a member of a
   supercell is
%selected then removed, the entire group of cells within that
   supercell
%will be removed as well.

myCircuit = getappdata(gcf,'myCircuit');

newCircuit = QCACircuit;
newCircuit = {};



cells2del=[];%cells we will delete


        for i=1:length(myCircuit.Device)
            if isa (myCircuit.Device{i},'QCASuperCell') %if any
                of the cells in a supercell are selected, the
               whole thing will be deleted
                for j=1:length(myCircuit.Device{i}.Device)
                    if strcmp(myCircuit.Device{i}.Device{j}.
                       SelectBox.Selected,'on')
                        cells2del(end+1) = i;
```

```
            end
        end

    else %any cell can be deleted also
        if strcmp(myCircuit.Device{i}.SelectBox.
           Selected,'on')

            cells2del(end+1) = i;
        end
    end

end

for j=1:length(cells2del) %empty each of the locations
   in the device list
    myCircuit.Device{cells2del(j)}={};
end

for i=1:length(myCircuit.Device)
    if ~isempty(myCircuit.Device{i})
        newCircuit{end+1}=myCircuit.Device{i}; %new
           circuit gets all the remaining cells
    end
end

myCircuit.Device = newCircuit;
```

```
        myCircuit = myCircuit.CircuitDraw(0,gca);




setappdata(gcf,'myCircuit',myCircuit);
end
```

Listing A.45. RemoveNode Function

```
function ResetCells()
%All cells except for drivers will be reset to an activation of
    1 and a
%polarization of 0.

    myCircuit = getappdata(gcf,'myCircuit');

        for i=1:length(myCircuit.Device)
            if isa (myCircuit.Device{i},'QCASuperCell') %if any
                of the cells in a supercell are selected, the
               whole thing will be deleted
                for j=1:length(myCircuit.Device{i}.Device)
                    if strcmp( myCircuit.Device{i}.Device{j}.
                       Type,'Node')

                        myCircuit.Device{i}.Device{j}.
                           Polarization = 0;
                        myCircuit.Device{i}.Device{j}.
                           Activation = 1;
```

```
                        myCircuit.Device{i}.Device{j}.
                            NeighborList=[];
                    end
                end

            else %any cell can be deleted also
                if strcmp( myCircuit.Device{i}.Type,'Node')
                    myCircuit.Device{i}.Polarization = 0;
                    myCircuit.Device{i}.Activation = 1;
                    myCircuit.Device{i}.NeighborList=[];
                end
            end

        end
        myCircuit = myCircuit.CircuitDraw(gca);


        setappdata(gcf,'myCircuit',myCircuit);

end
```

Listing A.46. ResetCells Function

```
function RightClickThings()
%Upon right-clicking, the user will see a uicontext menu where
   he or she
%can selected multiple options, all of which are functions
   within the gui
%for the purpose of increased ease of use.
```

```
%get the current figure and axis
f = gcf;
ax = gca;

%create uicontrol
c = uicontextmenu;

% Assign the uicontextmenu to the plot line
%if the user right-clicks anywhere, he can open the menu
ax.UIContextMenu = c;
f.UIContextMenu = c;

% Create child menu items for the uicontextmenu.
%selecting any menu item calls its respective function
m1 = uimenu(c,'Label','Make Super Cell (Ctrl+S)','Callback',
   @changeThings);
m2 = uimenu(c,'Label','Disband Super Cell (Ctrl+L)','Callback',
   @changeThings);

m4 = uimenu(c,'Label','Align');
m4_1 = uimenu('Parent',m4,'Label','Horizontal (Ctrl+H)','
   Callback',@changeThings);
m4_2 = uimenu('Parent',m4,'Label','Vertical (Ctrl+U)','Callback
   ',@changeThings);
```

```
m5 = uimenu(c,'Label','Box Select (Ctrl+B)','Callback',
   @changeThings);
m6 = uimenu(c,'Label','Remove Node (Del)','Callback',
   @changeThings);

m7 = uimenu(c,'Label','Add');
m7_1 = uimenu(m7,'Label','Driver (Ctrl+D)','Callback',
   @changeThings);
m7_2 = uimenu(m7,'Label','Node (Ctrl+F)','Callback',
   @changeThings);

m8 = uimenu(c,'Label','Copy (Ctrl+C)','Callback',@changeThings)
   ;
m9 = uimenu(c,'Label','Paste (Ctrl+V)','Callback',@changeThings
   );




%all the function calls for the callback upon selecting a menu
   item
    function changeThings(source,callbackdata)
        switch source.Label
            case 'Make Super Cell'
                MakeSuperCellGUI();
            case 'Disband Super Cell'
                DisbandSuperCell();
            case 'Vertical'
```

```
                AlignVert();
            case 'Horizontal'
                AlignHoriz();
            case 'Box Select'
                RectangleSelect();
            case 'Remove Node'
                RemoveNode();
            case 'Driver'
                QCALayoutAddDriver();
            case 'Node'
                QCALayoutAddNode();
            case 'Copy'
                CopyCells();
            case 'Paste'
                PasteCells();
        end
    end
end
```

Listing A.47. RightClickThings Function

```
function SaveCircuit()
% Add capability to save other parameters: clocking field-
   wavelength, and
% period, speed of bit packet (from sinusoid)



Path = getappdata(gcf,'Path');
```

```
Circuit=getappdata(gcf, 'myCircuit'); %attain the circuit we
   want to save
clockSignalsList = getappdata(gcf,'clockSignalsList');

[File , pathname] = uiputfile('*.mat');



if File == 0 %if the user cancels the save operation
    cd(Path.home);

else%they don't cancel the save operation
    cd(pathname)



    save(File,'Circuit','clockSignalsList');

    cd(Path.home);
end



setappdata(gcf,'myCircuit',Circuit);
setappdata(gcf,'clockSignalsList',clockSignalsList);

end
```

Listing A.48. SaveCircuit Function

```
function SaveEditedSignal(handles)
%This is different from the main save function.  Saving a
   signal within the
```

```
%signal panel overwrites the new information into the signal
   that is
%currently selected and under editing.

contents = cellstr(get(handles.signalList,'String')); %get the
   list of signals from handles



clockSignalsList = getappdata(gcf,'clockSignalsList'); %
   retrieve the signals from appdata
myCircuit = getappdata(gcf,'myCircuit');

%find which signal to save
if ~isempty(contents)

    spot = get(handles.signalList,'Value');
    sigName = contents{spot}; %this is the signal we selected



    for i=1:length(clockSignalsList)
        if strcmp(sigName,clockSignalsList{i}.Name)
            mySignal = clockSignalsList{i};  %this is the
               signal with the same name (they are the same)
            pick = i;
        end
    end

    if ~isempty(mySignal)
```

```
sigType = mySignal.Type;

handles.signalEditor.String = ''; %clear the gui boxes
   to indicate editing is over
handles.signalEditType.String = '';

%edit data based on the signal type
switch sigType

    case 'Sinusoidal'

        mySignal.Amplitude = str2num(handles.changeAmp.
           String);
        mySignal.Wavelength = str2num(handles.
           changeWave.String);
        mySignal.Period = str2num(handles.changePeriod.
           String);
        mySignal.Phase = str2num(handles.changePhase.
           String);
        %mySignal.MeanValue = str2num(handles.changeMV.
           String);



    case 'Custom'

    case 'Fermi'
```

```
            mySignal.Amplitude = str2num(handles.
               changeAmpFermi.String);
            mySignal.Sharpness = str2num(handles.
               changeSharpnessFermi.String);
            mySignal.Period = str2num(handles.
               changePeriodFermi.String);
            mySignal.Phase = str2num(handles.
               changePhaseFermi.String);
            mySignal.MeanValue = str2num(handles.
               changeMeanValueFermi.String);


        case 'Electrode'
            mySignal.InputField = str2num(handles.
               changeInputField.String);
    end

    types = cellstr(get(handles.signalType,'String'));
    newSigType = types{get(handles.signalType,'Value')};
    mySignal.Type = newSigType;


    mySignal.Name = handles.signalName.String;

    contents{spot} = mySignal.Name;
    handles.signalList.String = contents;
```

```
        clockSignalsList{pick} = mySignal;%replace the signal
           with the edited one



        setappdata(gcf,'clockSignalsList',clockSignalsList);
        myCircuit = myCircuit.CircuitDraw(gca);
        setappdata(gcf,'myCircuit',myCircuit);

        plot(handles.plotAxes,0,0); %clear the plot axis



        handles.signalName.String = 'Input Name';
    end
end

end
```

Listing A.49. SaveEditedSignal Function

```
function Select(p)
% Select functionality, then once that object is selected in
   the gui,
% it can be dragged and dropped.  Once deselected, it cannot be
% dragged and dropped until it is selected again.


%assigning the callback function, the deselect function will
   get its
%callback in the sel function
```

```
p.ButtonDownFcn=@selObject;


    function selObject(hObject, eventdata)

        if strcmp(p.Selected,'off')%check if it's not selected
            if ~isempty(p.ButtonDownFcn)
                p.Selected='on';%select it
                p.ButtonDownFcn=@deSelObject;


                DragDrop();

            end

        end
    end



    function deSelObject(hObject,eventdata)%using callback to
       deselect

        if strcmp(p.Selected,'on')%same logic as before,
           checking
            if ~isempty(p.ButtonDownFcn)%the button and
               condition
```

```
                p.Selected='off';

                p.ButtonDownFcn=@selObject;


                %                   setappdata(gcf,'myCircuit',
                   myCircuit);
            end
        end


    end

end
```

Listing A.50. Select Function

```
classdef Signal
    %Signal class used to input in to various systems
    %   Detailed explanation goes here
    %   Actually add a explaination of properties
    %   One use of this class might be a signal generator for
       an electric
    %   field of a circuit


    properties

        Name;
```

```
    Type = 'Sinusoidal';%  'Fermi' 'Custom'(Piecewise) '
       Imported'(COMSOL) there may be others

    %These properties are only used for the Sinusoidal and
       Fermi type
    Amplitude = 1;
    Wavelength = 1;
    Period = 1;
    Phase = pi/2;
    MeanValue = 0;
    Sharpness = .05;

    CellIds = []; %used to associate certain signals with
       certain cells. Most commonly driver polarizations



    %Electrode Properties
    InputField=0;
    CenterPosition = [0 0 0];
    Height = 1.5;
    Width = .5;
    IsDrawn = 'off';
    TopPatch;
    BottomPatch;


end
```

```
methods
    function obj = Signal( varargin )
        % obj = QCACell( varargin )


        if( strcmp(obj.Type, 'Sinusoidal') )

            switch nargin
                case 0

                case 1 % S = Signal( Amplitude )
                    if( isnumeric(varargin{1}) )
                        obj.Amplitude = varargin{1};
                    else
                        error('Incorrect data input type.')
                    end

                case 2 % S = Signal( Amplitude, wavelength
                   )
                    if( isnumeric(varargin{1}) && isnumeric
                       (varargin{2}) )
                        obj.Amplitude = varargin{1};
                        obj.Wavelength = varargin{2};
                    elseif (~isnumeric(varargin{1}))
```

```
                error('Incorrect data input type
                    for Amplitude.')
            elseif(~isnumeric(varargin{2}))
                error('Incorrect data input type
                    for Wavelength.')
            else
                error('Incorrect data input type.')
            end

        case 3 % S = Signal( Amplitude, wavelength
            , Period )
            if( isnumeric(varargin{1}) && isnumeric
                (varargin{2}) && isnumeric(varargin
                {3}) )
                obj.Amplitude = varargin{1};
                obj.Wavelength = varargin{2};
                obj.Period = varargin{3};
            elseif ( ~isnumeric(varargin{1}) )
                error('Incorrect data input type
                    for Amplitude.')
            elseif( ~isnumeric(varargin{2}) )
                error('Incorrect data input type
                    for Wavelength.')
            elseif( ~isnumeric(varargin{3}) )
                error('Incorrect data input type
                    for Period.')
            else
                error('Incorrect data input type.')
```

```
            end



        case 4 % S = Signal( Amplitude, wavelength
            , Period )
            if( isnumeric(varargin{1}) && isnumeric
                (varargin{2}) && isnumeric(varargin
                {3}) )
                obj.Amplitude = varargin{1};
                obj.Wavelength = varargin{2};
                obj.Period = varargin{3};
                obj.Phase = varargin{4};
            elseif ( ~isnumeric(varargin{1}) )
                error('Incorrect data input type
                    for Amplitude.')
            elseif( ~isnumeric(varargin{2}) )
                error('Incorrect data input type
                    for Wavelength.')
            elseif( ~isnumeric(varargin{3}) )
                error('Incorrect data input type
                    for Period.')
            elseif( ~isnumeric(varargin{4}) )
                error('Incorrect data input type
                    for Phase.')
            else
                error('Incorrect data input type.')
            end
        otherwise
```

```
                    error('Invalid number of inputs for
                        QCACell');

            end % END: Switch nargin

        elseif ( strcmp(obj.Type, 'Fermi') )

            % Fermi signal



        elseif ( strcmp(obj.Type, 'Piecewise') )

            % piecewise signal




        else
            %nothing for now, but eventually other types

        end

    end

    function obj = set.Type(obj,value)
        if (~isequal(value, 'Sinusoidal') && ~isequal(value
            ,'Fermi') && ~isequal(value,'Custom') && ~
            isequal(value,'Electrode') && ~isequal(value,'
            Driver') )%edit this to add more types
```

```
            error('Invalid Type. Must be Standard signal
                Type')
        else
            obj.Type = value;

        end
    end

    function obj = set.Amplitude(obj,value)

        if ~isnumeric(value)
            error('Invalid Type. Must be Standard signal
                Type')
        else
            obj.Amplitude = value;

        end
    end

    function obj = set.Period(obj,value)
        if ~isnumeric(value)
            error('Invalid Type. Must be Standard signal
                Type')
        else
            obj.Period = value;

        end
    end
```

```
function obj = set.Wavelength(obj,value)
    if ~isnumeric(value)
        error('Invalid Type. Must be Standard signal
           Type')
    else
        obj.Wavelength = value;

    end
end

function obj = set.Phase(obj,value)
    if ~isnumeric(value)
        error('Invalid Type. Must be Standard signal
           Type')
    else
        obj.Phase = value;

    end
end

function obj = set.MeanValue(obj,value)
    if ~isnumeric(value)
        error('Invalid Type. Must be Standard signal
           Type')
    else
        obj.MeanValue = value;
```

```
        end
    end

    function obj = set.Sharpness(obj,value)
        if ~isnumeric(value)
            error('Invalid Type. Must be Standard signal
               Type')
        else
            obj.Sharpness = value;

        end
    end

    function EField = getClockField(obj, centerposition,
       time)
        %THIS FUNCTION ONLY ASSIGNS z Field RIGHT NOW

        %centerposition = [0,0,0]; %uncomment this line if
           you want uniform clock field

        if( isnumeric(centerposition) )
            if(size(centerposition) == [1, 3])
                EField = [0,0,0];
                switch obj.Type
                    case 'Sinusoidal'
                        EField(3)=( cos((2*pi*(
                           centerposition(1)/obj.Wavelength
                            - time/obj.Period ) )+ obj.
```

```
                Phase ) )*obj.Amplitude+ obj.
                MeanValue;



            case 'Fermi'
                EField(3) = obj.Amplitude *
                PeriodicFermi(mod(centerposition
                (1) - time - obj.Phase , obj.
                Period), obj.Period, obj.
                Sharpness) + obj.MeanValue;

            case 'Driver'
                EField(2) = obj.Amplitude *
                PeriodicFermi(mod(centerposition
                (1) - time - obj.Phase , obj.
                Period), obj.Period, obj.
                Sharpness) + obj.MeanValue;
                %EField(2) = ( cos((2*pi*(
                centerposition(1)/obj.Wavelength
                 - time/obj.Period ) )+ obj.
                Phase ) )*obj.Amplitude+ obj.
                MeanValue;

            otherwise
                error(['ClockType = ''', obj.Type,
                ...
                 ''' is invalid.'])
```

```
                end % END [ switch obj.Type ]









            else
                error('Incorrect data input size.')
            end
        else
            error('Incorrect data input type.')
        end

    end

    function EField = getInputField(obj, centerposition,
       time)
        if( isnumeric(centerposition) && isequal(size(
           centerposition), [1, 3]))

            centerposition = [0, 0, 0]; % force input
               signals to use same position for all nodes;
            EField = [0,0,0];
            switch obj.Type
                case 'Sinusoidal'
                    %EField(2)=( cos((2*pi*(centerposition
                       (1)/obj.Wavelength - time/obj.Period
```

```
                    ) )+ obj.Phase ) )*obj.Amplitude+
                    obj.MeanValue;


            case 'Fermi'
                EField(2) = obj.Amplitude *
                    PeriodicFermi(mod(centerposition(1)
                    - time - obj.Phase , obj.Period),
                    obj.Period, obj.Sharpness) + obj.
                    MeanValue;
            case 'Driver'
                %EField(2) = obj.Amplitude *
                    PeriodicFermi(mod(centerposition(1)
                    - time - obj.Phase , obj.Period),
                    obj.Period, obj.Sharpness) + obj.
                    MeanValue;
                %EField(2) = ( cos((2*pi*(
                    centerposition(1)/obj.Wavelength -
                    time/obj.Period ) )+ obj.Phase ) )*
                    obj.Amplitude+ obj.MeanValue;

            otherwise
                error(['ClockType = ''', obj.Type, ...
                    ''' is invalid.'])

        end % END [ switch obj.Type ]
```

```
        else
            error('centerposition has incorrect data type
               or format')
        end %centerpos is correct input type and size




    end

    function obj = drawSignal(obj, varargin)



        if( strcmp('Sinusoidal', obj.Type) || strcmp('Fermi
           ', obj.Type) )
            if nargin == 4
                nx = 200;
                xlims = varargin{1};
                ylims = varargin{2};
                t = varargin{3};

                xq = linspace(xlims(1), xlims(2), nx);
                xp = mod(xq, obj.Period);

                yq = linspace(ylims(1), ylims(2), nx);

                for idx = 1:nx
```

```
    efield_temp = obj.getClockField([xp(idx
       ), 0, 0], t);
    efield(idx) = efield_temp(3);
end %for



Eplot = repmat(efield,[nx,1]);
pcolor(xq' * ones(1, nx), ones(nx, 1)* yq,
   Eplot');




% Red column
temp7(:,1) = [ linspace(1,0,51) zeros(1,50)
    ]';
% Green column
temp7(:,2) = [ linspace(1,0,51) zeros(1,50)
   ]';
% Blue column
temp7(:,3) = [ linspace(1, 0.5625, 51)
   0.5625*ones(1, 50)]';
BlueWhite = temp7;
clear temp7
```

```
            colormap(BlueWhite);
            shading interp;
            colorbar;
            caxis([-obj.Amplitude obj.Amplitude])

            %plot(xq,efield)

        else
            error('incorrect number of inputs for
               signal type')
        end %if nargin


        %case 'Fermi'

    elseif( 'Custom' )

    elseif( 'Planar')

    else


    end %end if



end %drawSignalFunction
```

```
function obj = drawElectrode(obj, varargin)
    if nargin > 2
        centerpos = varargin{1};
        height = varargin{2};
        width = varargin{3};
        Efield = varargin{4};




        obj.CenterPosition = centerpos;
        obj.Height = height;
        obj.Width = width;
        obj.InputField = Efield;

    else
        centerpos = obj.CenterPosition;
        height = obj.Height;
        width = obj.Width;
        Efield = obj.InputField;



    end
    if strcmp(obj.Type,'Electrode')

        %draw the text box showing the electric field,
           and
        %lower/upper patches to denote electrodes
```

```
txt = text(centerpos(1) - width/2-.7 ,
   centerpos(2) , [num2str(Efield) ' V/m']);

if and(height ,width)
    name = text(centerpos(1) + width/2 ,
       centerpos(2)+height/2 , num2str(obj.Name
       ));

elseif and(~height,width)
    name = text(centerpos(1) + width/2 ,
       centerpos(2)+.75 , num2str(obj.Name));

elseif and(height,~width)
    name = text(centerpos(1) + .5 , centerpos
       (2)+height/2 , num2str(obj.Name));

else
    name = text(centerpos(1) + .5 , centerpos
       (2)+.75, num2str(obj.Name));

end

obj.TopPatch = patch('FaceColor','red','XData'
   ,[centerpos(1) - width/2-.25  centerpos(1) +
    width/2+.25  centerpos(1) + width/2+.25
   centerpos(1) - width/2-.25]...
```

```
    ,'YData',[centerpos(2)+ height/2+.75
       centerpos(2)+height/2+.75  centerpos(2)
       + height/2+.95   centerpos(2) + height/
       2+.95]);

obj.BottomPatch = patch('FaceColor','black','
   XData',[centerpos(1) - width/2-.25
   centerpos(1) + width/2+.25   centerpos(1) +
   width/2+.25    centerpos(1) - width/
   2-.25]...
    ,'YData',[centerpos(2)- height/2-.95
       centerpos(2)-height/2-.95  centerpos(2)
       - height/2-.75   centerpos(2) - height/
       2-.75]);
Select(obj.TopPatch);
Select(obj.BottomPatch);

x0 = centerpos(1) - width/2-.25;
x1 = centerpos(1) + width/2+.25;

if height == 0
    low = centerpos(2) - .75;
    high= centerpos(2) + .75;
else
    low = centerpos(2) - height/2-.75;
    high= centerpos(2) + height/2+.75;
end
```

```
Efield;
num=0;

%intervals of E field magnitude to determine
   number of
%electric field lines between the electrodes
if  Efield > 8
    num = (x1-x0)/20;

    for i=x0 : num : x1



        p1 = [i low];
        p2 = [i high];
        dp = p2-p1;

        hold on;
        q=quiver(p1(1),p1(2),dp(1),dp(2),0);
        %                                q.
           LineWidth = 5;
        q.MarkerEdgeColor = 'black';
        q.Color = 'black';
        p.Parent = gca;

    end

elseif Efield > 6 && Efield <= 8
```

```
            num = (x1-x0)/20;
            for i=x0 : num : x1

                p1 = [i low];
                p2 = [i high];
                dp = p2-p1;

                hold on;
                q=quiver(p1(1),p1(2),dp(1),dp(2),0);
                %                           q.LineWidth =
                    3;
                q.MarkerEdgeColor = 'black';
                q.Color = 'black';
                p.Parent = gca;
            end

        elseif Efield > 4 && Efield <= 6
            num = (x1-x0)/9;
            for i=x0 : num : x1

                p1 = [i low];
                p2 = [i high];
                dp = p2-p1;

                hold on;
                q=quiver(p1(1),p1(2),dp(1),dp(2),0);
                %                           q.LineWidth =
                    2;
```

```
                q.MarkerEdgeColor = 'black';
                q.Color = 'black';
                p.Parent = gca;
            end

        elseif Efield > 2 && Efield <= 4
            num = (x1-x0)/5;
            for i=x0 : num : x1

                p1 = [i low];
                p2 = [i high];
                dp = p2-p1;

                hold on;
                q=quiver(p1(1),p1(2),dp(1),dp(2),0);
                %                           q.LineWidth =
                    1;
                q.MarkerEdgeColor = 'black';
                q.Color = 'black';
                p.Parent = gca;
            end



        elseif Efield <= 2 && Efield > 0
            num = (x1-x0)/2;
            for i=x0 : num : x1
```

```
                    p1 = [i low];
                    p2 = [i high];
                    dp = p2-p1;

                    hold on;
                    q=quiver(p1(1),p1(2),dp(1),dp(2),0);
                    %                           q.LineWidth =
                        .1;
                    q.MarkerEdgeColor = 'black';
                    q.Color = 'black';
                    p.Parent = gca;
                end



            elseif Efield >= -2 && Efield < 0
                num = (x1-x0)/2;
                for i=x0 : num : x1

                    p1 = [i low];
                    p2 = [i high];
                    dp = p1-p2;

                    hold on;
                    q=quiver(p2(1),p2(2),dp(1),dp(2),0);
                    %                           q.LineWidth =
                        .1;
                    q.MarkerEdgeColor = 'black';
                    q.Color = 'black';
```

```
            p.Parent = gca;
        end




    elseif Efield < -2 && Efield >= -4
        num = (x1-x0)/5;
        for i=x0 : num : x1


            p1 = [i low];
            p2 = [i high];
            dp = p1-p2;


            hold on;
            q=quiver(p2(1),p2(2),dp(1),dp(2),0);
            %                           q.LineWidth =
                1;
            q.MarkerEdgeColor = 'black';
            q.Color = 'black';
            p.Parent = gca;
        end


    elseif Efield < -4 && Efield >= -6
        num = (x1-x0)/9;
        for i=x0 : num : x1


            p1 = [i low];
            p2 = [i high];
```

```
                dp = p1-p2;


                hold on;
                q=quiver(p2(1),p2(2),dp(1),dp(2),0);
                %                           q.LineWidth =
                    2;
                q.MarkerEdgeColor = 'black';
                q.Color = 'black';
                p.Parent = gca;
            end


        elseif Efield < -6 && Efield >= -8
            num =(x1-x0)/15;
            for i=x0 : num : x1


                p1 = [i low];
                p2 = [i high];
                dp = p1-p2;


                hold on;
                q=quiver(p2(1),p2(2),dp(1),dp(2),0);
                %                           q.LineWidth =
                    3;
                q.MarkerEdgeColor = 'black';
                q.Color = 'black';
                p.Parent = gca;
            end
```

```
elseif Efield < -8
    num = (x1-x0)/20;
    for i=x0 : num : x1

        p1 = [i low];
        p2 = [i high];
        dp = p1-p2;

        hold on;
        q=quiver(p2(1),p2(2),dp(1),dp(2),0);
        %                          q.LineWidth =
            5;
        q.MarkerEdgeColor = 'black';
        q.Color = 'black';
        p.Parent = gca;
    end

elseif Efield == 0

    %don't draw E field lines

end

%we must now change the order of the Children
   of
%axes.Children so the arrows are behind the
   cell drawings
num;
```

```
            num=(x1-x0)*num^(-1)+1; %interval multiplied by
                the inverse of the iteration length plus 1;

            ax=gca;

            graphicsList = get(ax,'children'); %list we are
                going to change
            newList = [];

            for j=1:num
                newList(end+1)= graphicsList(j);%put all
                   the arrows into the newList
            end

            set(ax,'children',[ax.Children(j+1:end);
               newList']); %the arrows are now at the end
               of the Children handle



            hold off;
        end

        myCircuit=getappdata(gcf,'myCircuit');

        for i=1:length(myCircuit.Device)
            if isa(myCircuit.Device{i},'QCASuperCell')
                for j=1:length(myCircuit.Device{i}.Device)
```

```
                    circCenter = myCircuit.Device{i}.Device
                       {j}.CenterPosition;
                    if circCenter(1) > centerpos(1) &&
                       circCenter(2) > centerpos(2) &&
                       circCenter(1) < centerpos(1)+width/2
                        && circCenter(2) > centerpos(2)+
                       height/2
                        myCircuit.Device{i}.Device{j}.
                           ElectricField = Efield;


                    end
                end
            else

                circCenter = myCircuit.Device{i}.
                   CenterPosition;
                if circCenter(1) > centerpos(1) &&
                   circCenter(2) > centerpos(2) &&
                   circCenter(1) < centerpos(1)+width/2 &&
                   circCenter(2) > centerpos(2)+height/2
                    myCircuit.Device{i}.ElectricField = [0
                       Efield 0];


                end
            end

        end
```

```
        end

    end

end
```

Listing A.51. Signal Class

```
function SignalTypePanelSwitch(handles, sigType)
%The purpose of this function is to switch between the 4
   different signals
%panels within the major signal panel.



%switching visibility on or off
switch sigType

    case 'Sinusoidal'

        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'on';
        handles.electrodePanel.Visible = 'on';
        handles.fermiPanel.Visible = 'on';

    case 'Fermi'
        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'on';
        handles.electrodePanel.Visible = 'on';
        handles.fermiPanel.Visible = 'on';
```

```
    case 'Custom'
        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'on';
        handles.electrodePanel.Visible = 'on';
        handles.fermiPanel.Visible = 'on';

    case 'Electrode'
        handles.sinusoidPanel.Visible = 'on';
        handles.customSignal.Visible = 'on';
        handles.electrodePanel.Visible = 'on';
        handles.fermiPanel.Visible = 'on';



end



end
```

Listing A.52. SignalTypePanelSwitch Function

```
function Simulate(handles)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
ax=gca;




myCircuit = getappdata(gcf,'myCircuit');
```

```
%     handles.layoutchange.Value=0;

clockSignalsList = getappdata(gcf,'clockSignalsList');



% x=[];
% y=[];

%regenerate neighbor list once we hit the simulate button

myCircuit=myCircuit.GenerateNeighborList();



% eps0 = 8.854E-12;%set constants
% a=1e-9;
% q=1;
% E0 = q^2 * (1.602e-19) / (4*pi*eps0*a)* (1-1/sqrt(2));
% clk= str2num(get(handles.chngClock,'String'));
%
%
% for i=1:length(myCircuit.Device)%set clock field for all
   cells
%
%     if isa(myCircuit.Device{i},'QCASuperCell')
%         for j=1:length(myCircuit.Device{i}.Device)
%             myCircuit.Device{i}.Device{j};
%             %
%
```

```
% %               if (myCircuit.Device{i}.Device{j}.ElectricField
   (2)==0)
%                   myCircuit.Device{i}.Device{j}.ElectricField
   (3)=clk*E0;
% %               end
%
% %               if strcmp(myCircuit.Device{i}.Device{j}.
   SelectBox.Selected,'on')
% %                   x(end+1)=myCircuit.Device{i}.Device{j}.
   CenterPosition(1);
% %                   y(end+1)=myCircuit.Device{i}.Device{j}.
   CenterPosition(2);
% %
% %               end
%           end
%
%
%      else
%           myCircuit.Device{i};
% %           if (myCircuit.Device{i}.ElectricField(2)==0)
% %               myCircuit.Device{i}.ElectricField(3) = clk*E0;
% %           else
%               myCircuit.Device{i}.ElectricField(3) =  clk*E0;
% %           end
%
% %           if strcmp(myCircuit.Device{i}.SelectBox.Selected,'
   on')
% %               x(end+1)=myCircuit.Device{i}.CenterPosition(1);
```

```
% %                y(end+1)=myCircuit.Device{i}.CenterPosition(2);
% %
% %           end
%           myCircuit.Device{i};
%       end
%
% end


% x;
% y;


%relax and redraw
myCircuit=myCircuit.Relax2GroundState(0);


% xdiff=0;
% ydiff=0;


% must draw the circuit before we can draw the arrows denoting
    the electric
% field lines
myCircuit=myCircuit.CircuitDraw(0,gca);



setappdata(gcf,'myCircuit',myCircuit);


end
```

Listing A.53. Simulate Function

```
classdef SixDotCell < QCACell
```

```
%ThreeDotCell Defines ThreeDotCell subclass of QCACell
%   Detailed explanation goes here



properties (Constant) %Helpful Constants used in
   ThreeDotCell

    %basis States
    one  = [0;0;1];
    null = [0;1;0];
    zero = [1;0;0];

    %helpful operators
    Z = [-1 0 0; 0 0 0; 0 0 1];
    Pnn = [0 0 0; 0 1 0; 0 0 0 ];




end


properties
    Polarization = 0;
    Activation = 1;
    Hamiltonian = zeros(3);
    Wavefunction = zeros(3,1);
    SelectBox;
```

```
    Overlapping='off';

    radiusOfEffect = 3.1;

    intP = [0;0;0];

    fixedChargeLocation = 0.5*[ 1, 1, 1; ...
                         1, 0,0; ...
                         1,-1,1; ...
                        -1,-1,1; ...
                        -1, 0,0; ...
                        -1, 1,1];
    fixedCharge = -1/3;  %not used, look in potential
       energy function for real implementation


end


methods
    function obj = SixDotCell( varargin )

        DotPosition = 0.5*[ 1, 1,1; ...
                            1, 0,0; ...
                            1,-1,1; ...
                           -1,-1,1; ...
                           -1, 0,0; ...
```

```
                        -1, 1,1]; %Dot relative position
                            in Characteristic Lengths

    SelectBox=patch('Visible','off');
    switch nargin
        case 0
            Position = [0,0,0];

        case 1
            Position = varargin{1};

        case 2
            Position = varargin{1};
            DotPosition = varargin{2};

        otherwise
            error('Invalid number of inputs for
               SixDotCell().')
    end

    obj = obj@QCACell( Position );
    obj.DotPosition = DotPosition;


    obj.intP = obj.internal_potential();

end
```

```
        function obj = set.Polarization(obj,value)
            if ( (isnumeric(value) && value >= -1 && value <=
               1) || isa(value, 'Signal')) %value must be
               numeric in between -1 and 1
%                  error('Invalid Polarization. Must be a number
    inbetween -1 and 1')

                obj.Polarization = value;



            else
                error('Invalid Polarization. Must be a number
                   inbetween -1 and 1')

            end
        end

        function pol = getPolarization(obj, time)
            if isnumeric(obj.Polarization)
                pol = obj.Polarization;
                %disp([num2str(obj.CellID), ': ', num2str(obj.
                   Polarization)])
            else
                temppol = obj.Polarization.getClockField
                   ([0,0,0], time);
                pol = temppol(2);
                %disp([ num2str(obj.CellID), ' has a signal obj
                   '])
```

```
    end
end


function Int = internal_potential(obj)
    import QCALayoutPack.*
    qe = QCA_Constants.qe;
    h = -1*qe;
    c = 2*qe/3;


    k = (1/(4*pi*QCA_Constants.epsilon_0)*QCA_Constants
       .qeC2e);
    Int = [0;0;0];

    objDotPosition = obj.getDotPosition();

             %%%%ZERO
    displacementVector = objDotPosition(1,:)-
       objDotPosition(3,:);
    distance = sqrt( sum(displacementVector.^2, 2) );
    u13 = c*h/distance;

    displacementVector = objDotPosition(1,:)-
       objDotPosition(6,:);
    distance = sqrt( sum(displacementVector.^2, 2) );
```

```
u16 = c*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(3,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u23 = c*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u26 = qe*h/distance;

displacementVector = objDotPosition(3,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u36 = c*c/distance;

displacementVector = objDotPosition(3,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u35 = c*h/distance;

displacementVector = objDotPosition(3,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u34 = c*h/distance;
```

```
displacementVector = objDotPosition(4,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u46 = c*h/distance;

displacementVector = objDotPosition(5,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u56 = c*h/distance;



Int(1) = k*(u13 + u16 + u23 + u26 + u34 + u35 + u36
    + u46 + u56);

%%%%NULL
displacementVector = objDotPosition(1,:)-
   objDotPosition(2,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u12 = c*h/distance;

displacementVector = objDotPosition(1,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u15 = c*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(3,:);
distance = sqrt( sum(displacementVector.^2, 2) );
```

```
u23 = c*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u26 = qe*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u25 = c*c/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u24 = c*h/distance;

displacementVector = objDotPosition(3,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u35 = c*h/distance;

displacementVector = objDotPosition(6,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u65 = c*h/distance;
```

```
displacementVector = objDotPosition(5,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u54 = c*h/distance;



Int(2) = k*(u12 + u15 + u23 + u26 + u25 + u24 + u35
    + u65 + u54);

%%%%ONE
displacementVector = objDotPosition(1,:)-
   objDotPosition(2,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u12 = c*h/distance;

displacementVector = objDotPosition(1,:)-
   objDotPosition(3,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u13 = c*h/distance;

displacementVector = objDotPosition(1,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u14 = c*c/distance;

displacementVector = objDotPosition(1,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
```

```
u15 = c*h/distance;

displacementVector = objDotPosition(1,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u16 = c*h/distance;

displacementVector = objDotPosition(2,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u24 = c*h/distance;

displacementVector = objDotPosition(3,:)-
   objDotPosition(4,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u34 = c*h/distance;

displacementVector = objDotPosition(4,:)-
   objDotPosition(5,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u45 = c*h/distance;

displacementVector = objDotPosition(4,:)-
   objDotPosition(6,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u46 = c*h/distance;
```

```
        Int(3) = k*(u12 + u13 + u14 + u15 + u16 + u24 + u34
            + u45 + u46);




    end

    function mobileCharge = getMobileCharge(obj, time)
        import QCALayoutPack.*
        qe = QCA_Constants.qe;

        mobileCharge = -1*[qe*obj.Activation*(1/2)*(obj.
           getPolarization(time)+1);1-obj.Activation;qe*obj
           .Activation*(1/2)*(1-obj.getPolarization(time));
           qe*obj.Activation*(1/2)*(obj.getPolarization(
           time)+1);1-obj.Activation;qe*obj.Activation*(1/
           2)*(1-obj.getPolarization(time))];

    end

    function pot_energy = Potential(obj, obsvPoint, time )
        import QCALayoutPack.*
        qe = QCA_Constants.qe;
        epsilon_0 = QCA_Constants.epsilon_0;
        qeC2e = QCA_Constants.qeC2e;

        selfDotPos = obj.getDotPosition();
```

```
            numberofDots = size(selfDotPos, 1);

            numberofFixedCharge = size(obj.fixedChargeLocation,
                1);

            totalnumofDots = numberofDots+numberofFixedCharge;


            allDots = [selfDotPos; obj.fixedChargeLocation];

            %charge = qe*obj.Activation*[(1/2)*(obj.
               Polarization+1);-1;(1/2)*(1-obj.Polarization);(1
               /2)*(obj.Polarization+1);-1;(1/2)*(1-obj.
               Polarization)]; %[eV]
            %charge = qe*obj.Activation*[(1/2)*(obj.
               Polarization+1);
                                        %-1;
                                        %(1/2)*(1-obj.
                                           Polarization);
                                        %(1/2)*(obj.
                                           Polarization+1);
                                        %-1;
                                        %(1/2)*(1-obj.
                                           Polarization)]; %[
                                           eV]

            charge = [qe*obj.Activation*(1/2)*(obj.
               getPolarization(time)+1);1-obj.Activation;qe*obj
```

```
            .Activation*(1/2)*(1-obj.getPolarization(time));
            qe*obj.Activation*(1/2)*(obj.getPolarization(
            time)+1);1-obj.Activation;qe*obj.Activation*(1/
            2)*(1-obj.getPolarization(time))];
        allcharge = charge + [-1/3;-1/3;-1/3;-1/3;-1/3;-1/
            3];

        displacementVector = ones(numberofDots,1)*obsvPoint
            - selfDotPos;
        distance = sqrt( sum(displacementVector.^2, 2) );
        %d=sqrt(displacementVector(:,1).^2+
            displacementVector(:,2).^2+displacementVector
            (:,3).^2);

        %temp = (1/(4*pi*epsilon_0)*qeC2e)*(allcharge
            (1:6,:)+allcharge(7:12,:));
        %pot_energy = sum(temp./(d*1e-9));
%           if distance < (obj.radiusOfEffect+10)
              pot_energy = (1/(4*pi*epsilon_0)*qeC2e)*sum(
                  allcharge./(distance*1E-9));
%           else
%               pot_energy = 0
%           end

        %disp(['potential of ' num2str(obj.CellID) ' is '
            num2str(pot)])
    end
```

```
function V_neighbors = neighborPotential(obj, obj2,
   time) %obj2 should have a potential function(ie, a
   QCACell or QCASuperCell. Each will call potential at
    spots)
    %returns the potential of a neighborCell

    %find out who the neighbors are (decide between
       giving this func the list of all neighbors or
       having this func figure that out. for now just 1
        neighbor)
    %possible option of asking the QCACircuit obj.

    %find the potential at each dot then. self1 + self2
        + self3.

    objDotPosition = obj.getDotPosition();
    V_neighbors = zeros(size(objDotPosition,1),1);

    for x = 1:length(objDotPosition)
        V_neighbors(x,:) = obj2.Potential(
           objDotPosition(x,:), time );
    end



end
```

```
        function hamiltonian = GetHamiltonian(obj, neighborList
           , time)

            objDotpotential = zeros(size(obj.DotPosition,1),1);


            for x = 1:length(neighborList)
%                 disp([num2str(obj.CellID) '---' num2str(
   neighborList{x}.CellID)])
                objDotpotential = objDotpotential + obj.
                   neighborPotential(neighborList{x},time);
            end

            gammaMatrix = -obj.Gamma*[0,1,0;1,0,1;0,1,0];


            hamiltonian = -1*[objDotpotential(3)+
               objDotpotential(6),0,0;...
                           0,objDotpotential(2)+objDotpotential
                              (5),0;...
                           0,0,objDotpotential(1)+
                              objDotpotential(4)]...
                           +gammaMatrix;


            h = abs(obj.DotPosition(2,3)-obj.DotPosition(1,3));
                %Field over entire height of cell
            x = abs(obj.DotPosition(6,1)-obj.DotPosition(1,1));
```

```
            y = abs(obj.DotPosition(3,2)-obj.DotPosition(1,2));
            lengthh = [x,y,0];


            inputFieldBias = -obj.ElectricField*lengthh';


            hamiltonian(2,2) = hamiltonian(2,2) + -obj.
               ElectricField(1,3)*h; %add clock E
            hamiltonian(1,1) = hamiltonian(1,1);% + (-
               inputFieldBias)/2;%add input field to 0 dot
            hamiltonian(3,3) = hamiltonian(3,3);% +
               inputFieldBias/2;%add input field to 1 dot


            %Calculate internal potential and add them to
               hamiltonian
            hamiltonian(1,1) = hamiltonian(1,1) + obj.intP(1);
            hamiltonian(2,2) = hamiltonian(2,2) + obj.intP(2);
            hamiltonian(3,3) = hamiltonian(3,3) + obj.intP(3);

%             disp(['neighbor list of ' num2str(obj.CellID) '
   is:     ' num2str(obj.NeighborList)])
%             disp('and the hamiltonian is ')

%             hamiltonian
        end
```

```
function obj = Calc_Polarization_Activation(obj,
   varargin)

    if length(varargin) == 1

        if( strcmp(obj.Type , 'Driver' ))
            %don't try to relax this cell
        else
            normpsi = varargin{1};
            obj.Polarization = normpsi' * obj.Z *
               normpsi;
            obj.Activation = 1 - normpsi' * obj.Pnn *
               normpsi;

            obj.Wavefunction = normpsi;
        end
    else


        if( strcmp(obj.Type , 'Driver' ))
            %don't try to relax this cell
        else
            %relax
            %testing getHamiltonian.

            [V, EE] = eig(obj.Hamiltonian);
            psi = V(:,1); %ground state
            obj.Wavefunction = psi;
```

```
            % Polarization is the expectation value of
               sigma_z
            obj.Polarization = psi' * obj.Z * psi;
            %disp(['P of cell ' num2str(obj.CellID) '
               is ' num2str(obj.Polarization)])
            obj.Activation = 1 - psi' * obj.Pnn * psi;
            %disp(['A of cell ' num2str(obj.CellID) '
               is ' num2str(obj.Activation)])
        end


    end
end

function obj = ColorDraw(obj, time, varargin)
    targetAxes = [];
    a= obj.CharacteristicLength;

    r= obj.CenterPosition;
    x=a*.75*[-1,1,1,-1] + r(1);
    y=a*.75*[1,1,-1,-1] + r(2);
    %r(3) would be in the z direction


    cell_patch = patch(x,y,'r');
    faceColor = getFaceColor(obj, time);
    cell_patch.FaceColor = faceColor;
```

```
%             c1 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2), a*.125*abs(obj.Polarization), [1 1 1]);
%             c2 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)+a*.4, a*.125, [1 1 1]);
%             c3 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)-a*.4, a*.125, [1 1 1]);



              %text(obj.CenterPosition(1), obj.CenterPosition
                 (2)+.8*a, num2str(obj.CellID), '
                 HorizontalAlignment', 'center','FontSize',12);



          if length(varargin)==1
              targetAxes = varargin{1};
              %axes(targetAxes);
              hold off;
          end




      end


      function color = getFaceColor(obj, time)
          pol = obj.getPolarization(time);
          if(pol < 0)
              %color = [abs(pol) 0 0];
```

```
            color = [1-abs(pol) 1 1-abs(pol)];
        elseif(pol > 0)
            %color = [0 abs(pol) 0];
            color = [1 1-abs(pol) 1-abs(pol)];
        else
            color = [1 1 1];
        end
    end

    function obj = BoxDraw(obj)
        obj.SelectBox=patch;
        obj.SelectBox.XData=[obj.CenterPosition(1)-.75;obj.
           CenterPosition(1)+.75;obj.CenterPosition(1)+.75;
           obj.CenterPosition(1)-.75];
        obj.SelectBox.YData=[obj.CenterPosition(2)-.75;obj.
           CenterPosition(2)-.75;obj.CenterPosition(2)+.75;
           obj.CenterPosition(2)+.75];

        obj.CellID;
        obj.Overlapping;
        switch obj.Overlapping
            case 'off'
                obj.SelectBox.FaceColor=[1 1 1];
                obj.SelectBox.FaceAlpha = .01;
            case 'on'
                obj.SelectBox.FaceColor=[1 0 0];
                obj.SelectBox.FaceAlpha = .4;
```

```
        end

        obj.SelectBox.UserData = obj.CellID;
    end

    function obj = ElectronDraw(obj, time, varargin)

            targetAxes = [];
            a= obj.CharacteristicLength;
            centerpos = obj.CenterPosition;
            reldotpos = obj.DotPosition;
            dotpos = obj.getDotPosition();
            act = obj.Activation;
            pol = obj.getPolarization(time);
            radiusfactor = 0.2;

            if length(varargin)==1
                targetAxes = varargin{1};
                %axes(targetAxes);
                hold off;

            end

            %Tunnel junctions
            x_dist13 = [dotpos(1,1), dotpos(3,1)];
            y_dist13 = [dotpos(1,2), dotpos(3,2)];
            l13 = line(x_dist13, y_dist13, 'LineWidth', 2,
               'Color', [0 0 0]);
```

```
x_dist64 = [dotpos(6,1), dotpos(4,1)];
y_dist64 = [dotpos(6,2), dotpos(4,2)];
l64 = line(x_dist64, y_dist64, 'LineWidth', 2,
   'Color', [0 0 0]);

%text(obj.CenterPosition(1), obj.CenterPosition
   (2)+.8*a, num2str(obj.CellID), '
   HorizontalAlignment', 'center','FontSize
   ',12);




%Electron Sites
c1 = circle(dotpos(1,1), dotpos(1,2), a*
   radiusfactor, [1 1 1],'Points',25, '
   TargetAxes', targetAxes);
c2 = circle(dotpos(2,1), dotpos(2,2), a*
   radiusfactor, [1 1 1],'Points',25, '
   TargetAxes', targetAxes);
c3 = circle(dotpos(3,1), dotpos(3,2), a*
   radiusfactor, [1 1 1],'Points',25, '
   TargetAxes', targetAxes);
c4 = circle(dotpos(4,1), dotpos(4,2), a*
   radiusfactor, [1 1 1],'Points',25, '
   TargetAxes', targetAxes);
```

```
                c5 = circle(dotpos(5,1), dotpos(5,2), a*
                   radiusfactor, [1 1 1],'Points',25, '
                   TargetAxes', targetAxes);
                c6 = circle(dotpos(6,1), dotpos(6,2), a*
                   radiusfactor, [1 1 1],'Points',25, '
                   TargetAxes', targetAxes);

                %extra circle
%                 c123 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2), obj.radiusOfEffect, [1 1 1], 'EdgeColor',
    [0 0 0], 'Points',25);
%                 c123.FaceColor = 'None';

                %electron color for driver/node
                if strcmp(obj.Type,'Node')
                    electronColor = [1 0 0];
                elseif strcmp(obj.Type,'Driver')
                    electronColor = [0 1 0];
                end

                %Electrons Position Probability
                q1 = (act/2)*(1+pol);
                q2 = 1 - act;
                q3 = (act/2)*(1-pol);
                q4 = q1;
                q5 = q2;
                q6 = q3;
```

```
scalefactor = 0.90;
e1 = circle(dotpos(1,1), dotpos(1,2), q1*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
e2 = circle(dotpos(2,1), dotpos(2,2), q2*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
e3 = circle(dotpos(3,1), dotpos(3,2), q3*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
e4 = circle(dotpos(4,1), dotpos(4,2), q4*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
e5 = circle(dotpos(5,1), dotpos(5,2), q5*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
e6 = circle(dotpos(6,1), dotpos(6,2), q6*a*
   radiusfactor*scalefactor, electronColor,'
   EdgeColor', 'None','Points',25, 'TargetAxes'
   , targetAxes);
```

```
                obj = obj.BoxDraw();


        end
    end
end
```

Listing A.54. SixDotCell Class

```
classdef ThreeDotCell < QCACell
    %ThreeDotCell Defines ThreeDotCell subclass of QCACell
    %   Detailed explanation goes here




    properties (Constant) %Helpful Constants used in
       ThreeDotCell

        %basis States
        one  = [0;0;1];
        null = [0;1;0];
        zero = [1;0;0];

        %helpful operators
        Z = [-1 0 0; 0 0 0; 0 0 1];
        Pnn = [0 0 0; 0 1 0; 0 0 0 ];
        a
```

```
end

properties
    Polarization = 0;
    Activation = 1;
    Hamiltonian = zeros(3);
    Wavefunction = zeros(3,1);
    SelectBox;
    Overlapping='off';

    radiusOfEffect = 3.1;

    intP = [0;0;0];

    side = 'r'; % or 'l'

end



methods
    function obj = ThreeDotCell( varargin )

        DotPosition = 0.5*[0,1,1; ...
            0,0,0; ...
            0,-1,1; ]; %Dot relative position in
                Characteristic Lengths

        SelectBox=patch('Visible','off');
```

```
        switch nargin
            case 0
                Position = [0,0,0];

            case 1
                Position = varargin{1};

            case 2
                Position = varargin{1};
                DotPosition = varargin{2};

            otherwise
                error('Invalid number of inputs for
                   ThreeDotCell().')
        end




        obj = obj@QCACell( Position );
        obj.DotPosition = DotPosition;
    end

    function obj = set.Polarization(obj,value)
        if ( (isnumeric(value) && value >= -1 && value <=
           1) || isa(value, 'Signal')) %value must be
           numeric in between -1 and 1 or a signal class

            obj.Polarization = value;
```

```
        else
            error('Invalid Polarization. Must be a number
                inbetween -1 and 1')

        end
    end

    function pol = getPolarization(obj, time)
        if isnumeric(obj.Polarization)
            pol = obj.Polarization;
            %disp([num2str(obj.CellID), ': ', num2str(obj.
                Polarization)])
        else
            temppol = obj.Polarization.getClockField
                ([0,0,0], time);
            pol = temppol(2);
            %disp([ num2str(obj.CellID), ' has a signal obj
                '])


        end
    end

    function Int = internal_potential(obj)
        import QCALayoutPack.*
        qe = QCA_Constants.qe;
        h = -1*qe;
```

```
c = 2*qe/3;



k = (1/(4*pi*QCA_Constants.epsilon_0)*QCA_Constants
   .qeC2e);
Int = [0;0;0];

objDotPosition = obj.getDotPosition();

%%%%ZERO
displacementVector = obj.DotPosition(1,:)-obj.
   DotPosition(2,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u12 = qe*h/distance;

Int(1) = k*u12;

%%%%NULL
Int(2) = 0;

%%%%ONE
displacementVector = objDotPosition(2,:)-
   objDotPosition(3,:);
distance = sqrt( sum(displacementVector.^2, 2) );
u23 = qe*h/distance;

Int(3) = k*u23;
```

```
end


function mobileCharge = getMobileCharge(obj, time)
    import QCALayoutPack.*
    qe = QCA_Constants.qe;

    mobileCharge = [qe*obj.Activation*(1/2)*(1-obj.
       getPolarization(time));1-obj.Activation;qe*obj.
       Activation*(1/2)*(obj.getPolarization(time)+1)];

end

function pot = Potential(obj, obsvPoint, time )
    import QCALayoutPack.*
    qe = QCA_Constants.qe;
    epsilon_0 = QCA_Constants.epsilon_0;
    qeC2e = QCA_Constants.qeC2e;

    selfDotPos = getDotPosition(obj);
    numberofDots = size(selfDotPos, 1);

    %charge = qe*obj.Activation*[(1/2)*(1-obj.
       getPolarization(time));-1;(1/2)*(obj.
       getPolarization(time)+1)]; %[eV];
```

```
            charge = [qe*obj.Activation*(1/2)*(1-obj.
               getPolarization(time));1-obj.Activation;qe*obj.
               Activation*(1/2)*(obj.getPolarization(time)+1)];
            %allcharge = charge + ones(3,1)*(-1/3);
            allcharge = charge + [0; -1; 0];

            displacementVector = ones(numberofDots,1)*obsvPoint
                - selfDotPos;
            distance = sqrt( sum(displacementVector.^2, 2) );

%             if distance < obj.radiusOfEffect
            pot = (1/(4*pi*epsilon_0)*qeC2e)*sum(allcharge./(
               distance*1E-9));
%             else
%                 pot = 0;
%             end

            %disp(['potential of ' num2str(obj.CellID) ' is '
               num2str(pot)])
        end

        function V_neighbors = neighborPotential(obj, obj2,
           time) %obj2 should have a potential function(ie, a
           QCACell or QCASuperCell. Each will call potential at
            spots)
            %returns the potential of a neighborCell

            %find out who the neighbors are. (Neighbor List)
```

```
            %find the potential at each dot then. self1 + self2
                + self3.

            objDotPosition = obj.getDotPosition();
            V_neighbors = zeros(size(objDotPosition,1),1);

            for x = 1:length(objDotPosition)
                V_neighbors(x,:) = obj2.Potential(
                   objDotPosition(x,:), time );
            end



        end

        function hamiltonian = GetHamiltonian(obj, neighborList
           , time)

            objDotpotential = zeros(size(obj.DotPosition,1),1);


            for x = 1:length(neighborList)

%                 disp([num2str(obj.CellID) '---' num2str(
   neighborList{x}.CellID)])
                objDotpotential = objDotpotential + obj.
                   neighborPotential(neighborList{x}, time);
```

```
end

gammaMatrix = -obj.Gamma*[0,1,0;1,0,1;0,1,0];

obj.CellID;
hamiltonian = -diag(objDotpotential) + gammaMatrix;

h = abs(obj.DotPosition(2,3)-obj.DotPosition(1,3));
    %Field over entire height of cell
x = abs(obj.DotPosition(3,1)-obj.DotPosition(1,1));
y = abs(obj.DotPosition(3,2)-obj.DotPosition(1,2));
lengthh = [x,y,0];


inputFieldBias = -obj.ElectricField*lengthh';


hamiltonian(2,2) = hamiltonian(2,2) + -obj.
   ElectricField(1,3)*h; %add clock E
hamiltonian(1,1) = hamiltonian(1,1) + (-
   inputFieldBias)/2;%add input field to 0 dot
hamiltonian(3,3) = hamiltonian(3,3) +
   inputFieldBias/2;%add input field to 1 dot

%Calculate internal potential and add them to
   hamiltonian
hamiltonian(1,1) = hamiltonian(1,1) + obj.intP(1);
hamiltonian(2,2) = hamiltonian(2,2) + obj.intP(2);
```

```
            hamiltonian(3,3) = hamiltonian(3,3) + obj.intP(3);

%               disp(['neighbor list of ' num2str(obj.CellID) '
   is:      ' num2str(obj.NeighborList)])
%               disp('and the hamiltonian is ')

%               hamiltonian
        end

        function obj = Calc_Polarization_Activation(obj,
           varargin)

            if length(varargin) == 1

                if( strcmp(obj.Type , 'Driver' ))
                    %don't try to relax this cell
                else
                    normpsi = varargin{1};
                    obj.Polarization = normpsi' * obj.Z *
                       normpsi;
                    obj.Activation = 1 - normpsi' * obj.Pnn *
                       normpsi;

                    obj.Wavefunction = normpsi;
                end
            else
```

```
            if( strcmp(obj.Type , 'Driver' ))
                %don't try to relax this cell
            else
                %relax
                %testing getHamiltonian.

                [V, EE] = eig(obj.Hamiltonian);
                psi = V(:,1); %ground state
                obj.Wavefunction = psi;

                % Polarization is the expectation value of
                   sigma_z
                obj.Polarization = psi' * obj.Z * psi;
                %disp(['P of cell ' num2str(obj.CellID) '
                   is ' num2str(obj.Polarization)])
                obj.Activation = 1 - psi' * obj.Pnn * psi;
                %disp(['A of cell ' num2str(obj.CellID) '
                   is ' num2str(obj.Activation)])
            end


        end
    end

    function obj = ColorDraw(obj, varargin)
        targetAxes = [];
        a= obj.CharacteristicLength;
```

```
            r= obj.CenterPosition;
            x=a*.25*[-1,1,1,-1] + r(1);
            y=a*.75*[1,1,-1,-1] + r(2);
            %r(3) would be in the z direction


            cell_patch = patch(x,y,'r');
            faceColor = getFaceColor(obj);
            cell_patch.FaceColor = faceColor;

%             c1 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2), a*.125*abs(obj.Polarization), [1 1 1]);
%             c2 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)+a*.4, a*.125, [1 1 1]);
%             c3 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)-a*.4, a*.125, [1 1 1]);




            if length(varargin)==1
                targetAxes = varargin{1};
                %axes(targetAxes);
                hold off;
            end
```

```
        end

        function color = getFaceColor(obj)
            pol = obj.Polarization;
            a= obj.CharacteristicLength;
            %text(obj.CenterPosition(1), obj.CenterPosition(2)
               +.8*a, num2str(obj.CellID), 'HorizontalAlignment
               ', 'center','FontSize',12);

            cID = obj.CellID;

            if mod(cID, 2) == 0
                % x is even
                obj.side = 'l';
            else
                % x is odd
                obj.side = 'r';
            end


%             if cID ~= floor(cID)
%                 integ=floor(cID);
%                 fract=fix((cID-integ)*100);
%                 if mod(fract, 2) == 0
%                     obj.side = 'l';%x is even
%                 else
%                     obj.side = 'r';%x is odd
%
```

```
%                   end
%               end
%



            if strcmp(obj.side, 'r') || obj.CellID <= 5
                if(pol > 0)
                    %color = [abs(pol) 0 0];
                    color = [1-abs(pol) 1 1-abs(pol)];
                elseif(pol < 0)
                    %color = [0 abs(pol) 0];
                    color = [1 1-abs(pol) 1-abs(pol)];
                else
                    color = [1 1 1];
                end
            else
                if(pol < 0)
                    %color = [abs(pol) 0 0];
                    color = [1-abs(pol) 1 1-abs(pol)];
                elseif(pol > 0)
                    %color = [0 abs(pol) 0];
                    color = [1 1-abs(pol) 1-abs(pol)];
                else
                    color = [1 1 1];
                end
            end
```

```
end

function obj = BoxDraw(obj)
    obj.SelectBox=patch;
    obj.SelectBox.XData=[obj.CenterPosition(1)-.25;obj.
       CenterPosition(1)+.25;obj.CenterPosition(1)+.25;
       obj.CenterPosition(1)-.25];
    obj.SelectBox.YData=[obj.CenterPosition(2)-.75;obj.
       CenterPosition(2)-.75;obj.CenterPosition(2)+.75;
       obj.CenterPosition(2)+.75];
    obj.SelectBox.EdgeColor = 'None';

    obj.CellID;
    obj.Overlapping;
    switch obj.Overlapping
        case 'off'
            obj.SelectBox.FaceColor=[1 1 1];
            obj.SelectBox.FaceAlpha = .01;
        case 'on'
            obj.SelectBox.FaceColor=[1 0 0];
            obj.SelectBox.FaceAlpha = .4;

    end

    obj.SelectBox.UserData = obj.CellID;
end
```

```
function obj = ElectronDraw(obj, time, varargin)

        targetAxes = [];
        a= obj.CharacteristicLength;
        r= obj.CenterPosition;
        act = obj.Activation;
        pol = obj.getPolarization(time);
        radiusfactor = 0.2;



        if length(varargin)==1
            targetAxes = varargin{1};
            hold off;

        end



        %Tunnel junctions
        x_dist = [obj.CenterPosition(1), obj.
           CenterPosition(1)];
        y_dist13 = [obj.CenterPosition(2)+a*.5, obj.
           CenterPosition(2)-a*.4];
        l13 = line(x_dist, y_dist13, 'LineWidth', 2, '
           Color', [0 0 0]);

        %Print CellID
        %text(obj.CenterPosition(1), obj.CenterPosition
           (2)+.8*a, num2str(obj.CellID), '
```

```
    HorizontalAlignment', 'center','FontSize
    ',12);

%extra circle
%c123 = circle(obj.CenterPosition(1), obj.
    CenterPosition(2), obj.radiusOfEffect, [1 1
    1], 'EdgeColor', [0 0 0], 'Points',25);
%c123.FaceColor = 'None';



%Electron Sites
c1 = circle(obj.CenterPosition(1), obj.
    CenterPosition(2), a*radiusfactor, [1 1 1],'
    Points',25, 'TargetAxes', targetAxes);
c2 = circle(obj.CenterPosition(1), obj.
    CenterPosition(2)+a*.5, a*radiusfactor, [1 1
     1],'Points',25, 'TargetAxes', targetAxes);
c3 = circle(obj.CenterPosition(1), obj.
    CenterPosition(2)-a*.5, a*radiusfactor, [1 1
     1],'Points',25, 'TargetAxes', targetAxes);


%electron color for driver/node
if strcmp(obj.Type,'Node')
     electronColor = [1 0 0];
elseif strcmp(obj.Type,'Driver')
     electronColor = [0 1 0];
```

```
end

%Electrons Position Probability
q0 = (act/2)*(1-pol);
qN = 1 - act;
q1 = (act/2)*(1+pol);

scalefactor = 0.90;
e0 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)+a*.5, q0*a*radiusfactor*
   scalefactor, electronColor,'EdgeColor', '
   None','Points',25, 'TargetAxes', targetAxes)
   ;
eN = circle(obj.CenterPosition(1), obj.
   CenterPosition(2),      qN*a*radiusfactor*
   scalefactor, electronColor,'EdgeColor', '
   None','Points',25, 'TargetAxes', targetAxes)
   ;
e1 = circle(obj.CenterPosition(1), obj.
   CenterPosition(2)-a*.5, q1*a*radiusfactor*
   scalefactor, electronColor,'EdgeColor', '
   None','Points',25, 'TargetAxes', targetAxes)
   ;
```

```
        end
    end
end

% make a neighbor list.
%QCACiruit: assign neighbors (based on center-center distance)
%testing just assign neightbors
% QCACell
%NeighborList [unique cell ID's]
```

Listing A.55. ThreeDotCell Class

```
classdef Transition
    %UNTITLED Summary of this class goes here
    %   Detailed explanation goes here

    properties
        Type; %linear, fermi, random?
        Initial=0;  %x0
        Final;      %x1
        CenterPoint;
        Alpha;  %y0
        Beta;   %y1

    end

    methods
        function obj = Transition(varargin)
```

```
        end



    end

end
```

Listing A.56. Transition Function

```
function [h,yy,zz] = arrow(varargin)
% ARROW  Draw a line with an arrowhead.
%
%  ARROW(Start,Stop) draws a line with an arrow from Start to
   Stop (points
%        should be vectors of length 2 or 3, or matrices with 2
    or 3
%        columns), and returns the graphics handle of the arrow
   (s).
%
%  ARROW uses the mouse (click-drag) to create an arrow.
%
%  ARROW DEMO & ARROW DEMO2 show 3-D & 2-D demos of the
   capabilities of ARROW.
%
%  ARROW may be called with a normal argument list or a
   property-based list.
```

```
%       ARROW(Start,Stop,Length,BaseAngle,TipAngle,Width,Page,
   CrossDir) is
%       the full normal argument list, where all but the Start
    and Stop
%       points are optional.  If you need to specify a later
   argument (e.g.,
%       Page) but want default values of earlier ones (e.g.,
   TipAngle),
%       pass an empty matrix for the earlier ones (e.g.,
   TipAngle=[]).
%
%  ARROW('Property1',PropVal1,'Property2',PropVal2,...) creates
    arrows with the
%       given properties, using default values for any
   unspecified or given as
%       'default' or NaN.  Some properties used for line and
   patch objects are
%       used in a modified fashion, others are passed directly
    to LINE, PATCH,
%       or SET.  For a detailed properties explanation, call
   ARROW PROPERTIES.
%
%       Start          The starting points.
                       B
%       Stop           The end points.
   /|\            ^
%       Length         Length of the arrowhead in pixels.     /
   |||\           |
```

```
%        BaseAngle     Base angle in degrees (ADE).          //
   |||||\        L|
%        TipAngle      Tip angle in degrees (ABC).          ///
   ||||||\       e|
%        Width         Width of the base in pixels.        ////
   |||||||\      n|
%        Page          Use hardcopy proportions.          /////
   |D||||||\     g|
%        CrossDir      Vector || to arrowhead plane.     ////
   |||  \\\\    t|
%        NormalDir     Vector out of arrowhead plane.   ///
   |||    \\\   h|
%        Ends          Which end has an arrowhead.     //<-
   ---->||      \\   /
%        ObjectHandles Vector of handles to update.   /   base
   |||        \  V
%                                                    E    angle
   ||<--------->C
%  ARROW(H,'Prop1',PropVal1,...), where H is a
   |||tipangle
%        vector of handles to previously-created arrows
   |||
%        and/or line objects, will update the previously-
   |||
%        created arrows according to the current view
   -->|A|<-- width
%        and any specified properties, and will convert
```

```
%       two-point line objects to corresponding arrows.  ARROW
   (H) will update
%       the arrows if the current view has changed.  Root,
   figure, or axes
%       handles included in H are replaced by all descendant
   Arrow objects.
%
%  A property list can follow any specified normal argument
   list, e.g.,
%  ARROW([1 2 3],[0 0 0],36,'BaseAngle',60) creates an arrow
   from (1,2,3) to
%  the origin, with an arrowhead of length 36 pixels and 60-
   degree base angle.
%
%  Normally, an ARROW is a PATCH object, so any valid PATCH
   property/value pairs
%  can be passed, e.g., ARROW(Start,Stop,'EdgeColor','r','
   FaceColor','g').
%  ARROW will use LINE objects when requested by ARROW(...,'
   Type','line') or,
%  using LINE property/value pairs, ARROW(Start,Stop,'Type','
   line','Color','b').
%
%  The basic arguments or properties can generally be
   vectorized to create
%  multiple arrows with the same call.  This is done by passing
    a property
```

```
%  with one row per arrow, or, if all arrows are to have the
   same property
%  value, just one row may be specified.
%
%  You may want to execute AXIS(AXIS) before calling ARROW so
   it doesn't change
%  the axes on you; ARROW determines the sizes of arrow
   components BEFORE the
%  arrow is plotted, so if ARROW changes axis limits, arrows
   may be malformed.
%
%  This version of ARROW uses features of MATLAB 6.x and is
   incompatible with
%  earlier MATLAB versions (ARROW for MATLAB 4.2c is available
   separately);
%  some problems with perspective plots still exist.

% Copyright (c)1995-2016, Dr. Erik A. Johnson <JohnsonE@usc.edu
   >, 5/25/2016
% http://www.usc.edu/civil_eng/johnsone/

% Revision history:
%    5/25/16  EAJ  Add documentation of 'Type','line'
%                  Add documentation of how to set color
%                  Add 'Color' property (which sets both '
   EdgeColor' and 'FaceColor' for patch objects)
%    5/24/16  EAJ  Remove 'EraseMode' in HG2
%    7/16/14  EAJ  R2014b HandleGraphics2 compatibility
```

```
%    7/14/14  EAJ  5/20/13 patch extension didn't work right in
    HG2
%                  so break the arrow along its length
   instead
%    5/20/13  EAJ  Extend patch line one more segment so EPS/
   PDF printed versions
%                  have nice rounded tips when the LineWidth
   is wider
%    2/06/13  EAJ  Add ShortenLength property to shorten length
    if arrow is short
%    1/24/13  EAJ  Remove some old comments.
%    5/20/09  EAJ  Fix view direction in (3D) demo.
%    6/26/08  EAJ  Replace eval('trycmd','catchcmd') with try,
   trycmd; catch,
%                  catchcmd; end; -- break's MATLAB 5
   compatibility.
%    8/26/03  EAJ  Eliminate OpenGL attempted fix since it didn
   't fix anyway.
%   11/15/02  EAJ  Accomodate how MATLAB 6.5 handles NaN and
   logicals
%    7/28/02  EAJ  Tried (but failed) work-around for MATLAB 6.
   x / OpenGL bug
%                  if zero 'Width' or not double-ended
%   11/10/99  EAJ  Add logical() to eliminate zero index
   problem in MATLAB 5.3.
%   11/10/99  EAJ  Corrected warning if axis limits changed on
   multiple axes.
%   11/10/99  EAJ  Update e-mail address.
```

```
%    2/10/99  EAJ  Some documentation updating.
%    2/24/98  EAJ  Fixed bug if Start~=Stop but both colinear
   with viewpoint.
%    8/14/97  EAJ  Added workaround for MATLAB 5.1 scalar
   logical transpose bug.
%    7/21/97  EAJ  Fixed a few misc bugs.
%    7/14/97  EAJ  Make arrow([],'Prop',...) do nothing (no old
    handles)
%    6/23/97  EAJ  MATLAB 5 compatible version, release.
%    5/27/97  EAJ  Added Line Arrows back in.  Corrected a few
   bugs.
%    5/26/97  EAJ  Changed missing Start/Stop to mouse-selected
    arrows.
%    5/19/97  EAJ  MATLAB 5 compatible version, beta.
%    4/13/97  EAJ  MATLAB 5 compatible version, alpha.
%    1/31/97  EAJ  Fixed bug with multiple arrows and
   unspecified Z coords.
%   12/05/96  EAJ  Fixed one more bug with log plots and
   NormalDir specified
%   10/24/96  EAJ  Fixed bug with log plots and NormalDir
   specified
%   11/13/95  EAJ  Corrected handling for 'reverse' axis
   directions
%   10/06/95  EAJ  Corrected occasional conflict with SUBPLOT
%    4/24/95  EAJ  A major rewrite.
%    Fall 94  EAJ  Original code.

% Things to be done:
```

```
%  - in the arrow_clicks section, prompt by printing to the
   screen so that
%    the user knows what's going on; also make sure the figure
   is brought
%    to the front.
%  - segment parsing, computing, and plotting into separate
   subfunctions
%  - change computing from Xform to Camera paradigms
%     + this will help especially with 3-D perspective plots
%     + if the WarpToFill section works right, remove warning
   code
%     + when perpsective works properly, remove perspective
   warning code
%  - add cell property values and struct property name/values (
   like get/set)
%  - get rid of NaN as the "default" data label
%     + perhaps change userdata to a struct and don't include (
   or leave
%       empty) the values specified as default; or use a cell
   containing
%       an empty matrix for a default value
%  - add functionality of GET to retrieve current values of
   ARROW properties
%
% New list of things to be done:
%  - rewrite as a graphics or class object that updates itself
   in real time
```

```
%    (but have a 'Static' or 'DoNotUpdate' property to avoid
   updating)

% Permission is granted to distribute ARROW with the toolboxes
   for the book
% "Solving Solid Mechanics Problems with MATLAB 5", by F.
   Golnaraghi et al.
% (Prentice Hall, 1999).

% Permission is granted to Dr. Josef Bigun to distribute ARROW
   with his
% software to reproduce the figures in his image analysis text.

% global variable initialization
persistent ARROW_PERSP_WARN ARROW_STRETCH_WARN ARROW_AXLIMITS
   ARROW_AX
if isempty(ARROW_PERSP_WARN  ), ARROW_PERSP_WARN  =1; end;
if isempty(ARROW_STRETCH_WARN), ARROW_STRETCH_WARN=1; end;

% Handle callbacks
if (nargin>0 & isstr(varargin{1}) & strcmp(lower(varargin{1}),'
   callback')),
  arrow_callback(varargin{2:end}); return;
end;

% Are we doing the demo?
c = sprintf('\n');
if (nargin==1 & isstr(varargin{1})),
```

```
  arg1 = lower(varargin{1});
  if strncmp(arg1,'prop',4), arrow_props;
  elseif strncmp(arg1,'demo',4)
    clf reset
    demo_info = arrow_demo;
    if ~strncmp(arg1,'demo2',5),
      hh=arrow_demo3(demo_info);
    else,
      hh=arrow_demo2(demo_info);
    end;
    if (nargout>=1), h=hh; end;
  elseif strncmp(arg1,'fixlimits',3),
    arrow_fixlimits(ARROW_AX,ARROW_AXLIMITS);
    ARROW_AXLIMITS=[]; ARROW_AX=[];
  elseif strncmp(arg1,'help',4),
    disp(help(mfilename));
  else,
    error([upper(mfilename) ' got an unknown single-argument
       string ''' deblank(arg1) '''.']);
  end;
  return;
end;

% Check # of arguments
if (nargout>3), error([upper(mfilename) ' produces at most 3
   output arguments.']); end;

% find first property number
```

```
firstprop = nargin+1;
for k=1:length(varargin), if ~isnumeric(varargin{k}) && ~all(
   ishandle(varargin{k})), firstprop=k; break; end; end; %eaj 5
   /24/16   for k=1:length(varargin), if ~isnumeric(varargin{k
   }), firstprop=k; break; end; end;
lastnumeric = firstprop-1;

% check property list
if (firstprop<=nargin),
  for k=firstprop:2:nargin,
    curarg = varargin{k};
    if ~isstr(curarg) | sum(size(curarg)>1)>1,
      error([upper(mfilename) ' requires that a property name
         be a single string.']);
    end;
  end;
  if (rem(nargin-firstprop,2)~=1),
    error([upper(mfilename) ' requires that the property '''
       ...
           varargin{nargin} ''' be paired with a property value
              .']);
  end;
end;

% default output
if (nargout>0), h=[]; end;
if (nargout>1), yy=[]; end;
if (nargout>2), zz=[]; end;
```

```
% set values to empty matrices
start      = [];
stop       = [];
len        = [];
baseangle  = [];
tipangle   = [];
wid        = [];
page       = [];
crossdir   = [];
ends       = [];
shorten    = [];
ax         = [];
oldh       = [];
ispatch    = [];
defstart      = [NaN NaN NaN];
defstop       = [NaN NaN NaN];
deflen        = 16;
defbaseangle  = 90;
deftipangle   = 16;
defwid        = 0;
defpage       = 0;
defcrossdir   = [NaN NaN NaN];
defends       = 1;
defshorten    = 0;
defoldh       = [];
defispatch    = 1;
```

```
% The 'Tag' we'll put on our arrows
ArrowTag = 'Arrow';

% check for oldstyle arguments
if (firstprop==2),
  % assume arg1 is a set of handles
  oldh = varargin{1}(:);
  if isempty(oldh), return; end;
elseif (firstprop>9),
  error([upper(mfilename) ' takes at most 8 non-property
     arguments.']);
elseif (firstprop>2),
  {start,stop,len,baseangle,tipangle,wid,page,crossdir};
  args = [varargin(1:firstprop-1) cell(1,length(ans)-(firstprop
     -1))];
  [start,stop,len,baseangle,tipangle,wid,page,crossdir] = deal(
     args{:});
end;

% parse property pairs
extraprops={};
for k=firstprop:2:nargin,
  prop = varargin{k};
  val  = varargin{k+1};
  prop = [lower(prop(:)') '      '];
  if     strncmp(prop,'start'  ,5),   start      = val;
  elseif strncmp(prop,'stop'   ,4),   stop       = val;
  elseif strncmp(prop,'len'    ,3),   len        = val(:);
```

```
elseif strncmp(prop,'base'   ,4),   baseangle  = val(:);
elseif strncmp(prop,'tip'    ,3),   tipangle   = val(:);
elseif strncmp(prop,'wid'    ,3),   wid        = val(:);
elseif strncmp(prop,'page'   ,4),   page       = val;
elseif strncmp(prop,'cross'  ,5),   crossdir   = val;
elseif strncmp(prop,'norm'   ,4),   if (isstr(val)), crossdir
   =val; else, crossdir=val*sqrt(-1); end;
elseif strncmp(prop,'end'    ,3),   ends       = val;
elseif strncmp(prop,'shorten',5),   shorten    = val;
elseif strncmp(prop,'object' ,6),   oldh       = val(:);
elseif strncmp(prop,'handle' ,6),   oldh       = val(:);
elseif strncmp(prop,'type'   ,4),   ispatch    = val;
elseif strncmp(prop,'userd'  ,5),   %ignore it
else,
  % make sure it is a valid patch or line property
  try
    get(0,['DefaultPatch' varargin{k}]);
  catch
    errstr = lasterr;
    try
      get(0,['DefaultLine' varargin{k}]);
    catch
      errstr(1:max(find(errstr==char(13)|errstr==char(10))))
         = '';
      error([upper(mfilename) ' got ' errstr]);
    end
  end;
  extraprops={extraprops{:},varargin{k},val};
```

```
  end;
end;

% Check if we got 'default' values
start     = arrow_defcheck(start    ,defstart    ,'Start'
          );
stop      = arrow_defcheck(stop     ,defstop     ,'Stop'
           );
len       = arrow_defcheck(len      ,deflen      ,'Length'
         );
baseangle = arrow_defcheck(baseangle,defbaseangle,'BaseAngle'
      );
tipangle  = arrow_defcheck(tipangle ,deftipangle ,'TipAngle'
       );
wid       = arrow_defcheck(wid      ,defwid      ,'Width'
          );
crossdir  = arrow_defcheck(crossdir ,defcrossdir ,'CrossDir'
       );
page      = arrow_defcheck(page     ,defpage     ,'Page'
           );
ends      = arrow_defcheck(ends     ,defends     ,''
               );
shorten   = arrow_defcheck(shorten  ,defshorten  ,''
               );
oldh      = arrow_defcheck(oldh     ,[]          ,'
   ObjectHandles');
ispatch   = arrow_defcheck(ispatch  ,defispatch  ,''
               );
```

```
% check transpose on arguments
[m,n]=size(start   );   if any(m==[2 3])&(n==1|n>3),   start
      = start';       end;
[m,n]=size(stop    );   if any(m==[2 3])&(n==1|n>3),   stop
       = stop';        end;
[m,n]=size(crossdir);   if any(m==[2 3])&(n==1|n>3),   crossdir
    = crossdir';   end;

% convert strings to numbers
if ~isempty(ends) & isstr(ends),
  endsorig = ends;
  [m,n] = size(ends);
  col = lower([ends(:,1:min(3,n)) ones(m,max(0,3-n))*' ']);
  ends = NaN*ones(m,1);
  oo = ones(1,m);
  ii=find(all(col'==['non']'*oo)'); if ~isempty(ii), ends(ii)=
     ones(length(ii),1)*0; end;
  ii=find(all(col'==['sto']'*oo)'); if ~isempty(ii), ends(ii)=
     ones(length(ii),1)*1; end;
  ii=find(all(col'==['sta']'*oo)'); if ~isempty(ii), ends(ii)=
     ones(length(ii),1)*2; end;
  ii=find(all(col'==['bot']'*oo)'); if ~isempty(ii), ends(ii)=
     ones(length(ii),1)*3; end;
  if any(isnan(ends)),
    ii = min(find(isnan(ends)));
    error([upper(mfilename) ' does not recognize ''' deblank(
       endsorig(ii,:)) ''' as a valid ''Ends'' value.']);
```

```
  end;
else,
  ends = ends(:);
end;
if ~isempty(ispatch) & isstr(ispatch),
  col = lower(ispatch(:,1));
  patchchar='p'; linechar='l'; defchar=' ';
  mask = col~=patchchar & col~=linechar & col~=defchar;
  if any(mask),
    error([upper(mfilename) ' does not recognize ''' deblank(
       ispatch(min(find(mask)),:)) ''' as a valid ''Type''
       value.']);
  end;
  ispatch = (col==patchchar)*1 + (col==linechar)*0 + (col==
     defchar)*defispatch;
else,
  ispatch = ispatch(:);
end;
oldh = oldh(:);

% check object handles
if ~all(ishandle(oldh)), error([upper(mfilename) ' got invalid
   object handles.']); end;

% expand root, figure, and axes handles
if ~isempty(oldh),
  ohtype = get(oldh,'Type');
```

```
  mask = strcmp(ohtype,'root') | strcmp(ohtype,'figure') |
     strcmp(ohtype,'axes');
  if any(mask),
    oldh = num2cell(oldh);
    for ii=find(mask)',
      oldh(ii) = {findobj(oldh{ii},'Tag',ArrowTag)};
    end;
    oldh = cat(1,oldh{:});
    if isempty(oldh), return; end; % no arrows to modify, so
       just leave
  end;
end;

% largest argument length
[mstart,junk]=size(start); [mstop,junk]=size(stop); [mcrossdir,
   junk]=size(crossdir);
argsizes = [length(oldh) mstart mstop
                                  ...
            length(len) length(baseangle) length(tipangle)
                         ...
      length(wid) length(page) mcrossdir length(ends) length(
         shorten)];
args=['length(ObjectHandle)  '; ...
      '#rows(Start)          '; ...
      '#rows(Stop)           '; ...
      'length(Length)        '; ...
      'length(BaseAngle)     '; ...
      'length(TipAngle)      '; ...
```

```
      'length(Width)          '; ...
      'length(Page)           '; ...
      '#rows(CrossDir)        '; ...
      '#rows(Ends)            '; ...
      'length(ShortenLength) '];
if (any(imag(crossdir(:))~=0)),
  args(9,:) = '#rows(NormalDir)      ';
end;
if isempty(oldh),
  narrows = max(argsizes);
else,
  narrows = length(oldh);
end;
if (narrows<=0), narrows=1; end;

% Check size of arguments
ii = find((argsizes~=0)&(argsizes~=1)&(argsizes~=narrows));
if ~isempty(ii),
  s = args(ii',:);
  while ((size(s,2)>1)&((abs(s(:,size(s,2)))==0)|(abs(s(:,size(
     s,2)))==abs(' '')))),
    s = s(:,1:size(s,2)-1);
  end;
  s = [ones(length(ii),1)*[upper(mfilename) ' requires that  ']
      s ...
       ones(length(ii),1)*['  equal the # of arrows (' num2str(
          narrows) ').' c]];
  s = s';
```

```
  s = s(:)';
  s = s(1:length(s)-1);
  error(setstr(s));
end;

% check element length in Start, Stop, and CrossDir
if ~isempty(start),
  [m,n] = size(start);
  if (n==2),
    start = [start NaN*ones(m,1)];
  elseif (n~=3),
    error([upper(mfilename) ' requires 2- or 3-element Start
       points.']);
  end;
end;
if ~isempty(stop),
  [m,n] = size(stop);
  if (n==2),
    stop = [stop NaN*ones(m,1)];
  elseif (n~=3),
    error([upper(mfilename) ' requires 2- or 3-element Stop
       points.']);
  end;
end;
if ~isempty(crossdir),
  [m,n] = size(crossdir);
  if (n<3),
    crossdir = [crossdir NaN*ones(m,3-n)];
```

```
  elseif (n~=3),
    if (all(imag(crossdir(:))==0)),
      error([upper(mfilename) ' requires 2- or 3-element
         CrossDir vectors.']);
    else,
      error([upper(mfilename) ' requires 2- or 3-element
         NormalDir vectors.']);
    end;
  end;
end;

% fill empty arguments
if isempty(start     ),   start      = [Inf Inf Inf];      end;
if isempty(stop      ),   stop       = [Inf Inf Inf];      end;
if isempty(len       ),   len        = Inf;                end;
if isempty(baseangle ),   baseangle  = Inf;                end;
if isempty(tipangle  ),   tipangle   = Inf;                end;
if isempty(wid       ),   wid        = Inf;                end;
if isempty(page      ),   page       = Inf;                end;
if isempty(crossdir  ),   crossdir   = [Inf Inf Inf];      end;
if isempty(ends      ),   ends       = Inf;                end;
if isempty(shorten   ),   shorten    = Inf;                end;
if isempty(ispatch   ),   ispatch    = Inf;                end;

% expand single-column arguments
o = ones(narrows,1);
if (size(start     ,1)==1),   start      = o * start     ;
   end;
```

```
if (size(stop      ,1)==1),   stop       = o * stop      ;
   end;
if (length(len       )==1),   len        = o * len       ;
   end;
if (length(baseangle )==1),   baseangle  = o * baseangle ;
   end;
if (length(tipangle  )==1),   tipangle   = o * tipangle  ;
   end;
if (length(wid       )==1),   wid        = o * wid       ;
   end;
if (length(page      )==1),   page       = o * page      ;
   end;
if (size(crossdir  ,1)==1),   crossdir   = o * crossdir  ;
   end;
if (length(ends      )==1),   ends       = o * ends      ;
   end;
if (length(shorten   )==1),   shorten    = o * shorten   ;
   end;
if (length(ispatch   )==1),   ispatch    = o * ispatch   ;
   end;
ax = repmat(gca,narrows,1);   %eaj 7/16/14  ax=gca; if ~
   isnumeric(ax), ax=double(ax); end; ax=o*ax;

% if we've got handles, get the defaults from the handles
if ~isempty(oldh),
  for k=1:narrows,
    oh = oldh(k);
    ud = get(oh,'UserData');
```

```
ax(k) = get(oh,'Parent');   %eaj 7/16/14  get(oh,'Parent');
    if ~isnumeric(ans), double(ans); end; ax(k)=ans;
ohtype = get(oh,'Type');
if strcmp(get(oh,'Tag'),ArrowTag), % if it's an arrow
   already
  if isinf(ispatch(k)), ispatch(k)=strcmp(ohtype,'patch');
     end;
  % arrow UserData format: [start' stop' len base tip wid
     page crossdir' ends shorten]
  start0 = ud(1:3);
  stop0  = ud(4:6);
  if (isinf(len(k))),           len(k)        = ud( 7);
     end;
  if (isinf(baseangle(k))),     baseangle(k)  = ud( 8);
     end;
  if (isinf(tipangle(k))),      tipangle(k)   = ud( 9);
     end;
  if (isinf(wid(k))),           wid(k)        = ud(10);
     end;
  if (isinf(page(k))),          page(k)       = ud(11);
     end;
  if (isinf(crossdir(k,1))),    crossdir(k,1) = ud(12);
     end;
  if (isinf(crossdir(k,2))),    crossdir(k,2) = ud(13);
     end;
  if (isinf(crossdir(k,3))),    crossdir(k,3) = ud(14);
     end;
```

```
      if (isinf(ends(k))),          ends(k)       = ud(15);
         end;
      if (isinf(shorten(k))),       shorten(k)    = ud(16);
         end;
    elseif strcmp(ohtype,'line')|strcmp(ohtype,'patch'), % it's
        a non-arrow line or patch
      convLineToPatch = 1; %set to make arrow patches when
         converting from lines.
      if isinf(ispatch(k)), ispatch(k)=convLineToPatch|strcmp(
         ohtype,'patch'); end;
      x=get(oh,'XData');  x=x(~isnan(x(:)));  if isempty(x), x=
         NaN; end;
      y=get(oh,'YData');  y=y(~isnan(y(:)));  if isempty(y), y=
         NaN; end;
      z=get(oh,'ZData');  z=z(~isnan(z(:)));  if isempty(z), z=
         NaN; end;
      start0 = [x(1)   y(1)   z(1)  ];
      stop0  = [x(end) y(end) z(end)];
    else,
      error([upper(mfilename) ' cannot convert ' ohtype '
         objects.']);
    end;
    ii=find(isinf(start(k,:)));  if ~isempty(ii),  start(k,ii)=
       start0(ii);  end;
    ii=find(isinf(stop( k,:)));  if ~isempty(ii),  stop( k,ii)=
       stop0( ii);  end;
  end;
end;
```

```
% convert Inf's to NaN's
start(     isinf(start    )) = NaN;
stop(      isinf(stop     )) = NaN;
len(       isinf(len      )) = NaN;
baseangle( isinf(baseangle)) = NaN;
tipangle(  isinf(tipangle )) = NaN;
wid(       isinf(wid      )) = NaN;
page(      isinf(page     )) = NaN;
crossdir(  isinf(crossdir )) = NaN;
ends(      isinf(ends     )) = NaN;
shorten(   isinf(shorten  )) = NaN;
ispatch(   isinf(ispatch  )) = NaN;

% set up the UserData data (here so not corrupted by log10's
   and such)
ud = [start stop len baseangle tipangle wid page crossdir ends
   shorten];

% Set Page defaults
page = ~isnan(page) & trueornan(page);

% Get axes limits, range, min; correct for aspect ratio and log
    scale
axm       = zeros(3,narrows);
axr       = zeros(3,narrows);
axrev     = zeros(3,narrows);
ap        = zeros(2,narrows);
```

```
xyzlog    = zeros(3,narrows);
limmin    = zeros(2,narrows);
limrange  = zeros(2,narrows);
oldaxlims = zeros(6,narrows);
oneax = all(ax==ax(1));
if (oneax),
  T    = zeros(4,4);
  invT = zeros(4,4);
else,
  T    = zeros(16,narrows);
  invT = zeros(16,narrows);
end;
axnotdone = true(size(ax));
while (any(axnotdone)),
  ii = find(axnotdone,1);
  curax = ax(ii);
  curpage = page(ii);
  % get axes limits and aspect ratio
  axl = [get(curax,'XLim'); get(curax,'YLim'); get(curax,'ZLim'
     )];
  ax==curax; oldaxlims(:,ans)=repmat(reshape(axl',[],1),1,sum(
     ans));
  % get axes size in pixels (points)
  u = get(curax,'Units');
  axposoldunits = get(curax,'Position');
  really_curpage = curpage & strcmp(u,'normalized');
  if (really_curpage),
    curfig = get(curax,'Parent');
```

```
  pu = get(curfig,'PaperUnits');
  set(curfig,'PaperUnits','points');
  pp = get(curfig,'PaperPosition');
  set(curfig,'PaperUnits',pu);
  set(curax,'Units','pixels');
  curapscreen = get(curax,'Position');
  set(curax,'Units','normalized');
  curap = pp.*get(curax,'Position');
else,
  set(curax,'Units','pixels');
  curapscreen = get(curax,'Position');
  curap = curapscreen;
end;
set(curax,'Units',u);
set(curax,'Position',axposoldunits);
% handle non-stretched axes position
str_stretch = { 'DataAspectRatioMode'    ; ...
                'PlotBoxAspectRatioMode' ; ...
                'CameraViewAngleMode'      };
str_camera  = { 'CameraPositionMode'  ; ...
                'CameraTargetMode'    ; ...
                'CameraViewAngleMode' ; ...
                'CameraUpVectorMode'    };
notstretched = strcmp(get(curax,str_stretch),'manual');
manualcamera = strcmp(get(curax,str_camera),'manual');
if ~arrow_WarpToFill(notstretched,manualcamera,curax),
  % give a warning that this has not been thoroughly tested
  if 0 & ARROW_STRETCH_WARN,
```

```
  ARROW_STRETCH_WARN = 0;
  strs = {str_stretch{1:2},str_camera{:}};
  strs = [char(ones(length(strs),1)*sprintf('\n    ')) char
     (strs)]';
  warning([upper(mfilename) ' may not yet work quite right
     ' ...
           'if any of the following are ''manual'':' strs
              (:).']);
end;
% find the true pixel size of the actual axes
texttmp = text(axl(1,[1 2 2 1 1 2 2 1]), ...
               axl(2,[1 1 2 2 1 1 2 2]), ...
               axl(3,[1 1 1 1 2 2 2 2]),'');
set(texttmp,'Units','points');
textpos = get(texttmp,'Position');
delete(texttmp);
textpos = cat(1,textpos{:});
textpos = max(textpos(:,1:2)) - min(textpos(:,1:2));
% adjust the axes position
if (really_curpage),
  % adjust to printed size
  textpos = textpos * min(curap(3:4)./textpos);
  curap = [curap(1:2)+(curap(3:4)-textpos)/2 textpos];
else,
  % adjust for pixel roundoff
  textpos = textpos * min(curapscreen(3:4)./textpos);
  curap = [curap(1:2)+(curap(3:4)-textpos)/2 textpos];
end;
```

```
end;
if ARROW_PERSP_WARN & ~strcmp(get(curax,'Projection'),'
   orthographic'),
  ARROW_PERSP_WARN = 0;
  warning([upper(mfilename) ' does not yet work right for 3-D
      perspective projection.']);
end;
% adjust limits for log scale on axes
curxyzlog = strcmp(get(curax,{'XScale' 'YScale' 'ZScale'})','
   log');
if (any(curxyzlog)),
  ii = find([curxyzlog;curxyzlog]);
  if (any(axl(ii)<=0)),
    error([upper(mfilename) ' does not support non-positive
       limits on log-scaled axes.']);
  else,
    axl(ii) = log10(axl(ii));
  end;
end;
% correct for 'reverse' direction on axes;
curreverse = strcmp(get(curax,{'XDir' 'YDir' 'ZDir'})','
   reverse');
ii = find(curreverse);
if ~isempty(ii),
  axl(ii,[1 2])=-axl(ii,[2 1]);
end;
% compute the range of 2-D values
```

```
try, curT=get(curax,'Xform'); catch, num2cell(get(curax,'View
   ')); curT=viewmtx(ans{:}); end;
lim = curT*[0 1 0 1 0 1 0 1;0 0 1 1 0 0 1 1;0 0 0 0 1 1 1 1;1
    1 1 1 1 1 1 1];
lim = lim(1:2,:)./([1;1]*lim(4,:));
curlimmin = min(lim')';
curlimrange = max(lim')' - curlimmin;
curinvT = inv(curT);
if (~oneax),
  curT = curT.';
  curinvT = curinvT.';
  curT = curT(:);
  curinvT = curinvT(:);
end;
% check which arrows to which cur corresponds
ii = find((ax==curax)&(page==curpage));
oo = ones(1,length(ii));
axr(:,ii)      = diff(axl')' * oo;
axm(:,ii)      = axl(:,1)    * oo;
axrev(:,ii)    = curreverse  * oo;
ap(:,ii)       = curap(3:4)' * oo;
xyzlog(:,ii)   = curxyzlog   * oo;
limmin(:,ii)   = curlimmin   * oo;
limrange(:,ii) = curlimrange * oo;
if (oneax),
  T    = curT;
  invT = curinvT;
else,
```

```
    T(:,ii)    = curT    * oo;
    invT(:,ii) = curinvT * oo;
  end;
  axnotdone(ii) = zeros(1,length(ii));
end;

% correct for log scales
curxyzlog = xyzlog.';
ii = find(curxyzlog(:));
if ~isempty(ii),
  start(   ii) = real(log10(start(   ii)));
  stop(    ii) = real(log10(stop(    ii)));
  if (all(imag(crossdir)==0)), % pulled (ii) subscript on
     crossdir, 12/5/96 eaj
    crossdir(ii) = real(log10(crossdir(ii)));
  end;
end;

% correct for reverse directions
ii = find(axrev.');
if ~isempty(ii),
  start(   ii) = -start(   ii);
  stop(    ii) = -stop(    ii);
  crossdir(ii) = -crossdir(ii);
end;

% transpose start/stop values
start     = start.';
```

```
stop      = stop.';

% take care of defaults, page was done above
ii=find(isnan(start(:)       ));  if ~isempty(ii),  start(ii)
         = axm(ii)+axr(ii)/2;                end;
ii=find(isnan(stop(:)        ));  if ~isempty(ii),  stop(ii)
          = axm(ii)+axr(ii)/2;                end;
ii=find(isnan(crossdir(:)    ));  if ~isempty(ii),  crossdir(ii
   )    = zeros(length(ii),1);              end;
ii=find(isnan(len            ));  if ~isempty(ii),  len(ii)
           = ones(length(ii),1)*deflen;        end;
ii=find(isnan(baseangle      ));  if ~isempty(ii),  baseangle(
   ii)   = ones(length(ii),1)*defbaseangle;  end;
ii=find(isnan(tipangle       ));  if ~isempty(ii),  tipangle(ii
   )    = ones(length(ii),1)*deftipangle;   end;
ii=find(isnan(wid            ));  if ~isempty(ii),  wid(ii)
           = ones(length(ii),1)*defwid;        end;
ii=find(isnan(ends           ));  if ~isempty(ii),  ends(ii)
          = ones(length(ii),1)*defends;       end;
ii=find(isnan(shorten        ));  if ~isempty(ii),  shorten(ii)
        = ones(length(ii),1)*defshorten;    end;

% transpose rest of values
len       = len.';
baseangle = baseangle.';
tipangle  = tipangle.';
wid       = wid.';
page      = page.';
```

```
crossdir  = crossdir.';
ends      = ends.';
shorten   = shorten.';
ax        = ax.';

% given x, a 3xN matrix of points in 3-space;
% want to convert to X, the corresponding 4xN 2-space matrix
%
%   tmp1=[(x-axm)./axr; ones(1,size(x,1))];
%   if (oneax), X=T*tmp1;
%   else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T.*tmp1;
%         tmp2=zeros(4,4*N); tmp2(:)=tmp1(:);
%         X=zeros(4,N); X(:)=sum(tmp2)'; end;
%   X = X ./ (ones(4,1)*X(4,:));

% for all points with start==stop, start=stop-(verysmallvalue)*
   (up-direction);
ii = find(all(start==stop));
if ~isempty(ii),
  % find an arrowdir vertical on screen and perpendicular to
     viewer
  % transform to 2-D
    tmp1 = [(stop(:,ii)-axm(:,ii))./axr(:,ii);ones(1,length(ii)
       )];
    if (oneax), twoD=T*tmp1;
    else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T(:,ii).*tmp1;
          tmp2=zeros(4,4*length(ii)); tmp2(:)=tmp1(:);
          twoD=zeros(4,length(ii)); twoD(:)=sum(tmp2)'; end;
```

```
    twoD=twoD./(ones(4,1)*twoD(4,:));
  % move the start point down just slightly
    tmp1 = twoD + [0;-1/1000;0;0]*(limrange(2,ii)./ap(2,ii));
  % transform back to 3-D
    if (oneax), threeD=invT*tmp1;
    else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT(:,ii).*tmp1;
          tmp2=zeros(4,4*length(ii)); tmp2(:)=tmp1(:);
          threeD=zeros(4,length(ii)); threeD(:)=sum(tmp2)'; end
             ;
    start(:,ii) = (threeD(1:3,:)./(ones(3,1)*threeD(4,:))).*axr
       (:,ii)+axm(:,ii);
end;

% compute along-arrow points
% transform Start points
  tmp1=[(start-axm)./axr;ones(1,narrows)];
  if (oneax), X0=T*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        X0=zeros(4,narrows); X0(:)=sum(tmp2)'; end;
  X0=X0./(ones(4,1)*X0(4,:));
% transform Stop points
  tmp1=[(stop-axm)./axr;ones(1,narrows)];
  if (oneax), Xf=T*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        Xf=zeros(4,narrows); Xf(:)=sum(tmp2)'; end;
  Xf=Xf./(ones(4,1)*Xf(4,:));
```

```
% compute pixel distance between points
  D = sqrt(sum(((Xf(1:2,:)-X0(1:2,:)).*(ap./limrange)).^2));
  D = D + (D==0);  %eaj new 2/24/98
%       shorten the length if requested % added 2/6/2013
  numends = (ends==1) + (ends==2) + 2*(ends==3);
  mask = shorten & D<len.*numends;
  len(mask) = D(mask) ./ numends(mask);
% compute and modify along-arrow distances
  len1 = len;
  len2 = len - (len.*tan(tipangle/180*pi)-wid/2).*tan((90-
     baseangle)/180*pi);
  slen0 = zeros(1,narrows);
  slen1 = len1 .* ((ends==2)|(ends==3));
  slen2 = len2 .* ((ends==2)|(ends==3));
  len0 = zeros(1,narrows);
  len1  = len1 .* ((ends==1)|(ends==3));
  len2  = len2 .* ((ends==1)|(ends==3));
  % for no start arrowhead
    ii=find((ends==1)&(D<len2));
    if ~isempty(ii),
      slen0(ii) = D(ii)-len2(ii);
    end;
  % for no end arrowhead
    ii=find((ends==2)&(D<slen2));
    if ~isempty(ii),
      len0(ii) = D(ii)-slen2(ii);
    end;
  len1  = len1  + len0;
```

```
  len2  = len2  + len0;
  slen1 = slen1 + slen0;
  slen2 = slen2 + slen0;
  % note:  the division by D below will probably not be
     accurate if both
  %        of the following are true:
  %           1. the ratio of the line length to the arrowhead
  %              length is large
  %           2. the view is highly perspective.
% compute stoppoints
  tmp1=X0.*(ones(4,1)*(len0./D))+Xf.*(ones(4,1)*(1-len0./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  stoppoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;
% compute tippoints
  tmp1=X0.*(ones(4,1)*(len1./D))+Xf.*(ones(4,1)*(1-len1./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  tippoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;
% compute basepoints
  tmp1=X0.*(ones(4,1)*(len2./D))+Xf.*(ones(4,1)*(1-len2./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
```

```
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  basepoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;
% compute startpoints
  tmp1=X0.*(ones(4,1)*(1-slen0./D))+Xf.*(ones(4,1)*(slen0./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  startpoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;
% compute stippoints
  tmp1=X0.*(ones(4,1)*(1-slen1./D))+Xf.*(ones(4,1)*(slen1./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  stippoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;
% compute sbasepoints
  tmp1=X0.*(ones(4,1)*(1-slen2./D))+Xf.*(ones(4,1)*(slen2./D));
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=invT.*tmp1;
        tmp2=zeros(4,4*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,narrows); tmp3(:)=sum(tmp2)'; end;
  sbasepoint = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)).*axr+axm;

% compute cross-arrow directions for arrows with NormalDir
   specified
if (any(imag(crossdir(:))~=0)),
  ii = find(any(imag(crossdir)~=0));
```

```
  crossdir(:,ii) = cross((stop(:,ii)-start(:,ii))./axr(:,ii),
     ...
                          imag(crossdir(:,ii))).*axr(:,ii);
end;

% compute cross-arrow directions
basecross  = crossdir + basepoint;
tipcross   = crossdir + tippoint;
sbasecross = crossdir + sbasepoint;
stipcross  = crossdir + stippoint;
ii = find(all(crossdir==0)|any(isnan(crossdir)));
if ~isempty(ii),
  numii = length(ii);
  % transform start points
    tmp1 = [basepoint(:,ii) tippoint(:,ii) sbasepoint(:,ii)
       stippoint(:,ii)];
    tmp1 = (tmp1-axm(:,[ii ii ii ii])) ./ axr(:,[ii ii ii ii]);
    tmp1 = [tmp1; ones(1,4*numii)];
    if (oneax), X0=T*tmp1;
    else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T(:,[ii ii ii ii]).*
       tmp1;
          tmp2=zeros(4,16*numii); tmp2(:)=tmp1(:);
          X0=zeros(4,4*numii); X0(:)=sum(tmp2)'; end;
    X0=X0./(ones(4,1)*X0(4,:));
  % transform stop points
    tmp1 = [(2*stop(:,ii)-start(:,ii)-axm(:,ii))./axr(:,ii);
       ones(1,numii)];
    tmp1 = [tmp1 tmp1 tmp1 tmp1];
```

```
  if (oneax), Xf=T*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=T(:,[ii ii ii ii]).*
     tmp1;
        tmp2=zeros(4,16*numii); tmp2(:)=tmp1(:);
        Xf=zeros(4,4*numii); Xf(:)=sum(tmp2)'; end;
  Xf=Xf./(ones(4,1)*Xf(4,:));
% compute perpendicular directions
  pixfact = ((limrange(1,ii)./limrange(2,ii)).*(ap(2,ii)./ap
     (1,ii))).^2;
  pixfact = [pixfact pixfact pixfact pixfact];
  pixfact = [pixfact;1./pixfact];
  [dummyval,jj] = max(abs(Xf(1:2,:)-X0(1:2,:)));
  jj1 = ((1:4)'*ones(1,length(jj))==ones(4,1)*jj);
  jj2 = ((1:4)'*ones(1,length(jj))==ones(4,1)*(3-jj));
  jj3 = jj1(1:2,:);
  Xf(jj1)=Xf(jj1)+(Xf(jj1)-X0(jj1)==0); %eaj new 2/24/98
  Xp = X0;
  Xp(jj2) = X0(jj2) + ones(sum(jj2(:)),1);
  Xp(jj1) = X0(jj1) - (Xf(jj2)-X0(jj2))./(Xf(jj1)-X0(jj1)) .*
      pixfact(jj3);
% inverse transform the cross points
  if (oneax), Xp=invT*Xp;
  else, tmp1=[Xp;Xp;Xp;Xp]; tmp1=invT(:,[ii ii ii ii]).*tmp1;
        tmp2=zeros(4,16*numii); tmp2(:)=tmp1(:);
        Xp=zeros(4,4*numii); Xp(:)=sum(tmp2)'; end;
  Xp=(Xp(1:3,:)./(ones(3,1)*Xp(4,:))).*axr(:,[ii ii ii ii])+
     axm(:,[ii ii ii ii]);
  basecross(:,ii)  = Xp(:,0*numii+(1:numii));
```

```
    tipcross(:,ii)   = Xp(:,1*numii+(1:numii));
    sbasecross(:,ii) = Xp(:,2*numii+(1:numii));
    stipcross(:,ii)  = Xp(:,3*numii+(1:numii));
end;

% compute all points
% compute start points
  axm11 = [axm axm axm axm axm axm axm axm axm axm axm];
  axr11 = [axr axr axr axr axr axr axr axr axr axr axr];
  st = [stoppoint tippoint basepoint sbasepoint stippoint
     startpoint stippoint sbasepoint basepoint tippoint
     stoppoint];
  tmp1 = (st - axm11) ./ axr11;
  tmp1 = [tmp1; ones(1,size(tmp1,2))];
  if (oneax), X0=T*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=[T T T T T T T T T T T
     ].*tmp1;
        tmp2=zeros(4,44*narrows); tmp2(:)=tmp1(:);
        X0=zeros(4,11*narrows); X0(:)=sum(tmp2)'; end;
  X0=X0./(ones(4,1)*X0(4,:));
% compute stop points
  tmp1 = ([start tipcross basecross sbasecross stipcross stop
     stipcross sbasecross basecross tipcross start] ...
       - axm11) ./ axr11;
  tmp1 = [tmp1; ones(1,size(tmp1,2))];
  if (oneax), Xf=T*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=[T T T T T T T T T T T
     ].*tmp1;
```

```
        tmp2=zeros(4,44*narrows); tmp2(:)=tmp1(:);
        Xf=zeros(4,11*narrows); Xf(:)=sum(tmp2)'; end;
  Xf=Xf./(ones(4,1)*Xf(4,:));
% compute lengths
  len0  = len.*((ends==1)|(ends==3)).*tan(tipangle/180*pi);
  slen0 = len.*((ends==2)|(ends==3)).*tan(tipangle/180*pi);
  le = [zeros(1,narrows) len0 wid/2 wid/2 slen0 zeros(1,narrows
     ) -slen0 -wid/2 -wid/2 -len0 zeros(1,narrows)];
  aprange = ap./limrange;
  aprange = [aprange aprange aprange aprange aprange aprange
     aprange aprange aprange aprange aprange];
  D = sqrt(sum(((Xf(1:2,:)-X0(1:2,:)).*aprange).^2));
  Dii=find(D==0); if ~isempty(Dii), D=D+(D==0); le(Dii)=zeros
     (1,length(Dii)); end; %should fix DivideByZero warnings
  tmp1 = X0.*(ones(4,1)*(1-le./D)) + Xf.*(ones(4,1)*(le./D));
% inverse transform
  if (oneax), tmp3=invT*tmp1;
  else, tmp1=[tmp1;tmp1;tmp1;tmp1]; tmp1=[invT invT invT invT
     invT invT invT invT invT invT invT].*tmp1;
        tmp2=zeros(4,44*narrows); tmp2(:)=tmp1(:);
        tmp3=zeros(4,11*narrows); tmp3(:)=sum(tmp2)'; end;
  pts = tmp3(1:3,:)./(ones(3,1)*tmp3(4,:)) .* axr11 + axm11;

% correct for ones where the crossdir was specified
ii = find(~(all(crossdir==0)|any(isnan(crossdir))));
if ~isempty(ii),
  D1 = [pts(:,1*narrows+ii)-pts(:,9*narrows+ii) ...
        pts(:,2*narrows+ii)-pts(:,8*narrows+ii) ...
```

```
        pts(:,3*narrows+ii)-pts(:,7*narrows+ii) ...
        pts(:,4*narrows+ii)-pts(:,6*narrows+ii) ...
        pts(:,6*narrows+ii)-pts(:,4*narrows+ii) ...
        pts(:,7*narrows+ii)-pts(:,3*narrows+ii) ...
        pts(:,8*narrows+ii)-pts(:,2*narrows+ii) ...
        pts(:,9*narrows+ii)-pts(:,1*narrows+ii)]/2;
  ii = ii'*ones(1,8) + ones(length(ii),1)*[1:4 6:9]*narrows;
  ii = ii(:)';
  pts(:,ii) = st(:,ii) + D1;
end;



% readjust for reverse directions
iicols=(1:narrows)'; iicols=iicols(:,ones(1,11)); iicols=iicols
   (:).';
tmp1=axrev(:,iicols);
ii = find(tmp1(:)); if ~isempty(ii), pts(ii)=-pts(ii); end;

% change from starting/ending at the stop point to doing it at
   the midpoint %eaj 7/14/2014
(pts(:,2*narrows+1:3*narrows)+pts(:,3*narrows+1:4*narrows))/2;
                %eaj 7/14/2014
pts = [ans pts(:,[3*narrows+1:end narrows+1:3*narrows]) ans];
                 %eaj 7/14/2014

% readjust for log scale on axes
tmp1=xyzlog(:,iicols);
ii = find(tmp1(:)); if ~isempty(ii), pts(ii)=10.^pts(ii); end;
```

```
% compute the x,y,z coordinates of the patches;
ii = narrows*(0:size(pts,2)/narrows-1)'*ones(1,narrows) + ones(
   size(pts,2)/narrows,1)*(1:narrows);
ii = ii(:)';
x = zeros(size(pts,2)/narrows,narrows);
y = zeros(size(pts,2)/narrows,narrows);
z = zeros(size(pts,2)/narrows,narrows);
x(:) = pts(1,ii)';
y(:) = pts(2,ii)';
z(:) = pts(3,ii)';

% do the output
if (nargout<=1),
% % create or modify the patches
  newpatch = trueornan(ispatch) & (isempty(oldh)|~strcmp(get(
     oldh,'Type'),'patch'));
  newline = ~trueornan(ispatch) & (isempty(oldh)|~strcmp(get(
     oldh,'Type'),'line'));
  if isempty(oldh), H=zeros(narrows,1); else, H=oldh; end;
% % make or modify the arrows
  for k=1:narrows,
    if all(isnan(ud(k,[3 6])))&arrow_is2DXY(ax(k)), zz=[]; else
       , zz=z(:,k); end;
    xx=x(:,k); yy=y(:,k);
    if (0), % this fix didn't work, so let's not use it -- 8/26
       /03
      % try to work around a MATLAB 6.x OpenGL bug -- 7/28/02
```

```
    mask=any([ones(1,2+size(zz,2));diff([xx yy zz],[],1)
       ],2);
    xx=xx(mask); yy=yy(mask); if ~isempty(zz), zz=zz(mask);
        end;
end;
% plot the patch or line
if newpatch(k) || trueornan(ispatch(k)) %eaj 7/14/2014, 5/
   25/2016
  % patch is closed so don't need endpoints %eaj 7/14/2014
  if ~isempty(xx), xx(end)=[]; end; %eaj 7/14/2014
  if ~isempty(yy), yy(end)=[]; end; %eaj 7/14/2014
  if ~isempty(zz), zz(end)=[]; end; %eaj 7/14/2014
end %eaj 7/14/2014
xyz = {'XData',xx,'YData',yy,'ZData',zz,'Tag',ArrowTag};
if newpatch(k)|newline(k),
  if newpatch(k),
    H(k) = patch(xyz{:});
  else,
    H(k) = line(xyz{:});
  end;
  if ~isempty(oldh), arrow_copyprops(oldh(k),H(k)); end;
else,
  if strcmp(get(H(k),'Type'),'patch') %eaj 5/25/16  if
     ispatch(k)
    xyz = {xyz{:},'CData',[]};
  end;
  set(H(k),xyz{:});
end;
```

```
  end;
  if ~isempty(oldh), delete(oldh(oldh~=H)); end;
% % additional properties
  set(H,'Clipping','off');
  set(H,{'UserData'},num2cell(ud,2));
  if length(extraprops)>0
    ii = find(strcmpi(extraprops(1:2:end),'color')); %eaj 5/25/
       16
    ispatch = strcmp(get(H,'Type'),'patch');
    %eaj start 5/25/16
      while ~isempty(ii) && any(ispatch)
        if ii>1, set(H,extraprops{1:2*ii-2}); end;
        c = extraprops{2*ii};
        extraprops(1:2*ii) = [];
        ii(1) = [];
        if all(ispatch) || ischar(c)&&size(c,1)==1 || isnumeric
           (c)&&isequal(size(c),[1 3])
          set(H,'EdgeColor',c,'FaceColor',c)
        elseif iscell(c) && numel(c)~=numel(H)
          set(H(ispatch),'EdgeColor',c(ispatch),'FaceColor',c(
             ispatch));
          set(H(~ispatch),'Color',c(~ispatch));
        elseif isnumeric(c) && isequal(size(c),[numel(H) 3])
          set(H(ispatch),'EdgeColor',num2cell(c(ispatch,:),2),'
             FaceColor',num2cell(c(ispatch,:),2));
          set(H(~ispatch),'Color',num2cell(c(~ispatch,:),2));
        else
```

```
        warning('ignoring unknown or invalid ''Color''
           specification');
      end
    end
  if ~isempty(extraprops)
  %eaj end   5/25/16
    set(H,extraprops{:});
  end %eaj   5/25/16
end
% handle choosing arrow Start and/or Stop locations if
   unspecified
[H,oldaxlims,errstr] = arrow_clicks(H,ud,x,y,z,ax,oldaxlims);
if ~isempty(errstr), error([upper(mfilename) ' got ' errstr])
   ; end;
% set the output
if (nargout>0), h=H; end;
% make sure the axis limits did not change
if isempty(oldaxlims),
  ARROW_AXLIMITS = [];
  ARROW_AX = [];
else,
  lims = get(ax(:),{'XLim','YLim','ZLim'})';
  lims = reshape(cat(2,lims{:}),6,size(lims,2));
  mask = arrow_is2DXY(ax(:));
  oldaxlims(5:6,mask) = lims(5:6,mask);
  % store them for possible restoring
  mask = any(oldaxlims~=lims,1); ARROW_AX=ax(mask); ARROW_
     AXLIMITS=oldaxlims(:,mask);
```

```
    if any(mask),
      warning(arrow_warnlimits(ARROW_AX,narrows));
    end;
  end;
else,
  % don't create the patch, just return the data
  h=x;
  yy=y;
  zz=z;
end;




function out = arrow_defcheck(in,def,prop)
% check if we got 'default' values
  out = in;
  if ~isstr(in), return; end;
  if size(in,1)==1 & strncmp(lower(in),'def',3),
    out = def;
  elseif ~isempty(prop),
    error([upper(mfilename) ' does not recognize ''' in(:)' '''
        as a valid ''' prop ''' string.']);
  end;




function [H,oldaxlims,errstr] = arrow_clicks(H,ud,x,y,z,ax,
   oldaxlims)
```

```
% handle choosing arrow Start and/or Stop locations if
   necessary
  errstr = '';
  if isempty(H)|isempty(ud)|isempty(x), return; end;
  % determine which (if any) need Start and/or Stop
  needStart = all(isnan(ud(:,1:3)'))';
  needStop  = all(isnan(ud(:,4:6)'))';
  mask = any(needStart|needStop);
  if ~any(mask), return; end;
  ud(~mask,:)=[]; ax(:,~mask)=[];
  x(:,~mask)=[]; y(:,~mask)=[]; z(:,~mask)=[];
  % make them invisible for the time being
  set(H,'Visible','off');
  % save the current axes and limits modes; set to manual for
     the time being
  oldAx  = gca;
  limModes=get(ax(:),{'XLimMode','YLimMode','ZLimMode'});
  set(ax(:),{'XLimMode','YLimMode','ZLimMode'},{'manual','
     manual','manual'});
  % loop over each arrow that requires attention
  jj = find(mask);
  for ii=1:length(jj),
    h = H(jj(ii));
    axes(ax(ii));
    % figure out correct call
    if needStart(ii), prop='Start'; else, prop='Stop'; end;
    [wasInterrupted,errstr] = arrow_click(needStart(ii)&
       needStop(ii),h,prop,ax(ii));
```

```
    % handle errors and control-C
    if wasInterrupted,
      delete(H(jj(ii:end)));
      H(jj(ii:end))=[];
      oldaxlims(jj(ii:end),:)=[];
      break;
    end;
  end;
  % restore the axes and limit modes
  axes(oldAx);
  set(ax(:),{'XLimMode','YLimMode','ZLimMode'},limModes);


function [wasInterrupted,errstr] = arrow_click(lockStart,H,prop
   ,ax)
% handle the clicks for one arrow
  fig = get(ax,'Parent');
  % save some things
  oldFigProps = {'Pointer','WindowButtonMotionFcn','
     WindowButtonUpFcn'};
  oldFigValue = get(fig,oldFigProps);
  oldArrowProps = {'EraseMode'};
  if ~isnumeric(fig), oldArrowProps={}; end %eaj 5/24/16 % only
      use in HG2
  oldArrowValue = get(H,oldArrowProps);
  if isnumeric(fig), %eaj 5/24/16
    set(H,'EraseMode','background'); %because 'xor' makes shaft
        invisible unless Width>1 -- only use in HG2
  end %eaj 5/24/16
```

```
global ARROW_CLICK_H ARROW_CLICK_PROP ARROW_CLICK_AX ARROW_
   CLICK_USE_Z
ARROW_CLICK_H=H; ARROW_CLICK_PROP=prop; ARROW_CLICK_AX=ax;
ARROW_CLICK_USE_Z=~arrow_is2DXY(ax)|~arrow_planarkids(ax);
set(fig,'Pointer','crosshair');
% set up the WindowButtonMotion so we can see the arrow while
    moving around
set(fig,'WindowButtonUpFcn','set(gcf,''WindowButtonUpFcn'','''
   '')', ...
        'WindowButtonMotionFcn','');
if ~lockStart,
  set(H,'Visible','on');
  set(fig,'WindowButtonMotionFcn',[mfilename '(''callback'','
     'motion'');']);
end;
% wait for the button to be pressed
[wasKeyPress,wasInterrupted,errstr] = arrow_wfbdown(fig);
% if we wanted to click-drag, set the Start point
if lockStart & ~wasInterrupted,
  pt = arrow_point(ARROW_CLICK_AX,ARROW_CLICK_USE_Z);
  feval(mfilename,H,'Start',pt,'Stop',pt);
  set(H,'Visible','on');
  ARROW_CLICK_PROP='Stop';
  set(fig,'WindowButtonMotionFcn',[mfilename '(''callback'','
     'motion'');']);
  % wait for the mouse button to be released
  try
    waitfor(fig,'WindowButtonUpFcn','');
```

```
    catch
      errstr = lasterr;
      wasInterrupted = 1;
    end;
  end;
  if ~wasInterrupted, feval(mfilename,'callback','motion'); end
     ;
  % restore some things
  set(gcf,oldFigProps,oldFigValue);
  set(H,oldArrowProps,oldArrowValue);


function arrow_callback(varargin)
% handle redrawing callbacks
  if nargin==0, return; end;
  str = varargin{1};
  if ~isstr(str), error([upper(mfilename) ' got an invalid
     Callback command.']); end;
  s = lower(str);
  if strcmp(s,'motion'),
    % motion callback
    global ARROW_CLICK_H ARROW_CLICK_PROP ARROW_CLICK_AX ARROW_
       CLICK_USE_Z
    feval(mfilename,ARROW_CLICK_H,ARROW_CLICK_PROP,arrow_point(
       ARROW_CLICK_AX,ARROW_CLICK_USE_Z));
    drawnow;
  else,
    error([upper(mfilename) ' does not recognize ''' str(:).' '
       '' as a valid Callback option.']);
```

```
  end;


function out = arrow_point(ax,use_z)
% return the point on the given axes
  if nargin==0, ax=gca; end;
  if nargin<2, use_z=~arrow_is2DXY(ax)|~arrow_planarkids(ax);
     end;
  out = get(ax,'CurrentPoint');
  out = out(1,:);
  if ~use_z, out=out(1:2); end;


function [wasKeyPress,wasInterrupted,errstr] = arrow_wfbdown(
   fig)
% wait for button down ignoring object ButtonDownFcn's
  if nargin==0, fig=gcf; end;
  errstr = '';
  % save ButtonDownFcn values
  objs = findobj(fig);
  buttonDownFcns = get(objs,'ButtonDownFcn');
  mask=~strcmp(buttonDownFcns,''); objs=objs(mask);
     buttonDownFcns=buttonDownFcns(mask);
  set(objs,'ButtonDownFcn','');
  % save other figure values
  figProps = {'KeyPressFcn','WindowButtonDownFcn'};
  figValue = get(fig,figProps);
  % do the real work
  set(fig,'KeyPressFcn','set(gcf,''KeyPressFcn'','''','''
     WindowButtonDownFcn'','''');', ...
```

```
          'WindowButtonDownFcn','set(gcf,''WindowButtonDownFcn'
             ',''''')');
  lasterr('');
  try
    waitfor(fig,'WindowButtonDownFcn','');
    wasInterrupted = 0;
  catch
    wasInterrupted = 1;
  end
  wasKeyPress = ~wasInterrupted & strcmp(get(fig,'KeyPressFcn')
     ,'');
  if wasInterrupted, errstr=lasterr; end;
  % restore ButtonDownFcn and other figure values
  set(objs,'ButtonDownFcn',buttonDownFcns);
  set(fig,figProps,figValue);



function [out,is2D] = arrow_is2DXY(ax)
% check if axes are 2-D X-Y plots
  % may not work for modified camera angles, etc.
  out = false(size(ax)); % 2-D X-Y plots
  is2D = out;                       % any 2-D plots
  views = get(ax(:),{'View'});
  views = cat(1,views{:});
  out(:) = abs(views(:,2))==90;
  is2D(:) = out(:) | all(rem(views',90)==0)';
```

```
function out = arrow_planarkids(ax)
% check if axes descendents all have empty ZData (lines,patches
   ,surfaces)
  out = true(size(ax));
  allkids = get(ax(:),{'Children'});
  for k=1:length(allkids),
    kids = get([findobj(allkids{k},'flat','Type','line')
                findobj(allkids{k},'flat','Type','patch')
                findobj(allkids{k},'flat','Type','surface')],{'
                   ZData'});
    for j=1:length(kids),
      if ~isempty(kids{j}), out(k)=logical(0); break; end;
    end;
  end;



function arrow_fixlimits(ax,lims)
% reset the axis limits as necessary
  if isempty(ax) || isempty(lims), disp([upper(mfilename) '
     does not remember any axis limits to reset.']); end;
  for k=1:numel(ax),
    if any(get(ax(k),'XLim')~=lims(1:2,k)'), set(ax(k),'XLim',
       lims(1:2,k)'); end;
    if any(get(ax(k),'YLim')~=lims(3:4,k)'), set(ax(k),'YLim',
       lims(3:4,k)'); end;
    if any(get(ax(k),'ZLim')~=lims(5:6,k)'), set(ax(k),'ZLim',
       lims(5:6,k)'); end;
```

```
  end;



function out = arrow_WarpToFill(notstretched,manualcamera,curax
   )
% check if we are in "WarpToFill" mode.
  out = strcmp(get(curax,'WarpToFill'),'on');
  % 'WarpToFill' is undocumented, so may need to replace this
     by
  % out = ~( any(notstretched) & any(manualcamera) );



function out = arrow_warnlimits(ax,narrows)
% create a warning message if we've changed the axis limits
  msg = '';
  switch (numel(ax))
    case 1, msg='';
    case 2, msg='on two axes ';
    otherwise, msg='on several axes ';
  end;
  msg = [upper(mfilename) ' changed the axis limits ' msg ...
         'when adding the arrow'];
  if (narrows>1), msg=[msg 's']; end;
  out = [msg '.' sprintf('\n') '         Call ' upper(mfilename
     ) ...
         ' FIXLIMITS to reset them now.'];
```

```
function arrow_copyprops(fm,to)
% copy line properties to patches
  props  = {'EraseMode','LineStyle','LineWidth','Marker','
     MarkerSize',...
            'MarkerEdgeColor','MarkerFaceColor','ButtonDownFcn'
               ,      ...
            'Clipping','DeleteFcn','BusyAction','
               HandleVisibility',   ...
            'Selected','SelectionHighlight','Visible'};
  if ~isnumeric(findobj('Type','root')), props(strcmp(props,'
     EraseMode'))=[]; end; %eaj 5/24/16
  lineprops  = {'Color',    props{:}};
  patchprops = {'EdgeColor',props{:}};
  patch2props = {'FaceColor',patchprops{:}};
  fmpatch = strcmp(get(fm,'Type'),'patch');
  topatch = strcmp(get(to,'Type'),'patch');
  set(to( fmpatch& topatch),patch2props,get(fm( fmpatch&
     topatch),patch2props)); %p->p
  set(to(~fmpatch&~topatch),lineprops,  get(fm(~fmpatch&~
     topatch),lineprops  )); %l->l
  set(to( fmpatch&~topatch),lineprops,  get(fm( fmpatch&~
     topatch),patchprops )); %p->l
  set(to(~fmpatch& topatch),patchprops, get(fm(~fmpatch&
     topatch),lineprops)  ,'FaceColor','none'); %l->p
```

```
function arrow_props
% display further help info about ARROW properties
  c = sprintf('\n');
  disp([c ...
  'ARROW Properties:  Default values are given in [square
     brackets], and other' c ...
  '                    acceptable equivalent property names are
     in (parenthesis).' c c ...
  '  Start             The starting points. For N arrows,
                 B' c ...
  '                    this should be a Nx2 or Nx3 matrix.
              /|\           ^' c ...
  '  Stop              The end points. For N arrows, this
              /|||\          |' c ...
  '                    should be a Nx2 or Nx3 matrix.
                 //|||\\        L|' c ...
  '  Length            Length of the arrowhead (in pixels on
     ///|||\\\       e|' c ...
  '                    screen, points on a page). [16] (Len)    /
     ///|||\\\\      n|' c ...
  '  BaseAngle         Angle (degrees) of the base angle       //
     ///|D|\\\\\     g|' c ...
  '                    ADE.  For a simple stick arrow, use    ///
     /  |||  \\\\    t|' c ...
  '                    BaseAngle=TipAngle. [90] (Base)       ///
        |||    \\\   h|' c ...
```

```
'  TipAngle        Angle (degrees) of tip angle ABC.    //<-
   ---->||      \\   |' c ...
'                  [16] (Tip)                          /
   base |||        \  V' c ...
'  Width           Width of the base in pixels.  Not  E
   angle ||<-------->C' c ...
'                  the ''LineWidth'' prop. [0] (Wid)
              |||tipangle' c ...
'  Page            If provided, non-empty, and not NaN,
           |||' c ...
'                  this causes ARROW to use hardcopy
              |||' c ...
'                  rather than onscreen proportions.
               A' c ...
'                  This is important if screen aspect
   -->   <-- width' c ...
'                  ratio and hardcopy aspect ratio are
   ----CrossDir---->' c ...
'                  vastly different. []' c...
'  CrossDir        A vector giving the direction towards
   which the fletches' c ...
'                  on the arrow should go.  [computed such
   that it is perpen-' c ...
'                  dicular to both the arrow direction and
   the view direction' c ...
'                  (i.e., as if it was pasted on a normal 2-D
    graph)]  (Note' c ...
```

```
'                 that CrossDir is a vector.  Also note that
    if an axis is' c ...
'                 plotted on a log scale, then the
   corresponding component' c ...
'                 of CrossDir must also be set appropriately
   , i.e., to 1 for' c ...
'                 no change in that direction, >1 for a
   positive change, >0' c ...
'                 and <1 for negative change.)' c ...
'  NormalDir      A vector normal to the fletch direction (
   CrossDir is then' c ...
'                 computed by the vector cross product [Line
   ]x[NormalDir]). []' c ...
'                 (Note that NormalDir is a vector.  Unlike
   CrossDir,' c ...
'                 NormalDir is used as is regardless of log-
   scaled axes.)' c ...
'  Ends           Set which end has an arrowhead.  Valid
   values are ''none'',' c ...
'                 ''stop'', ''start'', and ''both''. [''stop
   ''] (End)' c...
'  ShortenLength  Shorten length of arrowhead(s) if line is
   too short' c ...
'  ObjectHandles  Vector of handles to previously-created
   arrows to be' c ...
'                 updated or line objects to be converted to
    arrows.' c ...
'                 [] (Object,Handle)' c ...
```

```
  '  Type            ''patch'' creates the arrow with a PATCH
     object (the default)' c ...
  '                  and ''line'' creates it with a LINE object
      [''patch''].' c ...
  '  Color           For patch arrows (the default), set both '
     'FaceColor'' and' c ...
  '                  ''EdgeColor'' to the given value.  For
     line arrows, set' c ...
  '                  the ''Color'' property to the given value.
     ' c ...
  ]);



function out = arrow_demo
  % demo
  % create the data
  [x,y,z] = peaks;
  [ddd,out.iii]=max(z(:));
  out.axlim = [min(x(:)) max(x(:)) min(y(:)) max(y(:)) min(z(:)
     ) max(z(:))];

  % modify it by inserting some NaN's
  [m,n] = size(z);
  m = floor(m/2);
  n = floor(n/2);
  z(1:m,1:n) = NaN*ones(m,n);
```

```
  % graph it
  clf('reset');
  out.hs=surf(x,y,z);
  out.x=x; out.y=y; out.z=z;
  xlabel('x'); ylabel('y');


function h = arrow_demo3(in)
  % set the view
  axlim = in.axlim;
  axis(axlim);
  zlabel('z');
  %set(in.hs,'FaceColor','interp');
  view(3); % view(viewmtx(-37.5,30,20));
  title(['Demo of the capabilities of the ARROW function in 3-D
     ']);

  % Normal blue arrow
  h1 = feval(mfilename,[axlim(1) axlim(4) 4],[-.8 1.2 4], ...
             'EdgeColor','b','FaceColor','b');

  % Normal white arrow, clipped by the surface
  h2 = feval(mfilename,axlim([1 4 6]),[0 2 4]);
  t=text(-2.4,2.7,7.7,'arrow clipped by surf');

  % Baseangle<90
  h3 = feval(mfilename,[3 .125 3.5],[1.375 0.125 3.5],30,50);
  t2=text(3.1,.125,3.5,'local maximum');
```

```
% Baseangle<90, fill and edge colors different
h4 = feval(mfilename,axlim(1:2:5)*.5,[0 0 0],36,60,25, ...
           'EdgeColor','b','FaceColor','c');
t3=text(axlim(1)*.5,axlim(3)*.5,axlim(5)*.5-.75,'origin');
set(t3,'HorizontalAlignment','center');

% Baseangle>90, black fill
h5 = feval(mfilename,[-2.9 2.9 3],[-1.3 .4 3.2],30,120,[],6,
   ...
           'EdgeColor','r','FaceColor','k','LineWidth',2);

% Baseangle>90, no fill
h6 = feval(mfilename,[-2.9 2.9 1.3],[-1.3 .4
   1.5],30,120,[],6, ...
           'EdgeColor','r','FaceColor','none','LineWidth',2);

% Stick arrow
h7 = feval(mfilename,[-1.6 -1.65 -6.5],[0 -1.65
   -6.5],[],16,16);
t4=text(-1.5,-1.65,-7.25,'global mininum');
set(t4,'HorizontalAlignment','center');

% Normal, black fill
h8 = feval(mfilename,[-1.4 0 -7.2],[-1.4 0 -3],'FaceColor','k
   ');
t5=text(-1.5,0,-7.75,'local minimum');
set(t5,'HorizontalAlignment','center');
```

```
% Gray fill, crossdir specified, 'LineStyle' --
h9 = feval(mfilename,[-3 2.2 -6],[-3 2.2
   -.05],36,[],27,6,[],[0 -1 0], ...
          'EdgeColor','k','FaceColor',.75*[1 1 1],'LineStyle
             ','--');

% a series of normal arrows, linearly spaced, crossdir
   specified
h10y=(0:4)'/3;
h10 = feval(mfilename,[-3*ones(size(h10y)) h10y -6.5*ones(
   size(h10y))], ...
           [-3*ones(size(h10y)) h10y -.05*ones(size(h10y))],
              ...
           12,[],[],[],[],[0 -1 0]);

% a series of normal arrows, linearly spaced
h11x=(1:.33:2.8)';
h11 = feval(mfilename,[h11x -3*ones(size(h11x)) 6.5*ones(size
   (h11x))], ...
           [h11x -3*ones(size(h11x)) -.05*ones(size(h11x))])
              ;

% series of magenta arrows, radially oriented, crossdir
   specified
h12x=2; h12y=-3; h12z=axlim(5)/2; h12xr=1; h12zr=h12z; ir
   =.15;or=.81;
h12t=(0:11)'/6*pi;
```

```
h12 = feval(mfilename,
                                                ...
            [h12x+h12xr*cos(h12t)*ir h12y*ones(size(h12t))
                     ...
             h12z+h12zr*sin(h12t)*ir],[h12x+h12xr*cos(h12t)*
                or   ...
             h12y*ones(size(h12t)) h12z+h12zr*sin(h12t)*or],
                    ...
            10,[],[],[],[],
                                                      ...
            [-h12xr*sin(h12t) zeros(size(h12t)) h12zr*cos(
               h12t)],...
            'FaceColor','none','EdgeColor','m');

% series of normal arrows, tangentially oriented, crossdir
   specified
or13=.91; h13t=(0:.5:12)'/6*pi;
locs = [h12x+h12xr*cos(h13t)*or13 h12y*ones(size(h13t)) h12z+
   h12zr*sin(h13t)*or13];
h13 = feval(mfilename,locs(1:end-1,:),locs(2:end,:),6);

% arrow with no line ==> oriented downwards
h14 = feval(mfilename,[3 3 .100001],[3 3 .1],30);
t6=text(3,3,3.6,'no line'); set(t6,'HorizontalAlignment','
   center');

% arrow with arrowheads at both ends
```

```
  h15 = feval(mfilename,[-.5 -3 -3],[1 -3 -3],'Ends','both','
     FaceColor','g', ...
               'Length',20,'Width',3,'CrossDir',[0 0 1],'
                  TipAngle',25);

  h=[h1;h2;h3;h4;h5;h6;h7;h8;h9;h10;h11;h12;h13;h14;h15];

function h = arrow_demo2(in)
  axlim = in.axlim;
  dolog = 1;
  if (dolog), set(in.hs,'YData',10.^get(in.hs,'YData')); end;
  shading('interp');
  view(2);
  title(['Demo of the capabilities of the ARROW function in 2-D
     ']);
  hold on; [C,H]=contour(in.x,in.y,in.z,20,'-'); hold off;
  for k=H',
    set(k,'ZData',(axlim(6)+1)*ones(size(get(k,'XData'))),'
       Color','k');
    if (dolog), set(k,'YData',10.^get(k,'YData')); end;
  end;
  if (dolog), axis([axlim(1:2) 10.^axlim(3:4)]); set(gca,'
     YScale','log');
  else,       axis(axlim(1:4)); end;

  % Normal blue arrow
  start = [axlim(1) axlim(4) axlim(6)+2];
  stop  = [in.x(in.iii) in.y(in.iii) axlim(6)+2];
```

```
  if (dolog), start(:,2)=10.^start(:,2); stop(:,2)=10.^stop
     (:,2); end;
  h1 = feval(mfilename,start,stop,'EdgeColor','b','FaceColor','
     b');

  % three arrows with varying fill, width, and baseangle
  start = [-3   -3   10; -3   -1.5 10; -1.5 -3   10];
  stop  = [-.03 -.03 10; -.03 -1.5 10; -1.5 -.03 10];
  if (dolog), start(:,2)=10.^start(:,2); stop(:,2)=10.^stop
     (:,2); end;
  h2 = feval(mfilename,start,stop,24,[90;60;120],[],[0;0;4],'
     Ends',str2mat('both','stop','stop'));
  set(h2(2),'EdgeColor',[0 .35 0],'FaceColor',[0 .85 .85]);
  set(h2(3),'EdgeColor','r','FaceColor',[1 .5 1]);
  h=[h1;h2];

function out = trueornan(x)
if isempty(x),
  out=x;
else,
  out = isnan(x);
  out(~out) = x(~out);
end;
```

Listing A.57. arrow Function

```
%%%
% Author: Jackson Henry
% 6 Dot QCA Cells
```

```
clear all;
close all;

%% Constants

epsilon_0 = 8.854E-12; % [C/(V*m)]
qeV2J = 1.602E-19; % [J]
qe = 1; %[eV]



one = [0;0;1];
null = [0;1;0];
zero = [1;0;0];




%% Parameters
a = 10; %[nm]
gamma = 60;

k = 1/(4*pi* epsilon_0*qeV2J*a);



%% Operators
Z = one*one' - zero*zero'
Pnn = null*null'

%% Create Driver and Node
```

```
%Start with a Driver at position 0,0,0
Driver = ThreeDotCell(); %Spelled out super hard for now
Driver.Type = 'Driver';  %make it type driver
Driver.Polarization = 1; %make polarization -1
Driver.Activation = 1; %make activation 1
Driver.CenterPosition = [0,0,0];

Driver2 = ThreeDotCell();
Driver2.Type = 'Driver';
Driver2.Polarization = 1;
Driver2.Activation = 1;
Driver2.CenterPosition = [2,0,0];

%Now make a Node position 1,0,0
node2 = ThreeDotCell(); %defaults are good for the most part
node2.CenterPosition = [1,0,0];
node2.ElectricField = [0,0,-1];

mycircuit = QCACircuit();




%% Test

dt = 501;
Pdrv = linspace(-1,1,dt);
Efield = linspace(-2.5,1,dt);
```

```
P = zeros(dt);
A = zeros(dt);



% for x=1:dt
%     %set Clock field of Target
%     node2.ElectricField = [0,0,Efield(x)];
%
%     for y=1:dt
%         %set polarization of driver
%         Driver.Polarization = Pdrv(y);
%
%         %calculate hamiltonian
%         [V, EE] = eig(node2.GetHamiltonian({Driver}));
%         %get the ground state
%         psi = V(:,1);
%
%         %calculate polarization
%         P(x,y) = psi' * Z * psi;
%         A(x,y) = 1 - psi' * Pnn * psi;
%
%     end %y
% end %x



%testing getHamiltonian.
[V, EE] = eig(node2.GetHamiltonian({Driver, Driver2}));
psi = V(:,1); %ground state
```

```
% Polarization is the expectation value of sigma_z
%P = psi' * Z * psi
%A = 1 - psi' * Pnn * psi

%modify gamma, pdrv, adrv and see if everything is correct.



%% Visualization

% figure
% pcolor(Pdrv, Efield, P);
%
% c = colorbar;
%
% shading interp
% grid on;
% set(gca, 'FontSize', 18, 'Fontname', 'Times');
% ylabel('$E_z$ [V/nm]', 'Interpreter', 'latex');
% xlabel('$P_{drv}$', 'Interpreter', 'latex');
% zlabel('$P_{tgt}$', 'Interpreter', 'latex');
```

Listing A.58. basic3DotInteraction Function

```
function choosedialog(handles)
%This opens a dialog box where it double checks to make sure
   that the user
%wants to clear the entire figure of all signals and devices.
   If the user
```

```
%selects 'Yes,' then the function ClearAll() is called after
   the box is deleted,
%otherwise, the box is only deleted.

    d = dialog('Position',[300 300 250 150],'Name','Clear All')
       ;
    txt = uicontrol('Parent',d,...
           'Style','text',...
           'Position',[20 80 210 40],...
           'String','Clear all? (nothing will be saved)');


       btn1 = uicontrol('Parent',d,...
           'Position',[49 40 70 25],...
           'String','Yes',...
           'Callback',@Yes);

       btn2 = uicontrol('Parent',d,...
           'Position',[139 40 70 25],...
           'String','Close',...
           'Callback',@No);

    function Yes(source,callbackdata)

        delete(d);
        ClearAll(handles);

    end
```

```
    function No(source,callbackdata)

        delete(d);
    end




end
```

Listing A.59. choosedialog Function

```
function h = circle(varargin)
%
% H = CIRCLE(x0, y0, R, C) creates a circular patch at position
    (x0,y0),
% with radius R, and color C.  Returns handle to circle.
%
%
% H = CIRCLE(x0, y0, R, C, prop1, val1, ...)
%
% PROPERTIES
%
%    'Points' is used to specify the number of points
   specifying a polygon
%             which approximates a circle. Specify this as an
   integer. The
%             default value is 15.
%
```

```
%    'EdgeColor' is used to specify the color of the line
   defining the
%                circle. Specify this as a RGB triple [R G B],
   with each
%                element in the color value on the interval [0,
    1].
%
%    'FillColor' is used to specify the color of the interior
   of the
%                circle. Specify this as a RGB triple [R G B],
   with each
%                element in the color value on the interval [0,
    1].
%
%    'LineWidth' is used to specify the width of the line
   defining the
%                circle. Specify this as a one-by-one number.
%
%    'LineStyle' is used to specify the style of the line
   defining the
%                circle. Specify this as a character string.
   Some options:
%
%                '-'     Solid line
%
%                '--'    Dashed line
%
%                ':'     Dotted line
```

```
%
nth=30;



x0 = varargin{1};
y0 = varargin{2};
R = varargin{3};
C = varargin{4};

EdgeColor = [0 0 0];
LineWidth = 2;
LineStyle = '-';
TargetAxes = gca;



args = varargin(5:end);
while length(args) >= 2
    prop = args{1};
    val = args{2};
    args = args(3:end);
    switch prop
        case 'Points'
            if ischar(val)
                nth = str2num(val);
            else
                nth = val;
            end
```

```
        case 'FillColor'
            FillColor = val;

        case 'EdgeColor'
            EdgeColor = val;

        case 'LineWidth'
            LineWidth = val;

        case 'LineStyle'
            LineStyle = val;

        case 'TargetAxes'
            TargetAxes = val;

        otherwise
            error(['CIRCLE.M: ', prop, ' is an invalid property
                specifier.']);
    end
end

thetas=[0:nth]*2*pi/nth;
dx=R*cos(thetas);
dy=R*sin(thetas);
x=x0+dx;
y=y0+dy;
h=patch(TargetAxes,x,y,C, 'EdgeColor', EdgeColor, 'LineWidth',
   LineWidth, ...
```

```
    'LineStyle', LineStyle);


%end function circle
```

Listing A.60. circle Function

```
global GreenBlueRedColorMap
% Red column
temp4(:,1) = [ zeros(1,51) linspace(0, 1, 50)]';
% Green column
temp4(:,2) = [ linspace(1, 0, 50) zeros(1, 51)]';
% Blue column
temp4(:,3) = [ zeros(1,25) linspace(0, 0.5625, 25) 0.5625 ...
        linspace(0.5625, 0, 25) zeros(1, 25)]';
GreenBlueRedColorMap = temp4;
clear temp4


global GreyScale
% Red column
temp5(:,1) = [ linspace(0, 1, 101)]';
% Green column
temp5(:,2) = temp5(:,1);
% Blue column
temp5(:,3) = temp5(:,1);
GreyScale = temp5;
clear temp5


global GreyScale
% Red column
temp8(:,1) = [ zeros(1,50) linspace(0,1,51) ]';
```

```
% Green column
temp8(:,2) = [ zeros(1,50) linspace(0,1,51) ]';
% Blue column
temp8(:,3) = [ zeros(1, 50) linspace(0,1,51) ]';
GreyScale = temp8;
clear temp8

global GreyScale8bit
% temp column
OneA = ones(1,7);
OneB = ones(1,8);
tempK = [zeros(1,50), 0*OneA, 42/256*OneA, 85/256*OneA, 127/256
   *OneA, ...
        170/256*OneA, 213/256*OneA, OneB]';
temp9(:,1) = tempK;
temp9(:,2) = tempK;
temp9(:,3) = tempK;

GreyScale8Bit = temp9;
clear temp9

global GreenWhiteRed
% Red column
temp6(:,1) = [ linspace(0,1,51) ones(1,50)]';
% Green column
temp6(:,2) = [ ones(1,50), linspace(1,0,51)]';
% Blue column
temp6(:,3) = [ linspace(0, 1, 50) 1 linspace(1, 0, 50)]';
```

```
GreenWhiteRed = temp6;
clear temp6


global BlueWhite
% Red column
temp7(:,1) = [ zeros(1,50) linspace(0,1,51) ]';
% Green column
temp7(:,2) = [ zeros(1,50) linspace(0,1,51) ]';
% Blue column
temp7(:,3) = [ 0.5625*ones(1, 50) linspace(0.5625, 1, 51) ]';
BlueWhite = temp7;
clear temp7
```

Listing A.61. createcustomcolormap Function

```
function [x,iter] = invitr(A, ep,  numitr)
        [m,n] = size(A);
        if m~=n
          disp('matrix A  is not square')  ;
          return;
        end;
        x=rand(n,1);
        for k = 1 :  numitr
        iter = k;
         xhat = A \ x;
         x = xhat/norm(xhat,2);
         if norm((A)* x , inf) <= ep
           break;
         end;
        end;
```

```
   end
```

Listing A.62. invitr Function

```
clear


%% Load Circuit


input = 'majoritygate6dotDriver';




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
wavelength = 50;
circ = 'majoritygate';
num = 'six/';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




type = 'driver';
wavel = num2str(wavelength);
output = strcat( circ, '_', type, '_w', wavel);
dir = strcat('Circuits_folder/', circ, 'Circuits/', num);


inputfullfile = strcat(input,'.mat');
load(fullfile(dir, inputfullfile));
%load('Circuits folder/faninCircuits/6dot/fanin_inputnodes_
   traditional_6dot.mat');
```

```
mycircuit = Circuit;

simnamefront = strcat('aaa_',output);

%% Set input node fields and clock field
epsilon_0 = 8.854E-12;
a=1e-9;%[m]
q=1;%[eV]
Eo = q^2*(1.602e-19)/(4*pi*epsilon_0*a)*(1-1/sqrt(2));



%% Signals
clf
myaxis = axes;

mycircuit = mycircuit.GenerateNeighborList();
mycircuit = mycircuit.Relax2GroundState(0);
mycircuit = mycircuit.CircuitDraw(0,myaxis);



numOfPeriods = 5;
TimeStepsPerPeriod = 400;

clockSignalsList{1} = Signal();
clockSignalsList{1}.Wavelength = wavelength;
clockSignalsList{1}.Amplitude = 15*Eo;
```

```
clockSignalsList{1}.Period=200;



inputsignal = Signal();
%inputfield = -0.85*Eo;
inputfield = -0.1*Eo;
inputsignal.Type = 'Fermi';
inputsignal.Amplitude = 2*inputfield;
inputsignal.Period = clockSignalsList{1}.Period*2;
inputsignal.Phase = inputsignal.Period/4;
inputsignal.Sharpness = 3;
inputsignal.MeanValue = inputsignal.Amplitude/2;




DriverSignal = Signal();
DriverPol = 1;
DriverSignal.Type = 'Driver';
DriverSignal.Amplitude = 2*DriverPol;
DriverSignal.Period = clockSignalsList{1}.Period*2;
DriverSignal.Phase = DriverSignal.Period/4;
DriverSignal.Sharpness = 3;
DriverSignal.MeanValue = DriverSignal.Amplitude/2;



MajoritySignal1 = Signal();
DriverPol = 1;
MajoritySignal1.Type = 'Driver';
```

```
MajoritySignal1.Amplitude = 2*DriverPol;
MajoritySignal1.Period = clockSignalsList{1}.Period*2;
MajoritySignal1.Phase = MajoritySignal1.Period/4;
MajoritySignal1.Sharpness = 3;
MajoritySignal1.MeanValue = MajoritySignal1.Amplitude/2;



MajoritySignal2 = MajoritySignal1;
MajoritySignal2.Amplitude = 2*DriverPol;
MajoritySignal2.Period = clockSignalsList{1}.Period*4;
MajoritySignal2.Phase = MajoritySignal2.Period/4;
MajoritySignal2.MeanValue = MajoritySignal2.Amplitude/2;

MajoritySignal3 = MajoritySignal1;
MajoritySignal3.Amplitude = 2*DriverPol;
MajoritySignal3.Period = clockSignalsList{1}.Period*8;
MajoritySignal3.Phase = MajoritySignal3.Period/4;
MajoritySignal3.MeanValue = MajoritySignal1.Amplitude/2;



MajoritySignal4 = Signal();
MajoritySignal4.Type = 'Driver';
MajoritySignal4.Amplitude = -2*DriverPol;
MajoritySignal4.Period = clockSignalsList{1}.Period*2;
MajoritySignal4.Phase = MajoritySignal4.Period/4;
MajoritySignal4.Sharpness = 3;
MajoritySignal4.MeanValue = MajoritySignal4.Amplitude/2;
```

```
inputSignalsList{1} = inputsignal;
%mycircuit.Device{1}.Polarization = DriverSignal;
%mycircuit.Device{2}.Polarization = DriverSignal;
%
% mycircuit.Device{1}.Polarization = MajoritySignal1;
% mycircuit.Device{3}.Polarization = MajoritySignal2;
% mycircuit.Device{2}.Polarization = MajoritySignal3;


mycircuit.Device{1}.Polarization = MajoritySignal1;
mycircuit.Device{2}.Polarization = MajoritySignal4;
mycircuit.Device{3}.Polarization = MajoritySignal1;


% mycircuit.Device{1}.Activation = 0;


%% Simulation


%simname = strcat(simnamefront, num2str(clockSignalsList{1}.
   Wavelength));
mycircuit = mycircuit.pipeline(clockSignalsList, '
   inputSignalsList', inputSignalsList, 'Filename',
   simnamefront, 'mobileCharge', 1,  'numOfPeriods',
   numOfPeriods, 'TimeSteps', numOfPeriods*TimeStepsPerPeriod,
   'randomizedRelaxation', 0);
```

```
%% Visualization

PipelineVisualization(simnamefront,myaxis,pwd,'CircuitVideo.mp4
   ',15);
```

Listing A.63. localtestbed Function

```
function parallelPipeline(inputidx)



    %% Load Circuit
    load(fullfile('Circuits folder', 'inputNodesBinaryWire.mat'
       ));
    mycircuit = Circuit;



    %% Set input node fields and clock field
    epsilon_0 = 8.854E-12;
    a=1e-9;%[m]
    q=1;%[eV]
    Eo = q^2*(1.602e-19)/(4*pi*epsilon_0*a)*(1-1/sqrt(2));
    clk = 5;

    inputandclock = [0,inputidx/10,clk]*Eo;

    mysignal = Signal();
    mysignal.Wavelength = 100;
```

```
mysignal.Amplitude = 5;
mysignal.Period = 1;



for i=1:length(mycircuit.Device)
    if isa(mycircuit.Device{i},'QCASuperCell')
        for j=1:length(mycircuit.Device{i}.Device)
            mycircuit.Device{i}.Device{j}.ElectricField =
               inputandclock;
            %disp('g')
        end

    else
        mycircuit.Device{i}.ElectricField = inputandclock;

    end
end

%%
%clf
%myaxis = axes;



simnamefront = 'inputFieldProblem_w';
simname = strcat(simnamefront, num2str(mysignal.Wavelength)
   , '_i',num2str(inputidx));

mycircuit = mycircuit.GenerateNeighborList();
```

```
    mycircuit = mycircuit.Relax2GroundState();


    % mycircuit = mycircuit.pipeline({mysignal}, simname);
    %clockSignalsList{1}.Wavelength = 100;
    mycircuit = mycircuit.pipeline({mysignal}, 'Filename',
       simname, 'TimeSteps', 50);







    WorkerID = getfield(getCurrentTask(), 'ID');
    disp(['Commenced job on worker ', num2str(WorkerID), ' for
       index ', num2str(inputidx)]);
```

Listing A.64. parallelPipeline Function

```
function textborder(x, y, string, text_color, border_color,
   varargin)
%TEXTBORDER Display text with border.
%   TEXTBORDER(X, Y, STRING)
%   Creates text on the current figure with a one-pixel border
   around it.
%   The default colors are white text on a black border, which
   provides
%   high contrast in most situations.
%
%   TEXTBORDER(X, Y, STRING, TEXT_COLOR, BORDER_COLOR)
%   Optional TEXT_COLOR and BORDER_COLOR specify the colors to
   be used.
%
```

```
%   Optional properties for the native TEXT function (such as '
   FontSize')
%   can be supplied after all the other parameters.
%   Since usually the units of the parent axes are not pixels,
   resizing it
%   may subtly change the border of the text out of position.
   Either set
%   the right size for the figure before calling TEXTBORDER, or
    always
%   redraw the figure after resizing it.
%
%   Author: Jo?o F. Henriques, April 2010
  if isempty(string), return; end

  if nargin < 5, border_color = 'k'; end  %default: black
     border
  if nargin < 4, text_color = 'w'; end  %default: white text

  %border around the text, composed of 4 text objects
  offsets = [0 -1; -1 0; 0 1; 1 0];
  for k = 1:4,
    h = text(x, y, string, 'Color',border_color, varargin{:});

    %add offset in pixels
    set(h, 'Units','pixels')
    pos = get(h, 'Position');
    set(h, 'Position', [pos(1:2) + offsets(k,:), 0])
    set(h, 'Units','data')
```

```
  end

  %the actual text inside the border
  h = text(x, y, string, 'Color',text_color, varargin{:});

  %same process as above but with 0 offset; corrects small
     roundoff
  %errors
  set(h, 'Units','pixels')
  pos = get(h, 'Position');
  set(h, 'Position', [pos(1:2), 0])
  set(h, 'Units','data')
end
```

Listing A.65. textborder Function

```
function vizAtCertainTimeButton(handles)
%This function will use the simulation .mat files to create a
   video with
%the help of the PipelineVisualization function



Sim = getappdata(gcf,'SimResults');
path = getappdata(gcf, 'SimResultsPath');
myCircuit = getappdata(gcf,'myCircuit');



if Sim
    load(Sim);
```

```
cla;

myCircuit = obj.CircuitDraw(0, gca);
timestep = str2num( get(handles.vizAtCertainTimeEditBox,'
   String') );
showClockField = str2num( get(handles.showClockFieldRadio,'
   String') );

if timestep < 1
    timestep = 1;
    myCircuit = myCircuit.CircuitDraw(timestep, gca, [pols(
       timestep,:); acts(timestep,:)]);

elseif timestep > size(pols,1)
    timestep = size(pols,1);
    myCircuit = myCircuit.CircuitDraw(timestep, gca, [pols(
       timestep,:); acts(timestep,:)]);

else









    xit = 1;
    for i=1:length(myCircuit.Device)
```

```
    if isa(myCircuit.Device{i},'QCASuperCell')
        for j=1:length(obj.Device{i}.Device)
            center = obj.Device{i}.Device{j}.
               CenterPosition;
            xpos(xit) = center(1);
            ypos(xit) = center(2);
            xit = xit + 1;
        end
    else
        center = myCircuit.Device{i}.CenterPosition;
        xpos(xit) = center(1);
        ypos(xit) = center(2);
        xit = xit + 1;
    end
end

xmax = max(xpos);
xmin = min(xpos);
ymax = max(ypos);
ymin = min(ypos);



x = xmin:xmax;
nx = 125;
xq = linspace(xmin-1, xmax+1, nx);
yq = linspace(ymin-2, ymax+2, nt);
```

```
    if (length(clockSignalsList) == 1)
        clockSignal = clockSignalsList{1};
    else
        error('Too many signals')
    end
    tperiod = clockSignal.Period*2;
    time_array = linspace(0,tperiod,nt);

    tp = mod(time_array, tperiod);

    %clockSignal.drawSignal([xmin-1,xmax+1], [ymin-2, ymax
       +2], tp(timestep));









    myCircuit = myCircuit.CircuitDraw(timestep, gca, [pols(
       timestep,:); acts(timestep,:)]);
    setappdata(gcf,'myCircuit',myCircuit);

    DragDrop();

end
```

```
else

    [Sim, path]= uigetfile('*.mat'); %path gets sent into
       Pipeline in order to change the path, that way we can
       put the video file anywhere
    load(Sim);

    setappdata(gcf,'myCircuit',myCircuit);
    setappdata(gcf,'SimResults', Sim);
    setappdata(gcf,'SimResultsPath', path);

    disp('click it again for now.....')



end



end
```

Listing A.66. vizAtCertainTimeButton Function

REFERENCES

[1] C. Lent, P. Tougaw, W. Porod, and G. Bernstein, "Quantum cellular automata," *Nanotechnology*, vol. 4, p. 49, 1993.

[2] C. Lent and P. Tougaw, "Lines of interacting quantum-dot cells: A binary wire," *J. Appl. Phys.*, vol. 74, pp. 6227–6233, 1993.

[3] K. Walus, F. Karim, and A. Ivanov, "Architecture for an external input into a molecular qca circuit," *Journal of Computational Electronics*, vol. 8, no. 1, pp. 35–42, March 2009.

[4] A. Pulimeno, M. Graziano, D. Demarchi, and G. Piccinini, "Towards a molecular qca wire: simulation of write-in and read-out systems," *Solid State Electron.*, vol. 77, pp. 101–107, 2012.

[5] D. Frank, "Power-constrained cmos scaling limits," *IBM J Res Dev*, vol. 46, no. 2/3, pp. 235–244, March/May 2002.

[6] A. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, pp. 117–157, 2015.

[7] C. S. Lent, "Molecular electronics - bypassing the transistor paradigm," *Science*, vol. 288, pp. 1597–1599, Jun. 2000.

[8] M. Lieberman, S. Chellamma, B. Varughese, Y. Wang, C. Lent, G. Bernstein, G. Snider, and F. Peiris, "Quantum-dot cellular automata at a molecular scale," *Ann. N.Y. Acad. Sci.*, vol. 960, pp. 225–239, 2002.

[9] C. Lent, B. Isaksen, and M. Lieberman, "Molecular quantum-dot cellular automata," *J. Am. Chem. Soc.*, vol. 125, pp. 1056–1063, 2003.

[10] E. Blair, S. Corcelli, and C. Lent, "Electric-field-driven electron-transfer in mixed-valence molecules," *J Chem Phys*, vol. 145, p. 014307, June 2016.

[11] A. O. Orlov, I. Amlani, G. H. Bernstein, C. S. Lent, and G. L. Snider, "Realization of a functional cell for quantum-dot cellular automata," *Science*, vol. 277, no. 5328, pp. 928–930, Aug. 1997.

[12] P. Tougaw and C. Lent, "Logical devices implemented using quantum cellular automata," *J. Appl. Phys.*, vol. 75, no. 3, pp. 1818–1825, Feb. 1994.

[13] M. Niemier, M. Kontz, and P. Kogge, "A design of and design tools for a novel quantum dot based microprocessor," in *Proc. of the 37th Design Automation Conf.*, 2000, pp. 227–232.

[14] I. Amlani, A. Orlov, G. Snider, and C. Lent, "Demonstration of a six-dot quantum cellular automata system," *Appl. Phys. Lett.*, vol. 72, pp. 2179–2181, 1998.

[15] C. Smith, S. Gardelis, A. Rushforth, R. Crook, J. Cooper, D. Ritchie, E. Linfield, Y. Jin, and M. Pepper, *SUPERLATTICES AND MICROSTRUCTURES*, vol. 34, no. 3-6, pp. 195–203, SEP-DEC 2003, 6th International Conference on New Phenomena in Mesoscopic Structures/4th International Conference on Surfaces and Interfaces of Mesoscopic Devices, Maui, HI, DEC 01-05, 2003.

[16] S. Gardelis, C. Smith, J. Cooper, D. Ritchie, E. Linfield, and Y. Jin, "Evidence for transfer of polarization in a quantum dot cellular automata cell consisting of semiconductor quantum dots," *PHYSICAL REVIEW B*, vol. 67, no. 3, JAN 15 2003.

[17] M. B. Haider, J. L. Pitters, G. A. DiLabio, L. Livadaru, J. Y. Mutus, and R. A. Wolkow, "Controlled coupling and occupation of silicon atomic quantum dots at room temperature," *Phys. Rev. Lett.*, vol. 102, p. 046805, 2009.

[18] J. Christie, R. Forrest, S. Corcelli, N. Wasio, R. Quardokus, R. Brown, S. Kandel, Y. Lu, C. Lent, and K. Henderson, "Synthesis of a neutral mixed-valence diferrocenyl carborane for molecular quantum-dot cellular automata applications," *Angewandte Chemie*, vol. 127, pp. 15 668–15 671, 2015.

[19] K. Hennessy and C. S. Lent, "Clocking of molecular quantum-dot cellular automata," *Journal of Vacuum Science & Technology B*, vol. 19, no. 5, pp. 1752–1755, Sep. 2001.

[20] R. Quardokus, N. Wasio, R. Forrest, C. Lent, S. Corcelli, J. Christie, K. Henderson, and S. Kandel, "Adsorption of diferrocenylacetylene on au(111) studied by scanning tunneling microscopy," *Phys. Chem. Chem. Phys.*, vol. 18, pp. 6973–6981, 2013.

[21] Y. Lu, R. Quardokus, C. Lent, F. Justaud, C. Lapinte, and S. Kandel, "Charge localization in isolated mixed-valence complexes: an stm and theoretical study," *J. Am. Chem. Soc.*, vol. 132, no. 38, pp. 13 519–13 524, 2010.

[22] B. Tsukerblat, A. Palii, and J. Clemente-Juan, "Self-trapping of charge polarized states in four-dot molecular quantum cellular automata: bi-electronic tetrameric mixed-valence species," *Pure and Applied Chemistry*, vol. 87, no. 3, pp. 271–282, 2015.

[23] J. Henry and E. Blair, "The role of the tunneling matrix element and nuclear reorganization in the design of quantum-dot cellular automata molecules," *J Appl Phys*, vol. 123, no. 6, p. 064302, 2018.

[24] E. P. Blair and C. S. Lent, "Environmental decoherence stabilizes quantum-dot cellular automata," *J. Appl. Phys.*, vol. 113, p. 124302, JAN 2013.

[25] J. S. Ramsey and E. P. Blair, "Operator-sum models of quantum decoherence in molecular quantum-dot cellular automata," *J Appl Phys*, vol. 122, p. 084304, August 2017.

[26] E. Blair, G. Toth, and C. Lent, "Entanglement loss in molecular quantum-dot qubits due to interaction with the environment," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195602, 2018.

[27] E. Winfree, F. Liu, L. Wenzler, and N. Seeman, "Design and self-assembly of two-dimensional dna crystals," *Nature*, vol. 349, pp. 539–544, 1998.

[28] P. Rothemund, "Folding dna to create nanoscale shapes and patterns," *Nature*, vol. 440, pp. 297–302, 2006.

[29] K. Sarveswaran, P. Huber, M. Lieberman, C. Russo, and C. Lent, "Nanometer scale rafts built from dna tiles," in *PROCEEDINGS OF THE THIRD IEEE CONFERENCE ON NANOTECHNOLOGY (IEEE NANO 2003)*, 2003, pp. 417–420.

[30] P. Tougaw and C. Lent, "The effect of stray charge on quantum cellular automata," *Jpn. J. Appl. Phys.*, vol. 34, pp. 4373–4375, 1995.

[31] M. LaRue, D. Tougaw, and J. Will, "Stray charge in quantum-dot cellular automata: A validation of the intercellular hartree approximation," *IEEE Trans Nanotechnol*, vol. 12, no. 2, pp. 225–233, 2013.

[32] P. Nielsen, M. Egholm, R. Berg, and O. Buchardt, "Sequence-selective recognition of dna by strand displacement with a thymine-substituted polyamide," *Science*, vol. 254, no. 5037, pp. 1497–1500, December 1991.

[33] T. Fulton and G. Dolan, "Observation of single-electron charging effects in small tunnel junctions," *Phys. Rev. Lett.*, vol. 59, pp. 109–112, 1987.

[34] S. Hile, M. House, E. Peretz, J. Verduijn, D. Widmann, T. Kobayashi, S. Rogge, and M. Simmons, "Radio frequency reflectometry and charge sensing of a precision placed donor in silicon," *Appl. Phys. Lett.*, vol. 107, 2015.

[35] G. Karbasian, A. Orlov, A. Mukasyan, and G. Snider, "Single-electron transistors featuring silicon nitride tunnel barriers prepared by atomic layer deposition," in *Joint International EUROSOI Workshop and International Conference on Ultimate Integration on Silicon (EUROSOI-ULIS)*, 2016, pp. 32–25.

[36] E. Blair, *Electric-field Inputs for Molecular Quantum-dot Cellular Automata Circuits*, arxiv:1805.04029 [quant-ph] ed., 2018.

[37] E. Blair, "Electric-field inputs for molecular quantum-dot cellular automata circuits," *IEEE Transactions on Nanotechnology*, vol. 18, pp. 453–460, 2019. [Online]. Available: https://doi.org/10.1109/tnano.2019.2910823

[38] G. Tóth and C. S. Lent, "Role of correlation in the operation of quantum-dot cellular automata," *Journal of Applied Physics*, vol. 89, no. 12, pp. 7943–7953, 2001. [Online]. Available: https://doi.org/10.1063/1.1368389

[39] K. Walus, T. J. Dysart, G. A. Jullien, and R. A. Budiman, "Qcadesigner: A rapid design and simulation tool for quantum-dot cellular automata," *IEEE Trans. Nanotechnol.*, vol. 3, no. 1, pp. 26–31, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1109/TNANO.2003.820815