

Implementing a Generic Three State Coherence Vector Model for QCA

Timothy J. Dysart
tdysart@cse.nd.edu

Last updated: Aug. 21, 2009

Abstract

This report details an implementation of general purpose QCA simulation method based on the three state coherence vector model. This model assumes the use of a six dot molecular cell [1–3]. This code has been used to generate the results in Sections 6.2 and 6.3 of [4]

1 Abbreviations Used

For reference, we provide a list of abbreviations, generally in order of their initial use, in this report.

- 3SCV: three state coherence vector
- E_k : kink energy
- SSCV: steady state coherence vector. Equivalent to $\vec{\lambda}_{ss}(t)$.
- DP: Dormand-Prince

2 Problem Overview

The three state coherence vector model utilizes Eq. 1 for the equation of motion.

$$\frac{\partial}{\partial t} \vec{\lambda} = \Omega \vec{\lambda} - \frac{1}{\tau} [\vec{\lambda} - \vec{\lambda}_{ss}(t)] \quad (1)$$

For this (system of) first-order ordinary differential equation(s), τ is the energy relaxation time representing the strength of the coupling between the system and the thermal bath.

Since this is a system of first-order ODEs, multiple solvers are available. In discussions with C. Lent, the typical solver for this problem has been Matlab's ode45 function. A GNU based form of

Matlab named Octave has implemented the ode45 function. The code for Octave's ode45 function can be found online (it was the 4th entry in Google for *octave, ode45*). These functions will guide our implementation of this model.

For reference, ode45 implements the Dormand-Prince method (a Runge-Kutta family member) for solving ODEs.

3 High Level Algorithm

Currently, this code is written in such a manner that only wire segments with a single input cell can be simulated. It should be noted that only straight wire segments are being used in this code for the time being, and it should not be considered accurate for circuits beyond that without more testing and verification. The remainder of this page is blank in order to put the entire code overview on one page.

In this implementation of the 3SCV modeling method, the basic code is as follows:

```
1  Set fixed parameters
2  Load input files (circuit, technical file, defect file)
3  Seed random number generator
4  Create physical cell list
5  Inject defects
6  Initialize output cell array
7  Initialize simulation
8    Find neighbors
9    Compute Ek for neighbors
10   Find ideal Ek
11   Setup clock signal
12   Rearrange cells
13   End initialize simulation
14 Run simulation
15   Initialize clock values
16   Set input cell polarization to -1
17   Initialize other cells
18   Compute SSCV
19   Set lambda (coherence vector), K-matrix, polarization for all
20     non-input/fixed cells
21   t = 0
22   while t <= EndTime
23     Reset MaxDelta,MaxTau (DP error estimate values)
24     Fill K-matrix (DP method)
25     Compute error estimates (DP method)
26     Error decision (DP method)
27     if good time step
28       update t
29       Set new lambda, K-matrix, polarization for all
30         non-input/fixed cells
31       Update output cell array
32     Update time step (DP method)
33   end while
34   Determine Pass/Fail from output cell array
35   End run simulation; repeat with input cell polarization = +1
36 Clean up space
37 End program
```

4 Pre-simulation setup

This section will detail the code in lines 1-6 of Sec. 3. Each line has a subsection, except for line 3 which is self-explanatory.

4.1 Line 1: Set Fixed Parameters

Here, we set several of the global variables used throughout the simulation. The first parameter, γ , is the tunneling energy between a polarized state and a null state. This value can be obtained from quantum chemistry calculations, but in [1, 2] has been assumed to be $0.1 \cdot E_k$ or $0.2 \cdot E_k$. For now, this value has been fixed to 0.02eV which is approximately $0.15 \cdot E_k$ for the cell structure assumed here (discussed later).

We also set a neighbor radius. If two cells have their centers within this distance, they are assumed to be neighbors. Since the interaction between cells falls in proportion to the fifth power of their distance apart, this is used to identify a neighborhood for each cell. Currently, this distance is set to 10 nm.

We calculate $k_B T$ here as well. k_B is Boltzmann's constant and T is the assumed temperature. We use $T = 300 \text{K}$ here.

We also compute $1/\tau$ where τ is the energy relaxation time between the system and the thermal bath. We currently set this value to the ideal E_k divided by $10 \cdot \hbar$ which was one of the values for $1/\tau$ in [1]. For reference, we initialize this value to $10 \cdot \hbar$ here and then finish setting it after calculating the ideal E_k value (line 10).

4.2 Line 2: Load Input Files

Three input files are expected to run a simulation. First, a circuit design file is input. This file identifies the location of the cells on a generic grid (20 units between cell centers). It should be noted that the first cell of this file must be the input cell for the circuit. Second, a cell technical file is input. This file identifies the distance between cell centers (vertical and horizontal, in nm), the distance between active dots (vertical and horizontal, in nm), the distance between active dots and null dots in the z-plane, and the range for stray charges in the z-plane. Lastly, a defects file is input. This file contains maximum ranges for displacements (x,y,z planes, in nm) and rotations (around xy,xz,yz, in degrees). It also determines the percentage of missing cells and the number of stray charges (positive integer). Currently, only xy rotations have been implemented; further details are presented when discussing line 5.

The basis for doing the input files in this way was formed in [5].

If the user wishes to keep some non-input/fixed/output cells defect free, the user just needs to provide a label to these cells. For regular cells, the code will check the label to see if a defect can be injected; if the label is "NO NAME" then it can have a defect.

Since we currently assume the flat cell model used in [3], a cell consists of 6 dots. The locations of the dots (all in nm), starting in the upper left corner and rotating counter clockwise are (x,y): (0.5,1.5), (0.5,1), (0.5,0.5), (1.5,0.5), (1.5,1), (1.5,1.5) with a cell center of (1,1). The cell center

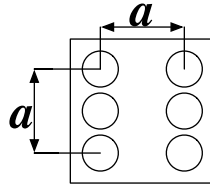


Figure 1: Sample cell (from [3]).

to cell center distance is 2nm in both the horizontal and vertical directions. A sample cell (with spacings) is shown in Fig. 1.

4.3 Line 4: Create Physical Cell List

In this component of the program, we combine the circuit design file with the technical file parameters to create a list of cells with the proper distances between them. In this function, we also create and initialize the variables each cell will need during the program. The list of cells is stored as a doubly-linked list.

As in the circuit design file, the first cell in this list should be the input cell for the circuit.

4.4 Line 5: Inject Defects

In this function, the defects described by the defect input file are injected into the circuit. When injecting defects, the displacements are done first, followed by rotations, missing cells, and stray charges.

For the displacements and rotations, each cell is given the prescribed defect based on two random values (unique to each cell): a sign value and a ratio value. The sign value determines which direction the defect is injected and the ratio value, in conjunction with the defect magnitude described by the defect input file, determines the final magnitude of the defect. The sign value is designed to be +1 half the time, -1 the other half of the time. The ratio value ranges from [0,1] and is then multiplied by the maximum displacement or rotation. For example, if the x-displacement in the defect file is 0.15nm, the sign value is -1, and the ratio value is 0.65, the selected cell is then moved $0.65 * 0.15 = 0.0975\text{nm}$ in the -x direction.

For rotation defects, only rotations in the xy plane are currently implemented. The code for xz and yz rotations has not been investigated, tested, and/or verified for correctness.

For the missing cells, any non-input/fixed/output cell is eligible to be removed. A random number generator is used to determine how many cells should be removed if the percentage of cells to be removed is not a whole number (e.g. dropping 5% of 75 cells is 3.75). The remaining fractional part (e.g. 0.75) determines the percentage of the time that the number of cells to be dropped will be rounded up. For this case, there is a 75% chance that 4 cells will be dropped and a 25% chance that 3 cells will be dropped. Running multiple instances of this program would then create an average number of dropped cells per instance equal to the calculated number (3.75 in this

case).

A stray charge model has not been implemented as of the present time. Stray charges are allowed to exist anywhere within a range roughly equivalent to the neighbor radius of the minimum and maximum x and y ranges of the circuit design (a rectangular region). Since we are currently using flat cells, they lie on the same plane as the cells. At any time, a stray charge is a full electron. To date, we have not considered any scenarios where stray charges exist.

4.5 Line 6: Initialize output cell array

In this function, we create a array where each element contains a pointer to an output cell, a value that is incremented to count clock steps, and a running polarization sum. In this function, we identify all of the output cells in the physical cell list and create an output cell array of that length. We then assign each element of the output cell array a pointer to an output cell in the physical cell list. The other values are initialized to zero for later use.

5 Initialize Simulation

In this section, we describe the sub-functions of *Initialize Simulation* which consists of lines 7-13 in the code shown in Sec. 3.

5.1 Line 8: Find Neighbors

For each cell in the circuit, we identify and store a list of pointers to all cells that are considered a neighbor of the given cell. To be considered a neighbor of a cell, the cell center to cell center distance must be within the neighbor radius set in Sec. 4.1.

In this function, once we know the number of neighbors of a cell, we create two Ek lists: one where the neighbor cell has a positive polarization and one where the neighbor cell has a negative polarization. These two lists are used since the results of Appendix C in [2] suggest that this energy is dependent upon the polarization of the neighboring cell. The values for these lists are computed in the next function.

5.2 Line 9: Compute Ek for Neighbors

In this function, we compute the kink energy between a given cell, A , and each of its neighbors, B . This energy, E_k , is found as follows: $E_k = E_{diff} - E_{same}$ where E_x is computed using Eq. 2. For this equation, $q_{i(j)}$ is the electronic charge located on dot $i(j)$, ϵ is the permittivity of the molecule, and r is the distance between dot i and dot j . When $x = diff$ cells A and B have different polarizations and when $x = same$ cells A and B have the same polarization. It should be noted here that each dot also has a neutralizing positive charge of $e/3$. This charge allows the cell to remain electronically neutral.

$$E_x = \sum_{i=0}^{A.dots} \sum_{j=0}^{B.dots} \frac{q_i * q_j}{4\pi\epsilon r} \quad (2)$$

As discussed in Sec. 5.1 each cell maintains two Ek lists. In calculating the values for the positive Ek neighbor list for cell *A*, the neighbor cells, *B*, are assumed to have a positive polarization (thus, the polarization of cell *A* is -1,+1 for *diff* and *same*). For the negative Ek neighbor list, the neighbor cells, *B*, are assumed to have a polarization of -1 (thus *A* is +1,-1 in *diff* and *same*).

At the completion of this step, each cell contains a list of neighbors and two filled in Ek lists. The Ek values generated by this function are in eV units.

5.3 Line 10: Find Ideal Ek

In this function, we compute the Ek value for two adjacent cells that are defect free. We use the energy equations of the previous function here, but in this case cell *A* has a fixed polarization of +1 while the neighbor cell *B* has polarizations of -1,+1 when computing the *diff* and *same* energies.

5.4 Line 11: Setup Clock Signal

This function has two major purposes: initialize/set the simulation timing parameters and to create three clocking arrays. The simulation timing parameters that are set here are the clock period, simulation length, and maximum time step. The latter two values are based off of the clock period and are currently equal to 1.01*clock period and clock period/10000 respectively. The clock period is currently set to 5ps as is done in Sec. V of [1].

The three clocking arrays (of equal length) created in this function define the following: x-locations of the clock, a “fixed” clock signal, and an “adjusted” time signal. The array of x-locations spans a range of x values larger than the circuit design and is used in conjunction with the other arrays to identify the value of the clock at a range of points in and outside of the circuit design. The “fixed” clock signal array holds the value of the clock signal, in eV, at the locations defined in the x-locations array, for the time *t* as shown in lines 21, 22, 28 of the high-level algorithm. The “adjusted” time signal is used to have clock signal values that vary as the time step is changed or modified. Using this second array prevents having to recompute the “fixed” array multiple times.

5.5 Line 12: Rearrange Cells

In this function, we rearrange the cells in the doubly-linked list of physical cells. This may or may not be necessary, but we are doing it to ensure some variability between simulations (to ensure mathematical consistency). In this function, a number of swaps is computed; currently 30% of the number of non-input cells is used (arbitrary percentage). We then randomly select two different, non-input cells to swap in the doubly-linked list. The swap performed simply changes the pointers in the list and does not move the location of any data.

6 Run Simulation, Pre-While Loop

In this section, we discuss the steps in the run simulation function that are not enclosed in the while loop. These steps consist of lines 15-21 of the code in Sec. 3. Lines 16 and 21 are self-explanatory and will not be discussed elsewhere.

6.1 Line 15: Initialize clock values

We have used two different clock signals in this work. In both cases, the magnitude of the clock has ranged from $-Ek$ to $+Ek$. The first clock signal type passes a simple wave (rising edge, high signal, falling edge, rest low) over the circuit. This is done to be approximately equivalent to the clocking wave used in MAquinas [6]. In the current implementation, each portion of the wave will have a width of 100nm at a specific time. During a clock period, this wave will pass over each cell only once.

The second clock type has been removed for the time being. This clock type is a true sinusoidal wave based on equation 13 in [7] and is repeated in Eq. 3. For this work, we assume $E_C^0 = Ek$, λ_c to approximately equal the x-range of the circuit, and T_c to be the clock period as set in Line 11.

$$E_c(t) = E_C^0 \sin\left(\frac{x}{\lambda_c} - \frac{t}{T_c}\right) \quad (3)$$

In this function, $t = 0$, and the “fixed” and “adjusted” clocks are equivalent.

6.2 Line 17: Initialize other cells

In this function, the non-input/fixed cells in the design are given an initial polarization value equal to the input cell. Random values between -1 and 1 could also be used if desired. This value is then inverted and fed into element 7 (6 in C code) of the SSCV that each cell holds. For the inversion and SSCV step, the fixed and input cells are included.

6.3 Line 18: Compute SSCV

In this function, each element j of the SSCV for each non-input/fixed cell is computed using Eq. 10 in [1], which is shown below as Eq. 4. For this equation, $\hat{\rho}_{ss}(t)$ is found using Eq. 9 in [1] and repeated in Eq. 5. The parameter $\hat{\lambda}_j$ is the j -th generator of SU(3).

$$\vec{\lambda}_{ss}^{(j)} = Tr\{\hat{\rho}_{ss}(t)\hat{\lambda}_j\} \quad (4)$$

$$\hat{\rho}_{ss}(t) = \frac{tmp}{Tr[tmp]} \quad (5)$$

$$tmp = e^{-\hat{H}(t)/k_B T} \quad (6)$$

To find the value of tmp in Eq. 5, we first find the Hamiltonian ($\hat{H}(t)$) for the cell (x). The Hamiltonian is shown in Eq. 7. In this equation, γ is the tunneling energy set in line 1 of the high-level algorithm. E_c is the value of the clock seen at the center of cell x . PEk is found using Eq. 8. In this equation, $NegEk$ is the array holding the E_k values for when the polarization of the neighbor (cell i) of cell x is negative (and similar for $PosEk$).

$$\hat{H}_x(t) = \begin{bmatrix} -0.5 * PEk & 0 & -\gamma \\ 0 & 0.5 * PEk & -\gamma \\ -\gamma & -\gamma & E_c \end{bmatrix} \quad (7)$$

$$PEk = \sum_{i=0}^{x.neighbors} i.Polarization * \begin{cases} x.NegEk[i] & \text{if } i.Polarization < 0 \\ x.PosEk[i] & \text{if } i.Polarization > 0 \end{cases} \quad (8)$$

Since finding tmp requires finding a matrix exponential, we use a decomposition involving the eigenvalues and eigenvectors of the Hamiltonian to compute tmp . The eigenvalues are stored in a diagonal matrix D and the corresponding eigenvectors are in a matrix Z . The elements of D (d_i values) then have the following operation applied to them: $d_i = e^{-d_i/k_B T}$. Then, $tmp = Z * D * Z^T$. From our testing, the operations we have applied to find $\hat{\rho}_{ss}(t)$ are equivalent to computing $\hat{\rho}_{ss}(t)$ via Matlab (where the matrix exponential is a native function).

With $\hat{\rho}_{ss}(t)$ computed, finding the individual elements of the SSCV for a cell (via Eq. 4) are straightforward and not shown here.

6.4 Line 19: Set values

Having computed the SSCV, for $t = 0$, we now set the initial values for our cells in simulation. Non-input/fixed cells are skipped in this step. For the remainder of the cells, their polarization is $-1 * SSCV[6]$ (7th element of SSCV) and their initial coherence vectors are set to the SSCV as well. The first column of the K-matrix (DP method) is also equivalent to the SSCV. This matrix will be discussed in more depth later.

7 Run Simulation, While Loop

In this section, we discuss the while loop component of the run simulation function. This encompasses lines 23-33 of the code presented in Sec. 3. Much of the work done in this section will reference the ode45 code of Octave. As mentioned in the introduction, this method has been used in the past to implement the 3SCV method. ode45 implements a Dormand-Prince solver (Runge-Kutta family) which uses an adaptive time-step to march forward in time.

7.1 Line 23: Reset values

In this segment of the code, we reset two values: MaxDelta and MaxTau. These are used in the DP method to estimate the error and determine the next time step. Currently, they are reset to zero,

but this should probably be changed to a large negative number since when setting these values, we find a maximum. These values are set in Line 26, Error Decision.

7.2 Line 24: Fill K-Matrix

Filling this matrix is an integral component of the DP method. While filling this matrix, multiple function evaluations (Eq. 1) are computed. We describe the process for these evaluations here.

The K-Matrix (one per cell) is an $N \times 7$ matrix where the rows are the coherence vector entries (thus, $N = 8$) and each column is a function evaluation where the coherence vector (aka lambda) and time step have been perturbed by some value. When filling this matrix, the result for a given column is built upon the results of the preceding columns. The initial column for this matrix is the coherence vector of the last known good time step.

The basic code for filling a column in this matrix is as follows:

```
24.1: Computed adjusted time step and update clock signal
24.2: For each column of the K-matrix
24.3:   Compute update lambda for all non-input/fixed cells
24.4:   Compute SSCV
24.5:   Calculate Omega*Lambda vector for all non-input/fixed cells
24.6:   Evaluate cell for all non-input/fixed cells
24.7: End for loop & function
```

7.2.1 Line 24.1

In line 24.1, we perturb (always increase) the current time by some fraction of the time step. This fraction is determined by the DP method coefficients. With the time step in hand, we recompute the “adjusted” clock signal to reflect a new current time for later use in this function.

7.2.2 Line 24.3

In this segment of the code, we perturb lambda (the coherence vector for a cell) by the amount prescribed by the DP method. This perturbation is roughly equivalent to the time step multiplied by a factor that encompasses the previous columns in the K-matrix and DP coefficients.

7.2.3 Line 24.4

In line 24.2, we compute the SSCV using the method described in Sec. 6.3 with the adjusted clock value.

7.2.4 Line 24.5

In this segment of the code, we compute the result of $\Omega * \vec{\lambda}$ (Eq. 6 in [1]). The matrix Ω is described by Eq. 7 in [1] and worked out in Eq. 58 (page 49) of [2]. The vector $\vec{\lambda}$ is the perturbed coherence vector computed in line 24.4.

7.2.5 Line 24.6

In this step, the cell is evaluated and a column is filled in the K-Matrix. This evaluation simply computes Eq. 1 using the results found here. In other terms, we take the result of line 24.5 and subtract $1/\tau*$ (perturbed lambda - SSCV) from it.

7.3 Line 25: Compute Error Estimates

The computing of these estimates comes directly from the DP method implemented in ode45 and are computed for each non-input/fixed cell. The first estimate takes the last good coherence vector for the cell and increases it by the time step multiplied by the K-matrix and a set of DP coefficients. The second estimate is computed the same way with a different set of DP coefficients. The difference between these estimates is then computed and stored (termed *GamErr* here).

7.4 Line 26: Error Decision

This step determines if the current time step is acceptable (good) and is based entirely on the DP method. In this step, all of the non-input/fixed cells are treated as one large array. As such, we find two single values. The first is MaxDelta and it is the largest element of the *GamErr* vector (actual error) for all of the non-input and fixed cells. The second is MaxTau and it is the maximum value of the coherence vector (last known good time step) for all of the non-input/fixed cells. If MaxTau is less than one, it is set equal to one. MaxTau is then multiplied by a tolerance value (currently 1×10^{-8}) – this is the allowable error. If MaxDelta is greater than MaxTau, the time step fails.

MaxDelta and MaxTau are then used to compute the next time step.

7.5 Lines 27-31: Good Time Step

If a time step is deemed to be acceptable, the current time (t) is updated, the last known good coherence vector for a cell and the first column of the cell's K-matrix are set to the last (7th) column of the current K-matrix, and the cell's polarization is set to the inverse of the 7th element of the new coherence vector.

We also examine the output cells to see if any of the output cell array entries need to be updated. Currently, if an output cell sees a clock value greater than or equal to $0.95*Ek$, its entry in the output cell array is updated. An update in the output cell array consists of incrementing the number of time steps for the entry by one and adding the output cell's polarization to the polarization sum of the entry.

7.6 Line 32: Update Time Step

This step is again taken directly from the DP method in ode45. If MaxDelta is zero, it is set to 1×10^{-16} . The time step (h) is then set to the minimum of either the maximum time step or $0.8 * h * (MaxTau/MaxDelta)^{(1/6)}$.

8 Remainder of Run Simulation (Lines 34 and 35)

Once the simulation has run to completion we determine if the circuit has passed or failed. To make this determination, we examine each entry in the output cell array. For each of these entries we divide the polarization sum by the number of time steps. This provides an average polarization for when the clock signal seen by this cell was high. If this average polarization is within 5% of the desired polarization, this output cell is considered to have passed. If all output cells have passed, the circuit is determined to have passed. If any output cell did not pass, the circuit is considered to have failed.

After the pass/fail status for a given polarization (-1) has been determined, we clear the output cell array (reset polarization sum and number of time steps to zero) and then repeat the run simulation function for the opposite polarization (+1).

9 Remainder of Program (Line 36)

In this segment of the program, we attempt to clear the memory used by the program. At the current time, no testing has been done to determine if all memory is properly cleared and that the program is free of memory leaks.

References

- [1] J. Timler and C. S. Lent, "Maxwell's demon and quantum-dot cellular automata," *Journ. of Applied Physics*, vol. 94, no. 2, pp. 1050–1060, July 2003.
- [2] J. P. Timler, "Energy dissipation and power gain in quantum-dot cellular automata," Ph.D. dissertation, U. of Notre Dame, 2003.
- [3] M. Liu, "Robustness and power dissipation in quantum-dot cellular automata," Ph.D. dissertation, University of Notre Dame, 2006.
- [4] T. J. Dysart, "It's all about the signal routing: Understanding the reliability of QCA circuits and systems," Ph.D. dissertation, U. of Notre Dame, 2009.
- [5] T. J. Dysart and P. M. Kogge, "XML Based File Format for QCADesigner," U. of Notre Dame, Dept. of Computer Science and Engineering, Tech. Rep. 26, 2004.
- [6] E. Blair, "Tools for the design and simulation of clocked molecular quantum-dot cellular automata circuits," Master's thesis, Univ. of Notre Dame, Dept. of Electrical Engineering, November 2003.
- [7] C. S. Lent, M. Liu, and Y. Lu, "Bennett clocking of quantum-dot cellular automata and the limits to binary logic scaling," *Nanotechnology*, vol. 17, pp. 4240–4251, 2006.