A Semiconductor Device Simulator Utilizing MATLAB^{TM}

[1]Hamid Fardi, [1]Shawn Pace, and [2]Gita Alaghband

[1]University of Colorado Denver
Department of Electrical Engineering
Campos Box 110., P.O. Box 173363, Denver, CO 80217

[2]University of Colorado Denver
Department of Computer Science and Engineering
Campos Box 109., P.O. Box 173363, Denver, CO 80217

ABSTRACT- The purpose of this project is to develop a general purpose semiconductor device simulator that is functional and modular in nature in order to allow for flexibility during programming and to allow for future development with relative ease. In addition, the program's main goal is to provide a tool that can supplement semiconductor device modeling, semiconductor device physics, and to construct basic semiconductor device equations using MATLAB tools. MATLAB's capability and inherent nature of handling matrices and matrix operations makes this approach an excellent technique to develop numerical analysis algorithms. A device modeling program is developed using the basic MATLAB tools necessary to understand the operation of the program.

This project was developed in the programming environment provided by MATLAB™ which is developed and supported by Mathworks [1]. Almost all academic institutions use MATLAB heavily in the learning process and is readily available to most students, hence the choice of this programming environment. Additionally, MATLAB is an extremely powerful numerical analysis tool that makes the solution of a program like this one rather straightforward. MATLAB's capability and inherent nature of handling matrices and matrix operations makes this an excellent tool to develop numerical analysis algorithms.

The current semiconductor simulation tools available do not lend to updates or modification simple or straightforward. In fact most tools are not open to be changed in any manner unless the source code is available. With the open nature of MATLAB the solution process can be studied throughout and all variables and parameters are available to study. The program does have the capability to be standalone which allows for the use of the tool without the requirement of the MATLAB environment.

Semiconductor Device Simulator (SDS)

In this Section, a brief overview of the SDS program is given with explanations for key aspects of the program. We will begin by looking at the "main" program and then investigate the functions that are implemented in support of the main program.

The goal is to write a program that solves the matrix-vector equation which was established from the classic differential equations. In most aspects the program can be developed however the developer likes, but in certain instances poor coding techniques will lead to longer computation times and ultimately slower convergence speeds for a given device setup. The main important goal is to manipulate the matrix coefficients efficiently and to solve the matrix-vector equation with as little iteration as possible. On key element to this is establishing the proper boundary conditions and trial values [2].

A basic flow diagram of the main program is given below [3]. The first few blocks simply establish any constants, or numerical parameters required to solve the device. The second block establishes any user defined inputs which also include device information for the device to be simulated.

```
clear all;                %Clears all previously stored variable information
close all;                %Closes all previously open plots
clc;                      %Clears the command window to blank
TSTART = tic;             %Starts a timer to record solution time
format long eng;          %Formats the data to engineering mode
```

This block is a simple setup that is used before any new simulation is run. All stored variable information in the Matlab workspace is cleared, all open plots are closed, and the command window is cleared. This section also starts a timer, which is used to investigate the speed of the computation algorithm for various device setups.

Every calculated value that is stored in a variable is accessible after the program is run in the main Matlab workspace. For example, the last iterated value for the error values is stored in an error vector. The program can be easily modified to run through one iteration at a time allowing the user to investigate the change in error over each iteration. Every opportunity for the user to access the calculated data has been provided so that any insight into those variables can be gained if necessary.

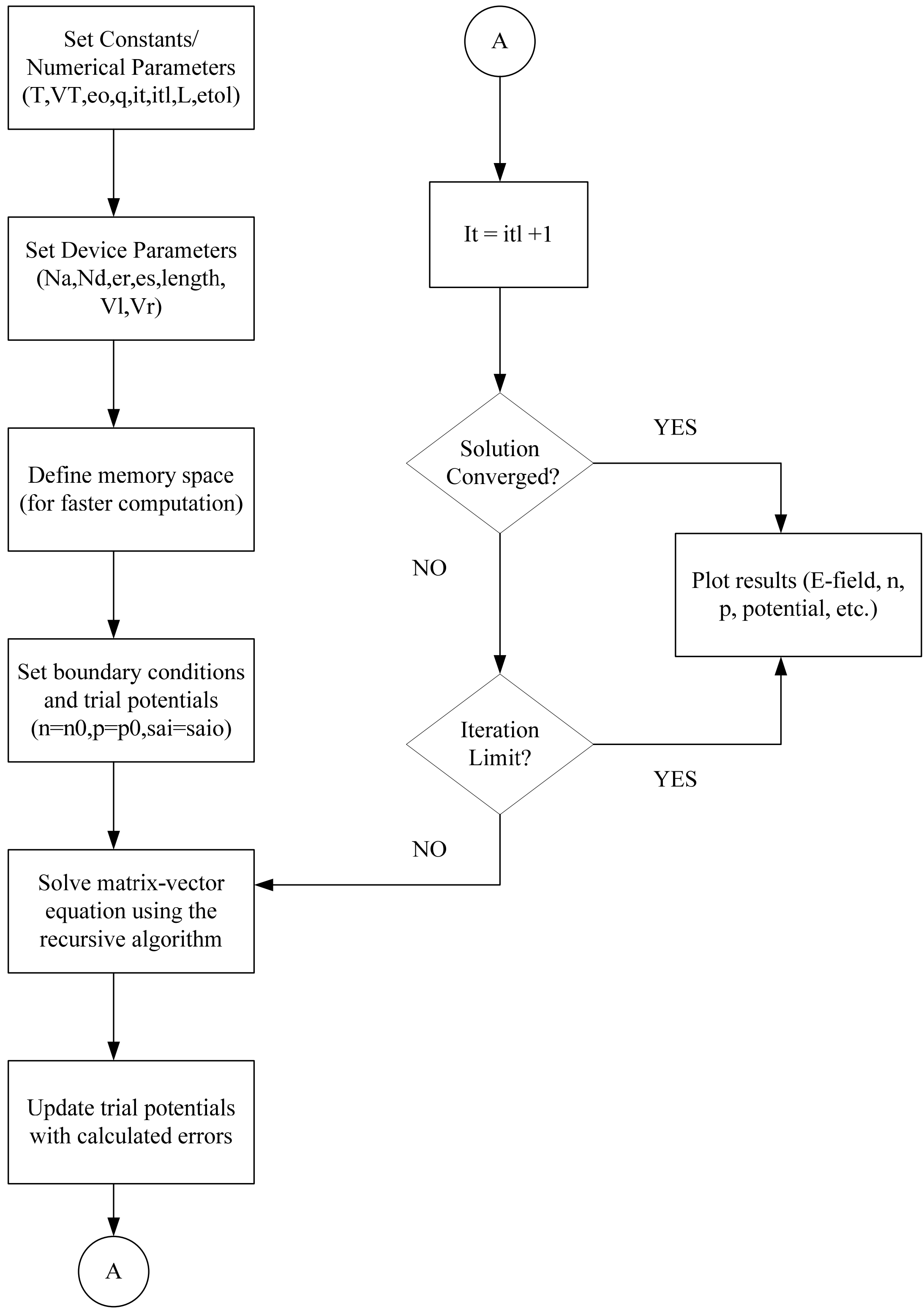

Figure 1 – SDS Basic Flowchart

One of the main reasons the author chose to use Matlab was due to the environments natural capability to work with matrices. Everything in Matlab, even a one-dimensional variable or constant is treated as a matrix. One of the key features in SDS is the use of Cell arrays in Matlab. This allows the user to store various types of data in each cell. In fact, the first cell could have a standard 3x3 matrix, while the second cell has a string text in it for example.

The cell arrays are treated like matrices as well and indexed similarly. All standard matrix operations work, provided that matrix dimensions agree, etc. This works elegantly with the tri-diagonal block matrices. Each matrix coefficient is a 3x3 matrix but the calculated values are stored in a cell array where each element is a 3x3 matrix. The cell array is used in conjunction with the recursive algorithm for the solution process. The delta_y array, yo, and y arrays are cell arrays as well and the instantiation process is shown below.

```
delta_y = cell(1,L);            %Defines an empty cell from 1 to L
delta_y(:) = {[0;0;0]};         %Initializes each cell to a 3x3 empty matrix
yo = cell(1,L);                 %Defines an empty cell from 1 to L
yo(:) = {[0;0;0]};              %Initializes each cell to a 1x3 empty matrix
y = cell(1,L);                  %Defines an empty cell from 1 to L
y(:) = {[0;0;0]};               %Initializes each cell to a 1x3 empty matrix

Apr = cell(1,L);
Bpr = cell(1,L);
Cpr = cell(1,L);
Fpr = cell(1,L);

delta_y{N} = inverse_func(Bpr{N})*Fpr{N}...
            - inverse_func(Bpr{N})*Cpr{N}*delta_y{N+1};
```

This calculates the error during one iteration. Following the matrix dimensions gives

delta_y{N} = 3x3 X 3x1 – 3x3 X 3x3 X 3x1 = 3x1, which agrees.

This method of using cell arrays is efficient and compact and is highly recommended. Even the update process is compact as shown below. In one line, the original trial potential, yo{N} is updated with the calculated error in delta_y{N} at each division point N. Remember the boundary conditions have no error associated with them as they are fixed.

```
y{N} = yo{N} + delta_y{N};
```

After each iteration, the solution needs to be checked against the error tolerance. There is effectively two main ways to accomplish this. The first method requires that the absolute value of each calculated error value for the three fundamental variables, n, p, and sai, are checked against the convergence tolerance at each division point. If every single point has converged then

the solution has been reached. This is somewhat inefficient, however is accurate for the whole device. A second method checks the normalized value of the entire error vector (normalized value of the error at each division point) against the convergence tolerance. If the normalized value is small enough the solution has converged. The process for one error vector is shown below.

```
temp1 = abs(norm(p_error)/norm(po));

if (temp1 > etol)
    flag1 = 1;
else
    flag1 = 0;
end
```

Once all three flags have been cleared to zero (they are initially set to one), then the device has converged. The solution algorithm loops until this occurs or the iteration limit set by the user has been reached. The default is 15 iterations.

The final part of the “main” code or function simply plots the results, whether the device solution converged or not. The electric field, carrier densities, and device potential are all plotted against the spatial variable, x.

The rest of the SDS program is a series of functions that are called by the “main” function or sub-functions. This is where the matrix coefficients are calculated and the partial derivatives as well. The code for one element of the B_func is shown below. It references several functions itself.

```
b(1,1)= ( 1/hpr(N) )*((mu_func(M,h,po,no,Ntot,saio,1)/h(M))...
    *lambda_func(M,saio,1)-(mu_func(M-1,h,po,no,Ntot,saio,1)/h(M-1))...
    *lambda_func(M-1,saio,-1))+ diffsrh_func(N,po,no,Ntot,1);
```

It can be seen that there are several functions to support the calculation of the matrix coefficient. Mobility is calculated through a function so that models of temperature and effective doping can be used, and the lambda terms and recombination terms are supported by function calls as well. This allows for efficient storage of the calculated data. We don’t need the lambda value stored across the whole device because unless you are troubleshooting, there is no interest in these values.

The lambda function code is shown below. This way beta(M) and beta(M-1) can be calculated in one function by passing in the current division point. The other important note for the lambda function is the use of the variable, I. This variable is passed in so that the proper sign is used for the exponential terms. In this manner, one function can support all of the lambda terms.

```
Vt = 0.02586;
theta = 1/Vt;

temp = beta_func(M,saio);

if(I == 1)
    if(abs(temp) < 1E-5)
        lambda_out = (1/theta)*(1/(1-(1/2)*temp+(1/6)*temp^2));
    else
        lambda_out = (1/theta)*(temp / (1 - exp(-temp)));
    end
elseif(I == -1)
    if(abs(temp) < 1E-5)
        lambda_out = -(1/theta)*(1/(1+(1/2)*temp+(1/6)*temp^2));
    else
        lambda_out = (1/theta)*(temp / (1 - exp(temp)));
    end

end

end
```

One final note on this function is that for special cases, the actual value of lambda needs to be approximated or some of the resulting matrix coefficient calculations will become undefined due to a division by zero. For the default device in this program, a step junction profile is used. This results in equal values of the device potential in each bulk region which will cause the output of the lambda function to go to zero or undefined. Therefore, depending upon the value of beta (which is the difference of the device potential between two division points), instead of using the direction equation and approximated equation is used.

In essence a series expansion of the lambda terms are performed so that the exponentials are approximated. The technique used here was known as Bernoulli's function.

SDS Example

In this section, we will review the output of the SDS program applied to a P-N junction semiconductor device, shown in Figure 2, to demonstrate the conceptual idea and implementation of the program.

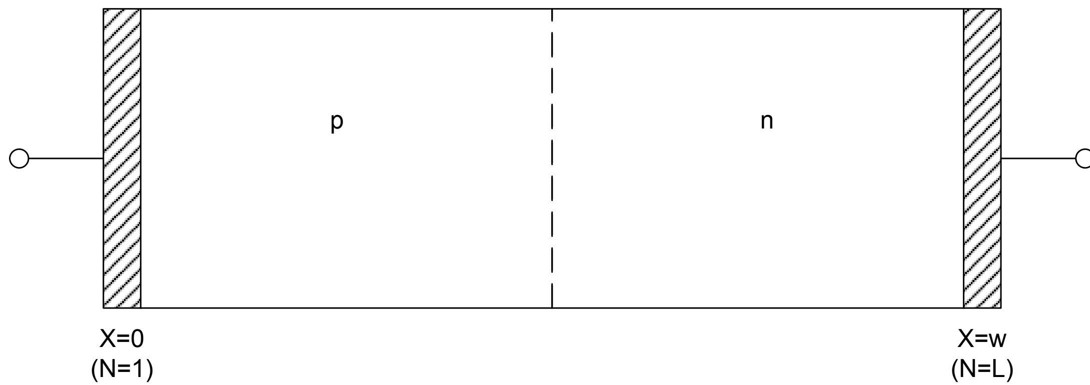

Figure 2 – Basic P-N Junction Diode

Figure 3 shows the output for the potential of the SDS program with an abrupt step junction of equal doping in both regions. The cursor in the graph is displaying the pertinent information.

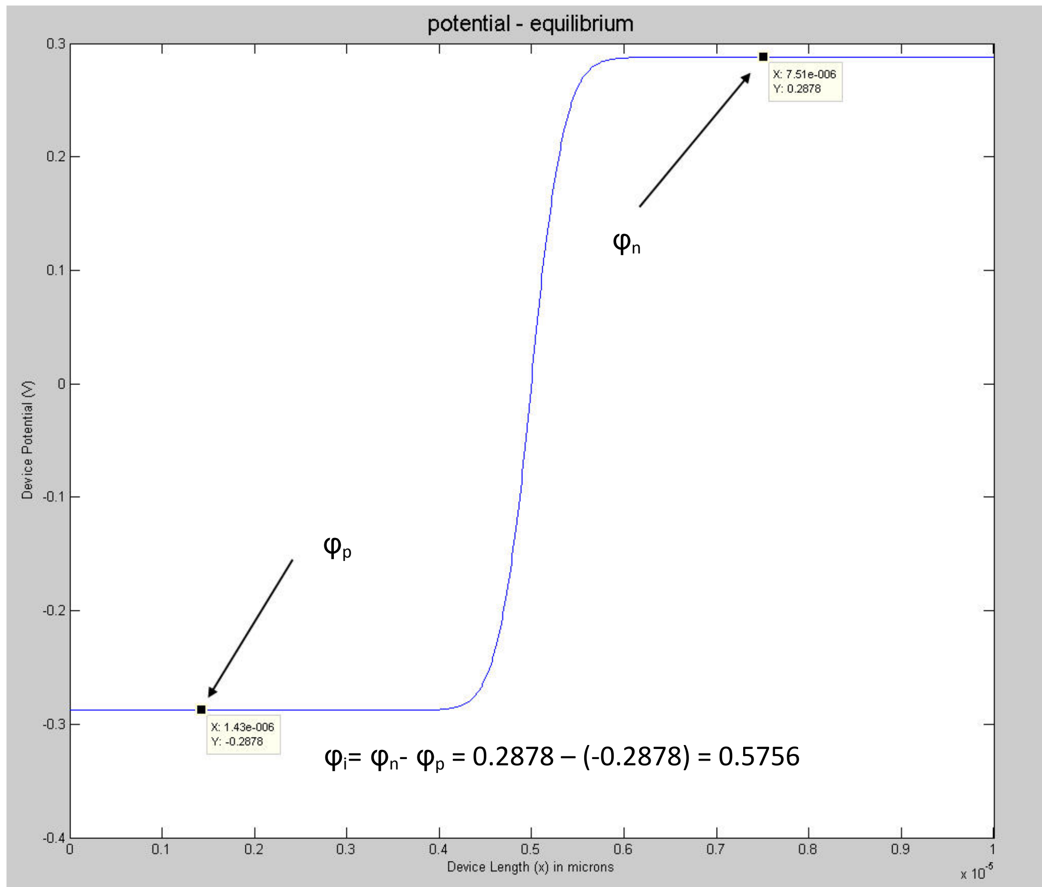

Figure 3 – Equilibrium device potential

We can also observe in Figure 3 that the potential is zero at the junction transition as we expect since the doping densities are equal in the bulk n and p-regions [4].

Bibliography

[1] The Mathworks, Inc. 1994-2014, Matlab.

[2] M. Kurata, *Numerical Analysis for Semiconductor Devices,* Lexington, D.C. Heath and Company, 1982.

[3] M. Pinto *et al.*, "PISCES-II," Dept. of Elec. Engr., Stanford Univ., California, Sept. 1984.

[4] H. Fardi, et al., "Numerical Modeling of energy balance equations in Quantum Well AlGaAs/GaAs p-i-n photodiodes," Transactions on Electron Devices, vol. 47, no. 4, 2000.