ESspice(2022)

Ethan Sifferman

[[1]](#footnote-1) ***Abstract*—This C++ implementation of Spice (created for ECE 594BB Fall 2022 taught by Peng Li) supports DC and Transient analysis of several linear and nonlinear circuit elements including voltage sources, current sources, resistors, capacitors, inductors, N-MOSFETS, and P-MOSFETS. This tool works by Modified Nodal Analysis (MNA) and the matrix multiplication library Eigen. It has an interface to MATLAB to allow plotting, and the netlist generation is separate from the simulation.**

**This design is posted to GitHub under** [**https://github.com/sifferman/ESspice**](https://github.com/sifferman/ESspice)**.**

# I. Introduction

This tool is derived from the code used for the ECE 594BB circuit simulator assignment. It has been heavily refactored to follow better organization practices for hardware simulation tools. The biggest issue with the previous code was that it did not isolate the netlist from the simulation done on the netlist. Now, the updated code allows for generic netlist generation, regardless of simulation type, and any simulation applied to the netlist is portable and does not affect the inputted netlist.

Additionally, the previous code had several computations and algorithms implemented in the main simulation loops, causing long and confusing code. This has been corrected by isolating MNA stamping of circuit elements, Norton equivalence generation, and non-linear device equations into separate functions, allowing for a cleaner implementation.

Finally, several bug-fixes have been implemented, including proper support for intrinsic capacitances and resistances, fixed piecewise-linear power-sources outputs, and divide by zero errors avoided. These bug-fixes will not be described further in this document, though they are important to the overall functionality of ESspice.

The result of all these changes is that the user-experience is much better. ESspice now has support for interfacing with MATLAB, allowing users to generate a netlist, run simulations, and plot the results all from the same C++ script.

# II. Running The Code

You must run ESspice via writing a C++ script. There are three classes you must instantiate: “matlab”, “circuit”, and “analysis”. “matlab” creates a MATLAB interface to call the circuit parser and plot generator. “circuit” specifies the netlist of your circuit. “analysis” lets you plot any node voltages or element currents according to the specified analysis type. The following is an example script:

|  |
| --- |
| #include "circuit.h"  #include "matlab.h"  #include "analysis.h"  #include <iostream>  #include <string>  #include <cstdio>  using namespace std;  int main(int argc, char const \*argv[]) {      if (argc!=2) {          cerr << "Usage: ./main <.ckt file>" << endl;          return 1;      }      // create matlab instance      matlab m;      // load circuit      string js = m.ckt\_to\_json(argv[1]);      circuit a{js};      cout << "Finished loading circuit." << endl;      // Run circuit      analysis \* run = a.run(&m);      run->plotnv(&m, 1);      // Allow time to look at figures      cout << "Press return to exit...";      getchar();      cout << "Exiting." << endl;      return 0;  } |

# III. MATLAB Interface

Although the tool is primarily written in C++, there are 2 MATLAB functions that need to be called: “ckt\_to\_json()” and “show\_plot()”. To avoid the user needing to run multiple scripts to use ESspice, ESspice provides a MATLAB interface file that instantiates a MATLAB engine and allows the user to run the two functions via a C++ function. The API for the MATLAB interface is as follows:

|  |
| --- |
| class matlab {      matlab(); // start MATLAB engine      ~matlab(); // stop MATLAB engine      std::string ckt\_to\_json(std::string filename);      void show\_plot(std::vector<double> data, std::string title, std::string xlabel, std::string ylabel, double xtick, double xlim);  }; |

Instantiating an entire MATLAB engine is obviously not the best way to perform the simple tasks required in “ckt\_to\_json()” and “show\_plot()”, due to the high overhead of sending commands to MATLAB. Fortunately, MATLAB has support for translating MATLAB code to C++ code, eliminating the need for MATLAB altogether. However, MathWorks has poor documentation for this feature, so I settled on the current method. In the future, this would be a great feature to support.

# IV. Class Hierarchy

The class hierarchy was quite challenging to get right. In the last report, I provided a description of the circuit elements. However, in this version of the code, all the calculated voltages and currents have been removed from the element classes and placed into the respective analysis class.

The analysis class and its derived classes store all information and algorithms required to simulate a given netlist. The primary shared members between all derived classes are the “plot” functions and the list of intrinsic linear elements, (such as parasitic capacitance and leakage current).

The following is a summary of the contents of the “analysis” class.

|  |
| --- |
| class analysis {  public:      analysis(circuit \* c);      enum TYPE\_t {          DC      = 0,          TRAN\_FE = 1,          TRAN\_BE = 2,          TRAN\_TR = 3      };      virtual void plotnv(matlab \* m, int node\_name) = 0;      virtual void printnv(int node\_name) = 0;  private:      static constexpr double precision = 1.0e-9;      circuit \* c;      std::vector<circuit::linelem\*> itrelems;  }; |

Currently, there are two derived analysis classes: “dc” and “tran” for DC and transient analysis respectively. The notable members that “dc” adds to “analysis” include voltage and current values for each node and device, the MOSFET equations, and the linear and non-linear stamping functions for MNA matrices.

The following is a summary of the contents of the “dc” class.

|  |
| --- |
| class dc : public analysis {  public:      dc(circuit \* c);      void plotnv(matlab \* m, int node\_name) const;      void printnv(int node\_name) const;      double voltage(circuit::node \* n) const;      double voltage(circuit::linelem \* e) const;      double current(circuit::linelem \* e) const;  private:      std::unordered\_map< circuit::node\*, double > node\_voltage;      std::unordered\_map< circuit::capacitor\*, double > capacitor\_current;      std::unordered\_map< circuit::V\_source\*, double > V\_source\_current;      struct mosfet\_solver {          static double conductance(circuit::mosfet \* m, double V\_GS, double V\_DS);          static double I\_DS(circuit::mosfet \* m, double V\_GS, double V\_DS);          static double NR\_G\_eq(circuit::mosfet \* m, double V\_GS,const double V\_DS);          static double NR\_I\_eq(circuit::mosfet \* m, double V\_GS,const double V\_DS);      };      void stamp\_A(Eigen::SparseMatrix<double> A, circuit::linelem \* e) const;      void stamp\_z(Eigen::SparseMatrix<double> z, circuit::linelem \* e) const;      void stamp\_A\_NR(Eigen::SparseMatrix<double> A\_NR, circuit::mosfet \* m, double V\_GS, double V\_DS) const;      void stamp\_z\_NR(Eigen::SparseMatrix<double> z\_NR, circuit::mosfet \* m, double V\_GS, double V\_DS) const;  }; |

As for the “tran” class, most of the header file is the same as the “dc” class, however the implementations of the voltage and current getters, and the stamping functions of the circuit elements are completely different. In the header file, the only visible differences between “dc” and “tran” are that “tran” now lets you retrieve a voltage or current at a given time step, and capacitors and inductors need to be represented in their Norton equivalence to be solved correctly.

The following is a summary of the contents of the “tran” class.

|  |
| --- |
| class tran : public analysis {  public:      tran(circuit \* c, double time\_step, double stop\_time);      void plotnv(matlab \* m, int node\_name) const;      void printnv(int node\_name) const;      double voltage(circuit::node \* n, int t);      double voltage(circuit::linelem \* e, int t) const;      double current(circuit::linelem \* e, int t) const;  private:      std::unordered\_map< circuit::node\*, std::vector<double> > node\_voltage;      std::unordered\_map< circuit::storage\_device\*, std::vector<double> > storage\_device\_current;      std::unordered\_map< circuit::V\_source\*, std::vector<double> > V\_source\_current;      struct storage\_device\_solver {          static double conductance(circuit::capacitor \* c, double time\_step);          static double conductance(circuit::inductor \* l, double time\_step);      };      struct mosfet\_solver {          static double conductance(circuit::mosfet \* m, double V\_GS, double V\_DS);          static double I\_DS(circuit::mosfet \* m, double V\_GS, double V\_DS);          static double NR\_G\_eq(circuit::mosfet \* m, double V\_GS, double V\_DS);          static double NR\_I\_eq(circuit::mosfet \* m, double V\_GS, double V\_DS);      };      void stamp\_A(Eigen::SparseMatrix<double> A, circuit::linelem \* e) const;      void stamp\_z(Eigen::SparseMatrix<double> z, circuit::linelem \* e) const;      void extract\_current(Eigen::SparseMatrix<double> x, circuit::linelem \* e);      void stamp\_A\_NR(Eigen::SparseMatrix<double> A\_NR, circuit::mosfet \* m, double V\_GS, double V\_DS) const;      void stamp\_z\_NR(Eigen::SparseMatrix<double> z\_NR, circuit::mosfet \* m, double V\_GS, double V\_DS) const;      double time\_step;      double stop\_time;  }; |

By separating each analysis type into different classes, adding another analysis type in the future becomes trivial because it will not interfere with any of the previous code. Additionally, the separation shows exactly which algorithms are required for which analysis. Following this organization style gives ESspice the potential to be used and developed by others in the future.

# V. Code Performance

Using C++ std::chrono, I measured the runtime analyzing each of the circuits in the “benchmark\_circuits” suite provided in the ECE 594BB starter code. The following table shows the time to run ESspice for each of the “benchmark\_circuits” at different levels of Newton-Raphson precision.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Time (seconds) for given Newton-Raphson Precision | | |
| Circuit Name | 1.0e-3 | 1.0e-6 | 1.0e-9 |
| *rc\_line* | *0.165* | | |
| *rlc\_line* | *0.290* | | |
| *rcmesh20* | *14.970* | | |
| *test\_inv* | *0.390* | *0.510* | *0.594* |
| *transmux* | *0.991* | *1.902* | *2.334* |
| *clocktree* | *1.170* | *2.340* | *2.798* |
| *nand3* | *0.917* | *1.979* | *3.097* |

We can observe from this graph that adding MOSFETs greatly increased the runtime of the simulations. Therefore, we can improve performance in a future version of ESspice by optimizing non-linear circuit analysis. Notably, there are optimizations that can be done for factoring the Newton-Raphson matrices.

As for a general optimization for all simulations, the obvious feature to add is dynamic step sizes. This would improve the speed of simulating circuits that reach a steady-state and improve the speed of simulating circuits that have nodes with changing impulse-responses.

# VI. Reflection

This project of refactoring my ECE 594BB circuit simulator code into a more organized and therefore scalable code base ended up being much more work than anticipated. It took approximately 60 hours to implement the base algorithms and class hierarchy, and approximately 40 more hours to create the MATLAB interface and separate out the analysis types from the netlist.

However, now ESspice has a formidable user-experience and lends itself to being improved upon in the future. By completing this project, I now have a much stronger understanding of how hardware can be described and verified with software. For example, I now have a better understanding for why [Yosys](https://github.com/YosysHQ/yosys), the leading open-source RTL synthesis tool places such an emphasis on separating its netlist types from its synthesis algorithms.

Overall, I am incredibly proud of my work and hope to use what I have learned from this project in as many ways as I can.

1. [↑](#footnote-ref-1)