

AMERICAN UNIVERSITY OF BEIRUT

ROBUSTNESS GUIDED VERIFICATION

by

RAMZI MOHAMAD SABRA

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
June 2020

AMERICAN UNIVERSITY OF BEIRUT

ROBUSTNESS GUIDED VERIFICATION

by

RAMZI MOHAMAD SABRA

Approved by:



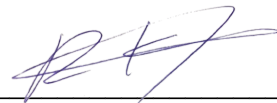
Prof. Ali Chehab, Professor
Electrical and Computer Engineering

Advisor



Prof. Mohamad Mansour, Professor
Electrical and Computer Engineering

Member of Committee



Prof. Rouwaida Kanj, Associate Professor
Electrical and Computer Engineering

Member of Committee

Date of thesis defense: June 22, 2020

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: _____Sabra_____Ramzi_____Mohamad_____

Last

First

Middle

Master's Thesis Master's Project Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:

One -X- year from the date of submission of my thesis, dissertation, or project.

Two ---- years from the date of submission of my thesis, dissertation, or project.

Three ---- years from the date of submission of my thesis, dissertation, or project.



Signature

15-7-2020

Date

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Ali Chehab, for his guidance throughout my Masters degree. I would like to thank Dr. Mohamad Mansour, whom, along with Dr. Chehab, planted the seedlings that budded into this thesis and offered valuable insights into research and industry trends; I would also like to thank Dr. Rouwaida Kanj for her sharing her extensive experience with electric circuits which brought to fruition the experiments we performed.

I would like to thank my parents for their unwavering support regardless of the difficulty of circumstances; for pushing me to persevere, often times despite myself. I would like to thank my partner in life and in crime, Imane, for her continuous, consistent presence in my waking moments; for driving me to dedicate my time to my academic endeavors, among all; for her understanding, coping, and dealing with my mental state, in all its intricacies and tunnels.

AN ABSTRACT OF THE THESIS OF

Ramzi Mohamad Sabra for

Master of Engineering

Major: Electrical and Computer Engineering

Title: Robustness Guided Verification

Exhaustive and thorough testing is the ideal form of testing for any system; it would not be possible for such a system to fail when all possible outcomes of its operation are known to succeed. However, with complex systems where the factors can be practically infinite, exhaustive testing is not feasible nor efficient. Novel approaches to testing systems and verifying that they adhere to their specifications are much needed. These approaches have to be able to test a wide variety of systems without necessarily knowing how these systems work. Such approaches to testing could potentially expose failures in systems with certain conditions that the tester could not have possibly imagined and consciously tested for. However, such approaches would be delegated to testing systems of high complexity, often with practically infinite parameter spaces. Therefore, testing algorithms have to be able to work with a limited set of possibilities, aiming to discover areas in which certain combinations of input parameters cause a failure in the system under test. Rare fail estimation is of particular importance in non-volatile memory cells such as the STT-MTJ based latch. Applying such novel approaches to non-volatile memory cells may accelerate yield estimation beyond what traditional tools are capable of. However, multiple tools are required to interoperate to integrate simulation of electric circuits with frameworks that implement such approaches. The tools required need to easily integrate Verilog-AMS models into ngspice, parametrize SPICE circuits from code, run several SPICE simulations in parallel. This allows to compute the ground truth in order to verify the accuracy of the Active Learning algorithm's predictions, run SPICE simulations from within the Active Learning framework, be efficient with resource usage (such as memory), and execute in minimal time in order for the approach to be useful over more traditional approaches.

CONTENTS

ACKNOWLEDGMENTS.....	5
ABSTRACT.....	6
ILLUSTRATIONS.....	9
Chapters	
1. INTRODUCTION.....	1
2. LITERATURE REVIEW.....	3
2.1 Temporal Logic Falsification for Hybrid Systems.....	3
2.2 Active Learning Framework.....	6
2.3 Rob-Guided-Verif.....	7
2.4 NVLatch Benchmark.....	8
2.4.1 Introduction.....	8
2.4.2 Circuit Diagram.....	9
2.4.3 Math Equations.....	9
2.4.4 Correct Behavior.....	10
2.4.5 Search Space Dimensions.....	10
2.4.6 System Inputs.....	11
2.4.7 Benchmark Significance.....	12
2.5 Yield Estimation via Multi-Cones.....	12
2.6 ngspice.....	13
2.6.1 ADMS.....	14
2.7 mp-units.....	14
3. PROBLEM DESCRIPTION.....	16
4. IMPLEMENTATION.....	18
4.1 mp-units.....	18
4.1.1 ASCII-Only Output Support.....	18
4.1.2 Alias Units.....	19
4.1.3 Random Number Distributions.....	20
4.2 ngspice C++ wrapper.....	21
4.2.1 Simulation Object.....	21
4.2.2 Feedback Simulation.....	21
4.2.3 Explicit Parallelism.....	22
4.3 SPICE Circuit Generator.....	23
4.3.1 Grammar.....	23

4.3.2 Code Generation.....	24
4.4 ngspice STT-MTJ Integration.....	25
4.5 Monte Carlo.....	26
4.6 Yield Estimation via Multi-Cones.....	27
4.7 Robustness Guided Verification (C++).....	27
4.7.1 Language Features.....	28
4.7.2 Dependencies.....	28
4.7.3 Models.....	29
4.7.4 ODE Solver.....	29
4.7.5 Input Signal Interpolation.....	30
4.7.6 Ground Truth Sweepers.....	30
4.7.7 Estimation Algorithms.....	30
4.7.7.1 Distance Metrics.....	31
4.7.7.2 Certainty Metrics.....	31
4.7.7.3 Candidate Selection Metrics.....	31
4.7.8 Active Learning Algorithm.....	31
4.7.9 Benchmarks.....	32
4.7.9.1 quadprog.....	32
4.7.9.2 ode15s.....	32
4.7.9.3 compute_robustness.....	32
4.7.9.4 nonlinear.....	32
4.7.9.5 nonlinear_al.....	32
4.7.9.6 Results.....	33
4.7.10 Verification.....	33
5. EXPERIMENTAL RESULTS.....	37
5.1 STT-MTJ 5.4ns Nominal Delay.....	39
5.2 STT-MTJ 5.6ns Nominal Delay.....	42
6. FUTURE WORK.....	46
6.1 SPICE Circuit Generator.....	46
6.2 ngspice STT-MTJ Integration.....	46
6.3 Robustness Guided Verification (C++).....	46
7. CONCLUSION.....	47
BIBLIOGRAPHY.....	48

ILLUSTRATIONS

2.1 Room heating benchmark HEAT30.....	5
2.2 Sample run of S-TaLiRo.....	8
2.3 Circuit diagram of a non-volatile latch.....	9
4.4.1 P -> AP Transition (manual).....	26
4.4.2 P->AP Transition (ngspice).....	26
4.4.3 AP -> P Transition (manual).....	26
4.4.4 AP -> P Transition (ngspice).....	26
4.7.9.6.1 MATLAB vs C++ Benchmark Results.....	33
4.7.10.1 “nonlinear” example robustness (1).....	34
4.7.10.2 “nonlinear” example robustness (2).....	34
4.7.10.3 “nonlinear” example robustness (3).....	35
4.7.10.4 “bmk_modulator” example robustness (1).....	35
4.7.10.5 “bmk_modulator” example robustness (2).....	36
4.7.10.6 “bmk_modulator” example robustness (3).....	36
5.1 STT-MTJ circuit, 5.4ns nominal delay – Gaussian Random Monte Carlo Sweeper Ground Truth – 1,000,000 points.....	38
5.2 STT-MTJ circuit, 5.4ns nominal delay – Grid Sweeper Ground Truth – 300x300 (90,000) points	38
5.1.1 Gaussian Random Monte Carlo Yield Estimate.....	40
5.1.2 Gaussian Random Monte Carlo Convergence Estimate.....	40
5.1.3 Multi-Cones Yield Estimate.....	41
5.1.4 Multi-Cones Convergence Estimate.....	41
5.1.5 Active Learning Accuracy / Certainty.....	42
5.2.1 Monte Carlo Yield Estimate.....	43
5.2.2 Monte Carlo Convergence Estimate.....	43
5.2.3 Multi-Cones Yield Estimate.....	44
5.2.4 Multi-Cones Convergence Estimate.....	44
5.2.5 Active Learning Accuracy / Certainty.....	45

CHAPTER 1

INTRODUCTION

In our current technological world, where various systems are relied on daily by billions of users worldwide, safety is a vital aspect of any technological system to be accepted, to succeed, and to proliferate. System failure is a common concern for any technology maker; a manufacturer has to avoid critical failure whenever possible. Every technological component can fail, whether it is a matter of time or simply a matter of being exposed to scenarios that are too stressful for proper and correct operation. Often, with complex systems, failures can manifest themselves in unforeseen situations, untested combinations of factors that passed deployed testing methodologies without alarm.

Exhaustive and thorough testing is the ideal form of testing for any system; it would not be possible for such a system to fail when all possible outcomes of its operation are known to succeed. However, with complex systems where the factors can be practically infinite, exhaustive testing is not feasible nor efficient.

Autonomous cars are such an example. A tester might perform evaluations of the object recognition system based on specific factors such as size, shape, proximity, or speed. However, untested scenarios whereby a failure would have been discovered are particularly disastrous; hackers were able to fool Tesla's Model S in Autopilot mode to perceive objects that weren't there and even miss objects that were^[1]. In other situations, the same car model seems to have a blind spot, which already lead to a fatal accident^[2]. This immediately implies that blind spots were not a factor that the testers had involved or focused on while laying down their test suites.

Novel approaches to testing systems and verifying that they adhere to their specifications are much needed. These approaches have to be able to test a wide variety of systems without necessarily knowing how these systems work. Such approaches to testing could potentially expose failures in systems with certain conditions that the tester could not have possibly imagined and consciously tested for. However, such approaches would be delegated to testing systems of high complexity, often with practically infinite parameter spaces. Therefore, testing algorithms have to be able to work with a limited set of possibilities, aiming to discover areas in which certain combinations of input parameters cause a failure in the system under test.

Rare fail estimation is of particular importance in non-volatile memory cells such as the STT-MTJ based latch. ^[11] Applying such novel approaches to non-volatile memory cells may accelerate yield estimation beyond what traditional tools are capable of. However, multiple tools are required to interoperate to integrate simulation of electric circuits with frameworks that implement such approaches.

CHAPTER 2

LITERATURE REVIEW

2.1 Temporal Logic Falsification for Hybrid Systems

Temporal verification is concerned with proving or falsifying temporal logic properties of systems under analysis^[4]. In [4], Annapureddy et al. develop a tool called S-TaLiRo that is used for temporal logic falsification of non-linear hybrid systems. S-TaLiRo searches for counterexamples to the Metric Temporal Logic (MTL) properties of the system through the global minimization of a robustness metric. This global optimization is carried out by stochastic optimization techniques that perform a random walk over the initial states, controls, and disturbances of the system in order to obtain a robustness score. S-TaLiRo simulates the system and returns the trace with the smallest robustness value found. A negative robustness score represents a falsification of the temporal logic properties. A low robustness score indicates proximity to falsifying traces.

S-TaLiRo is available as a MATLAB toolbox with a Command-Line Interface. It can interact with any system as long as its simulator is implemented as a Simulink Stateflow model, as a MATLAB function, or in any other framework that provides a MATLAB interface. Currently, S-TaLiRo provides Monte Carlo and Ant Colony Optimization as stochastic optimization algorithms. It is also possible to substitute the built-in robustness metric computation algorithms with external ones.

S-TaLiRo consists of a temporal logic robustness analysis engine and a stochastic sampler. The sampler feeds in input parameters to the simulator and the output results are analyzed by the robustness analysis engine in order to return a robustness score. The score is then used by the sampler to decide what next input to feed into the simulator. Results for which the computed

robustness score is negative, those that falsify the temporal logic properties of the system, are reported to the user. If no falsification is found, the least robust trace, that with the lowest robustness score, is reported.

Several benchmarks are provided with the toolbox as a proof-of-concept. One of interest is the HEAT30 room heating benchmark from, shown in Figure 1 [5]. The benchmark involves 10 rooms with 3,360 discrete locations and 4 heaters. Each room starts at a temperature in the range [17, 18] (initial conditions) and the input signal is in the range [1, 2]. The temporal logic of the system states that no room should ever drop below its threshold temperature in [14.5 14.5 13.5 14.0 13.0 14.0 14.0 13.0 13.5 14.0]. S-TaLiRo found a falsifying trace with a robustness score of -0.429 and with initial conditions [17.4705 17.2197 17.0643 17.8663 17.4316 17.5354 17.9900 17.6599 17.8402 17.2036].

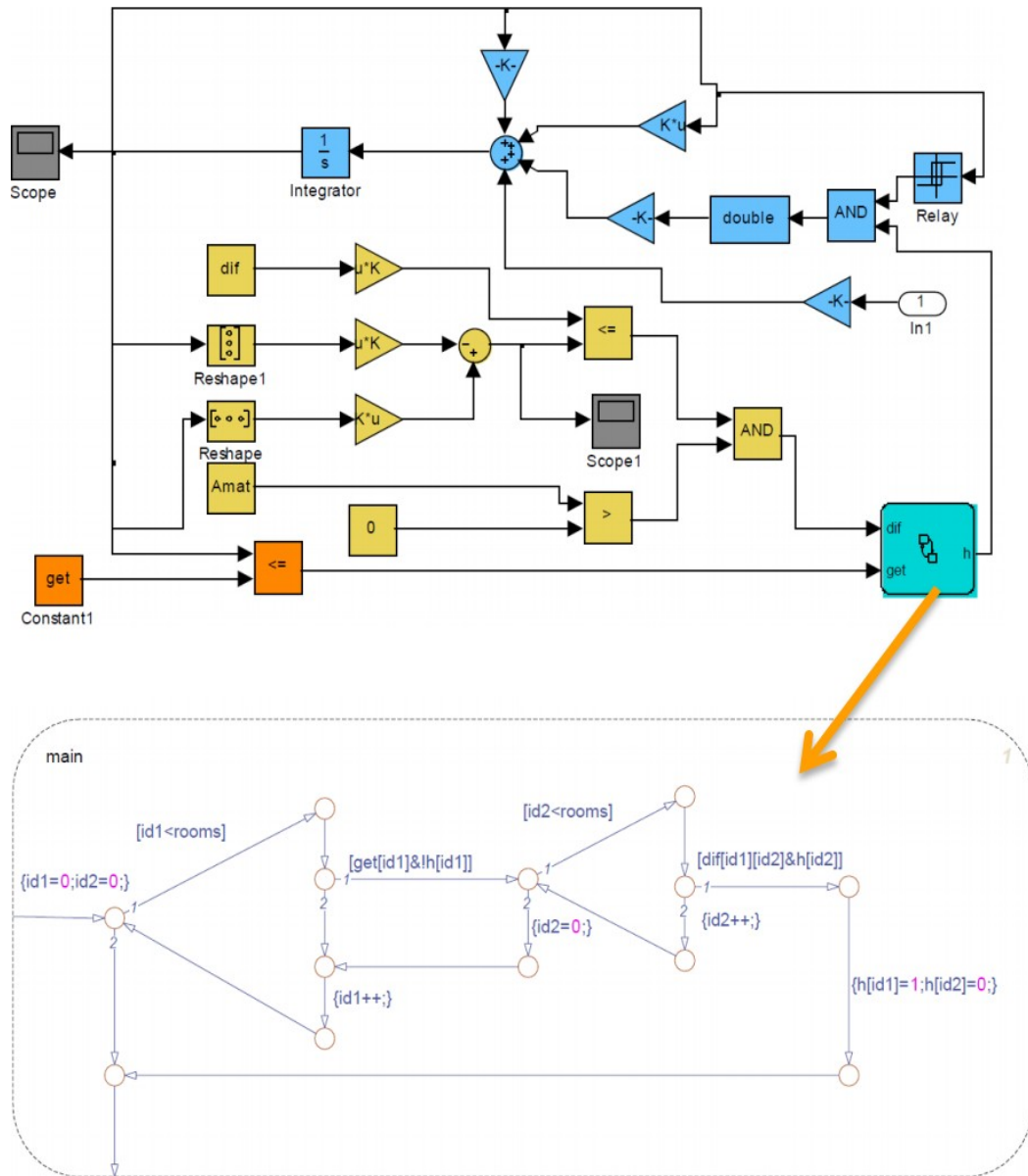


Figure 2.1: Room heating benchmark HEAT30 ^[3]

S-TaLiRo was conceived in response to the dearth of publicly available tools that can tackle the problem of temporal logic falsification for hybrid systems. While MathWorks provides comprehensive design verification tools, S-TaLiRo, on the other hand, aims to focus solely on solving the problem it was created to solve.

It is worth noting that S-TaLiRo requires absolutely no knowledge of the inner workings of the system under simulation. The system is treated as a black box with only the output of the simulation, resulting from the input chosen by the sampler, visible to S-TaLiRo as a basis for the robustness score to be calculated in order to determine whether a falsification of the system's temporal logic properties has been found.

2.2 Active Learning Framework

In [6], Settles defines Active Learning as a subfield of machine learning, a semi-supervised instance as opposed to a supervised one. Supervised machine learning systems are trained on large data sets that are labelled by human beings. Often, this labelling is very difficult and expensive time- and effort-wise. An example of supervised machine learning is speech recognition, in which the system is trained on extensive data sets prepared by trained linguists. Word annotation takes a large amount of time, let alone when a rare language or dialect is involved. On the other hand, semi-supervised machine learning involves sporadic reliance on an oracle (in this case, a human expert) by selecting, out of a large unlabeled pool of training data, a subset that it requires to be labelled in order to proceed with building the model it is concerned with. It requests the labelling of such in the form of queries to the user.

In [7], Bryan discusses an Active Learning framework useful for learning a level-set of a target function over an entire parameter space based on a limited set of observations. Active Learning frameworks are particularly useful and efficient in cases where the target function's parameter space is of high dimensionality as it becomes computationally infeasible to sweep over the entire parameter space in order to learn the entire function. Even more efficient are Active Learning algorithms that learn only specific properties of the target function.

Since the parameter space is assumed to be large or infinite, it is not feasible to estimate the value of the target function for all input parameter vectors. Thus, from an initial set of points in the parameter space, the algorithm selects a set of candidates uniformly at random and scores them according to a kriging model. Robustness is computed at the point with the highest score and the result is added to the data set. The selected point best refines the model (approximation) of the target function.

The next point is always chosen based on the robustness scores of the previously chosen points. A model of the function is constructed gradually. A greedy learning algorithm is used to choose the best set of points to sample in order to construct that model.

The function is approximated using Gaussian process regression. A weighted combination of the robustness scores for the points that have already been chosen before directs where to look next. A distance-based kernel function weighs the robustness scores of nearby points significantly more than distant points.

A simple weighted Euclidean distance function for the computation of robustness is deemed suitable as long as each dimension is linearly scaled such that the semi-variance along the axis is unity. This gives the parameters equal consideration considering their different ranges of values and derivatives.

2.3 Rob-Guided-Verif

Rob-Guided-Verif is a dedicated suite written by Houssam Abbas available in the GitHub repository in [8]. The suite makes use of S-TaLiRo for simulation and returns robustness scores for input vectors specified for the system under analysis. Other features of S-TaLiRo such as the temporal logic analysis engine are unused; instead, S-TaLiRo is used to compute the ground truth –

the set of robustness scores of each and every point in the search space of the system. Then, the Active Learning framework is summoned with no knowledge of the ground truth – the algorithms involved select a set of candidate points, summon S-TaLiRo to compute the robustness score of the candidate point with the highest candidate score, then chooses the next point for S-TaLiRo to compute its robustness score. Based on all the scores of the points that are gathered, the algorithm proceeds in picking the next point, proceeding iteratively up until a stopping criterion is met (see Figure 2). Examples of such are:

- The confidence level that the predicted model matches the actual system model reaches the required threshold
- The number of iterations reaches a determined timeout maximum

```
--> bmk_modulator, large search space
nvars = 5
Sample value: 2.652400e-01
Getting ground truth for bmk_modulator on grid with 3125 points
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
You are setting red_descent_in_ellipsoid_algo to UR - please make sure you set red_hard_limit_on_ellipsoid_nbse to 1.
Elapsed time: 26.8544s
Saved groundTruths/bmk_modulator-3125large.mat
Seed training
Active learning
GRADIENT STOPPING CRITERIA
-----
Total Iterations made: 17
Final accuracy = 8.755200e-01
Saved results/bmk_modulator-3125large.mat
```

Figure 2.2: Sample run of S-TaLiRo

2.4 NVLatch Benchmark

2.4.1 Introduction

In the modern era, the focus on driving down power consumption, particularly with the emergence of Internet of Things (IoT) devices, has led to a great interest in processors that can

preserve their memory state and shut down without having to maintain power to their memory elements. Therefore, it is essential that non-volatile, rather than volatile, memory elements accessible to the CPU are developed and utilized^[11]. Here, we investigate a non-volatile latch, the circuit diagram of which is shown below in Figure 6.

2.4.2 Circuit Diagram

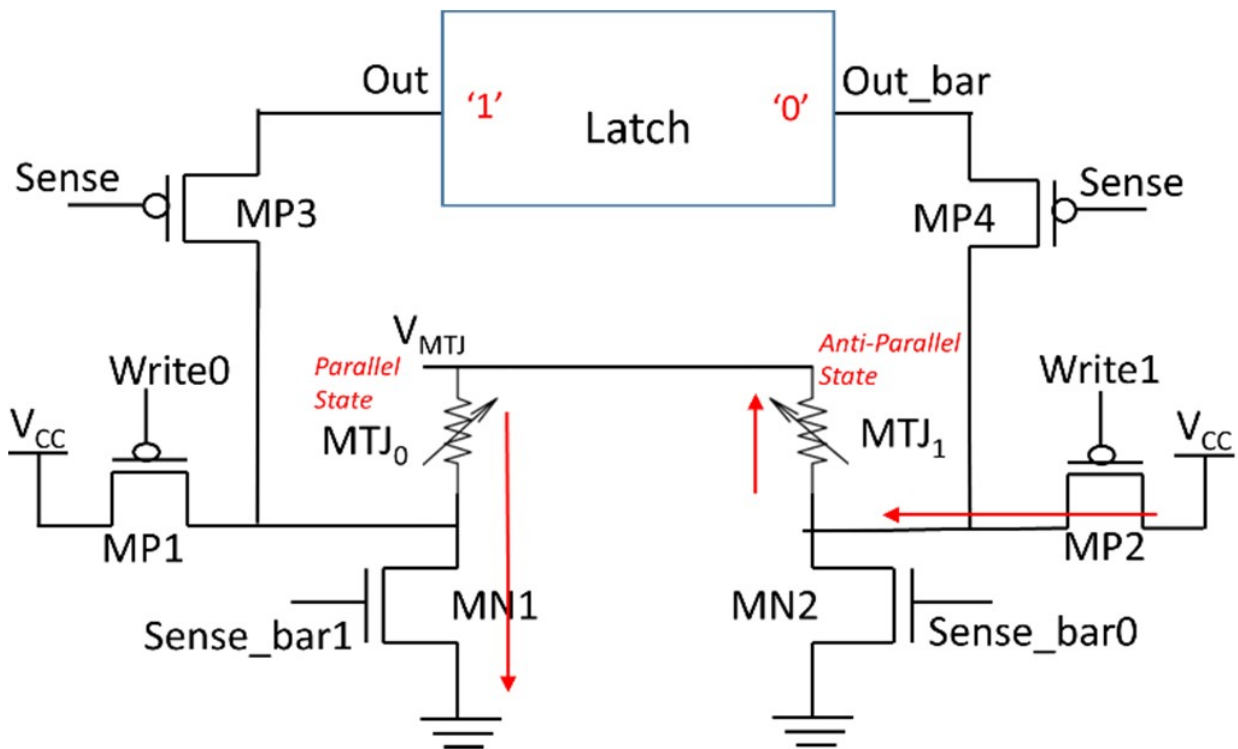


Figure 2.3: Circuit diagram of a non-volatile latch

2.4.3 Math Equations

The components of most interest in the circuit are the STT-RAM memory cells. The component is modeled in SPICE with the following equations:

1) Spin Transfer Torque:

$$STT = \eta \frac{\mu_B I}{eV}$$

2) Damping torque:

$$D = \alpha \gamma (\mu_0 M_s H \sin \theta + 2K \sin \theta \cos \theta)$$

3) Saturation Magnetization:

$$M_s = M_{s0} \left[4 \left(1 - \frac{1}{4r/ch - 1} \right) \cdot \exp \left(- \frac{2S_b}{3R} \frac{1}{4r/ch - 1} \right) - 3 \right]$$

4) STT-MTJ resistance in parallel state:

$$R_p = \frac{t_{ox}}{\sqrt{\phi_i} A} \exp(1.205 t_{ox} \sqrt{\phi_i})$$

where $A = \pi r^2$, area of STT-MTJ.

5) STT-MTJ resistance in transition:

$$R(\theta) = 2R_p \left(\frac{1 + TMR}{2 + TMR + TMR \cdot \cos \theta} \right) \quad [12]$$

2.4.4 Correct Behavior

The STT-RAM memory cell is required to transition from the parallel state to the anti-parallel state in 7ns and from the anti-parallel state to the parallel state in 4ns.

After writing a value ('0', '1') to the non-volatile memory (in backup mode), then, restoring the value to the latch (in recovery mode), the output should match the written value.

2.4.5 Search Space Dimensions

We are concerned with only the backup mechanism and most particularly the parallel to anti-parallel transition. This mechanism is affected by the STT-RAM memory cells as well as the write transistors. The design contains 2 STT-RAM memory cells and 2 write transistors. Each STT-

RAM memory cell contains 1 MTJ and 1 transistor. In total, the backup mechanism thus contains 4 transistors and 2 MTJs ^[10].

For each transistor, the length can be varied to affect the circuit behavior. For each MTJ, the radius can be varied to affect the circuit behavior. The design is symmetrical across the y-axis, and the anti-parallel to parallel transition is not particularly worrisome. This means that the set of components {MP1, MTJ0, MN2} or {MP2, MTJ1, MN1} is enough on its own to be studied. Therefore, the search space has 3 dimensions.

2.4.6 System Inputs

The circuit has two modes of operation: backup and recovery. Backup mode is used to write the contents of the latch to the non-volatile memory, while recovery mode is used to restore the contents of the non-volatile memory to the latch. ^[10]

In backup mode, the active-low input “Write0” is used to write a ‘0’ to the non-volatile memory, while the active-low input “Write1” is used to write a ‘1’ to the non-volatile memory. “Write0” and “Write1” cannot be asserted at the same time, so the combinations of values for the input signals are:

- Write0 = 1, Write1 = 1 (no backup)
- Write0 = 0, Write1 = 1 (write a ‘0’)
- Write0 = 1, Write1 = 0 (write a ‘1’)

In recovery mode, the active-low input “Sense” is used. The possible values for the input signal are:

- Sense = 1 (no restore)
- Sense = 0 (restore)

The “Sense” input cannot be asserted at the same time as the “Write0” or “Write1” input. Therefore, we can condense all three input signals into a single input signal with the following possible values:

- 00: Write0 = 1, Write1 = 1, Sense = 1 (no backup, no restore)
- 01: Write0 = 0, Write1 = 1, Sense = 1 (backup: write a ‘0’)
- 10: Write0 = 1, Write1 = 0, Sense = 1 (backup: write a ‘1’)
- 11: Write0 = 1, Write1 = 1, Sense = 0 (restore)

2.4.7 Benchmark Significance

Due to variability in the transistor threshold voltages and the MTJ memory cells radii, the backup mechanism can be affected in terms of latency, potentially failing STT-RAM memory cell timing requirements^[10]. It is vital that the pulse width be large enough to accommodate proper transitions for the slower cells. This would ensure correct operation given a particular level of variability; however, it also has the negative effect of forcing a pulse width that might be too long for cells that transition quickly. It is thus in our interest to find the ranges of parameters for which the STT-RAM cells can transition with a sufficiently low delay so that the pulse width is lowered for the non-volatile memory of lower write latency, allowing the processor to power off earlier.

2.5 Yield Estimation via Multi-Cones

Rare fail event estimation relying on Monte Carlo often does not converge for a reasonable amount of samples. In [16], the Multi-Cones approach is laid out. The spherical input parameter space is divided into non-overlapping, approximately equi-probable cones by generating random equi-probable vectors from the center. Then, the failure region boundary is located using binary

search. The approach assumes that there are potentially multiple, necessarily monotonic failure regions.

Due to potential bias from random sampling of a limited set of direction vectors, weight refinement may then be performed to estimate the weight contribution of each vector. This involves issuing additional random vectors, preferably a sufficiently large multiple of the number of direction vectors. Then, the additional vectors are clustered around the direction vectors. The ratio of the cluster size of each direction vector to the total number of issued weight refinement vectors constitutes the weight contribution of the direction vector.

Devices are assumed to yield according to a Gaussian distribution, so the probability of a device falling in a discovered failure region is computed according to the Gaussian probability density function corresponding to the dimensionality of the parameter space. The sum of products of each probability of failure and the weight contribution of the corresponding direction vector estimates the yield.

The aforementioned steps of the algorithm are repeated until the yield estimate converges, using a suitable convergence scheme.

Yield estimates via this approach converge much more quickly than Gaussian sampling Monte Carlo.

2.6 ngspice

ngspice is a mixed-level / mixed-signal circuit simulation that is originally based on BERKELEY SPICE.^[17] It is free, open-source software available for download and compilation on Linux, macOS, and Windows. It is written in C and is maintained by Holger Vogt. It is battle-tested and stable, being used in many projects.

According to the ngspice manual in [18], ngspice implements intra-simulation parallelism via OpenMP, mostly for complex transistor models. CUDA is available as well, in the form of cuspice. The speedups observed are modest.

Ngspice provides a C API for running SPICE simulations from a C / C++ program. However, the C API does not provide explicit support for inter-simulation parallelism – instead, the programmer would have to rely on loading and closing copies of the ngspice shared library for each simulation that is run.

2.6.1 ADMS

According to [19], ADMS is a code generator that converts models written in Verilog-AMS into C code that is compatible with the target SPICE simulator. The generated code becomes part of the simulator and the entire codebase is compiled into an executable that is now compatible with the Verilog-AMS model.

A guideline for integration of Verilog-AMS models in ngspice using ADMS is provided. The process involves modification of several entries in ngspice source code, the creation of specific directories for ADMS to generate C code into, and the re-compilation of ngspice now with the generated C code added to the existing codebase.

Testing the implementation after the compilation process is complete is necessary to ensure correct operation of the integrated model.

2.7 mp-units

According to [20], mp-units is a compile-time enabled Modern C++ library that provides compile-time dimensional analysis and unit / quantity manipulation. The library handles quantity

conversion ratios at compile-time, ensuring that such operations are error-free, highly precise / exact, and zero-cost at run-time.

The library provides electrical, time, and length SI units, among others, along with SI prefixes (nano, micro, milli, etc...). Resistance (Ohm), capacitance (Farad), inductance (Henry), voltage (Volt), and current (Ampere) are the electrical units most frequently used for electric circuit simulations. Length SI units are necessary to control transistor model parameters such as oxide thickness. Time SI units are necessary to control simulation time step, start, and stop.

The library utilizes some of the latest C++20 and C++23 features such as Concepts and classes as NTTPs (Non-Type Template Parameters) and targets the latest versions of available compilers such as GCC^[21] and Clang^[22] where support for these features can often be very recent and experimental.

The library's author, Mateusz Pusz, is working with various C++ groups on potentially having the library standardized for C++23.

CHAPTER 3

PROBLEM DESCRIPTION

The active learning framework is implemented in MATLAB. The robustness computation function that it relies on, which is part of S-TaLiRo, is implemented in C with MATLAB hooks and compiled with mex.^[23] The STT-MTJ non-volatile memory model is implemented in Verilog-AMS. The STT-MTJ SPICE circuit is simulated in ngspice.

Tools need to be developed that can:

- Easily integrate Verilog-AMS models into ngspice
- Parametrize SPICE circuits from code
- Run several SPICE simulations in parallel, for the Multi-Cone approach and to compute the ground truth in order to verify the accuracy of the Active Learning algorithm's predictions
- Run SPICE simulations from within the Active Learning framework
- Be efficient with resource usage (such as memory)
- Execute in minimal time in order for the approach to be useful over more traditional approaches

MATLAB, being an interpreted language with dynamic typing^[22], suffers from several common overheads for operations such as function calls. Furthermore, licensing can be expensive and this potentially forms a barrier to entry for other researchers or institutions wanting to use these tools, particularly in commercial, non-academic environments. In addition, enhancing performance and extending the framework beyond what is currently implemented is best performed in a compiled language with static typing and compiler optimizations for architecture-specific

performance improvements (such as AVX support in modern processors). While MATLAB does call performance-oriented libraries such as MKL where it is suitable to do so ^[25], not all functionality in MATLAB can be guaranteed to rely on such libraries, particularly that which does not extensively rely on built-in functions or toolboxes.

C++ is the perfect candidate for the implementation of all the missing tools as well as the re-implementation of the Active Learning framework.

CHAPTER 4

IMPLEMENTATION

4.1 mp-units

mp-units provides SI units as strong types to spice-circuit-generator. This reduces programmer error in contrast to using raw floating-point types – any quantity which unit does not match a generator setter function’s argument type cannot be passed as an argument and the program will fail to even compile.

4.1.1 ASCII-Only Output Support

mp-unit’s default unit symbols are in Unicode format. ^[26] ngspice supports ASCII characters only, so any Unicode symbols present in a SPICE circuit that is loaded into ngspice would not be read correctly and would very likely cause ngspice to crash. It is therefore imperative that spice-circuit-generator does not generate circuits that contain Unicode symbols; however, spice-circuit-generator relies on mp-units for string formatting related to quantity, prefix symbol, and unit symbol.

We implemented ASCII-only output support ^{[27][28]} in mp-units. This required the implementation of a **basic_symbol_text** class that integrates two **basic_fixed_string** objects, one for Unicode and another for ASCII. Several constructs throughout the library required a change from using **basic_fixed_string** for their only Unicode symbol to using **basic_symbol_text**. The constructors provided to **basic_symbol_text** ensured that contributors to the library would not have a harder time declaring singular unit symbols in duplicate fashion as the majority of units had a single symbol that was available in both the ASCII and the Unicode character set.

The SI resistance unit **ohm** would incur the following change:

```
struct ohm : named_unit<ohm, {"Ω", "ohm"}, prefix> {};
```

It was crucial to avoid having to refactor all other units to declare two symbols, even if duplicated, in the following manner:

```
struct farad : named_unit<farad, {"F", "F"}, prefix> {};
```

The following constructor for **basic_symbol_text** avoided this duplication issue:

```
constexpr basic_symbol_text(StandardCharT s) noexcept: standard_(s), ascii_(s) { validate_ascii_char(s); }
```

The definition of **farad** would therefore stay the same as before:

```
struct farad : named_unit<farad, "F", prefix> {};
```

The string formatter specification parser had to be modified to accept the **A** modifier that forces ASCII-only output.

```
fmt::print("{:%Q%q}", 123q_kohm); // 123kQ  
fmt::print("{:%Q%Aq}", 123q_kohm); // 123kohm
```

Unit tests were added to verify that the implementation is correct.

4.1.2 *Alias Units*

Many prefixed SI units, such as **nanometer**, were missing. Nanometer is crucial for SPICE circuits as modern transistors have many parameters that operate at that scale.

In the process of adding missing prefixed SI units, **megagram** conflicted with **tonne** as it was considered a redeclaration. We implemented **alias units** – units that are operated on as if they were an equivalent unit but with a substituted symbol. ^[29] Alias units allow prefixing when suitable,

even when their equivalent units do not. **Megagram**, for instance, does not allow any prefix to be added (since it is already a prefixed unit), but its alias unit, **tonne**, does, (e.g. **kilotonne**).

Implementing alias units in such a manner avoids any further modifications to the library that might disturb core functionality and allows the implementation to be relatively unaffected by later modifications to the core.

Unit tests were similarly added to ensure that the implementation is correct.

4.1.3 *Random Number Distributions*

monte-carlo requires the generation of quantities following a Gaussian random distribution. We implemented wrappers for the STL (Standard Template Library) random distributions. ^{[30][31]} This allows us to generate random quantities straight from **mp-units** without having to use the raw STL random distribution classes then wrap the raw floating-point value in a quantity object.

Similar to how an STL random distribution is utilized:

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937_64 gen { rd() };

    std::uniform_real_distribution<double> dist { 10.0, 2.0 };
    double x = dist(gen);
    std::cout << x;
}
```

mp-units random distributions, which are available in the **units** namespace, can be utilized:

```
#include <iostream>
#include <units/physical/si/voltage.h>
#include <units/random.h>

int main()
{
    using namespace units::physical::si::literals;

    std::random_device rd;
    std::mt19937_64 gen { rd() };

    units::uniform_real_distribution dist { 10.0q_mV, 2.0q_mV };
    auto x = dist(gen);
    std::cout << x;
}
```

4.2 ngspice C++ wrapper

ngspice-cpp is a C++ wrapper for ngspice. ^[32] It provides:

- straightforward simulation object creation and usage
- easy iteration over results after completion
- feedback simulations
- explicit support for inter-simulation parallelism

4.2.1 Simulation Object

To load a SPICE circuit, run a simulation, and output a vector after completion:

```
#include <ngspice-cpp/simulation/sync.cpp>

int main()
{
    ngspice::Simulation simulation { "circuit.sp" };
    simulation.run();

    auto vector = simulation.vector("V(2)");
    for (const double& vec_value : vector)
    {
        std::cout << vec_value << " ";
    }
}
```

4.2.2 Feedback Simulation

Often, SPICE simulations reach a certain point at which the needed property has been realized. Beyond that point, it is wasteful to keep the simulation running. We implemented **feedback simulations** in **ngspice-cpp**. The programmer would derive the **FeedbackSimulation** base class and implement initialization, step, and finish callback functions that **ngspice-cpp** would use to pass information to the simulation object so that it could figure out when to stop the simulation. This saves a sizeable portion of execution time in many instances.

```

class FeedbackSimulation : public ngspice::FeedbackSimulation
{
    constexpr static double target_magnetic_angle = 3.141592;

    using Time = si::time<second, double>;
    using PTime = si::time<nanosecond, int>;

    ngspice::Vector time_vec;
    ngspice::Vector net3_vec;

    const PTime pulse_start_time;
    size_t pulse_start_idx;
    Time delay_;

    void (FeedbackSimulation::*f)(std::size_t);

public:
    FeedbackSimulation(std::unique_ptr<ngspice::Circuit>& circuit, PTime pulse_start_time = 1q_ns);

    void on_init() override;
    void on_step(size_t idx) override;
    void on_finish(size_t num_iterations) override;

    const Time& delay() const;

private:
    void wait_until_pulse_start(size_t idx);
    void wait_until_anti_parallel(size_t idx);
};

```

4.2.3 Explicit Parallelism

Inter-simulation parallelism is handled via OpenMP, ^[33] TBB, ^[34] or any shared memory parallel programming library that can either provide directives to the compiler to parallelize a for loop or that implements parallel for loops.

```

#pragma omp parallel for
for (size_t i = 0; i < num_samples; ++i)
{
    ...

    FeedbackSimulation simulation { circuit };
    simulation.run();

    ...
}

```

ngspice-cpp transparently handles the process of loading and closing **ngspice** shared library copies when multiple simulations are being instantiated, run in parallel, and destroyed.

Various usage examples can be found in the repository at [35].

4.3 SPICE Circuit Generator

ngspice can load SPICE circuit text files and simulate the loaded SPICE circuit. However, fixed circuits are of limited benefit – we need to be able to parametrize circuits on-the-fly often using complex metrics that are not directly supported in and provided by SPICE simulators.

Such metrics have to be able to parametrize a SPICE circuit straight from code.

Furthermore, it is obligatory to have a general approach to parametrization so that SPICE circuit generation is not re-implemented from scratch every time a different type of SPICE circuit, with a different set of parameters, is used.

4.3.1 Grammar

We implement a Parsing Expression Grammar (PEG) ^[36] as an extension on top of basic SPICE syntax.

An example of a basic RC SPICE circuit is:

```
RC_CIRCUIT
.TRAN 5ns 2ms
VSRC 1 0 PULSE( 0 10 0 0 0 1ms 0 )
RLOAD 1 2 10k
CCHARGE 2 0 10nF
.END
```

With the PEG extension, a template RC SPICE circuit looks like:

```
RC_CIRCUIT
.TRAN <time_step:time:5n> <time_stop:time:2m>
VSRC 1 0 PULSE( 0 10 0 0 0 <pulse_width:time:1m> 0 )
RLOAD 1 2 <rload:resistance:10k>
CCHARGE 2 0 <ccharge:capacitance:10n>
.END
```

The grammar allows for default values to be provided in the template so that they do not necessarily have to be specified in code. The specific grammar for a template SPICE circuit is:


```

template      = (line newline)* line newline*

line          = ignored_line / include_line / lib_line / source_line / epsilon
include_line  = include_start space+ ((single_quote filepath single_quote) / (double_quote filepath
double_quote)) space*
lib_line      = lib_start space+ ((single_quote filepath single_quote) / (double_quote filepath
double_quote)) (space+ text)? space*

ignored_line  = comment_line / print_line / space+
comment_line  = comment_start source_fragment*
print_line    = print_start source_fragment
source_line   = source_fragment+

source_fragment = space* (parameter / text) source_fragment_tail
source_fragment_tail = (space* ((parameter / text) source_fragment_tail?)) / epsilon

parameter     = parameter_open space* text space* colon space* parameter_type space*
parameter_default? parameter_close
parameter_default = colon space* ((number / dot) space*)? (unit_prefix space*)?

parameter_type = time / voltage / resistance / capacitance / length / unitless
time           = "time"
voltage        = "voltage"
resistance     = "resistance"
capacitance    = "capacitance"
length         = "length"
unitless       = "unitless"

filepath      = ~"[A-Z0-9_\- ./]+"i
single_quote  = "\""
double_quote  = "\""
include_start = ".include" / ".inc"
lib_start     = ".lib"

text          = ~"[A-z0-9_.\-=/\\"*(,)+\{\}\]"i
int           = ~"[0-9]+"i
decimal       = (int? "." int) / (int "." int?)
number        = '-'? (decimal / int) exponent?
exponent      = ('e' '-'? int)
unit_prefix   = ~"[acdfhkmnpuyzEGMPTYZ]"i / "da"
space         = " " / "\t"
newline       = "\n"
colon         = ":"
parameter_open = "<"
parameter_close = ">"
dot           = "."
epsilon       = ""
comment_start = "*"
print_start   = ".print"

```

4.3.2 Code Generation

We used **parsimonious**, a Python PEG parser, ^[37] to parse template SPICE circuits according to the aforementioned grammar. Then, from the generated Abstract Syntax Tree (AST),

we populate a Jinja2 ^[38] template to produce C++ code that can be included in projects that require parametrization of these SPICE circuits.

The generated code provides a SPICE circuit generator that can set the parameters and generate different circuits based on the template circuit. The circuit generator relies on **mp-units** to ensure that the quantities passed do not use the wrong units. A generator can be instantiated and used in the following manner:

```
#include <rc_circuit.hpp>

int main()
{
    ngspice::rc_circuit::Generator generator;

    generator.time_step(5q_ns);
    generator.time_stop(2q_ms);
    generator.pulse_width(1q_ms);
    generator.rload(10q_kohm);
    generator.ccharge(10q_nF);

    std::cout << *generator.circuit();
}
```

Integrating the generator with **ngspice-cpp** allows parametrization of the SPICE circuit with complex metrics entirely in code, without having to resort to the SPICE simulator implementing such functionality.

4.4 ngspice STT-MTJ Integration

Following the integration guide in [19], we integrated the STT-MRAM Verilog-AMS models according to the model manual in [12]. Due to how SPICE simulators function, the Verilog-AMS code required a few modifications to transform current input into voltage input and voltage output into current output. Consequently, SPICE circuits have to use current-controlled voltage sources on the input and output terminals in order to use these models.

The integrated models were unit tested extensively, using **ngspice-cpp** and **googletest**,^[39] to ensure that they function according to specification in the model manual^[12]. Test SPICE circuits were also run to graphically ensure that the models respond in the same way as the manual indicates they would.

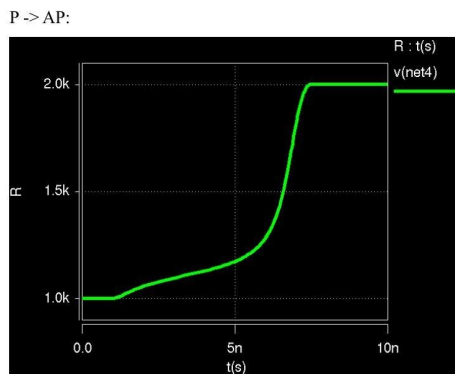


Figure 4.4.1: P -> AP Transition (manual)

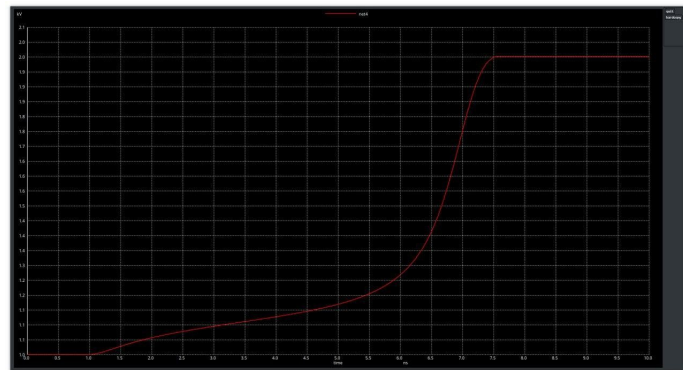


Figure 4.4.2: P->AP Transition (ngspice)

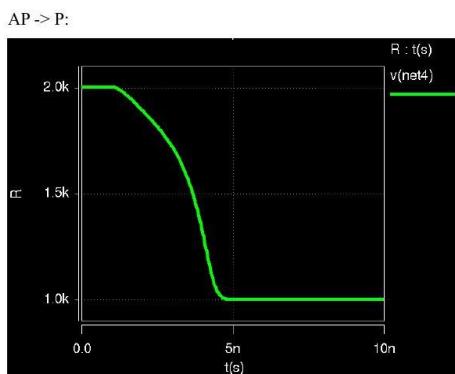


Figure 4.4.3: AP -> P Transition (manual)

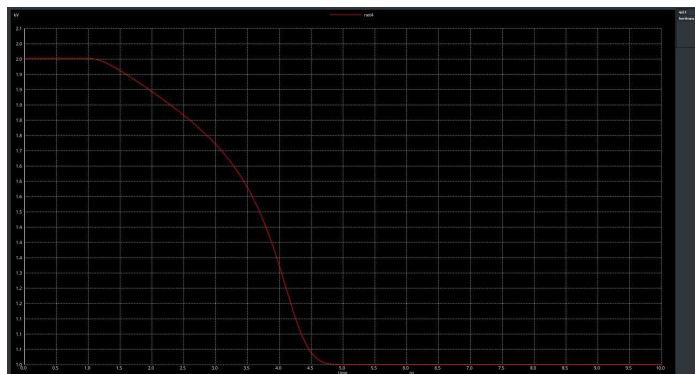


Figure 4.4.4: AP -> P Transition (ngspice)

Build scripts, code patches to fill in missing C code that the **ADMS** compiler did not generate, as well as scripts to facilitate integration of future Verilog-AMS models, are provided in [40]. These additions are continuously rebased on top of **ngspice** source code and the entire build process is performed to ensure that new changes to **ngspice** source code do not break the integration.

4.5 Monte Carlo

We implemented the Monte Carlo approach for the STT-MTJ model using:

- uniform real distributions with **mp-units** to generate random radii and transistor threshold voltages
- STT-MTJ parallel to anti-parallel state transition SPICE circuit generator with **spice-circuit-generator**
- ngspice feedback simulations running in parallel with **ngspice-cpp**, stopping when the STT-MTJ cell has fully transitioned from parallel to anti-parallel state

Yield estimation is performed by counting the number of passing circuits and dividing the count by the number of tested circuits.

4.6 Yield Estimation via Multi-Cones

We implemented the Multi-Cones approach in modern C++, relying heavily on template metaprogramming. This allows the determination of container sizes at compile-time, keeping heap allocations to a minimum. The floating-point type can be chosen as well, improving either precision, performance, or memory usage, or a combination thereof. The random number generator engine can be swapped, improving either quality or performance or both.

The implementation heavily utilizes C++17 parallel STL algorithms,^[41] allowing for shared memory parallelism via TBB.^[34]

A KDTree^[42] was used for weight refinement clustering in order to speed up nearest neighbor searches.

4.7 Robustness Guided Verification (C++)

We reimplemented **rob-guided-verif** as a header-only library in C++20 with extensive unit tests, benchmarks, and examples. ^[43]

4.7.1 *Language Features*

The implementation relies on the following C++ features:

- **Concepts:** ^[44] allow constraining template arguments such that only specific types are permitted, such as floating-point or particular user-defined types
- **Variadic Templates:** ^[45] allow an arbitrary number of entities to be aggregated into one container type (such as input signals) despite being of different types (template class instantiations with different template arguments)
- **Non-Type Template Parameters:** ^[46] allow passing in sizes as template arguments so that matrix and vector dimensions are determined at compile-time, avoiding heap allocations
- **Function Objects:** ^[47] allow passing in functions at compile-time so that they are inlined by the compiler to avoid function calls
- **Execution policies (parallelism):** ^[48] allow the explicit use or disabling of shared memory parallelism using TBB ^[34] for algorithms that can run in parallel, ^[41] including user-defined algorithms explicitly using TBB

4.7.2 *Dependencies*

Several libraries were used for the framework:

- **Eigen:** ^[49] C++ template library for linear algebra – provides matrix and vector types
- **Boost:** ^[50] a collection of portable C++ libraries – provides ODE solvers ^[51] and interpolators

^[52]

- **CGAL:** ^[53] a collection of geometric algorithms in C++ - provides quadratic programming solver ^[54]
- **googletest:** ^[55] C++ test framework – provides unit tests for various components
- **google/benchmark:** ^[56] C++ benchmark library – provides benchmarks for various components

The **robustness computation** function was ported over from C with MATLAB hooks. Parts of the code were replaced with C++ constructs to facilitate interoperability with the rest of the framework. Memory corruption bugs due to uninitialized variables were fixed.

4.7.3 Models

The framework supports several **models**:

- **Stiff / non-stiff ODEs:** in the form of function objects that derive a base class and specify the equations as well as the Jacobian matrix (for stiff ODEs)
- **MATLAB Simulink models:** in the form of .mdl files, provided with an input signal
- **Function objects:** can implement any functionality in C++, provided with an input signal

4.7.4 ODE Solver

All features of the ODE solver are provided by **Boost.Numeric.Odeint**:

- **Integration:**
 - constant step
 - adaptive step
- **Output:**
 - controlled

- dense
- **Steppers:**
 - rosenbrock4
 - runge_kutta_dopri5
 - other steppers provided by **Boost.Numeric.Odeint**

4.7.5 *Input Signal Interpolation*

Several types of input signal **interpolators** are supported:

- **Constant**
- **Piecewise Constant**
- **Linear**
- **Uniform Random**
- **Cardinal Cubic B-Spline** (provided by Interpolation in Boost.Math)
- **PCHIP** (provided by Interpolation in Boost.Math)

4.7.6 *Ground Truth Sweepers*

For **ground truth** computation, several types of sweepers are supported:

- **Grid:** the number of points per dimension can be specified independently
- **Uniform Random Distribution Monte Carlo:** the lower and upper bounds for each dimension can be specified independently
- **Gaussian Distribution Monte Carlo:** the mean and standard deviation for each dimension can be specified independently

4.7.7 *Estimation Algorithms*

For any estimation algorithms currently implemented or to be implemented in the future, distance metrics, certainty metrics, and candidate selection metrics are provided. ^[7]

4.7.7.1 Distance Metrics

The **distance metrics** provided are: ^[7]

- **Kriging**

4.7.7.2 Certainty Metrics

The **certainty metrics** provided are: ^[7]

- **Entropy**
- **Minmax**

4.7.7.3 Candidate Selection Metrics

The **candidate selection metrics** provided are: ^[7]

- **Uniform Random**
- **Variance**
- **Entropy**
- **Straddle**
- **Probability of Incorrect Classification**

4.7.8 Active Learning Algorithm

We implement the Active Learning algorithm specified in [7] in C++ utilizing the aforementioned distance metrics, certainty metrics, and candidate selection metrics.

4.7.9 Benchmarks

4.7.9.1 quadprog

We benchmark our C++ implementation of a distance function that relies on the quadratic program solver provided by **CGAL** against the “quadprog” function provided in MATLAB ^[57].

4.7.9.2 ode15s

We benchmark our C++ implementation of the stiff ODE solver that relies on **Boost.Numeric.Odeint** against the “ode15s” function provided in MATLAB ^[58].

4.7.9.3 compute_robustness

We benchmark our C++ porting of the robustness computation function against the C version with MATLAB hooks built with **mex**. ^[23]

4.7.9.4 nonlinear

We provide an example of a stiff ODE model using this framework, “**nonlinear**”. The model is already implemented in the MATLAB version of the framework as a benchmark. We measure only the tight loop where robustness computation is performed in MATLAB. Meanwhile, we record the entire C++ program execution time.

4.7.9.5 nonlinear_al

We run the “**nonlinear**” example model under the **Active Learning** framework for 20 iterations on a grid of 300x300 (90000) points in both the C++ version and the MATLAB version.

4.7.9.6 Results

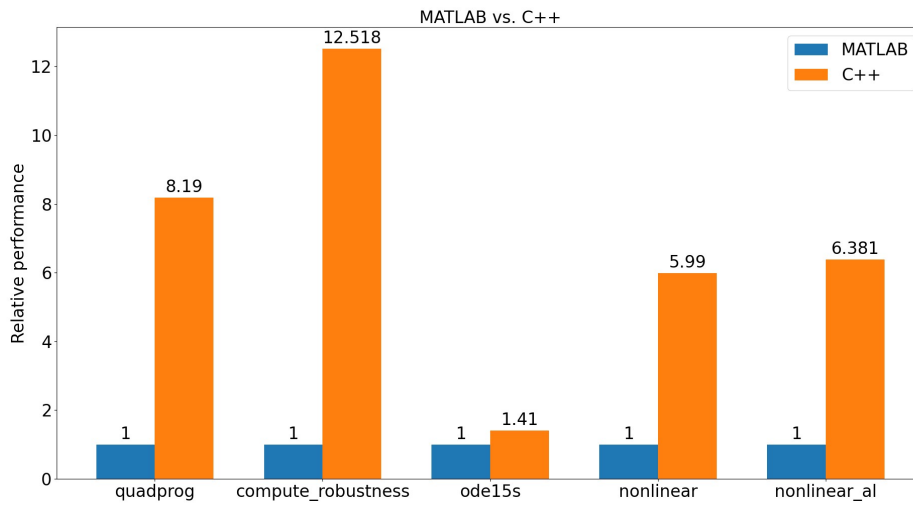


Figure 4.7.9.6.1: MATLAB vs C++ Benchmark Results

4.7.10 Verification

Aside from extensive unit tests, we verify the correctness of the framework implementation by plotting several figures of robustness values that match the MATLAB implementation figures.

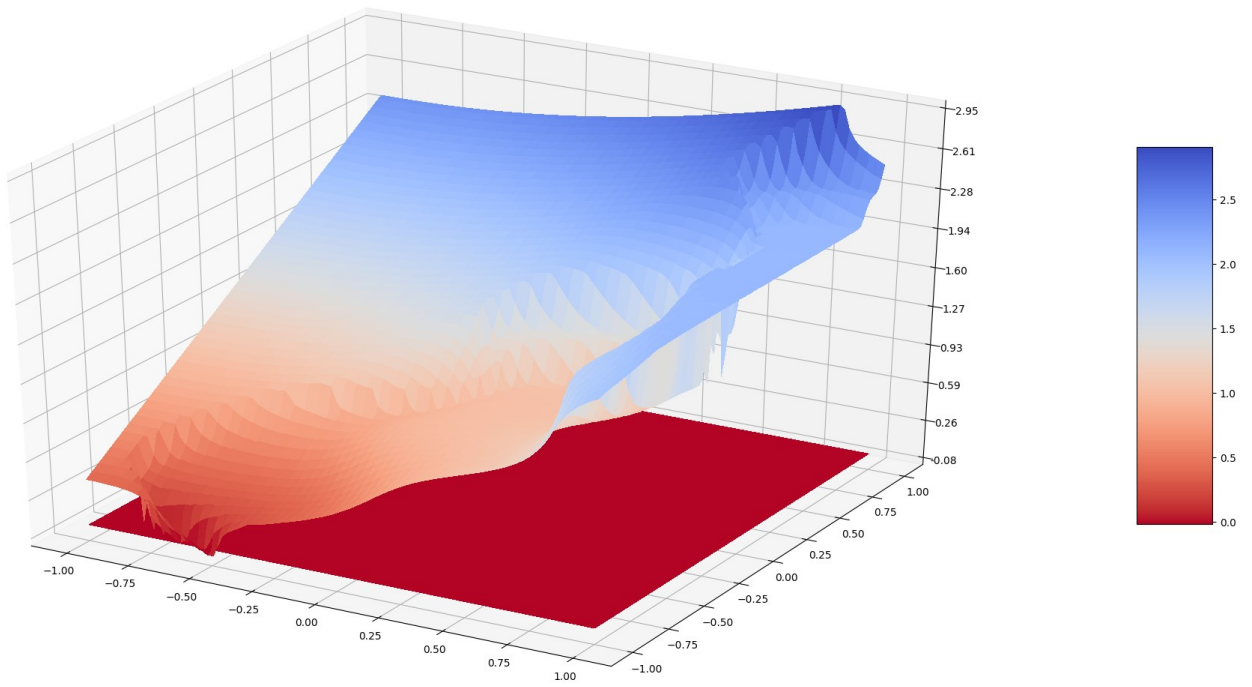


Figure 4.7.10.1: “nonlinear” example robustness (1)

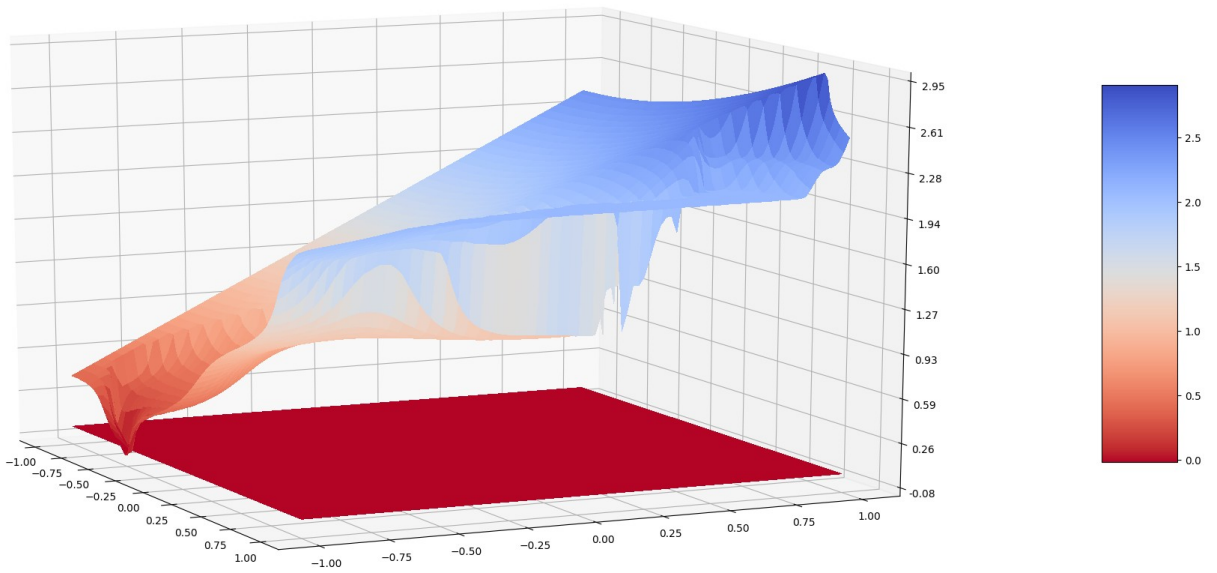


Figure 4.7.10.2: “nonlinear” example robustness (2)

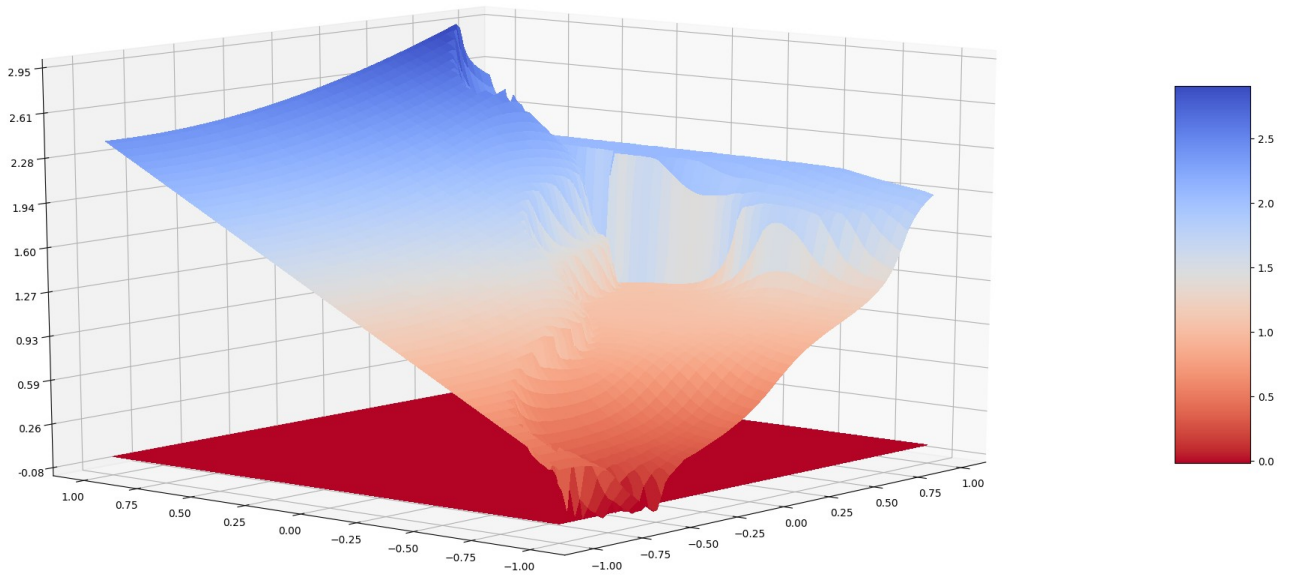


Figure 4.7.10.3: “**nonlinear**” example robustness (3)

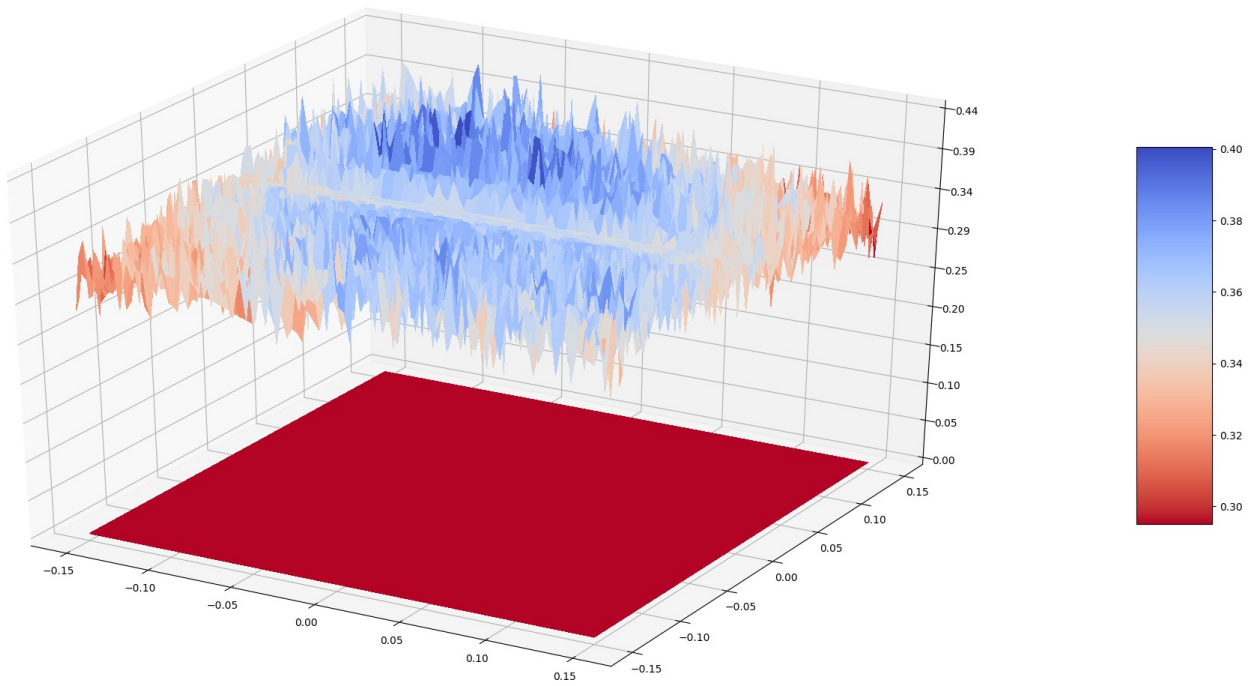


Figure 4.7.10.4: “**bmk_modulator**” example robustness (1)

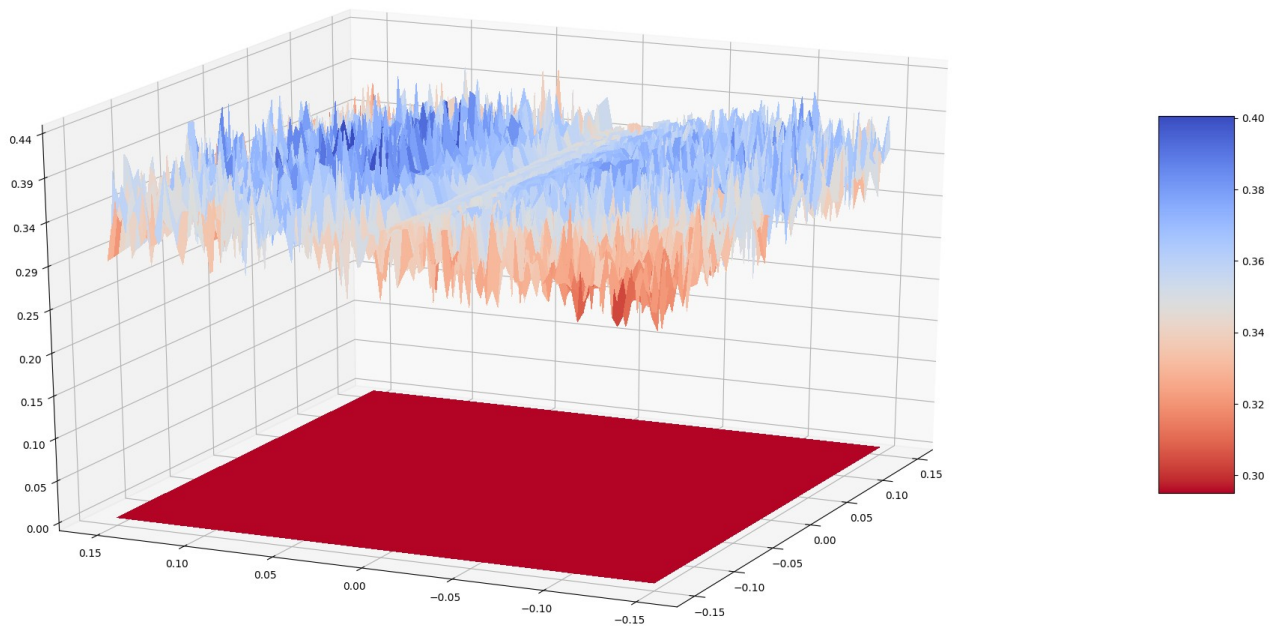


Figure 4.7.10.5: “**bmk_modulator**” example robustness (2)

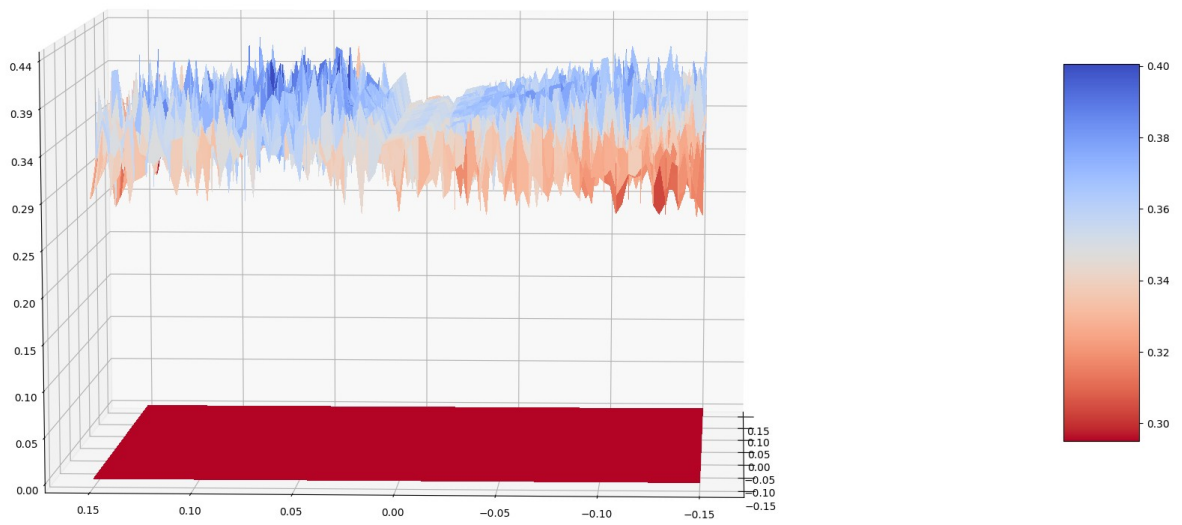


Figure 4.7.10.6: “**bmk_modulator**” example robustness (3)

CHAPTER 5

EXPERIMENTAL RESULTS

Due to differences in parameters used between our SPICE model and the model in [11], we test the STT-MTJ circuit with two nominal delays: 5.4ns and 5.6ns. This helps highlight differences in number of samples to converge between Monte Carlo, Multi-Cones, and Active Learning when failures become rarer. We use the same means and standard deviations for STT-MTJ cell radius and transistor threshold voltage.

Cell radius: mean = **52nm**, standard deviation = **1/3nm**

Transistor threshold voltage: mean = **0.469V**, standard deviation = **0.1250667V**

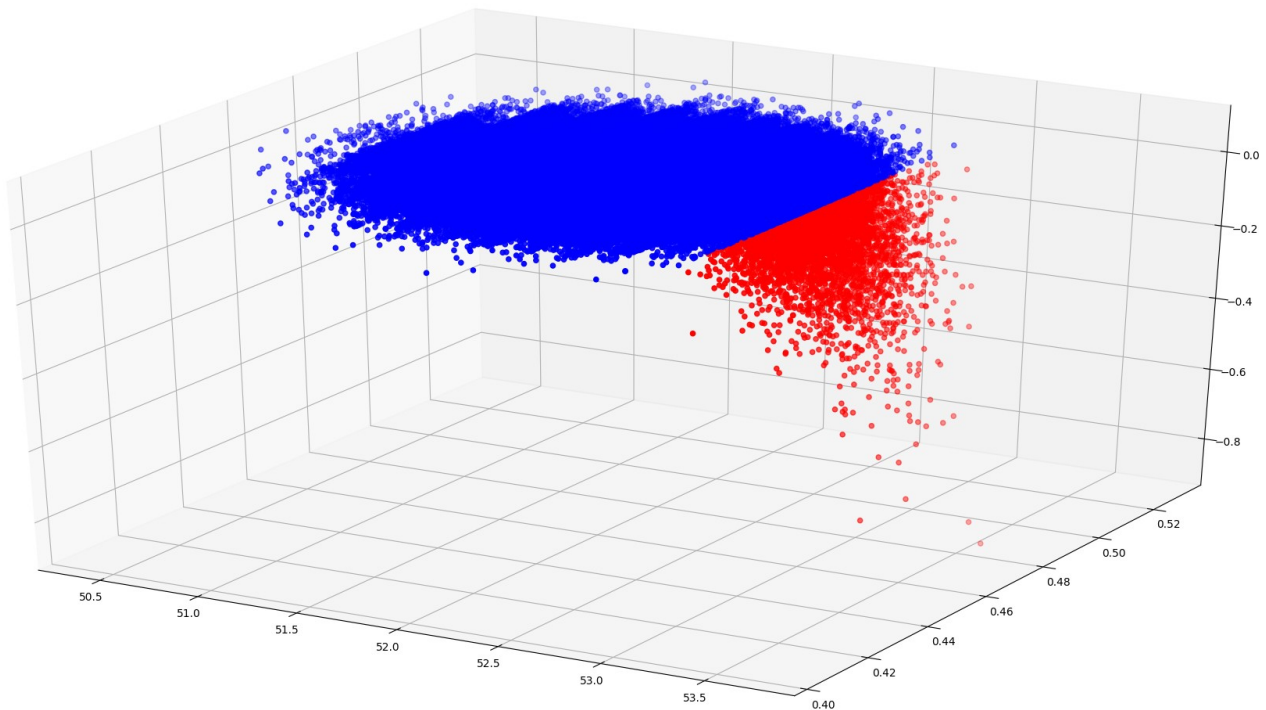


Figure 5.1: STT-MTJ circuit, 5.4ns nominal delay – Gaussian Random Monte Carlo Sweeper
Ground Truth – 1,000,000 points

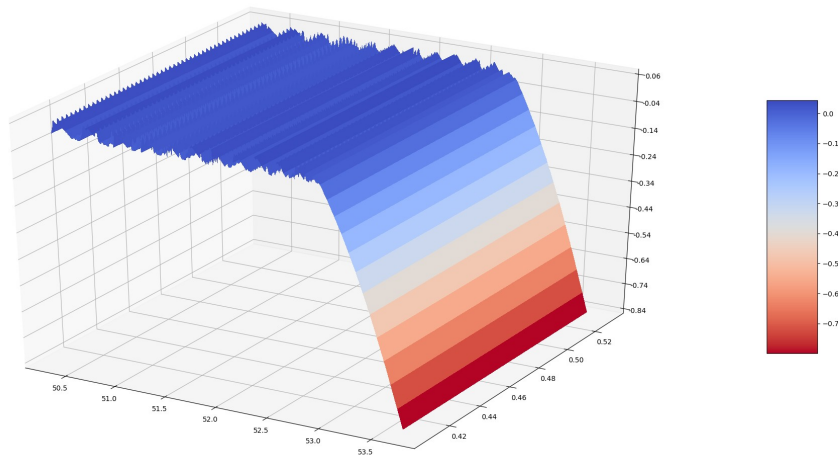


Figure 5.2: STT-MTJ circuit, 5.4ns nominal delay – Grid Sweeper Ground Truth – 300x300 (90,000) points

We run each experiment in Monte Carlo for 500,000 samples. Convergence is estimated by a rolling window of size 500.

We run each experiment in the Multi-Cones approach for 900 experiments each with 100 direction vectors, 1000 weight refinement vectors, and 8 search iterations for binary search. On average, each experiment runs around 440 samples.

For the Active Learning algorithm, we start with a seed training set of 15 samples.

5.1 STT-MTJ 5.4ns Nominal Delay

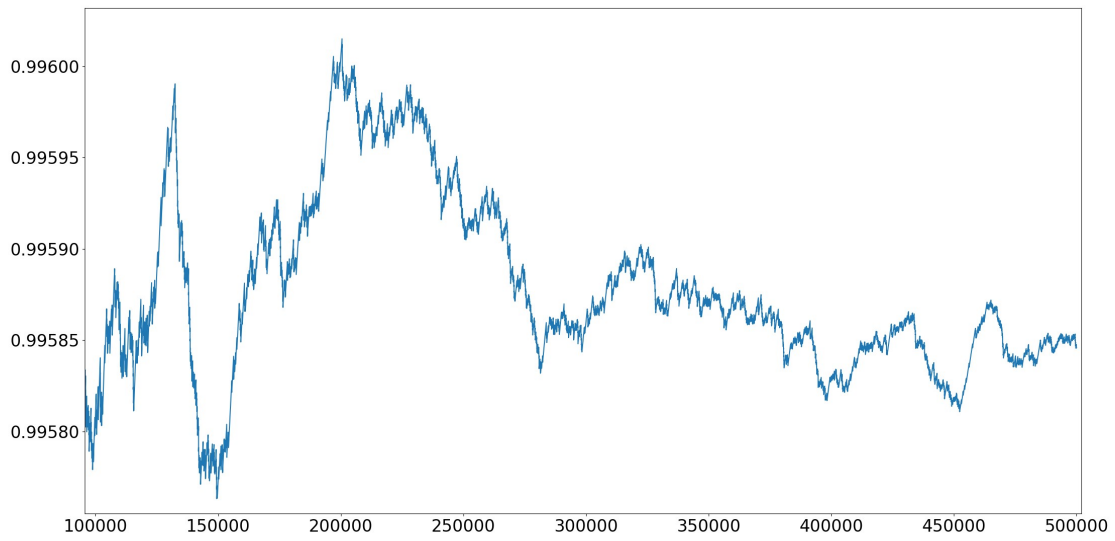


Figure 5.1.1: Gaussian Random Monte Carlo Yield Estimate

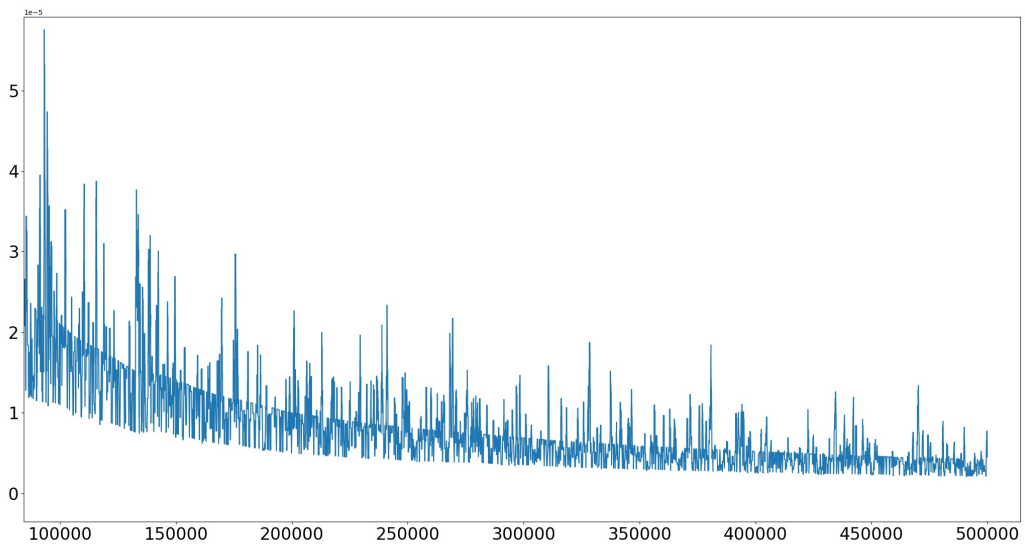


Figure 5.1.2: Gaussian Random Monte Carlo Convergence Estimate

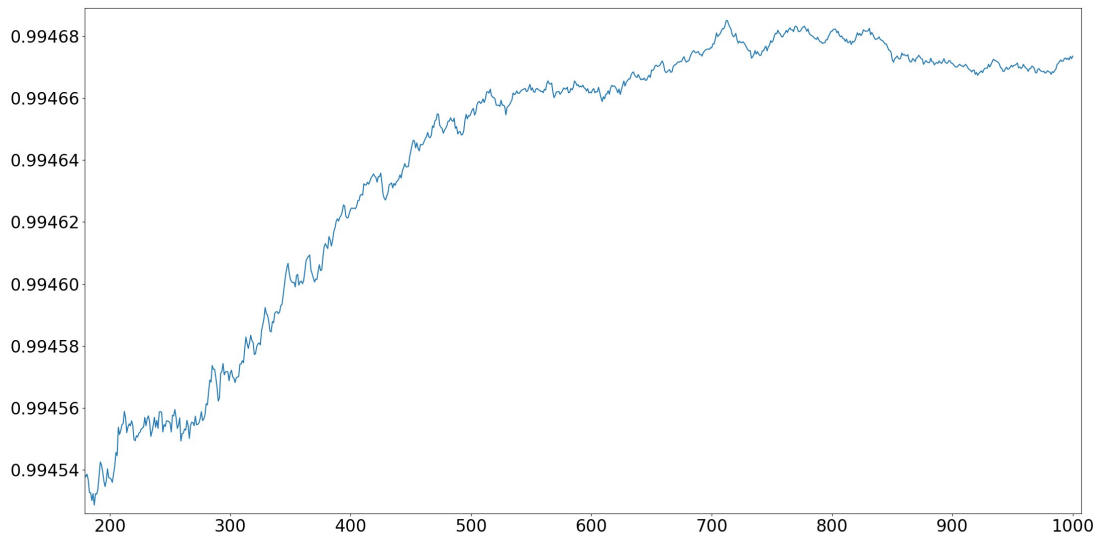


Figure 5.1.3: Multi-Cones Yield Estimate

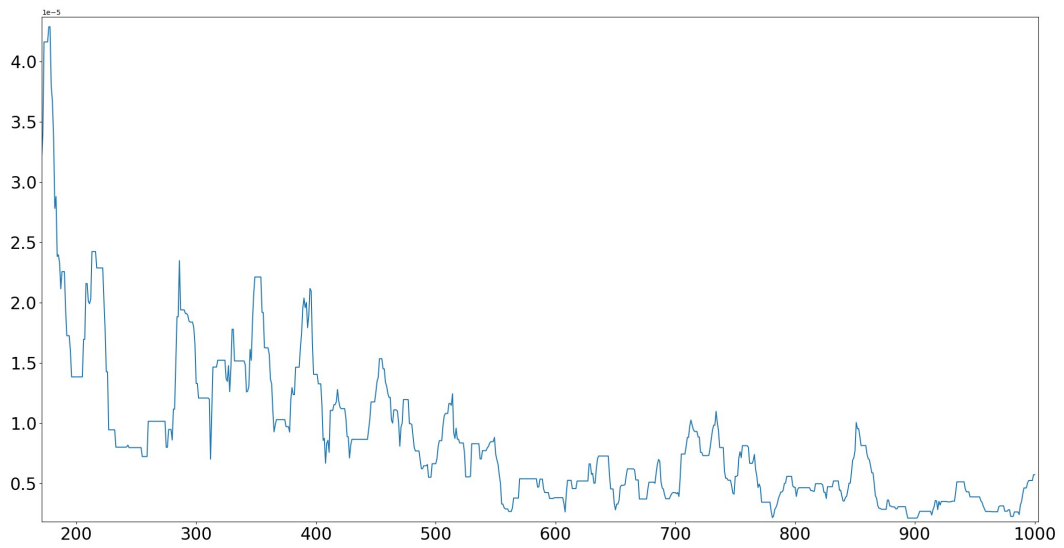


Figure 5.1.4: Multi-Cones Convergence Estimate

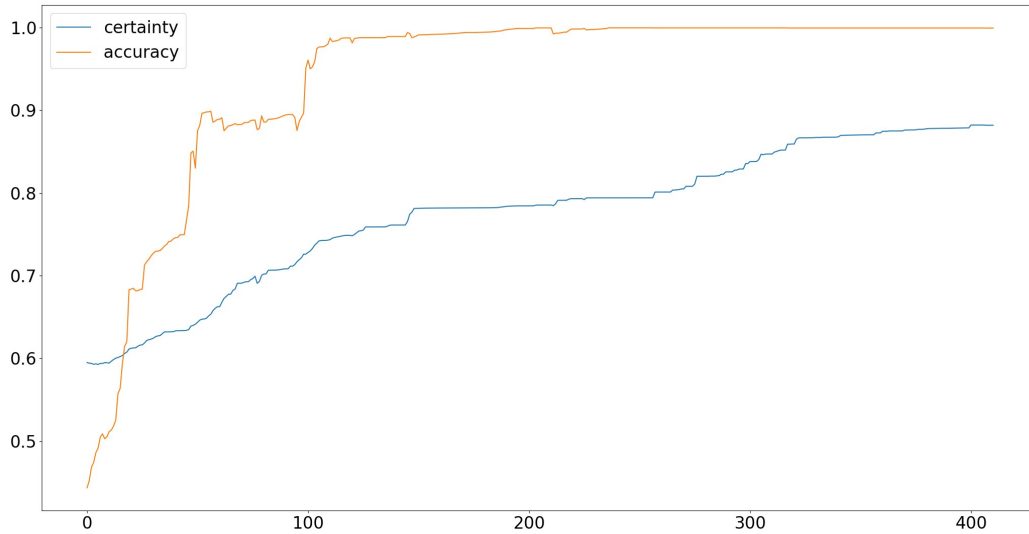


Figure 5.1.5: Active Learning Accuracy / Certainty

The Multi-Cones approach converges faster than Monte Carlo for the same number of samples. Both require in excess of 100,000 samples in order to converge. The Active Learning approach, on the other hand, requires only 237 iterations (252 samples) to learn the function with 99.98% accuracy and 79.41% certainty. As the number of iterations increases, certainty increases to 88.20% and accuracy decreases to 99.95%. When running the Active Learning algorithm on a model for which we have no ground truth, certainty is the metric which we can observe and use as the stopping criterion.

5.2 STT-MTJ 5.6ns Nominal Delay

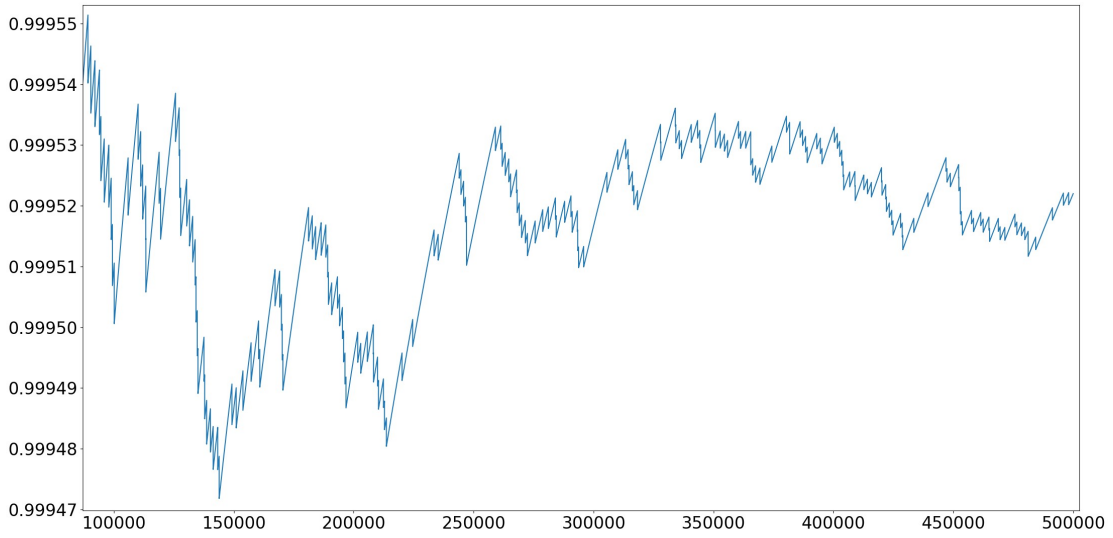


Figure 5.2.1: Monte Carlo Yield Estimate

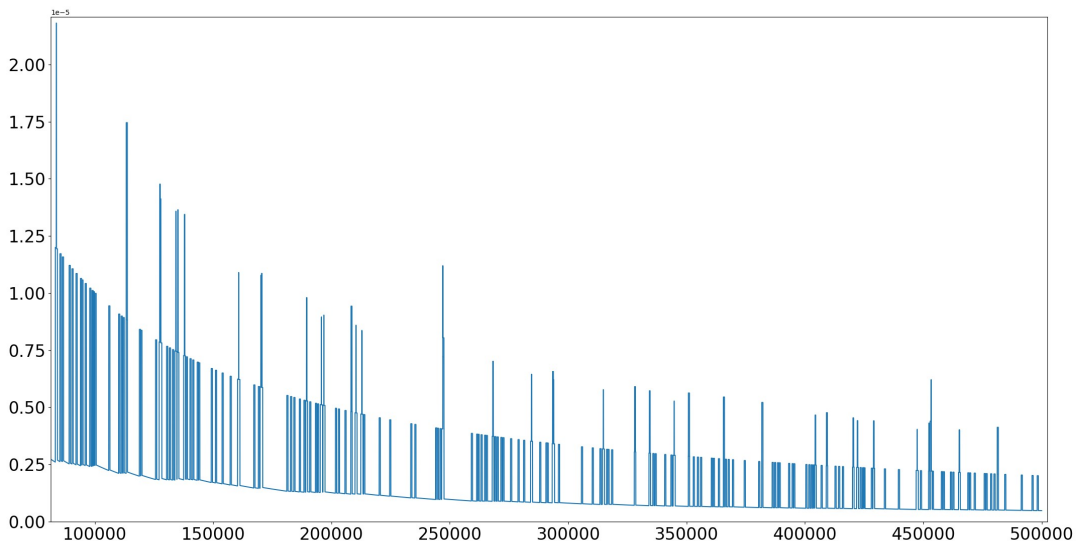


Figure 5.2.2: Monte Carlo Convergence Estimate

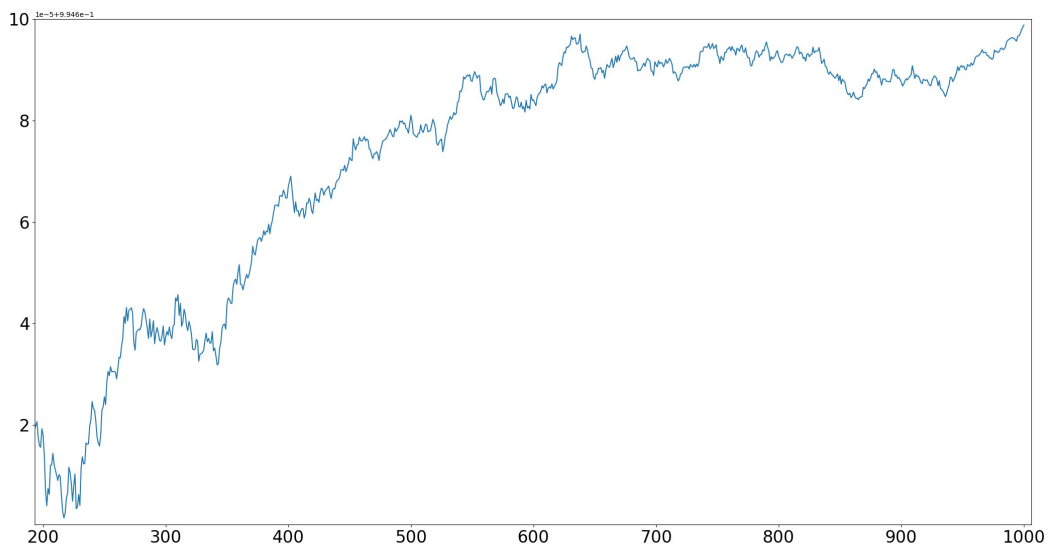


Figure 5.2.3: Multi-Cones Yield Estimate

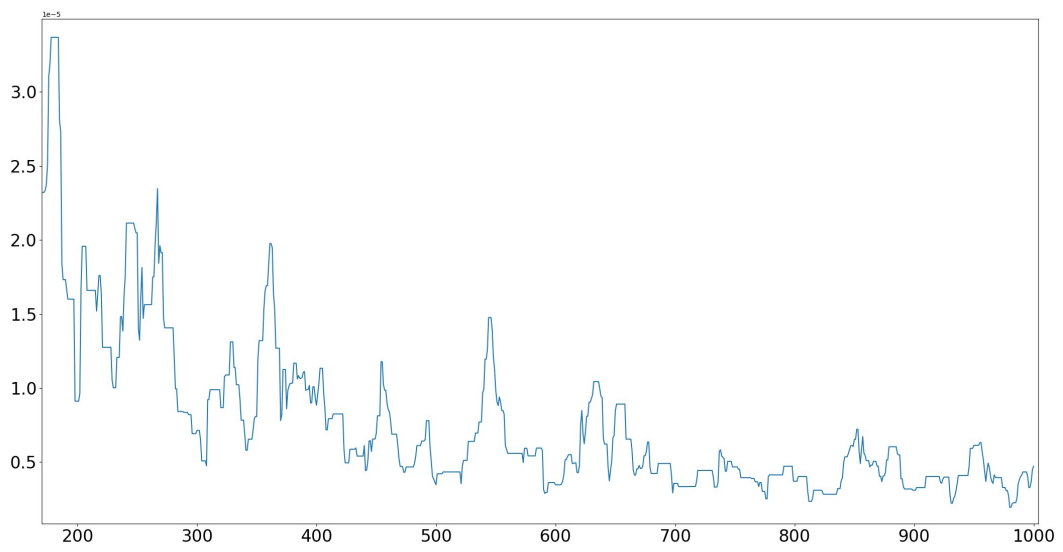


Figure 5.2.4: Multi-Cones Convergence Estimate

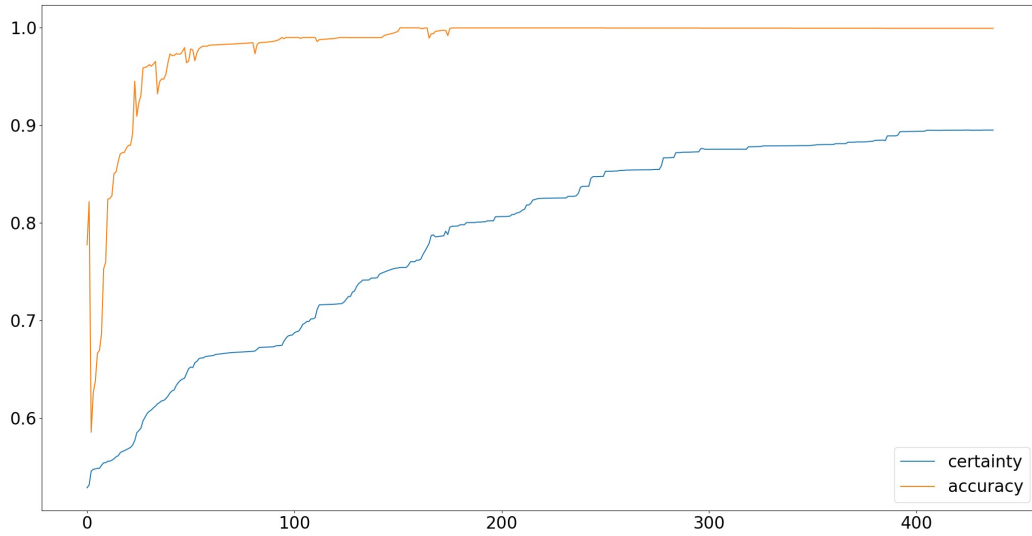


Figure 5.2.5: Active Learning Accuracy / Certainty

We chose a higher nominal delay so that failure becomes rarer. Similarly, the Multi-Cones approach converges faster than Monte-Carlo. Active Learning requires only 152 iterations (167 samples) to learn the function with 99.99% accuracy and 75.41% certainty. As the number of iterations increases, certainty increases to 89.5% and accuracy decreases to 99.94%.

CHAPTER 6

FUTURE WORK

6.1 SPICE Circuit Generator

The SPICE circuit generator currently relies on components implemented in the ngspice C++ wrapper. Separating those components into an independent project would allow the circuit generator to be used in conjunction with any other SPICE simulator if a C++ interface for the simulator is provided.

6.2 ngspice STT-MTJ Integration

Verilog-AMS model integration into ngspice using ADMS is an experimental feature that does not readily have support from ngspice maintainers. Documenting the integration process, from satisfying dependencies to compilation to C code, patching, building, then extensively unit testing would go a long way to transform this from an experimental feature to one that could be provided and maintained in the main trunk.

6.3 Robustness Guided Verification (C++)

Various benchmarks that are provided in the MATLAB version are not yet ported to the C++ version. Porting them in order to demonstrate the concise, yet expressive interface and performance improvements would be beneficial. Speeding up the active learning algorithm through running embarrassingly parallel matrix operations on the GPU would make the algorithm much more attractive due to significantly lower execution time. Adding more estimation algorithms, either inspired by the active learning approach or otherwise, would aid in making the framework the go-to framework for robustness guided verification in its various approaches, algorithms, and metrics.

CHAPTER 7

CONCLUSION

The dearth of readily-available tools to simulate electric circuits under the Active Learning approach required the synthesis of such tools, often from the ground up. Obtaining experimental results was made vastly easier due to the simplification of the needed interfaces, significant improvements to performance, and improvements to maintainability and extensibility while maintaining strong performance due to the use of modern C++.

Memory circuits stand to benefit greatly from the Active Learning approach over the use of Monte Carlo or the Multi-Cones approach, particularly circuits that are expensive to simulate (in terms of execution time). The Active Learning approach spends almost all of its execution time predicting the model iteration after iteration. At a certain point, the expensiveness of simulation, as well as the orders of magnitude more samples needed for the aforementioned two approaches, favors the use of the Active Learning approach to obtain experimental results more quickly.

BIBLIOGRAPHY

- [1] Greenberg, “Hackers Fool Tesla S’s Autopilot to Hide and Spoof Obstacles,” *Wired*, 2017. [Online]. Available: <https://www.wired.com/2016/08/hackers-fool-tesla-ss-autopilot-hide-spoof-obstacles/>. [Accessed: 05-Apr-2018].
- [2] J. Torchinsky, “Does Tesla’s Autopilot Suffer From A Dangerous Blind Spot?,” *Jalopnik*, 2016. [Online]. Available: <https://jalopnik.com/does-teslas-semi-autonomous-driving-system-suffer-from-1782935594>. [Accessed: 05-Apr-2018].
- [3] G. E. Fainekos, “Robust Testing and Testing Robustness for Cyber-Physical Systems”, School of Computing, Informatics, and Decision System Engineering, Arizona State University, Tempe, Arizona, 2011.
- [4] Y. Annapureddy, C. Liu, G. Fainekos and S. Sankaranarayanan, “S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems,” *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 254-257. 2011.
- [5] A. Fehnker, F. Ivancic, “Benchmarks for Hybrid Systems Verification”. In: R. Alur, G. J. Pappas (eds) “Hybrid Systems: Computation and Control”. *HSCC 2004. Lecture Notes in Computer Science*, Vol. 2993, 2004.
- [6] B. Settles, “Active Learning Literature Survey”, *Computer Sciences Technical Report, 1648*.
- [7] B. Bryan, “Actively Learning Specific Function Properties with Applications to Statistical Inference,” Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2008.
- [8] H. Abbas, rob-guided-verif, (2016), GitHub repository. [Online]
- [9] MATLAB parfor, 2020. [Online]. Available: <https://www.mathworks.com/help/distcomp/parfor.html>. [Accessed: 05-Apr-2018]
- [10] M. Herlihy and N. Shavit, “The art of multiprocessor programming”, *Morgan Kaufmann*, 2011.
- [11] A. Issa, R. Kanj, A. Chehab and R. Joshi “Yield and Energy Tradeoffs of an NVLatch Design using Radial Sampling,” *2017 IEEE International Conference on IC Design and Technology (ICICDT)*, pp. 1, May 23 2017, Accessed on: Mar, 13, 2018. [Online]. Available: <http://ieeexplore.ieee.org.ezproxy.aub.edu.lb/document/7993511/?part=1>
- [12] STT MRAM SPICE model manual. [Online]. Available: <http://nimo.asu.edu/memory/download/stt/manual.pdf>
- [13] R. Joshi, K. Kim and R. Kanj, “FinFET SRAM Design,” *Nanoelectronic Circuit Design*, pp. 55–95, 2010.
- [14] J. Mockus, “Bayesian approach to global optimization: theory and applications,” *Springer Science & Business Media*, Vol. 37, 2012.

- [15] Z. Wang, F. Hutter, M. Zoghi, D. Matheson and N. de Freitas, "Bayesian optimization in a billion dimensions via random embeddings," *Journal of Artificial Intelligence Research*, 55, pp. 361-387, 2016.
- [16] R. Kanj, R. Joshi, Z. Li, J. Hayes, and S. Nassid, "Yield estimation via multi-cones," *DAC Design Automation Conference 2012*, pp. 1107-1112, 2012. Accessed on: July, 6, 2020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6241643&casa_token=QkYkS5pS1LoAAAAA:o7iOp8-nUH6QSybfnq_3l8A5FPaZ9xvB9KInj8nB5jrNu8SpS24CwndyBurP4satqIhdrugEQ&tag=1.
- [17] H. Vogt, ngspice, SourceForge repository, 2020. Accessed on July, 6, 2020. [Online]. Available: <http://ngspice.sourceforge.net/>.
- [18] Ngspice 32 manual, Sourceforge, 2020. Accessed on July, 6, 2020. [Online]. Available: <http://ngspice.sourceforge.net/docs/ngspice-32-manual.pdf>.
- [19] Ngspice and ADMS for Verilog-AMS modeling, Sourceforge, 2020. Accessed on July, 6, 2020. [Online]. Available: <http://ngspice.sourceforge.net/adms.html>.
- [20] M. Pusz, mp-units, GitHub repository, 2020. Accessed on July, 6, 2020. [Online]. Available: <https://github.com/mpusz/units>.
- [21] "GCC, the GNU Compiler Collection", 2020. Accessed on July, 7, 2020. [Online]. Available: <https://gcc.gnu.org/>.
- [22] "Clang: a C language family frontend for LLVM," 2020. Accessed on July, 7, 2020. [Online]. Available: <https://clang.llvm.org/>.
- [23] The MathWorks, MATLAB mex function, 2020. Accessed on July, 6, 2020. [Online]. Available: <https://www.mathworks.com/help/matlab/ref/mex.html>.
- [24] The MathWorks, Parsing Function Inputs, 2020. Accessed on July, 6, 2020. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/ways-to-parse-function-inputs.html.
- [25] The MathWorks, "BLAS Calls for Matrix Operations in a MATLAB Function Block - MATLAB & Simulink," 2020. Accessed on July, 6, 2020. [Online]. Available: <https://www.mathworks.com/help/simulink/ug/blas-calls-for-matrix-operations-in-a-matlab-function-block.html>.
- [26] Unicode. 2020. Accessed on July, 7, 2020. [Online]. Available: <https://home.unicode.org/>.
- [27] R. Sabra, mp-units, GitHub pull request, 2020. Accessed on July, 7, 2020. [Online]. Available: <https://github.com/mpusz/units/pull/77>.
- [28] R. Sabra, mp-units, GitHub pull request, 2020. Accessed on July, 7, 2020. [Online]. Available: <https://github.com/mpusz/units/pull/78>.
- [29] R. Sabra, mp-units, GitHub pull request, 2020. Accessed on July, 7, 2020. [Online]. Available: <https://github.com/mpusz/units/pull/92>.

- [30] "Pseudo-random number generation," 2020. Accessed on July, 7, 2020. [Online]. Available: <https://en.cppreference.com/w/cpp/numeric/random>.
- [31] R. Sabra, mp-units, GitHub pull request, 2020. Accessed July, 7, 2020. [Online]. Available: <https://github.com/mpusz/units/pull/110>.
- [32] R. Sabra, ngspice-cpp, Bitbucket repository, 2020. Accessed July, 7, 2020. [Online]. Available: <https://bitbucket.org/rsabra94/ngspice-cpp>.
- [33] OpenMP, 2020. Accessed on July, 7, 2020. [Online]. Available: <https://www.openmp.org/>.
- [34] "Intel Threading Building Blocks," 2020. Accessed on July, 7, 2020. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html>.
- [35] R. Sabra, ngspice-cpp-example, Bitbucket repository, 2020. Accessed July, 7, 2020. [Online]. Available: <https://bitbucket.org/rsabra94/ngspice-cpp-example>.
- [36] B. Ford, "Parsing Expression Grammars: A Recognition Based Syntactic Foundation," *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004. Accessed on July, 7, 2020. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/964001.964011>.
- [37] parsimonious, Github, 2020. Accessed on July, 9, 2020. [Online] Available: <https://github.com/erikrose/parsimonious>.
- [38] Jinja, 2020. Accessed on July, 9, 2020. [Online]. Available: <https://jinja.palletsprojects.com/en/2.11.x/>.
- [39] Googletest, Github, 2020. Accessed on July, 9, 2020. [Online]. Available: <https://github.com/google/googletest>.
- [40] R. Sabra, ngspice, SourceForge bug tracker ticket, 2020. Accessed on July, 9, 2020. [Online]. Available: <https://sourceforge.net/p/ngspice/bugs/487/>.
- [41] Get Started with Parallel STL, Intel, 2020. Accessed July, 10, 2020. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/get-started-with-parallel-stl.html>.
- [42] J. L., Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18 (9), pp. 509-517, 1975. doi:10.1145/361002.361007
- [43] R. Sabra, rob-guided-verif-cpp, Github, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://github.com/yasamoka/rob-guided-verif-cpp>.
- [44] Constraints and Concepts, cppreference, 2020. Accessed July, 10, 2020. [Online]. Available: <https://en.cppreference.com/w/cpp/language/constraints>.

- [45] Parameter Pack, cppreference, 2020. Accessed on July, 10, 2020. [Online]: Available: https://en.cppreference.com/w/cpp/language/parameter_pack.
- [46] Template Parameters and Template Arguments, cppreference, 2020. Accessed on July, 10, 2020. [Online]. Available: https://en.cppreference.com/w/cpp/language/template_parameters.
- [47] Function Objects, cppreference, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://en.cppreference.com/w/cpp/utility/functional>.
- [48] Execution Policy, cppreference, 2020. Accessed on July,10, 2020. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t.
- [49] Eigen, 2020. Accessed on July, 10, 2020. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page.
- [50] Boost, 2020. Accessed July, 10, 2020. [Online]. Available: <https://www.boost.org/>.
- [51] Boost.Numeric.Odeint, Boost, 2020. Accessed on July, 10, 2020. [Online]. Available: https://www.boost.org/doc/libs/1_73_0/libs/numeric/odeint/doc/html/index.html.
- [52] Interpolation, Boost.Math, Boost, 2020. Accessed on July, 10, 2020. [Online]. Available: https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/interpolation.html.
- [53] The Computational Geometry Algorithms Library, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://www.cgal.org/>.
- [54] Linear and Quadratic Programming Solver, CGAL, 2020. Accessed on July, 10, 2020. [Online]. Available: https://doc.cgal.org/latest/QP_solver/index.html.
- [55] googletest, Github, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://github.com/google/googletest>.
- [56] google/benchmark, Github, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://github.com/google/benchmark>.
- [57] The Mathworks, MATLAB quadprog function, 2020. Accessed on July, 10, 2020. [Online]. Available: <https://www.mathworks.com/help/optim/ug/quadprog.html>.
- [58] The Mathworks, MATLAB ode15s function, 2020. Accessed on July 10, 2020. [Online]. Available: <https://www.mathworks.com/help/matlab/ref/ode15s.html>.