Simulated Silicon Dopant networks - optimizing simulations, searching for boolean gates and visualization

Indrek Klanberg

2019

Supervisors: Prof.dr.ir Arend Rensink, Prof.dr.ir H. J. Broersma, Prof.dr.ir. W.G. van der Wiel

Master Thesis

Abstract

In this thesis I worked with a physical model for silicon dopant networks, a potential future solution for computation in materio. The physical model uses Kinetic Monte Carlo simulation to simulate various scenarios, and can be used to make simulated experiments at a much lower cost than we could do them on physical devices. In this work I worked on improving the simulation speed for this model, but also explored many other aspects to move towards streamlining simulated experiments, such as visualization, more general Genetic algorithm for searches, and parallelism. I also made a few example experiments, which already took advantage of the increased performance of the simulations achieved during the Thesis. When I started the Thesis, B. de Wilde had already laid the grounds for the physical model in code. I built upon that and here I document my findings on different simulation optimization attempts, visualization functions, genetic search improvement attemps, and several parallelization attempts. In the end I give my recommendation of how to perform large scale simulated experiments and suggest a few experiments to be done in the future based on the example experiments made during the Thesis.

Contents

1 Introduction

Artificial neural networks is an increasingly important topic in todays world, as they are used to solve complex problems, which were at some point supposed to be out of reach from computers, such as image recognition, sound-to-text translation and mastering complex games like Go. However, currently artificial neural network solutions use a lot of power, as simulating these networks on traditional transistor-based computers is computationally expensive. It is believed that it can be made more efficient by taking advantage of physical processes in materio (cf. [8], [4]), such as in our case silicon dopant networks.

This thesis is close collaboration with the NanoElectronics research group from Center for Brain Inspired Nano Systems at University of Twente. By now they have shown that silicon dopant networks can perform 6 basic boolean operations (cf. [4] (OR, AND, XOR, NOR, NAND and NXOR)) and further exploration of their complexity capacity is taking place. In addition to the physical devices, the group has been working on two models of those devices: Machine learned neural network which predict the behaviour of a single real life physical device, and a Kinetic Monte Carlo simulation of a physical model. In my work I will be working with the latter. The advantages of the Machine learned model is mainly speed and that it simulates quite accurately an actual physical device. The main advantage of the Kinetic Monte Carlo model is that we can create any number of simulated devices, to perform simulated experiments on a larger scale than would currently be feasible to be done on physical devices, as the number of physical devices is limited, remember that the same limitation carries over to the machined learned model.

The goal of this Thesis is to pave a path for large scale simulated experiments using KMC simulations of a physical model. I aim to achieve this by improving the simulation speed of the Kinetic Monte Carlo model, attempting to optimize the Genetic Algorithm to perform various searches, and looking into ways to use parallelism to make use of cluster computing. In addition I attempt to answer a few example research questions using the new sped up solution, and discuss the considerations one would have to make regarding time, simulation solutions and parallelism in order to perform further simulated experiments in the future. In addition I did make a few visualization functions, that can be useful to make sanity checks in the process.

In Chapter 2, I introduce the work done in the NanoElectronics group and in more detail the Kinetic Monte Carlo model used in B. de Wilde's work (cf. [11]) that is the foundation for my Thesis. In Chapter 3, I talk about the different approaches I developed to improve the speed of simulations on that model. It culminates with a comparison in time performance and accuracy. In Chapter 4, I discuss my Genetic Algorithm, which has a few features not available to the Genetic Algorithm used by NanoElectronics group. I use my Genetic Algorithm along with simulation improvements to perform 2 simulated experiments explained in Chapters 6 and 7. In Chapter 6 I examine how the number of dopants and ease of finding a XOR gate is related. In Chapter 7 I

measure the complexity of simulated networks with different number of dopants using Vapnik–Chervonenkis dimension (cf. [9], later I call it VC dimension). In Chapter 5, I discuss some of the visualization functions I developed during the thesis including how to read these visuals and how they are made. Visualization was used as a sanity check while working on all the other aspects of the thesis and is considered as a supportive tool for the overall goal. Finally in Chapter 8, I talk in more detail about how my code base is structured, where various features reside and also discuss use cases of other people using the codebase developed during my Thesis and their feedback.

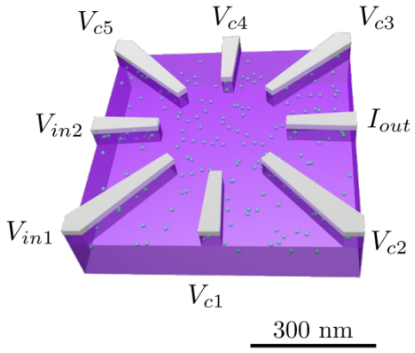

Figure 1: Figure 3.1 taken from Ref. [11]. This shows the general idea about boron dopant networks. You have a chip of silicon in which you have randomly placed dopants and 8 electrodes. Electrode endpoints form a cirlce of around 300 nm.

2 Background

2.1 Silicon dopant network

The devices that my work relates to are boron doped silicon networks. That means there is a piece of silicon around $1cm^2$ of size, which by itself is a very bad electricity conductor, and boron atoms have been added to it. Adding boron atoms creates so called "holes", which electrons can use to jump to and from. The process of making such a chip is quite complex (cf. [5]), however the important part is that at the end of it, there is around $300nm$ circle, inside which there are scattered around 100 boron atoms (from personal communication with dr T. Chen: "The number of boron dopant sites in the silicon is currently estimated to be around 100"). When I started my thesis, that number was believed to be around 30 - 40 atoms. This has had some impact on my Thesis.

Then 8 aluminium electrodes are connected to the edge of that circle, 2 for input, 1 for output and 5 for configuration. The configuration electrodes have some steady voltage, which has to be maintained during experiments. The input voltages are also maintained and the output current is measured. Now the idea is that since the electrodes are separated enough between themselves, the current that is formed must go through the dopant sites as electrons jump from hole to hole, using variable range hopping as a mechanism under 77K temperature (77K is due to the technique being used, which is cooling the experiment with liquid nitrogen, as nitrogen's boiling temperature is 77K).

In addition to the boron dopants in the silicon, which act as acceptors, there are also donor sites in silicon due to impurities. While acceptors create holes where electrons can jump to, in which case they get a negative charge, donor sites are impurities within Silicon which have given away their electron, and

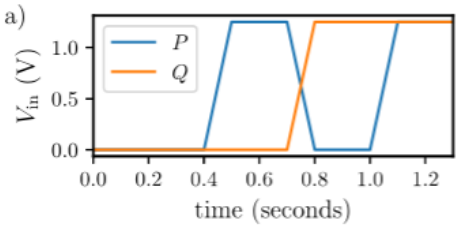

Figure 2: Figure 5.6 a taken from [4]. This shows the input voltages, that represent the true/false values for the experiments. P and Q represent the two inputs. At first both of them have 0 voltages, meaning they represent false values as input. Then P changes to true, then P to false and Q to true and finally both of them are true. This setup is used to go through all the input combinations to assert if the device is working as the intended boolean gate.

therefore have a positive charge. It is estimated that there would be 10 times fewer of them compared to boron dopant sites. They also influence the behaviour of the whole system electrostatically, meaning that based on the current model, they do not allow jumping of electrons to and from them, but since they have a charge, they do influence the behaviour of the system. For example, if a donor and acceptor sites are close to each other, then it is very likely that whenever an electron jumps to the acceptor site, it will get stuck there, because of the pull from the nearby donor site.

2.2 State of experiments

The NanoElectronics research group has demonstrated such networks to be able to perform basic Boolean logic. This means that by having high or no voltage for the input electrodes, representing true and false respectively (See figure 2), the output electrode had a current which could be mapped to a true or false value.

For operations such as XOR, negative differential resistance is required, which demonstrated that such systems have such a property. Negative differential resistance means, that as the voltage of one of the electrodes increases, then under certain conditions, the current of the output starts to instead decrease, meaning that as the voltage increases, the resistance between the input and output electrodes must increase even faster. The reason why XOR demonstrates a device having negative differential resistance, is that while for input combination (true, false), the output is true, meaning the output electrode has a high current, then for input combination (true, true), where the voltage of one of the inputs has indeed increased (needed to represent the true value), but the output current has become smaller in order to represent the expected false value. Since the system managed to perform as a XOR gate they therefore have negative differentila resistance.

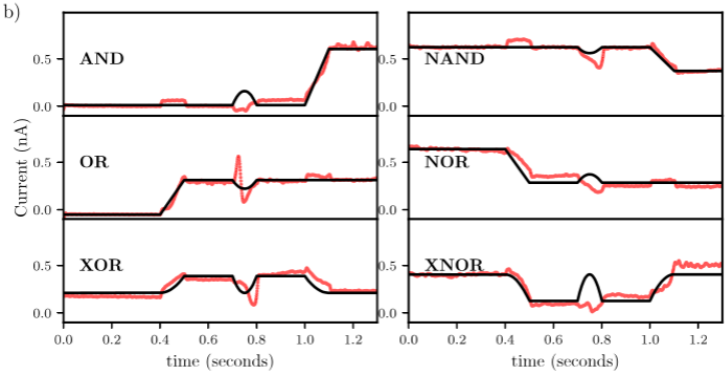

Figure 3: Figure 5.6 b from [4]. This shows the output voltages when performing different boolean operations with the input displayed in figure 2

Note that the output is typically not either high or none, but more like high or less high. It was however possible to draw a line between the expected true and false values, meaning they were separable (for example see output of XOR from figure 3). That means that there existed a current value *cur*, so that for each possible input configuration ((true, true), (true, false), (false, true), (false, false)), when we expected true for output it was higher than the chosen current value *cur*, and when we expected false, it was lower than the chosen current value *cur*.

For the whole network to be able to perform as one of the 6 Boolean gates, AND, OR, XOR, NAND, NOR or XNOR, they had to use a genetic algorithm to learn values for the 5 configuration electrodes ([4] Chapter 4.6). For each of these Boolean operations, a different configuration had to be learnt. And since the placement of boron atoms is different for all the samples (as we have little to no control over that), you also have to learn new configurations when using a new sample, as the results don't carry over. This is quite a limitation for practical use.

2.3 Kinetic Monte Carlo simulation

For part of his Master thesis, B. De Wilde ([11]) has implemented a new measurement and simulation system for these networks. For simulations he uses a rejection free kinetic Monte Carlo algorithm, which given electrode voltages and boron positions, calculates the currents for all the electrodes.

In general Monte Carlo algorithms work by choosing a small set of events or possible outcomes randomly instead of for example covering the entire search space. This is useful when the complete state space or combination of events is too vast and too complex to solve a problem using brute force or analytically. The hope is that by choosing enough random states or events we get a good enough sample size to arrive close to the most optimal/accurate solution.

The Kinetic Monte Carlo algorithm simulates some process, by starting in an initial state and then choosing occurring events one by one from all the possible events that can happen in a given state. The events are chosen using weighted randomness and the weights come from transition rates, which is not provided by the algorithm but has to be given by the model. Finally the transition rates are used to calculate the passing of time between all events.

In general there are two different KMC solutions - rejection KMC and rejection free KMC. The difference of rejection KMC and rejection free KMC is mainly that rejection KMC allows for not picking any event during an iteration, and because of that there is a slight difference in calculating the time step and picking of the events.

For this algorithm to work we need clear definitions of all the possible states and a starting state. We need to define what are the neighbouring states, and how to calculate or what are the rates between states. A state i is neighbouring another state j, if rate $r_{ji} > 0$. A rate shows how likely a state would transition to that neighbouring state in a single time step.

The overall flow of the Rejection free Kinetic Monte Carlo algorithm is the following:

1. set time $t = 0$, and choose a starting state and set it as current state. Let current state at all times be marked as k
2. For the current state k, form a list of transition rates N_k, so that all possible transitions from the current state to its neighbouring states are present in that list. Let $rate_{ki}$ be the i-th element of that list.
3. Use the transition rates to generate a list, from which to randomly choose an event. Let that list be R, then $R_i = \sum_{j=0}^{i} rate_{kj}$. Meaning the i-th element of R is the sum of 0 to i elements in N_k. We call R a cumulative list.
4. Generate a random number rnd in [0 .. 1)
5. pick the event index i so that $R_i \geq rnd * R_{last} > R_{i-1}$, where R_{last} is the last element in the cumulative list R. Note that R_{last} is the sum of all rates in step 2.
6. Execute the chosen event, by moving to the state that is matched by the transition $rate_{ki}$, and set k as the chosen neighbouring state i
7. Generate another random number rnd_2
8. increment time t by $t = t + R[last]^{-1} ln(1/rnd_2)$. This is given by the algorithm, but basically what it says is, that the higher the total rates of all possibilities, the lower is the length of the timestep for the current iteration.
9. If enough iterations have been run, then end, else go to step 2. This is either set by the user, based on simulation time constraints, or by a given convergence criterion.

As you can see the time is incremented inversely to the total sum of all rates. This is because the more likely any event is to happen, the more often some event happens and therefore in each step we can progress less time.

To decide when to end one can either provide a specific number of events after which the algorithm is terminated, or one can have some kind of convergence criterion, after achieving that, we terminate. In my thesis I don't use any criteria, but instead run simulation for a given number of hops (events, but we use the term hop, as each event is an electron hopping from one site to another, or between an electrode and a site).

2.4 Current implementation specifics

The core of using KMC in physical model is simulating single electron jumps, one at a time. Therefore, in this case, states represent the electron occupations in the dopant sites. Since boron atoms have 3 electrons in the outer layer but silicon has 4, then it creates a sort of hole, which can be occupied by an extra electron (this is due to the fact that optimal number of combined electrons is 8).

Each site either has an extra electron or it doesn't. Therefore in a 30 site system there would be 2^{30} possible different states. A neighbouring state is a state which can be reached by making an electron jump between two sites, or between a site and an electrode. For an electron to jump from one site to another, the first site has to be occupied and the second one cannot be occupied. For an electron to jump from an electrode to a site, the site has to be unoccupied, and for the electron to jump from a site to an electrode, the site has to be occupied.

Since donor sites only effect the system electro-statically, their influences has to be calculated only once, and can therefore be reduced into a constant.

An event in our case is a single electron jump. If the electron either jumps to or from an electrode, we keep track of it, so that we can in the end estimate the current for all the electrodes by also using the time calculation method from the Kinetic Monte Carlo algorithm.

The rates are calculated based on the Miller-Abraham hopping rate. That in return requires to estimate the change of system energy, for which we only take into account Coulomb interaction between the dopant sites and local chemical potential. A more detailed explanation can be read in [11]. Note that in our calculations we separate the acceptor and donor sites.

Miller-Abraham hopping rate used to calculate rates for our algorithm (cf. [11], [7]).

$$rate_{ij} = \begin{cases} v_0 e^{-2\frac{d_{ij}}{a} - \frac{\Delta\varepsilon_{ij}}{kT}} & if \Delta\varepsilon_{ij} > 0 \\ v_0 e^{-2\frac{d_{ij}}{a}} & if \Delta\varepsilon_{ij} \leq 0 \end{cases} \tag{1}$$

$rate_{ij}$ is the rate at which an electron jumps from site i to site j. For our purposes v_0, a, k, T are constants given by the physicist, and they can be precalculated, $e = 2.71828$, d_{ij} is the distance between i-th and j-th site, which has to be calculated only once per site placement (which does not change during simulation), and $\Delta\varepsilon_{ij}$ is the energy difference caused by a jump from site i to site j, which is something we have to calculate and depends on the current state of the system (electron placements).

To calcualte $\Delta\varepsilon_{ij}$, we need to be able to calculate the total energy of the system. We do it so:

$$H = \frac{e_0^2}{4\pi\epsilon_0\epsilon}\left(\frac{1}{2}\sum_{i\in acc}\sum_{j\in acc\setminus\{i\}}^{acc}\frac{n_i n_j}{d_{ij}} - \sum_{i\in acc}\sum_{k\in don}\frac{n_i}{d_{ik}} + \frac{1}{2}\sum_{n\in don}\sum_{k\in don\setminus\{n\}}\frac{1}{d_{nk}}\right) - e_0\sum_{i}^{acc} n_i V(\mathbf{pos_i})$$

For our purposes $\pi, \epsilon, \epsilon_0$ and e_0 (in this case e_0 is elementary charge and is not related to $e = 2.71828$) are constants given by the physicists. What we do track is n_i, which is 1 if site i has an extra electron in it, or 0 otherwise. d_{ij} is once again the distance between sites i and j. V is a function of site position pos_i and electrode voltages, the latter staying the same during a single simulation. As an electron jumps from one site to another the value of H changes, and this change is the energy difference $\Delta\varepsilon_{ij}$. Events that would decrease the energy of the system are of course more likely to occur.

However calculating the entire energy each iteration and the energy in all the neighbouring states would be quite costly. We can improve that by first noting that the effect of donor sites don't change by changing the state:
$\frac{1}{2}\sum_{n\in don}\sum_{k\in don\setminus\{n\}}\frac{1}{d_{nk}}$ stays constant.
Secondly, to quickly calculate the change of energy a transition would effect, we can instead calculate for each site, the impact of adding an extra electron. To do that we just sum up all the terms which would be added to H, when we change n_i from 0 to 1.

$$\varepsilon_i = \frac{e_0^2}{4\pi\epsilon_0\epsilon}\left(\sum_{j\neq i}^{acc}\frac{n_j}{r_{ij}}\right) - \sum_{k}^{dom}\frac{1}{r_{ik}} - e_0 V(\mathbf{pos_i})$$

We call ε_i site energy for site i. Note that since donor sites don't change their charge in our model, and neither do the electrode voltages (at least during a single simulation), then these parts stay the same, and are not re-calculated during hops. However the electrostatic effect of donor sites, as well as the effect

of electrode voltages are captured by $eV(pos_i)$, which physicists call chemical potential at acceptor i position. This value does not change during the simulation. We can use these site energies to calculate $\Delta\varepsilon_{ij}$, the energy difference caused by electron jumping from site i to site j:

$$\Delta\varepsilon_{ij} = \varepsilon_j - \varepsilon_i - \frac{e_0^2}{4\pi\epsilon_0\epsilon r_{ij}}$$

To understand why this works let's consider what each of those terms mean. ε_j is the amount of energy that would change when an electron jumps to site j. ε_i is the amount of energy that would change when an electron jumps to site i. But since an electron jumps from site i we instead put a minus in front of it. However since since ε_j is a sum which also currently consists of term $\frac{e_0^2}{4\pi\epsilon_0\epsilon r_{ij}}$, since n_i used to be 1 and there will no longer be an electron in site i after the jump, we have to subtract that term from ε_j. Calculating energy difference for a jump between electrode and a site is even easier. As the electrode is considered to be able to maintain it's voltage stably, then for some acceptor i and electrode j, if we have a jump from i to j we get $\Delta\varepsilon_{ij} = -\varepsilon_i$. And if we have a jump from j to i we get $\Delta\varepsilon_{ji} = \varepsilon_i$.

Now we have everything we need to calculate the rates at each step.

2.5 From physical to model

One important question is how does the model relate to the real life experiments. And for that I will first discuss the scales of various forces inside the physical devices, and then I'll discuss what that translates to in our simulation.

As discussed previously the diameter of the area where the boron atoms reside is around $300nm$. The voltage range, which is used in the experiments is from -2 V to 2 V, but may be lower for some devices. If you would increase the voltages too large for a device, you would risk changing the properties of the device, meaning it will no longer give the same results as it gave before (for example if you trained a XOR gate for the device, you cannot use that result anymore).

The current seen in the output electrode has a really wide range. It is between the orders of $10^{-6}A$ and $10^{-11}A$. And the difference at which we detect true and false values varies similarly. For example in [4] Figure A.2 he demonstrated finding boolean gates so that the differences between true and false current values was in the orders between 10^{-10} and 10^{-11}.

When moving to computer simulations, it is however non-trivial to get a 1-on-1 match with reality. These are just a few complications:

- We get a current based on the number of electrons that have hopped to or from an electrode and dividing it by the time that has passed. However we increment time based on the calculated rates, which all are linearly related to a constant v_0 in equation 1, which we call hopping rate. The problem is we do not know it's exact value, which means we cannot know exactly what the current value would be that is arrived at in a simulation.

- Second parameter a in the same equation 1, refers to localization length and is also unknown and we currently only have a rough estimate for that (cf. [11]). In this Thesis I use a dimensionless value for $\frac{a}{R}$ with a value of 0.25, R in this case is the average distance between boron sites. Basically the smaller a/R is, the harder it is for electrons to jump over larger distances. This value is taken from [11]. The justifications for that are also described there and are a very important part of that Thesis, which is why I am not going to delve any deeper here. Shortly the parameter values were trained so that the resulting model most resembles real life experimental results.

- Finally, we have a coulumb interaction energy $I_0 = \frac{e_0^2}{4\pi\epsilon_0\epsilon}\frac{1}{R}$ (cf. [11] chapter 4.2.1). We do not know the exact value of I_0, but in our model we use a dimensionless parameter value for $\frac{I_0}{kT} = 100$, where kT is thermal energy. You can read about it in more details in [11], but the most important takeaway is that, the larger the value of $\frac{I_0}{kT}$, the more electrons repel each other between sites, making it hard for having many more electrons in the system than positively charged impurity sites. Secondly the used value for I_0 is physically impossible, but it does provide us with simulation results which resemble very closely the behaviour we have seen in real life samples. The current explanation is that its unfeasably large value is explained due to us using much fewer dopants in our simulated system than the latest estimations have estimated for real life samples ([11]).

3 Simulation Improvements

In the course of the Thesis I implemented several simulation approaches with the goal to find solutions that would perform faster with an acceptable margin of error. I generated 2 sets of tests that would be used to evaluate the error, but also the time performance of these approaches.

All the approaches were implemented in the Go language.

The approaches shortly are:

- Kinetic Monte Carlo (I will call it KMC from now on), but implemented in Go. This is essentially the same approach used by B. De Wilde in his work (cf. [11]), but with slight optimisations to calculations of site energies (see Chapter 2.4) and an option to record transition rate lists, which can be reused if we return to the same state.

- Using an approach with Probabilistic occupation. This means that instead of discretely considering each site to either have an electron or not, we consider it having an electron with a certain probability. And at each time-step we instead change the probabilities based on all the possible transitions. The hope with this approach was to see faster convergence to a result, but the main concern was its range of error.

- Transition rates pruning, which means that I would not consider transitions between sites when the chemical potential between those sites was low enough (due to range, placement of donors or placement of electrodes) compared to other pairs. This would provide me with certain gains in speed, but would risk in lowering the accuracy. However, this approach may also shed light into the importance of low probability events.

3.1 Using Go language

The original KMC solution for the silicon dopant networks was written in Python by Bram de Wilde. Since Python by default is not a compiled language, but instead runs the code on runtime, then based on my prior experience it is usually around 5 - 10 times slower compared to compiled languages like C++, Java or Go language. However this was already addressed by Numba JIT compiler, which could turn the KMC function into compiled code, meaning that this weakness of Python would disappear. Numba did however have a few problems. Overall it is only able to compile limited Python code, which meant that you cannot take the full use of the language. This became a problem for example when I wanted to save the probabilities of visited states into a hash map and generate 64 bit key using binary operations in order to do so.

Now the advantage of Python is its ease of use, and that many in the Nano Electronics group are familiar with it, and its two very popular libraries Numpy and Mathplotlib. So in order to try all the different optimizations and parallelisation, I decided to implement the KMC simulation function itself in Go language, and turn it into a library, which can then be called from Python using

builtin Python library CTypes and builtin function cdll to call code made in Go. This solution also has a few limitations, namely I cannot pass 2-dimensional arrays directly, but instead have to wrap them into 1-dimensional arrays and unpack them on the other side. However after you resolve such communication issues, you will have access to a full compiled language, and since we are using Go, we have also access to very simple and supposed-to-be efficient parallelism by using Go routines. Making go functions available to python was done following a tutorial in [2].

3.2 Kinetic Monte Carlo optimisations

3.2.1 Site energy calculation

In the baseline solution of Kinetic Monte Carlo simulation, we use a helper variable called site energy to quickly calculate the change to the overall energy in the system for all the possible jumps. As mentioned previously, the site energy for site i, called ε_i shows how much the energy changes when an electron would jump to site i. By knowing the site energies for all the sites we can quickly calculate the energy difference caused by any transition. However, as a transition takes place, the site energies for all the sites actually change.

The optimisation at hand is about knowing exactly how much it changes. If you consider the original calculation for site energies:

$$\varepsilon_i = \frac{e_0^2}{4\pi\epsilon_0\epsilon}\left(\sum_{j\neq i}^{acc}\frac{n_j}{r_{ij}}\right) - \sum_{k}^{dom}\frac{1}{r_{ik}} - e_0 V(\mathbf{pos_i})$$

Notice, that for most i values, the values in the sum stay the same, except for the 1 or 2 sites that were involved in the jump. This means that if we had a jump from site a to site b, then for all sites except a and b, ε_i can be calculated using its old value ε_{iold} and knowledge about a and b as follows:

$$\varepsilon_i = \varepsilon_{iold} + \frac{e_0^2}{4\pi\epsilon_0\epsilon}\frac{1}{r_{ib}} - \frac{e_0^2}{4\pi\epsilon_0\epsilon}\frac{1}{r_{ia}}$$

$$\varepsilon_a = \varepsilon_{aold} + \frac{e_0^2}{4\pi\epsilon_0\epsilon}\frac{1}{r_{ib}}$$

$$\varepsilon_b = \varepsilon_{bold} + \frac{e_0^2}{4\pi\epsilon_0\epsilon}\frac{1}{r_{ia}}$$

This lowers the number of calculations necessary for site energies from N^2 to $2N$ at each hop, where N is the amount of sites for which we have to calculate the site energy. Since we also have to calculate the values for transition rates, which there are also roughly around N^2 of, then the overall reduction to time performance is only around 2 times, since calculating the transition rates becomes the main bottleneck after this optimization.

3.2.2 Reuse of transition rate calculations

In order to tackle the large price that we pay by calculating the transition rates at each hop, I considered reusing these calculations when we arrive to the same state as previously. As we have N acceptor sites, which can either have or not have an electron, we have 2^N states in total, for which the transition rates could be different.

In practice we decoded a state using 64 bits, where each bit was 1 if corresponding site had an electron and 0 otherwise. This meant that this optimisation cannot be used with more than 64 acceptor sites, for now this is enough but one can consider expanding it, though as the number of unique states increases and the memory requirement for storing the information for each state also increases, there are quite a few things to keep in mind when doing so. At the start our main concern was how effective this solution would be with 30 dopant networks, as this was supposed to be around the number of dopants in our real life experimental samples. However, this turned out to not be the case, in real life samples it is now estimated that there are more than 100 dopant sites (see chapter 2.1).

In table 1 you can see how often a state found previously is revisited depending on the number of hops and the number of dopants in the network.

Now the number of unique states we find during KMC depends heavily on the parameter values of a and I_0. For testing the usability of this approach I used the parameter values provided by B. De Wilde $\frac{a}{R} = 0.25$ and $\frac{I_0}{kT} = 100$ (cf. [11] chapter 4.2.1). Also when generating the random layouts and electrode voltage values I used quite a high voltage range for electrodes - between -2V and 2V. This is the highest reasonable voltage that could be used so to see how the simulations perform under the most difficult conditions.

Table 1: This table shows how many different states on average were encountered for 20, 30 and 60 dopant networks. For each of those N values, we generated 100 random tests (random placement and random voltages for electrodes), and ran for 10 million hops. We recorded the state for every power of 10 hops, and in the table the average of these results is displayed.

hops	20D states	30D states	60D states
10	9.9	10.0	9.9
100	55.3	68.6	80.9
1000	129.2	178.0	258.8
10000	289.1	520.1	669.8
100000	600.1	1317.2	2216.6
1000000	1114.5	2986.7	6981.5
10000000	1870.5	6162.3	19932.7

As you can see from table 1 there are not many unique states for 30 and 60 dopant networks considering the total number of states is 2^{30} and 2^{60} respectively. That means we can use this optimization to easily simulate up to 60 dopant networks, and if we made changes to the key generation, then possibly

go even further. However the main thing to consider when dealing with this approach is memory. As you increase the size of the network, the number of transition rates you would have to store increases polynomially (N^2, where N is number of dopants). Even if the number of unique states continues to rise slowly, since remembering each unique state costs more it is still going to be a problem. Running a KMC without this optimization is however going to be even worse time wise for larger dopant networks.

To get a better idea, suppose with have N_A of acceptors and N_E of electrodes. Then the number of possible transitions from each state is $N_A * (N_A - 1) + N_A * N_E * 2$.

For example when storing all the transition rates for a single state for 60 dopant networks with 8 electrodes you would need at least $(3540 + 960) * 32 = 144000$ bits or 18000 bites (with the assumption one uses 32 bit float for the transition rate), so on average, since we have to save all the transitions for 20000 unique states, you would need around $360M$ of memory, and potentially much more in some scenarios.

3.3 Probabilistic occupation

One idea to improve the performance of the simulations was the following: instead of simulating the process discretely, one jump at a time, where each site either has an electron or doesn't, we would consider, what if each site had an electron with some probability. And instead of doing only one jump at each step, we would do all the jumps with a certain probability. There is no question that this model would be different from the one we had previously, and I have no physical justifications for this shortcut. But the hope was that this solution would converge to a solution faster, and if we measured the error compared to the KMC model, we could still find it usable. The question is if this is an useful approximation.

3.3.1 Algorithm

The algorithm itself has lot of similarities to the KMC solution. We again need to calculate both the rates and the site energies. The calculation of site energy does not actually change except for the fact that the values of n_j are a probability within range $[0, 1]$ in the following calculation:

$$\varepsilon_i = \frac{e_0^2}{4\pi\epsilon_0\epsilon}(\sum_{j\neq i}^{acc} \frac{n_j}{r_{ij}}) - \sum_{k}^{dom} \frac{1}{r_{ik}} - e_0 V(\mathbf{pos_i})$$

The second adjustment is that since we consider transitions between all the sites, and each site has an electron with a certain probability, we have to consider what is the probability that such a transition is possible. It is still only possible when one of them has an electron and the other one doesn't. So if we consider a jump from site a to site b, then we have to multiply the original $rate_{ab}$ to which

we arrive to with $n_a * (1 - n_b)$, since that is the probability for the scenario where site a has an electron and site b doesn't have one.

So the new rate formula is:

$$rate_{ij} = n_i * (1 - n_j) * \begin{cases} v_0 e^{-2\frac{d_{ij}}{a} - \frac{\Delta\varepsilon_{ij}}{kT}} & if \Delta\varepsilon_{ij} > 0 \\ v_0 e^{-2\frac{d_{ij}}{a}} & if \Delta\varepsilon_{ij} \leq 0 \end{cases}$$

Here n_i is the probability that site i has an electron (and same for n_j, but for site j). The rest is same as in KMC.

When in KMC at each step we did a single jump, then in probability occupation, I adjust all the probabilities. Instead of picking an event at random, I use all the rates and adjust the corresponding occupation probabilities using rate values as weights. I make sure that the total sum of change in a single step does not go above a value max_{change}, which by default is 1, but I also have to make sure that none of the values n_i go out of the range of [0, 1].

To achieve these goals I firstly calculate total rate affecting each site, I call it $diff_i$ (diff as difference from current state to the next one), where i is site index.

$$diff_i = \sum_{j \neq i}^{acc} rate_{ji} - \sum_{j \neq i}^{acc} rate_{ij}$$

Secondly I calculate the total sum of rates $total_{rates}$.

$$total_{rates} = \sum_{i}^{acc} \sum_{j \neq i}^{acc} rate_{ij}$$

Now for each site I can calculate if given that the total change is 1, will it's value remain within the range [0, 1]. For that I can calculate the expected new value n_{iNew}

$$n_{iNew} = n_{iOld} + diff_i / total_{rates}$$

If for some i, n_{iNew} would go outside the range of [0, 1], then I can calculate a value $required_{change}$. If max_{change} is larger than that, I set the value of max_{change} to $required_{change}$. At the beginning of each hop I set max_{change} back to 1.

If n_{iNew} is lower than 0, then $required_{change} = n_{iOld}/(-diff_i/total_{rates})$. If n_{iNew} is larger than 1, then $required_{change} = (1 - n_{iOld})/(diff_i/total_{rates})$. Now if I set the max_{change} to the lowest $required_{change}$ among all sites, then I can calculate the actual n_{iNew} like so:

$$n_{iNew} = n_{iOld} + diff_i / total_{rates} * max_{change}$$

Because of the value of max_{change}, it is guaranteed that n_{iNew} will be in the range [0, 1]. In order to calculate the current we also keep track of electrons moving to and from the electrodes. This is done similarly tp changing the electron odds in the dopant sites, however these values do not have any limit, so electrodes are not considered when calculating max_{change}.

So to calculate electrode occupation eo_i for some electrode i, firstly note, that the same formula for rate is used for between acceptors and between electrode and acceptor site. Secondly the energy difference caused by acceptor to electrode or electrode to acceptor is discussed in 2.4, and so we can update electrode occupation at each step as follows:

$$eo_{iNew} = eo_{iOld} + diff_i/total_{rates} * max_{change}$$

Finally the timestep is calculated similarly as in KMC, with the only difference that we also have to take into account max_{change}. So time t is adjusted like so $t = t + R[last]^{-1} ln(1/rnd_2) * max_{change}$ (compare with Chapter 2.3).

There were 2 main questions regarding this approach.

- How well does this solution predict the currents on the electrodes.
- How fast does the result converge.

Note that I have little to no reason to believe that this model would match the Kinetic Monte Carlo model, or that it would be somehow more accurate to the real world solution. However the hope was that this method would converge faster and if the error is not too big, could still be useful in different search algorithms where I need to simulate hundreds of test cases each step and need to do hundreds or thousands of steps to reach some kind of useful result.

3.4 Pruning

Another approach to reduce the time we spend to calculate transition rates is to decide not to consider some of the transitions. The hope is by not considering cases, that are very unlikely to occur, we will not introduce too big of an error while at the same time considerably reducing the number of transition rates we calculate.

In formula 1, the $\Delta\varepsilon_{ij}$ value depends on the actual state of the system (which sites are occupied by electrons) and therefore part of formula 1, $e^{-\frac{\Delta\varepsilon_{ij}}{kT}}$, has to be recalculated at each step. However the rest can be precalculated and we do so. In our code we call that part the transition constant. Let the transition constant for transition between sites i and j be TC_{ij} and $TCComp_{ij}$ be it's complement in the sense that $rate_{ij} = TC_{ij} * TCComp_{ij}$

$$TC_{ij} = v_0 e^{-2\frac{d_{ij}}{a}}$$

Then

$$TCComp_{ij} = \begin{cases} e^{-\frac{\Delta\varepsilon_{ij}}{kT}} & if \Delta\varepsilon_{ij} > 0 \\ 1 & if \Delta\varepsilon_{ij} <= 0 \end{cases} \quad (2)$$

The idea of pruning is to not look at pairs ij for which TC_{ij} is smaller than some threshold t. The hope is that at some point the precalculated value TC_{ij} has enough of an effect that in all relevent cases the actual jump from i to j is unlikely enough that we may justify ignoring it.

As you decrease the t value, you prune less transitions but increase in accuracy. I tried multiple values for t, and in the end results I demonstrated 3 such values, where you can see the accuracy increasing as you decrease t, but also the time consumption increasing.

The motivation for this idea comes from trying to improve performance, and I had no way of predicting what is the impact on accuracy for this approach. However I did test its performance against the initial unoptimized approach.

3.5 Validation and performance measurement

As I try different approaches, a very important question is how much time saving that actually provides me, and whether the solution is still correct. In the cases of pruning and probability occupation I also want to know how much do I lose in accuracy. When measuring these things I assume that the implementation in Python is correct and I also compare the other solutions with it.

First, I have to define test sets. I use 2 different test sets:

- 100 random dopant placements of 30 dopant networks with randomized electrode voltages in a -2V and 2V range. I call this test set RND.
- 100 tests based on 25 XOR search results, produced during the course of the Thesis. That means there are 25 different random placements, and 25 distinct control voltage values for all of those placements. But for each of them we change the input voltage values (see chapter 6.1) to generate 4 test cases per search result, hence total of 100 test cases. I call this set XOR. Note that these were XOR search results, in some of the cases a proper XOR gate was not found!

All the tests use the parameter values $\frac{a}{R} = 0.25$, $\frac{I_0}{kT} = 100$ as discussed in 2.5.

Now I would like to get the results for each of the test cases using the Python solution and then getting the same thing in the other solutions so I could compare them. However there are a few things to consider when deciding how to compare them.

The biggest problem is that the results can have quite a lot of noise. The level of noise may even change between test cases.

Secondly, the range of current output varies a lot, around 3 orders of magnitude (base 10). This means that simply comparing absolute results is not going

to provide us with much useful information. At the same time, if we compared the results on percentage basis, meaning given baseline current C_B and another current for the same test arrived using one of the other approaches C_{alt}, then mistake would be $|C_B - C_{alt}|/|C_B|$. This approach has however the problem, that solutions close to 0 have naturally very high variance, so we would see huge mistake values even if the actual behaviour is completely normal.

So the goal is to get a mistake in a normalized manner, also taking into account the noise. For that I decided to firstly, for each test run the simulation 5 times. For the baseline I ran it 5 times 1E6 hops. Secondly I decided to compare the results by using Bhattacharyya distance (cf. [1]). I used this to find the distance between two gaussian distributions. I got the gaussian distributions by calculating the mean and standard deviation based on the 5 simulations' current results and assumed that the noise follows normal distribution.

So given a test case t, which is some kind of dopant placement and electrode voltage combination, we can run simulations in the baseline solution b and get some current values on all electrodes. For all electrodes I calculated the mean and standard deviation over 5 simulations (for currents). So given any electrode e in test case t, I have mean current Ec_{bte} (E for mean, c for current, b for baseline, t for test case and e for electrode) and standard deviation σ_{bte}. For some other solution s we have similarly Ec_{ste} and σ_{ste}. The Bhattacharyya distance can be calculated using following formula (assuming normal distribution for current values):

$$D_B(b,s,t,e) = \frac{1}{4} ln(\frac{1}{4}(\frac{\sigma_{bte}^2}{\sigma_{ste}^2} + \frac{\sigma_{ste}^2}{\sigma_{bte}^2} + 2)) + \frac{1}{4}(\frac{(Ec_{bte} - Ec_{ste})^2}{\sigma_{bte}^2 + \sigma_{ste}^2}$$

D_B score has values in the range $[0, +\infty]$. When D_B score is 0, that means the two Gaussian distributions are identical. In reality because of the noise we will never get that, but in general I see as D_B score of less than 1 and even around that as good, and as it grows larger it becomes more and more useless. To get a better idea about D_B possible values and corresponding meanings I compiled a few example bell curves to be compared in figure 12.

Using all of this I did the following to measure the quality and speed of different solutions.

- Generate mean and standard deviation for each test case by simulating it using existing Python solution for 1E6 hops 5 times.
- Generate mean and standard deviation for each test case for each other solution by again running it 5 times.
- Calculate D_B score for each test case and for each other solution s by providing the baseline and s's mean and standard deviations.
- For each solution I calculated the average D_B score for both test sets, and also it's standard deviation. See table 2

- For each test set I drew all the solution performances on a graph (except the probability occupation, drawing them out would have been meaningless considering their extremely large error). See figures 13 and 14
- I also measured how long each solution took time in performing the simulations for both test sets separately. See table 2

The different solutions I tested were:

- Go Kinetic Monte Carlo solution with only site energy optimization. 1E6 hops.
- Go Kinetic Monte Carlo solution with recording transition rates and reusing them. 1E6 hops and 5E6 hops.
- Go Kinetic Monte Carlo solution with pruning. Pruning thresholds were at 1E-5, 1E-7 and 1E-9. These were chosen by trial and error to show around were the sweet spot to achieving any reasonable accuracy is. Again, 1E6 hops were used.
- Probability occupation with 2000, 5000, 25000 and 100000 hops. This range is selected to see when does it converge and to show that at some point we cannot expect better accuracy.

Table 2: The results of measuring different solutions on 2 test sets RND and XOR and comparing them to baseline python solution with 1 million hops. Note that the time given was for running all the simulations in corresponding test set. Both sets had 100 tests. that means the times are about running 100 simulations.

setup	RND time	XOR time	RND E_{DB}	RND σ_{DB}	XOR E_{DB}	XOR σ_{DB}
Python 1E6 hops	11881s	12231s	0.15	0.19	0.32	1.3
Go 1E6 hops	12394s	13343s	0.18	0.46	0.36	1.1
Go recording 1E6 hops	139s	85s	0.21	0.61	0.36	1.1
Go recording 5E6 hops	494s	377s	0.27	0.22	0.47	1.1
Go pruning 1E6 hops 1E-5 threshold	2879s	2977s	117	750	56	227
Go pruning 1E6 hops 1E-7 threshold	4291s	4373s	0.52	6.9	0.40	1.2
Go pruning 1E6 hops 1E-9 threshold	5898s	6115s	0.2	0.54	0.37	1.1
Probability occupation 2K hops	47s	46s	4.1E6	1E8	2.3E10	4.7E11
Probability occupation 5K hops	114s	118s	4.3E7	1.1E9	3E8	5.1E9
Probability occupation 25K hops	585s	576s	1.9E8	5.1E9	2.9E10	5.9E11
Probability occupation 100K hops	2291s	2319s	8.2E6	1.7E8	4.6E10	9.2E11

Based on the these results I could validate that the newly implemented solutions most likely work as intended, with the exception of Probability occupation, for which I fairly confident that it works as intended, but is just a inherently bad idea. Now that I have a very fast simulation solution using recording of states, I can easily get more accurate results by having 5 million hops as baseline.

Table 3: The results of measuring different solutions on 2 test sets RND and XOR and comparing them to a baseline of 5 million hops using Go simulation with recording of states.

setup	RND time	XOR time	RND E_{DB}	RND σ_{DB}	XOR E_{DB}	XOR σ_{DB}
Python 1E6 hops	11929s	12058s	0.3	0.4	0.49	1.2
Go 1E6 hops	13176s	14084s	0.46	4.3	2.5	44
Go recording 1E6 hops	138s	81s	0.34	0.73	3.7	53
Go recording 5E6 hops	484s	352s	0.17	0.19	0.2	0.67
Go pruning 1E6 hops 1E-5 threshold	3668s	3522s	340	2200	290	1300
Go pruning 1E6 hops 1E-7 threshold	5077s	5035s	2	25	2.2	24
Go pruning 1E6 hops 1E-9 threshold	6651s	6800s	0.4	1.5	3.1	62

3.5.1 Conclusion

To conclude there are some interesting results.

- The Go baseline solution does not perform faster compared to the Python solution. It could be due to Python making the library call, or due to the compiled Python solution Numba is actually very good.
- With these parameters the best performer is clearly using Go simulation with recording of transition rates as it is around 100 times faster.
- It may look from the first set of results that 5E6 hops performs worse, but remember, when calculating D_B score I am measuring the distance between 2 standard distributions. As we do more hops I expect the results to have less variance, which means even if they have similar means, since the standard deviation is going to be lower for 5E6 hops, then those distributions are also going to be further apart from each other. To understand this more intuitively look at the second row of bell curves in figure 12. This is the reason I decided to make a second set of results where I took 5E6 hops using Go recording solution as baseline.
- Another interesting result is that in almost all cases XOR test cases resulted in worse results than RND test case. That indicates that there is much more noise when trying to find XOR gates, and could be because setups which involve negative differential resistance are more noisy.
- One of the weird things was that occasionally Go 1E6 performed much worse than Python 1E6, and it performed consistently at least a little bit worse. It is possible that it is due to a few extreme cases, which dramatically increased the average and standard deviation. Other possibilities however include the potential inaccuracies of the calculations that rise from using 32 bit floats in Go, while in python the float size is not limited specifically.
- Finally we can see from the results that the probability occupation performed very bad, much worse than pruning. That means this is a dead end for now. Weather or not it converges within 1E5 hops is still inconclusive,

I would say it does, but there is no point in testing it with higher number of hops because it is unlikely to improve in accuracy and for this solution to be useful we would have to find usability at around 1E4 hops, otherwise it will not outperform simple reusing of rates speed wise.

3.6 Parallelism

One of the reasons for choosing Go as the language to do the simulation improvements in, was the promise of easy parallelism using Go routines. Go routines promise less overhead compared to traditional threads and all the thread handling is done in the background by Go. However in my work I did not manage to get a performance boost by using Go routines.

Before continuing I would like to explain 2 very important terms used in Go language:

- Go routine - a lightweight thread. Go routines run in parallel. Go starts actual threads in background and manages assigning Go routines for these threads in the background.
- channel - channels are used to read and write shared values.They have a few important properties. Whenever some Go routine writes to a channel, no one can write to that channel until someone has read from it. After some Go routine reads from a channel, noone else can read the same thing again.

I ended up trying 3 different solutions for parallelisation with the goal being able to run simulations or searches on a cluster.

- Start and check solution - The idea behind this solution is to start simulations from Python by calling some function in Go library and later calling another function that reads the results after the Go routine has finished.
- Start simulations in a bulk - The idea behind this solution is to send all the simulations data to the Go library that we want to start at once. Then Go starts simulations in Go routines in parallel and once everything is done, returns the results for everything.
- Start separate scripts - Instead of using Go routines, we just start separate Python scripts that perform separate and independent searches.

The use case for first two solutions was to start multiple simulations during the same generation in a genetic search algorithm.

3.6.1 Start and check

The idea here is to start multiple simulations from Python 1-by-1. Starting the simulations would be very similar to starting the none parallel simulation from coding perspective. However you do not receive the results immediately, but instead you receive a key to some channel. You can use this key to request the

results from that channel, which will be provided to you once the corresponding simulation finishes.

The idea is that you first start some number of simulations, and then you retrieve the results for all of them at once. When you call the retrieving function, you provide it the channel key, and that key is used to attempt to read from the channel. Once it manages to read from the channel we know that Go routine has finished the simulation.

In theory that solution would have been really nice, however in practice there was one huge problem. The current solution for calling Go functions from Python did not support Go routines. To be more exact whenever the function returned to Python, all the memory addresses used were unlocked for further use. That meant that once I made a second start Go routine call from Python, when the first Go routine was still running, I would get a memory error and the whole thing would crash. This was because the second Go routine started using the same memory locations as the first one.

3.6.2 Start simulations in bulk

The second solution was designed to address the problem in the first one. The limitation of the Python Go binding is that we can have only one Go function called from Python running at any given time. That function may however start multiple Go routines inside Go, as long as it doesn't terminate before all the Go routines terminate.

So the idea was that I pack all the data necessary for making multiple solutions, send it to go, unpack it and then run multiple solutions in parallel.

Technically it worked, but the problem with this solution was that I did not see any performance increase compared to not running anything in parallel, even though I did experience higher use of CPU (meaning it did use multiple cores).

3.6.3 Start separate scripts

Finally I considered what are the practical cases for using a cluster, and realized that since a single search could be run within a reasonable time (up to 1 hour), I may be satisfied with just running multiple searches in parallel. Since searches don't need to share data between each other, I could just start multiple Python scripts in parallel. This is exactly what I ended up doing on a cluster using SLURM, and that finally did provide me with substantial speed boost. For example I completed 600 searches within the span of 48 hours, which on my laptop would have taken around 300 - 600 hours. I used 2 nodes with 20 cores each, and I used a regime of 50 % efficiency, which was required since I wanted to use them for more than 8 hours (cluster configurations).

3.6.4 Conclusion

To conclude Go parallelism provided me of little use. It is possible that this is due to the binding of Python with Go and the performance could be better if

I used purely Go solution. However the latter is not practical. In the end the simplest solution proved to be the most useful one, even though it certainly has some limitations as well (smallest parallelizable unit is a single search, not a single simulation).

4 Genetic algorithm

To find boolean gates or any desired behaviour in silicon dopant networks, NanoElectronics group uses Genetic algorithm, to fine tune the voltage values of the control electrodes. In general having a search solution is very important in order to perform most simulated experiments on the physical model.

Quoting wikipedia (cf. [3]) "In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection."

In my thesis GA is used to solve a search problem. Originally I intended to search for dopant placement positions that would give me similar behaviour to real life test data, however during the course of the work, it became obvious that this is currently not very useful, because they can use machine learning to manufacture models that are already able to predict physical device behaviour. Currently GA in my work is used to find boolean logic in simulated dopant networks. Because of this legacy however, my GA solution is more generic and it is possible to adapt it to different search problems with minimal effort. In addition that has a few extra features, uniqueness and disparity, which I will discuss in the following chapters.

Let's first introduce basic but very important terms related to GA.

- Individual - this is used to refer to a single solution proposal, which may be a good or a bad solution for a given search problem.
- Population - a set of individuals.
- Fitness function, or error function - this is a function that is able to evaluate an individual. We want to find individuals which either high fitness or low error. In my thesis I use error functions.
- Initial generation - As we start GA process, we generate N_G random individuals, where N_G is given as an input to the algorithm. N_G therefore becomes the population size, as we intend to keep population size constant throughout the process.
- Generation - Every generation after initial generation is generated based on previous generation. Afterwards the individuals from the previous generation are forgotten. The generation may potentially use various crossover functions, mutation functions and various other techniques to arrive to a new generation. Each generation is numbered. So we could say something like "We found our first good solution in generation 6".
- Genes - The relevant properties of the individual with regards to the solution are decoded as a gene. For example I use a array of 16bit integers

for that. This is nice as you can separate operations on genes such as cross-over and mutation from the actual meaning of the properties.

- Crossover function - Given 2 parent individuals and their Genes we can generate a new Gene sequence for a child Individual using some crossover function. Some popular crossover functions are single point crossover, two point crossover and uniform crossover.
- Mutation function - Given an individual's gene, performs a random but minimal change on the gene.
- Elitism - To avoid losing best found results genetic algorithms sometimes remember X number of best candidates as they are, and transfer them to the next generation without crossover, mutation or any other alteration.

4.1 Technical details

As mentioned before, I implemented a generic GA solution, that can be used for searching boolean logic gates for simulated dopant networks as well as searching dopant placements. The core functionality is in a dn_search class, which has an entry point to the GA function, and all the helper functions. Also the legacy dopant position search is implemented in that class.

My generic GA solution does have one assumption, that the individuals are somewhat based on the dopant network object. This is a class object, which has all the necessary information about the dopant network, in order to perform simulations on it (positions of acceptors and donors, positions of electrodes, electrode voltages, physical constants etc.).

In order to use this as a boolean logic finder I implemented a new class voltage_seach which inherits from dn_search. In order to get GA working I had to reimplement following functionality:

- Error function
- Random dopant network - In order to initialize the first generation, GA needs a way to generate random individiuals. For example when finding boolean logic, the difference between individuals is only in the control electrode voltage values. The positions of the dopants are same for everyone. This means that this function returns a new individual with random control electrode voltage values which are within given voltage range. This function gets a base dopant network as an input.
- Get genes from dopant network - Since a lot of generic functionality (crossover functions, mutations) relies on having a uniform structure for genes, we have to implement a way to get gene values from an individual based on the relevant functionality. For example for boolean logic, we only take into account control electrode voltages when generating genes values.
- Get dopant network from genes - This does exactly the opposite from the previous function.

- Copy data from one dopant network to another - Since we assume individuals having most things in common, we can only copy electrode values and genes.
- Parallel simulation - This has to only be implemented if you want to evaluate individuals in parallel. However so far I haven't managed to see any performance improvements so far, so this is not recommended. (see chapter 3.6).

As you can see in order to use the genetic algorithm you have to reimplement quite a minimal number of of functionality.

To use the genetic algorithm you have to create a class object, which inherits dn_search such as voltage_search, and provide it with relevant information. This object must also define either manually or through input following things:

- initial dopant network - used as a basis for making the initial population. For example in boolean logic search we require the initial dn in order to know the positions of the dopants, and individuals in the search only differ in their control electrode voltage values.
- parallelism - weather or not parallelism is used when evaluation individuals in each generation. I suggest to set it to 0, as so far I haven't managed to gain any performance boost by using multiple cores.
- error function - this has to be set, as the current design allows for multiple error functions in a class, so to control which one of them is used, class object has a value which stores the right one. For example in voltage_search there are 3 error functions: one that only uses separation, one that uses separation and correlation and one that uses results from parallel simulations.
- simulation strategies - My generation algorithm allows to define list of strategies. When starting out the first strategy is used, and after we have reached a certain threshold for error we can change to another strategy. You can also use this to terminate the search when we find a good enough solution. A strategy includes list of arguments to be passed to the simulation. This has to include the number of hops, but may also include other arguments that is possible to be passed to simulations, such as the simulation method used.

After you have the search object you can use it to initiate the GA search. For that it is possible to pass a lot of parameters:

- Generation size - number of individuals in the population
- Available time in seconds - After each generation, if we have spent more than this amount of time on the search, we just terminate it.
- Disparity - usually set to 2. In short, the higher the disparity the more children high fitness or low error individuals will have. I will explain it in more details (see 4.3)

- Uniqueness - This value enforces the population to comprise of unique individuals. The higher the uniqueness value, the more different individuals have to be from each other. This is good, since evaluating equal or too similar individuals will result in just wasting computing power.
- Cross-over function - Which cross-over function to use to generate children from 2 parent individuals.
- Mutation power - The higher the number the more likely it is that higher bit values will change in the mutations. That means that a single mutation can have a more drastic effect.
- Unique schedule - It is possible, but one does not have to use that, to specify the schedule by which the uniqueness value changes. For example it is possible that one wants to start out with a large uniqueness value, and lower it gradually similarly to simulated annealing.
- Max generations - Similarly to max available time, if we have reached the maximum number of generations, the search terminates and returns the best result so far.
- Mutation rate - the probability of a child to mutate before considering uniqueness. In practice I either use mutation rate or uniquness but not both.

4.1.1 Gene encoding

As mentioned previously genes are encoded into a list of 16 bit integers. To encode for example 5 control electrode voltages (in all cases in this Thesis I used 5, but this is not hard coded, and it is possible to have different set ups) I use 5 16-bit integers, each of which represent one voltage value. To take full advantage of the range that 16 bits provide (65 536), I provide voltage_search with a voltage range V_r, so that given voltage value V_i for i-th control electrode, I can get the value of i-th gene $G_i = (V_i + V_r)/2/V_r * 65535$. This assumes that $|V_i| \leq V_r$, meaning that voltage values are within the desired range.

Later we can retrieve the new voltage values (after cross-over and mutations) like so: $V_i = G_i/65535 * 2 * V_r - V_r$.

4.2 Uniqueness feature

One problem GA in general faces, is how to maintain diversity. This is important both to avoid spending resources on evaluating identical results, but also to stay out of local optimum solutions. One way to solve this is by guaranteeing uniqueness (cf. [6]). This means that we have a distance function, which, given 2 individuals we provide a number value of how different they are from each other. For instance [6] uses Hamming distance, which counts the number of bits that do not match.

In my thesis, however, I use a different distance function. Given 2 individuals a and b and their gene values G_{ai} and G_{bi} where i is from 1 to number of control electrodes N_{ce} I can calculate the distance between them as $D_{ab} = \sum_{i=1}^{i<=N_{ce}} |G_{ai} - G_{bo}|$.

I would argue that in this case it is ok to use such distance function compared to counting number of bit differences, because each gene value is linearly related to the underlying voltage. And since we know that the system behaves mostly continuously, we know that depending on which bit you change, you will have much more drastic effects on the behaviour of the systems (higher bits effect more).

Given this distance function we can now talk about uniqueness. Given a set of individuals P and a new individual I, we can say that the uniqueness score of I compared to the individuals in P is $U_I = max(D_{Ib}$ for $b \in P)$. I use this concept in my GA as I add individuals to the new generation's population. I always compare the new individual I to the already added individuals, and if $U_I < U_R$, where U_R is the required uniqueness provided as an input to the GA function call, I mutate individual I until I get that $U_I \geq U_R$. One thing to keep in mind is to not have so large of an U_R that it could be impossible or unlikely to find a compatible individual through mutation.

So is uniqueness useful? To test that I compared using uniqueness without additional mutation rate to not using it and instead having a certain mutation rate. To compare results I generated 30 dopant simulated networks and tried to find XOR gate for them using my GA. For that I had 3 different setups. All of them shared these parameters:

- population size: 100
- max generations: 10
- hops per simulation: 1E6

The uniqueness of these setups were in U_R and mutation rate or R_m values:

- $U_R = 1000$, $R_m = 0$. I call this setup U1000.
- $U_R = 0$, $R_m = 0.05$ (5%). I call this setup M5.
- $U_R = 0$, $R_m = 0.3$ (30%). I call this setup M30.

For each of the setups I tried learning XOR gates for 20 different dopant placements. Results are in table 4.

After running these tests and rereading [6], I realized I could have reduced the amount of randomness by instead of generating a random placement for each search, I could generate 20 random placements and an initial population of size 100 for each of them, and use these as a starting point for each of the setups to have a more fair comparison.

This is exactly what I did to generate the second set of data. I also added a fourth setup named U5kto1k, which uses the uniqueness schedule feature. It

Table 4: Results for trying to find XOR gates using different parameter values for GA

setup	successes	failures
U1000	10	10
M5	5	15
M30	8	12

starts with uniqueness value of 5000, and reduces it to 1000 within half an hour, which is approximately how long it takes to go through 10 generations.

The results are in table 5.

Table 5: Results of second iteration of uniqueness testing

setup	successes	failures
U1000	8	12
M5	8	12
M30	7	13
U5kto1k	7	13

Since in the last set I had the exact same starting conditions for all the setup variants, I could analyze results in more detail. Even though each of the setups found only 7 or 8 XOR gates on its own, between all of them the number of found XOR gates increased to 12 out of 20. There were some obviously simpler cases as well, since for 5 starting conditions, an XOR gate was found every single time.

As you can see from the results using the uniqueness feature gave similar results as using simple mutation rate, so between the two sets of tests I would conclude that it is probably just fine using only mutation rate. However it is possible that trying different values for uniqueness or having a larger test set would give us different results. Another thing to consider is trying different distance functions.

Table 6: Results of third iteration of uniqueness testing in cluster. The total sample size is now 5 times larger.

setup	successes	failures
U1000	47	53
M5	38	62
M30	46	54
U5kto1k	41	59

Finally I decided to also try using a cluster to run the same experiment but with a larger sample size. The results are in table 6. From here it seems that mutation rate of 5 % and the strategy of using uniqueness schedule seem to perform worse than mutation rate of 30 % and only using single uniqueness value of 1000. This is interesting as it does show the importance of having a large sample size and the benefits of being able to run more simulations.

4.3 Disparity feature

Before I talk about disparity I will first have to explain in more detail how in GA I determine how many children any individual in a generation will have after I have already evaluated their fitness/error.

In [10] the author shows how you can assign a number of children to each individual by ranking them by their fitness and then assigning each of them a number from 2 to 0. The highest ranking individual is assigned 2, second one 1.9 and so on, down to 0 for the 21-st individual. Then each individual is assigned 0, 1 or 2 children based on the full part of their assigned number, and then 1 more if a random number between 0 and 1 is smaller than the leftover part of their number (cf. [10]).

Finally notice that since children have 2 parents, then the number of children we just calculated must actually be multiplied by 2 in order to get the exact number of times that individual is going to be a parent. You could see that number as a number of times that individual is going to be mother (left most parameter for generating new child individual).

Using this method you are very likely to end up with the same number of individuals in the next generation, however this is not guaranteed. In practice I want to have complete control over the population size so that it doesn't go smaller or larger by random chance. I achieved this by carrying over either a penalty or a bonus to the odds for the next individual based on the random result of the previous one. For example if rank 8 individual is assigned number 1.3 and because of luck is assigned 2 children, I carry over -0.7, so that the next, rank 9 individual, instead of having a number 1.2, now has 0.5. If however rank 8 individual would have failed the "dice roll", I would have carried over 0.3, so that rank 9 individuals number is 1.5.

Now about disparity. Suppose you wanted your best individuals to have more children than just 4. For that I have a parameter called disparity. Disparity shows what number is assigned to the highest ranking individual. So in the previous example the disparity would have been 2. If you however increase the disparity then you cannot reduce the numbers for next ranks at a linear pace like in [10], but instead have to use some other function for that, if you want to maintain the number of individuals in a population.

This problem is essentially an integration problem. Consider for example a line from $x = 0, y = 0$ to $x = 1, y = D$, where D is disparity. What we want to achieve is that the area under the line to be 1. If we had such a function $f(x)$, then if we had a population size P, then we could make another function $f_P(x) = f(x/P)$, which basically stretches out $f(x)$ so that $\int_0^P f_P(x) = P$. Which is what we wanted to achieve. Now such $f(x)$, given D is not hard to find. Suppose $f(x) = D * x^{D-1}$. Then its anti derivative would be $F(x) = x^D$. Now if you would integrate from 0 to 1 over $f(X)$ you would get $\int_0^1 f(x) = F(1) - F(0)$, which would always be 1, except when D would be 0, since $0^x = 0$ and $1^x = 1$ no matter the x value.

So, let i be an index from 1 to P and M_i be the number assigned to ranked i individual, which represents approximately the number of times that individual

is going to be a mother. Given disparity D I calculate $M_i = D * ((1 - (i - 0.5)/P)^{D-1}$. The -0.5 is so that the sum of all M_i would be closer to P. I flesh it out by calculating the difference and adjust all the numbers evenly based on the difference.

In practice I've been using $D = 2$, but in principle one could use other D values, since the Disparity feature is implemented as explained above.

4.4 Fitness functions

For control voltage search I've implemented 2 error functions, one which uses only separation and another one which uses both separation and correlation. What such an error function does, is given a list of input electrode values with corresponding expected boolean value, it evaluates how well the given control voltages achieve these desired boolean values by measuring the output current.

The error function performs X separate simulations using the KMC model, where X is the number of test cases (usually 4) and the one of the solutions explained in chapter 3. Each of the simulations' only difference is the input eletrode voltage values, which match the corresponding test cases. As a result of these simulations I get output current values.

To calculate the separation I keep track of the highest false current value (or $false_{max}$) and lowest true current (or $true_{min}$) value. The highest false current is equal to the highest current value from the test cases for which we expected the output to be false. The lowest true current value is equal to the lowest current value from the test cases for which we expected the output to be true. These values can be used to calculate separation $sepa = true_{min} - false_{max}$. If the separation is positive we have achieved separation.

Secondly we can calculate correlation *corr* by correlating the resulting output current values with a correlation list. For example, if we are looking for an XOR gate, where the input list is [(false, false), (true, false), (false, true), (true, true)] and expected outputs are (false, true, true, false) we would generate the following correlation list: [0, 1, 1, 0].

- The error function, that only uses separation will return $-sepa$
- The error function, which uses both separation and correlation is a bit more complicated. In addition it makes use of another parameter called $corr_{pow}$ or correlation power:
  - if separation is negative, we just return $-sepa$
  - If separation is positive, but correlation is negative we return 0, though I believe we should not arrive to this case.
  - Finally if separation is positive and correlation positive, we return $-sepa * corr^{corr_{pow}}$

  The idea of this is that first we find a solution which is able to separate true and false values, and then start to also take correlation into account. With $corr_{pow}$ you can adjust the importance of correlation.

5 Visualization tools

As part of the Thesis I developed several visualization functions, with the aim to make it easier to get an intuitive idea of what is happening in the simulated dopant networks and through that also perhaps get insight what could be happening in the real life samples.

To do that I either developed from scratch or built upon the code of B. De Wilde to get following visual functions:

- Traffic visualization - As we run the simulation we can keep track of how many jumps were made between each pair of dopants or each pair of dopant and electrode. This information can be used to visualize the traffic of electrons jumping between sites.
- Traffic subtraction visualization - if we make slight changes to a dopant network, then it is possible that it is hard to tell the difference between their traffic visualizations. What we can do instead, is subtract the traffic data of one of them from the other and visualize the result similarly as in traffic visualization. This would show us the change in behaviour. However one has to be careful as the change can also be caused by the general noise from one state to the other.
- Traffic and chemical potential combined visualization - B. De Wilde had already developed a chemical potential visualization. I did improve that a little bit and I used the improved version of it to put traffic and chemical potential landscape on a single picture.
- Swipe animation - Often in the project we have wanted to perform swipes. Swipe is when we gradually change the voltage value of one electrode from starting value to end value. By dividing the swipe into a certain number of steps to provide graduality and simulating the network at each of those steps, I could use the results at each step to generate a frame using already existing traffic and chemical potential visualizations and combining them.

5.1 Traffic visualization

Traffic visualization is supposed to highlight pairs of two dopant sites or dopant site and electrode with the highest currents, so that we get the idea which transitions are more likely to occur within some time frame.

The main challenge with visualizing traffic is how to separate the current strength between different sites and from which point you don't visualize it at all to avoid clutter in the picture.

I do that as following: given N sites S, which includes both dopant sites (acceptors in our case) and electrodes. Let P be a set of pairs (A, B), where $A, B \in S$ and $A \neq B$. For each pair $p \in P$ I know the net jumps of electrons, called $|e(p)|$. For example if we have sites $A, B \in S$, then if during the simulation we had 30 electrons jump from A to B and 100 electrons from B to A, we know

the net difference of 70, so $e((A,B)) = -70$ and $e((B,A)) = 70$. I know that the time frame for all of those net differences is the same, so I can compare them with each other as if I compared their currents (even though the underlying meaning is different).

I first find a pair p_{max}, which has the highest $e(p)$ value, let that value be $e_{max}(P)$. I use this value to set the base line for clearest and strongest visualizing and I compare all other values with that when deciding if or how strongly I should visualize the current.

To visualize traffic between a pair (A,B) I use arrows to determine the direction of the jumps and alpha channel to show how $p((A,B))$ compared to $e_{max}(P)$. Whenever $p((A,B)) * 100 < e_{max}(P)$ I do not visualize the traffic between that pair. Otherwise I set the alpha value to $alpha = \sqrt{p((A,B))/e_{max}(P)}$. Note that alpha value is between 0 and 1, where in the case of 1, the arrows are completely clear, and at 0 it is completely faded out.

Another feature that I visualize in the traffic visualization is the probability that any site holds a dopant. If it is close to 100 % then it is completely white. If it is close to 0 %, then it is completely black and otherwise it is somewhere in between. The color changes linearly with the percentage value. The necessary information was provided in the simulation if you set simulation parameter "record=True".

5.2 Chemical potential visualization

Dn object has a function, which is able to evaluate chemical potential caused by the electrode voltages at each point. Differences in chemical potential between two points show which direction the current takes. Current goes from negative to positive chemical potential. Funcion to calculate this value was already provided by the work done by B. De Wilde (cf. [11]). Using that he already had a chemical potential visualization. The only original part that I changed regarding that was to instead of having gradually changing color to visualize the change in chemical potential I divided it to discrete steps. For example, I had 10 discrete steps with easy to distinct color differences. This was useful so that it would be easier to spot shapes in the chemical potential landscape.

To compare previous and updated versions see figures 5 and 6.

5.3 Swipe animation

Final visual functionality that I created is a swipe animation. A swipe is when we change one electrode voltage from some value to another gradually and leave the rest of the electrode voltages unchanged. In practice I used this to visualize XOR or other gate behaviours as I changed the inputs between the 4 states.

The animation has 2 parts. Firstly a visualization of the dopant network which combines traffic visualization and chemical potential visualization. Secondly a graph that displays output current parallel with the animation.

To generate the animation I first run a simulation on the dopant network for each frame. This is necessary to generate the data that is going to be used

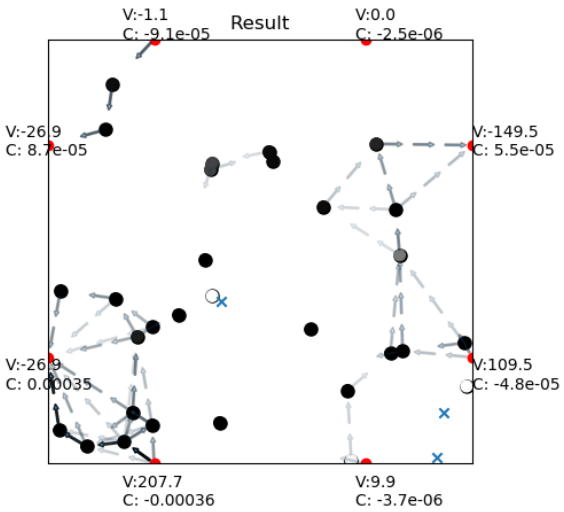

Figure 4: Example of traffic visualization. The black and white dots are the acceptor sites (dopants), x crosses are donor sites, which only interact electro statically so they are not in set S. As you can see, some acceptors are white, meaning they are very likely to hold an electron and thus have negative charge at any given time. There are arrows between all sites which have relatively large interaction between them. It doesn't mean that electrons don't jump between the other pairs, it just means they do so much more rarely or with less direction than the ones we have visualized. Finally the red dots at the edges are the 8 electrodes.

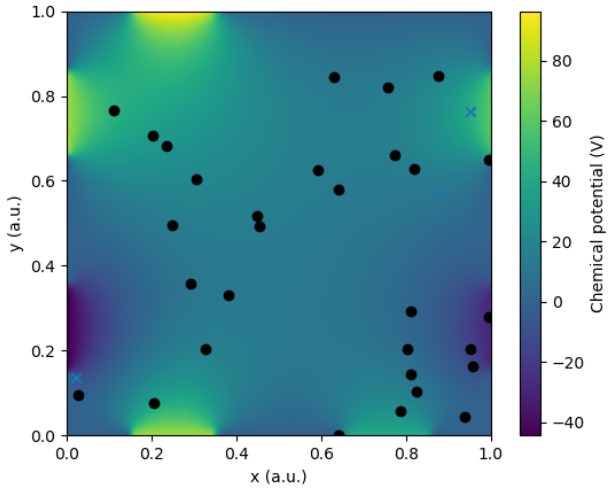

Figure 5: An example of chemical potential visualization previously. Black dots are the dopant sites, while corsses being donor sites. The color shows chemical potential at each location, which changes from blue (small) to green(medium) to yellow (high).

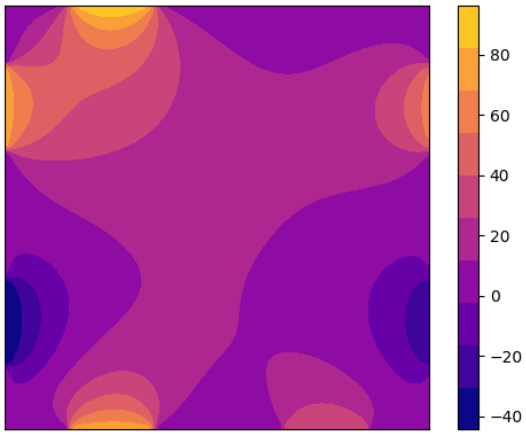

Figure 6: Example of chemical potential using discrete steps. As you can see some of the shapes come out much better than previously. The underlying dopant network is the same. In this figure I left out the dopant sites. Yellow areas have high chemical potential while dark purple areas the smallest.

in the animation. The exact number of frames can be controlled by parameters. As I visualize a XOR gate for example, I specify how many frames does each of the swipe take place, but also how many frames do I want to just hold still after the swipe. The latter is nice, because I can easily visualize the noise to have a better idea about the quality of a given solution.

The data required from the visualization is the following (this lists the items required by each frame):

- simulated time
- electrode voltages used for the simulation
- currents at the end of electrodes
- traffic information - this is same as used by traffic visualization
- expected output true value. This is in the range of 0 to 1. During the gradual transition it can be a value between that. It is used to draw an expected output line, which makes tracking simulated gate quality easier.

After I have all the data for all the frames, I have to find the maximum current, which is important for the traffic part of visualization. When traffic visualization on a single network calculated $e_{max}(P)$, then now this is no longer useful. Firstly between all the simulations for the frames, the time frames may no longer be the same even though we do simulate the same number of hops (see chapter 2.3). That means we have to instead find the maximum current between all pairs between all frames.

Current can be calculated easily: $c(p) = e(p)/time_f$, where p is a pair in P_f and f is some frame.

The maximum current is simply the highest $c(p)$ value between all frames. Let that be $C(P_{all})$. Now as I generate frame f, I pass $C(P_{all})$ to the traffic visualization along with $time_f$, then I replace $e_{max}(P)$ with $C(P_{all}) * time_f$ and rest of the traffic visualization behaves similarly.

Similar approach has to be taken with regards to chemical potential. I don't want the minimum and maximum to be calculated per frame, but instead get the minimum and maximum between all the frames and use that as a baseline so that color values would stay consistent.

For example see figure 7.

5.4 Validation / Testimonials

As part of the validation for this work I asked other members from the Nanoelectronics group to provide me with testimonials and feedback for the visualizations. Firstly the positive testimonials:

- Prof.dr. Peter A. Bobbert - "This is the first time, I've seen something like this".

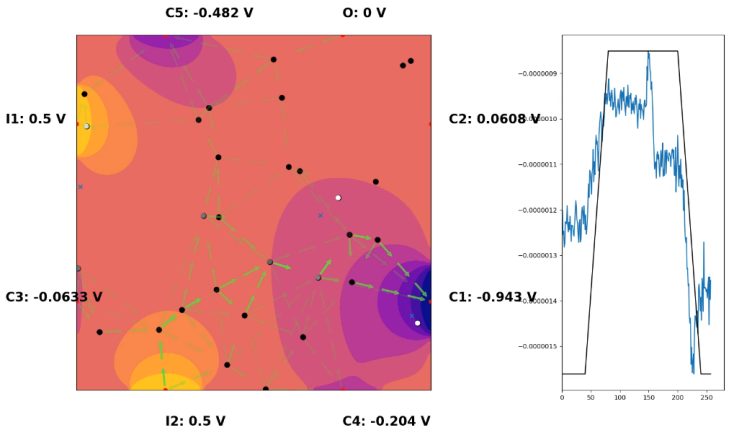

Figure 7: One frame in the swipe animation. The left side is the combination of traffic visualization on top of chemical potential landscape. On the right you see the current of the output electrode. This frame is at the end of the animation, as you can see the output current has already been mostly drawn out. I1 and I2 are inputs, O is output, C1-5 are control electrodes. To separate traffic arrows from the chemical potential, they are drawn green. Black line shows the expected output (this is an example of where we expect a XOR gate), and the blue line is the actual simulated current in output electrode. x axis is time during the animation, and y axis is the current.

- Prof.dr.ir W. G. van der Wiel - "This is going to be used in all the future presentations."
- Bram Van de Ven - "A nice tool to check the hypothesized inner workings of the devices".

During the feedback a few concerns were also raised:

- Requires a bit more polishing. For example some of the problems were a lack of legend, which might be useful for new users, and the voltage text overlapped with the graphics.
- How is this going to be useful.

Regarding polishing I've already addressed the text overlap. However regarding usability, that question is hard for me to answer. From the testimonials you can see that at least someone found some use for these visualizations, but overall I hope that over time others will also find questions to ask from the visualizations. Also it could be a way to get some sanity checks for extreme findings. For example throughout my own work, visualizations helped me to find bugs.

5.5 Conclusion

So to sum things up, visualizations are used as supplementary tool to give sanity checks while performing simulated experiments. However the resulting visualizations received generally positive feedback, and could be used in the future to also present the work done by the group more generally.

6 Finding XOR gate with varied number of dopants

So far the NanoElectronics team has worked with fixed sized silicon dopant networks.

Now it would be interesting to know if having more or less dopants would still retain the desired properties - negative differential resistance, none-linearity, ability to perform boolean operations.

We can attempt to predict that by using the simulations. However to test that we have two options, we could assume the same density, meaning that having less dopants means a smaller device, or we could assume that we can control the density, and we would instead keep the size of the device constant. In this experiment I chose the first option.

At the beginning of the thesis, the estimation I received of the number of dopants on the current networks was around 30-40. Based on the new calculations however it could range between 100 (see Chapter 2.1). Now the problem with such large values is, that KMC is going to be too slow to help make any sort of analysis for such networks. There are several reasons for that:

- calculating the rates at each step, where N is number of dopants in the system, has complexity of $O(N^2)$.
- As we increase the number of dopants in the network, the number of possible states also increases, it does so exponentially. However in practicality, with the current parameters the increase of unique states that we see in practice, does increase more like linearly. Storing these unique states however again takes $U * N^2$ space, where U is the number of unique states.
- It is very likely that as we increase the number of dopants, we also need to do more hops to achieve the same level of accuracy as previously.

In addition, since currently we are storing unique states' keys as 64 bit integers, I would need to rework this part to facilitate larger than 64 size systems.

That said we can still test the performance of simulated networks using a wider range of dopants, which can still tell us something about the size of the range where we could still see desired behaviour. For that we chose N values: 5, 10, 20, 30, 45 and 60. For each of those values of N, where N is the number of dopants, I generated 14 different random placements and by using genetic algorithm I tried to find them XOR gates. The parameters for this setup were $a_R = 0.25$ and $I_0 = 100kT$. These parameter values were chosen, because so far, using 30 dopant networks in the simulations, Bram de Wilde managed to observe the most similarities between the simulated network's and physical devices' behaviours (cf. [11]).

It is possible, but not confirmed that since simulated 30 dopant networks behave similarly to the physical devices, then lowering or increasing the number of dopants would also have similar effects for physical devices. Notice that since we have no control over the density of dopants in practice, then lowering and

increasing the number of dopants is done by decreasing or increasing the area while not changing the density.

The genetic algorithm uses simulation solution written in Go, which took advantage of recording of rates whenever we reach a new unique state (see chapter 3.2.2). I also tried to make use of parallelism but I failed to find any gains in time savings so far.

6.1 Finding XOR gates, problem definition

Finding XOR gate for a dopant network means that given X electrode positions relative to the dopant network, from which 2 are for input, 1 is for output and $X - 3$ are for control, we find voltage values for all the control electrodes, so that the dopant network is able to perform as an XOR gate. XOR means, that if exactly 1 of the inputs is true, the output has to be true. In both other cases (inputs both true or both false) the output is false.

Since we are dealing with electrode voltages here, then we have to translate these to true and false. In this setup true means having 0.5 V applied to the input electron and false means having 0 V applied to the input electron. For the output however, since it is not feasable to find a specific output value, what we are looking for instead are three things. Firstly, that you can separate the true values from the false values, meaning that the current values are distinct enough to be separable even after taking noise into account. Secondly, that the current for the true values is more positive that the current for the false values. Finally we also want all true values to be close to each other, and same for false values. The latter is achieved by using correlation.

To achieve all these goals I use an error function which uses both separation and correlation explained in 4.4. I aim to minimize this error function using Genetic algorithm.

The reason for trying to also maximize separation is that solutions with higher separation are more robust to noise and other potential system effects, which means that in practice you can determine their true false value faster. This is due to the fact that better signal to noise ratio allows us to have a larger bandwidth in practice. This is because increasing the bandwidth increases the noise, but having a better signal to noise ratio initially allows us to overcome that (source: personal communication with Dr. T. Chen).

6.2 Using genetic algorithm to find XOR gates

The genetic algorithm used is the one developed as part of this thesis (see Chapter 4) The total number of searches I had to perform was 84 (6 N values, 14 placements for all those values). To make the results fair, same settings were used for all the N values.

The genes in this case are the control voltage values. In this case we used 5 control voltages in all cases. (see figure 8). That meant, that a solution effectively consisted of 5 numbers, which we fit into 5 16 bit integers.

The Genetic search parameter values were:

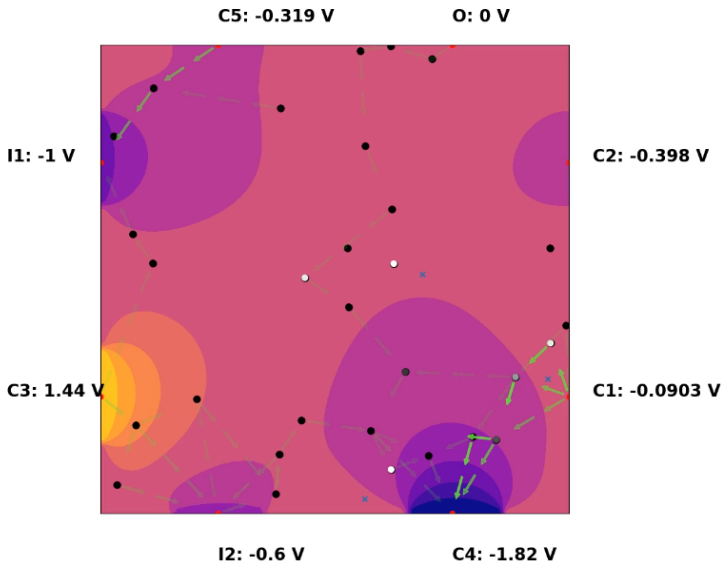

Figure 8: This shows the layout used in the simulated experiments. All red dots at the edges are the locations of electrodes. I1 and I2 are the inputs. C1 to C5 are the control voltages for which we aim to search voltage values. O is the output electrode. In between the electrodes dopants are randomly placed for each sample. In this sample there are 30 dopants and 3 donor sites (marked by x).

- Population size: 100
- Disparity: 2, this makes it effectively behave like in the example in external source [**6**]
- Maximum number of generations to run: 10
- Uniqueness value: 1000.
- Number of best solutions kept unchanged: 4 (Elitism)
- crossover function: single point crossover.
- Mutation is only done to maintain uniqueness.
- Each simulation is 1E6 hops.
- voltage range for control voltages: -1V to 1V

There are 2 main outputs for the genetic algorithm:

- Error score for the best configuration
- Dopant network object, which control electrodes have the learnt voltages.

6.3 Results

As a result of genetic algorithm we get an object which has the original placement of dopants but learnt voltages for the control electrodes. As we also receive the error score we can already guess how well that configuration will perform as a XOR gate. However since for calculating the error we only run each test cases once and only for 1E6 hops (which often is enough), it is possible that the error score is negative (meaning that there is separation) only by accident and in practice the noise of the current when doing more or longer simulations would actually not show any separation. This is especially important when the resulting error score is negative but still close to 0.

To get a more accurate idea about the results, I simulate each input combination (previously test case) for 40 times 5E6 hops. That way I get 40 data points, and by drawing them on a graph I get quite a clear idea about the noise and separation. I also calculate a new separation value, which takes into account all the data points and calculates more accurate $false_{max}$ and $true_{min}$.

After having such graphs I can determine if the solution is correct by seeing if the expected true and false values are separable.

6.4 Interpretation of results

Firstly I would like to note that there was no significant difference in finding XOR gates for N values 5 to 45. The differences of 1-3 successes could easily be caused by chance. But only finding an XOR gate 3 out of 14 times for 60 dopant networks does lead us to believe that it is harder to find a XOR gate as

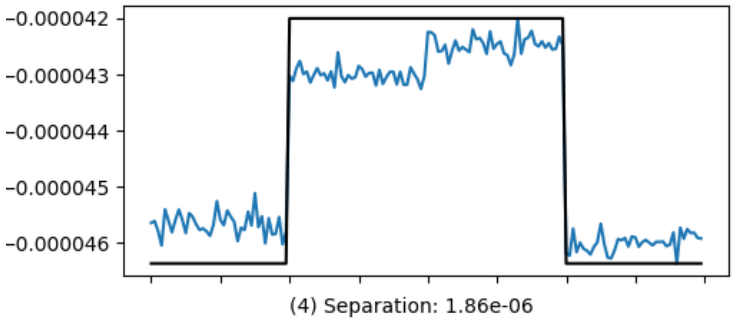

Figure 9: An example of near perfect XOR gate.

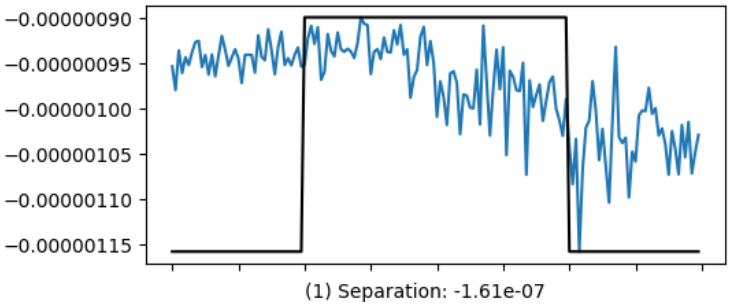

Figure 10: An example of clear failure to find XOR gate.

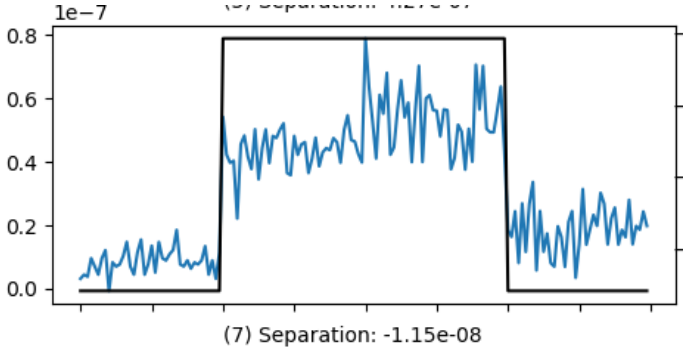

Figure 11: An example of almost finding a XOR gate, but due to the noise the final separation was not positive so I decided to count this example as a failure.

Table 7: Results of finding an XOR gate using genetic algorithm for 5, 10, 20, 30, 45 and 60 dopant networks.

results	5DOP	10DOP	20DOP	30DOP	45DOP	60DOP
success	9	8	7	6	9	3
fail	5	6	7	8	5	11

you increase the number of dopants further. Now this could however also be due to the fact that we use the same voltage ranges for control electrodes and same voltage values for input electrodes, as with fewer dopants. This is important, since it is possible that these values are just too small for 60 dopant networks. This theory is also supported by the fact that we know that increasing these voltage values improves finding the gates in general, which you can see from the results in Chapter 7.3.

Secondly it is important to note that even though we found XOR gates for 5 dopant networks, there are a few considerations one should take into account. In practice there are maximum voltage ranges which we can use without damaging or changing the device. It is possible that by making the device smaller the allowed voltage range also drops, as the total resistance goes down and current goes up (which is the probable cause for device changes). We can see from figure **??** compared to figure **??**, the average current values are much larger for 5 dopant systems compared to 30 dopant systems (current is y axis).

To be able to better read the graphs in the appendix I included 3 examples (see figures 9, 10 and 11), to show a an ideal example, a clear failure and a borderline failure when finding XOR gates.

6.5 Followup simulations on cluster

During the late stages of the Thesis I gained access to a cluster and had worked out a workflow to effectively use parallelism provided by cluster. So I could run the same experiment but instead of just having 14 data points for each number of dopants I now had 100 data points for each number of dopants. In addition since I wanted to confirm the theory that the main reason for low number of dopants to have such high success rate was the relatively high current that the same voltage range provides I also include the averages of currents on all the electrodes for each number of dopants.

Table 8: Results of 100 data points for each number of dopants, which we got from cluster

Number of dopants	success rate	average model current	average supposed physical current
5 DOP	52/100	0.00267	2674.8nA
10 DOP	48/100	0.000673	673.4nA
20 DOP	42/100	0.000219	218.9nA
30 DOP	41/100	7.75e-05	77.5nA
45 DOP	22/100	3.3e-05	33.0nA
60 DOP	9/100	1.49e-05	14.9nA

By looking table 8, there are 2 very important take aways. Firstly by increasing the number of data points the results draw out a much more convincing picture. For example previously I wasn't sure if the number of successful gates would actually be lower for 45 dopant networks compared to 30 dopant networks, while now I am. Secondly I can be fairly certain that the high success rate for lower number of dopants is due to allowing too large currents in the simulation. In practice these devices would most likely break long before reaching these levels of currents, while for networks with higher number of dopants we could most likely safely increase the voltage range.

6.6 Conclusion

To summerize it does indicate that there is probably a large window of picking the size of the device in terms of finding negative differential resistance effects to perform as XOR gate. However these results could be improved if we knew more about the exact relation between the model and the physical device, and if we knew more about the viable voltage ranges as we decrease or increase the size of the network.

So I would recommend to running a new experiment in the future with an adjusted error function, which also takes into account the currents at the end of electrodes, but with an increased voltage range to have more freedom for finding the most optimal but still realistic solution.

Finally using a cluster to increase the number of datapoints is clearly advantageous, clearly demonstrated by the fact that based on the initial results I would have guessed there is not difference between 30 or 45 dopants, while in the second set one could clearly see that there was.

7 Calculating VC dimensions in simulation

Another use case for which I used the improved simulation solution was to explore VC dimensionality of simulated networks. VC stands for Vapnik–Chervonenkis (cf. [9]), and VC dimension is a way to measure complexity of some system. In our case we want to measure the complexity of our simulated networks. We would say that a simulated dopant network has VC dimension N (VC N for short), when having a finite set of points P of size N, the system is able to perform all the possible functions in F such that $f : P \to \{true, false\} \implies f \in F$. It should be clear that there are 2^N different functions. If a system is only able to perform some of those functions, let that number be X, then we say that VC capacity for N points for the system is $\frac{X}{2^N}$.

Let's look at an example. In our case we have 2 input electrodes, that means a single point in P is a combination of voltages on those inputs. So $p \in P \implies p \in \mathbb{R} \times \mathbb{R}$. Let's consider VC 5. Let the 5 points be (1V, 1V), (-1V, 1V), (1V, -1V), (-1V, -1V) and (-2/3V, 0). We can consider each of these functions in F as a boolean gate. One such gate could expect true for the first, second and fifth input point, while false for the third and fourth input point. As with finding XOR gate previously, finding any of the functions in F means finding the right control voltages for the 5 control electrodes, so that the dopant network performs as that function.

As mentioned before there are 2^5 different boolean gates for 5 points. We say that 2 of them are trivial: when we expect always true or always false. We do not explore these trivial cases as we do not need to know the input values to calculate an output for those.

We say that our simulated network has VC dimension of 5 when we are able to perform all the 30 none trivial classifications. As part of my thesis I explored if I am able to find VC dimensionality of 5 (VC 5 for short) for simulated dopant networks similarly to what was found in real life samples. So far for a single real life sample VC 4 and 5 were found and around 90% capacity was measured for 6 points (source: personal communication with dr. H. Ruiz).

In my simulated experiment I try to find VC 4 and 5 for different sized networks and finally for a single 30 dopant network I also measure the capacity for 6 points.

7.1 Setup

As mentioned before when measuring VC dimensionality we first have to define some points. A point is a combination of 2 voltage values, which represent the voltages applied to input electrodes. The points we use are following: (-1V, -1V), (1V, -1V), (-1V, 1V), (1V, 1V), (-1/3V, 0V), (1/3V, 0V). When measuring VC 4 we only use first four points, for 5 the first five and for 6 all of them. Notice that the voltage values here are larger than used in the previous experiment. This has three reasons, firstly we believe higher voltages make finding gates easier, secondly, in real sample VC measurements also higher voltages were used.

Secondly we have the voltage range for the control electrodes. We allow their voltage values to be between -2V and 2V. This was chosen as this is the maximum allowed voltage in real life experiments. However after speaking with dr. T. Chen from the NanoElectronics group, it became clear that this range can vary from one sample to another. The important part is that the overall current going through the device wouldn't go too large.

To find the classifications I used the same genetic algorithm and parameters as in previous chapter, with only two differences (explain in Process). That means to prove that a dopant network is VC 4 I would have to perform 14 searches, for VC 5, another 30 searches and for VC 6 another 62 searches. Since I didn't have my parallel solution ready when performing this simulated experiment, I was fairly limited in the amount of samples I would go through. Since I also wanted to explore VC dimensionality for different sized networks I have a following set of samples:

- 1 sample of 30 dopant network, for which I attempted to find VC 4, 5 and 6.
- 1 sample of 20 dopant network, for which I attempted to find VC 4 and 5.
- 4 samples of 10 dopant networks, for which I attempted to find VC 4 and 5.

7.2 Process

I used the same genetic algorithm and parameter values as in chapter 6.2, except for two differences:

- The voltage range for measuring VC dimensionality was larger: -2 V to 2 V.
- I had no maximum number of generations, instead I just ran it for 1 hour on my laptop, however within 1 hour I did get more than 10 generations, but not more than 20.

I tried to search for each classification within the given 1 hour once, and for all the failed cases I tried searching for a second time, the results after the first and second iteration are presented separately. That means that not finding a classification doesn't mean there doesn't exist one. However we are able to talk about the lower bound of both VC dimensionality of a dopant network, and VC capacity for some dimension.

To evaluate if a found solution was actually a correct solution I used the same method as in chapter 6.3. That meant that for each point I made 40 simulations, and I could find a much more robust $true_{min}$ and $false_{max}$ current values to calculate a separation, that if being positive, I could confidently say that given classification is correct. You can see an example of VC 4, where all the test cases are drawn out for a single 30 dopant network in figure 15. There you can see all the non-trivial classifications (by looking at the black line), and what kind of solutions we found for a single sample of 30 dopant network.

7.3 Results

For results I can count how many classifications did the genetic algorithm manage to find within 1 hour of search. Firstly I am going to display the results based on the first run of searching. Later on I did try to run the search algorithm a second time for all of the missing cases.

Table 9: Results of finding VC dimensionality on 6 different simulated dopant networks after first run of searches. The results are in the form of X/Y, where X means the number of functions in F found and Y is the total number of functions looked for that VC dimension (all the none-trivial were looked).

test	VC 4	VC 5	VC 6
30 DOP, 1	14/14	27/30	59/62
20 DOP, 1	14/14	29/30	-
10 DOP, 1	14/14	30/30	-
10 DOP, 2	14/14	30/30	-
10 DOP, 3	14/14	30/30	-
10 DOP, 4	14/14	27/30	-

As you can see from table 9, I could find VC 4 for all samples and VC 5 for 3 out of 4 of the 10-dopant networks after the first run of searches. After doing a second run of searches for the missing classifications I found 9 out of 10 missing classifications. The only one I didn't find was a classification for VC 6 for 30 dopant network. One explanation for the high success rate in finding missed classifications could be having an unlucky initial population for the genetic algorithm, another simply arriving to a bad local optimum.

From the results it would seem that as you reduce the number of dopants you would find VC 5 easier. However similarly to the previous experiment, it is possible that I should not be using the same voltage ranges for smaller networks. I believe that for smaller dopant networks the allowed dopant range should be smaller. That could be further backed if the current on average increases as you have less dopants (more current means more heat, means more likely that the network will "scramble" and start acting differently).

One of the problems is that it is none trivial to translate the current we get in our model to some real world value. Currently I work under the approximation that $1nA = 10^{-6}$ of the current value in the simulations (source: personal communication with B. de Wilde, cf. [11]).

As you can see from table 10 the four samples of 10 dopant networks all have higher average currents compared to the 30 and 20 dopant networks samples. However the 20 dopant network has much smaller currents compared to 30 dopant network, so that was wierd. However I currently only took into account the average current of the output electrodes. Since I am working with simulated dopant networks I do have access to currents at the end of all electrodes. I show them in table 11 and as you can see from there the current clearly does go up as you lower the number of dopants. Now it could be possible that this difference

Table 10: Average currents for all the 6 samples, over the 14 classifications made for VC 4. The average is only over the output electrodes.

test	average model current	average supposed physical current
30 DOP, 1	3.6e-05	36.0nA
20 DOP, 1	3.77e-06	3.8nA
10 DOP, 1	0.000159	159.1nA
10 DOP, 2	0.000136	135.9nA
10 DOP, 3	0.00073	730.3nA
10 DOP, 4	0.00113	1126.6nA

Table 11: A verage currents for all the 6 samples, over the 14 classifications made for VC 4. The average is over all the electrodes.

test	average model current	average supposed physical current
30 DOP, 1	0.000129	129.5nA
20 DOP, 1	0.000222	222.2nA
10 DOP, 1	0.000502	502.1nA
10 DOP, 2	0.00134	1339.4nA
10 DOP, 3	0.000323	322.8nA
10 DOP, 4	0.000887	886.8nA

in average could be made by a few outliers, but if you compare figures 15 and 16 you can see this is not the case.

7.4 Follow up simulations on cluster

Similarly to the previous experiment (see Chapter 6.5), I used my access to the cluster, which I gained during the late stages of my Thesis to get more data points and see if that would improve the quality of the experiments. I decided to test VC 4 and 5 on 12 different samples: 4 with 30 dopants, 4 with 20 dopants and 4 with 10 dopants. The first notable improvement was that when the first set of results it took me more than a week to generate, then it only took around 3 days on a cluster to get all the new results. The results are in table 12.

This time the cluster results provided little new insight for me. The results are very similar to the previous ones, but do confirm that finding VC 4 and 5 is very likely for a wide range of dopant network sizes. However you can see from the data that there are quite a wide range of average current values even among same sized networks.

One thing I noticed that in for 3 cases, where the number of cases found was significantly lower than average (30 DOP,3 VC 5; 10 DOP, 3, VC 4 and 10 DOP 4, VC 5), the average absolute currents were also lower than other samples with similar size. So I thought that perhaps there is a relation there. But then again we have a counter example of 20 DOP, 3 VC 5, for which we found all gates, but the average current was the lowest of all the samples. Perhaps the latter was an outlier, or perhaps the whole observation is a result of pure randomness.

Table 12: Results for finding VC 4 and 5 for 12 different samples.

sample	VC 4	VC 5	current VC 4	current VC 5
30 DOP, 1	14/14	28/30	139.7nA	123.9nA
30 DOP, 2	13/14	29/30	107.5nA	85.4nA
30 DOP, 3	14/14	25/30	68.8nA	59.7nA
30 DOP, 4	14/14	30/30	252.3nA	210.2nA
20 DOP, 1	14/14	29/30	84.9nA	130.7nA
20 DOP, 2	14/14	30/30	418.5nA	498.6nA
20 DOP, 3	14/14	30/30	272.9nA	28.5nA
20 DOP, 4	14/14	30/30	198.8nA	155.5nA
10 DOP, 1	14/14	28/30	525.1nA	488.8nA
10 DOP, 2	13/14	30/30	876.6nA	823.0nA
10 DOP, 3	11/14	30/30	296.0nA	520.6nA
10 DOP, 4	14/14	26/30	280.7nA	275.8nA

This is a potential thing to explore in a future more larger experiment.

7.5 Conclusion

To summerize it is possible to find VC 4 and 5 in the simulated networks as it was demonstrated on a real life device. I've also outlined some of the criticism for my solution for the current experiment with regards to choosing the voltage range and I suggest to look into ways to incorporate electrodes' current values into the fitness function to incorporate the fact that high currents may be harmful to the real life devices (of course we cannot see that harm in simulations, as this is not part of the model). The problem with this currently is that the exact dangerous current values for the real life samples is not known since only the output current is measured. Hopefully in the future more information will be known about that.

In this experiment using the cluster provided less benefits, but it could be because the increase in data points was not very dramatic. By running for more time or using more cluster resources one could consider making larger experiments to perhaps find some interesting patterns.

8 Implementation overview

8.1 Structure

The code written in this Thesis can be accessed from . The repository has both the code made for this Thesis and for B. de Wilde's Thesis ([11]). In this Chapter I am going to discuss both my additions to B. de Wilde's code and my original code.

The relevant code can be divided into 3 parts:

- root functionality - most reusable code. Dopant network object, genetic algorithm, utility functions.
- Go code - everything written in go language, that mostly includes various simulation functions explained in Chapter 3
- thesis related scripts - These are python scripts that use the root functionality to achieve various research goals. These include generating test sets and validating different simulation functions on those test sets for Chapter 3, starting lot's of GA searches for Chapters 6 and 7, generating animations and compiling graphs (like in figure 15) for GA search results which relates to Chapters 5, 6 and 7, a script for uniqueness validation used in Chapter 3.

To use any functionality in the repository make sure to read README.md in the root, as that explains what you need to install. Both me and B. de Wilde used Linux based systems to run this code, but in principle it should also work in Windows, as both Python and Go are cross platform.

8.1.1 Root functionality

There are 5 important files in the root folder. All of them are python code:

- kmc_dopant_networks.py - This defines the dopant network object. It has the dopant placements, donor placements, electrode information (placement and voltage), various pre-calculations. This code is mostly written by B. de Wilde, but I did make a few additions. Namely saving and loading that object from a file, and adding the ability to start simulations on that object using different simulation methods (for both go and the original python simulation).
- kmc_dopant_networks_utils.py - This has various utility functions, mainly related to making visuals. This was also started by B. de Wilde, but I've added all the visual related functions here (see Chapter 5). I've also added a few additional functions that I keep reusing in thesis related scripts.
- dn_search.py - My general implementation of Genetic search algorithm as described in 4. It also has several other search algorithms like simulated annealing and greedy, but since I did not end up using them much in the Thesis, I'm not going to discuss them further.

- voltage_search.py - This implements my dn_search object to be used for searching control voltage values as needed in Chapters 6 and 7. It is a good example of what has to be reimplemented if you would want to use the GA, but for slightly different purpose.

- dn_animation.py - This is in the root, and it has various functions that are used to create animations in the Thesis.

8.1.2 Go code

Various simulation solutions explained in Chapter 3 were written in Go language. They are still called from Python, and in order to do so I have to export go functions into a C shared library (by following tutorial from [2]), which Python then uses. One big problem with this solution was passing multi dimensional arrays. I couldn't find examples of how to do that directly, so I ended up wrapping them into 1 dimensional arrays and unpacking them in go. To do that nicely I separated wrapping part from functionality. There are 4 important files in goSimulation folder:

- simulation.go - has the functions for optimized go simulation (you can control weather or not you use recording of states or not), and pruning solution explained in Chapters 3.2 and 3.4

- probabilitySimulation.go - has the functions for probability occupation simulation explained in Chapter 3.3.

- simulationWrapper.go - has the functions that are exported into C shared library. All issues related to transferring data from python to go and back are dealt in this file. Functions in simulation.go and probabilitySimulation.go are called from here, but could also be used independently if one was simply working in pure Go environment. Also the second solution for the parallelism described in Chapter 3.6.2 is implemented here.

- pythonBind.py - this wraps the technical parts of calling a simulation from a C shared library in python. This opens two functions up for the rest of the python code - one for single simulation and one for paralleled bulk simulations discussed in Chapter 3.6.2. The packing of data and formatting it to acceptable data structures (since python itself is very loose with it's data structures) is all done here.

8.1.3 Thesis related scripts

In order to arrive to many of the results in this Thesis additional code had to be written to use the core functionality in an intelligent way. They can serve as examples for future work on this code base. Some of these scripts take use of passable parameters. This was mainly done so I could easily use these scripts in the paralleled way as explained in 3.6.3.

They include following files:

- generate_tests.py - this file has functions which were used to generate the test sets used in Chapter 3.5.
- validate_tests.py - this file has functions which were used to validate different solutions with different parameter values (like number of hops or pruning threshold). Based on the results it outputs statistics as reported in Table 2 and graphs as seen in Figures 13 and 14.
- voltage_search_tests.py - this file is used to initiate searches and was used to get the results in Chapter 6 and 7. It has functions to perform all the searches for some VC dimension (like 4, 5 or 6), to start a second run of searches for a subset of cases only (used for cases, for which we didn't find a solution), or to just run searches on cases one is interested in (like XOR). It is also possible to pass parameters for this script so that we could use this script in a cluster.
  - -m : mode, currently either "xor_only" or "vc", which decide what kind of searches are going to be performed.
  - -d : number of dopants, if we are generating a new random dopant network to perform searches, how many dopants does it have?
  - -i : starting index, as the search results are going to be saved into a file, this index is used as a starting point when deciding the file name. This is used to avoid overrides when executing scripts in parallel, or when I want to preserve previous results.
  - -t : If you choose the mode "xor_only", then how many different random placements are we going to use to search for XOR boolean gate. The larger the number, the more time it is going to take to execute the script.
  - -v : If you choose the mode "vc", then what vc are we going to explore - 4, 5 or 6? Currently I don't support more as I haven't defined the points for higher dimensions, but once someone does that, it should work for higher dimensions as well.
  - -f : If you choose the mode "vc", and want to load the dopant network from a file, you can specify the file name to load it from. This is used so I could start multiple different VC searches in parallel.
- AnimateSwipe.py - this script can generate the animations explained in Chapter 5.3 from a dopant network object file, that is generated as a result of search algorithm. In addition to creating the animation this script does lot's of simulations on the final voltage configuration, that is used to get a better idea about the noise and actual separation of the solution.
- SaveResults.py - This is used to do lot's of simulations on some dopant network object saved in a file, similarly as in AnimateSwipe.py, but without generating an animation.

- CompileGraphs.py - this script is used to compile several results into a single pictures as for example seen in figures **??** or 15. Either AnimateSwipe.py or SaveResults.py has to run before this script can be used as it relies on having the necessary simulations made for generating these pictures.
- generate_samples.py - This is used to create random dopant network objects. This can be used in parallel experiments that want to use the same dopant network object.
- UniquenessValidation.py - This is used to generate the results seen in Tables 4 and 5.
- printTables.py - This is used to generate the latex tables for my thesis.

8.1.3.1 Experiment process

In order to carry out a simulated experiment there are a few steps one would most likely have to take. Firstly to decide the initial conditions (for example how many dopants), search target (for example which boolean gate, or are we measuring VC) and search strategies (what are the parameter values for Genetic search). Examples of defining these things can be found in voltage_search_tests.py and UniquenessValidation.py. The result of this step should be files of dopant network objects. For each result you will have one file (so if I check VC 4 for 1 network, I will get 14 files, as I have to do 14 searches).

Second step that I always did was to simulate the results in a higher detail. I would run 40 simulations with 5 million hops for each of the input combinations, that would give me a really good estimate for noise to signal ratio and the overall quality of the solution. This is done in both AnimateSwipe.py and SaveResults.py. The results of this step are again saved into a file, and the exact location depends on the script used. You can also use these scripts as an example starting point.

Finally I want to compile the results, most likely to get some statistics etc. In my thesis I used CompileGraphs.py to generate graphs to visualize the results (for example see Figure 15) and also to count the number of successful searches. If you used AnimateSwipe.py you could also see an animation of the behaviour of the found simulated dopant network.

If you want to run large scale experiments, one would have to consider using cluster and parallelism. Currently that however requires one to break the experiment down to smaller pieces, where each piece involves discrete number of searches. For example during my Thesis I played around with a cluster which used SLURM workload manager, for which I made several .sbatch files, which can be found in slurm folder in the repository. There you can see how I use command line arguments and starting scripts in the background to make a job that does many searches in parallel.

8.2 Other people using the library

As a lot of work was put into improving the simulation, trying new techniques for genetic algorithm and adding new visualization functionality, that all formed into something of a library. One very important question is if that functionality is also usable by other people now and in the future.

So far there have been 2 other users using my parts of my code. B. de Wilde, who mainly used the Go simulation with transition rate recording in his own thesis and B. van de Ven, who tried to look for VC dimensionality for lower electrode systems.

Since the sample size is very small, to evaluate the library usage I decided to use a free-form questionnaire to be answered by the 2 users. I am including the following questions:

- What did you want to do with the library?
- Did you manage to do what you set out to do?
- Did you have any hurdles using the library? Which of those were a one time problem, which were continous problems?
- What potential use do you see for that in the future?
- What are your general concerns related to the library?

8.2.1 Questionnaire results

8.2.1.1 Case 1

B. de Wilde wrote the initial KMC simulation and the core code for the silicon dopant network simulations in python. For both our theses we worked on the same code base, and he took advantage of my speed improvements for his work. These are the questionnaire results, all the answers are quotes from B. de Wilde:

Q: What did you want to do with the library?

A: "For me the main goal was to do the exact same simulations I was already doing, but faster. Also, I wanted to use the library in a way that modified my previous scripts as little as possible."

Q: Did you manage to do what you set out to do?

A: "Yes. Indrek also provided a Python wrapper function which I could use in place of one of my own functions. That way, most simulations ran 10x faster with hardly any work on my side."

Q: Did you have any hurdles using the library? Which of those were a one time problem, which were continous problems?

A: "I have had no continuous problems. In the process of debugging I believe there were one or two instances where the wrapper/library returned wrong results, but that is only natural in development. In the beginning I also had some trouble finding the right function arguments due to documentation, but that was very minor."

Q: What potential use do you see for that (I am assuming the library?) in the future?

A: "I think the library will remain the backbone of any kinetic Monte Carlo work within NanoElectronics. It works well and is general enough to be future proof. New people can develop new simulation tools using the library as a starting point."

Q: What are your general concerns related to the library?

A: "My main concern would be regarding documentation. As I mentioned above, I think the library is very nice and could help many people. A condition for that to be effective is that people can understand it, preferably without person to person interaction (i.e., just from text/documentation/thesis)."

8.2.1.2 Case 2

B. Van de Ven, who is doing PhD in the Nanoelectronics research group used the library to explore having different number of electrodes. These are the questionnaire results, all the answers are quotes from B. Van de Ven:

Q: What did you want to do with the library?

A: "I want to analyze the influence of the number of electrodes on the computation complexity of the devices used in the simulations."

Q: Did you manage to do what you set out to do?

A: "Not yet, I am still using the library since the search over the parameter space takes time. However, I do have some initial results that show that I can achieve what I want."

Q: Did you have any hurdles using the library? Which of those were a one time problem, which were continuous problems?

A: "It took me some time to find which parameters to tune to be able to make the variations I want to achieve. Now that I know where I can change the parameters this is no longer a problem for me."

Q: What potential use do you see for that in the future?

A: "This library can now be used to perform simulations and analysis that reduce the timescale of such an optimization project as described above to 1 or 2 weeks instead of months when making the hardware. This can help us define design rules that can be iteratively improved using this library."

Q: What are your general concerns related to the library?

A: My main concern is that some things could be slightly more user friendly, especially for potential new users.

8.3 Summary

Based on the questionnaire it is clear that the library has functional capacity to be useful. However the main concern is the ease of use, since it does currently rely on user having at least basic understanding of Python programming language.

Regarding the documentation this has been fixed by now, in addition there are some example uses, but in the end I believe that person from person knowl-

edge transition is still going to be vital for the usefulness of this library. Also as more use cases accumulate within the group, a second iteration to make it more user friendly could be considered as part of some future Thesis.

9 Summary

9.1 Recap

During this Thesis I had following successes:

- Improved KMC algorithm for simulated physical model of silicon dopant networks.
- Developed a code base that could be used to perform large scale simulated experiments for the physical model of silicon dopant networks, and also ran a few of such experiments on my own laptop but also on a cluster.
- Had other people successfully use my code base that demonstrate the usefulness of my work.

9.2 Contributions

The main contribution of this Thesis is making the Kinetic Monte Carlo algorithm that is used to simulate silicon dopant networks more efficient. Even if under different parameters the current best solution (remembering transition rates for all unique states) is no longer viable (because too many unique states), tools to evaluate the new best approach are still available. It is possible that certain parameters would increase the number of unique states significantly in which case we could probably make gains using pruning with certain threshold. However if we can show that the current solution with less dopants but physically impossible parameter values simulates reality well and we could actually have a 1 on 1 map between the number of dopants in simulation vs. real life, that would have the benefit of being a fast performing solution, while still being very useful for simulated experiments.

As part of the thesis a genetic algorithm solution, separate from the one used by NanoElectronics was developed. One important feature was supposed to be the uniqueness feature, which was supposed to help against getting stuck in a local mimimum. So far it did not show to do so, but further investigation into that could be made. The tools to try different uniqueness values and also to have an uniqueness schedule are all there. It is also possible that we should consider different distance functions.

In addition I laid out a path to making large scale simulated experiments, as there are sample code to carry out the experiments, analyze them and run these experiments on a cluster.

The usefulness of the developed code base (library) was validated in a few ways. Firstly I used it to perform some simulated experiments, which NanoElectronics group had interest in, and secondly a few members from that group have already used the code base in their own work as discussed in chapter 8.

The final potential contribution is to open up more cooperation between NanoElectronics group and Computer Science students. This Thesis should be

a good starting point to understand the silicon dopant networks enough to be useful, without going too deep into physics.

9.3 Conclusions

There are three big group of conclusions that can be drawn with this Thesis:

- Firstly regarding the performance of the Kinetic Monte Carlo simulation. With the parameters found in [11] the best solutions was the record transition rates and reuse them as we revisit the same states. This can mostly be explained by the fact that since I_0 is quite large, the electrons entering the system repel other electrons from entering the system, meaning that there are fairly small number of electrons in the system at any given time. Another promising improvement is pruning, but recording is currently just far superior.

- Secondly the simulated experiments have added more positive arguments to support that the simulation model resembles reality. For example I found very similar results when exploring VC dimensions as did dr. H. Ruiz in his experiment on the real device. Another thing I found that the output current may quite often be much smaller than the currents between other electrodes (compare tables 10 and 11).

- Thirdly as you can see from the cluster results in Chapters 4.2 and 6.5 you can see that by having more data points, we gain new results which smaller sample sizes may miss. This is a clear case for using cluster and fast simulations to run large scale simulated experiments.

- Finally the usability of the code base. The most positive take away is that it is functionally working and usable. Even though the user friendliness can be improved, it definitely saves a lot of effort when making new simulated experiments. As you can see from both of the use cases in chapter 8, the goals were achieved both times.

9.4 Outlook

There are multitude of paths to develop upon my work.

- Firstly one can use this library to run new simulated experiments but also to extend existing ones: One thing I would do if I had more time and knowledge is to adjust the current experiments to take into account the currents during the genetic algorithm. That means that after certain amount of current I would penalize the score to avoid solutions which would break the device in real life examples. The main consideration here however is what should the current be? And that requires additional insight from the NanoElectronics group. After that it could easily be made by simply programming a new error function and using that instead.

- Secondly another Computer Science student could work on making the library more user friendly for the physicians. I do believe however that this would be more reasonable to do once we have gathered more use cases, so that we could find commonality in both use cases and biggest hurdles, before we start tackling user friendliness.
- Thirdly further investigation into Genetic Algorithm uniqueness feature could prove to be useful. Also related to Genetic Algorithm another approach to make simulations faster is to adjust the total number of hops. From what I've seen so far, there are cases where we could allow much more noise to still validate viable solutions, and also the noise seems to be different depending on the conditions (dopant placement and electrode voltages), as seen when comparing the two test sets in chapter 3.5. If we could predict the noise in real time during search, we could theoretically adjust the number of hops accordingly to increase efficiency.

References

[1] *Bhattacharyya distance*. URL: https://en.wikipedia.org/wiki/Bhattacharyya_distance.

[2] *Calling go functions from other languages*. URL: https://medium.com/learning-the-go-programming-language/calling-go-functions-from-other-languages-4c7d8bcc69bf#.n73as5d6d.

[3] *Crossover (genetic algorithm)*. URL: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).

[4] J. van Gelder. "Using lowly doped silicon to create reconfigurable Boolean logic gates using an evolutionary algorithm". MSc Thesis. University of Twente, 2017.

[5] Ren Hori. "Darwin on a chip: Think Outside of Logic". MSc Thesis. University of Twente, 2018.

[6] Michael L. Mauldin. "Maintaining Diversity in Genetic Search". Tutorial. Carnegie-Mellon University, 1984.

[7] Allen Miller and Elihu Abrahams. "Impurity conduction at low concentrations". In: Physical Review 120.3 (1960), p. 745. Physics department, Rutgers University, 1960.

[8] Julian F. Miller and Keith Downing. "Evolution in materio: Looking Beyond the Silicon Box". In: Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware (EH'02). School of Computer Science, The University of Birmingham et al., 2002.

[9] *Vapnik–Chervonenkis dimension*. URL: https://en.wikipedia.org/wiki/Vapnik%5C%E2%5C%80%5C%93Chervonenkis_dimension.

[10] Darrel Whitley. "A genetic algorithm tutorial". Tutorial. Computer Science Department, Colorado State University, 1994.

[11] Bram de Wilde. "Dopant networks for energy-efficient classification". MSc Thesis. University of Twente, 2018.

10 Appendix

10.1 Bhattacharyya distance

Bhattacharyya distance is used to compare two Gaussian distributions. I use this in my work to compare different simulation solutions, and to deal with the inherint noise that my KMC simulations will have in their results. I use this to both have a sanity check for optimizations actually working as intended, but to also test the accuracy for solutions where I know beforehand that there should be no equivelance in the behaviour compared to the baseline KMC solution.

The main downside of Bhattacharyya distance is that it is potentially hard to interpret. In my work I refer to Bhattacharyya distance as D_B score. D_B score has values in the range $[0, +\infty]$. 0 meaning both Gaussian distributions being identical, and as the value grows larger the more different the Gaussian distributions are. The difference may be cause by both the mean and standard deviation. In general I consider a score of less or around 1 to be ok. In figure 12 you can see different pairs of Gaussian distributions and the resulting D_B score to get a better intuition about how the score changes. Each row in the picture is meant to show a different scenario as to what may be the underlying story behind a high score.

- First row shows the difference of means growing larger, while still having substantial amount of overlap.
- Second row shows the difference of standard deviation growing. This could happen if we compared two solutions where one of them simply runs more number of hops and therefore gets less noise.
- Third row shows the difference of means growing significantly larger, and having no overlap.
- Fourth row shows how even if having vastly different means, if standard deviation increases for one of them, the D_B score goes down.

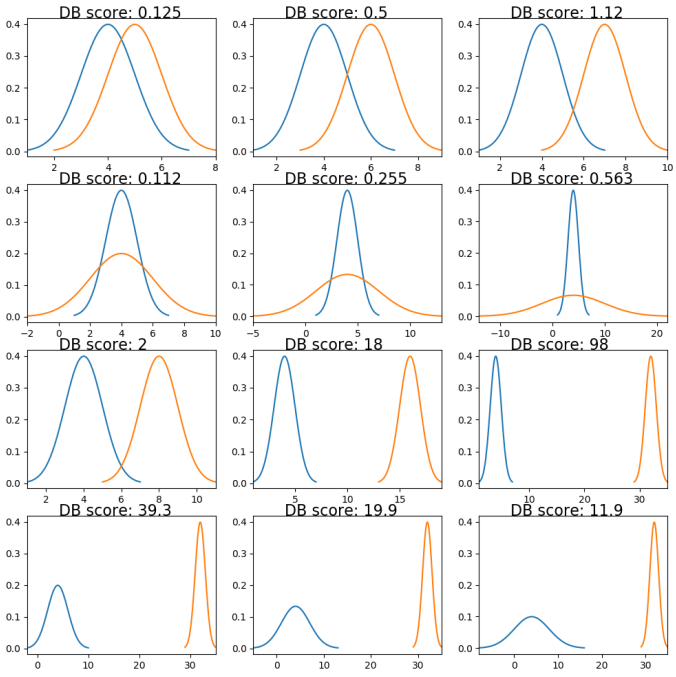

Figure 12: This is to give one an idea about what DB score means in practice. In general things with 1 or less DB score can still have some overlap, and in the case of comparing simulation solutions could easily be caused by noise error even after 5 simulations.

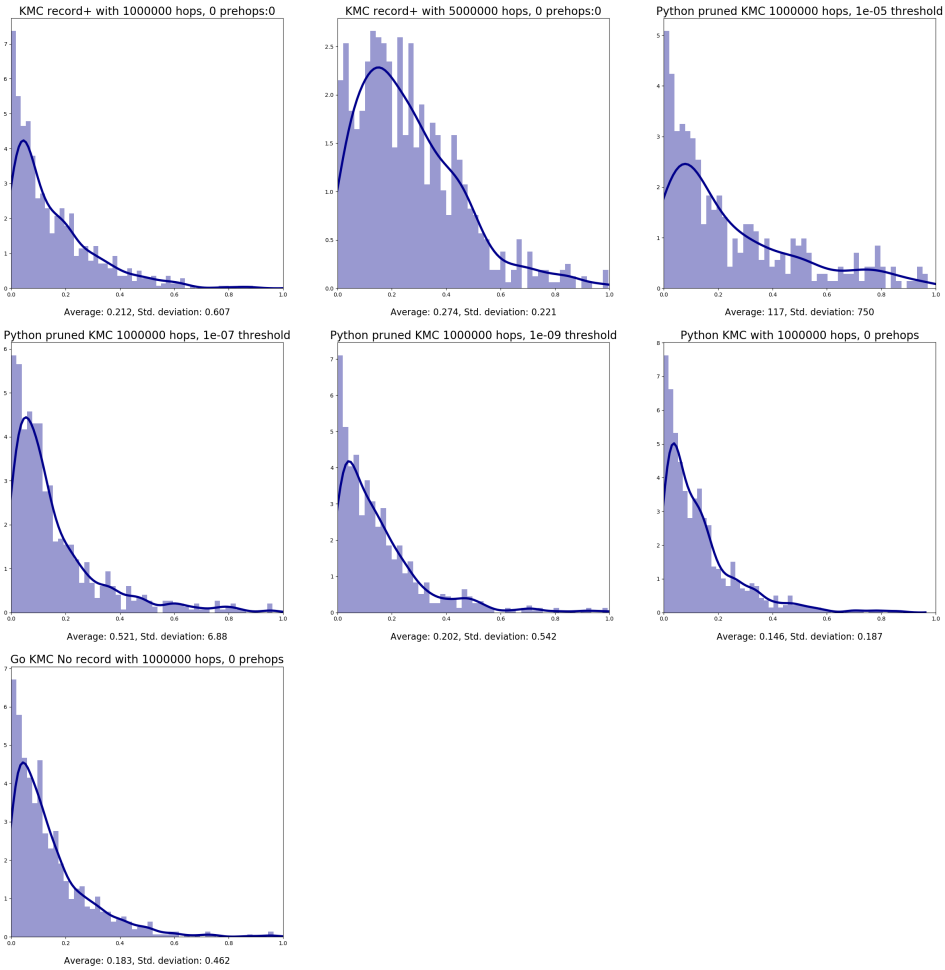

Figure 13: This is the D_B score for RND test set for all the different solutions. I only plot D_B values from 0 to 1.

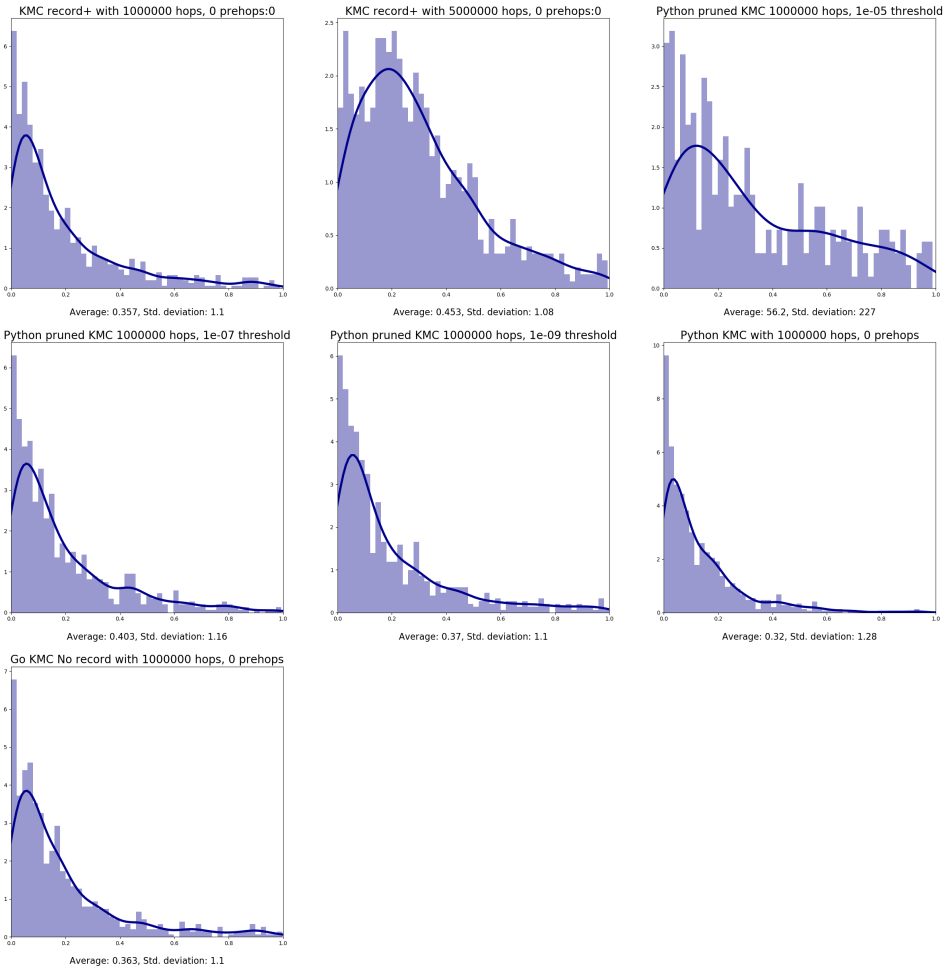

Figure 14: This is the D_B score for XOR test set for all the different solutions. I only plot D_B values from 0 to 1.

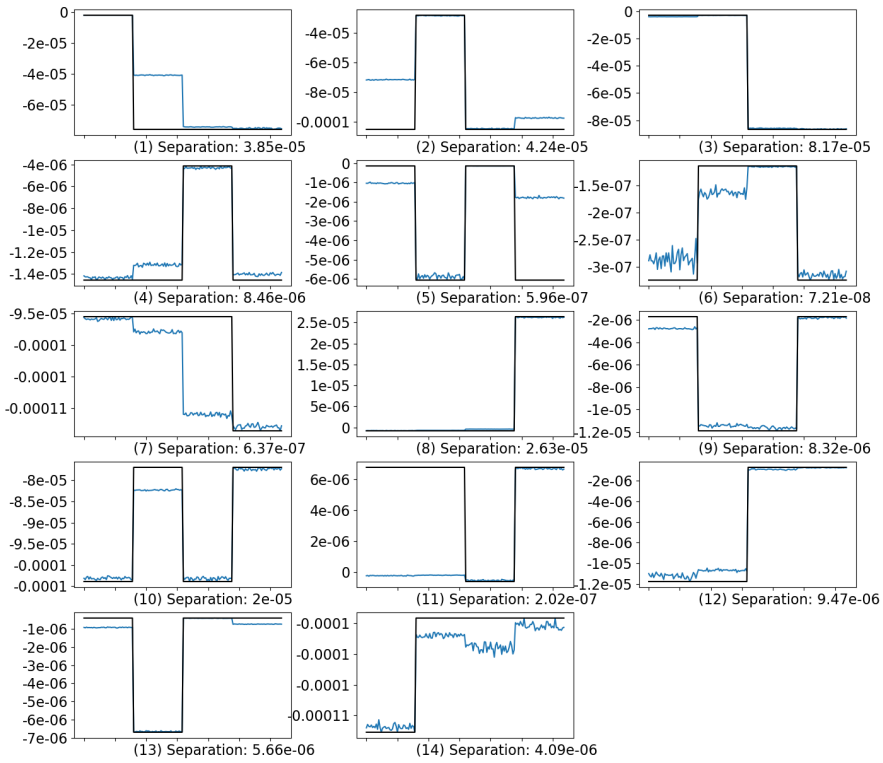

Figure 15: VC 4 Results for a single 30 dopant network. Here you can see all the different classifications and corresponding currents. In this case all 14 classifications were found.

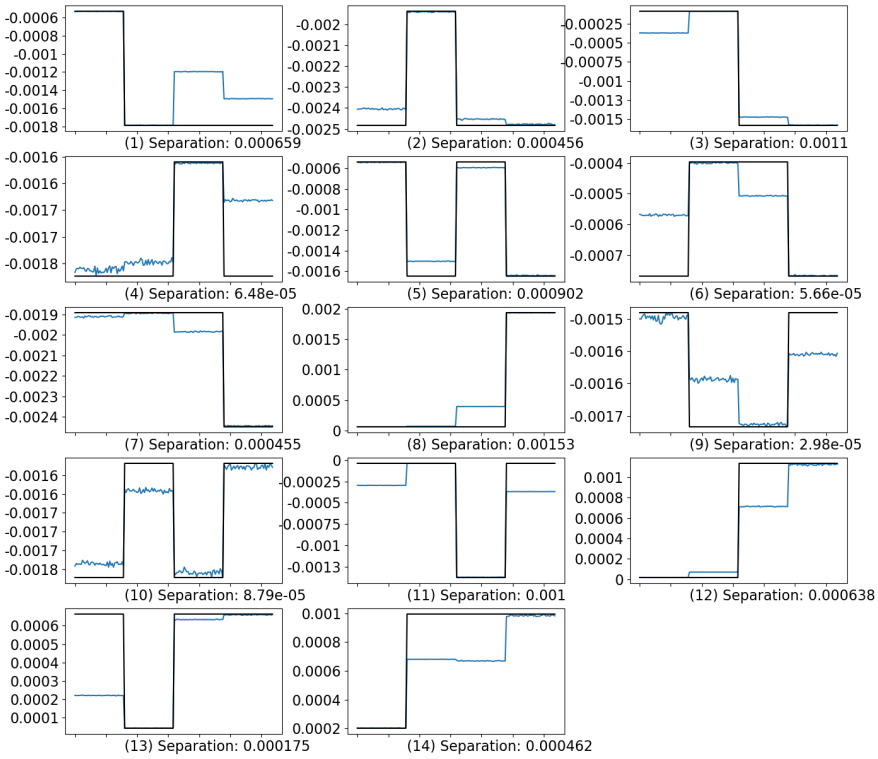

Figure 16: VC 4 Results for the forth sample of 10 dopant networks. Here you can see all the different classifications and corresponding currents. In this case all 14 classifications were found.