Budapest University of Technology and Economics

Study of dipole-dipole coupled protein-based circuits using self-developed simulation software

Scientific Students' Associations Report

Authors:
Edvárd Róbert Bayer
Richárd Krisztián Csáky

Supervised by
dr. Balázs Rakos

2016

Kivonat

Az IT iparban az alkatrészek mérete egyre kisebb és kisebb, hogy minél nagyobb sűrűséggel lehessen őket integrálni a termékekbe. Moore törvénye szerint a tranzisztorok száma másfél évenként megkettőződik egy azonos nagyságú processzorban. E jelenség miatt a jövő tranzisztorai olyan kicsik lesznek, hogy gyártásuk nehézkes lesz különböző kvantum fizikai jelenségek figyelembe vétele nélkül.

Már most sok más módszert tesztelnek a kutatók mellyel a tranzisztort helyettesíteni lehetne. Az egyik ilyen módszer a molekuláris számítógép, melyben molekulák látják el a tranzisztorok feladatát. Egy lehetséges molekula erre a célra a Dronpa, mely egy mesterséges fehérje. Ez és más hasonló tulajdonságú fehérjék képesek rá kapcsolt elektromos tér által generált jelek továbbítására környező molekulákra dipól-dipól csatolás segítségével. Kedvező tulajdonságaik közé tartozik még, hogy az elektromos jelet képesek terahertz nagyságrendű frekvenciával továbbítani, valamint kisebb teljesítményfelvételűek és disszipációjúak mint a tranzisztor. E tulajdonságokat felhasználva megmutatjuk, milyen módon tudnák helyettesíteni a tranzisztort a fehérjék digitális áramkörökben, és hogyan valósíthatóak meg különböző logikai áramkörök molekulák segítségével.

A szemléletes és átlátható szimulálás érdekében megalkottunk egy programot, 3D-s grafikai felülettel, melybe integráltuk a fehérjék viselkedését leíró egyenleteteket. A program segítségével könnyen meg lehet adni molekuláris struktúrákat és elektromos tereket, és ezek viselkedését is lehet vizsgálni illetve szimulálni bizonyos egyszerűsítések mellett. Továbbá bemutatunk egy algoritmust mely segítségével a program magától képes egyszerűbb, logikai függvényeket megvalósító struktúrák megkeresésére a megadott bemeneti és kimeneti feltételeket figyelembe véve.

Az algoritmus és szimuláció segítségével bemutatunk pár egyszerűbb logikai kaput és logikai függvényt megvalósító molekuláris struktúrát. E struktúrák viselkedésének vizsgálatát oly módon végezzük el, hogy párhuzamokat állítunk fel a most is használt tranzisztoros és a molekuláris logika között, ez által szemléltetve miként lehetne használni a Dronpa, és más hasonló tulajdonságú fehérjét digitális logika megvalósítására.

Abstract

In the IT industry the size of electronic components is constantly shrinking in order to integrate them with greater density in laptops, phones or other products. According to Moore's law the number of transistors occupying the same space in a processor is doubling every 18 months. Because of this phenomenon the transistors of the future will have to be so small, that their manufacturing will become nearly impossible without taking into consideration quantum physical phenomena.

Already a great number of approaches to replace transistors with something else are being researched and talked about by scientists. One such way is the molecular computer, where special molecules take care of the transistor's duties. A possible candidate is the molecule named Dronpa, an artificial protein. This, and other similar proteins have the ability to transmit electric field-induced signals to other nearby molecules with the aid of dipole-dipole coupling. Among their beneficial properties is the ability to transmit signals in the terahertz frequency regime, and lower power consumption and dissipation than transistors. Using these qualities we show how a transistor can be replaced by these proteins, and how various logic circuits can be achieved with the help of molecules.

In order for the simulations to be clear and visual we created a program containing a 3D graphical interface, and we integrated the equations which describe these proteins into the program. With the help of this program defining molecular structures and electric fields becomes an easy task, as well as, using some simplifications, examining and simulating the behaviour of such structures. Furthermore we present an algorithm which can be used to find smaller molecular structures that realize desired logic operations, by defining the required input and output conditions.

With the help of the algorithm and the simulations we present logic gates and a couple of easier logic functions, realized by molecular structures. The studying of the behaviour of such structures is done by correlating between present transistor logic and dipole-dipole coupled molecular logic, thus showing how Dronpa and other similar proteins can be used to accomplish digital logic.

Contents

1 Introduction **3**
1.1 Why are Dronpa and proteins in general important 3
1.2 NAMD 4
1.3 The reasons behind the need of our program 4

2 Background **6**
2.1 Dronpa Properties 6
2.2 Basic Equations 7
2.2.1 Electric field interactions 7
2.2.2 Differential equations 8

3 Program **10**
3.1 Logical and Structural build-up part of the program 10
3.1.1 Handling the molecules 10
3.2 Graphical build-up part of the program 11
3.2.1 Basic Setup 11
3.2.2 Draw.c 11
3.2.3 Shapes.c and Textures.c 12
3.3 Modified Equations 13
3.4 Explaining the Running of the Simulation 14
3.4.1 Applying the modified equations 14
3.4.2 Simulation Algorithm 14
3.4.3 User Inputs 15
3.4.4 Exporting the data 16
3.5 Explaining the Searching Algorithm 17
3.5.1 Reasons and goals 17
3.5.2 User Inputs 17
3.5.3 Algorithm background and basic rules 18
3.5.4 Structure Search 19
3.5.5 Detailed Search 21
3.5.6 Finding the result 22
3.6 Other Functions 23
3.6.1 Save function 23
3.6.2 Load function 23
3.7 User Manual 24
3.7.1 First start of the program and navigating the graphical interface . 24
3.7.2 Other keyboard inputs and parameters 25
3.7.3 Switching between parameter and coordinate input 26

4 Simulations, results **28**
4.1 Initial Molecule Structure . . . 28
4.1.1 3 molecule structure . . . 28
4.1.2 4 molecule structure . . . 29
4.1.3 The XOR gate . . . 30
4.1.4 Resetting a protein's dipole moment . . . 33
4.2 Logic Gates as found by the *Searching Algorithm* . . . 34
4.2.1 AND Gate . . . 34
4.2.2 OR Gate . . . 35
4.2.3 NAND Gate . . . 36
4.2.4 NOR Gate . . . 36
4.2.5 XOR Gate . . . 37
4.2.6 XNOR Gate . . . 38
4.3 Logic Functions . . . 38
4.3.1 The Half-adder . . . 38
4.3.2 The Half-subtractor . . . 39

5 Discussion **41**
5.1 Concept related issues . . . 41
5.1.1 Signal transfer . . . 41
5.1.2 Reset problem . . . 41
5.1.3 Logic . . . 41
5.2 Comments on the program . . . 42
5.2.1 Neighbours . . . 42
5.2.2 Searching Algorithm related problems . . . 42
5.3 Improving the Simulation . . . 43
5.3.1 Programming improvements . . . 43
5.3.2 Further modules . . . 44
5.4 Improving the Searching Algorithm . . . 44
5.4.1 Improving the structure finding algorithm . . . 44
5.4.2 Further ides to the structure finding algorithm . . . 45
5.4.3 Further improvements to the Searching Algorithm . . . 46

6 Conclusion **47**

1 Introduction

In this chapter we give a short introduction of the Dronpa protein, and discuss the reasons and benefits of our program.

1.1 Why are Dronpa and proteins in general important

New computing methods are needed more than ever, now that the transistor is reaching its theoretical size limit, thus making it impossible to further increase the density of microelectronic circuits. Novel technologies to replace transistors are researched more and more. DNA-based computing architectures have been proposed before [1].

Proteins are potentially promising candidates to serve as the building blocks of novel computing architectures. Their benefits from our point of view include low cost, wide variety, and that they can be engineered in order to provide desirable properties [2, 3]. A promising solution for the integration of molecular devices is Coulomb coupling, and this is our main focus as well [4, 5, 6, 7, 8, 9]. Theoretical proposals for both electric field and photon pulse-driven solutions for the realization of Coulomb coupled protein logic circuits have been offered before [2, 3].

Dronpa is a special artificial protein, and already several simulations were performed on it [10, 11]. These showed that electric fields induce a proportional change in the protein's dipole moment. Dronpa molecules placed next to each other transmit electric field induced dipole moment change in an inverting way [12]. Another important property is the memory, meaning that after the electric field used is turned off, the dipole value won't return to its initial value [12]. Furthermore the size of a Dronpa protein is less than that of the smallest transistor today, and it has been shown through simulations that it can reach switching frequencies in the terahertz range [12]. These and other qualities show that Dronpa and other proteins which have good polarizing qualities are good candidates to replace transistors.

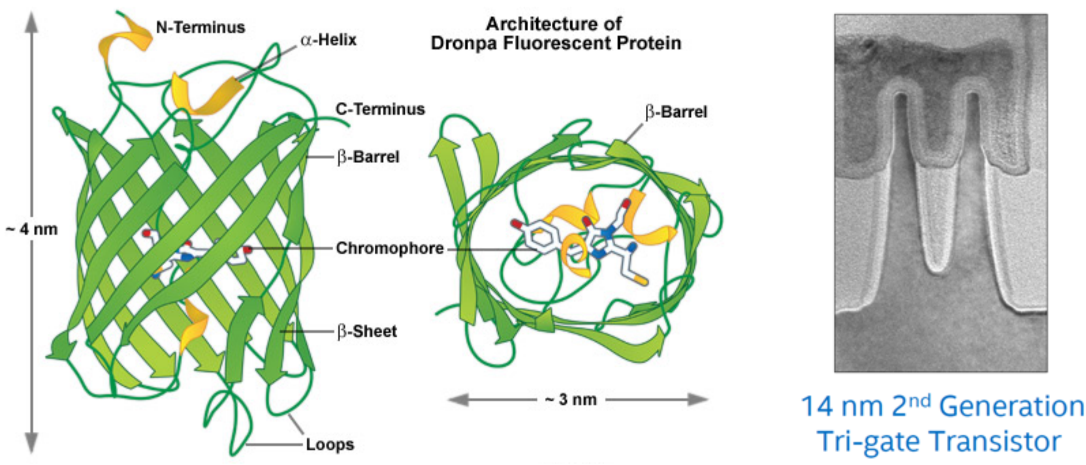

Figure 1.1: Comparing the sizes of Dronpa and the smallest Intel transistor [13]

1.2 NAMD

NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems [14]. The simulations that we used to check the validity of our own program were also run using NAMD [12]. The software is great for detailed molecular simulations, but it lacks an actual graphical user interface. VMD can be used to visualize the results of NAMD simulations, the picture in Section 1.1 was also taken in VMD [15]. In Section 2.1 we show more simulation results on Dronpa done with NAMD. A problem with the program is that it's too detailed for our needs, and takes a relatively long time to simulate the effect of an electric field even on only 1 protein. We don't really need to simulate the proteins in detail since we have the differential equations, with which we can describe the behaviour of Dronpa in a simple, compact form, further described in Section 2.2 [16]. NAMD is a very powerful program but it lacks user-friendly tools, and it isn't specific enough for Dronpa simulations, thus we decided to make our own simulation software. Since we built our on program for the specific task it can simulate the effect of an electric field on a protein in less than 1 second.

1.3 The reasons behind the need of our program

As described in the previous section we decided to build our own simulation program. This is beneficial since we can make it as specific as we want to the Dronpa molecule, thus only including equations and functions that are necessary, effectively making the program much faster than other simulation software. This is partly due to our approach to base the entire simulation on the differential equations that describe the behaviour of Dronpa and similar proteins with great accuracy [16]. Obviously our program can be used in general to simulate other proteins as well, which might even

have better polarizability properties, but we focused on Dronpa since the constants used in the equations were already at hand. Another important aspect is the 3D graphical interface, which other simulation programs might lack, making the building of molecular structures very user-friendly, and visually pleasing. In the following chapters, after giving a basic theoretical background we discuss the challenges that we faced while building this program, and how we overcame them. Scalability and ease of use were the primary focus points when deciding what shape the program should take.

2 Background

In this chapter we are going to give a basic background of Dronpa, its properties and behaviour. A brief summary of the equations needed for simulation will also be given.

2.1 Dronpa Properties

As mentioned in the introduction we can load an electric field to a protein, and its dipole moment will react to it.

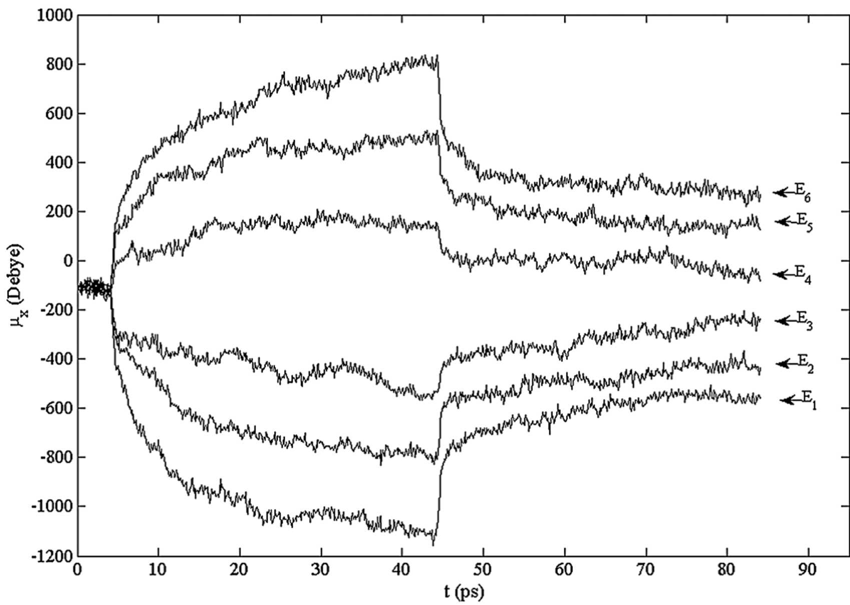

Figure 2.1: Electric field-induced response of Dronpa [12]

When a protein is affected by an electric field, its shape and dipole moment will change as shown in Figure 2.1. The protein has a viscoelastic property, meaning that after we turn off the electric field this part will completely reset. In the figure above however after turning off the electric field at 44 ps, the protein's dipole moment value does not reset completely, showing that it also has a viscoplastic property, which acts as some kind of memory [12]. These properties can be modelled using differential equations [16]. These equations are the mathematical model of the circuit analogy, and can be used for any protein that behaves similarly [12].

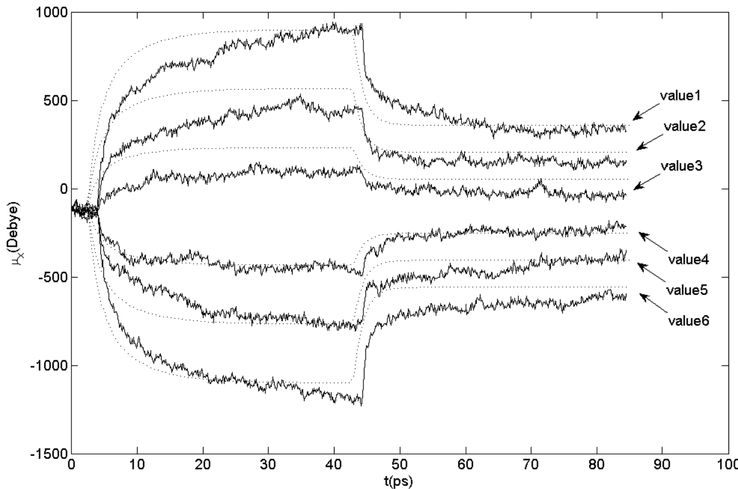

Figure 2.2: Comparison between NAMD and Matlab (using the differential equations) simulation results [16]
dotted line – Matlab
continuous line – NAMD

With the help of these equations we can closely approximate the behaviour of Dronpa as shown in Figure 2.2. Furthermore these equations are easy to implement into the program. Another benefit is that we don't have to use the electronic circuit analogy, thus using directly electric fields, and dipole moments in the differential equations described in Section 2.2.

2.2 Basic Equations

In this subsection we give a short description of the equations on which we based our simulation program. These are able to accurately describe the behaviour of Dronpa proteins when subjected to external electric fields. The model is not necessarily specific to Dronpa, and can be used for other coulomb coupled proteins, since it is the mathematical model of the circuit analogy discussed in [12].

2.2.1 Electric field interactions

After turning on an electric field on the protein, its dipole moment changes according to

$$\mu = \mu_0 + \alpha E, \tag{1}$$

where μ_0 is the dipole moment of the protein prior to the external electric field, μ is its dipole moment during the influence of field E, and α is its polarizability [16].

If the dipole moments of neighbouring proteins are parallel to each other, the magnitude of electric field E at the protein induced by its neighbour is

$$E = \frac{\mu}{4\pi\varepsilon_0 r^3}, \tag{2}$$

where μ is the dipole moment of the neighbouring protein and r is the distance between the protein and its neighbour [16]. In our simulations we used the physically lowest possible distance of 7 nm, and we applied electric fields in only 1 direction, thus the dipole moment of all the proteins also changes only in 1 direction.

If the dipole moment is given in Debyes and the electric field in kcal/(molÅe), then using the above equation we can calculate the following K constant

$$E = K\mu \tag{3}$$
$$K = 0.07 \cdot 343 \left[\frac{kcal}{D \cdot mol\text{Å}e}\right].$$

We implement this K constant with a negative sign in the program, because of the inverting nature of Dronpa. This constant represents the dipole-dipole interaction of the proteins, meaning that it is the factor by which a protein's dipole moment creates an electric field, which changes the neighbour's dipole moment.

2.2.2 Differential equations

The following equations are quoted [16].

The dipole moment can be calculated by the following equation

$$\mu_x(t) = \mu_{ex}(t) + \mu_{px}(t) + \mu_{0x} \tag{4}$$

where μ_x is the x component of the dipole moment of the protein during the influence of the field E. μ_{ex} is the viscoelastic-like change and μ_{px} is the viscoplastic-like change of the x component of the dipole moment. μ_{0x} is the x component of the dipole moment of the protein prior to the application of the external electric field.

The viscoelastic μ_{ex} part, due to the x component of the external electric field, E_x can be expressed by

$$\frac{d\mu_{ex}(t)}{dt} + \frac{\mu_{ex}(t)}{C_{e1x}C_{e2x}} = \frac{E_x(t)}{C_{e1x}}. \tag{5}$$

where C_{e1x}, and C_{e2x} constants can be determined from the electric field-induced response of the protein in question obtained either experimentally or with the aid of a molecular dynamics simulation software (see values later).

The viscoplastic $\mu_{px} = \mu_{px1} + \mu_{p2x}$ part, due to the same electric field can be expressed by

$$\frac{d\mu_{p1x}(t)}{dt} + A_x \frac{\mu_{p1x}(t)}{C_{p1x}C_{p2x}} = A_x \frac{E_x(t)}{C_{p1x}}, \tag{6}$$

where

$$A_x = \frac{1}{2}\left(sign\left(E_x(t) - \mu_{p1x}(t)/C_{p2x}\right) + abs\left(sign\left(E_x(t) - \mu_{p1x}(t)/C_{p2x}\right)\right)\right), \tag{7}$$

and

$$\frac{d\mu_{p2x}(t)}{dt} + B_x \frac{\mu_{p2x}(t)}{C_{p1x}C_{p2x}} = B_x \frac{E_x(t)}{C_{p1x}}, \tag{8}$$

where

$$B_x = \frac{1}{2}\left(abs\left(sign\left(E_x(t) - \mu_{p2x}(t)/C_{p2x}\right)\right) - sign\left(E_x(t) - \mu_{p2x}(t)/C_{p2x}\right)\right). \tag{9}$$

The C_{p1x}, and C_{p2x} constants can be obtained from the electric field-generated characteristics of the molecule, as well.

For the $Ce1x, Ce2x, Cp1x, Cp2x$ constants we used the following values [16].

$C_{e1x} = 0.008\ (ps \cdot kcal)/(D \cdot mol\text{Å}e)$
$C_{p1x} = 0.037\ (ps \cdot kcal)/(D \cdot mol\text{Å}e)$
$C_{e2x} = 180\ (D \cdot mol\text{Å}e)/(kcal)$
$C_{p2x} = 153\ (D \cdot mol\text{Å}e)/(kcal)$

The exact equations used in the program are given in Section 3.3.

3 Program

In this chapter we will talk about the logical, structural and graphical build-up of our program, discussing the modified equation which we used as well. We also explain simulation and other algorithms in detail. Furthermore we present a short user manual to explain and help navigate the program's features.

3.1 Logical and Structural build-up part of the program

This subsection focuses on the core of our program, the system of handling the properties of examined molecules. We wrote our whole program in C++, and used *Microsoft Visual Studio Enterprise 2015* for editing and debugging [17].

3.1.1 Handling the molecules

To be able to try several molecular structures, we needed a way to store their place and actual dipole moment in every step of the simulation. Therefore we needed the variables *double dip*, and *double dipA, dipB* (the reason for these two will be explained in Section 3.4).

We had the variables for simulation, but we needed to separate them for each and every molecule. For this purpose the easiest and most organized way is to pack the values belonging together in one object, so we defined *class molekula*.

In order to make enough objects for all proteins we decided to declare a three-dimensional array of the class (*molekula[36][36][36]*), so we could identify all molecules individually. However, we didn't need every one of the 46656 molecules in all cases, therefore we definitely needed one more variable in the class, which indicates if at a certain place there really exists an initialized molecule or just empty space for it. This variable is the *bool van*.

We needed only one last variable, which would show whether there is an external electric field on the protein (electric field values are set in a different part of the program). In the class this variable is the *bool ter*.

Ultimately, for changing the structure or the values of a protein's variables we would use external functions, hence every variable in the class had to be public. Finally the definition of the class is:

```
class molekula
{
public:
    bool van;
    bool ter;
    double dipA, dipB, dip;
};
```

The class for the molecules was made, but it still did not do anything. So like in any program, we needed functions, which would modify the objects – these are just mentioned in this section, and are explained in other sections.

The most important function of the program is the *futas*. It's called with the r key, and it simulates the structure, calculates the dipole moments, and creates the data file. It uses the modified equations, discussed in Section 3.3, and works exactly as it is written in Section 3.4.

The other two functions mentioned here are the *save* and *load* functions. In brief, with these one can save and load a given molecular structure to a file, so they can later run simulations on the same system. These functions are explained in detail in Section 3.6.

3.2 Graphical build-up part of the program

All the graphical parts of the program, as well as the interface were programmed using *OpenGL 2.1 Glut*, specifically the *freeglut* version [18]. All the *OpenGL* code was written in *C* language. In the subsections below we discuss the several parts which make up the graphical part of the program.

3.2.1 Basic Setup

Our basic setup of the *.c* and *.h* files follows the setup done in this *github* code [19]. Furthermore the basic setup of the scene, the camera rotation and movement, and the printing on screen was also coded accordingly to the github code. In the *main()* function we only initialize the Glut window, some opengl functions, our objects, and our global variables.

The *display.c* file contains important opengl functions which draw the scene, set up the camera, and the field of view, as well as deal with the reshaping of the window. The most important function here is *glutPostRedisplay()*, which has to be called whenever the scene drawing is changed.

In *print.c* the main print function is declared, which we can call to easily print characters directly on the window. This is done through *glutBitmapCharacter()*, and it basically allows to use the *print()* function as a normal *C* printing function.

In *interaction.c* all the keyboard and mouse interaction functions are declared. This is also the file where we declare all our own functions, like the simulation and the searching algorithm.

3.2.2 Draw.c

In this file, everything that needs to be drawn in the scene is declared. The light is drawn here in the *drawLight()* function. *drawParameters()* contains the drawing of *print()* bitmap characters. In the *drawAxes()* function place-holder black cubes are

drawn. The *drawCube()* function has a cube object input, and draws the specific cube if it was initialized.

The *drawScene()* function calls all the previously mentioned drawing functions. Also if the coordinate input counter reaches 3 it will initialize a cube for the given coordinates, through the *InitializeObjs()* function.

3.2.3 Shapes.c and Textures.c

In the *Textures.c* file a special function is used, which transforms 256x256 pixel or smaller *.gif* pictures into usable texture objects by *Opengl* [19]. This is very important because the specifying of coordinates is greatly simplified by these textures. We created 36 *.gif* pictures of the numbers from *0* to *9*, and of the characters from *a* to *z*. These textures mapped on the cubes, so it's very easy to notice what coordinates a specific cube has.

The *shapes.c* file is where we declared the *cube()* function, which is called through the specific coordinate parameters, from the *drawCube()* function. In this function a size parameter can be changed, which changes the size of the cubes. Also if the protein is specified to have a field on it, then the cube's colour will be green, or if not it will be red. To draw the cubes 24 3D vertexes are needed, in order to draw the sides of the cubes by using those vertexes. The reason why we need 24 points, is because we actually need to give 3 rectangles, when drawing 1 side of the cube. Each rectangle contains one texture, so each side of the cube will have 3 coordinate textures on it.

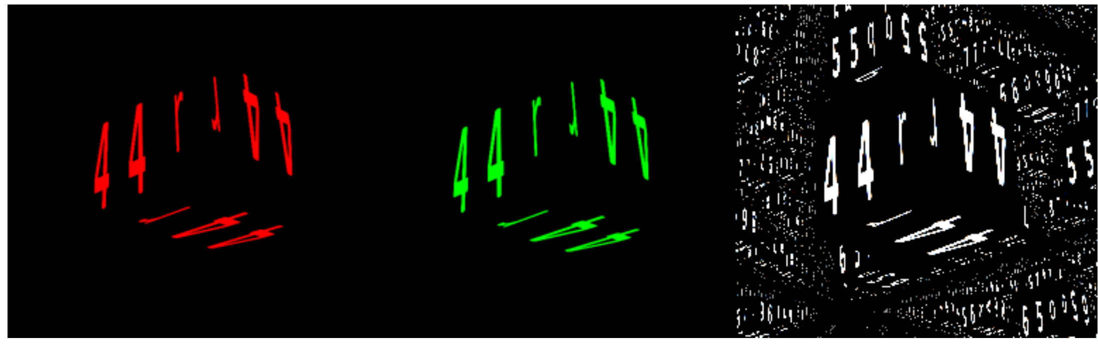

Figure 3.1: From left to right
Showing an initialized box protein, a field loaded box protein, and a place holder one

The same box with coordinates *4,4,r* is shown in the figure above. As discussed, each side of the box consists of 3 coordinate textures. Red colour of the coordinates means that the protein box is initialized, but doesn't have a field on it, while green means that it also has a field loaded. White coordinates mean that the box is uninitialized, and if the *x* key is pressed the visibility of these boxes can be turned on and off. Furthermore they are smaller than the other 2, which are the same size.

3.3 Modified Equations

The original equations are shown in Section 2.2, but since we wrote our whole program in C++, we could not directly implement these equations into our code including the neighbouring molecules interactions. Instead we had to use the differential equations with a chosen, small enough time interval *(dt)*. Changes in the molecules' dipole moment could be obtained, by rearranging the differential equations. [16].

Calculating the actual viscoelastic part of the dipole moment, from the previous state and its change during dt time

$$\mu_{ex}(t+dt) = \mu_{ex}(t) + d\mu_{ex}(t) = \mu_{ex}(t) + \left(\frac{E_x(t)}{C_{e1x}} - \frac{\mu_{ex}(t)}{C_{e1x}C_{e2x}}\right)dt \tag{10}$$

Getting the actual viscoplastic dipole moments, by rearranging the equations and instead of using the sign and abs functions, separating the different cases.

If $E_x(t) > \frac{\mu_{p1x}(t)}{C_{p2x}}$, then

$$\mu_{p1x}(t+dt) = \mu_{p1x}(t) + d\mu_{p1x}(t) = \mu_{p1x}(t) + \left(\frac{E_x(t)}{C_{p1x}} - \frac{\mu_{p1x}(t)}{C_{p1x}C_{p2x}}\right)dt. \tag{11}$$

If $E_x(t) \leq \frac{\mu_{p1x}(t)}{C_{p2x}}$, then

$$\mu_{p1x}(t+dt) = \mu_{p1x}(t). \tag{12}$$

And the same way,

If $E_x(t) > \frac{\mu_{p2x}(t)}{C_{p2x}}$, then

$$\mu_{p2x}(t+dt) = \mu_{p2x}(t) + d\mu_{p2x}(t) = \mu_{p2x}(t) + \left(\frac{E_x(t)}{C_{p1x}} - \frac{\mu_{p2x}(t)}{C_{p1x}C_{p2x}}\right)dt. \tag{13}$$

If $E_x(t) \leq \frac{\mu_{p2x}(t)}{C_{p2x}}$, then

$$\mu_{p2x}(t+dt) = \mu_{p2x}(t). \tag{14}$$

Finally, the actual overall dipole moment can be obtained by

$$\mu_x(t+dt) = \mu_{ex}(t+dt) + \mu_{p1x}(t+dt) + \mu_{p2x}(t+dt) + \mu_{0x} \tag{15}$$

The $C_{e1x}, C_{e2x}, C_{p1x}, C_{p2x}$ parameters are introduced and defined in Section 2.2. For dt, we used 0.01, because in the equations $dt = 0.01$ [s], represents $dt = 0.01$ [ps] in reality, and it was a small enough value, with which we could get correct results and the program worked as planned.

3.4 Explaining the Running of the Simulation

In this subsection, we discuss the Simulation part of the program in detail.

3.4.1 Applying the modified equations

In the previous subsection, we explain how the differential equations were changed in order to use them in a *C++* environment, which we wrote our simulation program in.

In general, we had to make the numerical solution of the differential equations, which means calculating the difference of the dipole moment for small enough time intervals, and adding them to the previous dipole value. Therefore we had to make two global variables, for the calculation. One was called dt representing the time interval, and the other was t, marking the actual time with which we can set the length of the simulation.

The last obstacle was calculating the electric field on a protein. For a constant field, it is easy, but the effect of the adjacent molecules' dipole moment was a problem, because it needed larger changes in the flow of the program. For fixed electric fields on the molecules one dipole variable would be enough for one protein, but for a changing field, its value being calculated in every step, every molecule needed at least two variables for storing its dipole moment. The reason behind this is very simple, yet not necessarily obvious. Let's say there are two neighbour molecules, call them A and B. At a given t_0 time A's dipole is X_0, and B's is Y_0. The simulation goes to the next step: $t_1 = t_0 + dt$, and we calculate A's dipole with the help of Y_0 and get X_1. For calculating Y_1 we use A's dipole, which had already changed to X_1, but we should use the dipole value from the previous time step (X_0). Thus B's dipole value would not be calculated exactly as it should be, and the problem becomes larger with every additional molecule. Hence we came up with the solution to separate the steps by their number. The steps with even numbers were called A, and steps with odd ones B, and according to this we created the variables *dipA* and *dipB* in the class, dedicated to the molecules, mentioned in Section 3.1. This way we store the dipole values calculated in the A steps from the neighbouring molecules' *dipB* values in *dipA* and vice versa.

3.4.2 Simulation Algorithm

After we figured out the equations and resolved the problems regarding their implementation in the program – as mentioned above – writing the algorithm was not a hard task. Taking everything in account, we came to the following algorithm,

Algorithm 1 Simulation Algorithm

1: **for** $t \leftarrow 0$ **to** desired time **do**
2:     **if** no. of step is even **then**
3:         **for all** initialized molecules **do**
4:             $dipA \leftarrow (dipB + d\mu\,(dt,$ neighbours' $dipB))$
5:         **end for**
6:     **else**
7:         **for all** initialized molecules **do**
8:             $dipB \leftarrow (dipA + d\mu\,(dt,$ neighbours' $dipA))$
9:         **end for**
10:     **end if**
11: **end for**

where everything is as discussed above, and $d\mu$ is the function to calculate the change in dipole moment, depending on the dt variable and adjacent molecules' dipole moment. Note that proteins are only used in the simulation if they have been initialized with or without an electric field. In order to shorten the simulation time, we first determine which proteins are initialized with a simple function.

3.4.3 User Inputs

In order to use the simulation algorithm on a protein structure we first need to define it. The *enter* variable, printed on the program window, needs to be in the *pressed* state to be able to initialize proteins without loaded fields, by pressing the *enter* key. If we want to have a field loaded (green coloured) protein, the *field* mode is needed, achieved by pressing the *t* key. Lastly if we want to delete an already initialized protein, the *delete* mode is needed, by pressing the *delete* key. If the *enter* variable is in either of these 3 states then the specifying of the protein's 3 coordinates is simply done by pressing the appropriate number or letter keys. After typing 3 coordinates the program will automatically perform the task respective to the *enter* variable.

The green coloured, field-loaded proteins are affected by the value printed on the program window as the *field* variable. This value (given in kcal/(molÅe)) can be changed using the *field magnitude* mode of the *enter* variable, by pressing the *i* key. After pressing it we can define the desired value using the number keys. If the first key pressed is 0, then it will be a negative value, if it's any other number key it will be a positive value. The second and third number keys pressed give the actual value of the field, so it can be set between -99 and 99 kcal/(molÅe). After we've specified the desired field value, the *i* key needs to be pressed again to set it, and this will also print it on the program window.

After defining the protein structure, we can start a simulation step of 50 [*ps*], by pressing the *r* key. Between each simulation step the configuration of the structure can be changed, meaning that we can redefine which proteins are field-loaded, and

the magnitude of the electric field. To start a new simulation from 0 time the program needs to be restarted.

For example the simulation of the 3 molecule structure mentioned in Section 4.1, can be done in the following way:

1. Initialize the 3 proteins by pressing the *enter* key, and typing the 9 coordinates one after another. We should now see 3 red coloured boxes on the canvas, with the appropriate coordinates.
2. Run the first simulation step by pressing the *r* key.
3. Press the *t* key, and type the coordinates of one of the proteins on the side, its colour should change to green.
4. Press the *i* key, type *103*, and press the *i* key again. Now the *field* variable should be equal to 3.
5. Run the second simulation step by pressing the *r* key.
6. Press the *enter* key, and type the coordinates of the field-loaded green molecule, its colour should change to red.
7. Run the third simulation step by pressing the *r* key.

Note that if the protein's coordinates are above 9, then the letter keys need to be used when specifying coordinates, and this can only be done if the *switch* variable is in *parameter* mode, further described in Section 3.7.

3.4.4 Exporting the data

The simulation is running and calculating each protein's dipole value in every step, but without exporting the data, we can't use the results, so we needed a method to export them. For that reason we wrote in the code a *StreamWriter* which is responsible for writing a file, with a given structure we had defined. The format is as follows: the exported file is a CSV, as in *Comma-separated values* file, thus we can store the values belonging to the same molecule in a column. Therefore the export file opened in *Microsoft Office Excel* [20], would look like a table, with each column containing a molecule's dipoles moments one under another for each time step.

Two more things were needed in the tables, made from the *.csv* files, a header showing which column belongs to which molecule, and putting the actual time values in the first column, to see how much time is needed for a steady state, and for plotting the dipole values respective to time.

To make such a file, the program had to cycle through the whole canvas, and write in the first line the coordinates of initialized proteins. After this we just had to print out the actual time in every step, and the values of the existing molecules' dipole moment, right after calculating them. The last thing was finishing every step with a line break, at the end of the simulation closing the file, and the exported data was waiting to be

analysed in an external file.

3.5 Explaining the Searching Algorithm

In this subsection we will go over the reasons the Searching Algorithm was created, and explain in detail how it works.

3.5.1 Reasons and goals

As we described in earlier sections of this documentation, there are some difficulties when trying to put together a molecular structure for a specific logic function. Obviously as it will be shown in Section 4, easier logic gates can be realized even without the help of the algorithm presented here, but for example a XOR gate already needed a lot of speculation and guessing to be put together. The reasons for this are that Dronpa proteins have a memory like quality, and react differently to different magnitudes of electric fields. These qualities make them very distinctive from transistors, thus they can't be approached with the same mind-set. Realizing simple logic gates, and building other logic functions using these gate structures is not an efficient method when using proteins. There are nearly infinite number of structures that realize the same logic function, and building logic functions using basic gates is nearly impossible to do, because of Dronpa's qualities. Furthermore if such logic functions happen to be found they will be a lot bigger and more complex than needed, meaning that finding the smallest structure that realizes a specific logic function can be very hard, and isn't intuitive.

Because of these reasons we realized that building logic functions out of proteins can't be done in a structured logical way, but rather using random methods to find the specific molecular structure for the required logic function. Out of all the structures that can achieve a specific logic function, the smallest one should be found and used. We will present our findings of how much simpler structures can be found using this algorithm in Section 4.

3.5.2 User Inputs

In the current version 4 inputs and 7 outputs can be set, but the algorithm is written in such a way that scaling can be done very easily. Virtually any number of outputs could be set, but for our tests we found 7 to be more than enough. The 4 inputs can't be modified, since they are the same for every logic function: 0000, 0001, . . . , 1111. So if all 4 inputs are used then a maximum of 16 bits can be achieved for every output.

Outputs of the logic function can be set by the user, when the *enter* variable is in *output* mode (further discussed in Section 3.7). In this mode the first number key pressed will be the number of the output channel. When specifying multiple output channels it needs to be done in an ascending order, from 1 to 7. After specifying the number of the output channel the output bits can be set, which can be either 0, 1 or 2,

the latter meaning that we do not care whether that bit is 0 or 1. Lastly after specifying the desired output bits, the number of which can be either 2, 4, 8 or 16 the *k* key needs to be pressed again to save those bits to the current output channel. When an output channel is specified the program will print the output bits, and the required number of outputs and inputs on the left side of the program window.

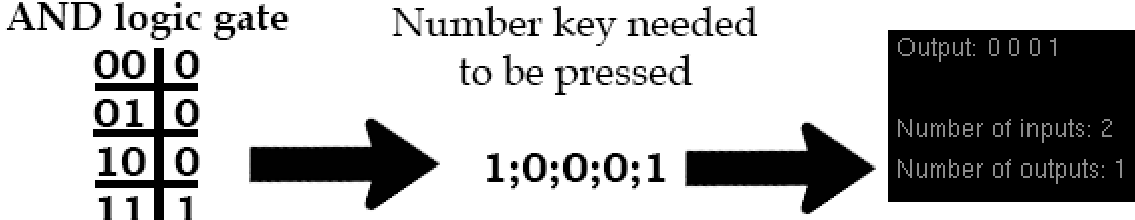

Figure 3.2: Example for logical inputs

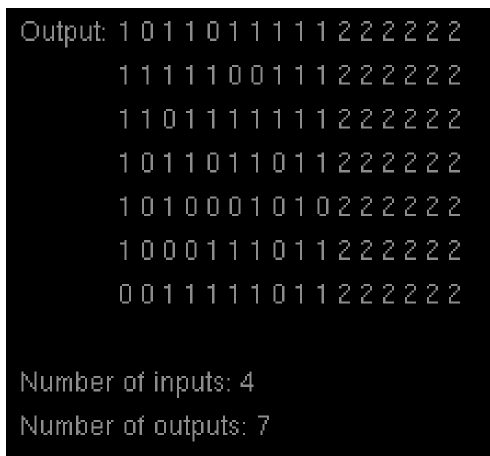

Figure 3.3: The output text of the 7 segment display logic function

3.5.3 Algorithm background and basic rules

Before we started writing the algorithm we set some basic rules to the type of structures it will look for, in order to make the structures as simple and close to transistor logic as possible.

For each input and output channel a different molecule will be used. Only input molecules can be directly affected by electric fields. Its value is between -10 and +10 kcal/(molÅe), and it can only be an integer.

The simulation part of the algorithm is run similar to earlier descriptions. The first step is running without any electric field. In the next step an electric field will be put on an input, and in the third step, the simulation will be run again without any electric field. Steps 2 and 3 will be repeated until an electric field was applied to all input

molecules. After the simulation part all the input and output proteins will be tested, whether they achieved the required logic bit. For each molecule an increase in its dipole at the end of the simulation compared to its dipole after the first step corresponds to logic 1, and a decrease to logic 0. These are the basic rules by which the algorithm tries to find a structure achieving the desired logic function.

3.5.4 Structure Search

The algorithm will first start with a minimum number of molecules needed for the logic function, meaning the *number of inputs* + *number of outputs*, and then if it doesn't find the satisfying structure it will move on to a higher number of proteins.

At each number of molecules there is a minimum number of different structures that need to be checked, all other structures being the same in function to these. The low amount of different structures is due to the fact that in the simulation we only take into account the 6 closest neighbours, and don't care about proteins placed diagonally.

Number of molecules	Minimum number of structures	Structures
3	1	
4	3	
5	4	

Figure 3.4: Minimally required structures

Theoretically for the algorithm to be perfect regarding both speed and precision it should check exactly, only these structures and no others. Unfortunately we couldn't find a method by which to find only exactly these structures. Since the run time of the algorithm is already tremendously huge because of other aspects we decided to leave out some structures, sacrificing a little on precision. This way the algorithm

won't always find the absolute minimum number of molecules needed to achieve a logic function. On the other hand it runs faster, and this way we found several logic functions during the time we had to run it.

The algorithm that finds separate structures will be now discussed.

The main logic behind it is that it takes 3 manually given molecules, and tries to add a 4th one in such a way that it satisfies the *minimally required structures*. The logic behind not finding the same (in function) structure twice is assigning the number of neighbours to each molecule in the structure. Then an algorithm takes 2 structures with the same molecule number, and makes an ordered list of the assigned numbers to each molecule. If the lists are identical then the 2 structures are same in function, thus only 1 is required to be checked further. After some tests we found that this algorithm while almost finding all the required structures, it leaves a little portion of them out, thus sacrificing on precision, but reducing run time.

Algorithm 2 Sorting Algorithm

Returns the number of neighbours for all the proteins of a specific structure
number of initialized proteins: ***n***
array of initialized protein's first index: ***i[n]****, second index:* ***j[n]****, third index:* ***k[n]***
array to contain the protein's number of neighbours: ***proteins[n]***

1: **for** $l \leftarrow 0$ **to** $n-1$ **do**
2: **for** $m \leftarrow 0$ **to** $n-1$ **do**
3: **if** $abs(i[m]-i[l]) + abs(j[m]-j[l]) + abs(k[m]-k[l]) = 1$ **then**
4: $proteins[m] \leftarrow proteins[m]+1$
5: **end if**
6: **end for**
7: **end for**
8: $swapped \leftarrow true$
9: **while** $swapped$ **do**
10: $swapped \leftarrow false$
11: **for** $j \leftarrow 0$ **to** $n-2$ **do**
12: **if** $proteins[j] < proteins[j+1]$ **then**
13: swap $proteins[j]$ and $proteins[j+1]$
14: $swapped \leftarrow true$
15: **end if**
16: **end for**
17: **end while**

If this algorithm returns the same proteins[] array for 2 separate structures, then the two structures are same in function.

The other part where precision is sacrificed is after finding all the 4 molecule structures, the algorithm will take the last one found, and it will move on to add a 5th

molecule using that 4 molecule structure, the same way that we described adding a 4th molecule to the 3 molecule structure took place. This iteration will continue adding more and more molecules until the desired structure is found. Since we only use 1 structure to generate the next, higher protein count structure, this is the main reason why the algorithm doesn't test all the minimally required structures.

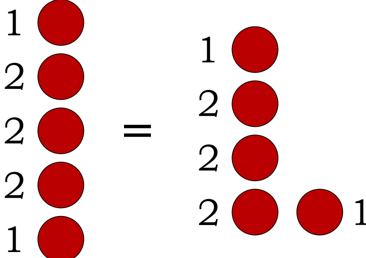

Figure 3.5: Showing equal (in function) protein structures

3.5.5 Detailed Search

We will now discuss how the detailed search takes place after finding a possible structure candidate.

When a candidate structure is found the *terteszt* function is started to continue the detailed search. First of all the algorithm will search all the different input molecule positions within the structure. For a 4 molecule structure with 2 input molecules given that means $3 + 2 + 1 = 6$ possible candidates. For each of these possible input molecule arrangements the algorithm will apply different electric fields to the two input molecules, meaning $21 * 21 = 441$ possible candidates, if the bounds for the electric field are -10 and 10 kcal/(molÅe).

Furthermore all of these configurations will be checked for all the bits given, at 2 input molecules this number is $2^2 = 4$. This is needed because 1 configuration can only be tested for 1 possible outcome at once, so all the rows of the logic function's logic table have to be tested separately. Two more checks are needed for the algorithm to be completely thorough. The first one we found while testing is that for applying the respective electric fields to the respective input molecules in switched order might lead to a different outcome, so a *for loop* for this aspect was also implemented. The other one that is needed is checking all the molecules that are not used as input molecules in the current configuration, as the possible output molecule (multiplied by the number of outputs channels).

Thus a specific structure and electric field – input molecule configuration will only be selected if all the rows of the logic function's logic table are satisfied by the selected output and input molecules dipole moment change as described above.

3.5.6 Finding the result

While the algorithm runs the graphical window isn't active, so one way of finding out whether the algorithm is done is by checking whether the window is active again. When the intended structure is found the algorithm will print the result in a *txt* file named *talalatok.txt*.

17 18 18
18 17 18
18 18 17
18 18 18
field 1: -10
field 1 applied to protein: 18 17 18
field 2: -10
field 2 applied to protein: 17 18 18
output protein: 18 18 17
field 1: 1
field 1 applied to protein: 18 17 18
field 2: -10
field 2 applied to protein: 17 18 18
output protein: 18 18 17
field 1: -10
field 1 applied to protein: 18 17 18
field 2: 1
field 2 applied to protein: 17 18 18
output protein: 18 18 17
field 1: 0
field 1 applied to protein: 18 17 18
field 2: 1
field 2 applied to protein: 17 18 18
output protein: 18 18 17

Figure 3.6: AND gate output

In Figure 3.5.6 we can see the output of the algorithm in *talaaltok.txt*, searching for an AND gate. The first rows show the coordinates of the proteins used in the structure (in this case 4 rows were needed). After that we can divide the rest of the text into blocks, each block representing one row of the logic functions logic table, the first block being, when the inputs are 00, the second when the inputs are 01 and so on. One block consists of the rows between the 2 nearest rows starting with field 1. In this example each block consists of 5 rows, and there are 4 blocks in total, but if more input or output channels are specified the *txt* file's structure will adapt dynamically.

Each block starts with the electric fields, and the proteins' coordinates on which those fields are applied. The electric fields are applied in the order stated in the specific

block, but once in a while this order might be switched in order to achieve the desired result on the output molecule. This property is not printed in the *txt* file, so we can only find out the true order of the fields applied when we manually test the structure and configurations. After the input proteins are listed, the output proteins are next. If more output channels are listed, than there will be no distinction, so again to find out which output protein corresponds to which output channel we have to check the structure and simulate it manually. We discuss such easily implementable features and others in Section 5.4.

3.6 Other Functions

In this subsection we will discuss further functions that we consider important to mention. These functions are connected to the simulation part of the program.

3.6.1 Save function

This function makes it easier to reuse protein structures. We implemented it because we realized that we have to make many simulations on the same structure, so if we could save it, and have it even after closing the program, it would help a lot. When in *parameter* mode (explained in Section 3.7), we can start this function by pressing the *s* key. It will save all the initialized protein's coordinates to a file called *strukt.csv*, as well as whether the initialized protein's had electric fields loaded on them or not. Furthermore it will save the current rotation, movement and zoom parameters to *params.dat*. This way if a specific structure is saved while it can be clearly visualized, we don't even have to rotate or move it when loading it again to see it well.

3.6.2 Load function

This function is the pair of the save function. When in *parameter* mode the function starts by pressing the *o* key. It will load the previously mentioned files, and initialize the proteins given, with electric fields if specified. Furthermore it will load the structure in exactly the same orientation and dimension that we saved it. In short if we press the *s* key, then close the program, reopen it and press the *o* key, the window will look exactly like when we closed it. An important note is that when initializing the proteins from the *strukt.csv* file the function won't delete any other proteins which were already initialized since starting the program.

These functions are a convenient method to store structures, and when a specific one is needed, we can just paste the *strukt.csv* and *params.dat* files of the saved structure into the program library.

3.7 User Manual

In this subsection we will go over the various keyboard and mouse inputs with which the program can be operated and navigated, as well as explain the various variables printed as text directly on the program window. Some functions or parameters mentioned here might be further discussed in another part of the documentation.

3.7.1 First start of the program and navigating the graphical interface

When first starting the program it loads all the empty, placeholder boxes for the molecules. Since loading so many 3D boxes with textures is very resource intensive, the first step after starting the program should be hitting the *x* key to hide them.

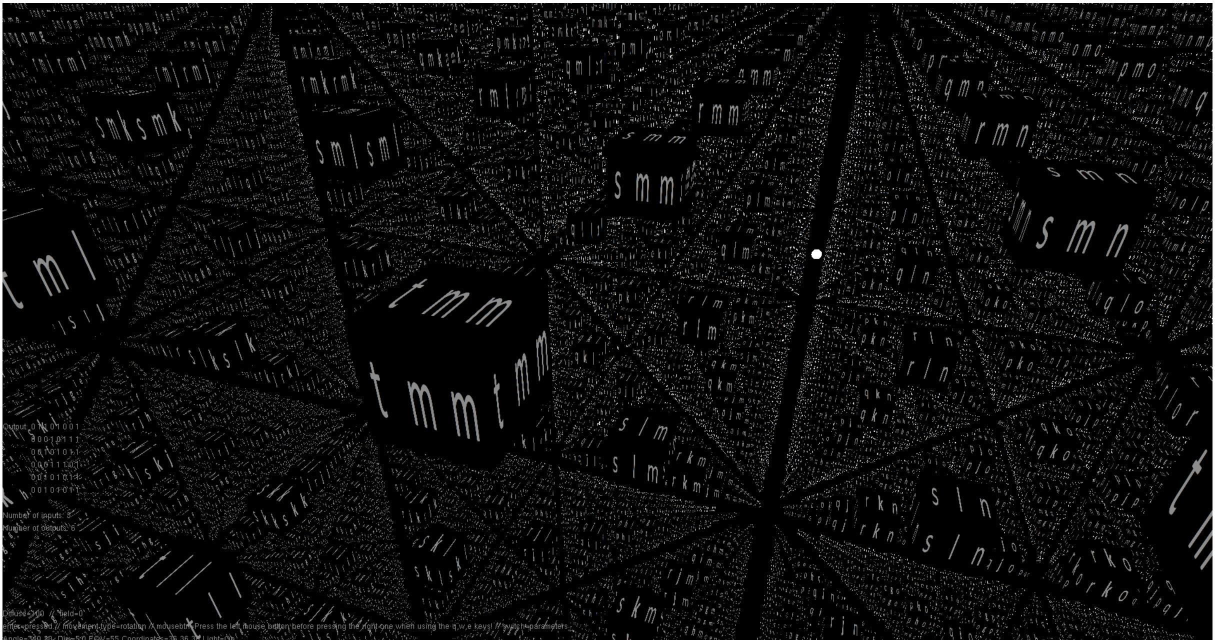

Figure 3.7: Starting program window

The structure can be rotated around and moved along 2 axes, both features being available by right clicking anywhere in the program window with the mouse. After the first start, before right clicking to use these functions it is required to left click anywhere in the window, since the algorithm needs a starting point for the mouse's coordinates. Obviously the rotation and linear displacement can't be done simultaneously, so switching between the two modes is required. To switch to rotation the *w* key needs to be pressed, and to switch to movement the *q* key needs to be pressed. Switching also requires a left click first to exit whichever mode the program is in. Doing this stops the rotation or movement function. Checking whether the program is in rotation or movement mode is made simpler by the *movement* variable being displayed on the

lower left corner. *movement=rotation* means that the program is in rotation mode, while *movement=movement* means that the movement function is on.

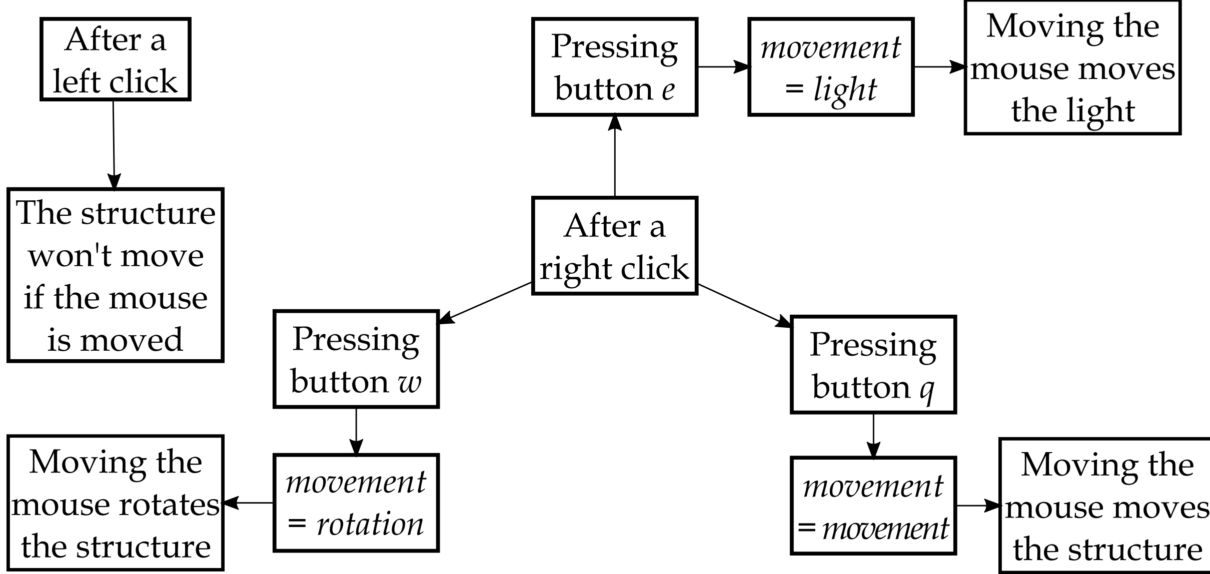

Figure 3.8: Graphical interface navigation flow chart

The state of the mouse (which mouse button was last clicked) is shown in the program window with the help of the *mousebtn* variable, its value either equal to *Left* or *Right*. Furthermore the 2 rotation angles are also displayed in the *Angle* variable, separated by a comma.

Besides rotating and moving a third minor option is also available using the mouse's movement. Pressing the *e* key will switch the mouse to rotate the small white light. The *movement* variable's value is set to *light* in this mode. Using the mouse wheel zooms in and out of the structure. The current zoom-level is displayed on the screen using the *Dim* variable. Going below 1.0 zoom is not advised because it will result in the disappearance of the structure.

3.7.2 Other keyboard inputs and parameters

Upon pressing the *Esc* key, the program will close. As mentioned before the *x* key toggles on and off the black placeholder boxes. The *l* key toggles on and off the light, acting as a light switch, this feature also being shown on screen, as the *Light* variable, which is either equal to *On* or *Off*. The light can also be elevated with the use of the *]* key, and lowered with the *[* key. With the *d* key the diffuse parameter of light can be increased if lowercase, or decreased if uppercase. The current state of the parameter is shown on the screen as the *Diffuse* variable, and its value ranges from 0 to 100. The *v* key toggles on and off the text printed directly on the program window. The field of

view can be decremented with the "-" key and incremented with the "+" key, its value being displayed on the screen as the FOV parameter.

Furthermore there are some keys that will start specific functions. These functions are presented with more detail in the *3.6 Other functions* section of the documentation. The *r* key will start the *futas* function which is responsible for starting the simulation. Starting the searching algorithm is done by pressing the *f* key, which calls the *fofuggveny* function. The *s* key is responsible for starting the *save* function, and the *o* key initiates the *load* function.

The *enter* variable further discussed in other parts of the documentation has 5 modes, each reachable by pressing different keys. In each mode the number keys are used for setting different inputs. The *field*, *pressed* and *delete* modes are used to specify molecule coordinates, while the use of the *field magnitude* mode is further described in Section 3.4, and the *output* mode is discussed in greater detail in Section 3.5.

key	**mode**
t	field
i	field magnitude
k	output
enter	pressed
delete	delete

Table 1: Modes of the *enter* variable

The current mode in which the variable resides is displayed in the program window as *enter=*current mode**. The magnitude of the field given by the user is also printed on the screen, as the *field* variable, its value is set to 0 as default.

```
Diffuse=100  //  field=0
enter=pressed // movement type=rotation // mousebtn=Left // switch=parameters
Angle=26,87  Dim=5.0 FOV=55 Coordinates=36 36 36 Light=On
```

Figure 3.9: Printed parameters

3.7.3 Switching between parameter and coordinate input

In order to be able to use the letters a-z for coordinate input as well as parameter input, the *space* key can be used. Its value is either *parameters* or *coordinates*, the former being the default when starting the program, represented by the *switch* variable printed on the program window. The coordinates typed are printed on the screen using the *Coordinates* variable, the default being 36, since we only use numbers from 0 to 35. Only the first 3 numbers given are printed, so if the *enter* variable's value is set to *output* mode, then further number inputs will not be shown.

The entering of coordinates is done with the help of number keys as well as letters, ranging from 0 to 35 in alphanumerical order. All the number keys can be used in *parameter* mode as well as *coordinates* mode, making easier the entering of molecule coordinates which only require numbers 0 to 9, and using the keyboard functions in *parameter* mode at the same time.

4 Simulations, results

In this section we will go over the various simulations which we ran, and explain the results in detail. We will show how dipole-dipole coupled logic can correlate to transistor logic, and how logic gates and functions can be achieved with the help of molecular structures.

4.1 Initial Molecule Structure

This subsection will focus on discussing the several molecular structures tested and found by us, before creating the *Searching Algorithm*. We think that it's important to mention these for the sake of comparison later on.

4.1.1 3 molecule structure

The structure simulated and discussed here is basically the same one as used in this paper [12]. We decided to first test the simulation part of the program with this simple structure, so we can compare our results with those in the paper [12]. We can easily check the validity of our results this way, and whether the equations work as intended in the program.

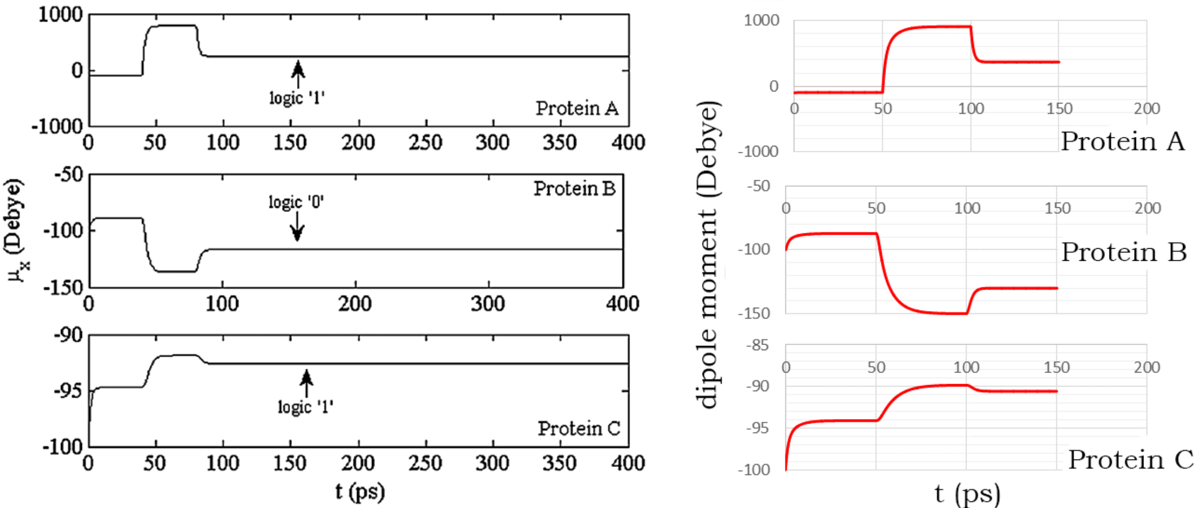

Figure 4.1: 3 molecule structure comparison

The simulation run is on 3 molecules placed along one axis linearly, at 7 nm from each other. The base dipole value is -100 Debyes as always, and we chose 50 ps as time step instead of 40, the reason for this being, that the longer the simulation runs, the closer the molecules will get to their final dipole value. In the first step no external electric field is applied, so the molecules only affect themselves. In the second step an

electric field of 3 kcal/(molÅe) is applied to protein A, then in the final step the electric field is stopped, meaning that no external field is applied.

As seen in Figure 4.1 protein A reaches a state of logic 1, protein B is equal to logic 0, while protein C reaches logic 1 as well, thus showing that the inverter gate is realized simply by putting 2 molecules next to each other, and loading an electric field on one of them, as further described in this paper [12]. The simulation result from the paper is very similar to our result, the slight differences are there because of the different computing algorithm, and the differential equations used [16]. We conclude that the simulation part of the program works as intended.

4.1.2 4 molecule structure

This 4 molecule structure discussed below is another example of comparison between the results from the paper mentioned before, and our simulation program, also showing how the majority gate could be realized [12].

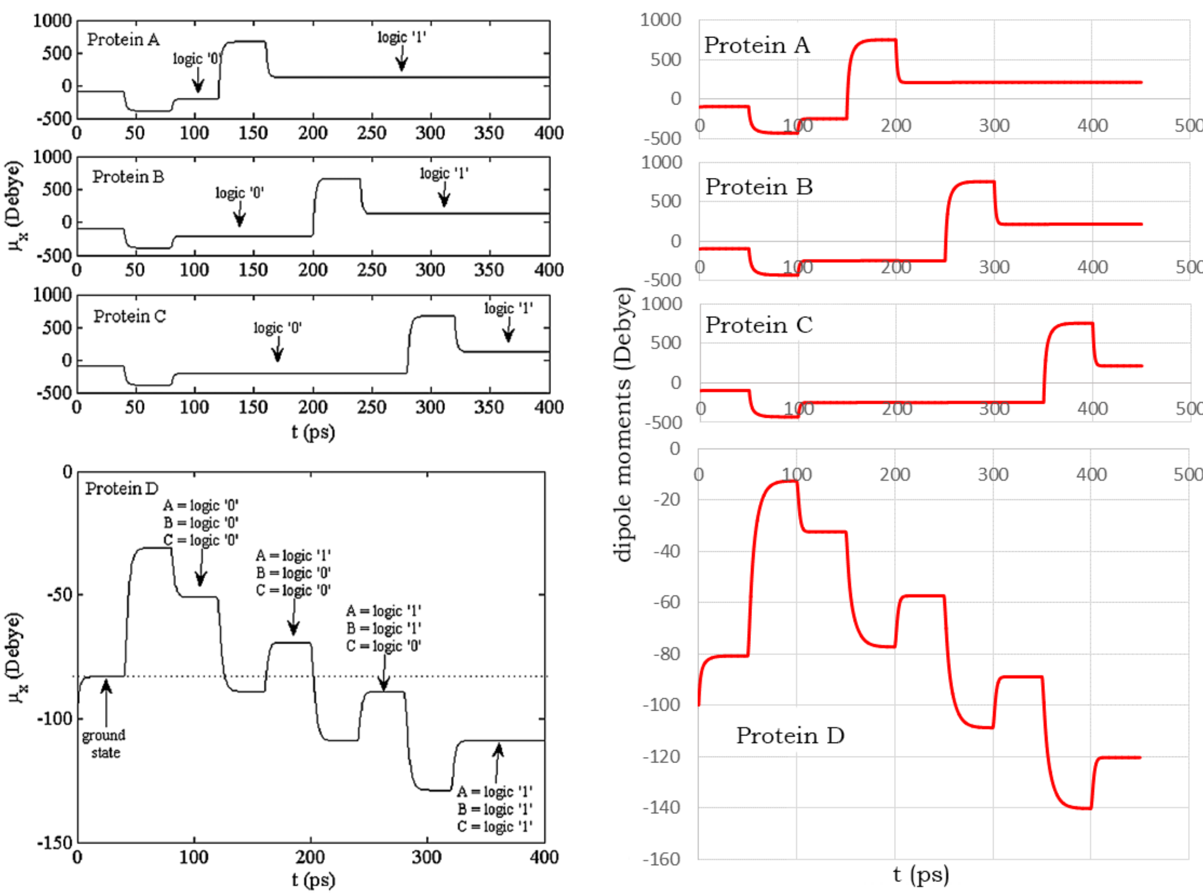

Figure 4.2: 4 molecule structure comparison

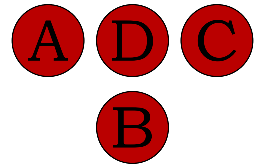

Figure 4.3: The examined 4 molecule structure

The 4 molecules are placed next to each other as shown in Figure 4.3, protein D being in the center and the other 3 around it. A, B and C are the input proteins, to which we load the same electric field corresponding of -1 kcal/(molÅe) to set the output protein D to logic 1. After that an electric field equivalent of 3 kcal/(molÅe) is applied to each input molecule consecutively. Thus the output protein reaches logic 0 after 2 electric fields have been applied, realizing the majority gate. As seen in Figure 4.2 the simulation results are very similar, achieving the same kind of functionality: the majority gate, as further described in [12].

Using as example the above structure and results we can further show how the NOR and NAND gate can be realized. If we leave out the step where we apply an electric field to protein C we will have a NAND gate, protein A and B being the inputs, and protein D the output. When both input proteins are set to logic 0 in the first step, the output reaches logic 1. After that it will only switch to logic 0 if both input proteins are set to logic 1, thus achieving the NAND logic table.

The NOR logic gate can be accomplished by taking the above model and regarding protein B and C as inputs, and protein D as output. For this configuration we need to apply an electric field to protein A before, for the gate to be functional. Same as in the previous example if both input proteins are set to logic 0, the output is logic 1. After that setting either protein to logic 1 (or both of them), will put the output protein to logic 0, thus accomplishing the NOR logic table.

Using these structures the AND and OR gate is also easily achievable. Since the AND gate is the inverse of the NAND gate, and inverting is done by simply transmitting the signal from one protein to another, if we put a protein next to the output of the NAND gate, and we consider that protein as ouput protein, we will have and AND gate. The OR gate is similarly derived from the NOR gate.

Obviously these configurations only work if at the start of the simulation all the proteins start with their base dipole value of -100 Debyes, and the respective electric fields are applied using the aforementioned values. Since Dronpa proteins are linearly sensitive to electric fields, using a greatly higher or lower value might make the molecular structure not function as the logic gate it is intended to achieve.

4.1.3 The XOR gate

In this sub-section we will go over how a slightly more complex gate, the XOR gate can be accomplished trying to use the same, building it from simpler gates logic, as used

in transistor gate structures.

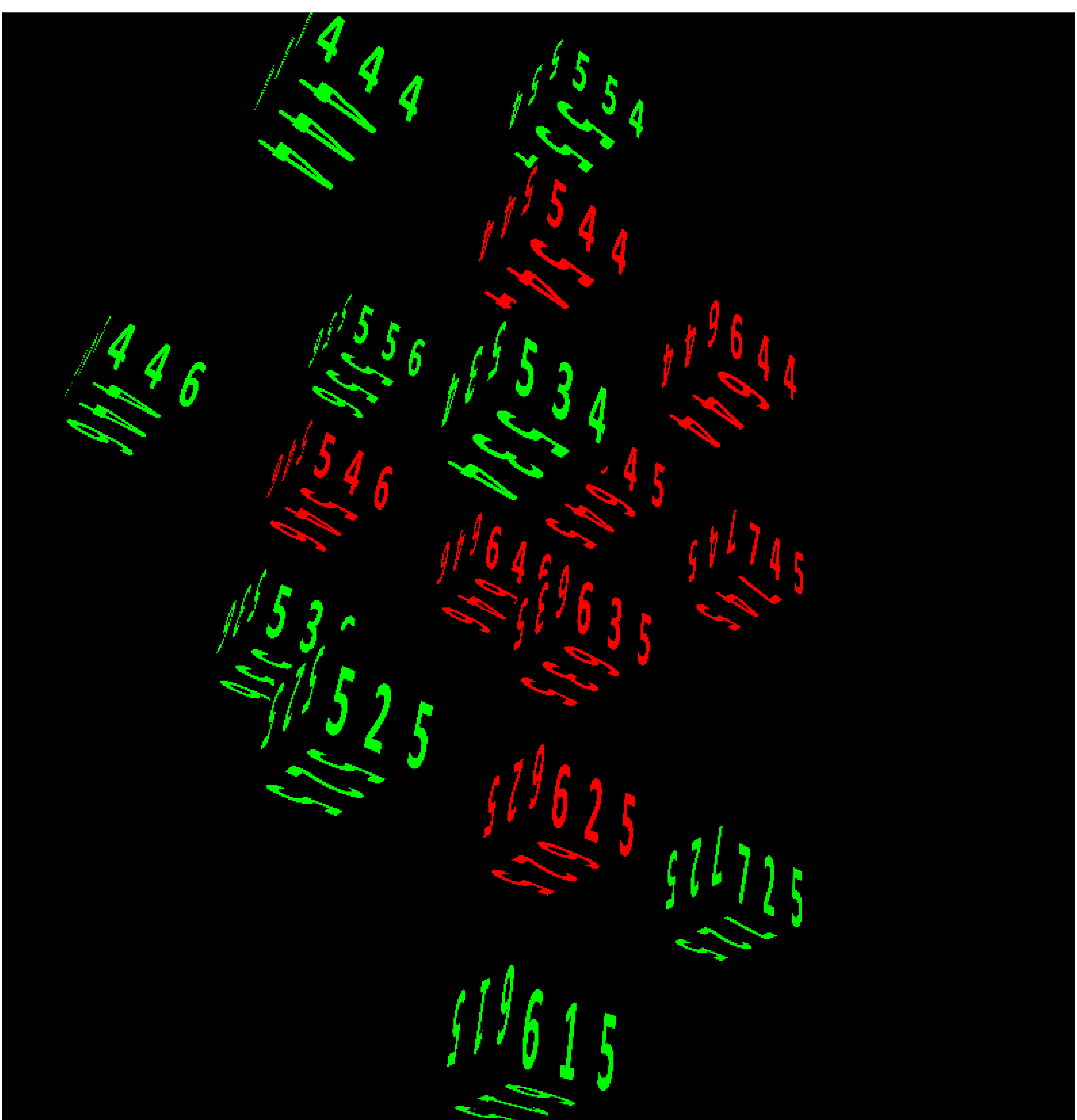

Figure 4.4: XOR gate structure

When building the XOR gate from simpler gates, we use 2 AND gates and 1 OR gate, assuming that we have both the base and negated form of the 2 inputs. As shown in the previous subsection we use 5 molecule structures for both the AND and OR gates. The first AND gate consists of molecules (444, 554, 544, 534, 644), the second one (446, 556, 546, 536, 646), and the OR gate (635, 646, 644, 645, 745). As you can see some

proteins are used both as outputs of AND gates and inputs of the OR gate. In order for the OR gate to work we need its third input (which isn't the output of an AND gate), to be set to logic 1. This though is not that simple since, it can't have any logic 1 dipole moment value, it has to be exactly in the order of magnitude of the other inputs. Using this logic we arrive to the conclusion that in order to be the same dipole value as other inputs, the third input needs to be the output of a third AND gate. Thus the structure has 9 inputs in total (inputs of the 3 AND gates), shown with green in Figure 4.4, and 1 output, the protein 745.

The structure works similarly to AND and OR structures, the first step being applying a field of -5 kcal/(molÅe) to all inputs, setting them to 0, also setting the output protein to 0. After that we found that the structure works only if the input proteins are set to logic 1 when needed with a specific electric field value of 23 kcal/(molÅe). We divide the simulation into 2 parts after setting all the inputs to logic 0, the first being when the output is 0, and the second being when the output is 1. In both cases for the OR gate to work, we set 2 inputs of the third AND gate to logic 1. In addition to this, for the output to be 0 we set 1 input of the first and second AND gate to logic 1. For the output to be logic 1, we set 2 inputs of the first AND gate to logic 1, leaving the inputs of the second AND gate in the initial logic 0 state. Since it makes the problem much more complex using only the base signal of the inputs we assume that the negated signal is available. The A input of the XOR gate is applied to protein 536, its negated signal to protein 444. The B input is applied to protein 534, its negated to protein 446. The output protein is 745.

A input	**B input**	**Output**
logic 0 2nd AND gate protein 536=logic 0 1st AND gate protein 444=logic 1	logic 0 1st AND gate protein 534=logic 0 2nd AND gate protein 446=logic 1	logic 0
logic 0 2nd AND gate protein 536=logic 0 1st AND gate protein 444=logic 1	logic 1 1st AND gate protein 534=logic 1 2nd AND gate protein 446=logic 0	logic 1
logic 1 2nd AND gate protein 536=logic 1 1st AND gate protein 444=logic 0	logic 0 1st AND gate protein 534=logic 0 2nd AND gate protein 446=logic 1	logic 1
logic 1 2nd AND gate protein 536=logic 1 1st AND gate protein 444=logic 0	logic 1 1st AND gate protein 534=logic 1 2nd AND gate protein 446=logic 0	logic 0

Table 2: XOR gate extended logic table

As shown at other gates we can get the XNOR gate simply by adding a molecule next to the output of the XOR gate, and considering that protein as the output of the same structure configuration.

4.1.4 Resetting a protein's dipole moment

An important topic to discuss is the memory aspect of Dronpa proteins, and how electric fields can be used to reset their dipole moment value.

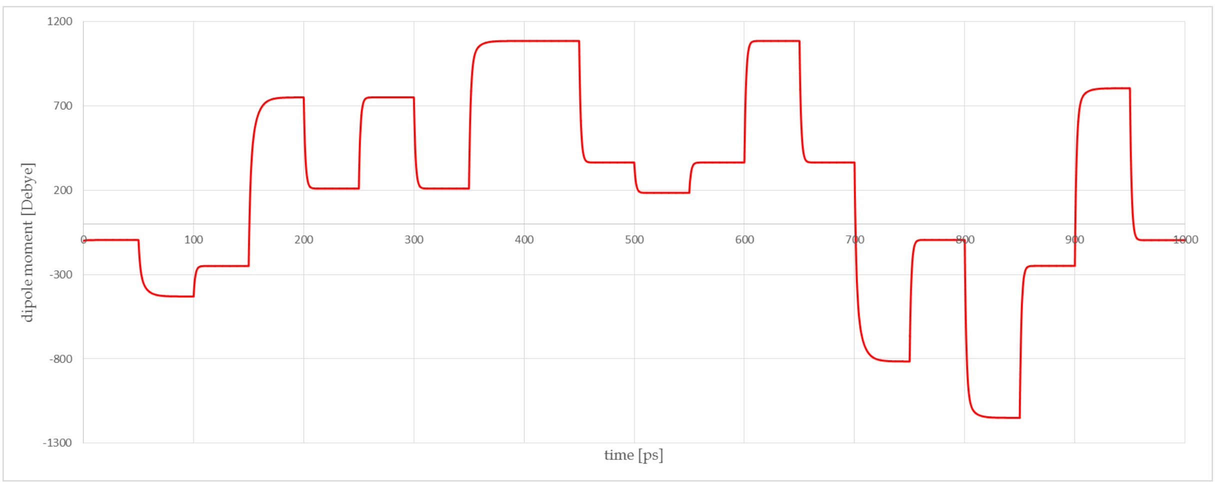

Figure 4.5: Resetting

As shown in the figure above, Dronpa has some special properties, that make it very different from transistors. Because of its nature, we found by studying the equations and testing that after applying an electric field, if we try to apply another electric field with an absolute value equal or less (if the electric field's direction is opposite, it has to be strictly less) than the previous field's value, the protein will go back to the same dipole value, meaning that the protein has a dipole saturation property. This is demonstrated on the graph above, when we first apply a field of 3 kcal/(molÅe), setting the molecule to logic 1. After that we applied the same electric field, and the protein's dipole value didn't change. Continuing we applied an electric field of 4 kcal/(molÅe), which being greater than 3 finally changes the dipole moment value. After that, to prove that applying an electric field with opposite direction, but less than the previous, we applied an electric field of -1 kcal/(molÅe), still not changing the dipole moment.

An important thing to note here is, that it isn't exactly the previous field which we have to compare to when speaking about saturation, but rather the field with the greatest absolute value among the previous fields applied after reaching a specific saturation state. To demonstrate this after applying the -1 kcal/(molÅe) electric field, which didn't change the saturation state, we applied a 4 kcal/(molÅe) electric field, which because it's equal to the electric field, which put the molecule in the current saturation state doesn't change the dipole moment.

Finally we show that by applying an electric field in the opposite direction but equal to the previous biggest field, we can reset the protein's dipole moment value to its starting value of -100. In this case that meant applying a -4 kcal/(molÅe) electric field, reaching the starting dipole state by the 800 [*ps*] time step. We demonstrate this

property again after that, with switched direction electric fields, but of equal value, again resetting the protein's dipole moment.

Obviously this resetting will only work perfectly if no other molecules are nearby affecting the protein which we want to reset. In the above simulations we actually placed several proteins next to the one which we tested, showing that even if there are some molecules nearby, it won't change anything visually. Although if we check the exact values we can see that the dipole moment doesn't go back to exactly the same value, in this case being between -1 to 1 Debye difference. Again this difference depends a lot on what dipole moment the nearby molecules have, thus making this resetting method very hard to be exact, in such cases.

4.2 Logic Gates as found by the *Searching Algorithm*

This subsection will be dedicated to discuss the 6 basic logic gates, and the protein structures that the *Searching Algorithm* found for them. The inverter gate can be easily achieved by putting 2 molecules next to each other, because of the properties of Dronpa, as earlier described in the documentation, so we won't include it here.

We will present the results as well as the manual simulations of the results in a comprehensive table. In the results of the manual simulation, we subtract the dipole value after the very first step when no electric fields are applied from the end state dipole moment, thus if this number is positive than the protein is set to logic 1, otherwise it's logic 0. We give this number as the *dipole* variable in the tables. For each simulation we gave a specific electric field boundary value, for which the searching algorithm tests the structure, so it usually first finds the configuration with the minimum electric field, because of the nature of the algorithm, but obviously closer to 0 electric fields could also achieve similar results. Furthermore the reason behind the coordinates of the proteins being found around 18 is because the algorithm starts looking for protein structures at the centre of the 36x36x36 box.

4.2.1 AND Gate

The Searching Algorithm found a structure consisting of 4 proteins, (16,18,18), (17,17,18), (17,18,18), (18,18,18) respectively. Of these the first two are the input proteins, and the last one is the output protein.

B input (16,18,18)	A input (17,17,18)	Output (18,18,18)
3rd step: -3 field applied dipole= -462,463 logic 0	2nd step: -3 field applied dipole= -461,479 logic 0	dipole= -3,530 logic 0
3rd step: -3 field applied dipole= -459,056 logic 0	2nd step: 1 field applied dipole= 152,083 logic 1	dipole= -0,895 logic 0
3rd step: 1 field applied dipole= 150,274 logic 1	2nd step: -3 field applied dipole= -460,116 logic 0	dipole= -1,5734 logic 0
3rd step: 1 field applied dipole= 153,682 logic 1	2nd step: No field applied dipole= 0,704 logic 1	dipole= 0,704 logic 1

Table 3: AND Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.2.2 OR Gate

The *Searching Algorithm* found a structure consisting of 4 proteins, (16,18,18), (17,17,18), (17,18,18), (18,18,18) respectively. Of these the first two are the input proteins, and the last one is the output protein.

An interesting thing to note here is that both the AND and OR gates found are very similar to the structures found manually, discussed in Section 4.1.

B input (16,18,18)	A input (17,17,18)	Output (18,18,18)
3rd step: -5 field applied dipole= -772,476 logic 0	2nd step: -5 field applied dipole= -770,833 logic 0	dipole= -7,587 logic 0
3rd step: -3 field applied dipole= -456,895 logic 0	2nd step: 2 field applied dipole= 306,557 logic 1	dipole= 0,601 logic 1
3rd step: 5 field applied dipole= 766,537 logic 1	2nd step: -3 field applied dipole= -456,894 logic 0	dipole= 1,648 logic 1
3rd step: 1 field applied dipole= 153,682 logic 1	2nd step: No field applied dipole= 0,704 logic 1	dipole= 0,704 logic 1

Table 4: OR Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.2.3 NAND Gate

The structure found for the NAND Gate consists of 3 proteins, (16,18,18), (17,18,18), (18,18,18) respectively. The middle one is the output protein, and the other two are the input proteins.

B input (16,18,18)	**A input (18,18,18)**	**Output (17,18,18)**
3rd step: -5 field applied dipole= -772,233 logic 0	2nd step: -5 field applied dipole= -770,597 logic 0	dipole= 132,996 logic 1
3rd step: -5 field applied dipole= -767,115 logic 0	2nd step: 1 field applied dipole= 150,751 logic 1	dipole= 65,442 logic 1
3rd step: 1 field applied dipole= 147,259 logic 1	2nd step: -5 field applied dipole= -768,470 logic 0	dipole= 75,080 logic 1
3rd step: 1 field applied dipole= 153,882 logic 1	2nd step: No field applied dipole= 0,904 logic 1	dipole= -10,272 logic 0

Table 5: NAND Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.2.4 NOR Gate

The structure found for the NOR Gate is exactly the same as the NAND Gate. Again the NOR and NAND gate structures are very similar to the ones found by manual simulation, presented in 4.1 Initial molecule structures.

B input (16,18,18)	A input (18,18,18)	Output (17,18,18)
3rd step: -5 field applied dipole= -772,233 logic 0	2nd step: -5 field applied dipole= -770,597 logic 0	dipole= 132,996 logic 1
3rd step: -2 field applied dipole= -302,123 logic 0	2nd step: 2 field applied dipole= 307,347 logic 1	dipole= -4,170 logic 0
3rd step: 5 field applied dipole= 766,736 logic 1	2nd step: -3 field applied dipole= -456,487 logic 0	dipole= -12,046 logic 0
3rd step: 1 field applied dipole= 153,882 logic 1	2nd step: No field applied dipole= 0,904 logic 1	dipole= -10,272 logic 0

Table 6: NOR Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.2.5 XOR Gate

The structure found for the XOR Gate is a little bit more complex than the previous ones, but still far simpler than the one which we found manually, described in Section 4.1. It consists of only 6 molecules (15 18 18), (16 17 18), (16 18 18), (17 17 18), (17 18 18), (18 18 18), respectively.

B input (16,17,18)	A input (17,17,18)	Output (16,18,18)
3rd step: -1 field applied dipole= -0,922 logic 0	2nd step: -7 field applied dipole= -1077,57 logic 0	dipole= -3,834 logic 0
3rd step: -10 field applied dipole= -1529,91 logic 0	2nd step: -1 field applied dipole= 11,142 logic 1	dipole= 160,988 logic 1
2nd step: -1 field applied dipole= 11,142 logic 1	3rd step: -10 field applied dipole= -1529,91 logic 0	dipole= 2,839 logic 1
3rd step: 1 field applied dipole= 138,712 logic 1	2nd step: 1 field applied dipole= 144,740 logic 1	dipole= -5,333 logic 0

Table 7: XOR Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.2.6 XNOR Gate

The structure found for the XNOR Gate consists of 4 molecules (15 18 18), (16 18 18), (17 18 18), (18 18 18), respectively. Again this structure is also much smaller than the one built manually.

B input (16,18,18)	A input (17,18,18)	Output (15,18,18)
3rd step: -10 field applied dipole= -1337,59 logic 0	2nd step: -10 field applied dipole= -1401,69 logic 0	dipole= 142,792 logic 1
2nd step: 1 field applied dipole= -0,806 logic 0	3rd step: 10 field applied dipole= 1539,833 logic 1	dipole= -5,618 logic 0
3rd step: -1 field applied dipole= 62,666 logic 1	2nd step: -10 field applied dipole= -1546,57 logic 0	dipole= -3,019 logic 0
3rd step: 1 field applied dipole= 11,597 logic 1	2nd step: 7 field applied dipole= 1084,195 logic 1	dipole= 1,142 logic 1

Table 8: XNOR Gate table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.3 Logic Functions

In this sub-section we will cover two more complex logic functions found by the *Searching Algorithm*. The functions are the half-adder and the half-subtractor. Other, more complex logic functions were also tried, but since the algorithm scales exponentially with more inputs specified, we didn't have enough time or resources for it to finish.

4.3.1 The Half-adder

A structure of 8 proteins was found suitable by the *Searching Algorithm*, given below. With green are marked the inputs, and red represents the 2 outputs.

	15 17 18	16 17 18	17 17 18	
14 18 18	15 18 18	16 18 18	17 18 18	18 18 18

Figure 4.6: Half-adder structure

B input (15,17,18)	**A input (16,17,18)**	**Output S (15,18,18)**	**Output C (17,18,18)**
3rd step: -1 field appl. dipole= -0,272 logic 0	2nd step: -7 field appl. dipole= -1086,2 logic 0	dipole= -3,93 logic 0	dipole= -10,21 logic 0
3rd step: -10 field appl. dipole= -1530,3 logic 0	2nd step: -1 field appl. dipole= 12,748 logic 1	dipole= 161,06 logic 1	dipole= -0,444 logic 0
2nd step: -1 field appl. dipole= 11,728 logic 1	3rd step: -10 field appl. dipole= -1540,3 logic 0	dipole= 2,758 logic 1	dipole= -17,9 logic 0
3rd step: 1 field appl. dipole= 138,174 logic 1	2nd step: 1 field appl. dipole= 148,991 logic 1	dipole= -5,15 logic 0	dipole= 1,754 logic 1

Table 9: Half-adder table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

4.3.2 The Half-subtractor

The structure of the half-subtractor is exactly the same as the half-adder, the only difference being that the input and output proteins are chosen differently.

15 17 18   16 17 18   17 17 18
14 18 18   15 18 18   16 18 18   17 18 18   18 18 18

Figure 4.7: Half-subtractor structure

B input (15,17,18)	**A input (15,18,18)**	**Output D (16,17,18)**	**Output B (17,18,18)**
3rd step: -1 field appl. dipole= -0,282 logic 0	2nd step: -7 field appl. dipole= -1085,9 logic 0	dipole= -4,09 logic 0	dipole= -4,245 logic 0
3rd step: -10 field appl. dipole= -1530,3 logic 0	2nd step: -1 field appl. dipole= 12,942 logic 1	dipole= 160,89 logic 1	dipole= 0,62 logic 1
2nd step: -1 field appl. dipole= 11,715 logic 1	3rd step: -10 field appl. dipole= -1539,9 logic 0	dipole= 2,613 logic 1	dipole= -5,886 logic 0
2nd step: 3 field appl. dipole= 458,219 logic 1	3rd step: 1 field appl. dipole= 99,396 logic 1	dipole= -40,69 logic 0	dipole= -0,137 logic 0

Table 10: Half-subtractor table
All electric field values are given in kcal/(molÅe)
All dipole moment values are given in Debyes

5 Discussion

During our testing and analysing we encountered a handful of problems and issues, of which several were resolved. In this chapter we will go over what problems the simulation and the searching algorithm might have, as well as go over some logical and theoretical challenges. Solutions to these problems and further improvements to the program are also discussed. Furthermore we talk about future plans as well.

5.1 Concept related issues

In this subsection we describe some problems that we've encountered when we tried manually building protein structures, and other related comments. While theoretical simulations might overcome them, they definitely pose a challenge in practice.

5.1.1 Signal transfer

One of the reasons why it's hard to create structures of proteins using similar logic to transistor circuits is that the signal transferred from one protein to another falls greatly after each jump. The signal is decreased by roughly 99% each time it jumps from one protein to another. This poses 2 problems, the first being that after several jumps the signal would be hard to detect, and other protein's noise could alter the function of the structure. Thus some kind of amplifier or repeater has to be used in order to maintain a big enough signal value when implementing such structures in practice. The other issue is that we can't predict exactly how much the signal falls when creating molecular logic functions. This is one of the reasons why we can't just take the output of one structure, and use it as the input for the next one.

5.1.2 Reset problem

As mentioned earlier in Section 4.1.4, the resetting of a protein's dipole moment can be done exactly only if there are no neighbours affecting it. The reset method still works if neighbours are present, but it does not set the dipole moment of the protein to its initial value. This is a problem because if we want to reuse a molecular structure for a specific logic function, the proteins have to possess their initial properties. Of course for simulation purposes by simply restarting our program this can be achieved, but in practice a resetting method has to be used if we want to re-use these structures.

5.1.3 Logic

We used a specific logic when approaching protein structures. We tried to make it work as close to transistor logic as possible, but since Dronpa is a special protein, many types of logical methods can be used, and other approaches might prove more useful than ours. Furthermore other proteins with better polarizability properties could also

be used. The main issue with our approach is that for each logic function a specific molecular structure has to be found, and we can't build complex functions from smaller structures, because of the nature of the protein. As the complexity increases so does the running time of the *Searching Algorithm*, so it is only able to find smaller structures within a reasonable period of time. If we want to implement protein logic into practice, we have to be able to build complex structures following a specific method or logic to achieve computing architectures.

5.2 Comments on the program

In this subsection we discuss some topics related to the program, that might need improving.

5.2.1 Neighbours

The proteins form a 3D grid and apart from those situated by the boundaries, every molecule has 6 neighbours (up, down, right, left, in front, behind). As mentioned before, while calculating the dipole moment of a chosen protein, only these are taken into consideration, although other proteins which are farther away may also have an effect on its dipole moment. We used this simplification because of 2 reasons. Firstly, it would greatly increase the simulation time, and secondly the effect of a protein on another one is reduced in proportion to the third power of the distance between them. If greater precision is needed, we can easily implement the use of more neighbouring proteins into the program, but we decided that the 6 closest neighbours are enough to achieve representative results. In practice it would be dangerous for all proteins to affect each other, because the structure might not work as intended, so some kind of insulation should be used to shield the chosen molecule or structure from unwanted effects.

5.2.2 Searching Algorithm related problems

As we mentioned before the *Searching Algorithm* has a couple of issues. The main problem is that the algorithm doesn't find exactly the minimum number of proteins needed to perform the tasks of a certain logic function. We present some ways of improvement in this regard in Section 5.4. As in other simulation processes, the main factor to determine the running time is precision. The more precise we want to be, the more time the algorithm will take to finish.

Another issue is with the scalability of the algorithm. While adding more outputs doesn't significantly increase the running time, more inputs certainly does. For each consecutive input the algorithm has to run roughly, *the number of different electric fields* times more. Thus with a boundary of -10 to 10 kcal/(molÅe), each consecutive input multiplies the checking time of one structure by about 20. We can play around with this boundary, but setting it lower might result in more proteins needed for the same logic

function, since the higher the boundary is, the more possibilities there are to find the logic function for the same structure. Again we arrive at a decision between running time and minimising the number of proteins needed for the same logic function. Even though lowering the boundary decreases the running time, with more and more inputs and outputs added, because of the nature of the algorithm, it will eventually reach running times of years for complex logic functions.

We conclude that this algorithm is only suitable to search for smaller structure, and less complex logic functions. A different approach for the entire problem would be needed if finding many input, many output logic functions is the goal.

5.3 Improving the Simulation

In this subsection we discuss some improvements that we have in mind for the program, regarding an algorithmic and an optimizing aspect as well. We also mention some modules with which the program could be expanded.

5.3.1 Programming improvements

First of all some graphical improvements could be made to the program. An issue right now is when all 46 656 boxes are loaded the program only runs with about 1 frame per second. Since the coordinates are in order we don't really need to see all these boxes in order to build structures, so we don't consider this a major issue. Still it could be improved by using multiple cpu cores or using the gpu more heavily to handle the drawing of so many boxes. We tested the program for up to 1000 boxes, and it still ran with 60 fps.

Currently the program can only be operated with keyboard inputs, but there are libraries for *freeglut* with which textboxes and buttons could be implemented. This would make the program more user friendly, making it easier to operate and navigate.

Furthermore the program's scalability could be improved by specifying variables that the user can modify. For example the number of boxes/proteins that can be used, or the precision with which the *Searching Algorithm* operates, in order to decrease running time. The number of steps used in the simulation could also be reduced, or optimized in such a way that the algorithm adapts to the slope of the dipole moment function. If there aren't any abrupt changes less integration steps can give a pretty exact result as well. Lastly when the simulation runs instead of writing to a file at each step, we could store the simulation results in variables, and only write them out when the user needs them. These options and improvements are beneficial since they could further cut down the running time of the simulation and the *Searching Algorithm*.

Currently there is an animation function in the program but it isn't used. With animation we could make visual descriptions of how the signals are transferred through the protein structures. We could also show how the protein's size changes when it is affected by an electric field. With animation we could simulate the real world behaviour

of the proteins when interacting with each other, using simplified equations, in order to be able to run it in real time.

5.3.2 Further modules

Dronpa among other properties is photoswitchable [11]. This means that with well defined electromagnetic radiation frequencies the protein can be switched between 2 forms [21]. Digital signal propagation and basic gates can be realized as well. The positives of photon-coupling are that it is more powerful, and it depends less on distance that coulomb-coupling. Adding a photon-coupling module to our program wouldn't be a complicated task, since we already have the graphical interface. The equations describing the behaviour of Dronpa when affected by different radiation frequencies could be implemented fairly easily. All the current framework for the simulation can be reused, with some minor alterations, thus the program could also be usable for photon-coupling simulations.

The mathematical model which we implemented in the program and based the differential equations on is not specific to Dronpa. It can be used with some alterations to simulate other proteins as well, which have similar or even better qualities than Dronpa. This means that the framework and software we developed can be used for several proteins, and the only way in that it is specific to Dronpa is the constants and actual equations used.

5.4 Improving the Searching Algorithm

As mentioned in Section 3.5.4, the program does not find every possible molecular structure which is needed. In this section we are going to discuss ways for improvement, as well as cover some other parts where the searching algorithm could be further improved.

5.4.1 Improving the structure finding algorithm

We know that the searching algorithm has flaws as mentioned in Section 3.5.4, so we decided to run some tests, and improve our building algorithm, when the simulation works as expected. By the time we finished several test runs, we have encountered another problem, which was not with the program, but our limitation of resources. The simulation time was too long, even with the missing structures, thus we decided to leave the program the way it is for the time being, because even if it doesn't test every possibility, it still did found the basic logic gates.

However even if we did not implement the improved structure-building algorithm, we still managed to come up with a solution to find every possible molecular structure. We made the choice not to complicate the existing problem with other functions, but make a separate smaller program, its only purpose being, to look for structures. This way it would be even more convenient, because the building needs to be done only

once, and not each time a simulation is run. The second program finds every structure, writes them out in a data file, thus the main program just has to open the file, load the structures and try them out, one after another, until it finds a working arrangement.

Let's say that the number of molecules we want to work with is n. In three dimensions they can be placed in a cube, with n molecules in each direction, an $n * n * n$-sized grid. The number of combinations to place n molecules in n^3 place is $\binom{n^3}{n}$. The algorithm handles the cube as a queue, by assigning a number to every place of the grid. After the empty n^3-sized array is made, the program has to fill it with the molecules. The program places the first molecule to the first place, then the next one to the next place, and so on, until the last one. Then it simply pushes the last molecule to the next place, and repeats this process until it is in the last place. If the program reaches this point, it then puts the previous molecule one place away than it was before and repeats this operation, until there are no more empty places after the molecules.

Another key feature is to write the structures to a file during the process, so after it is done, the main program can read it, and run the simulations. The main issue with this program was that, because n is a variable, the program had to be made with recursion, so the filling function had to call itself.

5.4.2 Further ides to the structure finding algorithm

This smaller program was not further optimised, since we decided not to use it in the end, because of the reasons written in Section 5.4.1. That being said we have several ideas, if the resources are available to run simulations on more structures. Furthermore we will optimise this program, to filter the structures. In the following lines a brief description of these methods is given.

The simplest way to see if two structure are the same, is translation. We have to subtract the same number from every molecule's every coordinate, until they are non-negative. This way we can filter structures in which the molecules are in the same arrangement and in the same orientation. The next problem is with the structures which are in the same arrangement, but in different orientation. This can be solved, by rotating the whole structures around the three axes, and getting their reflection through the three planes, defined by two axes.

The last case is not a geometrical problem, like the ones above, but the fact, that there are structures, which are not identical, but the simulation could consider them as such. For example knowing that if there are 2 structures made of the same number of molecules, but in the first the molecules are lined up one after another, and in the second one, they are in an 'L'-shape. This should be a different solution, but in the simulation the only thing that matters is the adjacent molecules, so it would give the same result in function. Therefore in terms of simulation results, the example mentioned above shall not be tested with both of the structures, but only with one of them, to reduce the searching algorithm's duration.

5.4.3 Further improvements to the Searching Algorithm

Minor improvements can be made to the txt file printed when the *Searching Algorithm* finds the result. One such improvement is the specifying of the order in which the electric fields are applied to the input proteins. This feature would reduce manual simulation checking times, and could be easily implemented. Another minor improvement could be made to show which output channel corresponds to which output protein. Currently determining this can only be done after the results are checked manually by simulation.

An annoying issue that we found with the algorithm is that after it finds the structure and prints it out into a file, we have to manually check if the structure actually realizes the desired logic function. A simple idea that we have could automate this part as well. A small algorithm would take the file generated by the *Searching Algorithm*, and perform the simulations that we perform manually based on this file. This would also create the excel files generated in the simulation part of the program, thus we can easily create the data and graphs we need, for proving that the *Searching Algorithm* did indeed find the desired structure. This is an easy to implement feature that would greatly decrease the time needed for checking the results.

Other ideas that we have could decrease the running time of the *Searching Algorithm*, by further optimizing the process. One such thing that we could improve is to decrease the number of input protein configurations that are tried out, but identical in function for the specific structure. For example let's say we have a structure consisting of 3 proteins place linearly. We can see that putting the 2 inputs at one and or the other will yield the same configuration, but still the algorithm in its current state will try out both. Another improvement would be to further increase the number of checks that are implemented in the algorithm. These checks are placed at various parts of the algorithm, using some parameters and deciding whether moving on to the next configuration is possible, thus terminating a *for loop* faster. One such feature, that we already implemented is the following: After trying out one row of the logic function's table for the current configuration, if there were no viable electric field values, which would yield the right output, trying out further rows would be pointless, thus the algorithm will skip to the next input protein configuration.

We already described how hard it is to create an algorithm that only checks the structures needed, so until that problem is not resolved these ideas could prove useful to somewhat lessen the running time of the *Searching Algorithm*. However these improvements prove to be a greater challenge than expected, since they also require the algorithm to be more intelligent.

6 Conclusion

With the help of theoretical findings discussed in the first 2 chapters of the documentation we have developed and presented a simulation program which can be used to simulate coulomb-coupled protein structures. With the help of a graphical user interface we've shown how easy it is to build molecular structures, and run simulations on them. Visually interesting design and scalability of the simulation software were two important aspects that we took into consideration. We've also shown how molecular structure finding algorithms can be automated for specific logic functions, and how these could be further improved. We presented the simulation results and the findings of the logic gates, and some logic functions, comparing the protein structures to transistor electronic circuits. Challenges and difficulties were overcome, problems and issues were presented, and improvements and future plans were discussed. All in all, we showed how Dronpa and other coulomb-coupled proteins can replace the modern day transistor, presenting the molecular structures that achieve some of the most basic logic functions, and gave a framework software which can be used for further simulation and building of such protein structures.

So far everything that we've discussed and shown are purely theoretical simulations. A very important subject would be to try out these simulations in practice. Unfortunately we don't have the resources or knowledge necessary to conduct such research, but nevertheless it would be very important to further prove the theoretical findings of our program. The engineering involved in creating the molecular structures given is very complex and of high level, but we urge scientists to try out such experiments in practice, because the end goal is to make physical real-world molecular structures working as computing architectures.

References

[1] Benenson Y, Gil B, Ben-Dor U, Adar R, Shapiro E. An autonomous molecular computer for logical control of gene expression. *Nature* 2004; 429(6990):423–429, DOI: 10.1038/nature02551.

[2] Rakos B. Simulation of Coulomb-coupled, protein-based logic. *Journal of Automation, Mobile Robotics & Intelligent Systems* 2009; 3(4):46–48.

[3] Rakos B. Coulomb-coupled, protein-based computing arrays. *Advanced Materials Research* 2011; 222:181-184. DOI: 10.4028/`www.scientific.net/AMR.222.181`.

[4] Csurgay ÁI, Porod W. Equivalent circuit representation of arrays composed of Coulomb-coupled nanoscale devices. *International Journal of Circuit Theory and Applications* 2001; 29:3–35. DOI: 10.1002/1097-007X(200101/02) 29:1<3::AID-CTA130>3.0.CO;2-Y.

[5] Körner H, Mahler G. Optically driven quantum networks: applications in molecular electronics. *Physical Review B* 1993; 48(4):2335–2346. DOI: 10.1103/PhysRevB.48.2335.

[6] Csaba Gy, Csurgay ÁI, Porod W. Computing architecture composed of next-neighbour-coupled optically pumped nanodevices. *International Journal of Circuit Theory and Applications* 2001; 29:73–91. DOI: 10.1002/1097-007X (200101/02)29:1<73::AID-CTA134>3.0.CO;2-A.

[7] Csurgay ÁI, Porod W, Rakos B. Signal processing by pulse-driven molecular arrays. *International Journal of Circuit Theory and Applications* 2003; 31:55–66. DOI: 10.1002/cta.225.

[8] Rakos B, Csurgay ÁI, Porod W. Recovering pure states in two-state quantumsystems. *Superlattices and Microstructures* 2003; 34:503-507. DOI: 10.1016/j.spmi.2004.03.049.

[9] Rakos B, Porod W, Csurgay ÁI. Computing by pulse-driven nanodevice arrays. *Semiconductor Science and Technology* 2004; 19:472-474. DOI: 10.1088/0268-1242/19/4/155.

[10] Xu D, Phillips JC, Schulten K. Protein response to external electric fields: relaxation, hysteresis, and echo. *Journal of Physical Chemistry* 1996; 100:12108–12121. DOI: 10.1021/jp960076a.

[11] Habuchi S, Ando R, Dedecker P, Verheijen W, Mizuno H, Miyawaki A, Hofkens J. Reversible single-molecule photoswitching in the GFP-like fluorescent protein Dronpa. *Proceedings of the National Academy of Sciences* 2005; 102(27):9511-9516. DOI: 10.1073/pnas.0500489102.

[12] Rakos B, Modeling of dipole-dipole-coupled, electric field-driven, protein-based computing architectures, *International Journal of Circuit Theory and Applications* 2015, 43, 60-72.

[13] Intel, an American multinational corporation and technology company. URL: http://www.intel.eu/content/www/eu/en/homepage.html

[14] NAMD, a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. URL: http://www.ks.uiuc.edu/Research/namd/

[15] VMD, a molecular visualization program for displaying, animating, and analyzing large biomolecular systems using 3-D graphics and built-in scripting. URL: http://www.ks.uiuc.edu/Research/vmd/

[16] Rakos B, Multiple-valued Computing by Dipole-dipole Coupled Proteins. Submitted for publication 2016

[17] Microsoft Visual Studio Enterprise 2015, an integrated development environment (IDE) from Microsoft. URL: https://www.visualstudio.com/

[18] FreeGLUT, a free-software/open-source alternative to the OpenGL Utility Toolkit (GLUT) library. URL: http://freeglut.sourceforge.net/

[19] https://github.com/davidwparker/opengl-screencasts-2

[20] Microsoft Office Excel, a spreadsheet developed by Microsoft. URL: https://products.office.com/excel

[21] Rakos B, Pulse-driven, Photon-coupled, Protein-based Logic Circuits. *Advances in Intelligent Systems and Computing* 2017, 519:123-127. DOI: 10.1007/978-3-319-46490-9_18