Macalester Journal of Physics and Astronomy

Volume 12
Issue 1 *Spring 2024*

Article 5

May 2024

# Simulation of Polymerization on Surfaces: Implications for Abiogenesis

Sylvia Greene
*Macalester College*, sgreene4@macalester.edu

# Simulation of Polymerization on Surfaces: Implications for Abiogenesis


## Abstract

How did life arise from the prebiotic conditions of the early earth? This problem has vexed scientists for decades with no consensus on its solution. Significant spontaneous formation of biopolymers such as proteins and nucleic acids in the aqueous phase appears to be improbable due to thermodynamic constraints. It has been proposed that mineral surfaces could have served as a catalyst for the initial formation of biopolymers. However, the feasibility of this mechanism has not been thoroughly studied. In this study, a particle simulation of polypeptide formation on surfaces is developed to assess the feasibility of this mechanism. Elementary processes such as monomer adsorption, monomer and dimer diffusion, desorption, and peptide bond formation are included in the model. The production of long polymers that could serve as building blocks of proteins is considered as a function of bonding activation energy, polymer desorption energy, and the number of wet-dry cycles experienced by the surface.



## Keywords

Origins of Life, Polymerization, Simulation

## Cover Page Footnote

I would like to thank Professor Jim Doyle for expertise, support and guidance throughout this project and my college career.

# Simulation of Polymerization on Surfaces: Implications for Abiogenesis

Sylvia Greene

Advisor: Prof. Jim Doyle

April 29, 2024

**Abstract**

How did life arise from the prebiotic conditions of the early earth? This problem has vexed scientists for decades with no consensus on its solution. Significant spontaneous formation of biopolymers such as proteins and nucleic acids in the aqueous phase appears to be improbable due to thermodynamic constraints. It has been proposed that mineral surfaces could have served as a catalyst for the initial formation of biopolymers. However, the feasibility of this mechanism has not been thoroughly studied. In this study, a particle simulation of polypeptide formation on surfaces is developed to assess the feasibility of this mechanism. Elementary processes such as monomer adsorption, monomer and dimer diffusion, desorption, and peptide bond formation are included in the model. The production of long polymers that could serve as building blocks of proteins is considered as a function of bonding activation energy, polymer desorption energy, and the number of wet-dry cycles experienced by the surface.

# Contents

# Chapter 1

# Introduction

## 1.1 Background

The investigation into the origins of life on Earth, or abiogenesis, has captivated scientists for centuries and is one of the most important unsolved problems in science today. Simply put, how did life arise from non-living matter? Amino acids, nucleotides, sugars, and lipids are recognized as the fundamental building blocks of life. The first three are capable of polymerization, forming proteins, and nucleic acids which constitute the basic biochemical machinery for cellular function and reproduction . In the 1950s, the groundbreaking Miller-Urey experiments demonstrated the synthesis of amino acids from simple molecules in prebiotic-like conditions [1]. Subsequent research has demonstrated the synthesis of amino acids and nucleobases under diverse prebiotic scenarios, with amino acids even being detected on meteorites [12]. The next step in the origin of life involves the assembly of polymers from these building blocks. Once these polymers become a critical length, they could begin folding and exhibiting enzymatic properties or promoting self-replication. Eventually, they would need to be encapsulated within the earliest cellular structures. However, the mere presence of monomers does not guarantee the existence of polymers. The problem of biogenesis can perhaps be stated more specifically as follows. Assuming that some form of biopolymers such

as proteins and nucleic acids were necessary building blocks of primordial life the question becomes: how did the first biopolymers arise without the complex cellular machinery that is now apparently essential to all known life?

Geological evidence indicates a time frame between Earth's formation approximately 4.5 billion years ago and the earliest fossil evidence of cellular life around 3.5 billion years ago as the likely window for life to have originated [2]. However, the formation of biopolymers under prebiotic conditions presents a considerable challenge for several reasons. One problem is the unfavorable kinetics and thermodynamics associated with bond formation. In modern life enzyme catalysis and chemical energy input from ATP (ultimately originating from photosynthesis) promote the formation of biopolymers, but these reaction mechanisms were obviously not available in the prebiotic earth. Another challenge is the difficulty in discerning the actual environmental conditions of early Earth. Researchers have proposed ideas for the composition of the atmosphere, oceans, and land masses during that era, but clearly, all of these proposals are currently very speculative.

The production of biopolymers in the solution phase, without enzymes and ATP energy input, is not favorable. The formation of a peptide bond can be written as

$$\mathrm{A} + \mathrm{A} \longrightarrow \mathrm{A{-}A} + \mathrm{H_2O} \tag{1.1}$$

and is commonly referred to as a condensation reaction. The reverse reaction where the peptide bond is split is known as hydrolysis. Estimates of free energy formation for peptide bonds range from about 7 to about 28 kJ/mol (or 0.07 to 0.28 eV) depending on conditions such as pH and the type of amino acids[3]. These numbers do not include activation barriers that may also exist above the thermodynamic free energy change. The endoergic nature of this reaction poses a problem for the spontaneous synthesis of polypeptide chains.

For example, based on a free energy of formation of peptide bonds of approximately 10.5 kJ/mol, Lambert calculates that the equilibrium concentration of a 14-monomer polypeptide is roughly $3 \times 10^{-30}$ mol/l, and for an 18-monomer polypeptide, it's $1.5 \times 10^{-38}$ mol/l. Lambert remarks that this scenario "is not a promising start for metabolism" [4]. Such thermodynamic constraints on amino acid polymerization in aqueous solutions extend to the synthesis of nucleic acids like RNA. As a result, researchers have concluded that the formation of biopolymers within solution phases aqueous environments is unlikely, at least under conditions assumed for the prebiotic earth [6].

A potential solution to this challenge was proposed in 1951 by J.D. Bernal, who suggested that biopolymers might have formed through adsorption onto mineral surfaces [6]. Surfaces have several possible advantages in this regard. By reducing the dimensionality of the problem, they can increase the "collision" rates of monomers, thereby increasing the probability of a favorable bonding event. In addition, it has been proposed that the surface could have a catalytic effect by altering the peptide bond formation mechanism resulting in a lower activation energy. As emphasized by Lambert [4], catalytic effects cannot change the overall free energy increase going from aqueous monomers to a desorbed aqueous polymer. However, as Lambert points out, cataytic effects cannot overcome the overall increase in free energy from aqueous monomers to polymers. On the other hand, Lambert notes that in the absence of water, the hydrolysis of the peptide bond (i.e. the reverse reaction of peptide bond formation) will not occur, thereby driving the equilibrium towards polymerization. To circumvent this equilibrium problem a dual-stage wet-dry process has been proposed. In its simplest implementation, amino acids adsorb on the surface during the wet phase. In the dry phase, the surface-adsorbed monomers can react to form peptide bonds without the reverse hydrolysis reaction occuring. The problem of the activation barrier for bond formation still needs to be considered, but the higher temperature expected in the dry phase will assist in overcoming the barrier. In the next wet phase, the polymers can desorb and

perhaps contribute to some kind of protocell formation. This sequence of events is usually envisioned to occur in a lagoon-like environment with tides producing the needed wet-dry cycle.

Since Bernal's proposal, a number of studies have investigated the adsorption of amino acids on surfaces and explored the potential catalytic effects of these surfaces, both experimentally and theoretically. However, we are not aware of a comprehensive study that attempts to model the wet-dry hypothesis over many cycles. This paper aims to explore what such a model might look like using particle Monte Carlo methods. By varying parameters within the model, we can observe how polymerization on catalytic surfaces influences the length of produced polymers. Because we lack robust input data on kinetic parameters needed for a comprehensive model of biopolymer formation on surfaces, our specific results will be somewhat speculative. Thus, rather than provide a definitive answer to the proposal of surface-mediated abiogenesis, our intention is to produce and test a viable model for such an approach and consider trends with parameters such as activation energies which can be updated as more experimental constraints become available. An important problem in surface-mediated abiogenesis arises if it is assumed that the biopolymer must desorb from the surface in order to be incorporated in a protocell. Presumably, with successive polymerization events, the resulting polymer will become increasingly adsorbed to the surface. This presents another challenge for theories on the origin of life, as longer polymers could become permanently bound to the surface. The wet-dry cycle may help this problem, in that the polymers could form on the surface during dry periods and then desorb during wet phases, where the desorption is assisted by solvation of the polymer. We will consider various options to model this aspect of the problem in Chapter 3.

### 1.1.1 Modeling Surface Polymer Formation

In this study, we choose to implement a particle simulation using Monte Carlo methods. This is in contrast to a rate equation approach to surface polymerization described by the

following set of equations for the ith surface species:

$$\frac{\partial n_i}{\partial t} = D_x \frac{\partial^2 n_i}{\partial x^2} + D_y \frac{\partial^2 n_i}{\partial y^2} + R_i a - k_d n_i - \sum_m k_b n_i n_m \quad i = 1, 2, \ldots, m$$

On the right-hand side, we have terms representing 2-D diffusion ($D_x \frac{\partial^2 n_i}{\partial x^2} + D_y \frac{\partial^2 n_i}{\partial y^2}$ ), adsorption rate ( $R_i a$ ), desorption rate ($k_d n_i$), and bonding ($\sum_m k_b n_i n_m$). The sum over m in the last term is over all other species on the surface including monomers, dimers, and polymers. Solving such a set of equations is a formidable computational problem and to our knowledge has not yet been attempted. The problem is greatly magnified when there is more than one type of monomer species.

Alternatively, in a particle simulation, the system is simulated by a finite number of particles, and the various elementary processes are treated using probabilities and Monte Carlo sampling methods. This approach has the virtue of greatly simplified mathematics and programming, and the addition of multiple processes and species is straightforward and does not affect the basic algorithmic structure. The main disadvantage of this approach is a typically much higher computational time with the need for a large number of particles and simulation area to generate good statistics. In addition, simulations of this kind can sometimes obscure the underlying physics and chemistry, which must be teased out by running the simulations under a wide variety of conditions. In a sense the particle simulation approach is really more like an experiment, the results of which can be used to inform a more general and complementary theoretical approach, as well as inspire particular experimental approaches.

# Chapter 2

# Methods

## 2.1 Experimental Setup

The simulation consists of two distinct phases: an aqueous stage and a dry stage. During the aqueous (wet) stage, a lattice surface is immersed in an aqueous solution containing monomers. At the interface between the aqueous solution and the surface, monomers can undergo processes of adsorption onto the surface, desorption from the surface, or diffusion along the surface. During the dry stage, desorption, diffusion, and the formation or destruction of bonds can also take place, but likely at rates distinct from those observed during the wet stage.

## 2.2 Assumptions of the Model

The model has several simplifying assumptions in order to make the computation tractable and minimize the number of adjustable parameters. It is important to emphasize that the aim of this model is not to present a definitive representation of polypeptide formation on surfaces. Given the enormous uncertainties surrounding abiogenesis mechanisms and the conditions of the early earth, drawing precise conclusions from any model cannot be justified

at this time. Instead, we view this simulation as an initial exploration, aimed at developing the algorithmic approach and providing a preliminary understanding of the relative significance of various parameters, in particular activation energies, in surface reactions.

With the exception of the adsorption of monomers in the wet phase, all processes are assumed to be thermally activated with a rate that follows the general Arrhenius form [7].

$$k_i = A_i e^{-\frac{E_{ai}}{kT}} \tag{2.1}$$

In equation 2.1 $Ea_i$ is the activation energy for the $i$th process, $k$ is Boltzmann's constant, $T$ is the temperature in Kelvin, and $A_i$ is the prefactor. In this expression, $A_i$ has units of $s^{-1}$ and can be considered to be a kind of "attempt frequency". Theoretically $A_i$ can be calculated by using some version of transition state theory [8]. In this approach, the rate is determined by the properties of the reactants and a putative transition state. We have:

$$A_i = \frac{kT}{h}\frac{Q_i^*}{Q_i} \tag{2.2}$$

where $h$ is Planck's constant and $Q_i$ and $Q_i^*$ are the partition functions for the reactant state and the transition state respectively. For the processes considered here, there is insufficient theoretical or experimental data to unambiguously determine the parameters $A_i$ and $Ea$. Our approach will be more phenomenological and instead focus on a parameterization of the rates in terms of the activation energies for the various processes. As discussed below the simulation ultimately only requires the relative rates of the elementary processes. In order to minimize the number of adjustable parameters, we will make the assumption that the prefactors $A_i$ are the same for every process. Typically the reaction rate is dominated by the activation energy. That is, the relative rates are assumed to follow:

$$\frac{R_i}{R_j} = \frac{P_i}{P_j} = \frac{A_i e^{-\frac{E_{ai}}{kT}}}{A_j e^{-\frac{E_{aj}}{kT}}} = e^{-\frac{(E_{ai}-E_{aj})}{kT}} \tag{2.3}$$

which also represents the relative probability of the two processes. Thus we are assuming that the relative probabilities of the elementary processes are dominated by their differences in activation energies. Although this approach neglects potentially important effects such as entropy differences between reactants and transition states, at the current level of sophistication of our simulation it is not a serious limitation, since the wet and dry phases are assumed to occur at fixed temperatures. A more general expression for the relative rates is:

$$\frac{P_i}{P_j} = \rho_{ij} e^{-(E_{ai} - E_{aj})/kT} \tag{2.4}$$

where $\rho_{ij}$ represents the ratio of the prefactors. The "real" probability ratio can be accommodated by simply setting $\rho_{ij} = 1$ and adjusting the difference between the activation energies. Because of the exponential dependence on the difference in activation energies, even small adjustments to the activation energies can have significant effects on the relative rates. The main point is that in the simulation, only the relative probabilities of the processes are relevant. The use of activation energies to determine these ratios is a heuristic guide to simplify the calculation and minimize the number of parameters.

Assuming we can characterize the relative rates based on activation energies, a crucial difference between desorption and diffusion processes lies in whether the monomers are physisorbed or chemisorbed. Physisorption involves weak Van der Waals forces and is easily reversible, whereas chemisorption involves stronger chemical bonds and is typically considered irreversible. Factors influencing adsorption include the surface area and properties of the solid material, the nature of the molecules in the fluid, temperature, pressure, and the presence of other substances. Physisorbed molecules generally exhibit desorption and diffusion energies on the order of tens to hundred of of meV, whereas chemisorbed molecules typically have corresponding energies in the eV range [9]. Limited data is available for desorption energies of amino acids on surfaces, and that information is primarily from molecular dynamics simulations on single crystalline surfaces, giving values from about 0.4 to 1.5 eV

[10]. Some experimental data on the desorption and diffusion of organic molecules on dust grains can be found in the astrophysics literature, where physisorption energies typically range from 0.07 to 0.35 eV. Since it is unknown what type of adhesion would occur under abiogenesis conditions, we opt for a middle ground, assuming desorption energies near the upper end of the physisorbed values and the lower end of the chemisorbed values. Tables 2.1 and 2.2 summarize our default parameters for desorption and diffusion.

| | Activation Energy (eV) | |
|---|---|---|
| | Desorb | Diffuse |
| Monomer | 0.3 | 0.18 |
| Dimer | 0.4 | 0.24 |
| Polymer | 0.5 | - |

Table 2.1: Default Activation Energies Wet Phase

| | Activation Energy (eV) | |
|---|---|---|
| | Desorb | Diffuse |
| Monomer | 0.4 | 0.24 |
| Dimer | 0.6 | 0.36 |

Table 2.2: Default Activation Energies Dry Phase

| | Probabilities | |
|---|---|---|
| | Desorb | Diffuse |
| Monomer | $3.05 \times 10^{-3}$ | 0.2 |
| Dimer | $9.33 \times 10^{-5}$ | $2.47 \times 10^{-2}$ |
| Polymer | $2.86 \times 10^{-6}$ | - |

Table 2.3: Default Probabilities in Wet Phase

| | Probabilities | |
|---|---|---|
| | Desorb | Diffuse |
| Monomer | $1.77 \times 10^{-3}$ | 0.2 |
| Dimer | $4.81 \times 10^{-6}$ | $5.77 \times 10^{-3}$ |

Table 2.4: Default Probabilities in the Dry Phase

The assumption is made that the adsorption of monomers from solution onto the surface oc-

curs without a barrier. The rate of this adsorption process is therefore determined primarily by the concentration of amino acids in the wet phases of the cycle. According to elementary kinetic theory, the rate per unit area for adsorption of particles onto a surface is given by [11]:

$$R_a = \frac{1}{4} sn\langle v \rangle \tag{2.5}$$

where s is a sticking coefficient, n is the volume concentration of the particles in solution, and $\langle$v$\rangle$ is the average speed of the particles when they arrive at the surface. For impingement from a solution where the arrival of particles at the surface is diffusion-limited, $\langle$v$\rangle$ is interpreted as diffusional velocity. In our simulation, the rate of adsorption is an adjustable parameter that is specified as a probability per unit time step. This parameter thus serves as a proxy for the concentration of monomers in the solution wet phase. In Chapter 3 we show that the results are not strongly dependent on the choice of this probability as long as the number of cycles is large and the initial seeding on the surface is adequate.

Monomers, dimers, and polymers can all desorb from the surface during the wet and dry phases. As discussed in Chapter 1, presumably the activation energies for desorption increase as the polymer grows, given the greater number of contact points between the polymer and the surface. This suggests that the likelihood of desorption diminishes rapidly as the number of monomer units, N, increases. Indeed, as discussed in Chapter 1, this is considered to be one of the primary obstacles regarding abiogenesis biopolymer formation on surfaces. In our simulation, we examine several possible models for the desorption of polymers. To compare models our default assumption assumes constant probability of desorption regardless of polymer length. We compare this with the case that the desorption energy scales linearly with the number of monomers:

$$E_{aN} = N \times E_{adM} \tag{2.6}$$

Where $E_{adM}$ is the desorption energy of a single monomer and $N$ is the length of the

monomer. This is quite different from the previously mentioned model which assumes that beyond a certain value of N, the desorption energy does not increase. This latter situation might occur if, as monomers are added, the polymer becomes more loosely bound because the monomer repeat distance is not commensurate with the surface binding sites. In essence, a kind of "buckling" occurs. A variation of this idea is that the binding energy increases and then decreases with polymer length periodically as the polymer length increases. Intermediate possibilities include a decrease in polymer desorption probability with N that is weaker than that implied by a linear increase in activation energy, for example $1/N$. Additionally, the desorption energy is likely to be lower in the wet phase than in the dry phase, due to the expected free energy advantage of solvation in the aqueous phase.

We assume that monomers and dimers can diffuse on the surface during the wet and dry phases, although dimers have a higher activation energy to do so. Based on the literature, the activation energy for surface diffusion is typically 0.2 to 0.8 times the activation barrier for desorption [12]. In our simulation, this fraction is fixed at 0.6 (Tables 2.1 and 2.2). Polymers containing three or more units do not diffuse. This assumption is somewhat inconsistent with the fact that polymers can desorb, but simplifies the calculations and is unlikely to influence the results given the significantly slower diffusion rates of polymers compared to monomers and dimers.

Given our assumption that the simulation is nominally supposed to represent polypeptide formation, we incorporate additional characteristics typical of these systems. Specifically, we assume that in the absence of water during the dry phase, hydrolysis of the peptide bond does not occur. Peptide bond formation, however, can occur during the dry phase but requires an activation energy equal to or greater than the magnitude of the enthalpy for the (exoergic) hydrolysis reaction. Several values have been cited for this energy in the literature spanning from about 0.1 to 0.3 eV (10 kJ/mol to 30 kJ/mol). This quantity does not include a possible

activation energy beyond the reaction enthalpy. Variation of the bonding activation energy is one of the main parameters we wish to study in this paper. Values between 0.16 eV (16 kJ/mol) and 0.48 eV (48 kJ/mol) were considered, with 0.24 eV (24 kJ/mol) as the default value. This activation energy is an important parameter in our simulation, as any catalytic effect of the surface would suggest a lower activation energy compared to the reaction in the solution phase. The activation energy for uncatalyzed peptide bond hydrolysis (the reverse of peptide bond formation) is around 1 eV [13]. Such a high activation energy for hydrolysis is consistent with the observation that proteins do not spontaneously hydrolyze in aqueous solutions, despite the fact that the reaction is thermodynamically favorable. If the mechanism for peptide bond formations is simply the reverse of the hydrolysis reaction, this would imply that the peptide bond formation barrier is about 1.24 eV without catalysis. The energetics of the bond formation are shown in Figure 2.1 below.

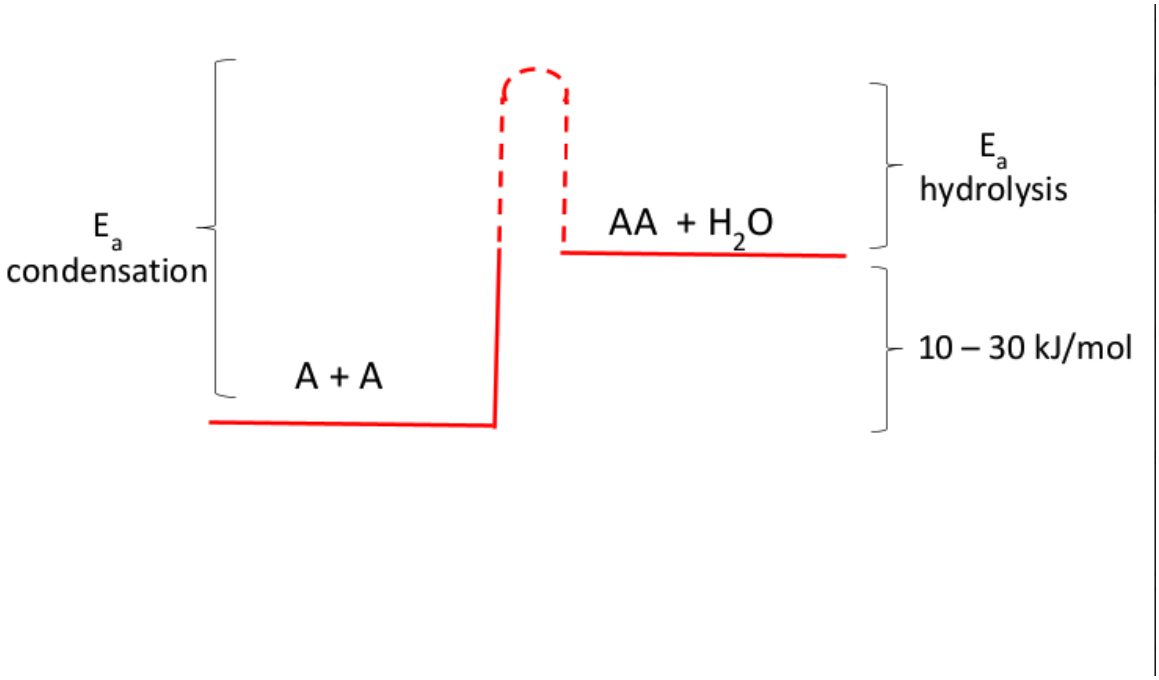



Figure 2.1: Figure 2.1. Energy Diagram for Polymerization

In the wet phase we assume that the excess water present will tend to suppress peptide bond

formation due to the reverse hydrolysis reaction:

$$\mathrm{A{\equiv}A + H_2O \longrightarrow A + A} \tag{2.7}$$

Thus, in our simulation we will assume that no net peptide bond formation occurs during the wet phase. Since water is not present in the dry phase, no hydrolysis occurs during the dry phase and once a peptide bond is formed it does not dissociate. The processes and their default activation energies that are used in our model are given in Table 1, and Figure 2.2 summarizes the processes considered in the model.

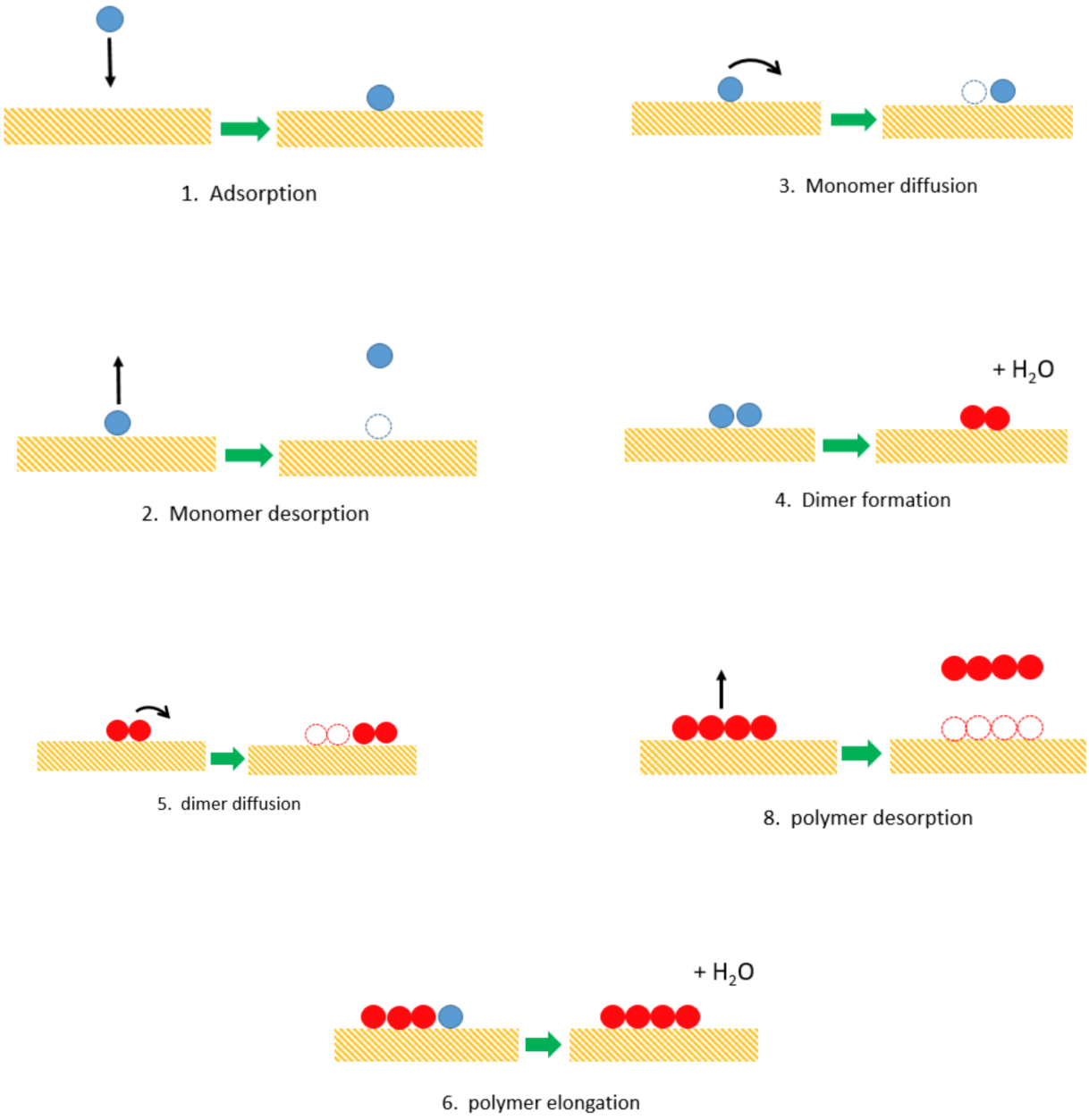
## 2.3 The Simulation

The main output of the simulation is the average length of the longest polymer desorbed for a given set of activation energies and a given number of wet-dry cycles. The 2-D simulation space was discretized as a 1000 x 1000 lattice grid. Individual events can take place only at grid points. Periodic boundary conditions were used for particles that reached the boundaries by diffusion. The simulation begins with an initial seeding of the surface with monomers. Surface diffusion corresponded to jumps between lattice points, and in both the wet and dry phases this was the fastest process. Thus we set the time scale by setting the diffusion probability at any given time step to be 0.2. All other probabilities were scaled to the diffusion probability except for the adsorption of monomers from the aqueous phase, which was set independently. Thus the probability of the $i$th process is:

$$P_i = P_{\mathrm{diff}} \times e^{-(Ea_i - E_{\mathrm{diff}})} \tag{2.8}$$

For a given particle at a given time step, the probabilities for each possible event were summed to produce a cumulative distribution function (CDF) map. By sampling from this CDF map using a pseudo-random number generator, the simulation probabilistically

Figure 2.2: Elementary processes considered in the simulation



selects the event that will occur for the monomer by iterating through the map, summing up each probability until the cumulative probability exceeds the random number. At that point, the function selects the event corresponding to the cumulative probability that first exceeds the random number and returns it. If the random number exceeds the sum of all probabilities, indicating that no event occurs, the function returns "None". The algorithm was tested by generating events based on known probabilities using the provided simulation setup. By comparing the observed event frequencies with the expected frequencies based on the probabilities, the algorithm's accuracy and reliability were confirmed. Appendix A gives a flow chart for the algorithm and Appendix B contains the entire program. The program itself was implemented using the C++ programming language and takes up to 14 hours to run on a Mac Studio desktop, depending on the conditions. The image below illustrates the state of the program after a certain time. The red dots represent bonded polymers (which are dominated by dimers), whereas the blue dots are monomers that are free to move around the surface.

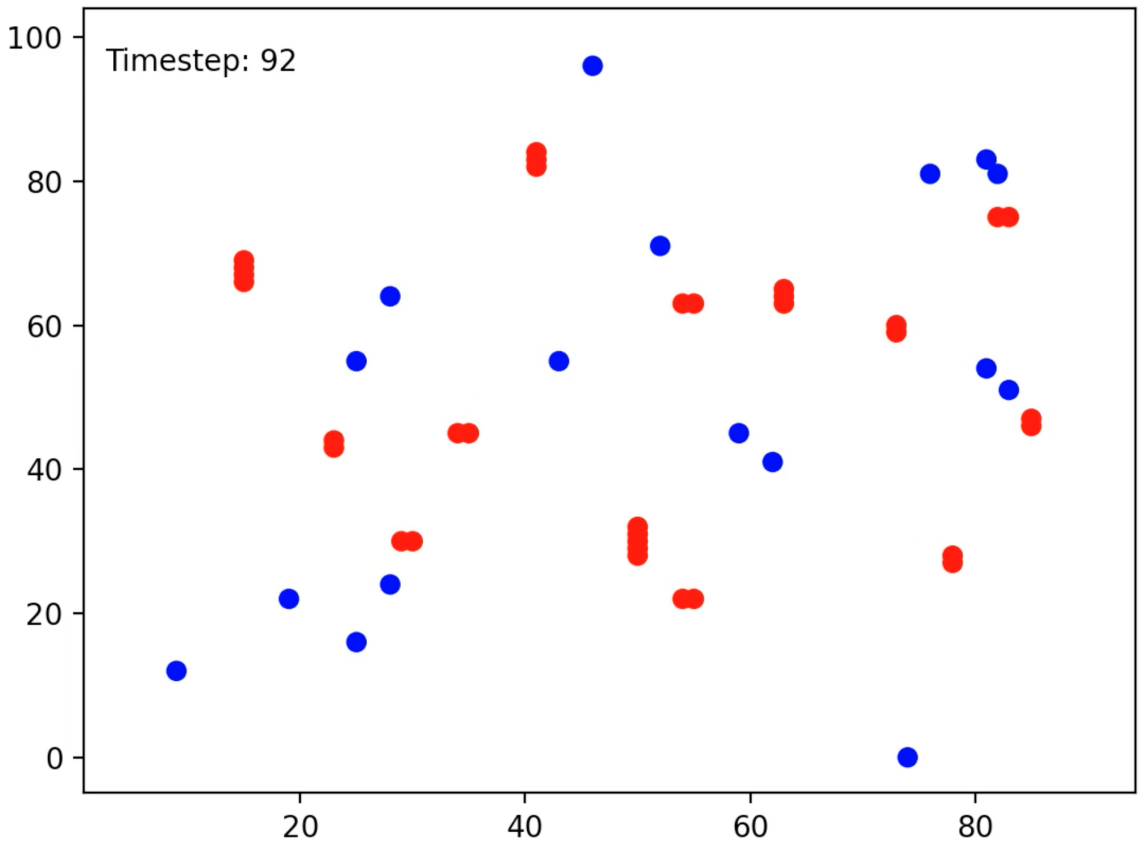



Figure 2.3: Simulation

# Chapter 3

# Results

We ran a series of trials aimed at understanding the relationship between the activation energy (Ea) for bonding and the length of the longest desorbed polymer across varying trial conditions. The activation energy for bonding was varied from 0.16 eV to 0.60 eV in increments of 0.04 eV. Furthermore, we investigated the influence of the total number of cycles on this relationship by conducting trials comprising 30, 300, and 600 cycles, respectively. Throughout all trials, the initial surface particle count remained constant at 100, and a uniform probability of 0.2 was assigned to the diffusion probability (normally the fastest process) to which all other probabilities were scaled as described in Chapter 2. The desorption and diffusion activation energies used the default values in Tables 2.1 and 2.2 unless otherwise noted.

The results from these trials are shown in Figure 3.1

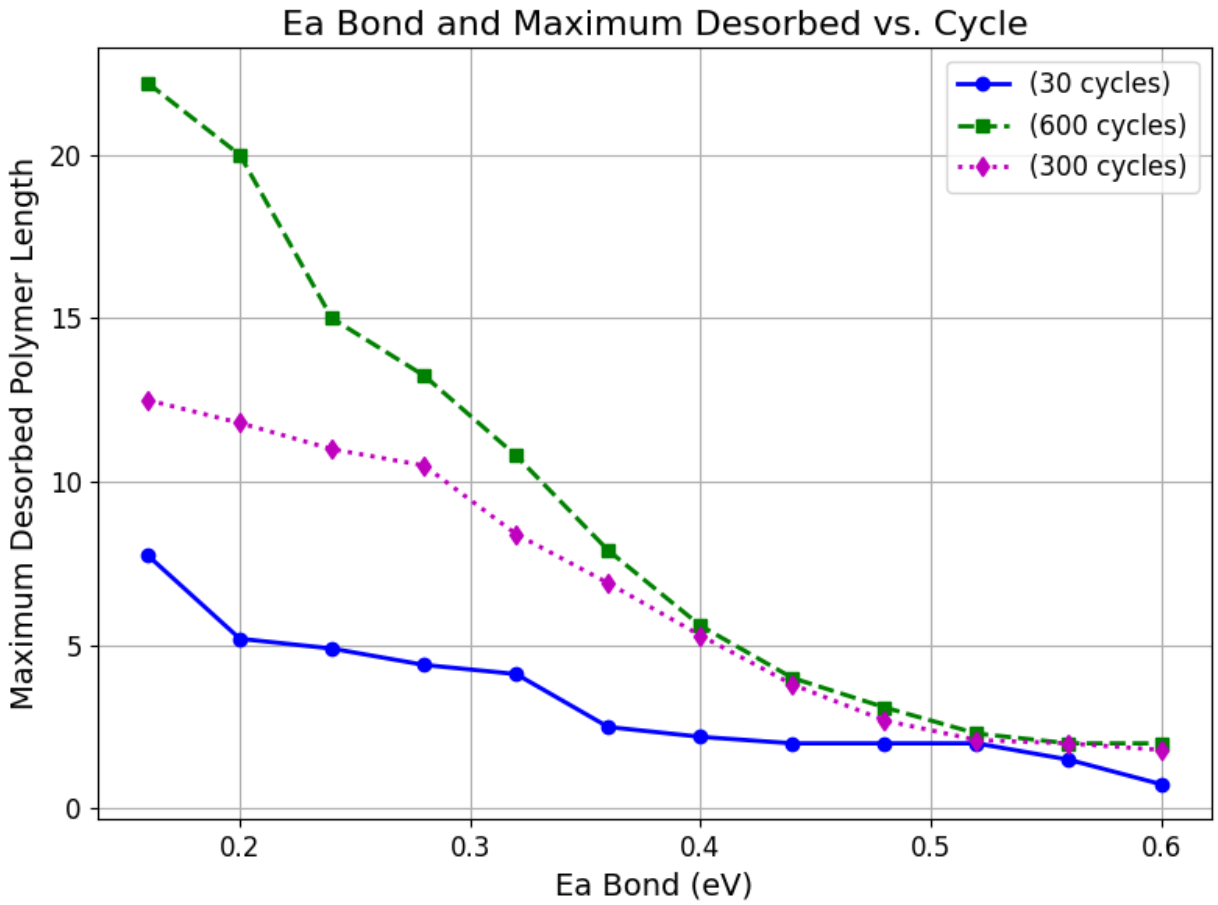



Figure 3.1: Maximum Polymer Length vs Bond Ea for various cycles

It is clear from the figure that for low activation energies (below 0.4 eV), there seems to be a strong dependence of the maximum desorbed polymer length on the total number of wet-dry cycles. Furthermore, for high activation energies, the largest desorbed polymers are dimers (N = 2) across all of the trials. For trials with 30 cycles and an activation energy of 0.60 eV for bonding, the maximum desorbed polymer had an average length of less than 2 units. Of course to be a polymer it must consist of two or more monomers, the value of less than 2 indicates that for some of the trials no polymers desorbed. When these trials are averaged with those where a dimer desorbed, the overall results give an average length of less than 2 units.

To further understand the relationship between the longest desorbed polymer and the number of cycles, we looked specifically at the bonding activation energy of 0.24 eV. We ran experiments for 30, 500, 600,1000 and 2000 cycles, with the results given in Figure 3.2

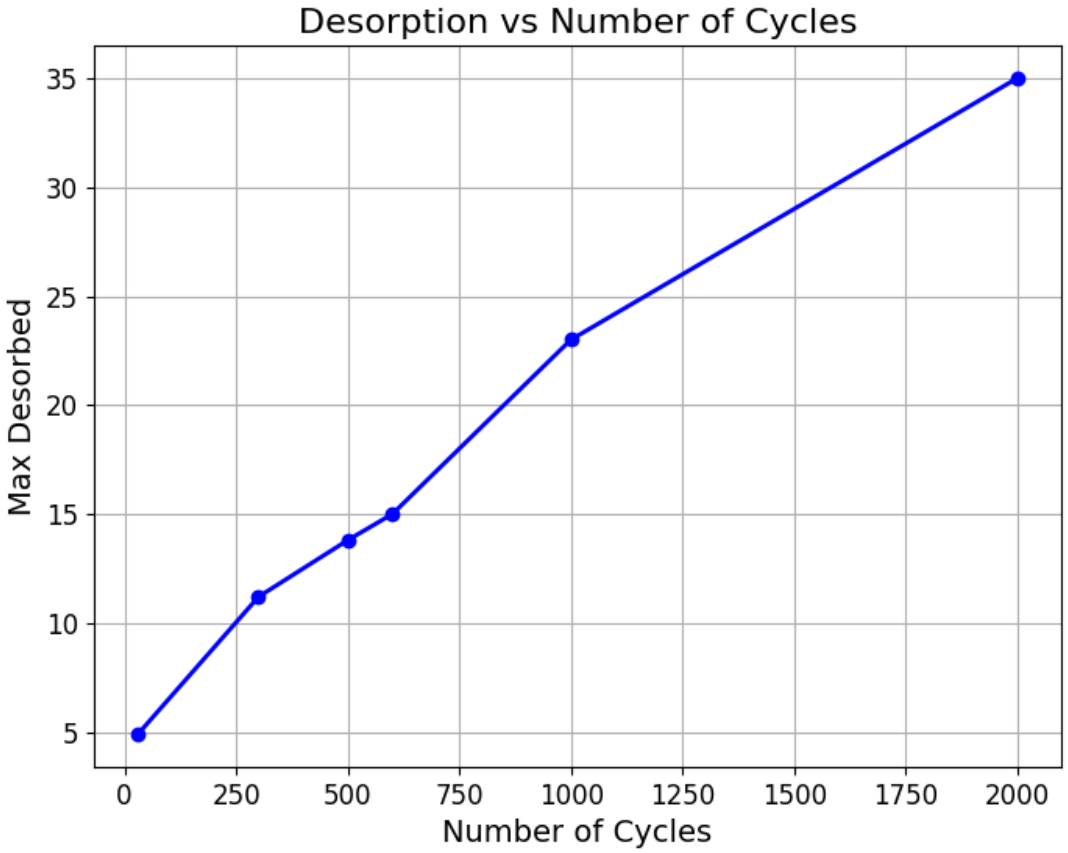


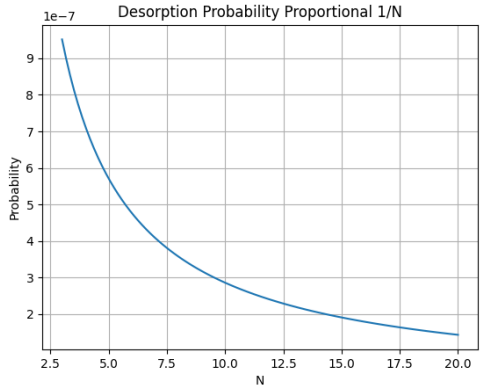

Figure 3.2: Maximum desorbed Polymer vs Number of Cycles For Ea = 0.24

The graph above shows a strong correlation between the number of cycles and the maximum size of the desorbed polymer. For the case of 30 cycles, the maximum desorbed polymer was only 5 units long, but for 2000 cycles the maximum desorbed polymer was 35 units long. This trend is promising for origin of life theories as it shows that given enough time long polymers would form. It is interesting to note that from the graph it seems that the rate of increase in polymer length is slightly decreasing as the number of cycles increases, so instead of being linear may have a slight sublinear relationship. To understand this behavior better

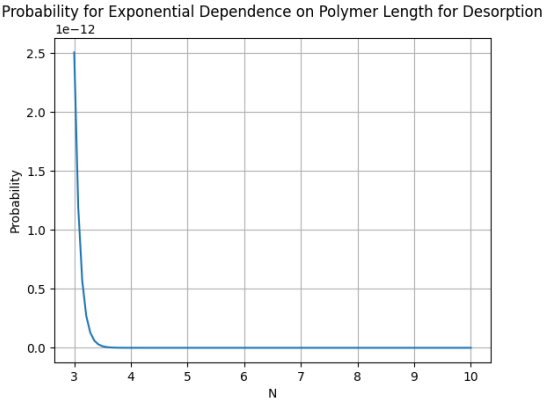
more trials would need to be run at a larger number of cycles.

As mentioned in Chapter 1, the ability of long polymers to desorb from the surface was likely a crucial step in biogenesis. Some researchers have proposed that as polymers on the surface grow, they would become more tightly bound to the surface. To simulate this effect, we varied the desorption probability as a function of the length of the polymer using several models. In addition to the default case where the desorption probability was constant for $N > 2$, we looked at the cases where the desorption probability was proportional to $\frac{1}{N}$, where $N$ is the length of the polymer. Additionally we looked at the case for which the activation energy for desorption is given by $N \times Ea_{dM}$, that is, the desorption energy of a polymer of length N is simply the sum of the desorption energies of the individual monomeric units. Additionally to simulate a potential steric or periodic "buckling" effect, a probability proportional to $|\sin(\frac{\pi}{4}Ea \times (N-1))| + Ea$, shown in figure 3.3 d) was studied. For these trials we ran the various desorption models with a fixed number of cycles (300). The probabilities based on the length are shown in the graphs below for clarity:

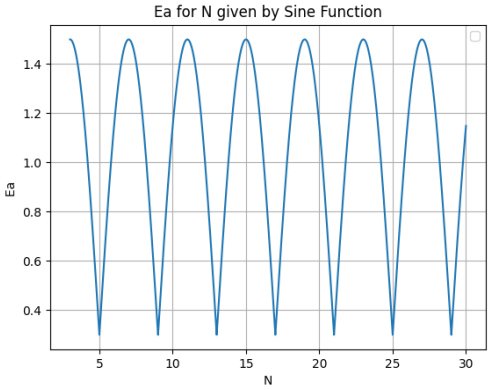
Figure 3.3: Desorption Models showing the probability of bonding versus the length of the polymer



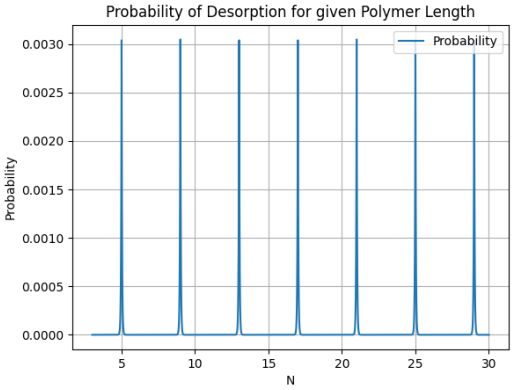

(a) Desorption Probability




(b) Exponential Function




(c) Function for determining bond Ea for Probability




(d) Sine Probability

Figure 3.3 a) presents a function for determining the probability of a polymer of length N desorbing, given by $\mathrm{Pr}_{\mathrm{desorbP}} = \frac{1}{N} \times \mathrm{Pr}_{\mathrm{desorbM}}$. Here, $\mathrm{Pr}_{\mathrm{desorbM}}$ represents the probability of a single monomer desorbing. Figure 3.3 b) illustrates the function for the probability of a polymer of length N desorbing, denoted as $\mathrm{Pr}_{\mathrm{desorbP}} = \mathrm{Pr}_{\mathrm{dM}} \cdot \exp^{(-(N \cdot Ea_{\mathrm{desorbM}} - Ea_{dM})/kT)}$. In this context, $\mathrm{Pr}_{\mathrm{dM}}$ denotes the probability of a monomer diffusing, $Ea_{\mathrm{desorbM}}$ represents the activation energy for a single monomer to desorb, and $Ea_{dM}$ stands for the activation energy for a single molecule to diffuse. Figure 3.3 c) shows the function assumed for the activation energy of a buckling polymer, expressed as as $Ea_{\mathrm{desorbP}} = |4 * Ea_{\mathrm{dM}} * \sin(\frac{\pi}{4} Ea \times (N-1))| + Ea_{\mathrm{dM}}$.

Using these methods, we calculated the average longest polymer resulting from each activation energy over multiple trials. We then compared the results to a scenario where the desorption probability remains constant for $N > 2$, our default condition. The findings are presented in Figure 3.4.

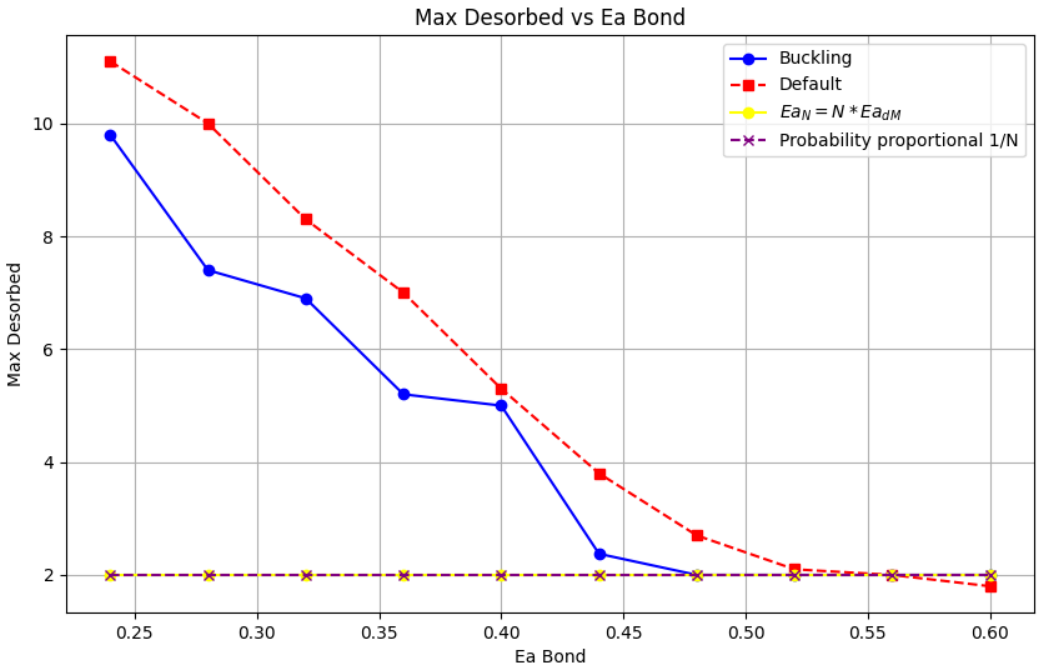



Figure 3.4: Maximum desorbed Polymer for various Models

It is notable that for the cases of desorption proportional to $\frac{1}{N}$ and $e^{-NEa_{\text{desorbM}}}$, only dimers desorb. For the desorption probability given the buckling function, there are longer desorbed polymers than in the previous two cases, but shorter than for the default constant desorption probability for N greater than 2.

For these same cases, we can also look at the maximum polymer that remained on the surface, shown in Figure 3.5

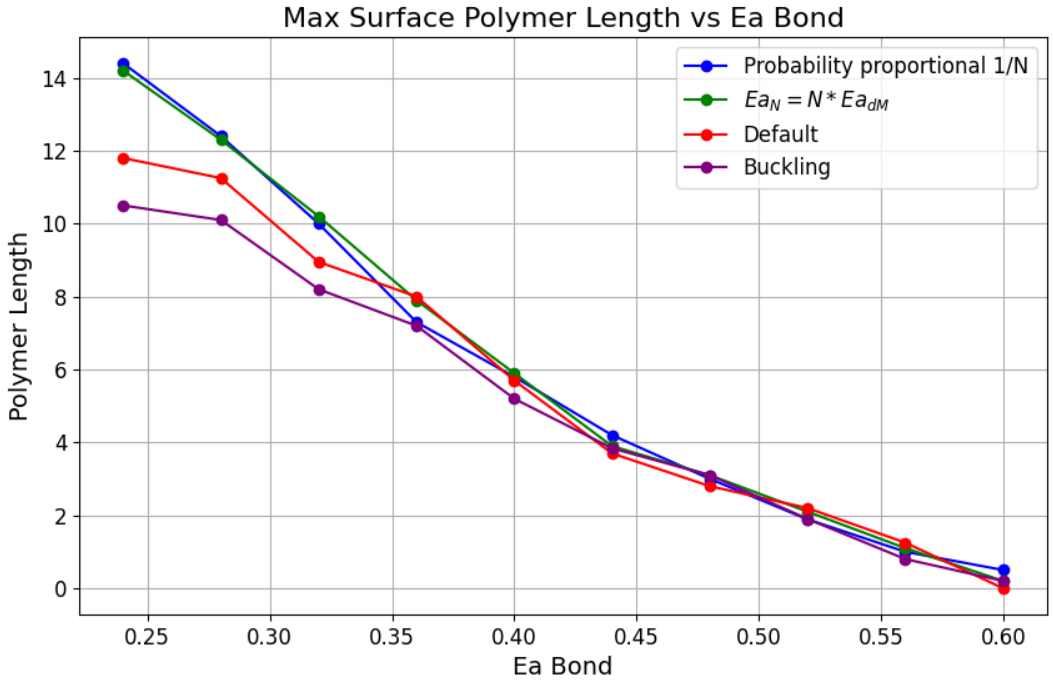
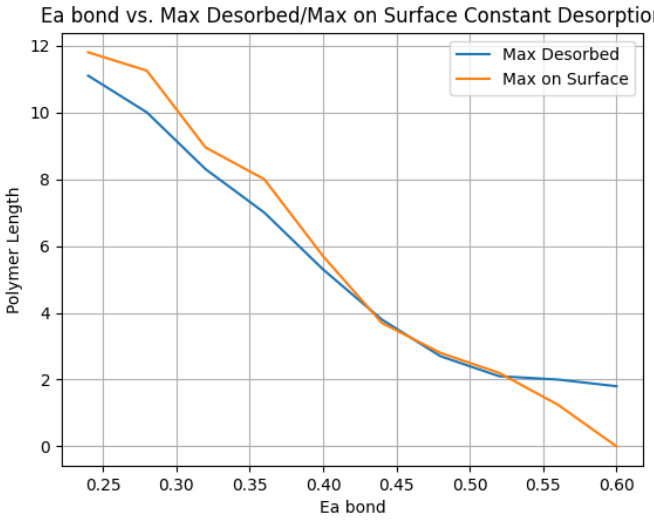


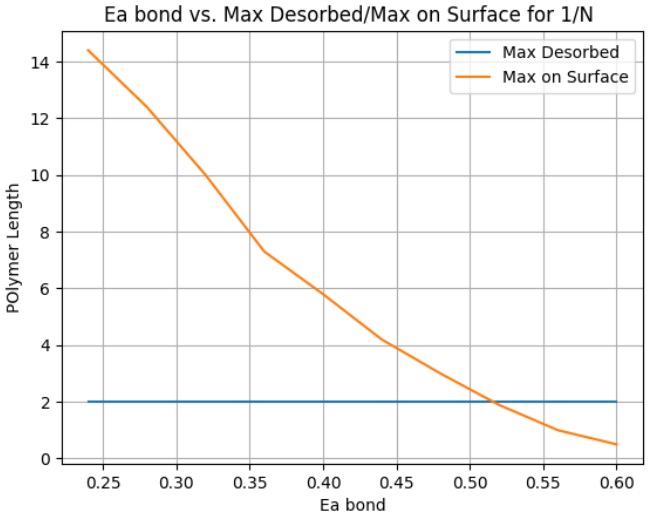

Figure 3.5: Maximum length polymer on the Surface for various activation energies and desorption relationships

For bond-forming activation energies above .36 eV, there is almost no difference in the maximum adsorbed polymer across the various models. At lower activation energies there appears to be a slightly larger difference between the models. In the model with constant desorption probabilities, at low energies the largest desorbed polymer becomes quite large, this could potentially cause the largest polymer on the surface to differ slightly. In general, the results are consistent with our understanding of the growth of polymers on the surface. The relationship between the maximum desorbed and the surface for each model is shown below. It is important to note that in figures a), b), and c) the maximum polymer on the surface drops below 2 for high activation energies. This is due to the averaging of the results over multiple trials and indicates that there were some trials for which there were no polymers on the surface at the end of the simulation.

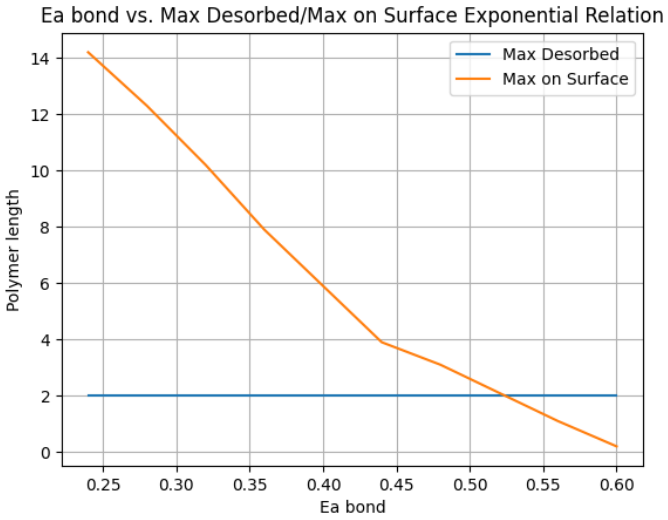
Figure 3.6



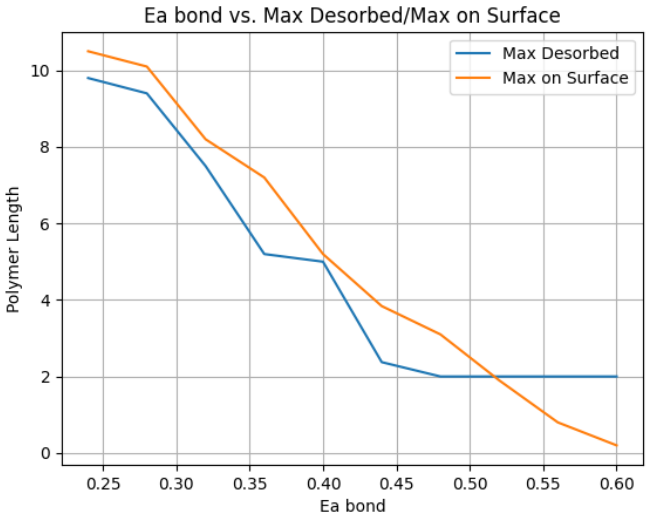

(a) Maximum desorbed polymer and polymer on the surface for desorption independent of length




(b) Maximum desorbed polymer and polymer on the surface for desorption proportional to 1/N




(c) Maximum Desorbed polymer and polymer on the surface for desorption activation energy proportional to $N \times Ea_{\mathrm{dM}}$ with $Ea_{\mathrm{dM}} = 0.3$ eV




(d) Maxium desorbed polymer and polymer on the surface for the desorption probability based on the "buckling function"

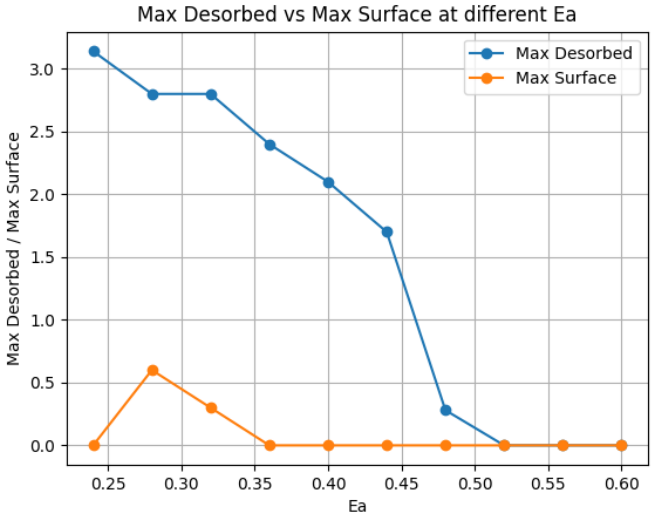
From Figure 3.6 it can be seen that in general, the maximum polymer on the surface seems to be constant across models. This is interesting because it seems to imply the polymer desorption mechanism does not greatly affect the maximum length of polymers formed on the surface. However, this result should be interpreted along with the fact that the number

of long polymers on the surface does change with the desorption model. For Figures 3.6 b and c there is no polymer desorption, only dimers are desorbing from the surface, indicating that these models will not result in significant long polymer desorption. Figures a) and d) both show that events of long polymers desorbing occurred. However, the plot in d) appears to have a slightly lower cut-off for desorbed polymers. In d) polymers greater than 2 units don't desorb until energies less than 0.42 are reached. Under a constant desorption relationship, as shown in Figure a) polymers greater than 2 units begin desorbing from the surface below 0.52 eV.

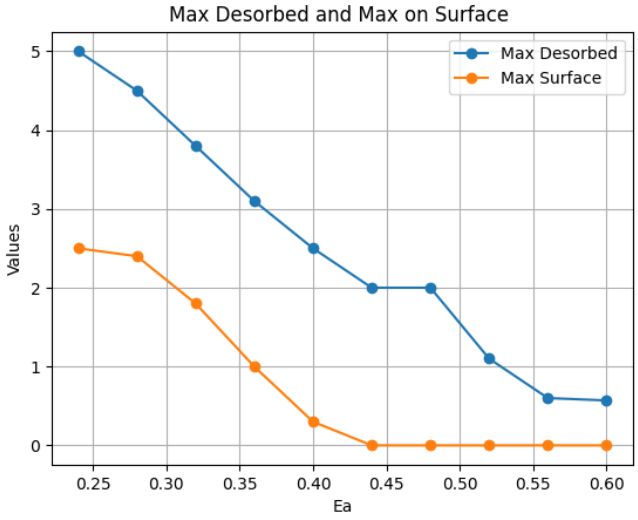
Finally, we considered the possibility that in the wet phase, the polymer desorption activation energy followed the rule $N \times Ea_{dM}$, but the value of $Ea_{dM}$ is much lower than our default value of 0.3 eV. This may be the case in the wet phase where the solvation effect is stronger than we have allowed for. In Figure 3.8a we show the results where we set $Ea_{dM} = 0.03$ eV in the wet phase, but where we have kept the default values for the dry phase. Despite the much lower barrier for desorption of the polymers, we do not see significant polymer formation in this case. The reason is simply that as we decrease $Ea_{dM}$ we are also increasing the probability of monomer desorption, and thus (for a given flux of monomers) the surface concentration of monomers decreases inhibiting the formation of long polymers. Increasing the flux of monomers to the surface would presumably counter to some extent this decrease in monomer surface concentration. When we increase the flux of monomers by a factor of 5 we observe a 2/3 increase for an activation energy of 0.24 eV (N = 5 versus N = 3). See Figure 3.8. In addition, we tried decreasing the activation for bonding in the dry phase to 0.16 eV under these conditions. We observed a desorbed average polymer length of N = 4.5, which though larger than the N = 3 result at 0.24 eV, demonstrates that large increases in polymer length are not achievable by reducing the bond activation energy under conditions where the desorption of polymers is high. It's important to note that at bonding activation energy of 0.16 eV, we are approaching the thermodynamic limit, which is

the enthalpy of the reaction. We conclude that if the desorption energy of polymers follows the rule $N \times Ea_{dM}$ , the desorption of large polymers is difficult to achieve, at least with the number of wet-dry cycles considered in this work.

Figure 3.8: Graphs of Desorbed and Surface Polymers for Varying Flux




(a) Maximum Desorbed Polymer and polymer on the surface with flux of 1 monomer per 5 timesteps




(b) Maximum Desorbed Polymer and polymer on the surface with a flux of 1 monomer per timestep

# Chapter 4

# Discussion and Future Work

The results from Chapter 3 provide insights into the characteristics of this model. It is clear that for the specified parameters, the length of the maximum desorbed polymer is highly dependent on the number of cycles run. This is supported by the results that largest polymer that was formed was for the trial with the most cycles run. Furthermore we see a dependence of the maximum polymer length with the bonding activation energy. With our default activation energy of 0.24, the maximum desorbed polymer was on average 23 units long, where as for 0.60 eV only dimers formed. This is consistent with our understanding that as the activation energy increases, the barrier for bond formation is higher and therefore less probable. For our lowest activation energy of 0.16 eV, there were polymers greater than 7 units being formed for all the various trials we did with differing numbers of cycles. Additionally for all of the trails with greater than 30 cycles, for an activation energy of 0.16, polymers greater than 10 units formed. This is significant because it has been speculated that in order to facilitate the origin of life, biopolymers would have needed to be a minimum of 10 units long [2]. The trend depicting an increase in polymer length over cycles is promising, suggesting that given sufficient time, polymers of critical length could plausibly emerge.

Under our assumptions, we see that the activation energy in general needs to be less than

0.32 eV to generate any polymers of significant length. However, it is possible that if the number of cycles was increased significantly, a larger polymer would form at higher activation energies. Beyond 0.50 eV bond activation energy, our data shows that regardless of the number of cycles there will not be large polymers forming or desorbing. Providing a maximum value of the bond activation energy for which the surface can catalyze the bond-forming reaction.

It is reasonable to expect that as the polymer length grows, it experiences an increasingly strong attraction to the surface, making it less likely to desorb. Our results indicate that for monomer desorption energies in the tenths of eV range, it is indeed difficult to get polymers of any significant length to desorb, even when allowing for a modest decrease in the desorption energy in the wet phase due to solvation. The polymer desorption will be enhanced by lowering the monomer desorption energy. However, this introduces another problem, in that the surface concentration of monomers will now decrease, resulting in less polymer formation on the surface, for a given bond activation energy.

It is of interest to compare our results with some experimental work that has been reported in the literature. Most studies to date have not directly detected long polypeptide chains on surfaces, but a number of students have demonstrated the formation of peptide bonds catalyzed by mineal surfaces. El Samrout et al. observed clear evidence for glycine-glycine bonds on amorphous silica [14]. Lahav et al. report similar findings on clay minerals using wet dry cycles. Kitadai and Nishiuchi found clear evidence of aspartate dimerization on the mineral goethite [15]. Liu and Rihe claim to have observed polymers up to 45 units of B-glutamic acid on hydroxylapatite [16]. In general, many studies have observed peptide bond formation on surfaces, but in general very long biopolymers have not been observed. Nonetheless such studies can provide potentially valuable insights and fundamental data for models and inspire further experiments.

In this paper, we have developed and tested a simplified model for biopolymer formation on surfaces, nominally using conditions appropriate for polypeptide formation. The model predicts expected trends with the activation energy of bond formation, the number of wet-dry cycles, and desorption models for the surface-produced polymers. Of necessity, most of the input parameters into the model are speculative but can be adjusted as more relevant experimental or theoretical data becomes available.

Future developments of the model will include the incorporation of effects neglected here, such as bond-breaking events, polymer diffusion, and inclusion of various species (with potentially different desorption and bond energies). Further automation of the program to provide a more comprehensive exploration of the parameter space can lead to further insights.

Finally, more attention needs to be given to time-scale issues. Diffusion, desorption, and bonding do not necessarily occur on the same time scale as the adsorption of monomers and may even be vastly different than the total time we simulate for the wet and dry cycles. We need to recognize that the time scales for the formation of biopolymers could occur over many years, far more than those simulated here if we assume that one wet and dry cycle corresponds to one day. This is a common problem in simulations, and there are so-called multi-scale approaches that can help.

# Bibliography

[1] Miller, S. L. (1953). A production of amino acids under possible primitive earth conditions. *Science*, **117**(3046), 528-529.

[2] Grover, M. A., et al. (2015). A chemical engineering perspective on the origins of life. *Processes*, **3**(2), 309-338.

[3] Tsioptsias, C. (2023). Thermodynamic and vibrational aspects of peptide bond hydrolysis and their potential relationship to the harmfulness of infrared radiation. *Molecules*, **28**(23), 7902.

[4] Lambert, J.-F. (2008). Adsorption and polymerization of amino acids on mineral surfaces: a review. *Origins of Life and Evolution of Biospheres*, **38**, 211-242.

[5] Bedoin, L., Alves, S., & Lambert, J.-F. (2020). Origins of life and molecular information: Selectivity in mineral surface-induced prebiotic amino acid polymerization. *ACS Earth and Space Chemistry*, **4**(10), 1802-1812.

[6] Bernal, J. D. (1951). *The physical basis of life.* Routledge and Paul.

[7] Weston, R. E., Schwarz, H. A. (1972). Chemical kinetics. Prentice-Hall.

[8] Keith J. Laidler and M. Christine King, The Development of Transition-State Theory, J. Phys. Chem. 1983, 87, 2657-2664.

[9] 1. Zangwill A. Physics at Surfaces. Cambridge University Press; 1988.

[10] Arnrich, S., Kalies, G., Bräuer, P. (2010). Studies on adsorption energy distributions computation from adsorption isotherms by the ansatz method. Applied surface science, 256(17), 5198-5203.

[11] Kittel, C., Kroemer, H. (1980). Thermal physics. Macmillan.

[12] Furuya, K., Hama, T., Oba, Y., Kouchi, A., Watanabe, N., & Aikawa, Y. (2022). Diffusion activation energy and desorption activation energy for astrochemically relevant species on water ice show no clear relation. *The Astrophysical Journal Letters*, **933**(1), L16.

[13] Tsioptsias, C. (2023). Thermodynamic and vibrational aspects of peptide bond hydrolysis and their potential relationship to the harmfulness of infrared radiation. *Molecules*, **28**(23), 7902.

[14] El Samrout, O., Fabbiani, M., Berlier, G., Lambert, J. F., Martra, G. (2022). Emergence of Order in Origin-of-Life Scenarios on Mineral Surfaces: Polyglycine Chains on Silica. Langmuir, 38(50), 15516-15525.

[15] Lahav, N., White, D., Chang, S. (1978). Peptide formation in the prebiotic era: thermal condensation of glycine in fluctuating clay environments. Science, 201(4350), 67-69.

[16] Liu, R., & Orgel, L. E. (1998). Polymerization on the Rocks: -amino Acids and Arginine. *Origins of Life and Evolution of the Biosphere*, **28**, 245-257.

[17] Kitadai, N., & Nishiuchi, K. (2019). Thermodynamic Impact of Mineral Surfaces on Amino Acid Polymerization: Aspartate Dimerization on Goethite. *Astrobiology*, **19**(11).

# .1 Appendix A

**Algorithm 1** Full Simulation in C++ (Part 1)

```
Initialize arrays: particles[], polymers[] (both initially empty)
Define total number of cycles and time steps
Let dryProbabilities be a map containing the probabilities of each event: desorb, diffuse, bond, or stay in the dry state
Let wetProbabilities be a map containing the probabilities of each event in the wet state.
for each cycle do
    Aqueous step:
    for each time step from t = 0 to t = TotalTime incrementing by t++ do
        Call adsorption function to add monomers to the surface
        for each particle on the surface do
            if not bonded then
                Let randomNumber be a randomly generated number between 0 and 1
                Let eventType be the type of event selected
                eventType ← SelectEvent(wetProbabilities, randomNumber)
                if eventType = desorb then
                    DESORB(particle) {Call desorb function}
                    Remove particle from particles[]
                else if eventType = diffuse then
                    MOVE(particle) {Call move function}
                else if eventType = bond then
                    BOND(particle) {Call bond function}
                end if
            end if
        end for
        for each polymer in polymers[] do
            Choose random event: desorb, diffuse, break, bond, or stay
            if eventType = desorb then
                DESORB(polymer) {Call desorb function}
                Remove particle from polymers[]
            else if eventType = diffuse then
                MOVE(polymer) {Call move function}
            else if eventType = bond then
                BOND(particle) {Call bond function}
            else if eventType = break then
                BREAK(polymer) {Call Break function}
            end if
        end for
    end for
end for=0
```

**Algorithm 2** Full Simulation in C++ (Part 2)

**for** each cycle (continued) **do**
  Dry Stage:
  **for** each time step from $t = 0$ to $t =$ TotalTime incrementing by $t++$ **do**
    **for** each particle on the surface **do**
      **if** not bonded **then**
        Let $randomNumber$ be a randomly generated number between 0 and 1
        Let $eventType$ be the type of event selected
        $eventType \leftarrow$ SELECTEVENT($dryProbabilities, randomNumber$)
        **if** $eventType =$ desorb **then**
          DESORB(particle) {Call desorb function}
          Remove particle from particles[]
        **else if** $eventType =$ diffuse **then**
          MOVE(particle) {Call move function}
        **else if** $eventType =$ bond **then**
          BOND(particle) {Call bond function}
        **end if**
      **end if**
    **end for**
    **for** each polymer in polymers[] **do**
      Choose random event: desorb, diffuse, break, bond, or stay
      **if** $eventType =$ desorb **then**
        DESORB(polymer) {Call desorb function}
        Remove polymer from polymers[]
        add polymer to desorbedPolymers[]
      **else if** $eventType =$ diffuse **then**
        MOVE(polymer) {Call move function}
      **else if** $eventType =$ bond **then**
        BOND(polymer) {Call bond function}
      **end if**
    **end for**
  **end for**
**end for**=0

# .2 Appendix B

```
#include <iostream>
#include <vector>
#include <fstream>
```

```
#include <utility>
#include <set>
#include<filesystem>
#include <random>
#include <map>

using namespace std;
int dim = 1000; //Dimensions of grid
int DIM = dim - 1;

//when the neighbor array includes 4 nieghbors, randomly selects one neighbor to collide
    with
void collideW4(int x, int y, vector<int> &nArrX, vector<int> &nArrY, int& CollideX, int&
    CollideY){
    srand(static_cast<unsigned int>(time(nullptr)));

    // Generate a random integer between 0 and 3
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(0, 3);

    // Generate a random integer between 0 and 3
    int RND = distribution(gen);

    CollideX = nArrX[RND];
    CollideY = nArrY[RND];

}

//when the neighbor array includes 3 nieghbors, randomly selects one neighbor to collide
    with
void collideW3(int x, int y, vector<int> &nArrX, vector<int> &nArrY, int& CollideX, int&
    CollideY){
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(0, 2);

    // Generate a random integer between 0 and 3
    int RND = distribution(gen);
    CollideX = nArrX[RND];
    CollideY = nArrY[RND];
```

```
}

//when the neighbor array includes 2 nieghbors, randomly selects one neighbor to collide
    with
void collideW2(int x, int y, vector<int> &nArrX, vector<int> &nArrY, int& CollideX, int&
    CollideY){

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(0, 1);

    // Generate a random integer between 0 and 1
    int RND = distribution(gen);
    CollideX = nArrX[RND];
    CollideY = nArrY[RND];
}

//when the neighbor array includes 1 nieghbors, randomly selects one neighbor to collide
    with
void collideW1(int x, int y, vector<int> &nArrX, vector<int> &nArrY, int& CollideX, int&
    CollideY){

    CollideX = nArrX[0];
    CollideY = nArrY[0];

}

bool checkEdge(int x, int y){
    if( x == 0 || y == 0 || y == DIM || x == DIM){
        return true;
    }
    return false;
}

template <typename TwoD>
//Checks if the squares next to location x,y are occupied based on the occupation 2d array
void checkNeighborsToBond(int x, int y, TwoD& array, vector<int> &nArrX, vector<int> &nArrY)
    {
    //first make sure its not an edge
    if(0<x && x<DIM && 0<y && y<DIM){
```

```
    if(array[x][y+1] == 1  || array[x][y+1] == 3){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(y+1);

    }
    if(array[x][y-1] == 1 ||array[x][y-1] == 3){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
    if((array)[x+1][y] == 1 || (array)[x+1][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(x+1);
        nArrY.push_back(y);
    }
    if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
        //if there is a nigherbo to the left
        nArrX.push_back(x-1);
        nArrY.push_back(y);
    }

}
// if its at the top but not a corner
else if(y== DIM && 0<x && x<DIM){
    if((array)[x][0] == 1 || (array)[x][0] == 3){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(0);

    }
    if((array)[x][y-1] == 1 ||(array)[x][y-1] == 3){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);

    }
    if((array)[x+1][y] == 1 ||(array)[x+1][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(x+1);
        nArrY.push_back(y);
```

```
        }
        if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }

    //if its at the bottom but not a corner
    else if(y == 0 && 0<x && x<DIM){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
        }
        if((array)[x][DIM] == 1 || (array)[x][DIM] == 3 ){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(DIM);
        }
        if((array)[x+1][y] == 1 || (array)[x+1][y] ==3){
            // if there is a nieghor to hte right
            nArrX.push_back(x+1);
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }

// if its on the left but not a corner
    else if(x == 0 && 0<y && y<DIM){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
```

```
    }
    if((array)[x][y-1] == 1 || (array)[x][y-1] == 3){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
    if((array)[x+1][y] == 1 || (array)[x+1][y] == 3){
        // if there is a nieghor to hte right
        nArrX.push_back(x+1);
        nArrY.push_back(y);
    }
   if((array)[DIM][y] == 1 || (array)[DIM][y] == 3){
        //if there is a nigherbo to the left
        nArrX.push_back(DIM);
        nArrY.push_back(y);
    }

}

// if its on the right but not a corner
else if(x == DIM && 0<y && y<DIM){
    if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(y+1);
    }
    if((array)[x][y-1] == 1 ||(array)[x][y-1] == 3 ){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
   if((array)[0][y] == 1 || (array)[0][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(0);
        nArrY.push_back(y);
    }
    if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
        //if there is a nigherbo to the left
        nArrX.push_back(x-1);
        nArrY.push_back(y);
    }
```

```
}

//if its at the top left corner
else if(x==0 && y== DIM){
    if((array)[x][0] == 1 || (array)[x][0] == 3 ){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(0);
    }
    if((array)[x][y-1] == 1 || (array)[x][y-1] == 3){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
   if((array)[x+1][y] == 1 || (array)[x+1][y] == 3){
        // if there is a nieghor to hte right
        nArrX.push_back(x+1);
        nArrY.push_back(y);
    }
    if((array)[DIM][y] == 1 || (array)[DIM][y] == 3){
        //if there is a nigherbo to the left
        nArrX.push_back(DIM);
        nArrY.push_back(y);
    }

}
//if its at the top right corner
else if(x == DIM && y == DIM){
    if((array)[x][0] == 1 || (array)[x][0] == 3){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(0);
    }
    if((array)[x][y-1] == 1 || (array)[x][y-1] == 3 ){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
    if((array)[0][y] == 1 || (array)[0][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(0);
```

```
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 1 ||(array)[x-1][y] == 3){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }
    //if its at the bottom left corner
    else if(x==0 && y==0){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
        }
        if((array)[x][DIM] == 1 || (array)[x][DIM] == 3){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(DIM);
        }
        if((array)[x+1][y] == 1 || (array)[x+1][y] == 3 ){
            // if there is a nieghor to hte right
            nArrX.push_back(x+1);
            nArrY.push_back(y);
        }
        if((array)[DIM][y] == 1 || (array)[DIM][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(DIM);
            nArrY.push_back(y);
        }
    }
    //if its at the bottom right corner
    else if(y==0 && x==DIM){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
        }
        if((array)[x][DIM] == 1 || (array)[x][DIM] == 3 ){
            //if there is a nieghbor below
```

```
            nArrX.push_back(x);
            nArrY.push_back(DIM);
        }
        if((array)[0][y] == 1 ||(array)[0][y] == 3){
            // if there is a nieghor to hte right
            nArrX.push_back(0);
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }

}

template <typename TwoD>
//Checks if the squares next to location x,y are occupied based on the occupation 2d array
void checkNeighborsToMove(int x, int y, TwoD& array, vector<int> &nArrX,vector<int> &nArrY )
    {
    //first make sure its not an edge
    //move up = 0
    //movedown = 1
    //moveleft = 2
    //moveright = 3
    if(0<x && x<DIM && 0<y && y<DIM){
        if(array[x][y+1] == 0 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);


        }
        if(array[x][y-1] == 0 ){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(y-1);
        }
        if((array)[x+1][y] == 0  ){
```

```
            // if there is a nieghor to hte right
            nArrX.push_back(x+1);
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 0 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }
// if its on the left but not a corner
    else if(x == 0 && 0<y && y<DIM){
        if((array)[x][y+1] == 1 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
        }
        if((array)[x][y-1] == 1 ){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(y-1);
        }
        if((array)[x+1][y] == 1 || (array)[x+1][y] == 3){
            // if there is a nieghor to hte right
            nArrX.push_back(x+1);
            nArrY.push_back(y);
        }
       if((array)[DIM][y] == 1 || (array)[DIM][y] == 3){
            //if there is a nigherbo to the left
            nArrX.push_back(DIM);
            nArrY.push_back(y);
        }

    }

    // if its on the right but not a corner
    else if(x == DIM && 0<y && y<DIM){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
```

```
            nArrY.push_back(y+1);
        }
        if((array)[x][y-1] == 1 ||(array)[x][y-1] == 3 ){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(y-1);
        }
       if((array)[0][y] == 1 || (array)[0][y] == 3 ){
            // if there is a nieghor to hte right
            nArrX.push_back(0);
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }
    }

    //if its at the top left corner
    else if(x==0 && y== DIM){
        if((array)[x][0] == 1 || (array)[x][0] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(0);
        }
        if((array)[x][y-1] == 1 || (array)[x][y-1] == 3){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(y-1);
        }
       if((array)[x+1][y] == 1 || (array)[x+1][y] == 3){
            // if there is a nieghor to hte right
            nArrX.push_back(x+1);
            nArrY.push_back(y);
        }
        if((array)[DIM][y] == 1 || (array)[DIM][y] == 3){
            //if there is a nigherbo to the left
            nArrX.push_back(DIM);
            nArrY.push_back(y);
        }
```

```
}
//if its at the top right corner
else if(x == DIM && y == DIM){
    if((array)[x][0] == 1 || (array)[x][0] == 3){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(0);
    }
    if((array)[x][y-1] == 1 || (array)[x][y-1] == 3 ){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(y-1);
    }
    if((array)[0][y] == 1 || (array)[0][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(0);
        nArrY.push_back(y);
    }
    if((array)[x-1][y] == 1 ||(array)[x-1][y] == 3){
        //if there is a nigherbo to the left
        nArrX.push_back(x-1);
        nArrY.push_back(y);
    }

}
//if its at the bottom left corner
else if(x==0 && y==0){
    if((array)[x][y+1] == 1 || (array)[x][y+1] == 3){
        //if there is a neighbor above
        nArrX.push_back(x);
        nArrY.push_back(y+1);
    }
    if((array)[x][DIM] == 1 || (array)[x][DIM] == 3){
        //if there is a nieghbor below
        nArrX.push_back(x);
        nArrY.push_back(DIM);
    }
    if((array)[x+1][y] == 1 || (array)[x+1][y] == 3 ){
        // if there is a nieghor to hte right
        nArrX.push_back(x+1);
```

```
            nArrY.push_back(y);
        }
        if((array)[DIM][y] == 1 || (array)[DIM][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(DIM);
            nArrY.push_back(y);
        }
    }
    //if its at the bottom right corner
    else if(y==0 && x==DIM){
        if((array)[x][y+1] == 1 || (array)[x][y+1] == 3 ){
            //if there is a neighbor above
            nArrX.push_back(x);
            nArrY.push_back(y+1);
        }
        if((array)[x][DIM] == 1 || (array)[x][DIM] == 3 ){
            //if there is a nieghbor below
            nArrX.push_back(x);
            nArrY.push_back(DIM);
        }
        if((array)[0][y] == 1 ||(array)[0][y] == 3){
            // if there is a nieghor to hte right
            nArrX.push_back(0);
            nArrY.push_back(y);
        }
        if((array)[x-1][y] == 1 || (array)[x-1][y] == 3 ){
            //if there is a nigherbo to the left
            nArrX.push_back(x-1);
            nArrY.push_back(y);
        }

    }

}


class Particle
{public:
    Particle() : x(NULL), y(NULL), name("Null"), num(NULL) {}
    Particle(int X, int Y,string Name, int partNum,bool recombined) : x(X), y(Y), name(Name)
        , num(partNum), recombined(recombined) {}
```

```
bool recombined;
bool readyToBond = false;

int getX(){
    return x;
}

int getY()
{
    return y;
}

void setBonded(bool b){
    bonded = b;
}

bool getBonded(){
    return bonded;
}

void setX(int X){
    x = X;
}

void setParticleNumber(int i){
    num = i;
}

int getParticleNumber()
{
    return num;
}
void setY(int Y){

    y = Y;
}

string getName()
{
    return name;
```

```
}

void move(){
    if(0 < y && y < DIM && 0 < x && x < DIM){
        //move any direction

        float RND = (float) rand()/RAND_MAX;
        if(RND <= 0.25){
            y = y + 1;
        }
        else if (RND > 0.25 && RND <= 0.5){
            x = x + 1;
        }
        else if(RND > 0.5 && RND <= 0.75){
            x = x - 1;
            }
        else{
            y = y - 1;
        }
    }

    else if (y == DIM && 0 < x && x < DIM){
        // top edge not corner
        float RND = (float) rand()/RAND_MAX;
        if(RND <= 0.25){
            y = 0;
        }
        else if (RND > 0.25 && RND <= 0.5){
            x = x + 1;
        }
        else if(RND > 0.5 && RND <= 0.75){
            x = x - 1;
            }
        else{
            y = y - 1;
        }
    }

    else if(x == DIM && 0 < y && y < DIM){
        //Right Edge
        float RND = (float) rand()/RAND_MAX;
```

```
    if(RND <= 0.25){
        y = y + 1;
    }
    else if (RND > 0.25 && RND <= 0.5){
        x = x - 1;
    }
    else if(RND > 0.5 && RND <= 0.75){
        x = 0;
        }
    else{
        y = y - 1;
    }
}

else if(y == 0 && 0 < x && x< DIM)
{
    // Bottom edge
    float RND = (float) rand()/RAND_MAX;
    if(RND <= 0.25){
        y = y + 1;
    }
    else if (RND > 0.25 && RND <= 0.5){
        x = x + 1;
    }
    else if(RND > 0.5 && RND <= 0.75){
        x = x - 1;
        }
    else{
        y = DIM;
    }
}

else if(x == 0 && 0 < y && y < DIM){
    // Left edge
    float RND = (float) rand()/RAND_MAX;
    if(RND <= 0.25){
        y = y + 1;
    }
    else if (RND > 0.25 && RND <= 0.5){
        x = x + 1;
    }
```

```
    else if(RND > 0.5 && RND <= 0.75){
        x = DIM;
        }
    else{
        y = y - 1;
    }
}

else if(x == 0 && y == 0){
    //Bottom left corner
    float RND = (float) rand()/RAND_MAX;
    if(RND <= 0.25){
        y = y + 1;
    }
    else if (RND > 0.25 && RND <= 0.5){
        x = x + 1;
    }
    else if(RND > 0.5 && RND <= 0.75){
        x = DIM;
        }
    else{
        y = DIM;
    }
}

else if(x==0 && y == DIM){
    //Top left corner
    float RND = (float) rand()/RAND_MAX;
    if(RND <= 0.25){
        y = 0;
    }
    else if (RND > 0.25 && RND <= 0.5){
        x = x + 1;
    }
    else if(RND > 0.5 && RND <= 0.75){
        x = DIM;
        }
    else{
        y = y - 1;
    }
}
```

```
        else if(x == DIM && y == DIM){
            // top right corner
            float RND = (float) rand()/RAND_MAX;
            if(RND <= 0.25){
                y = 0;
            }
            else if (RND > 0.25 && RND <= 0.5){
                x = 0;
            }
            else if(RND > 0.5 && RND <= 0.75){
                x = x - 1;
                }
            else{
                y = y - 1;
            }
        }
        else if(x == DIM && y == 0){
            // x==dim y==0
            float RND = (float) rand()/RAND_MAX;
            if(RND <= 0.25){
                y = y + 1;
            }
            else if (RND > 0.25 && RND <= 0.5){
                x = 0;
            }
            else if(RND > 0.5 && RND <= 0.75){
                x = x - 1;
                }
            else{
                y = dim - 1;
            }
        }


    }

private:
    int x,y,num;
    string name;
    bool bonded;
```

```
};


int findParticle(int x, int y, vector<Particle*> &pArray) {
    for (int i = 0; i < pArray.size(); i++) {
        if (pArray[i]->getX() == x && pArray[i]->getY() == y) {
            return i;
        }
    }
    return -1;
}

class Chain
{
private:
    int length, orientation;
    vector<Particle*> memberArr;
    Particle* leadParticle;
    Particle* bottomParticle;
    vector<int> xBondingSites;
    vector<int> yBondingSites;
public:
    Chain(){}
    set<pair <int,int> > bondingSites;
    bool readyToBond = false;
    int particleToBond = -1;
    int polymerToBond = -1;


    vector<Particle*> getMembers(){
        return memberArr;
    }

    int getLength() const {
        return memberArr.size();
    }

    void setBondingSites(int x, int y) {
        xBondingSites.push_back(x);
        yBondingSites.push_back(y);
    }
```

```
void clearBondingSites(){
    yBondingSites.clear();
    xBondingSites.clear();
}

vector<int> getXbondingSites(){
     return xBondingSites;
}


vector<int> getYbondingSites(){
    return yBondingSites;
}

bool isPointInBondingSites(int x, int y) const {
    pair<int, int> pointToFind = make_pair(x, y);
    return (bondingSites.find(pointToFind) != bondingSites.end());
}


void addMember(Particle& part){
    if(memberArr.size() == 0){
        setLeadParticle(part);
        setBottomParticle(part);
    }
    else if(memberArr.size() >= 1){
        if(leadParticle->getX() < part.getX()){

            setLeadParticle(part);
        }
        else if(bottomParticle->getX() > part.getX()){
            setBottomParticle(part);
        }
        if(leadParticle->getY() < part.getY()){
            setLeadParticle(part);
        }
        else if(bottomParticle->getY() > part.getY()){
            setBottomParticle(part);
        }
    }
```

```
        memberArr.push_back(&part);
    }

    void removeMember(int i){
        memberArr.erase(memberArr.begin() + i);
    }

    void resetLeadAndBottom(){
        setLeadParticle(*memberArr[0]);
        setBottomParticle(*memberArr[0]);

        for(int i =0; i < this->memberArr.size(); i++){
//            cout << " member x and y " << this->memberArr[i]->getX() << " , " <<  this->
    memberArr[i]->getY() << endl;
            if(this->memberArr[i]->getX() > getLeadParticle()->getX() || this->memberArr[i
                ]->getY() > getLeadParticle()->getY()){
                setLeadParticle(*memberArr[i]);
//                  cout << "set top " << endl;
            }
            if(this->memberArr[i]->getX() < getBottomParticle()->getX() || this->memberArr[i
                ]->getY() < getBottomParticle()->getY()){
//                  cout << " set bottom " << this->memberArr[i]->getX() << " ,  " << this->
    memberArr[i]->getY() << endl;
                setBottomParticle(*memberArr[i]);
            }
        }


    }


    void setLeadParticle(Particle& p){
        leadParticle = &p;
    }

    void setBottomParticle(Particle& p){
        bottomParticle = &p;
    }

    Particle* getLeadParticle(){
        return leadParticle;
```

```
}

Particle* getBottomParticle(){
    return bottomParticle;
}

int getOrientation(){
    // 1 if up down 0 if left right
    int topX = getLeadParticle()->getX();
    int topY = getLeadParticle()->getY();
    int bottomX = getBottomParticle()->getX();
    int bottomY = getBottomParticle()->getY();

    if(topX - bottomX == 0){
        return 1;
    }

    if(topY - bottomY == 0){
        return 0;
    }
    return 5;
}

template <typename TwoD>
bool getTopNieghbor(TwoD& grid){
    if(orientation == 1){
        if(grid[getLeadParticle()->getX()][getLeadParticle()->getY() + 1 ] == 1){

        }
        else if(grid[getLeadParticle()->getX()][getLeadParticle()->getY() + 1 ] == 2){

        }
    }
    if(orientation == 0){
        if(grid[getLeadParticle()->getX() + 1][getLeadParticle()->getY() ] == 1){

        }
        else if(grid[getLeadParticle()->getX() + 1][getLeadParticle()->getY()] == 2){

        }
    }
```

```
}

template <typename TwoD>
bool getBottomNieghbor(TwoD& grid){
    if(orientation == 1){
        if(grid[getLeadParticle()->getX()][getLeadParticle()->getY() - 1 ] == 1){
            return true;
        }
        else if(grid[getLeadParticle()->getX()][getLeadParticle()->getY() - 1 ] == 2){
            return true;
        }
    }
    if(orientation == 0){
        if(grid[getLeadParticle()->getX() - 1][getLeadParticle()->getY() ] == 1){
            return true;
        }
        else if(grid[getLeadParticle()->getX() - 1][getLeadParticle()->getY()] == 2){
            return true;
        }
    }
}


template <typename TwoD>
void updateGrid(TwoD& grid, vector<Particle*> &particles){

    int topX = getLeadParticle()->getX();
    int topY = getLeadParticle()->getY();
    int bottomX = getBottomParticle()->getX();
    int bottomY = getBottomParticle()->getY();
    int o =  getOrientation();
    grid[topX][topY] = 3;
    grid[bottomX][bottomY] = 3;

    if(checkEdge(topX, topY) == false && checkEdge(bottomX, bottomY) == false){
        grid[topX][topY] = 3;
        grid[bottomX][bottomY]  = 3;
        for(int i = 0; i < memberArr.size(); i++){
            if (!(memberArr[i]->getX() == topX && memberArr[i]->getY() == topY) &&
                !(memberArr[i]->getX() == bottomX && memberArr[i]->getY() == bottomY)) {
                grid[memberArr[i]->getX()][memberArr[i]->getY()] = 2;
```

```
                }
            }
        }
    }


};


int generateRandomNumber(int min, int max) {
    return min + std::rand() % (max - min + 1);
}


template <typename TwoD>
void populate(int nP, std::vector<Particle*>& pArray, TwoD& array) {
    std::srand(static_cast<unsigned>(std::time(nullptr))); // Seed the random number
        generator

    int i = 0;
    while (i < nP) {
        int RND1 = generateRandomNumber(dim * 2 / 10, dim * 8 / 10); // Adjusted range for
            the middle 80% of the x dimension
        int RND2 = generateRandomNumber(dim * 2 / 10, dim * 8 / 10);

//         cout << RND1 << " < " << RND2 << endl;
//         int RND1 = generateRandomNumber(500, 2000); // Adjusted range for the middle 80%
    of the x dimension
//         int RND2 = generateRandomNumber(500, 2000);
        // Check if the position is within the bounds of the array
        if (RND1 >= 0 && RND1 < dim && RND2 >= 0 && RND2 < dim) {
            // Check if the position is already occupied
            if (array[RND1][RND2] == 0) {
                Particle* p = new Particle(RND1, RND2, "Particle " + std::to_string(i), i,
                    false);
                array[RND1][RND2] = 1;
                pArray.push_back(p);
                i++;
            }
        }
    }
```

```
}


int findPolymer(int x,int y, vector<Chain*> &pArray){
    for(int i=0; i < pArray.size(); i++){
        for(int j = 0; j< pArray[i]->getMembers().size(); j++){
            if(pArray[i]->getMembers()[j]->getX() == x &&pArray[i]->getMembers()[j]->getY()
                == y ){
                return i;
            }
        }
    }
    return -1;
}

void eraseValueFromVector(vector<int>& vec, int value) {
    int i = 0;
    int currentValue = -1;

    while (i < vec.size() && currentValue != value) {
        currentValue = vec[i];
        if (currentValue != value) {
            i++;
        }
    }

    if (i < vec.size()) {
        vec.erase(vec.begin() + i);
    }
}

template <typename TwoD>
void polymerCheckMove(int polymerIndex,TwoD& grid, Chain* &polymer, vector<Particle *> &
    particles, vector<Chain*> &polymers, vector<int> &moveArr){
    // move up = 0
    //move down = 1
    //move left = 2
    //move right = 3
    moveArr.push_back(0);
    moveArr.push_back(1);
    moveArr.push_back(2);
```

```
moveArr.push_back(3);

    //orienation is up and down
    if(polymer->getOrientation() == 1){

        for(int i =0; i < polymer->getMembers().size(); i++){
            // check move right
            if(grid[polymer->getMembers()[i]->getX() + 1][polymer->getMembers()[i]->getY
                ()] != 0 ){
                eraseValueFromVector(moveArr, 3);
                break;
            }
        }

        for(int i = 0; i < polymer->getMembers().size(); i++){
            if(grid[polymer->getMembers()[i]->getX() - 1][polymer->getMembers()[i]->getY
                ()] != 0){
                eraseValueFromVector(moveArr, 2);
                break;
            }
        }
        if(grid[polymer->getLeadParticle()->getX()][polymer->getLeadParticle()->getY() +
             1] != 0){
            eraseValueFromVector(moveArr, 0);
        }
        if(grid[polymer->getLeadParticle()->getX()][polymer->getLeadParticle()->getY() -
             1] != 0 ){
            eraseValueFromVector(moveArr, 1);
        }

    }

    else if(polymer->getOrientation() == 0){

        for(int i =0; i < polymer->getMembers().size(); i++){
            // check move up
            if(grid[polymer->getMembers()[i]->getX() ][polymer->getMembers()[i]->getY()
                + 1] != 0 ){
                eraseValueFromVector(moveArr, 0);
                break;
            }
```

```
            }
                // check move down
            for(int i = 0; i < polymer->getMembers().size(); i++){
                if(grid[polymer->getMembers()[i]->getX() ][polymer->getMembers()[i]->getY()
                    - 1] != 0){
                    eraseValueFromVector(moveArr, 1);
                    break;
                }
            }

                if(grid[polymer->getLeadParticle()->getX()+1][polymer->getLeadParticle()->
                    getY()] != 0 ){
                    eraseValueFromVector(moveArr, 3);
                }
                if(grid[polymer->getBottomParticle()->getX() - 1][polymer->getBottomParticle
                    ()->getY()]  !=0){
                    eraseValueFromVector(moveArr, 2);
                }

        }

}


template <typename TwoD>
void desorb(TwoD& grid, vector<Particle*> &particles, int& desorbCount, int index){
                    grid[particles[index]->getX()][particles[index]->getY()] = 0;
                    particles[index]->recombined = true;
                    particles.erase(particles.begin() + index);
                    desorbCount++;
}

template <typename TwoD>
void desorbPolymer(TwoD& grid, vector<Particle*> &particles, int& desorbCount, int index,
    vector<Chain*> &polymers, vector<Chain*> &desorbedPolymers){
    desorbedPolymers.push_back(polymers[index]);

    for(int j = 0; j < polymers[index]->getMembers().size(); j++){
        desorbCount++;
        int x = polymers[index]->getMembers()[j]->getX();
        int y =polymers[index]->getMembers()[j]->getY();
```

```
            grid[x][y] = 0;
            polymers[index]->getMembers()[j]->recombined = true;
            int i = findParticle(x, y, particles);
            if(i != -1){
                particles.erase(particles.begin() + i);
            }
        }
        polymers.erase(polymers.begin() + index);

}

template <typename TwoD>
void diffuse(Particle* particle, TwoD& grid ,vector<Particle*> &particles, vector<Chain*> &
    polymers) {
    vector<int> neighborArrayX;
    vector<int> neighborArrayY;

    checkNeighborsToMove(particle->getX(), particle->getY(), grid, neighborArrayX,
        neighborArrayY);
    int Cx = -1;
    int Cy = -1;
    int x = particle->getX();
    int y = particle->getY();

    if (neighborArrayX.size() == 4) {

        collideW4(x, y, neighborArrayX, neighborArrayY, Cx, Cy);

        grid[x][y] = 0;
        particle->setX(Cx);
        particle->setY(Cy);
        grid[particle->getX()][particle->getY()] = 1;
        neighborArrayX.clear();
        neighborArrayY.clear();

    } else if (neighborArrayX.size() == 3) {

        collideW3(x, y, neighborArrayX, neighborArrayY, Cx, Cy);

        grid[x][y] = 0;
        particle->setX(Cx);
```

```
            particle->setY(Cy);
            grid[particle->getX()][particle->getY()] = 1;
            neighborArrayX.clear();
            neighborArrayY.clear();
        } else if (neighborArrayX.size() == 2) {
            collideW2(x, y, neighborArrayX, neighborArrayY, Cx, Cy);
            grid[x][y] = 0;

            particle->setX(Cx);
            particle->setY(Cy);
            grid[particle->getX()][particle->getY()] = 1;

            neighborArrayX.clear();
            neighborArrayY.clear();
        } else if (neighborArrayX.size() == 1) {

            collideW1(x, y, neighborArrayX, neighborArrayY, Cx, Cy);
            grid[x][y] = 0;
            particle->setX(Cx);
            particle->setY(Cy);
            grid[particle->getX()][particle->getY()] = 1;

            neighborArrayX.clear();
            neighborArrayY.clear();
        }
    }

    template <typename TwoD>
    void polymerMove(int i,TwoD& grid ,vector<Particle*> &particles, vector<Chain*> &polymers){

        vector<int> moveArray;
        polymerCheckMove(i, grid, polymers[i], particles, polymers, moveArray);

        if(moveArray.size() == 0){
            return;
        }
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> dist(0, moveArray.size() - 1);

        int random_index = dist(gen);
```

```
int movement = moveArray[random_index];
if(movement == 0) {
    //move up
    if(polymers[i]->getOrientation() == 1){


        for(int j = 0; j < polymers[i]->getMembers().size(); j++){
            int currentX = polymers[i]->getMembers()[j]->getX();
            int currentY = polymers[i]->getMembers()[j]->getY();
            grid[currentX][currentY] = 0;
            polymers[i]->getMembers()[j]->setY(currentY + 1);
            grid[polymers[i]->getMembers()[j]->getX()][polymers[i]->getMembers()[j]->
                getY()] = 2;
        }
        polymers[i]->clearBondingSites();
        polymers[i]->updateGrid(grid, particles);
    }

    if(polymers[i]->getOrientation() == 0){


        for(int j = 0; j < polymers[i]->getMembers().size(); j++){
            int currentX = polymers[i]->getMembers()[j]->getX();
            int currentY = polymers[i]->getMembers()[j]->getY();
            grid[currentX][currentY] = 0;

            polymers[i]->getMembers()[j]->setY(currentY + 1);

        }
        polymers[i]->clearBondingSites();
        polymers[i]->updateGrid(grid, particles);
    }
} else if(movement == 1) {

    //move down
    if(polymers[i]->getOrientation() == 1){

        for(int j = 0; j < polymers[i]->getMembers().size(); j++){
            int currentX = polymers[i]->getMembers()[j]->getX();
            int currentY = polymers[i]->getMembers()[j]->getY();
            grid[currentX][currentY] = 0;
```

```
                polymers[i]->getMembers()[j]->setY(currentY - 1);
                grid[polymers[i]->getMembers()[j]->getX()][polymers[i]->getMembers()[j]->
                    getY()] = 2;
            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);

        } else if(polymers[i]->getOrientation() == 0) {

            for(int j = 0; j < polymers[i]->getMembers().size(); j++){
                int currentX = polymers[i]->getMembers()[j]->getX();
                int currentY = polymers[i]->getMembers()[j]->getY();
                grid[currentX][currentY] = 0;

                polymers[i]->getMembers()[j]->setY(currentY - 1);

            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);
        }
    }
    else if (movement == 2){
        //move left

        if(polymers[i]->getOrientation() == 1){
            for(int j = 0; j < polymers[i]->getMembers().size(); j++){
                int currentX = polymers[i]->getMembers()[j]->getX();
                int currentY = polymers[i]->getMembers()[j]->getY();
                grid[currentX][currentY] = 0;
                polymers[i]->getMembers()[j]->setX(currentX - 1);
                grid[polymers[i]->getMembers()[j]->getX()][polymers[i]->getMembers()[j]->
                    getY()] = 2;
            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);
        }
        else if(polymers[i]->getOrientation() == 0){
            for(int j = 0; j < polymers[i]->getMembers().size(); j++){
                int currentX = polymers[i]->getMembers()[j]->getX();
                int currentY = polymers[i]->getMembers()[j]->getY();
                grid[currentX][currentY] = 0;
```

```
                polymers[i]->getMembers()[j]->setX(currentX - 1);

            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);
        }
    }
    else if(movement == 3){
        //move right

        if(polymers[i]->getOrientation() == 1){

            for(int j = 0; j < polymers[i]->getMembers().size(); j++){
                int currentX = polymers[i]->getMembers()[j]->getX();
                int currentY = polymers[i]->getMembers()[j]->getY();
                grid[currentX][currentY] = 0;
                polymers[i]->getMembers()[j]->setX(currentX + 1);
                grid[polymers[i]->getMembers()[j]->getX()][polymers[i]->getMembers()[j]->
                    getY()] = 2;
            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);
        }
        else if(polymers[i]->getOrientation() == 0){

            for(int j = 0; j < polymers[i]->getMembers().size(); j++){
                int currentX = polymers[i]->getMembers()[j]->getX();
                int currentY = polymers[i]->getMembers()[j]->getY();
                grid[currentX][currentY] = 0;

                polymers[i]->getMembers()[j]->setX(currentX + 1);

            }
            polymers[i]->clearBondingSites();
            polymers[i]->updateGrid(grid, particles);
        }
    }
    moveArray.clear();
}
```

```
template <typename TwoD>
void bond(Particle* &p, TwoD& grid,vector<Particle*>& particles, vector<Chain*>& polymers){
//     cout << " P X " << p.getX() << " , " << p.getY() << endl;
    vector<int> neighborArrayX;
    vector<int> neighborArrayY;
    int Cx = -1;
    int Cy = -1;
    checkNeighborsToBond(p->getX(), p->getY(), grid, neighborArrayX, neighborArrayY);
    if(neighborArrayX.size() == 4){

        collideW4(p->getX(),p->getY(),neighborArrayX,neighborArrayY,Cx,Cy);
    }
    else if(neighborArrayX.size() == 3){
        collideW3(p->getX(),p->getY(),neighborArrayX,neighborArrayY,Cx,Cy);
    }
    else if(neighborArrayX.size() == 2){
        collideW2(p->getX(),p->getY(),neighborArrayX,neighborArrayY,Cx,Cy);
    }
    else if(neighborArrayX.size() == 1){
        collideW1(p->getX(),p->getY(),neighborArrayX,neighborArrayY,Cx,Cy);
    }
    if(grid[Cx][Cy] == 3){

        int polymerIndex = findPolymer(Cx, Cy, polymers);

        if(polymerIndex != -1){
            int orientation = polymers[polymerIndex]->getOrientation();
            if(orientation == 0 && p->getY() == polymers[polymerIndex]->getLeadParticle()->
                getY()){
                grid[p->getX()][p->getY()] = 3;
                p->setBonded(true);
                polymers[polymerIndex]->addMember(*p);
                polymers[polymerIndex]->updateGrid(grid,particles);
            }
            else if(orientation == 1 && p->getX() == polymers[polymerIndex]->getLeadParticle
                ()->getX()){
                grid[p->getX()][p->getY()] = 3;
                p->setBonded(true);
                polymers[polymerIndex]->addMember(*p);
                polymers[polymerIndex]->updateGrid(grid,particles);
```

```
            }

        }
    }
    else if(grid[Cx][Cy] == 1){
        int index = findParticle(Cx, Cy, particles);
        if(index != -1){
            Chain* poly = new Chain();

            polymers.push_back(poly);
            poly->addMember(*p);
            particles[index]->setBonded(true);
            poly->addMember(*particles[index]);
            poly->updateGrid(grid,particles);
            p->setBonded(true);
            particles[index]->setBonded(true);
        }
     }

}
double rnd() {
    // Use std::random_device to obtain a seed for the random number engine
    std::random_device rd;

    // Use std::mt19937 as the random number engine
    std::mt19937 gen(rd());

    // Use std::uniform_real_distribution to generate a random number between 0 and 1
    std::uniform_real_distribution<double> dis(0.0, 1.0);

    // Generate and return the random number
    return dis(gen);
}


template <typename TwoD>
void polymerCheckBond(int polymerIndex,TwoD& grid, Chain* &polymer, vector<Particle *> &
    particles, vector<Chain*> &polymers, vector<int> &nArrX, vector<int> &nArrY){
    int leadX = polymer->getLeadParticle()->getX();
    int leadY = polymer->getLeadParticle()->getY();
    int bottomX = polymer->getBottomParticle()->getX();
```

```
int bottomY = polymer->getBottomParticle()->getY();
if(polymer->getOrientation() == 1){
    if( grid[leadX][leadY + 1] == 1 ){
        int index = findParticle(leadX ,leadY + 1, particles);
        if(index != -1){
                nArrX.push_back(leadX);
                nArrY.push_back(leadY + 1);
        }
    }
    if(grid[leadX][leadY + 1] == 3){
        int pIndex = findPolymer(leadX, leadY + 1, polymers);
        if(pIndex != -1){
            if(polymers[pIndex]->getOrientation() == 1){
                nArrX.push_back(leadX);
                nArrY.push_back(leadY + 1);
            }
        }

    }
    if( grid[bottomX][bottomY - 1] == 1 ){
        nArrX.push_back(bottomX);
        nArrY.push_back(bottomY - 1);
    }
    if(grid[bottomX][bottomY - 1] == 3){
        int pIndex = findPolymer(bottomX, bottomY - 1, polymers);
        if(pIndex != -1){
            if(polymers[pIndex]->getOrientation() == 1){
                nArrX.push_back(bottomX);
                nArrY.push_back(bottomY - 1);
            }
        }
    }
}

if(polymer->getOrientation() == 0 ){
    if(grid[leadX + 1][leadY] == 1 ){
        nArrX.push_back(leadX + 1);
        nArrY.push_back(leadY);

    }
    if(grid[leadX + 1][leadY] == 3){
```

```
            int pIndex = findPolymer(leadX+1, leadY , polymers);
            if(pIndex != -1){
                if(polymers[pIndex]->getOrientation() == 0){
                    nArrX.push_back(leadX + 1);
                    nArrY.push_back(leadY);
                }
            }
        }
        if(grid[bottomX - 1][bottomY] == 1){
            nArrX.push_back(bottomX - 1);
            nArrY.push_back(bottomY);

        }
        if(grid[bottomX - 1][bottomY] == 3){
            int pIndex = findPolymer(bottomX-1, bottomY , polymers);
            if(pIndex != -1){
                if(polymers[pIndex]->getOrientation() == 0){
                    nArrX.push_back(bottomX - 1);
                    nArrY.push_back(bottomY);
                }
            }
        }

    }


}

template <typename TwoD>
void polymerBond(int polymerIndex,TwoD& grid, Chain* &polymer, vector<Particle *> &particles
    , vector<Chain*> &polymers){
    vector<int> neighborArrayX;
    vector<int> neighborArrayY;
    int Cx = -1;
    int Cy = -1;

    int leadX = polymer->getLeadParticle()->getX();
    int leadY = polymer->getLeadParticle()->getY();
    int bottomX = polymer->getBottomParticle()->getX();
    int bottomY = polymer->getBottomParticle()->getY();
```

```
polymerCheckBond(polymerIndex, grid, polymer, particles, polymers, neighborArrayX,
    neighborArrayY);
if(neighborArrayX.size() == 4){

    collideW4(polymer->getLeadParticle()->getX(),polymer->getLeadParticle()->getY(),
        neighborArrayX,neighborArrayY,Cx,Cy);
}
else if(neighborArrayX.size() == 3){
    collideW3(polymer->getLeadParticle()->getX(),polymer->getLeadParticle()->getY(),
        neighborArrayX,neighborArrayY,Cx,Cy);
}
else if(neighborArrayX.size() == 2){
    collideW2(polymer->getLeadParticle()->getX(),polymer->getLeadParticle()->getY(),
        neighborArrayX,neighborArrayY,Cx,Cy);
}
else if(neighborArrayX.size() == 1){
    collideW1(polymer->getLeadParticle()->getX(),polymer->getLeadParticle()->getY(),
        neighborArrayX,neighborArrayY,Cx,Cy);
}
if(grid[Cx][Cy] == 3){
    int index = findPolymer(Cx, Cy, polymers);
    if(index != -1){
        for(int j = 0; j < polymers[index]->getMembers().size(); j++){
            polymer->addMember(*polymers[index]->getMembers()[j]);
        }
        polymers.erase(polymers.begin() + index);
        polymer->clearBondingSites();
        polymer->updateGrid(grid, particles);
    }
}
else if(grid[Cx][Cy] == 1){
    int index = findParticle(Cx, Cy, particles);
    if(index != -1){
        grid[Cx][Cy] = 3;
        polymer->addMember(*particles[index]);
        particles[index]->setBonded(true);
        polymer->clearBondingSites();
        polymer->updateGrid(grid, particles);
    }
}
```

```
}

template <typename TwoD>
void breakPolymer(int i,TwoD& grid,vector<Particle *> &particles, vector<Chain*> &polymers){
    if(polymers[i]->getMembers().size() == 2){
        for(int j =0; j < polymers[i]->getMembers().size(); j++){
            polymers[i]->getMembers()[j] ->setBonded(false);
            grid[polymers[i]->getMembers()[j]->getX()][polymers[i]->getMembers()[j]->getY()]
                = 1;
        }
        polymers.erase(polymers.begin() + i);
    }
    if(polymers[i]->getMembers().size() > 2){

        float RND = rnd();

        if(RND > .5){

            int X = polymers[i]->getLeadParticle()->getX();
            int Y =polymers[i]->getLeadParticle()->getY();
            for(int j =0 ; j < polymers[i]->getMembers().size(); j++){
                if(polymers[i]->getMembers()[j]->getX() == X && polymers[i]->getMembers()[j
                    ]->getY() == Y){
                    polymers[i]->removeMember(j);
                }
            }
            grid[X][Y] = 1;
            int particleIndex = findParticle(X, Y, particles);
            if(particleIndex != -1){
                particles[particleIndex]->setBonded(false);
//                cout << " calling reset lead and bottom " << endl;
                polymers[i]->resetLeadAndBottom();
                polymers[i]->updateGrid(grid, particles);
            }
        }
        else if(RND < .5){

                int X = polymers[i]->getBottomParticle()->getX();
                int Y = polymers[i]->getBottomParticle()->getY();
                for(int j =0 ; j < polymers[i]->getMembers().size(); j++){
                    if(polymers[i]->getMembers()[j]->getX() == X && polymers[i]->
```

```
                        getMembers()[j]->getY() == Y){
                        polymers[i]->removeMember(j);
                    }
                }
                grid[X][Y] = 1;
            int particleIndex = findParticle(X, Y, particles);
        if(particleIndex != -1){
            particles[particleIndex]->setBonded(false);
            polymers[i]->resetLeadAndBottom();
            polymers[i]->updateGrid(grid, particles);
        }
            }
    }
}

string generate_event(const map<string, float>& probabilities) {

    // Generate a random number between 0 and 1

    std::random_device rd;

    std::mt19937 gen(rd());

    std::uniform_real_distribution<double> dis(0.0, 1.0);

    // Generate a random number between 0 and 1

    double rand_num = dis(gen);

// Calculate cumulative probabilities

    double cumulative_prob = 0;

    for (const auto& pair : probabilities) {

        cumulative_prob += pair.second;

        // If the random number is less than the cumulative probability,

        // the corresponding event occurs
```

```
        if (rand_num < cumulative_prob) {

            return pair.first;

        }

    }

    // If the random number exceeds the sum of all pr
    // then no event occurs

    return "None";

}

double generatePolymerDesorbProbability(int N){

    double EaMonoDiffuse = 0.18;

    double EaDesorbMono = 0.3;

    double T = 333;

    double probDesorb = 0.2 * exp(-(N * EaDesorbMono - EaMonoDiffuse)/T * 11610);

    return probDesorb;

}
double generateSinProbability(double Ea){

    double Ediff = 0.18;

    int T = 333;

    double prob = 0.2 * exp(-(Ea - Ediff)/T * 11610);

    return prob;

}

string generateMonomerProbabilitiesDry(double diffuseToNothingRatio, double
```

```
    diffuseToDesorbRatio, double diffuseToBondRatio) {
    double diffuseProbability = 0.2;
    double desorbProbability = 0.00177;
    double bondProbability = 0.2;

//    double diffuseProbability = .6;
//    double desorbProbability = 0;
//    double bondProbability = .4;
//    cout << "Diffuse Probability: " << diffuseProbability << " Desorb Probability: " <<
    desorbProbability << " BOND probability " << bondProbability << endl;
    // Store probabilities in a map
    map<string, float> monomer_probabilities = {{"Desorb", desorbProbability}, {"Diffuse",
        diffuseProbability}, {"Bond", bondProbability}};

    // Generate event using probabilities
    return generate_event(monomer_probabilities);
}

string generateMonomerProbabilitiesWet(double diffuseToNothingRatio, double
    diffuseToDesorbRatio) {
    double diffuseProbability = .2;
    double desorbProbability = 0.003048;
    double bondProbability = 0;
    map<string, float> monomer_probabilities = {{"Desorb", desorbProbability}, {"Diffuse",
        diffuseProbability}, {"Bond", bondProbability}};
    return generate_event(monomer_probabilities);
}


string generatePolymerProbabilitiesDry(double desorbToNothingRatio, double desorbToBondRatio
    ) {
    double desorbProbability = 0;
//    double desorbProbability = 0;
//    double bondProbability = 0.5;
    double bondProability = 0;
    map<string, float> polymer_probabilities = {{"Desorb", desorbProbability}, {"Bond",
        bondProability}};

    // Generate event using probabilities
    return generate_event(polymer_probabilities);
}
```

```
string generatePolymerProbabilitiesWet(double desorbToNothingRatio, double desorbToBreak,
    int length) {
//    double Ea =generatePolymerDesorbProbability(length);
//    double desorbProbability = generateSinProbability(Ea);
    double desorbProbability = 0.000002855696835;
    double breakProbability = 0;
    map<string, float> polymer_probabilities = {{"Desorb", desorbProbability}, {"Break",
        breakProbability}};
    return generate_event(polymer_probabilities);
}

string generateDiamerProbabilitiesDry(double diffuseToNothingRatio, double
    diffuseToDesorbRatio, double diffuseToBondRatio){
    double diffuseProbability = 0.00577;
    double desorbProbability = 0.0000048;
//    double desorbProbability = 0;
    double bondProbability = 0.2;

    map<string, float> diamer_probabilities = {{"Desorb", desorbProbability}, {"Diffuse",
        diffuseProbability}, {"Bond", bondProbability}};
    return generate_event(diamer_probabilities);
}

string generateDiamerProbabilitiesWet(double diffuseToNothingRatio, double
    diffuseToDesorbRatio, double diffuseToBreak ) {
    double diffuseProbability = 0.02468;
    double desorbProbability = 0.00009329;
    double breakProbability = 0;
//    cout << "Diamer Probabilities Dry: " << diffuseProbability << " , " <<
    desorbProbability << " , " << breakProbability << endl;
    map<string, float> diamer_probabilities = {{"Desorb", desorbProbability}, {"Diffuse",
        diffuseProbability}, {"Break", breakProbability}};
    return generate_event(diamer_probabilities);
}

//void checkOutOfBounds(int x, int y, int index, vector<Particle *> &particles, vector<Chain
    *> &polymers){
//    if(x > dim || x < 0 || y > dim || y < 0){
//        if(particles[index]->getBonded() == true){
//            int polymerIndex
```

```
//         }
//     }
//}

template <typename TwoD>
void particleLoopDry(std::ofstream& outputFile, int t,  int i,TwoD& grid, vector<Particle*>
    &particles,vector<Chain*> &polymers, int& totalEventCount , int& bondEventCount, int&
    desorbEventCount, int& moveEventCount){

    double diffuseToNothingRatioMonomerDry = 1.0 / 5;
    double diffuseToDesorbRatioMonomerDry = 111.7;
    double diffuseToBondRatioMonomerDry = 1;

    outputFile << t << " , " << i << " , " << particles[i]->getX() << " , " << particles[i
        ]->getY() << " , " << particles[i]->getBonded() << "\n";

    vector<int> neighborArrayX;
    vector<int> neighborArrayY;

    bool bondOption = false;

    checkNeighborsToBond(particles[i]->getX(), particles[i]->getY(), grid, neighborArrayX,
        neighborArrayY);

    if(particles[i]->getBonded() == true || particles[i]->recombined == true){
        return;
    }
    if(neighborArrayX.size() != 0){
        bondOption = true;
    }

    string event = generateMonomerProbabilitiesDry(diffuseToNothingRatioMonomerDry,
        diffuseToDesorbRatioMonomerDry, diffuseToBondRatioMonomerDry);

    totalEventCount += 1;
    //         cout << "Generated Event: " << event << endl;
    // change so it always bonds
    if(bondOption == true && event == "Bond"){
                bond(particles[i], grid, particles, polymers);
                bondEventCount += 1;
            }
```

```
    else if(event == "Diffuse"){
        moveEventCount += 1;
        diffuse(particles[i], grid, particles, polymers);
    }
//    else if(event == "Bond" && bondOption == true){
//                bond(particles[i], grid, particles, polymers);
//                bondEventCount += 1;
//            }
    else if(event == "Desorb"){
//        desorb(grid, particles, desorbEventCount, i);
//        desorbEventCount += 1;
    }
//    cout << " particles.size() : " << particles.size() << endl;
}


template <typename TwoD>
void particleLoopWet(std::ofstream& outputFile, int t,  int i,TwoD& grid, vector<Particle*>
    &particles,vector<Chain*> &polymers, int& totalEventCount , int& bondEventCount, int&
    desorbEventCount, int& moveEventCount){

    double diffuseToNothingRatioMonomerWet = 1.0 / 5;
    double diffuseToDesorbRatioMonomerWet = 65;
    double diffuseToBondRatioMonomerWet = 0;

//    outputFile << t << " , " << i << " , " << particles[i]->getX() << " , " << particles[i
    ]->getY() << " , " << particles[i]->getBonded() << "\n";

    vector<int> neighborArrayX;
    vector<int> neighborArrayY;

    bool bondOption = false;

    checkNeighborsToBond(particles[i]->getX(), particles[i]->getY(), grid, neighborArrayX,
        neighborArrayY);

    if(particles[i]->getBonded() == true || particles[i]->recombined == true){
        return;
    }
    if(neighborArrayX.size() != 0){
        bondOption = true;
```

```
//         cout << " Bond option == true " << endl;
    }

    string event = generateMonomerProbabilitiesWet(diffuseToNothingRatioMonomerWet,
        diffuseToDesorbRatioMonomerWet);

    totalEventCount += 1;
    //         cout << "Generated Event: " << event << endl;
//     if(bondOption == true && rnd() < .5){
//         bond(particles[i], grid, particles, polymers);
//         bondEventCount += 1;
//     }
    if(event == "Diffuse"){

        moveEventCount += 1;
        diffuse(particles[i], grid, particles, polymers);
    }
//             if(event == "Bond" && bondOption == true){
//                 bond(*particles[i], grid, particles, polymers);
//                 bondEventCount += 1;
//             }
    else if(event == "Desorb"){

        desorb(grid, particles, desorbEventCount, i);
        desorbEventCount += 1;
    }
//     cout << " particles.size() : " << particles.size() << endl;
}

template <typename TwoD>
void polymerLoopDry(std::ofstream& outputFile, int t,  int i, TwoD& grid, vector<Particle*>
    &particles,vector<Chain*> &polymers, int& totalEventCount , int& bondEventCount, int&
    desorbEventCount, int& moveEventCount, vector<Chain*> &desorbedPolymers, int&
    desorbedPolymerCount, int& breakEventCount){
    vector<int> neighborArrayX;
    vector<int> neighborArrayY;

    double desorbToNothingRatioPolymerDry = 1.0 / 200;
    double desorbToBondRatioPolymerDry = 1;

    double diffuseToNothingDiamerDry = 1.0/328.06 ;
```

```
    double diffuseToDesorbDiamerDry = 1181.09;
    double diffuseToBondDiamerDry = 0.02886;

    bool bondOption = false;
    string event;


    polymerCheckBond(i, grid, polymers[i], particles, polymers, neighborArrayX,
        neighborArrayY);

    if(neighborArrayX.size() != 0){
        bondOption = true;
    }

    if(polymers[i]->getMembers().size() > 2){
        event = generatePolymerProbabilitiesDry(desorbToNothingRatioPolymerDry,
            desorbToBondRatioPolymerDry);
    }
    else{
        event = generateDiamerProbabilitiesDry(diffuseToNothingDiamerDry,
            diffuseToDesorbDiamerDry, diffuseToBondDiamerDry);
    }

    totalEventCount += 1;
    //         cout << "Generated Event: " << event << endl;
    //changed so it always bonds
    if(bondOption == true && event == "Bond"  ){
        bondEventCount += 1;
        polymerBond(i, grid, polymers[i], particles, polymers);
    }
    else if(event == "Diffuse"){
        moveEventCount += 1;

        polymerMove(i, grid, particles, polymers);
    }
//             if(event == "Bond" && bondOption == true){
//                 bond(*particles[i], grid, particles, polymers);
//                 bondEventCount += 1;
//             }
    else if(event == "Desorb"){
```

```
        desorbPolymer(grid, particles, desorbedPolymerCount, i, polymers, desorbedPolymers);
//          desorbEventCount += 1;

    }
    else if(event == "Break"){

        breakPolymer(i, grid, particles, polymers);
        breakEventCount += 1;
    }
    neighborArrayX.clear();
    neighborArrayY.clear();
}


template <typename TwoD>
void polymerLoopWet(std::ofstream& outputFile, int t,  int i, TwoD& grid, vector<Particle*>
    &particles,vector<Chain*> &polymers, int& totalEventCount , int& bondEventCount, int&
    desorbEventCount, int& moveEventCount, vector<Chain*> &desorbedPolymers, int&
    desorbedPolymerCount, int& breakEventCount){
    vector<int> neighborArrayX;
    vector<int> neighborArrayY;

    double desorbToNothingRatioPolymerWet = 1.0 / 345829.9;
    double desorbToBreakPolymerWet = 0;

    double diffuseToNothingRatioDiamerWet = 1/40; // changethis later
    double diffuseToDesorbRatioDiamerWet = 261.2;
    double diffuseToBreakRatioDiamerWet = 0;

    bool bondOption = false;
    string event;

    polymerCheckBond(i, grid, polymers[i], particles, polymers, neighborArrayX,
        neighborArrayY);

    if(neighborArrayX.size() != 0){
        bondOption = true;
    }

    if(polymers[i]->getMembers().size() > 2){
        event = generatePolymerProbabilitiesWet(desorbToNothingRatioPolymerWet,
```

```
            desorbToBreakPolymerWet, polymers[i]->getMembers().size());
    }
    else{
        event = generateDiamerProbabilitiesWet(diffuseToNothingRatioDiamerWet,
            diffuseToDesorbRatioDiamerWet, diffuseToBreakRatioDiamerWet);
    }

    totalEventCount += 1;
    //          cout << "Generated Event: " << event << endl;
    if(bondOption == true && event == "Bond"){
        bondEventCount += 1;

        polymerBond(i, grid, polymers[i], particles, polymers);
    }
    else if(event == "Diffuse"){

        moveEventCount += 1;
        polymerMove(i, grid, particles, polymers);
    }
    else if(event == "Desorb"){

        desorbPolymer(grid, particles, desorbedPolymerCount, i, polymers, desorbedPolymers);
        desorbEventCount += 1;

    }
    else if(event == "Break"){

        breakPolymer(i, grid, particles, polymers);
        breakEventCount += 1;
    }
}


int main() {
    int totalTime = 1000;
//    int totalCycles = 100;

    //running for sheet 14

    //ea 0.24
```

```
vector<int> cycleVector{1};

const std::string csvFileName = "run1_particle_positions.csv";
std::ofstream outputFile(csvFileName);
outputFile << "Timestep,ParticleID,X,Y,Bonded\n";



const std::string csvFileName2 = "run1_particle_positions.csv";
std::ofstream outputFile2(csvFileName2);
outputFile2 << "Timestep,ParticleID,X,Y,Bonded\n";


const std::string csvFileName3= "run1_inalPolymers.csv";
std::ofstream outputFile3(csvFileName3);
outputFile3 <<"PolymerNum,Length\n";

const std::string csvFileName4= "run1_desorbedPolymers.csv";
std::ofstream outputFile4(csvFileName4);
outputFile4 <<"PolymerNum,Length\n";

const std::string csvFileName5= "manyCylcles.csv";
std::ofstream outputFile5(csvFileName5);
outputFile5 << "Total Cycles,Total Particles,Total Polymers,Total Desorbed Polymers,Max
    Polymer, Max Desorbed Polymer\n";


const std::string csvFileName6= "cyclesVsTime.csv";
std::ofstream outputFile6(csvFileName6);
outputFile6 << "Cycles,Total Particles,Total Polymers,Total Desorbed Polymers, Max
    Polymer, Max Desorbed Polymer,Dimers on Surface\n";


for(int k =0; k < cycleVector.size(); k++){

int totalCycles = cycleVector[k];

vector<Particle*> particles;
vector<Chain*> polymers;
vector<Chain*> desorbedPolymers;
```

```
    double adsorptionProbability = 0.2;

    int desorbCount = 0;

    int bondEventCount = 0;
    int moveEventCount = 0;
    int totalEventCount = 0;
    int desorbEventCount = 0;
    int breakEventCount = 0;
    int adsorbEventCount = 0;
    int desorbedPolymerCount = 0;

    int grid[1000][1000];
    for(int i =0; i < dim; i++){
        for(int j = 0; j < dim; j++){
            grid[i][j] = 0;
        }
    }
        int totalParticles = 100;
    populate(totalParticles,particles,grid);


        for(int c =0 ; c < totalCycles; c++){

            cout << " cycle: " << c  << "particles size: " << particles.size() << endl;
            for(int t =0; t < totalTime; t++){
                if(rnd() < adsorptionProbability){
                    populate(1,particles,grid);
                    adsorbEventCount += 1;
                }

                for(int i = 0 ; i < particles.size(); i++){
//                      int particlesToAdd = totalParticles - particles.size();
//                      populate(particlesToAdd, particles, grid);
                    particleLoopWet(outputFile2, t, i, grid, particles, polymers,
                        totalEventCount, bondEventCount, desorbEventCount, moveEventCount);
                }

                for(int i =0; i < polymers.size(); i++){
                    polymerLoopWet(outputFile2, t, i, grid, particles, polymers,
```

```
            totalEventCount, bondEventCount, desorbEventCount, moveEventCount,
            desorbedPolymers, desorbedPolymerCount, breakEventCount);
    }
}

for(int t = 0; t < totalTime; t++){

    for(int i = 0 ; i < particles.size(); i++){

        particleLoopDry(outputFile, t, i, grid, particles, polymers,
            totalEventCount, bondEventCount, desorbEventCount, moveEventCount);
    }

    for(int i =0; i < polymers.size(); i++){
        polymerLoopDry(outputFile, t, i, grid, particles, polymers,
            totalEventCount, bondEventCount, desorbEventCount, moveEventCount,
            desorbedPolymers, desorbedPolymerCount, breakEventCount);
    }
}
int maxPolymer = 0;
for(int i =0; i < polymers.size(); i++){
    int polymerLength = polymers[i]->getMembers().size();
    if(polymerLength > maxPolymer){
        maxPolymer = polymerLength;
    }
    outputFile3 << i << " , " << polymers[i]->getMembers().size() << endl;
}

int maxDesorbedPolymer = 0;
for(int i =0; i < desorbedPolymers.size(); i++){
    int polymerLength = desorbedPolymers[i]->getMembers().size();
    if(polymerLength > maxDesorbedPolymer){
        maxDesorbedPolymer = polymerLength;
    }
    outputFile4 << i << " , " << polymerLength << endl;



}

int dimerCount = 0;
for(int i =0; i < polymers.size(); i++){
```

```
                if(polymers[i]->getMembers().size() == 2){
                    dimerCount += 1;
                }
            }

            outputFile6 << c << " , " << particles.size() << " , " << polymers.size() << " ,
                 " << desorbedPolymers.size() << " , " << maxPolymer << " , " <<
                maxDesorbedPolymer << ", " << dimerCount<< endl;


//             cout << c << " , " << particles.size() << " , " << polymers.size() << " , " <<
     desorbedPolymers.size() << " , " << maxPolymer << " , " << maxDesorbedPolymer << endl;
        }
//         cout << "Total " << totalEventCount << " bond " << bondEventCount << " desorb " <<
     desorbEventCount << " diffuse " << moveEventCount << " breakCount : " <<
    breakEventCount << " adsorb events: " << adsorbEventCount << endl;
//
        int FinalMaxPolymer = 0;
        for(int i =0; i < polymers.size(); i++){
            int polymerLength = polymers[i]->getMembers().size();
            if(polymerLength > FinalMaxPolymer){
                FinalMaxPolymer = polymerLength;
            }
            outputFile3 << i << " , " << polymers[i]->getMembers().size() << endl;
        }

        int fmaxDesorbedPolymer = 0;
        for(int i =0; i < desorbedPolymers.size(); i++){
            int polymerLength = desorbedPolymers[i]->getMembers().size();
            if(polymerLength > fmaxDesorbedPolymer){
                fmaxDesorbedPolymer = polymerLength;
            }
            outputFile4 << i << " , " << polymerLength << endl;



        }

        cout << "Total Particles: " << particles.size() << endl;
        cout << "Total Polymers: " << polymers.size() << endl;
        cout << "Total Desorbed Polymers: " << desorbedPolymers.size() << endl;
        cout << "Maximum Polymer length " << FinalMaxPolymer << endl;
```

```
            cout << "Maximum Desorbed Polymer Length: " << fmaxDesorbedPolymer << endl;
            outputFile5 << cycleVector[k] << " , " << particles.size() << " , " << polymers.size
                () << " , " << desorbedPolymers.size() << " , " << FinalMaxPolymer << " , " <<
                fmaxDesorbedPolymer << endl;
            cout << "desorb events " << desorbEventCount << endl;
        }

        cout << " DONE " << endl;
        return 0;
}
```