Polymer Physics

Kiar Fatah

March 9, 2020

Contents

1 Introduction

A widely used application of polymers is rubbers. To improve the quality of rubbers it is of essence to understand the structure of a polymer. A polymer is built by repeating units of monomers connected together through chemical bonds. Monomers can be connected in various fashion. However, in this approach only linear monomers are of interest [1]. A physical property that polymers in a diluted solution, a chemical solution, possess is that the root mean square end-to-end distance results in

$$R_F \propto N^{\nu}. \tag{1.1}$$

Furthermore the root mean squared distance [2] of an actual polymer with equation 1.1 is given $\nu = 0.592$. A simple random walk model, in three dimensions on a cubical lattice that is allowed to cross itself, which is not a physical property, approximately has the value, $\nu = 1/2$. The error for ν is

$$\nu_{correct} - \nu_{model} = 0.08. \tag{1.2}$$

Therefore by developing the model the value ν_{model} can approach $\nu = 0.592$. Hence random walk models proves to be an acceptable simulation to approximate polymers.

Note that simulation of random walk can occur in multiple ways. To create a realistic model of random walk with the intention to mimic the behavior of polymers, it is important to include the excluded volume interaction. It other words monomers shall not be able to cross themselves, since this is not a real physical property. However, to apply and run this kind of behavior is complex and takes more computational power. Therefore this behavior can be simplified by not applying it. Models that contain the excluded volume interaction are called self avoiding walks.

Hence an efficient model is simple random walk. Simple random walk occurs on a cubical three dimensional lattice and each step has the same length. For every step each of the six directions is generated randomly and therefore has the equal probability to ensue. Applying self avoiding walk on a simple random walk model causes each step to not return to an already visited location.

However, the most non complex model for simulation of polymers is the freely jointed chain model in a continuous space. The length between each monomer is equivalent for each step. Every step has a free bond angle, meaning the angle between each monomer is randomly chosen. For the self avoiding walk of freely jointed chain, also known as real chains, each monomer is contained in a sphere, with a chosen radius, which is not allowed to be in bound with any other spheres of monomers.

[1] These are chains that only can connect from either the top or the bottom of the chain as a snake.
[2] root mean squared end-to-end distance

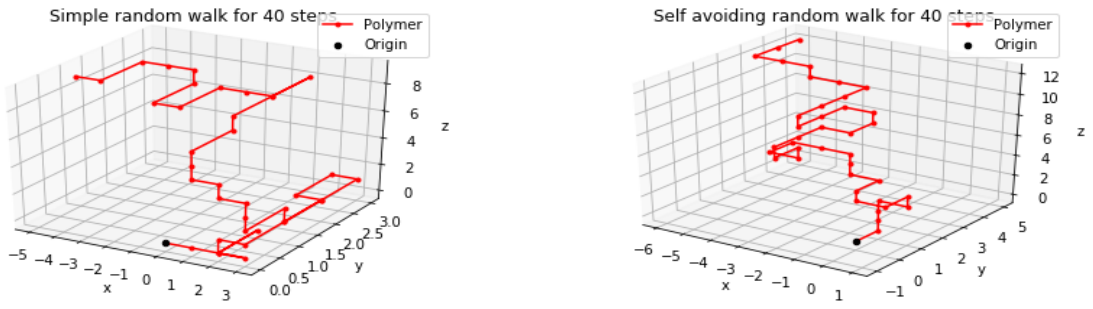
1.1 Problem

Two models of random walk will be simulated, analyzed and compared. The first one being random walk on a three dimensional grid and the second one is freely jointed chain in continuous space. Furthermore the implementation of these models allows for investigation of self avoiding walk. Meaning a comparison between the non excluded volume interaction and the excluded volume interaction will be made. The comparison will use statistical tools to showcase the contrast between the different models.

2 Simulation

2.1 Simple random walk

For the simple random walk model in discrete space a fixed length is chosen, for this project every random walk model will have the length 1. A random number between the interval 0 to 5 is generated, each representing a direction in the three dimensional grid cubical lattice, thus deciding the direction of the next walk. The result can be seen in figure 1a.



(a) Simulation of simple random walk model for 40 steps.

(b) Simulation of self avoiding random walk model for 40 steps.

Figure 1

2.2 Self avoiding random walk

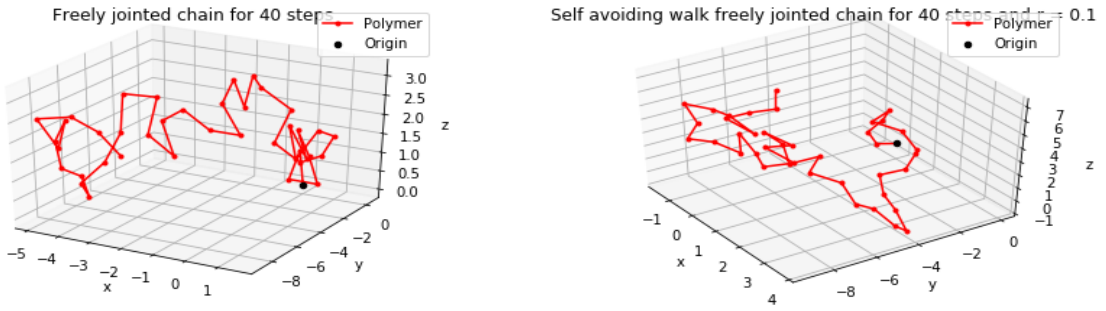
The direction of the next step is generated equally to the simple random walk model. However, each location visited is now stored in a list. Therefore each step that is generated is verified before the actual step is taken. Meaning if the next step is in a location where the random walk has already visited, the step is terminated. An implementation of this can be seen in figure 1b.

2.3 Freely jointed random walk

For the freely jointed random walk model in continuous space, a Gaussian distribution is applied under the interval N(0,1) for each direction in the Cartesian coordinate. The result is then normalized, resulting in the length between each monomer is 1. The simulation of freely jointed random walk can be seen in figure 2a.

2.4 Freely jointed self avoiding random walk

For real chains the excluded volume interaction is implemented for the freely jointed chain model. Each monomer is contained in a sphere with an chosen radius, r, as mentioned earlier. For each newly generated monomer, the radius for the sphere of that monomer is compared to every other spheres. If the radius is within a boundary less then 2r then the walk resets. This can be observed in figure 2b.



(a) Simulation of freely jointed chain model for 40 steps.

(b) Simulation of self avoiding freely jointed chain for 40 steps.

Figure 2

3 Measurement

Application of approximation of real polymers in a diluted solution and statistical tools is applied to highlight the difference between each models.

3.1 Root mean square

Due to the model of the random walk it is possible to obtain negative values. Hence to calculate the dimension and distance of the random walk models root mean square distance applied. Let x_1, x_2 ... be a measurement, the root mean square is calculated as

$$x_{rms} = \sqrt{\frac{1}{n}(x_1^2 + x_2^2....)}. \tag{3.1}$$

For a linear polymer the mean square distance of the end-to-end, R_F^2, is defined as

$$R_F^2 = \langle R^2 \rangle \equiv \sum_{i=1}^{N} \frac{1}{N}(\boldsymbol{r}_i - \boldsymbol{r}_0)^2. \tag{3.2}$$

It is the difference between the beginning and the end of a polymer chain. In this case the vector $\boldsymbol{r}_0$ is equal to the zero vector. Since each random walk model begins the walk at the origin. Thus the equation for the root mean square distance develops to

$$\sqrt{\langle R^2 \rangle} = \sqrt{\sum_{i=1}^{N} \frac{1}{N} \boldsymbol{R}_i^2}. \tag{3.3}$$

A second method of measurement for the dimension and distance of random walk is the root mean square radius of gyration, R_g. The radius of gyration is the root mean square distance of the monomers to the center of mass of the structure. Note that each monomer is assumed to have the same mass. Therefore the center of mass for a random walk model is given by

$$\boldsymbol{r}_g = \frac{1}{N+1} \sum_{i=0}^{N} \boldsymbol{r}_i \tag{3.4}$$

This results in the mean square radius of gyration

$$R_g^2 = \frac{1}{N+1} \sum_{i=0}^{N} \langle (\boldsymbol{r}_i - \boldsymbol{r}_g)^2 \rangle. \tag{3.5}$$

3.2 Application of variance

The measurements of displacement of random numbers from their average is given by the variance. The variance is calculated with the following formula

$$\sigma = \langle R^2 \rangle - \langle R \rangle^2. \tag{3.6}$$

For the case of error, the standard error estimate is applied. The standard error estimate is a measurement of how much the value deviates if it where to run an infinite amount of times. The formula is given by

$$SEE = \sqrt{\frac{\sigma}{M-1}}, \tag{3.7}$$

where M is the amount of samples. For this project M will be the amount of reruns of the same walk. And the standard error estimate of the mean square of the end-to-end distance can be calculated as

$$SEE_{R_F} = \sqrt{\frac{\langle R_F^2 \rangle - \langle R_F \rangle^2}{M-1}} = \sqrt{\frac{\sigma_{R_F}}{M-1}}. \tag{3.8}$$

The estimate of the root mean square fluctuation is given by

$$ERMSF_{R_F} = \sqrt{\frac{(\langle R_F^2 \rangle - \langle R_F \rangle^2)M}{M-1}} = \sqrt{\frac{\sigma_{R_F M}}{M-1}}. \tag{3.9}$$

3.3 Fraction of successful walk

Each step in self avoiding random walk models are dependent on the previous steps. Therefore a count variable is created to store the values for each time the models generate an already visited coordinate. If the model revisits a coordinate then the walk resets. Note that if this is done for M amount of reruns of the same step number. The result will be the average value needed for the self avoiding random walk to complete a simulation. The inverse is the fraction of successful walks. Thus a comparison between the real chains and self avoiding walk on a three dimensional grid can be showcased.

3.4 Variety of diluted solvents

As mentioned earlier the root mean square distance is known for a polymers in a diluted solution. However, it is dependent on the type of solution. For a good solvent the equation 1.1 has the value $\nu \approx 3/5$, bad solvent $\nu = 1/3$, and for the θ solvent $\nu = 1/2$. For this project the equation 1.1 becomes a tool to approximately examine if the implemented code is correct.

The solvents of interest are the good and the θ solvent. This can be used to give an insight of the root mean square distance for the self avoiding walk respectively the non self avoiding random walk. The θ solvent properties reject the excluded volume interaction and hence it becomes a model for the simple random walk. The good solvent properties is the opposite of the θ solvent and is therefore an approximation for the self avoiding walk.

For real chains the Flory exponent highlights the property between mean square radius of gyration and distance as

$$\frac{6R_g^2}{R_F^2} \approx 0.952. \tag{3.10}$$

3.5 Precision

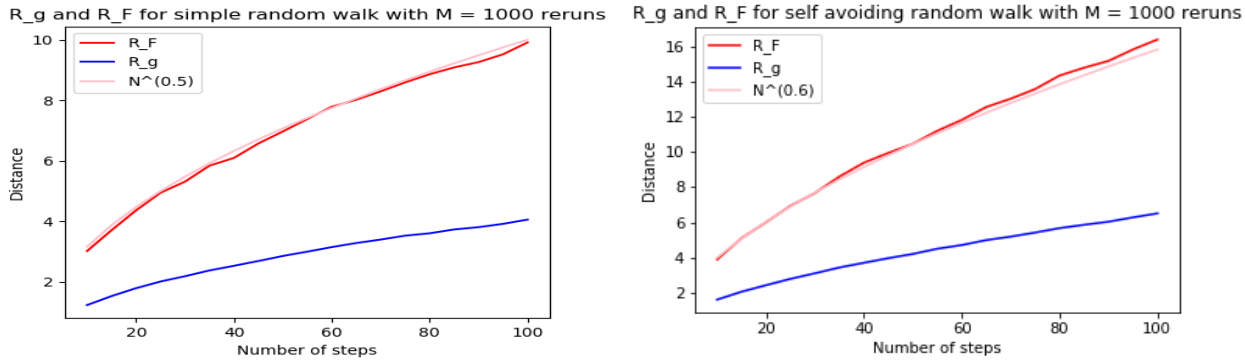
The simulations are based on randomness. Thus each run will result in a different value. To obtain steady values each walk is simulation M times. This allows for the study of precision of the values. By plotting R_F and R_g as a function of the amount of reruns, M, for a given step number, the relation of convergence and accuracy can be observed from the plots by plotting the last value as a straight line.

4 Results and Analysis

4.1 Random walk on a three dimensional grid

4.1.1 Root mean square of the distance and radius of gyration

The root mean square distance and the root mean square radius of gyration is calculated for $M = 1000$ reruns in the interval of $1 < N < 100$ steps. The result, figure 3a and 3b, is plotted for the number of steps made. Due to the simple random walk model being approximated as a θ solvent and self avoiding walk model as a good solvent, their respective approximations for the root mean square distance $N^{0.5}$ and $N^{3/5}$ are also plotted.



(a) R_g and R_F for simple random walk plotted against the number of steps.

(b) R_g and R_F for self avoiding random walk plotted against the number of steps.

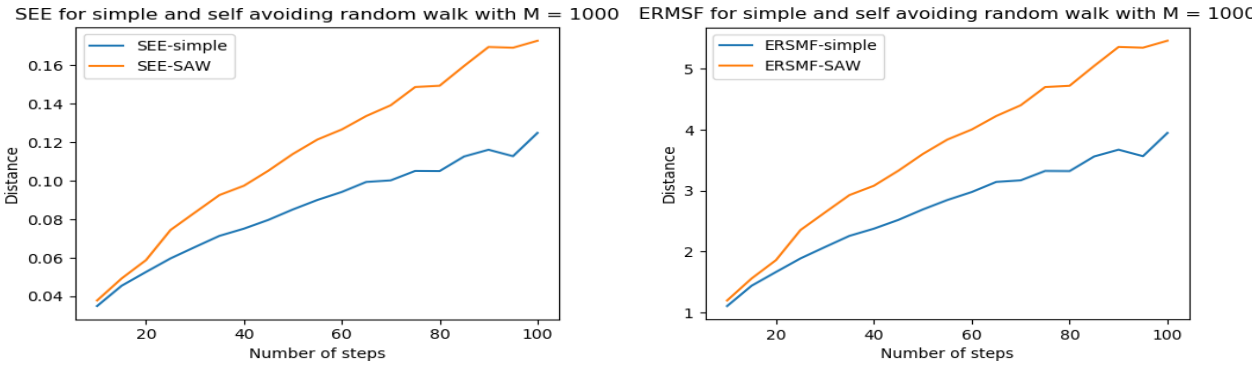
Figure 3

4.1.2 Standard error estimate and estimate of root mean square fluctuation

The standard error estimate and the estimate of the root mean square fluctuations [3] is calculated for the simulation of simple and self avoiding random walk. The result is shown

[3] Estimate of root mean square fluctuation is reduced to ERMSF in the plots.

in the figures 4a and 4b.



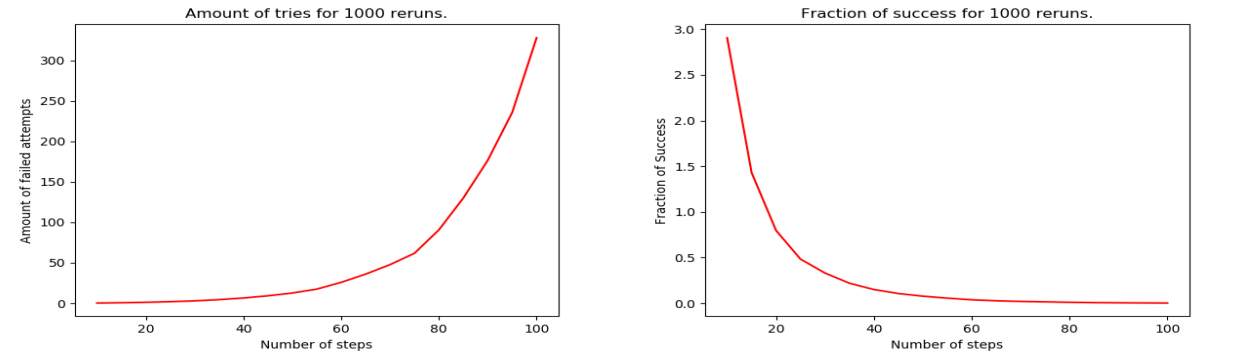
(a) Standard error estimate for simple and self avoiding random walk plotted against the number of steps.

(b) Estimate of root mean square fluctuation for simple and self avoiding random walk plotted against the number of steps.

Figure 4

4.1.3 Fraction of successful walks

The fraction of success can be calculated for the self avoiding walk. For $M = 1000$ reruns of each step the average of failed attempts for each rerun is calculated. Note the fraction of success is given by the inverse of the amount of failed attempts. The result is figure 5a and 5b and that shows the average amount of failed attempts and average fraction of success plotted against the number of walks.



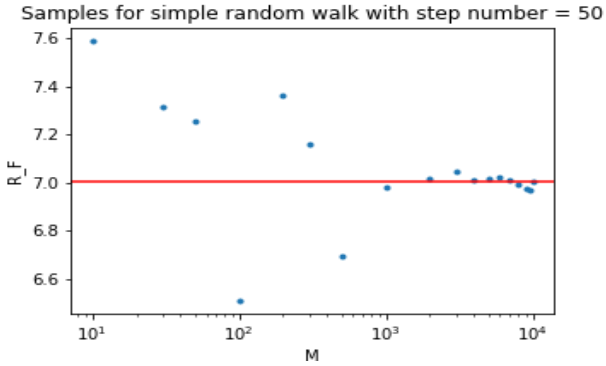
(a) Average amount of failed attempts for self avoiding random walk and number of steps.

(b) Average fraction of successful for self avoiding random walk and number of steps.

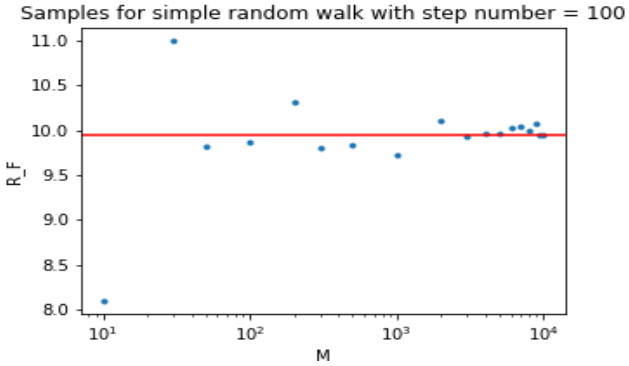
Figure 5

4.1.4 Precision

For the simple random walk model, R_F and R_g is function of M, whereas M is an array with the interval $10 < M < 10000$. The result is then plotted for $N = 50$ and $N = 100$, which can be seen in figures 6 and 7. The same parameters are used for the self avoiding walk which can been seen in figures 8 and 9.

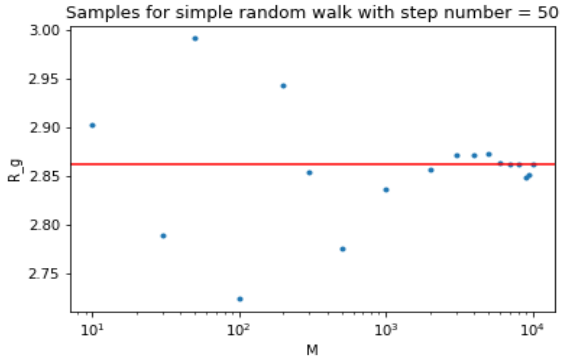


(a) R_F(M) and amount of reruns, M, for step number = 50.

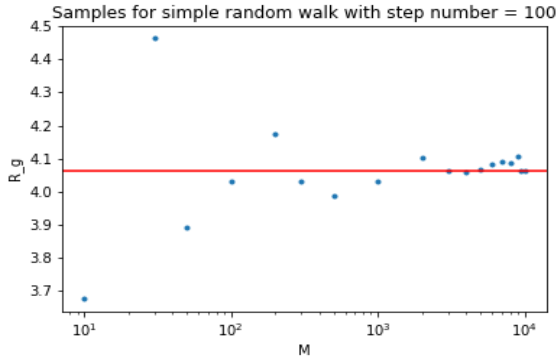


(b) R_F(M) and amount of reruns, M, for step number = 100.

Figure 6

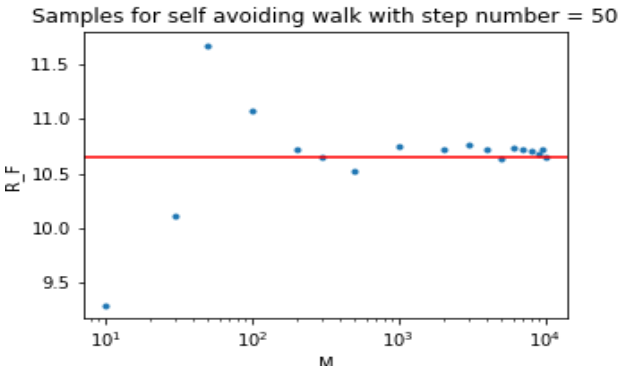


(a) R_g(M) and amount of reruns, M, for step number = 50.

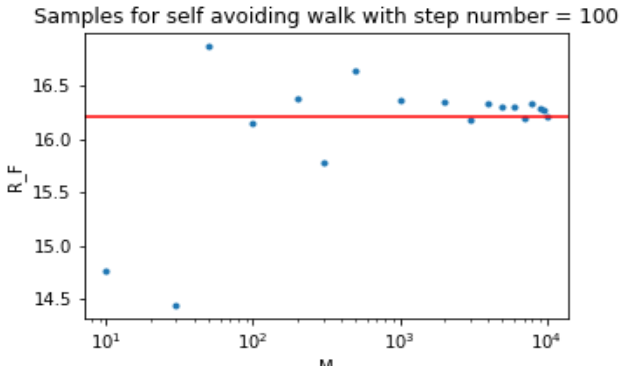


(b) R_g(M) and amount of reruns, M, for step number = 100.

Figure 7

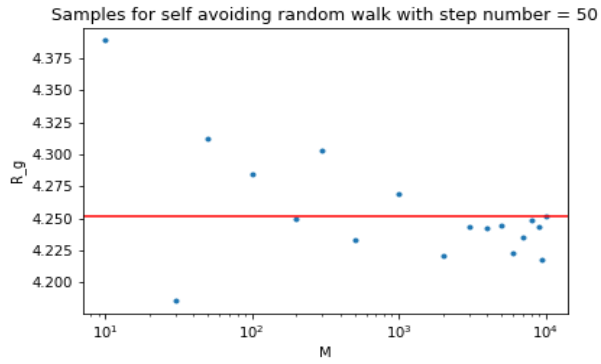


(a) R_F(M) and amount of reruns, M, for step number = 50.

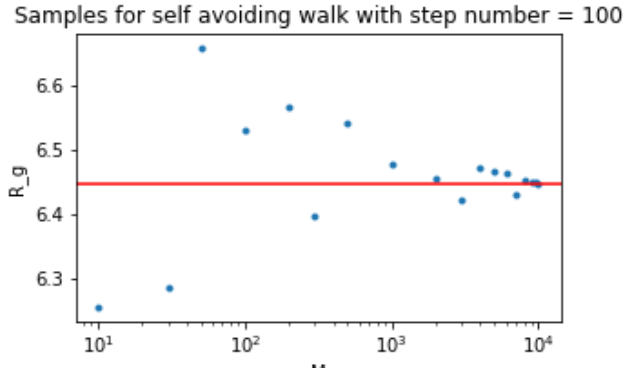


(b) R_F(M) and amount of reruns, M, for step number = 100.

Figure 8



(a) R_g(M) for self avoiding walk and the amount of reruns, M, for step number = 50.



(b) R_g(M) and amount of reruns, M, for step number = 100.

Figure 9

4.2 Freely jointed chain

4.2.1 Root mean square of distance and radius of gyration

In consideration of comparison, the equal value of reruns and steps are used to simulate the two cases of the non and excluded volume interaction of freely jointed chain. Due to the ideal freely jointed chain being approximated as a θ solvent the approximations $N^{0.5}$ is plotted along side for correlation.

For the excluded volume interaction of the freely jointed chain, the radius of the spheres was chosen as 0.1, 0.09 and 0.08. This was due to performance, which will be discussed more in the section 5.3, sources of error.

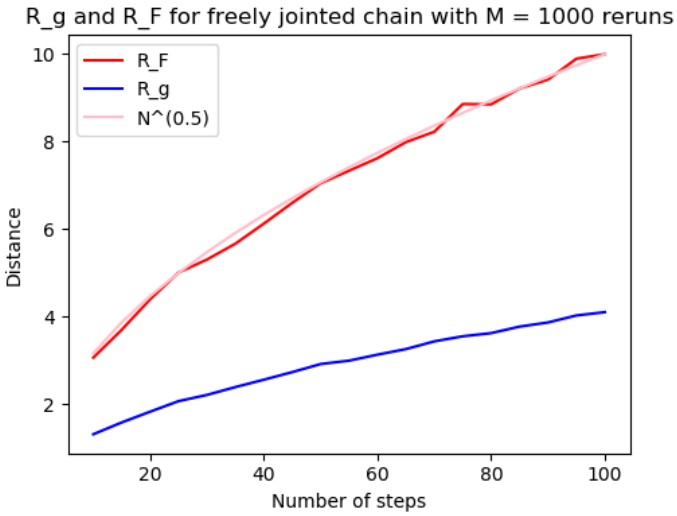
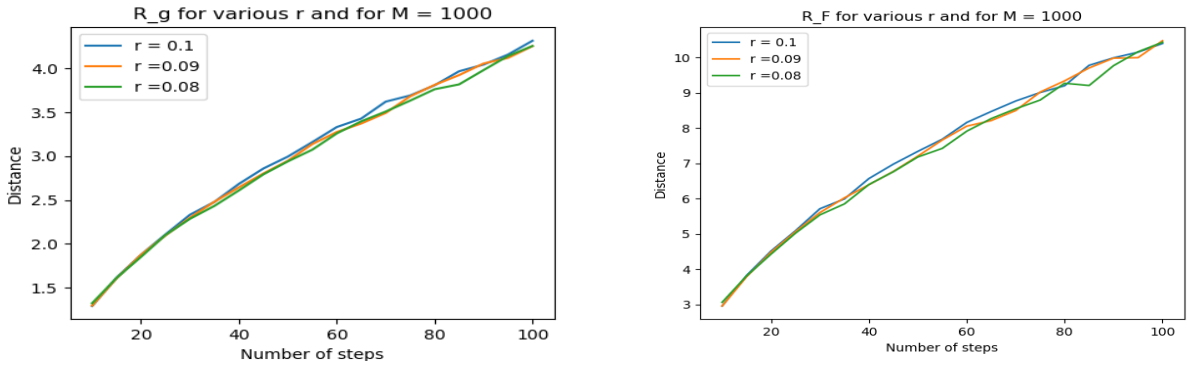


Figure 10: The root of R_g^2 and R_F^2 plotted against the number of steps. The approximation of freely jointed chain model is also plotted.



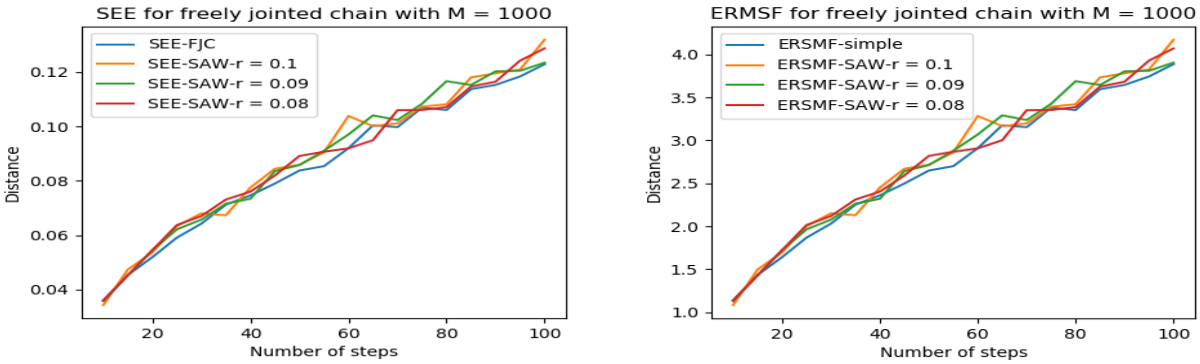
(a) R_g plotted against the number of steps for various different radius.

(b) Average fraction of success for self avoiding random walk and number of steps.

Figure 11

4.2.2 Standard error estimate and estimate of root mean square fluctuation

The standard error estimate and the estimate of the root mean square fluctuations is calculated for the radii 0.1, 0.9 and 0.8 for the freely jointed chain self avoiding walk with $M = 1000$ reruns. The result is shown in the figure 12.



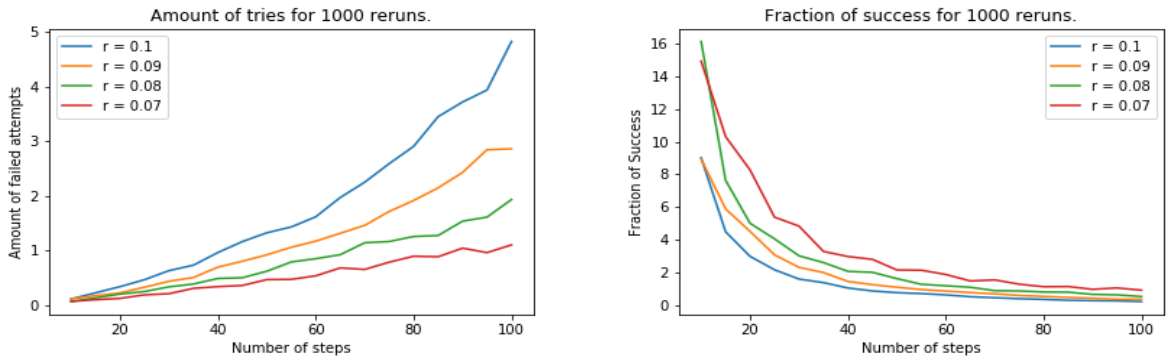
(a) Average amount of failed attempts for self avoiding freely jointed chain and number of steps.

(b) Average fraction of success for self avoiding freely jointed chain and number of steps.

Figure 12

4.2.3 Fraction of successful Walks

For the radii 0.1, 0.09 and 0.08 the average amount of failed attempts and the fraction of successful walks was calculated.



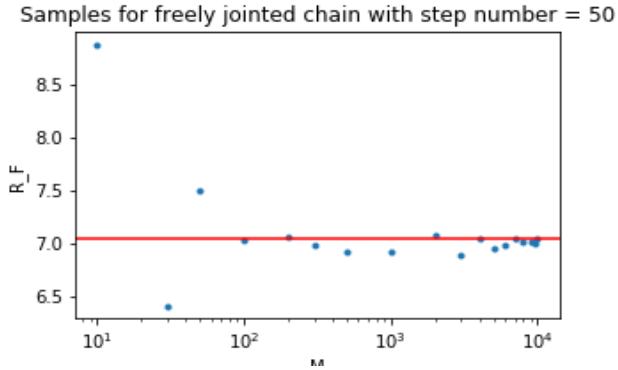
(a) Average amount of failed attempts for self avoiding freely jointed chain and number of steps.

(b) Average fraction of success for self avoiding freely jointed chain and number of steps.

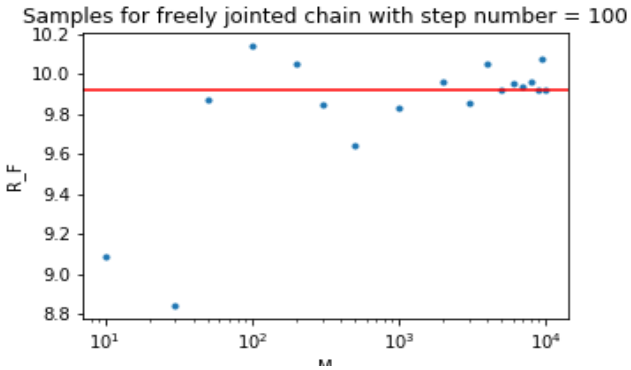
Figure 13

4.2.4 Precision

As in the section 4.1.4 of the random on a three dimensional grid, the precision for the freely jointed chain and the real chains is calculated. The radius for the real chain model is 0.07.

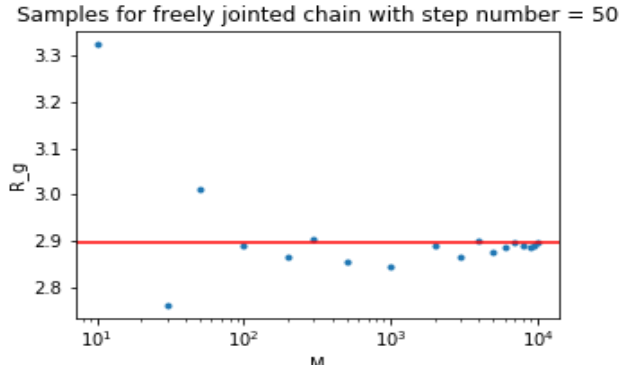


(a) R_F(M) for freely jointed chain and the amount of reruns, M, for step number = 50.

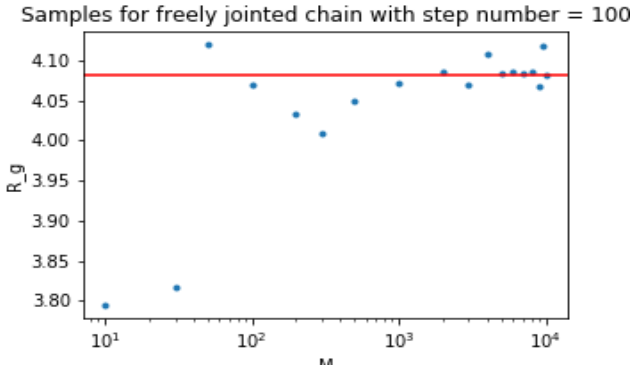


(b) R_F(M) for freely jointed chain and the amount of reruns, M, for step number = 100.

Figure 14

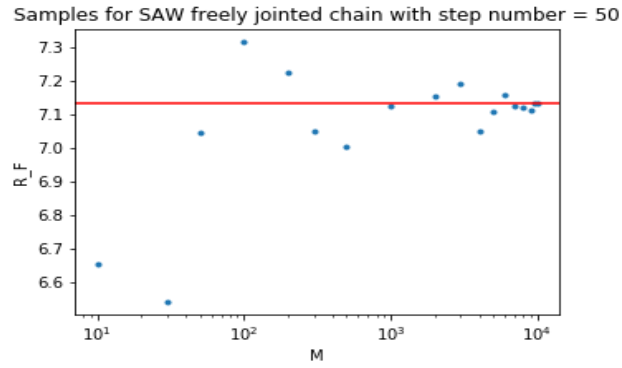


(a) R_g(M) for freely jointed chain and the amount of reruns, M, for step number = 50.

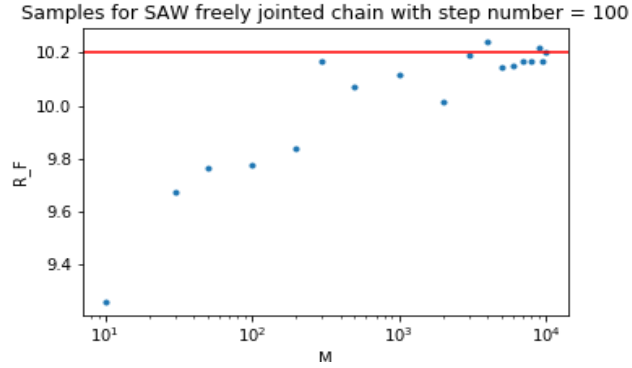


(b) R_g(M) for freely jointed chain and the amount of reruns, M, for step number = 100.

Figure 15

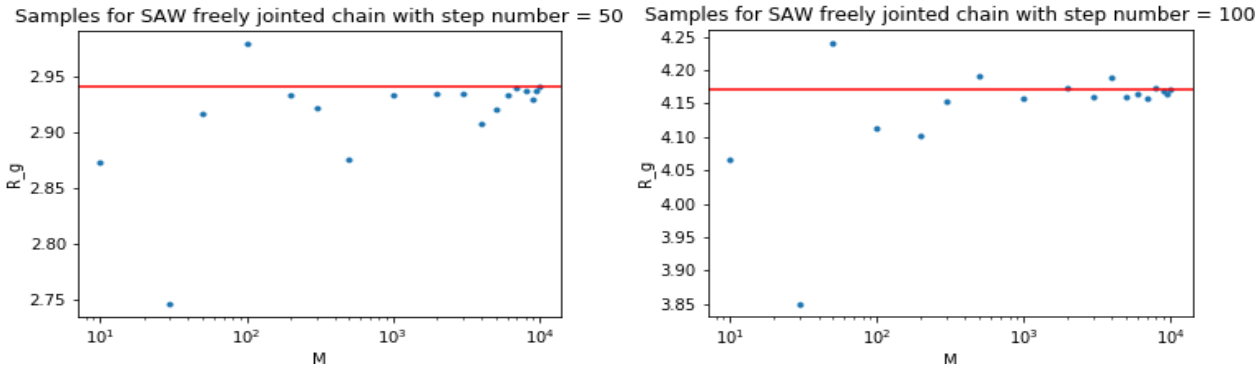


(a) R_F(M) for real chains and the amount of reruns, M, for step number = 100.



(b) R_F(M) for real chains and the amount of reruns, M, for step number = 100.

Figure 16



(a) R_g(M) for real chains and the amount of reruns, M, for step number = 50.

(b) R_g(M) for real chains and the amount of reruns, M, for step number = 100.

Figure 17

4.2.5 Flory exponent

The Flory exponent for the real chain is drawn in respect to the same parameters.

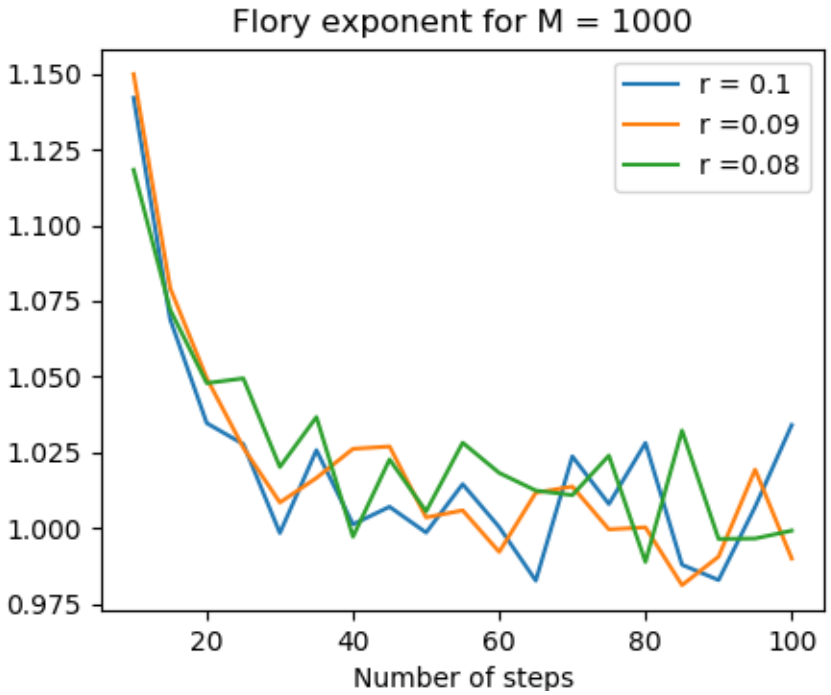


Figure 18: Flory exponent for real chains, M = 1000.

5 Discussion and conclusion

5.1 Comparison

5.1.1 Simple random walk and freely jointed chain

The two models are both non self avoiding walks. Meaning they are not allowed to cross themselves and therefore they are both approximated with a θ solvent as

$$R_F \propto N^{1/2}. \tag{5.1}$$

In consideration to the resulting plots, figure 3a and 10, it can be observed that both models of the simulations follow the approximation of the θ solvent for the number of steps and reruns.

The root mean square value for distance and radius of gyration for both models are approximately equal, which can be seen in figure 3a and 10. The difference for the standard error estimate is small, if not equal due to randomness, since the value of M was 1000 reruns but in reference to the section of precision for both models the root mean square distance converges at higher value of M for higher number of steps. Meaning a higher value on M would result in more detailed plots, the reason behind why this value of M was chosen is discussed in 5.3. In addition the results of precision can be interpreted as for a higher value of the step number the less does the value converge depending on the value on the reruns, M.

5.2 Self avoiding walk

For the excluded volume interactions models, the random walk on a three dimensional grid obtains a greater value, figure 3b and 11b, and standard error estimate, figure 4a and 17a, in comparison to multiple radii for the freely jointed chain model. However, for higher values on the radii the real chain will behave more as the self avoiding random walk.

For the section of precision for the self avoiding walks, 4.1.4 and 4.2.4, the plots behaves as simple random walk and freely jointed chain. For a greater number of steps a greater number of M is needed to obtain a precise value.

The self avoiding walk on a grid is approximate as a good solvent, which the plot follows. For real chains the Flory exponent is approximately in the boundary of value 1 which confirms the approximation.

The fraction of successful walks for the real chains is dependent on radius of the sphere. For a higher value on the radius the fraction of successful walks decreases faster. Therefore there should exist a radius that has a lower fraction of successful walks than the self avoiding

random walk on a grid. However, for the runs made, the fraction of the successful walks the real chains obtained a lower value faster.

5.3 Sources of error

This project was simulated on a 2015 MacBook Pro from Apple with the following specifications:

1. CPU: 2.7 GHz Intel Core i5.
2. GPU: Intel Iris Graphics 6100 1536 MB.
3. Memory: 8 GB.

The choice of hardware might not be considered industry or modern standard for physical simulations. Therefore each simulation took time to process. This was the bottleneck of the project, some 20 to 40 minutes. There could have been bigger possibilities of running a higher number of reruns, the value M, for the random walks. For the section of precision M was equal to 10000, but for the actual simulations of the root mean square of the distance and radius of gyration M had a value of 1000. There also could have been more analyzing on the convergence of each random walk if M was a higher value. The radius for the real chains could have also set to a higher value. The results might revealed information that was not obtained within this project.

6 Appendix

6.1 Simple random walk

```
def rand_walk(num_of_steps):
    x = [0] * num_of_steps
    y = [0] * num_of_steps
    z = [0] * num_of_steps
    coord = [[0,0,0]]

    for i in range(0,num_of_steps-1):
        #A random number between the interval 0-5 is generated
        rand_num = int(np.random.rand()*6)

        #Appending walk
        if rand_num == 0: # x += 1
            x[i+1] = x[i]+1
            y[i+1] = y[i]
            z[i+1] = z[i]
```

```
            #stores all the coordinates
            coord.append([x[i+1],y[i+1],z[i+1]])

        if rand_num == 1: # x -= 1
            x[i+1] = x[i]-1
            y[i+1] = y[i]
            z[i+1] = z[i]
            coord.append([x[i+1],y[i+1],z[i+1]])

        if rand_num == 2: # y += 1
            x[i+1] = x[i]
            y[i+1] = y[i]+1
            z[i+1] = z[i]
            coord.append([x[i+1],y[i+1],z[i+1]])

        if rand_num == 3: # y -= 1
            x[i+1] = x[i]
            y[i+1] = y[i]-1
            z[i+1] = z[i]
            coord.append([x[i+1],y[i+1],z[i+1]])

        if rand_num == 4: # z += 1
            x[i+1] = x[i]
            y[i+1] = y[i]
            z[i+1] = z[i]+1
            coord.append([x[i+1],y[i+1],z[i+1]])

        if rand_num == 5: # z -= 1
            x[i+1] = x[i]
            y[i+1] = y[i]
            z[i+1] = z[i]-1
            coord.append([x[i+1],y[i+1],z[i+1]])
    #Return a list of x,y,z,coord
    return x,y,z, coord
```

6.2 Self avoiding random walk

```
def summation(vec_len, vec): #Adds the newly generated vec to vec_len
    for i in range(0,3):
        vec_len[i] = vec_len[i] + vec[i]
    return vec_len
#Three dimensional random walk, SAW.
def SAW(n):
    count = 0
    #Each step that is allowed to take
    list_of_steps = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [-1, 0, 0], [0, -1, 0], [0, 0, -1]
    Boolean = False
    while Boolean == False:
        x = [0]
```

```
        y = [0]
        z = [0]
        #store every visited coordinate
        coord = [[0, 0, 0]]
        #Stores a temporary vector
        store_vec = [0, 0, 0]
        i = 0
        #previos direction
        prev_dir = [0, 0, 0]
        while i < n:
            #randomly chooses a direction for the next step
            if prev_dir == [0, 0, 0]:
                next_step = random.choice(list_of_steps)
            else:
                temp = list_of_steps.copy()
                temp.remove(prev_dir)
                next_step = random.choice(temp)
            for w in range(0, 3):
                prev_dir[w] = -next_step[w]
            store_vec = summation(store_vec,next_step)
            if store_vec in coord:
                count=count+1
                break
            coord.append([store_vec[0], store_vec[1], store_vec[2]])
            x.append(store_vec[0])
            y.append(store_vec[1])
            z.append(store_vec[2])
            if i == n-1:
                Boolean = True
            i = i + 1
    return x,y,z, coord, count
```

6.3 Freely jointed chain

```
def rand_walk(n):
    coord = [[0,0,0]]
    x = [0]
    y = [0]
    z = [0]
    for i in range(0,n):
        #Generates a normalized random vector
        vec = rand_vec()

        #Appends the normalized random vector
        x.append(vec[0] + x[i])
```

```
        y.append(vec[1] + y[i])
        z.append(vec[2] + z[i])
        coord.append([x[i+1],y[i+1],z[i+1]])

    return x,y,z,coord


#A function that generates a random vector and normalizes it
def rand_vec():
    #vec[0] = x, vec[1] = y ....
    vec = []
    for i in range(0,3):
        vec.append(np.random.normal(0,1,1))

    norm = np.sqrt(pow(vec[0],2) + pow(vec[1],2) + pow(vec[2],2))
    vec = np.multiply(vec,norm**(-1))
    return vec
```

6.4 Real chains

```
def rand_vec(): #Generates a normalized 3d vector with the help of np.random.normal(0,1,1)
    #vec[0] = x, vec[1] = y ....
    vec = []
    for i in range(0,3):
        vec.append(np.random.normal(0,1,1))
    vec = np.divide(vec,(vec[1]**2+vec[2]**2+vec[0]**2)**0.5)
    return vec
def summation(vec_len, vec): #Adds the newly generated vec to vec_len
    for i in range(0,3):
        vec_len[i] = vec_len[i] + vec[i]
    return vec_len

def rand_walk(num_of_step, r):
    Boolean = False
    fails = 0

    while Boolean == False:
        x = [0]
        y = [0]
        z = [0]
        coord = [[0,0,0]] #Stores [x,y,z]
        vec_len = [0,0,0] #Sum of x,y,z vecs, needed to compare the spheres
        i = 0
        while i < num_of_step:
            counter = 0
            vec = rand_vec() #Rand_vec generates a normalized vector
            vec_len = summation(vec_len, vec) #Adds each coordinate of vec to vec_len

            for co in coord:  #compares sphere radius for monomers
                if (vec_len[0]-co[0])**2+(vec_len[1]-co[1])**2+(vec_len[2]-co[2])**2 < (2*r)**2:
```

```
                counter += 1

        if counter > 0:
            fails += 1
            break #Breaks out of while i < num_of_step: and restarts
        coord.append([vec_len[0], vec_len[1], vec_len[2]])
        x.append(vec_len[0])
        y.append(vec_len[1])
        z.append(vec_len[2])
        if i == num_of_step-1:
            Boolean = True
        i = i + 1
return x, y, z, coord, fails
```