ECS129 Project: Volume of a Protein Structure

Chad Pickering and Cathy Wang

February 26, 2016

Premise: Proteins are typically represented as a union of spheres, each sphere corresponding to an atom. Calculating various statistics of a protein is critical to identifying and understanding the interaction of the protein with its environment, such as where potential active sites are located and how they are used. The stability of the protein can also be analyzed in terms of the hydrophobic and hydrophilic nature of the surfaces, and the ratio of surface area to volume is an indication of how globular the protein is. Here, the goal is to calculate the volume of a protein given each atom's radii and their respective coordinates in three dimensions. The estimate of the volume gives information about the protein's behavior in terms of water displacement and its interaction with other solutions.

An analytical method that will be used here to help explain the protein's physical properties is Monte Carlo integration; we will calculate the volume of the rectangular prism in which the protein resides and multiply that by the proportion of randomly generated points that fall within the protein to estimate the true integral. We will explore the details of the algorithm used, discuss the properties of the volume estimate when the number of points generated varies, and look at how efficiency is lost when the number of points exceeds a certain value. Additionally, we will explore how the algorithm and strategy can be adjusted to calculate surface area, and how importance sampling can be used if sampling the randomly generated points from a non-uniform distribution is desired.

The methods and analyses herein have been thoroughly explored and analyzed by experts in various fields long before us. Here, we attempt to capture the essence of the subject and examine only small, if not surface level, portions of the topic. We encourage the following efforts to elicit constructive responses from readers.

Methods/Algorithm: A dataset is given with m spheres, each defined by a radius r and x, y, and z coordinates. First we define a rectangular prism R with a volume

$$V(R) = X_{len} * Y_{len} * Z_{len}$$

where

$$\begin{aligned} X_{len} &= (x_{max} + r_{x_{max}}) - (x_{min} - r_{x_{min}}) \\ Y_{len} &= (y_{max} + r_{y_{max}}) - (y_{min} - r_{y_{min}}) \\ Z_{len} &= (z_{max} + r_{z_{max}}) - (z_{min} - r_{z_{min}}) \end{aligned}$$

so that R fully encompasses all spheres.

Now define a function which has a value x conditional on whether a uniformly distributed randomly generated point lies inside or on the boundary of one or more of the atoms $A \cup B \cup \cdots \cup \omega$ in the protein or not. ω is the m^{th} atom.

Define function f as:

For all $x \in R$,

$$f(x) = \begin{cases} 1, & \text{if } x \in \phi \\ 0, & \text{otherwise} \end{cases}$$

where

$$\phi = A \cup B \cup \cdots \cup \omega$$

To compute the volume occupied by the m spheres, denoted as $V(\phi)$, use the following algorithm:

Randomly generate N uniformly distributed points in R.

Next, compute $f(x_i)$ for $i = 1, \ldots, N$; apply the function f to each randomly generated point. For each point, we essentially ask: is the distance from the point x_i to the center of atom A less than the radius of A, or is the distance from the point x_i to the center of atom B less than the radius of B, and so on, until ω. If one or more of those conditions are true, $f(x_i) = 1$; else, $f(x_i) = 0$. This process is executed for all $i = 1, \ldots, N$, with the total number of points residing inside the protein ϕ represented as the variable

$$S_1 = \sum_{i=1}^{N} f(x_i).$$

The variable

$$S_2 = \sum_{i=1}^{N} f^2(x_i)$$

is a necessary component of our standard error term, so this should be calculated along with S_1. When generating uniform random points within R, the terms S_1 and S_2 are equivalent because $f(x_i)$ can only be 0 or 1. Therefore, the arithmetic means

$$\langle f \rangle = \frac{S_1}{N}$$

and

$$\langle f^2 \rangle = \frac{S_2}{N}$$

are also equivalent.

We then compute the volume of the protein, $V(\phi)$, which is the proportion of the randomly generated uniformly distributed points within R that falls within ϕ multiplied by the total volume of R, $V(R)$:

$$V(\phi) = V(R) \cdot \langle f \rangle.$$

Lastly, compute the one standard deviation error estimate term

$$V(R) \cdot \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

which varies depending on N. As $N \to \infty$, the error term goes to zero and $\langle f \rangle$ converges to the true proportion of points that fall within ϕ. In this way, $V(\phi)$ converges to its true volume.

All of the components of $V(\phi)$ and its error estimate have been calculated; the entire formula is as follows:

$$V(R) \cdot \langle f \rangle \pm V(R) \cdot \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

Program of Algorithm: The following is the program in R used to execute the algorithm, computing both $V(\phi)$ and the error estimate.

First, read in the protein coordinate and radius data, and initialize some R packages.

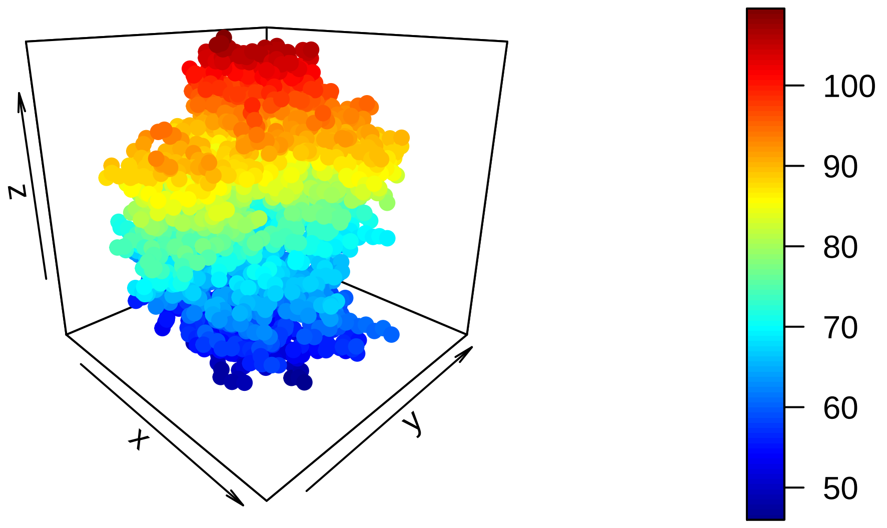
```r
protein <- read.table("C:/Users/cpickering/Syncplicity Folders/ChadSync/STATISTICS/ECS129/Protein.txt",
library(plot3D)
library(flexclust)
```

```
## Loading required package: grid
## Loading required package: lattice
## Loading required package: modeltools
## Loading required package: stats4
```

```r
library(ggplot2)
```

The following is a three-dimensional representation of the input protein, colored by the value of z. The planes in the background give an idea of where the bounds of the rectangular prism R are relative to the protein. Note: The radii are not to scale.

```r
scatter3D(protein$V1, protein$V2, protein$V3, pch = 19, phi = 20, theta = 45)
```

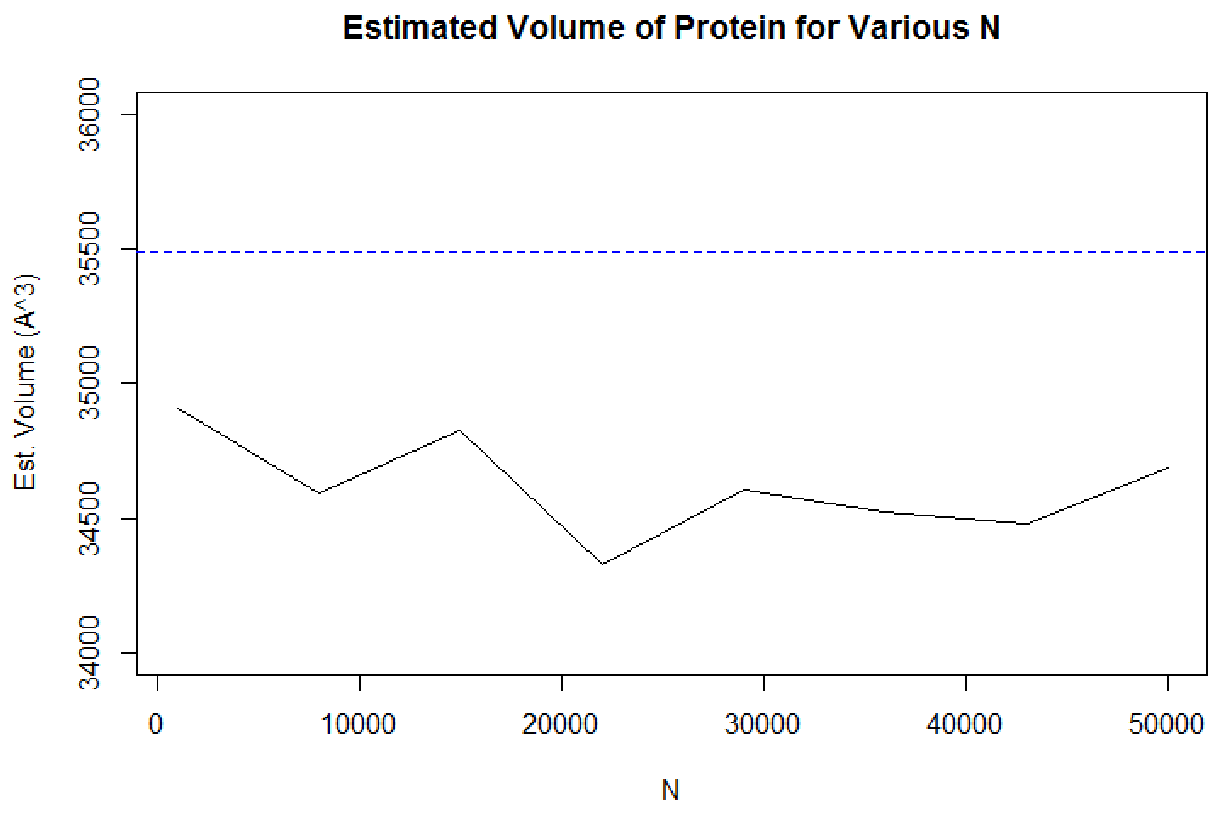


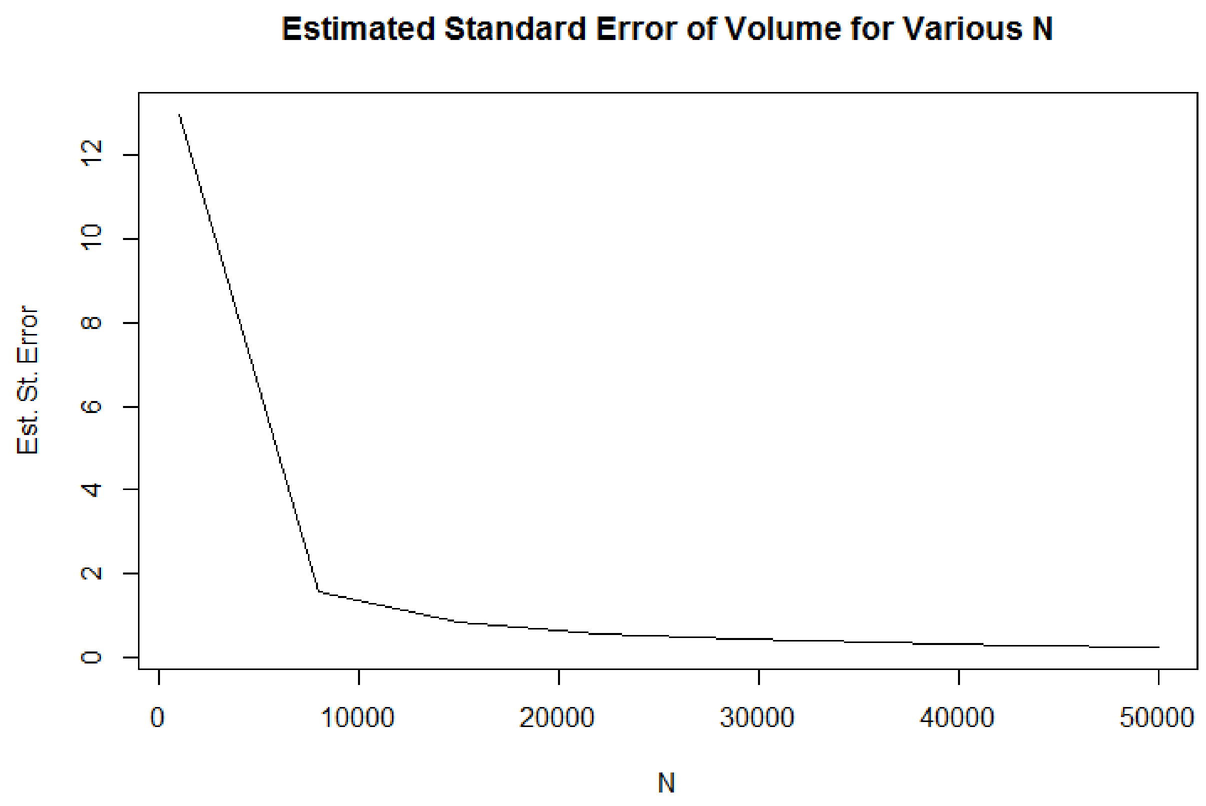
```r
volume_fun = function(N, data)
{
   # initialize S1, S2
   S1 <- 0
   S2 <- 0

   x <- data$V1
   y <- data$V2
   z <- data$V3
   rad <- data$V4

   x_dist <- (max(x)+rad[which.max(x)])-(min(x)-rad[which.min(x)])
   y_dist <- (max(y)+rad[which.max(y)])-(min(y)-rad[which.min(y)])
   z_dist <- (max(z)+rad[which.max(z)])-(min(z)-rad[which.min(z)])

   # calculate volume of rectangular box
```

```r
  volume_r <- x_dist*y_dist*z_dist

  # generate N x, y, and z coordinates
  x.coord <- runif(N, min(x)-rad[which.min(x)], max(x)+rad[which.max(x)])
  y.coord <- runif(N, min(y)-rad[which.min(y)], max(y)+rad[which.max(y)])
  z.coord <- runif(N, min(z)-rad[which.min(z)], max(z)+rad[which.max(z)])

  # combine into data.frame
  rangen_coords <- data.frame(V1 = x.coord, V2 = y.coord, V3 = z.coord)

  # create distance matrix
  dist_matrix <- data.frame(dist2(data[ ,1:3], rangen_coords))

  for(i in 1:N)
  {
    # find index with smallest distance value for i'th row
    idx <- which.min(dist_matrix[ ,i])
    # give smallest distance value for i'th row
    sm_dist <- dist_matrix[idx, i]
    # find corresponding radius
    radius <- data[idx, 4]
    if(sm_dist <= radius)
    {
      S1 <- S1 + 1 # S1 = S1 + f(x_i)
      S2 <- S2 + 1*1 # S2 = S2 + f(x_i) * f(x_i)
    } else {
      S1 <- S1 + 0
      S2 <- S2 + 0*0
    }
  }

  protein_volume <- volume_r * (S1/N)
  standard_error <- protein_volume * (sqrt((S2/N) - (S1/N)^2)/N)
  data.frame(volume = protein_volume, error = standard_error)
}
```

Varying N and the Weak Law of Large Numbers: The following code demonstrates the weak law of large numbers. As the total number of points generated within R grows larger and larger, the estimate of the volume $V(\phi)$ converges to the true volume. This also holds true for the standard error; as N increases, the error decreases, and will eventually converge to 0.

In the following code, the function volume_fun is repeated 20 times for each sample size N in the range [1000, 50000] by 7000 to obtain estimates of what the volume and errors are at that respective N. From the data frame that is generated, the weak law of large numbers is demonstrated as defined above.


Estimated Volume of Protein for Various N
Est. Volume (A^3)
36000
35500
35000
34500
34000
0
10000
20000
30000
40000
50000
N


Estimated Standard Error of Volume for Various N
Est. St. Error
12
10
8
6
4
2
0
0
10000
20000
30000
40000
50000
N

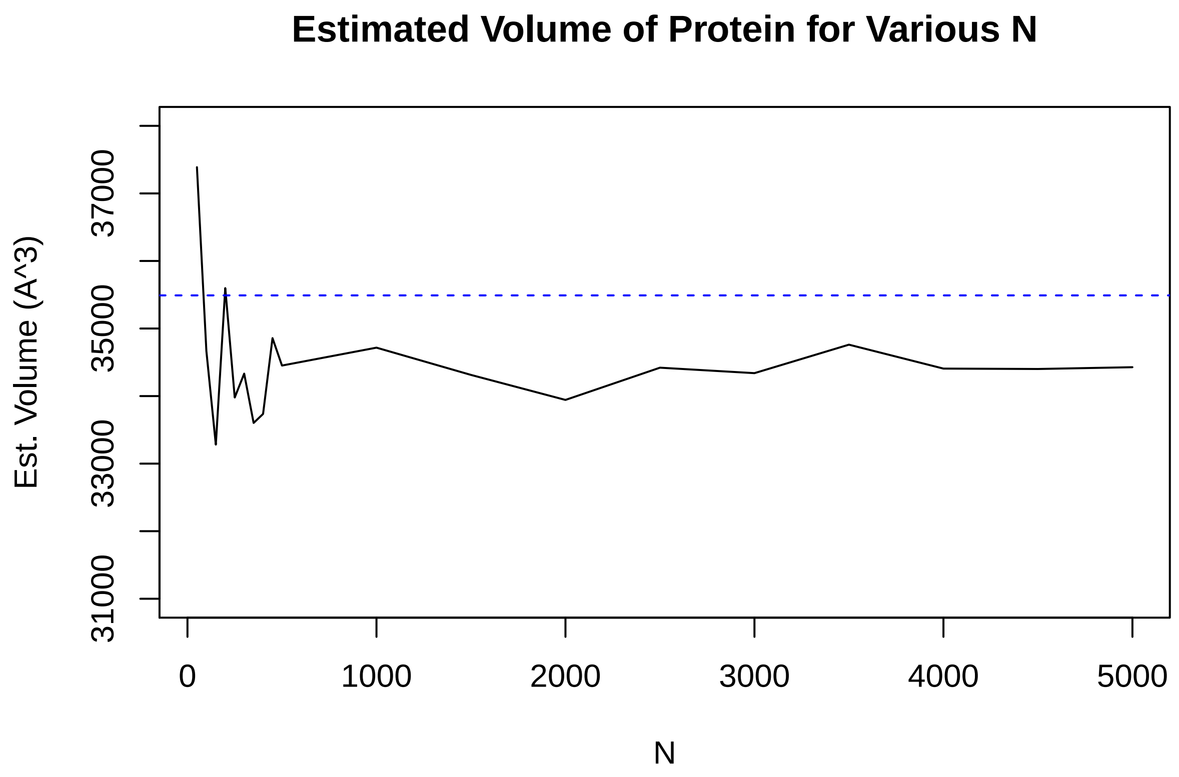
In this code, the function volume_fun is repeated 30 times for each sample size N in the ranges [50, 450] by 50 and [500, 5000] by 500 to obtain estimates of what the volume and errors are at that respective N. From the data frame that is generated, the weak law of large numbers convergence property is more obvious.

```
### Volume:

# Volume begins converging with larger and larger N:

lln_smallN <- sapply(seq(50, 450, by = 50),
                     function(i){ replicate(30,
                     volume_fun(i, protein)[1], simplify = "vector") })

# Volume converges well with large N:

lln_largeN <- sapply(seq(500, 5000, by = 500),
                     function(i){ replicate(30,
                     volume_fun(i, protein)[1], simplify = "vector") })

x_bar <- data.frame(volume = rep(0, 9))
for(i in 1:9)
{
   x_bar[i,1] <- mean(as.numeric(lln_smallN[,i]))
}

x_bar2 <- data.frame(volume = rep(0, 10))
for(i in 1:10)
{
   x_bar2[i,1] <- mean(as.numeric(lln_largeN[,i]))
}


### Standard error:

# Standard error decreases steadily with larger and larger N:

lln_smallN_se <- sapply(seq(50, 450, by = 50),
                        function(i){ replicate(30,
                        volume_fun(i, protein)[2], simplify = "vector") })

# Standard error starts coming close to 0 with large N:

lln_largeN_se <- sapply(seq(500, 5000, by = 500),
                        function(i){ replicate(30,
                        volume_fun(i, protein)[2], simplify = "vector") })

x_bar_se <- data.frame(error = rep(0, 9))
for(i in 1:9)
{
   x_bar_se[i,1] <- mean(as.numeric(lln_smallN_se[,i]))
}

x_bar_se2 <- data.frame(error = rep(0, 10))
for(i in 1:10)
{
```

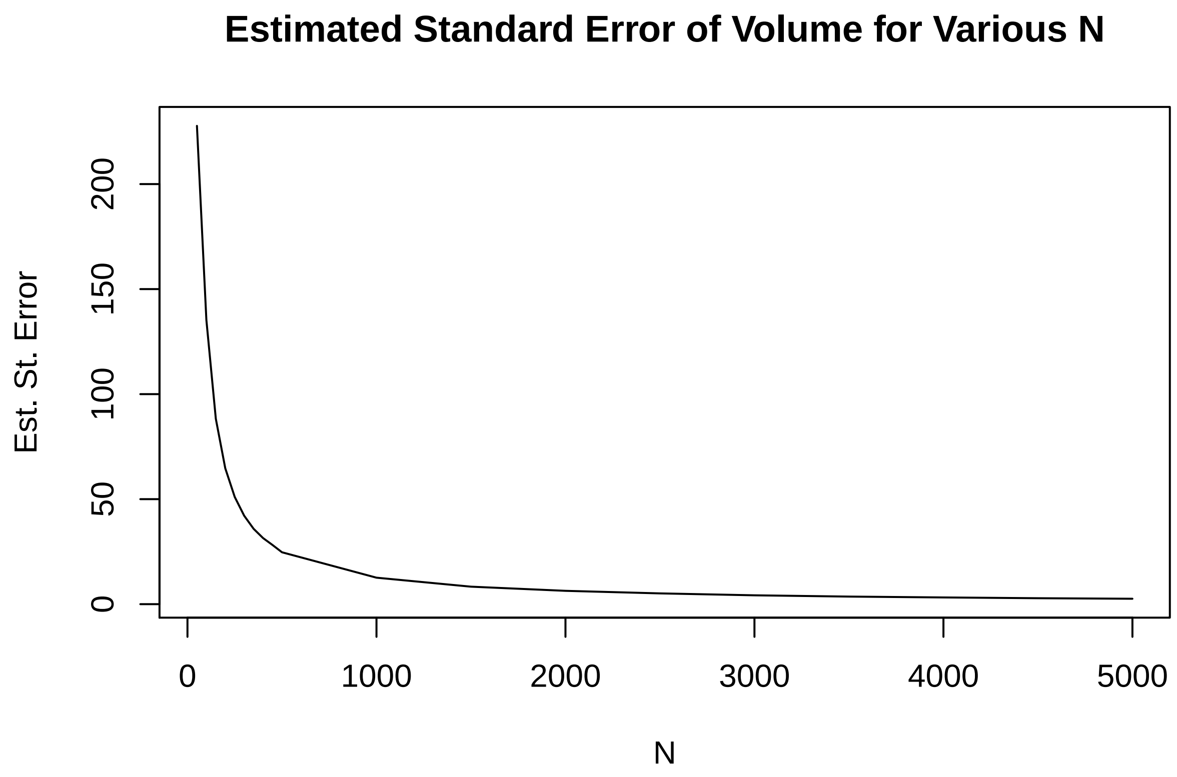
```r
   x_bar_se2[i,1] <- mean(as.numeric(lln_largeN_se[,i]))
}

volumes <- rbind(x_bar, x_bar2)
errors <- rbind(x_bar_se, x_bar_se2)
N_value <- data.frame(c("50","100","150","200","250","300","350",
                        "400","450","500","1000","1500","2000","2500",
                        "3000","3500","4000","4500","5000"))
names(N_value) <- "N"
final_df_N <- cbind(N_value, volumes, errors)
final_df_N
```

```
##       N   volume      error
## 1    50 37388.38 227.729461
## 2   100 34666.62 135.296541
## 3   150 33281.86  88.269828
## 4   200 35597.75  64.684180
## 5   250 33979.02  51.037303
## 6   300 34332.37  42.105127
## 7   350 33602.47  35.861582
## 8   400 33735.49  31.465786
## 9   450 34857.62  28.177572
## 10  500 34451.74  24.701326
## 11 1000 34716.76  12.607836
## 12 1500 34313.27   8.331497
## 13 2000 33943.20   6.345685
## 14 2500 34420.23   5.131879
## 15 3000 34339.53   4.208149
## 16 3500 34760.75   3.607606
## 17 4000 34406.98   3.194164
## 18 4500 34400.81   2.826365
## 19 5000 34427.39   2.579664
```

```r
plot(as.numeric(as.character(final_df_N$N)), final_df_N$volume,
     main = "Estimated Volume of Protein for Various N",
     xlab = "N", ylab = "Est. Volume (A^3)", ylim = c(31000, 38000), type = "l")
abline(h = 35490.34, col=4, lty=2)
```



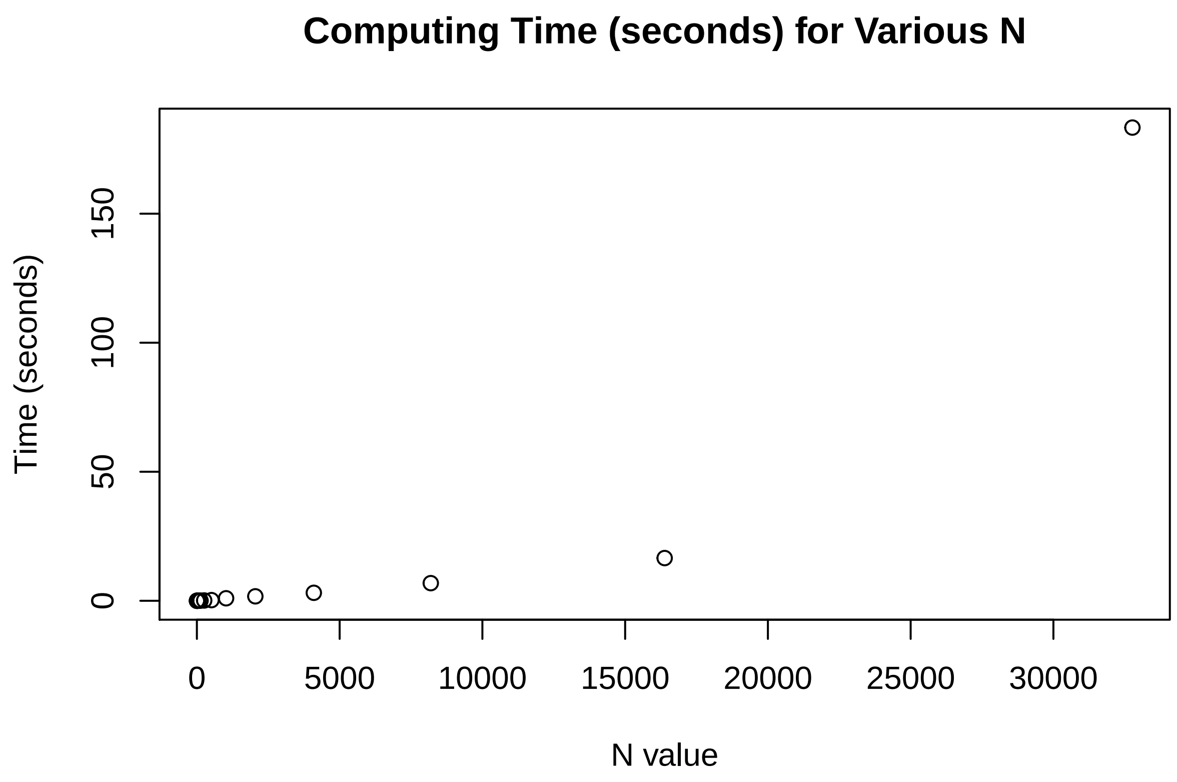
```r
plot(as.numeric(as.character(final_df_N$N)), final_df_N$error,
     main = "Estimated Standard Error of Volume for Various N",
     xlab = "N", ylab = "Est. St. Error", type = "l")
```



Time Consumption of Algorithm and Efficiency: If the main algorithm is run with a higher and higher value of N, the total elapsed time taken to run the code and the time taken by the CPU increases linearly until a certain N, around 15000, and then exponentially afterwards. This is likely caused by the for loop, which is iterating over all of the uniformly random points generated, calculating distance, and comparing them all to the radius. In R, using the apply family of functions is an often faster alternative method to the for loop, as using sapply(), for instance, is a vectorized operation - it applies a function to all elements of a list and returns a simplified output, usually a vector or matrix. Below, the number of points generated increase by a factor of 2 - it is evident that the total time taken is roughly proportional to N until a certain point around 15000, and then increases exponentially. A more feasible explanation is the machine on which the program is run has memory limitations so computationally expensive requests take much longer than what is expected. In conclusion, the computing power on the machine that this was generated is the likely culprit.

```r
vec <- vector()
for(x in 1:15)
{
   vec[x] <- 2^x
}

vec_times <- vector()
for(i in 1:15)
{
   vec_times[i] <- system.time(volume_fun(vec[i], protein))[3]
}
```

```r
plot(vec, vec_times, main = "Computing Time (seconds) for Various N",
     xlab = "N value", ylab = "Time (seconds)")
```

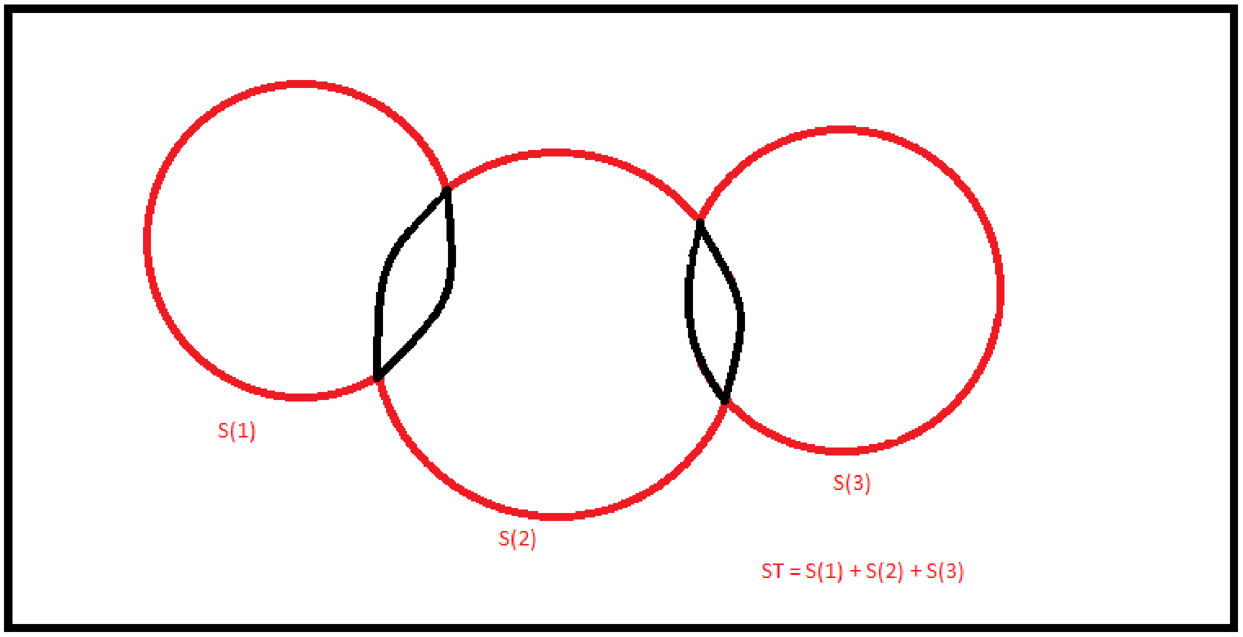


Also, when N is varied (when demonstrating the weak law of large numbers), it can be tested if these for loops, which are more time consuming when N is large and repetitions are large, are more or even less expensive. An example of the sapply() function alternative is below:

```r
A_1 <- function(i){mean(as.numeric(lln_smallN[,i]))}
sapply(1:nrow(x_bar_se), A_1)
```

Additionally, a method to improving efficiency of the algorithm overall is a major adjustment to how the distance matrix for every randomly generated point is rendered. Before, the distance matrix is calculated for every atom to the random point; instead, now the three-dimensional R can be partitioned into cuboids, dimensions determined by an atom in ϕ with the largest radius. For any given point, only the atoms' radii in the adjacent cuboids will be considered when generating the distance matrix. This implementation will drastically reduce computing time, especially for greater number of partitions.

Adapting the Algorithm for Surface Area: The algorithm used for volume can be adapted to compute surface area, the other major statistic we can use to examine a protein's physical properties. To begin, we can use a simple diagram in two dimensions to help us better understand the problem at hand.



Here, our goal is to estimate the surface area of the protein by sampling N_{points}. We are only interested in the surface area that is accessible (i.e. uncovered by another ball), so we only want the area that is in red. The total surface area of the protein would then be calculated as $TS = \sum S(i)$ for $i = 1, ..., N_{ball}$, where $S(i)$ is the uncovered surface area of the i-th ball.

The algorithm of our proposed program is as follows:

1) Initialize $TS = 0$, where TS represents the uncovered surface area of the entire protein.
2) For($i = 1$, N_{ball}){
    - Calcuate the surface area of the i-th ball: $ST(i) = 4\pi R(i)^2$.
    - Initialize N_{points}, the total number of points the user wishes to sample.
    - Initialize $N_{black} = 0$, the number of points that are not accessible i.e. covered by another ball.
3) For($j = 1$, N_{points}){
    - Generate N_{points} of x, y, z coordinates that lie on the surface of the balls.
    - Sample N coordinates from an uniform distribution.
    - Verify that the point is on the surface of a ball by checking that the distance from the point to the center of the i-th ball is equal to its corresponding radius, $R(i)$. (However, iterating over each ball is computationally inefficient, so it would be a good idea to implement preprocessing methods, e.g, divide the sample space into cells and only check the balls that are in nearby cells). Repeat this process until we have generated N_{points} on the surface.
4) For($k = 1$, N_{ball}, $k \neq i$){
    - If distance from the point to the center of the k-th ball is less than $R(k)$, then the point is not accessible.
    - $N_{black} = N_{black} + 1$
    - Break

   }

```
        5) Calculate number of accessible points by subtracting number of inaccessible points from total number
           of points: $N_{red} = N_{points}$ - $N_{black}$.
        6) Compute surface of $i$-th ball by multiplying proportion of accessible points by the surface area of $i$-th
           ball: $S(i) = \frac{N_{red}}{N_{points}} * ST(i)$.
        7) Add surface area of $i$-th ball to surface area of entire protein: $TS = TS + S(i)$
    }
}
```

Besides volume, the surface area of the protein is also integral to our understanding of molecular interactions. For example, both surface area and volume are used to quantify the interactions between large molecules and the solution they are in. In their paper, "Geometric Measures of Large Biomolecules: Surface, Volume and Pockets," Dr. Paul Mach and Dr. Patrice Koehl make the distinction between approximate and exact methods for computing volume and surface area of a molecule. Analytical, or exact methods based on the alpha shape theory for computing the metrics of a union of balls have been implemented in software packages. In particular, Dr. Mach and Dr. Koehl present UnionBall, a newer, and more efficient implementation used to characterize the geometry of molecules. UnionBall has an $O(n)$ behavior over large values of N, which is far more computationally efficient than our current algorithm. It would be interesting to explore the Delaunay tessalation and the detection and measurements of pockets, or internal cavities of biomolecules. Moreover, we could also look into biasing the order in which points are inserted to improve locality.

Extensions to Importance Sampling: The Monte Carlo integration technique that has been simulated by the main algorithm here has worked well when sampling from the uniform distribution. However, if it is not desired to sample from that distribution, a sample can instead be drawn from an alternative distribution. The integral can then be re-weighted using importance weights so that the original distribution that is desired, in this case uniform, is targeted instead. For example, in some integral

$$I = \int h(y)f(y)dy$$

h is a function and f is the PDF of Y. If f is difficult to sample from, one can introduce an alternative PDF of another distribution, g, to sample from instead:

$$I = \int \frac{h(y)f(y)}{g(y)} g(y)dy = E\left[\frac{h(Y)f(Y)}{g(Y)}\right].$$

In $g(y)$ selection, it is common to choose a distribution that has a similar shape to $f(y)$ but with thicker tails. Also, choose a distribution and parameters that estimate the true mean μ of the original distribution the best (low bias) and that has the lowest variance. If samples from the normal distribution were desired instead of those from the uniform distribution, the function f is uniform:

$$f(y) = \frac{1}{b-a}$$

and g is normal

$$g(y) = \frac{1}{\sqrt{2\pi}} e^{\frac{-t^2}{2}}.$$

The importance weights appear as $\int g(y)dy$ in the denominator so that the resulting PDF has a total area of 1. This entire process can also work just as well (or better) with f as a normal distribution and g as the uniform, if the conditions are appropriate.

Conclusion: We have explored the algorithms to find the volume and surface area of a protein, indicated some ways to improve efficiency, and dabbled in the concepts of the weak law of large numbers and importance sampling. From here, these algorithms can be used directly to make conclusions about some of the physical properties of proteins mentioned in the introduction. If the efficiency of the programs is improved, we may be able to exponentially increase the algorithm's speed and refine the accuracy of our estimates at large N, therefore optimizing user-friendliness.

The dataset that we have tested our algorithm on contains 2775 atoms; in other cases, we may want to use our program to estimate the volume of a protein that consists of millions of atoms, and that would be extremely inefficient given our current algorithm. As we have mentioned in the results section, we would like to explore pre-processing methods such as dividing the prism into cuboids and only checking the distance from the point to spheres in nearby cuboids rather than iterating over all of the spheres. The analytical method that Professor Patrice Koehl kindly showed us during office hours easily out-performs our program, so it would be very interesting to look into examples of exact methods and, in particular, the UnionBall software.

Our volume program can also be adapted to estimate surface area, and various sampling distributions can be used within the algorithms, with the underlying ideas of importance sampling, used carefully, playing a central role. With estimates of both the surface area and volume, we can then calculate the ratio between them to characterize the compactness and globularity of the protein. Having a better understanding of this ratio will give us more insight into how the proteins interact with each other and with their environment. Estimating the geometry of proteins is just the beginning; we hope to take what we learned from working on this project to further explore the dynamic interactions between biomolecules.

References: Cisewski, J. Importance Sampling [PowerPoint Slides]. Retrieved from Carnegie Mellon University, June 2014.
P. Mach and P. Koehl. Geometric measures of large biomolecules: Surface, volume, and pockets. J. Comp. Chem., 32:3023-3038, 2011.