Project 1: PCR Simulation in Python

Name: Alex Karwowski

Group Number: 8

Genome: nsp4

Table of Contents

Abstract

The project focuses on simulating PCR using python or other programing languages. PCR is short for polymerase chain reaction and is used to rapidly make millions to billions of copies of a specific DNA sample, in this case, a gene of the SARS-CoV-2 virus. In this paper I document the experiment and its outcome to better understand how computer modeling can be used in the field of Medicine and Biology.

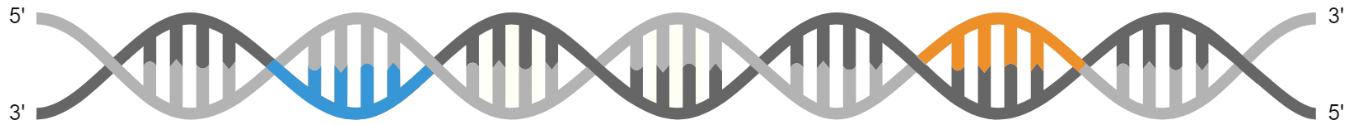
Goals and Objectives

The goal of this project was to make a simulation of the PCR process. The objectives are as follows:

1. Obtain our gene sequence
2. Find the compliment of our sequence so we can have the full DNA strand
3. Obtain our primers
4. Model the steps of PCR
    1. Start with the denaturation process
    2. Apply the Annealing process
    3. Apply the Elongation process
    4. Repeat the cycle
5. Perform statistical analysis on the sequences
6. Present the results of our simulation

Description of PCR Process

There are 3 basic steps of the PCR process: Denaturation, Annealing, and Elongation. But before you can start these steps, there is some preparation that is needed. First you must have the DNA sequence as well primers generated from BLAST.

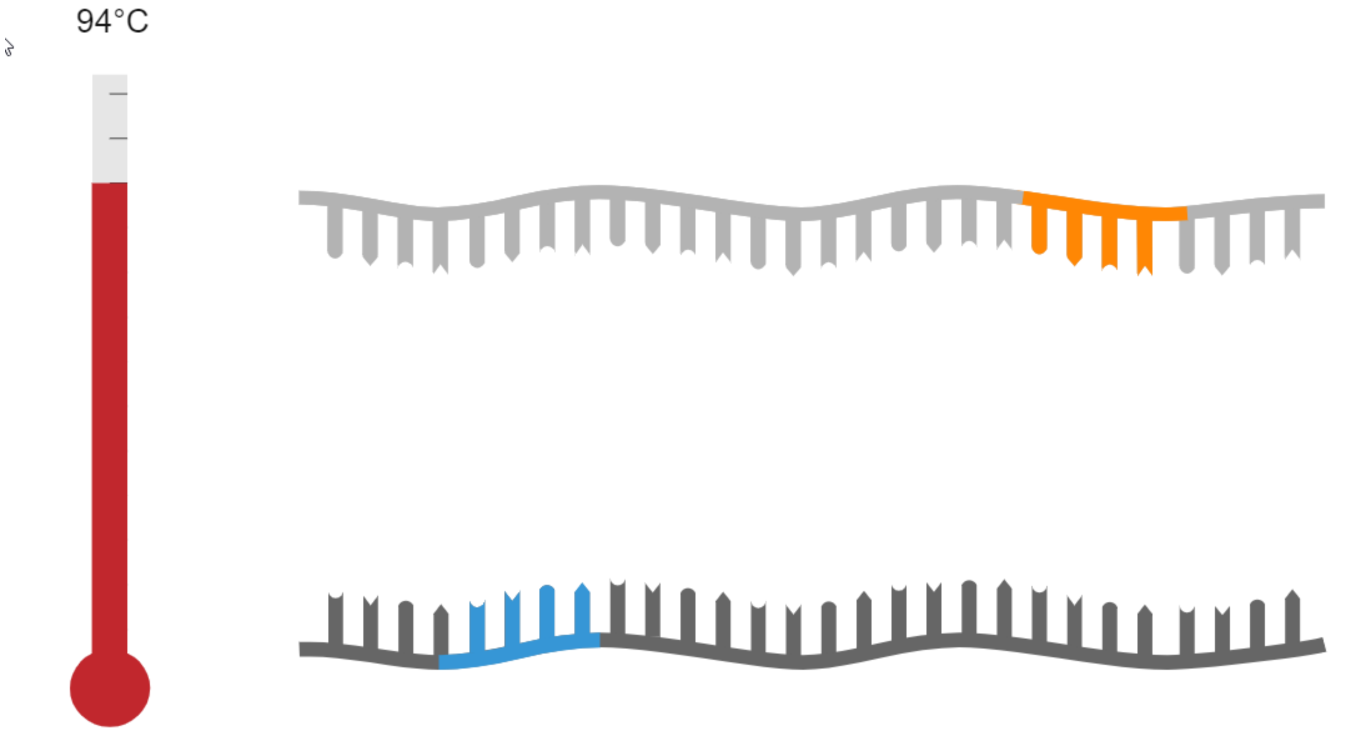


Note: Blue is our Forward Primer, and Orange is our Reverse Primer

---

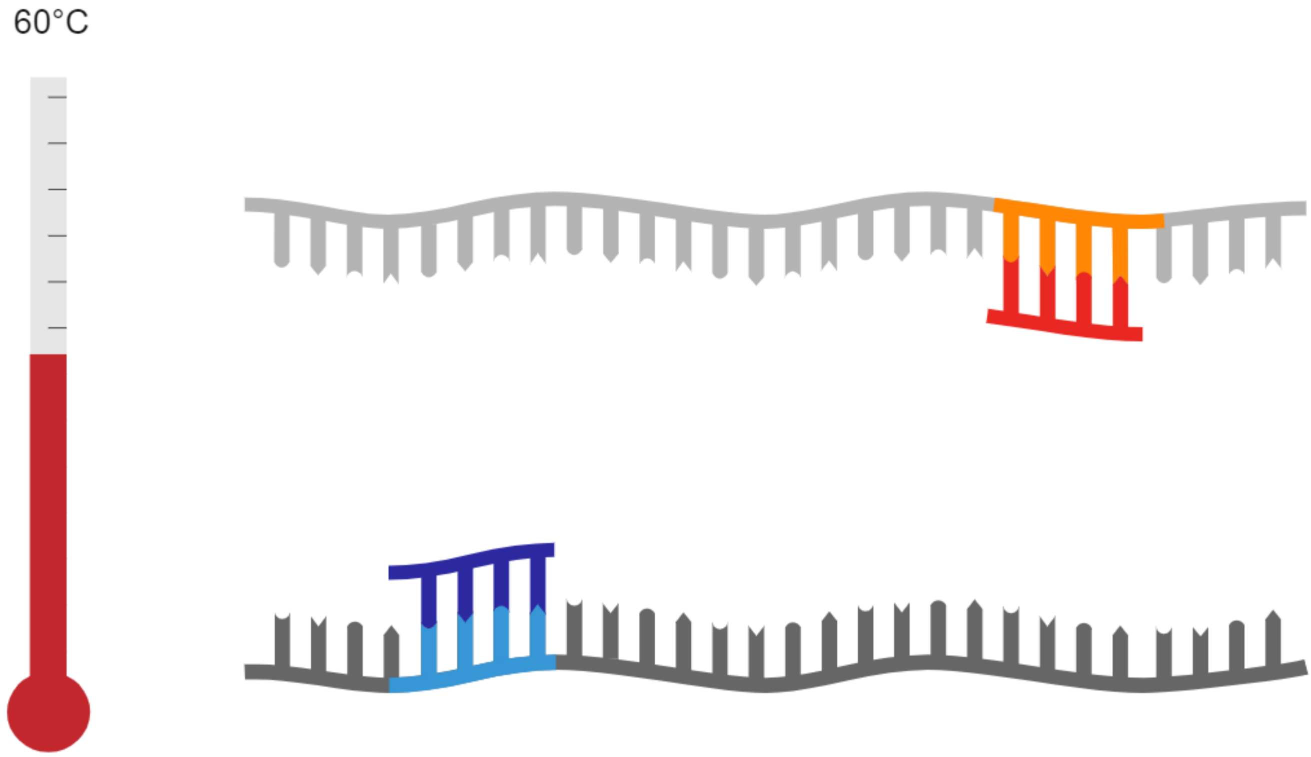
Step 1: Denaturation

This step involves splitting a peice of double stranded DNA into separate strands. This is done by heating up the DNA sequence to around 94 Degrees Celsius. By heating the DNA up the bonds between both halves of the DNA weaken and eventually split.



Step 2: Annealing

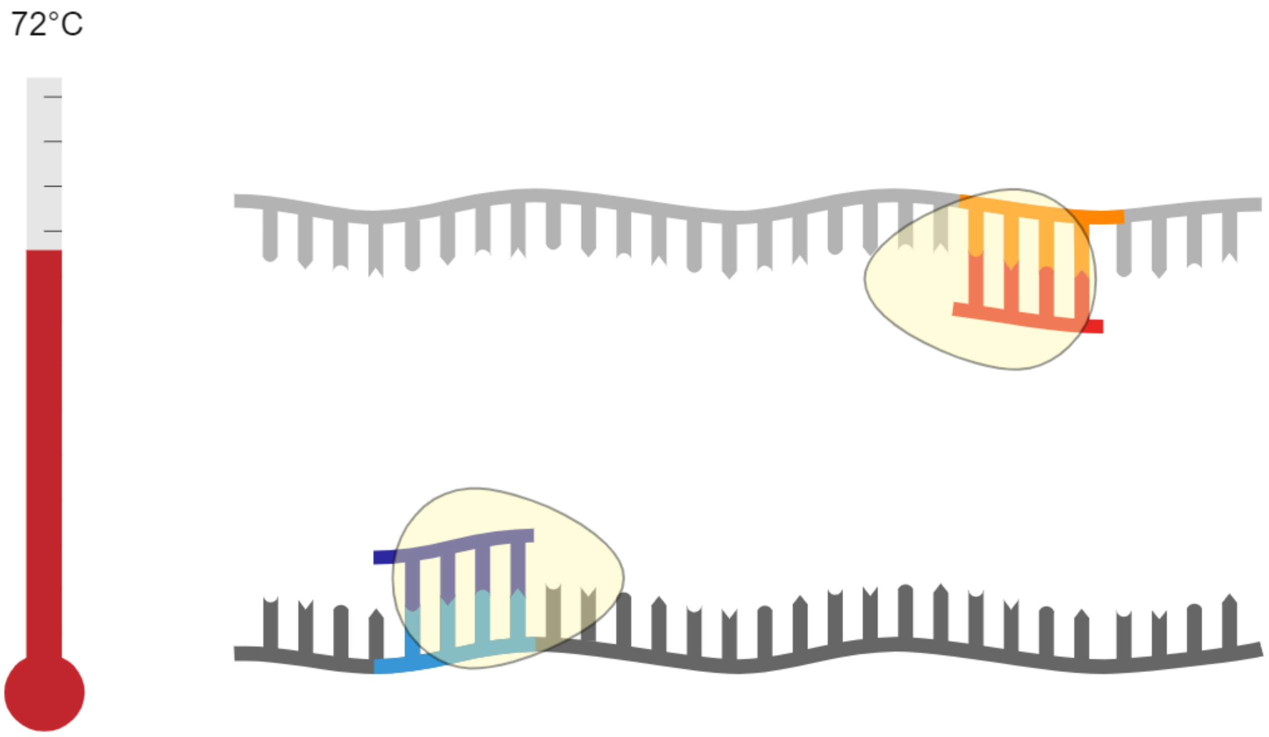
This step is fairly simple. First the temperature is lowered to around 60 Degrees Celsius, and a complimentary primer is attached to each strand. These primers are generally 15-30 nucleotides in length with a GC content of 40-60%.

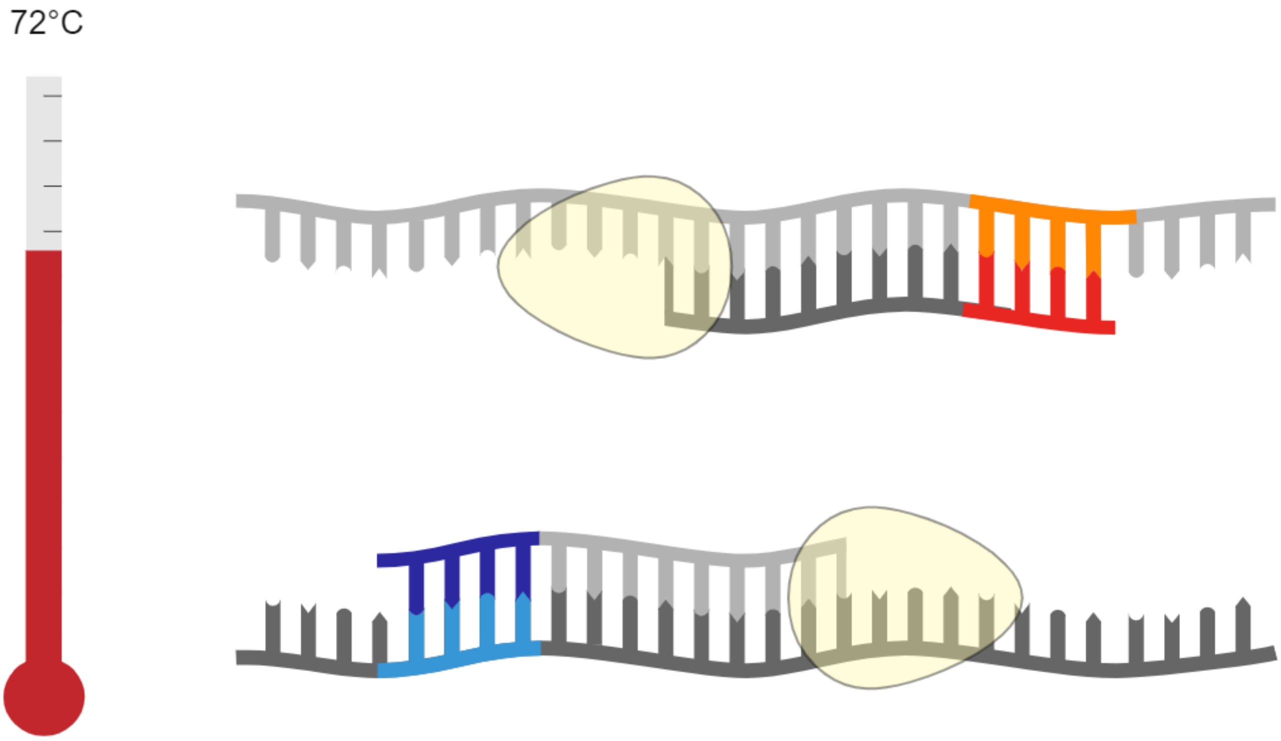


Note: Red and Blue are the primers generated by Blast and are compliments to the Light Blue and Orange segments respectively.

Step 3: Elongation

This is the last step of the PCR process. The temperature is raised to around 72 Degrees Celsius and 2 TAQ Polymerase Bind to the primers and then create a complimentary strand for the single strand to bind to. Then the cycle repeats.





Methods

We used driver code and 3 functions in our PCR simulation:

- `__main__`
- `find compliment`
- `run PCR`
- `find statistics`

main

The driver part of our code. This section of code performs all setup needed for PCR to run properly as well as calls the `run PCR` function

```
if __name__ == '__main__':
    random.seed(99)
    start_time = time.time()

    # Setup Step 1: Read Contents of File
    with open('genome.txt', 'r') as file:
        RNA = file.read()

    # Make the string all uppercase for ease of use
    RNA = RNA.upper()
    # Setup Step 2: Find Compliments
    DNA = find_compliment(RNA, 0)

    # Setup Step 3: Define Primers
    # ("Sequence, Starting Point, Ending point, GC Content")

    fPrimer = ("GGTTTTGTCGTGCCTGGTTT", 297, 317, .5)
    rPrimer = ("AGCAGCCAAAACACAAGCTG", 464, 443, .5)  # Sequence is reversed

    replicated_DNA = run_PCR(DNA, fPrimer, rPrimer,
                             cycles=20, falloff_base=180)
    print('PCR executed in: ', time.time() - start_time)
    find_statistics(replicated_DNA)
```

It reads the genome file that contains our RNA. Then it changes it to upper case for ease of use. Next we run `find compliment` and assign it to a tuple called DNA. Finally we assign the primers their sequence that we got from blast. All thats left is to run `run PCR` and `find statistics`.

find compliment

This is just a simple function that takes a string of RNA and a flag and either returns a tuple of the original string and the compliment or just the compliment based on the flag passed in.

```
# Find the Compliment
def find_compliment(RNA, flag):
    """
    Find the compliment to submitted RNA
    :param RNA: String of RNA
    :param flag: determine how to return, As a string, or a Tuple
    :return: Tuple of original string and complement, or a
             single compliment string
    """
    cDNA = RNA.upper()
    # Replace the nucleotides for the complementary DNA strand
    cDNA = cDNA.replace("A", "X")
    cDNA = cDNA.replace("T", "A")
    cDNA = cDNA.replace("X", "T")
    cDNA = cDNA.replace("C", "X")
    cDNA = cDNA.replace("G", "C")
    cDNA = cDNA.replace("X", "G")

    if flag == 1:
        # Return Single String
        return cDNA
    else:
        # Create DNA Object
        DNA = (RNA, cDNA)
        return DNA
```

---

run PCR

This function runs the meat of the PCR function. I will only be talking about specific parts of the program because the group had issues splitting the code up into readable functions (one of our many issues we faced). The parameters passed in are pretty self explanatory.

```
def run_PCR(dna, forward_primer, reverse_primer, cycles=10, falloff_base=180):
    """
    This function will run a simulation of PCR using the submitted DNA segment and primers.  For each cycle, replicate
    each double stranded DNA tuple within the list.
    Within each double stranded tuple, replicate each strand. Strand replication is done by finding which primer is the
    compliment to the input strand. The rest of the strand is then looped through, finding the rest of the base pairs.
    One large list of DNA tuples is then exported as a result.
    :param dna: Tuple with original RNA strand and cDNA strand
    :param forward_primer: Tuple for forward primer. Contains: sequence, starting point, ending point, GC content
    :param reverse_primer: Tuple for reverse primer. Contains: sequence, starting point, ending point, GC content
    :param cycles: Number of cycles to simulate. Defaults to 10.
    :param falloff_base: The base falloff rate to be incremented using a random int between -50 and 50.
    :return: list with each entry being half of a replicated DNA string. Next
```

```
entry is the other half of the strand
"""
```

Step 1: Denaturation

```
for dna in replicated_dna:
# New double strand of DNA (convert to tuple at end of iteration)
new_pair = list()
# For loop definition does the equivalent of denaturation
for strand in dna:
```

Step 2: Annealing

```
# Calculate fall off rate
falloff_rate = falloff_base + random.randint(-50,50)
# Start of Annealing Step: Add the correct primer to the strand
# Search for the reverse primer compliment in the strand
if reverse_compliment[1] in strand:
    # Reverse strings to iterate through lists forward -> back
    # Add the reverse primer to a str variable
    strand_to_add = reverse_sequence[::-1]

#.........#

# Otherwise search for the forward primer compliment in the strand
elif forward_compliment[1] in strand:
    # Add the forward primer to a str variable
    strand_to_add = forward_sequence[:]
```

Step 3: Elongation

```
# Start of Elongation Step: Attach
# copy up to the length of the falloff_rate
taq_polymerase = find_compliment(reverse_strand[start_index:end_index], 1)
strand_to_add = strand_to_add + taq_polymerase
# reverse string again for correct 5'-3' order
strand_to_add = strand_to_add[::-1]
# add to new strand to DNA pool
new_pair.append(strand_to_add)

#.........#

# Start of Elongation Step: Attach
# copy up to the length of the falloff_rate
taq_polymerase = find_compliment(strand[start_index:end_index], 1)
strand_to_add = strand_to_add + taq_polymerase
```

```
# add to new strand to DNA pool
new_pair.append(strand_to_add)
```

The elongation step first finds both the start and end index which is based off of the primer length and the falloff rate. Then the taq_polymerase stores the values of the compliment (based off of the original strands start and end positions). This is then added to a the strand with the primers all at once in an effort to improve function runtime.

---

find statistics

This function is fairly simple as well. The replicated DNA we got from the `run PCR` function is passed in. The following are calculated:

- Number of Strands
- Max Length
- Min Length
- Average Length
- Average GC content

Note: Statistical Analysis Genius **Justin Hare** wrote this function

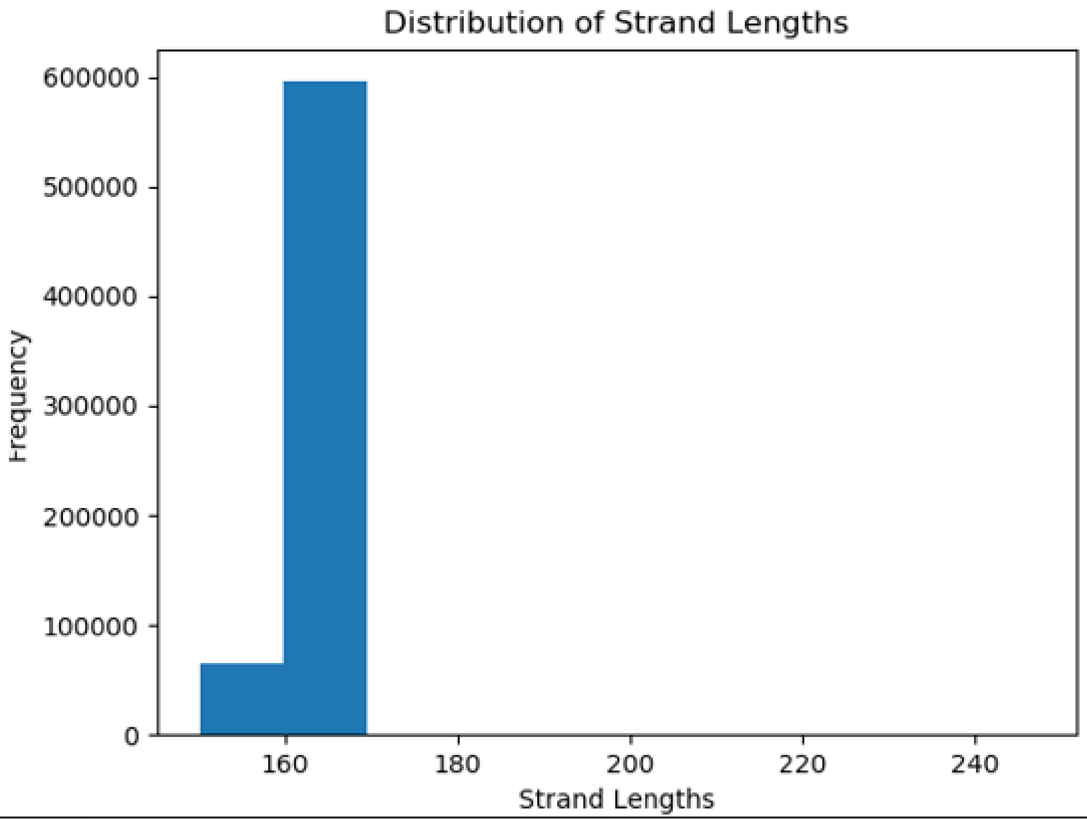
```
def find_statistics(replicated_dna):
    """
    Find statistics on replicated DNA. Finds strands,
    max strand length, min strand length, and average strand length.
    Finds average GC content of strands. Plots distributions of strands
    :param replicated_dna:
    :return:
    """
    segment_lengths = []
    gc_contents = []
    for pair in replicated_dna[1:]:
        for strand in pair:
            if strand != '':
                segment_lengths.append(len(strand))

                # Find GC contents of both strands
                num_of_c = strand.count('C')
                num_of_g = strand.count('G')
                gc_content = num_of_c + num_of_g
                gc_contents.append(gc_content)

    num_of_strands = len(segment_lengths)
    max_length = max(segment_lengths)
    min_length = min(segment_lengths)
    avg_length = sum(segment_lengths) / len(segment_lengths)
    avg_gc_content = (sum(gc_contents) / len(gc_contents)) / avg_length

    hist = plt.hist(segment_lengths)
```

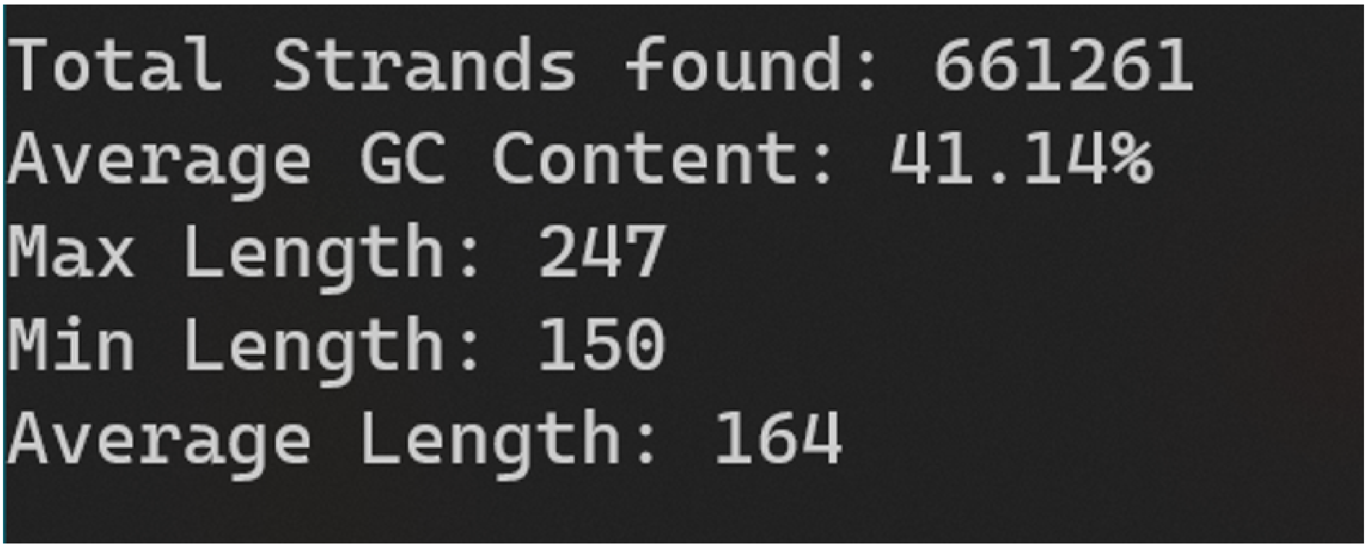
```
        plt.xlabel('Strand Lengths')
        plt.ylabel('Frequency')
        plt.title('Distribution of Strand Lengths')
        print(f'Total Strands found: {num_of_strands}')
        print(f'Average GC Content: {"%0.2f" % (avg_gc_content*100)}%', )
        print(f'Max Length: {max_length}')
        print(f'Min Length: {min_length}')
        print(f'Average Length: {avg_length}')
        plt.show()
        return
```

Results

Results are shown below.

Note: The distribution between lengths of 160 and 170 are so small its unnoticeable in graphs.





```
Total Strands found: 661261
Average GC Content: 41.14%
Max Length: 247
Min Length: 150
Average Length: 164
```

Discussion

The total number of strands found was 661,261 with an average GC content of 41.14%. This is low boardering failure. GC bonds contribute to stability because they are stronger, so the higher the GC content, the more stable the strand will be. While we did not account for this in our program, however it does have applications in the real world.

When adjusting the falloff base rate as well as our range the GC content rose and fell appropriately and accordingly, so I believe everything is correct.

We only ran 20 cycles due to performance issues. If I had more memory or increased the falloff base and range, then we could run near 30-40 cycles in my current setup with 32GB of ram. If I had 64Gb of ram I feel like I would be able to get between 40 and 50 cycles for more accurate results.

Some issues that we ran into programming wise, were modularity and breaking up our function into the appropriate steps. We also had issues figuring out how to implement the Elongation step. The bigest issue we had that I ended up solving was improving the run time of the program. Initially cycles took 1 second then every subsequent cycle took `1 * 2^n` seconds where n was the cycle number. This was extremely slow. So I replaced a bunch of if-else conditionals that were tasked with generating the compliment one at a time with a dictionary. This reduced the runtime of 20 cycles to roughly. Eventually I redid out compliment function and updated the Elongation step to use that instead and reduced a 20 Cycle runtime to about 2 seconds.

Conclusion

Some draw backs of the project are that this is a perfect enviroment and things can and would change in a real life setting.

An outside the matrix scenario would be using the PCR to isolate specific genes that have weak GC content. So we could experiment on breaking those genes down with medicine and other antibodys to distrupt the virus and reduce its effectiveness.

References

- Wikipedia - https://en.wikipedia.org/wiki/Polymerase_chain_reaction
- Bio-Rad - https://www.bio-rad.com/webroot/web/movies/lse/global/english/what-is-polymerase-chain-reaction/tutorial.html