Implementation of a customizable and simple to use Gene Transcription Simulator

Eduardo da Veiga Beltrame

May 9, 2016

Abstract

There is currently no unified framework for modeling gene expression at the level of individual polymerases and base pairs. There is a large diversity of published models which share many common features[1, 3, 4], but are generally inaccessible to newcomers. Using Python and the Gillespie algorithm[2], we have implemented a stochastic simulator for gene expression. The source code is less than 200 lines and is openly available.

Users can model gene architectures by changing 5 rates for each gene locus: elongation, backtracking, binding, termination and abortion. The program output is a graphical representation of gene occupancy over time, a histogram of time interval between consecutive terminations, a histogram of the time taken per polymerase to transcribe the gene, and a graph of total terminated polymerases.

This platform should be useful for individuals who are not interested in developing models from scratch either analytically or computationally, but would still like to investigate how changing parameters such as binding or elongation rate affects gene expression.

1 Introduction

We define a gene architecture as a set of gene elements and their specific order. Gene elements are stretches of DNA that when part of a gene have some effect - such as promoters, pause sites, termination sites, and elongation sequences. The aim at defining gene elements is to be able to think of them in terms of function and effect, at a higher abstraction level than that of the DNA sequence per se. Gene elements may be thought of as blocks, and genes are block diagrams, such that the specific blocks used define the gene architecture.

When developing quantitative models for gene expression, having a modular modeling framework makes it easy to alter the modeled gene by simply changing the gene elements (modules) or defining a new element. Being completely modular, elements may be thought of as functions, with inputs (for example, the times at which a polymerase arrives at the element, signaling molecules that may alter the elements status) and outputs (for example, how long a polymerase took to traverse the element, or whether it was knocked off).

We thus can develop a customizable simulation platform. Users can build their own architectures and easily run simulations using a already developed tool set, while being able to add their own elements when necessary. Such a

platform lowers the entry barrier for theoretical investigations, and it can reveal unexpected phenomena. If users are readily able to test by modeling them, it empower users to pursue investigation lines that be too burdensome to study otherwise. This can also yield results for systems too complicated to be treated analytically, whose dynamics which would not be intuitively apparent.

2 Modeling Framework

Instead of considering a certain number of base pairs between each polymerase, we instead consider that the smallest stretch of DNA we can simulate is as long as a the polymerase footprint.So the smallest 'hop' that the polymerase perform is equal to it's footprint, which is about 40 base pairs[5]. This should not affect simulation results, but considerably simplifies the model and minimizes the necessary computation.

We consider genes as unidimensional lattices of length L where each locus in the lattice correspond to a stretch of DNA the size of the polymerase footprint. Each position may be either empty or be occupied by one polymerase at a time.

Each locus takes 5 parameters for the different possible events:

1. **Elongation rate k:** If the locus is occupied by a polymerase, it can take a step forward, freeing its location and occupying the next one.
2. **Backtrack rate r:** If the locus is occupied by a polymerase, it can take a step back, freeing its location and occupying the previous one.
3. **Binding rate B:** If the locus is free, a new polymerase may bind to it.
4. **Termination rate T:** If the locus is occupied by a polymerase, it can become unoccupied, and an RNA transcript is produced.
5. **Abortion rate A:** If the locus is occupied by a polymerase, it can become unoccupied, but no RNA transcript is produced.

The rates define how many "events per unit of time" happen on average. They are all treated as simple biochemical reaction steps, and have an exponential probability distribution over time. The concatenation of several such simple steps leads to different resulting probability distributions.

Figure 1 illustrates an example gene we could build with 14 Loci. In it, locus 1 is a binding element, where polymerase can bind with rate B, and elongate with rate k. The backtrack, abortion and termination rates are all zero on locus 1. On loci 2 to 3, 5 to 10 and 12 to 13, it can elongate with rate k, or backtrack with rate r, while all other rates are zero. We could say that those are 3 elongation elements of length 2, 5 and 6 respectively. Locus 4 is also an elongation element with backtrack rate r, but with elongation rate p. If we consider $p < k$, then locus 4 would be a pausing element, where polymerase takes a longer time than normal to advance. On locus 11, in addition to normal elongation rates, the polymerase may abort with rate A, and so this is an abortion element. Finally on locus 14 it can either backtrack with rate r, or terminate with rate T and produce an RNA transcript. Note that it cannot elongate (the elongation rate is 0), this is the end of our gene.

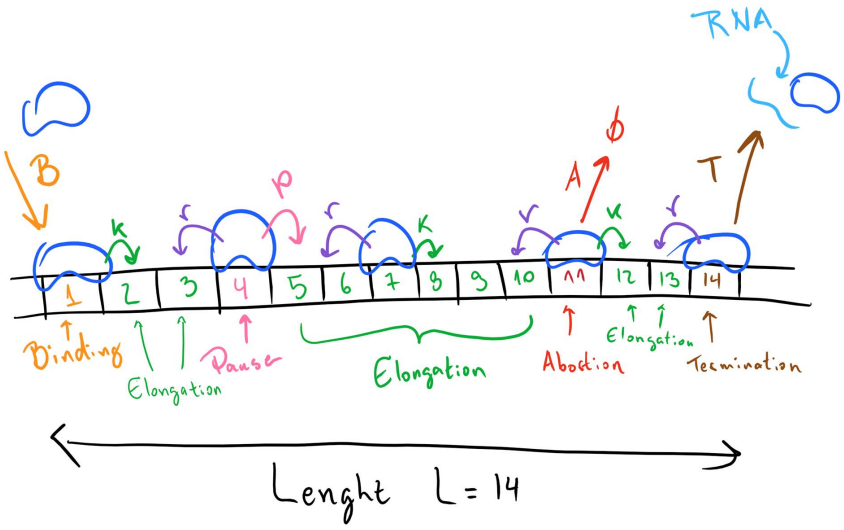

Figure 1: Example gene with 14 loci.

3 Simulation Implementation

The simulation was implemented as a Python script using the Gillespie algorithm. After reading the user input and initializing an empty gene, the algorithm loops trough the following steps until the specified end time is reached:

1. The current state of the gene is parsed through, and the rates for all possible actions A_i are summed into a value $Z = \sum_i A_i$.
2. One of the possible actions, A_i, is randomly chosen with probability $p_i = k_i/Z$, where r_i is the rate of A_i.
3. The time Δt the action A_i takes to happen is drawn from the distribution $p(t) = e^{-t \cdot r_i}$.
4. The total simulation time is increased by Δt, and the gene state is updated according to the action taken.
5. If the total time is less than the specified end time, the loop restarts, otherwise the simulation ends and the output is displayed.

In order to illustrate the algorithm, consider a gene with 3 loci, as shown in figure 2. In the current state there is only one polymerase bound, on location 2. Four different things can happen:

1. The polymerase can backtrack, at rate r.
2. It can elongate, taking a step forward at rate k
3. It can abort, leaving the location free at rate A
4. A new polymerase can bind on locus 1 with rate B, resulting in a state loci 1 and 2 occupied.

In this case $Z = B+r+A+k$. The probability of a given action A_i happening is dependent on it's corresponding rate normalized by Z, so $p_i = r_i/Z$. Then we randomly draw a time interval from the distribution $p(t) = e^{-r_i \cdot t}$

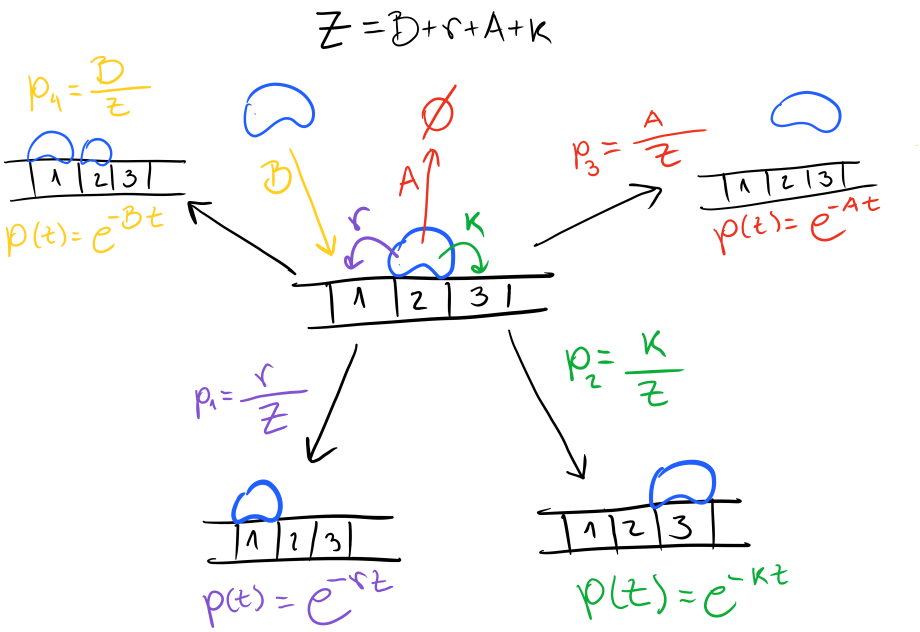

Figure 2: Depiction of the four possible transitions for the example gene discussed.

4 Simulation input

The current implementation of the simulator still lacks a graphical user interface, and input must be made altering the script itself. Simulation time is specified by the variable "totaltime".

The parameters for each element of the gene to be simulated, the user must alter the list "elements" at the beginning of the script. Each item in the list corresponds to a gene element,the parameters are specified as: [element length, elongation rate, backtracking, binding rate, termination rate, abortion rate].

So the gene discussed in figure 1 would be specified with the following:

```
elements = [
           [1, 1,   0,   0.1, 0, 0   ],      # 1 loci binding
           [2, 1,   1,   0,   0, 0   ],      # 2 locus elongation
           [1, 0.1, 0.2, 0,   0, 0   ],      # 1 loci pause
           [6, 1,   0.2, 0,   0, 0   ],      # 6 locus elongation
           [1, 1,   0.2, 0,   0, 0.1 ],      # 1 loci abortion
           [2, 1,   1,   0,   0, 0   ],      # 2 locus elongation
           [1, 0,   0.2, 0,   1, 0   ],      # 1 loci termination
           ]
```

5 Simulation Output and Example cases

The simulator will output a graph containing four plots conveying the results.

On the left there is a "Gene occupancy plot". Each line corresponds to a snapshot of the gene in a moment of time, with white pixels representing empty locations and colored pixels representing polymerases. Polymerases are colored in the order which they bound.

On the right there are two histograms, one of the times they took to transcribe the gene, and another of the intervals between two consecutive termination events. And at the bottom there is a plot of the number of terminated polymerases over time, so the user can get a sense of whether the transcription rate is changing over time.

5.1 Gene with pause and abortion

For our first investigated system, we discuss the gene shown on on figure 1.We investigate 3 cases: The output when the pause element is changed to a normal elongation element, and the output when in addition to that, the initiation rate is greatly increased.

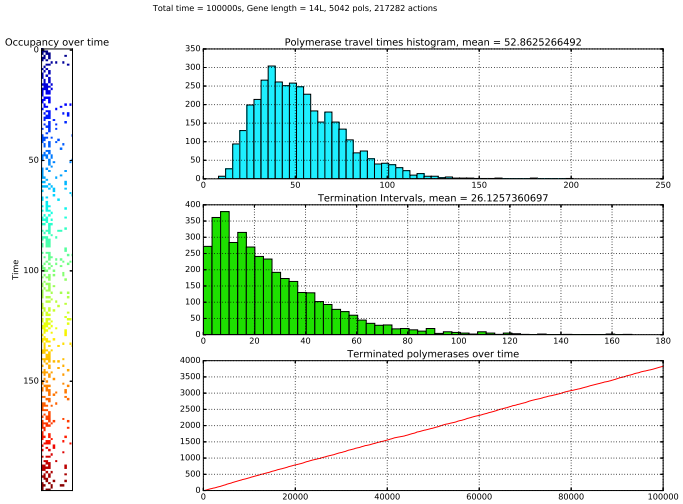

Figure 3: Simulation output for the gene discussed in figure 1. Notice that because the pause rate there is some traffic jam in the first 3 loci. Also note some polymerase stalled on the abortion element, but there is no traffic jam: they abort before polymerase have time to accumulate

5.2 Antenna model for transcription initiation

Although not a gene per se, our simulator is actually exactly able to simulate a model system for transcription initiation: the antenna model.

Most gene expression models consider, like we did, that the polymerase binds at a specific location, the initiation complex forms and transcription starts.

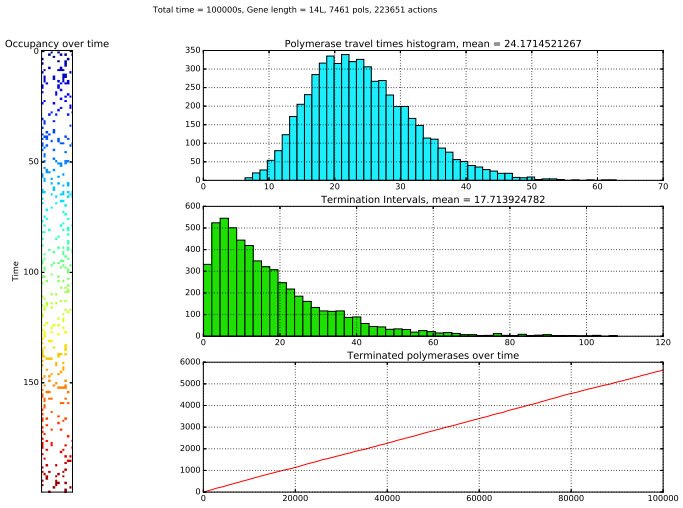

Figure 4: Simulation output for the gene discussed in figure 1, but with the pause element changed to a normal elongation element. Notice there is no more traffic jam in the beginning, and also the shifts in the histograms.

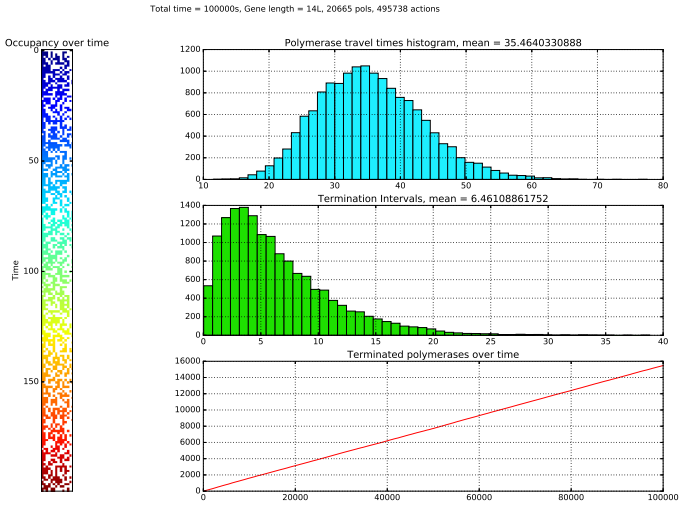

Figure 5: Simulation output for the gene discussed in figure 1, but with the pause element changed to a normal elongation element and a very high binding rate (4). Notice now the entire gene is occupied most of the time, but that most polymerases get knocked off at the abortion element. Notice that because of the traffic jams, the average transcription time increases, but the interval between terminations decreases.

There is an alternative model where the polymerase binds weakly and non-specifically all over the genome, without forming a initiation complex. It is then able to slide up and down the genome with no preferred direction, performing a random walk. Only when it reaches a promoter the initiation complex would be formed, and then transcription starts.

We model the antenna as a stretch of DNA of length L where polymerase can bind anywhere at the same rate B, and take a step forward or backward at the same rate K. If it walks all the way to the left, it falls off DNA, and disappears. If it goes all the way to the right, it initiates, so we increase our count of successful initiations, and the polymerase disappears.

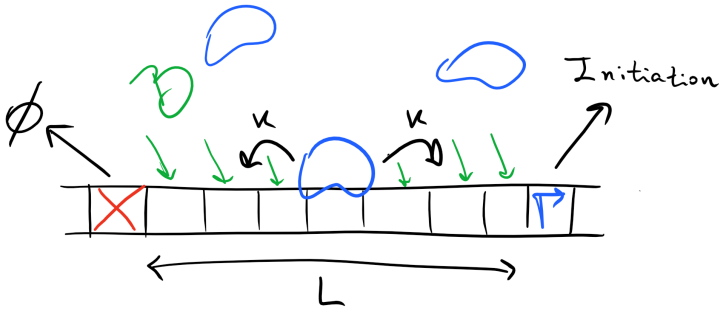

Figure 6: Illustration of the antenna model. Polymerase can bind to any of L loci at rate B, and random walk with rate K until it falls off at the left or initiates at the right.

The big question we can ask about the antenna model then is how would the transcription initiation profiles will differ from when initiation is a single step process? How do the rates of binding and random walking affect initiation rates? How does the length of the antenna affects initiation rates?

We find that instead of a simple exponential distribution, with the antenna model initiation rates look like a gamma distribution. The rate of random walking seems to be more important than the binding rate in dictating mean transcription times.

Interestingly, if the antenna is long and the binding rate is high, most of the antenna is always occupied with polymerases, and the ones farther from the promoter hardly get a chance to reach it - so making the antenna longer won't increase initiation rates by much.

Below we have a few representative cases showing simulator standard output, and some graphs plotting the average interval between initiations as a function of the antenna length.

We examine an antenna of length 40, under different K and B. In addition to the initiation times distribution, note how the occupancy changes and traffic jams occur, with trapped polymerases in the middle.

We also look at the averate time between initiations as a function of the atenna length for cases where $K = 1$ and B is 0.1 and 0.05. For L much greater than 10 there is effectively no change in the average time.

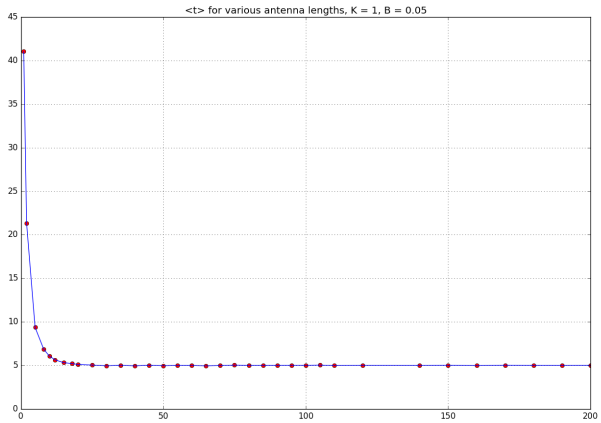

Figure 7: Average initiation interval as a function of antenna length for $K = 1$ and $B = 0.05$

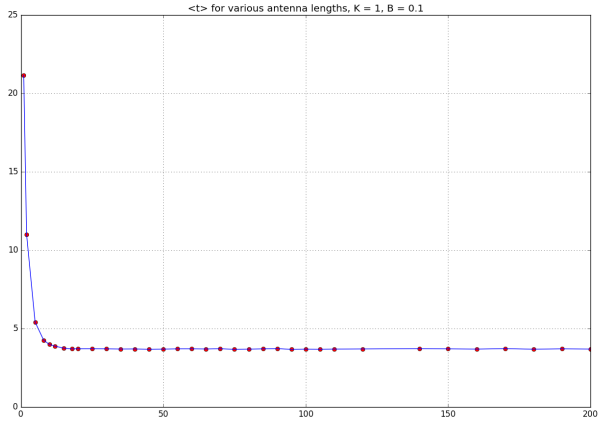

Figure 8: Average initiation interval as a function of antenna length for $K = 0.1$ and $B = 0.1$

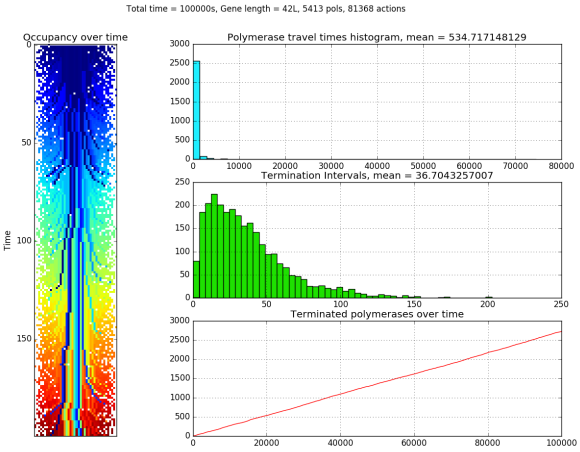

Figure 9: Antenna occupancy and initiation intervals for $L = 40$, $K = 0.1$ and $B = 0.01$

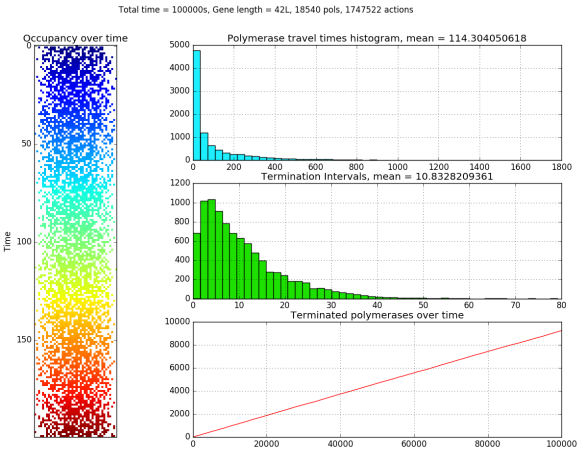

Figure 10: Antenna occupancy and initiation intervals for $L = 40$, $K = 1$ and $B = 0.01$. Because K is small, notice the formation of a traffic jam, with polymerases effectively trapped in the middle and slowly making their way out.

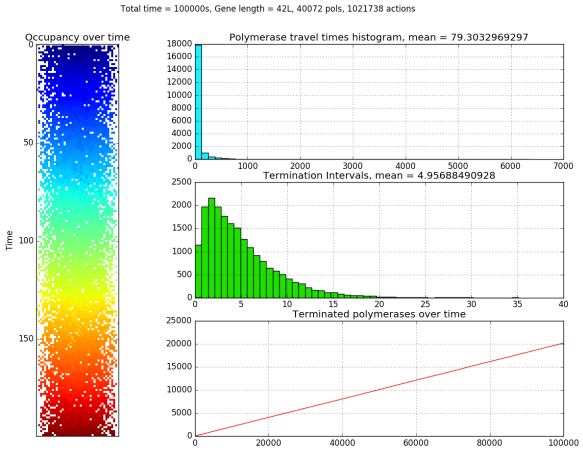

Figure 11: Antenna occupancy and initiation intervals for $L = 40$, $K = 1$ and $B = 0.05$

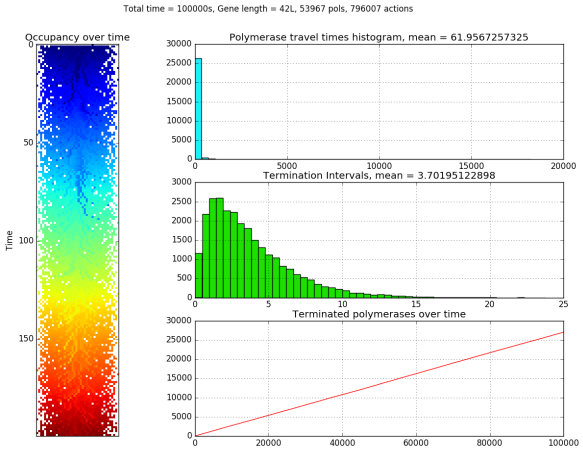

Figure 12: Antenna occupancy and initiation intervals for $L = 40$, $K = 1$ and $B = 0.1$

6 Conclusions

We have developed a modular, customizable and simple to use gene simulator in Python using the Gillespie algorithm. This simulator allows the user to readily test ideas and gather intuitive understanding of how specific gene architectures affect the transcription dynamics, as briefly explored in the examples section. The next steps in development are to implement a graphical user interface, giving the user easier control over the parameters. The simulation code is freely available on GitHub at: https://github.com/Munfred/genesimulator

Bibliography

[1] A. Ay and D. N. Arnosti. Mathematical modeling of gene expression: a guide for the perplexed biologist. *Critical Reviews in Biochemistry and Molecular Biology*, 46.

[2] F. Hayot. Single cell experiments and gillespie's algorithm.

[3] S. Klumpp and T. Hwa. Growth-rate-dependent partitioning of rna polymerases in bacteria.

[4] A. e. a. Sanchez. Effect of promoter architecture on the cell-to-cell variability in gene expression. *PLoS Comput Biol 7.3*, 2011.

[5] R. D. S. A. Selby CP, Drapkin R. Rna polymerase ii stalled at a thymine dimer: footprint and effect on excision repair. *Nucleic Acids Res*, Feb.

Appendix: Gene Simulator Python Script

```
import numpy as np
import matplotlib.pyplot as plt

class Polymerase(object):

    def __init__(self,id):
        self.id = id                    # polymerase ID
        self.startpos = 0               # loci where it binded
        self.starttime = 0              # time when it bound
        self.endpos = 0                 # where it terminated
        self.endtime = 0                # time of termination
        self.aborted = False            # if terminated though abortion, no RNA is
            ↪ producted
        self.times = []                 # times, position and tags matrix [time, pos, tags
            ↪ ]

class Locus (object):
    def __init__(self,k=1.0,r=0.05,bindrate=0.0,terminationrate=0.0,abortionrate
        ↪ =0.0):
        self.occupied = 0                               # if there is a polymerase on the
            ↪ locus, occupied == Polymerase.ID
        self.k = k                                      # locus elongation rate
        self.r = r                                      # locus backtrack rate
        self.bindrate = bindrate                        # locus polymerase binding rate (0
            ↪ if not a promoter)
        self.terminationrate = terminationrate          # locus termination rate (0 if not a
            ↪  terminator)
        self.abortionrate = abortionrate                # locus abortion rate (0 if not a
            ↪ terminator)


# each gelement is declared like this:
# [element lenght, k,r, bindrate, termination rate,abortionrate]

elements = [
            [1, 1, 0, 0.1, 0, 0],       # 1 locus binding element
            [2, 1, 1, 0, 0, 0],         # 2 loci elongation element
            [1, 0.1, 0.2, 0, 0, 0],     # 1 loci pause
            [6, 1, 0.2, 0, 0, 0],       # 6 loci elongatio
            [1,  1, 0.2, 0, 0, 0.1],    # 1 loci abortion
            [2, 1, 1, 0, 0, 0],         # 2 loci elongation
            [1, 0, 0.2, 0, 1, 0],       # 1 loci termination
            ]

gene = []   #gene is a list that is initialized by creating Locus in it by reading
    ↪ the elements list
genematrix = [] #matrix containing the rates for each locus in the gene
for line in elements:
    for i in range(line[0]):
        gene.append(Locus(*line[1:6]))
        genematrix.append(line[1:6])

# necessary element so that I can do gene[i-1] for i =0 and gene [i+1] for gene =
    ↪ length. Must be "occupied" so that no pols try to go there
end = Locus(0,0,0,0,0)
end.occupied = True
gene.append(end)


totaltime = 5000    # total simulation time
pols = [0]          # list to store each of the Polymerase info (they're objects)
length = len(gene)  # gene length
z = 0               # normalizer weight number
t = 0               # current time
dt = 0              # timestep
x = 0               # random number used for timesteps
y = 0               # random number used for picking action
picker = 0          # picker = z*y for picking action
w = 0               # used for counting when picking
polcount = 0        # polymerase global counter
plotstep = totaltime/200        # time interval between plots
plotcounter = 0     # timer counter for the plots
plotmatrix =[]      # stores the values to be plotted
totalsteps = 0      #counts number of iterations of simulation
genestate = [None] * ((length)-1)   # gene state matrix for plotting

### ~~~ Main while loop for Gillespie ~~~ ###

while (t < totaltime):                  #while loop of gillespie algorithm
    z = 0                               #z is the normalizing number (but instead of
        ↪  normalizing i multiply the weights by z because its lets costly)
    x = np.random.random_sample()       #draws random number x for choosing time of
        ↪ event
    y = np.random.random_sample()       #draws random number y for choosing which
        ↪ event to happen
    w = 0                               #w is the accumulator used for picking the
        ↪ event to happen
    totalsteps += 1
    if t > plotcounter*plotstep:        #every plot timestep append the gene state
        ↪ to the genestate matix for plotting
        plotcounter = plotcounter +1
        for p in range(length - 1):
            genestate[p] = gene[p].occupied
        plotmatrix.append(genestate[:])
        print ('Time = ' + str(t))
```

```
    for i in range (length - 1):          #this loop calculates z by summing the rates
        ↪  of all events that can happen
        if gene[i].occupied != 0 and gene[(i+1)].occupied == 0:
            z += gene[i].k
        if gene[i].occupied != 0 and gene[i - 1].occupied == 0:
            z += gene[i].r
        if gene[i].bindrate and gene[i].occupied == 0:
            z += gene[i].bindrate
        if gene[i].terminationrate and gene[i].occupied !=0:
            z += gene[i].terminationrate
        if gene[i].abortionrate and gene[i].occupied != 0:
            z += gene[i].abortionrate

    dt = np.log(1/x)/z          #calculates the timestep size
    t = t + dt                  #increases the time
    picker = z*y                #picker for picking the random number

    for i in range(length - 1): #this for loop keeps going and addings the weight of
        ↪  each event to w until w>picker, then it realizes the event and breaks
        ↪  off

        if gene[i].occupied != 0 and gene[i + 1].occupied == 0: #elongation event
            w += gene[i].k
            if w > picker:
                gene[i+1].occupied = gene[i].occupied   #moves to the location ahead
                gene[i].occupied = 0
                break

        if gene[i].occupied != 0 and gene[i - 1].occupied == 0: #backtrack event
            w += gene[i].r
            if w > picker:
                gene[i-1].occupied = gene[i].occupied   #moves to the location
                    ↪ behind
                gene[i].occupied = 0
                break

        if gene[i].bindrate and gene[i].occupied == 0:          #binding event
            w += gene[i].bindrate
            if w > picker:
                polcount = polcount +1;
                pols.append(Polymerase(polcount))   #creates a new polymerase with
                    ↪ id = polcount
                pols[polcount].starttime = t        #stores in the polymerase what
                    ↪ time it bound
                pols[polcount].startpos = i         #stores in the polymerase where
                    ↪ it bound
                gene[i].occupied = polcount         #puts in the binding location
                    ↪ the id of the polymerase
                break

        if gene[i].terminationrate and gene[i].occupied != 0:   #termination event
            w += gene[i].terminationrate
            if w > picker:
                pols[gene[i].occupied].endtime = t  #stores in the polymerase the
                    ↪ end time
                pols[gene[i].occupied].endpos = i   #stores in the polymerase the
                    ↪ end position
                gene[i].occupied = 0                #frees the location
                break

        if gene[i].abortionrate and gene[i].occupied != 0:      #abortion event
            w += gene[i].abortionrate
            if w > picker:
                #print('loc ' + str(i) + ' aborted')
                gene[i].occupied = 0
                break

########### ~~~ Shenanigans for plotting ~~~ ##########

genematrix = list(map(list, zip(*genematrix)))
cmap = plt.cm.jet
cmap.set_under(color='white')
endtimes = []
for i in range(1,polcount):
    if pols[i].endtime:
        endtimes.append(pols[i].endtime)
endtimes = sorted(endtimes)
intervals = np.diff(endtimes)
poltimes = []
for i in range(1,polcount):
    if pols[i].endtime:
        poltimes.append(pols[i].endtime - pols[i].starttime)

plt.figure(0)
ax1 = plt.subplot2grid((3,3), (0,0),rowspan=3)
occplot = plt.imshow(plotmatrix,cmap=cmap,interpolation='nearest',vmin=0.5)
plt.title('Occupancy_over_time')
plt.ylabel('Time')
ax1.axes.get_xaxis().set_ticks([])

meantravel = np.mean(poltimes)
ax2 = plt.subplot2grid((3,3), (0,1), colspan=2)
plt.hist(poltimes,bins =50, color='#1eefff')
plt.title('Polymerase_travel_times_histogram,_mean_=_' + str(meantravel))
plt.grid(True)

meaninterval = np.mean(intervals)
ax3 = plt.subplot2grid((3,3), (1, 1), colspan=2)
```

```
plt.hist(intervals,bins =50, color='#1ee000')
plt.grid(True)
plt.title('Termination Intervals, mean = ' + str(meaninterval))

ax4 = plt.subplot2grid((3,3), (2, 1), colspan=2)
polrange = range(len(endtimes))
plt.plot(endtimes,polrange, color = 'red')
plt.title('Terminated polymerases over time')
plt.grid(True)

plt.suptitle('Total time = ' + str(totaltime) + 's, Gene length = '+ str(length - 1)
    ↪ + 'L, \
' + str(polcount) + ' pols, ' + str(totalsteps) + ' actions \n')
plt.show()
```