UC Berkeley, CHEM 273, Project II

Simulation of E.coli Biased Random Walk

Brenda Angulo, Alex Chase, Ryan Chin, Chandler Goodwin, Yash Maheshwara, Seungho Yoo

**Abstract**: Survival and proliferation methods of bacteria such as *Escherichia coli* (*E.coli*) are fairly well studied. One such method is chemotaxis, where *E.coli* moves in response to environmental chemical factors.1,2 This simulation of chemotaxis predicts *E.coli* movement towards a food source, where a static concentration gradient radiates from a single origin point in a 2D plane. Movement is initiated by a random “tumble” phase where information about the concentration gradient is collected, followed by a “run” phase where *E.coli* ascends towards higher concentrations based on what it has learned. This process utilizes gradient “ascent” with an AdaGrad3,4 adaptive learning rate to direct *E.coli* towards their target. Several types of concentration gradients were assessed with different *E.coli* population sizes and tumble memory. This simulation model lays the groundwork for more complex models which could simulate 3D movement and introduce additional environmental stimuli.

Contents

[***1. Introduction 3***](#_heading=h.20utw568gxfw)

[***2. Methods 3***](#_heading=h.nu5q4xd2p6jb)

[2.1 Domain and Fields 4](#_heading=h.uly3ijfuomgt)

[2.2](#_heading=h.i7kylivgwiw) Biased Walk with Memory (Gradient “Ascent”) [4](#_heading=h.i7kylivgwiw)

[2.3 Adaptive Learning Rate (AdaGrad) 4](#_heading=h.d2pcmt6zbtdl)

[***3. Implementation***](#_heading=h.jn66av7gvnsp) [**5**](#_heading=h.54ox757tfikn)

[***4. Results 6***](#_heading=h.o7zko5vyjb86)

[***5. Discussion 7***](#_heading=h.hgl22hz4v2ol)

[5.1 Limits of Simulation 7](#_heading=h.9mz0332f07i0)

[5.2 Areas of Improvement 8](#_heading=h.pk5x23kgmdvh)

[***References 9***](#_heading=h.k5l9kdwvlfv)

[***Appendix 10***](#_heading=h.jo57qijgz1l7)

[Appendix I. Python Code **10**](#_heading=h.377rx16bdrxm)

# 

## 1. Introduction

At a microscopic level, mobility of entities such as bacteria is limited due to the inability to sense all but the most immediate environmental factors. In addition, there are other physical limitations to single cell scale movements, such as the strength of flagella used to “swim” through their environment. *E.coli* movement towards a food source can be described as a biased random walk.5 This involves a cycle of one longer run phase preceded by several random tumble phases that serve to collect information about the localized chemical environment which then influences the direction of the run. Environmental chemical factors, such as concentration of a nutrient media or chemicals released in response to a threat are detected by the *E.coli’s* chemoreceptors and influence their behavior.

The simulation described in this work provides a way to visualise biased random walk paths of *E.coli*. Population size andstarting locations, gradient type for the food source and memory of their environment are all parameters which can be adjusted through this model. The simulation is able to assess variations in path as well as time to reach a gradient maximum. For larger populations, the distance from the food source maximum can be visualized to demonstrate how a normal distribution of the population migrates towards a food source despite each individual path being unique. Improvements to this model in the future could provide additional information about movement through three dimensional space and include other environmental stimuli which affect movement.

## 2. Methods

# 2.1 Domain and Fields

*E.coli* are simulated within a square domain of side length L (1.1). Three different static gradient field C(x,y) profiles are available. Calculations for the Gaussian (1.2), linear absolute value (1.3) and paraboloid (1.4) are described as follows.

|  | (1.1) |
| --- | --- |
|  | (1.2) |
|  | (1.3) |
|  | (1.4) |

**2.2 Biased Walk with Memory (Gradient “Ascent”)**

During the simulation, an *E.coli* performs a random tumble (2.1) for a set number of iterations followed by a run (2.5, 2.6) ascending to a higher concentration. During the tumble phase, boundary handling is applied to prevent *E.coli* from exiting the simulation bounds. The partial derivatives with respect to x and y are approximated by a temporal comparison over a memory window (default of 4 tumble steps) using centered finite differences along each axis (2.2, 2.3). The combination of these partial derivatives yields the concentration gradient experienced by the *E.coli* (2.4). The run with respect to x and y are then calculated based on the *E.coli’s* memory of the gradient along with a scalar learning rate (ɛ) value (2.5, 2.6). The learning rate dictates the magnitude of the run. The run is also constrained by boundary conditions. Classic gradient descent directs towards a lower concentration by subtracting the gradient; here we direct towards higher concentration by adding the gradient. Through combining random tumbles followed by finite difference directed runs, the simulation produces a stochastic gradient ascent towards higher food concentrations.

|  | (2.1) |
| --- | --- |
|  | (2.2) |
|  | (2.3) |
|  | (2.4) |
|  | (2.5 |
|  | (2.6) |

**2.3 Adaptive Learning Rate (AdaGrad)**

For each axis, we maintain the cumulative sum of squared gradients (3.1)2 The AdaGrad sum grows as more iterations are performed (3.2). The base rate η = learning\_rate is applied and small ∊ prevents divide by zero errors. This approach attenuates steps as more runs are performed and prevents overshooting the gradient maximum.

|  | (3.1) |
| --- | --- |
|  | (3.2) |

## 3. Implementation

The simulation is built around a single *EColi* class. This class keeps track of various simulation parameters, bacterial positions, and defines how each individual bacterium behaves. The constructor initializes these crucial parameters of the simulation. It stores the number of iterations, grid size, learning rate, memory length, and gradient type. It also creates a 2D spatial grid with *np.meshgrid* for visualizing the gradient and allows for storage of positional information, adaptive learning rates, and distance measurements.

Putting this together, the environment that the *E.coli* will move through is generated by *create\_gradient*, which can create different types of concentration fields such as Gaussian, parabolic, or linear absolute value based on the user specified *grad\_type*. *conc\_calc* matches the *grad\_type* to perform the appropriate calculation.

Movement is modeled using *tumble\_ecoli(n)* and *run\_ecoli(n).* *tumble\_ecoli(n)* performs small, random movements to explore their surroundings. Clipping ensures that movement stays inside the grid. In *run\_ecoli(n)*, the learned gradient from the cumulative finite differences with respect to x and y positions during the tumble phase is applied to move the *E.coli* towards a higher food concentration. The magnitude of movement is then scaled by the adaptive learning rate so that the bacterium can move towards the chemically relevant substance.

The bacterium's positions relative to the concentration gradient are visualized through *plot\_routine\_track.* Once all iterations of the simulation are performed, *calculate\_distance* is used to measure each bacterium’s distance from the food source and *plot\_routine\_source\_distance* shows a histogram of these distances. All aspects of the bacterium movement is controlled through the use of the *ecoli\_simulation* function, which builds the environment, performs the tumble and run cycles, records the data points, and produces the plots. The *run\_test\_cases* function groups together relevant test cases which ensure the simulation is running as intended.

## 4. Results

The simulation can model paths of a large number of *E.coli* as they move toward one specified gradient maximum. An additional subplot of the *E.coli* learning rate is plotted against time to illustrate the adaptive nature of this rate. This plot has been clipped to the first 25 iterations to highlight the decrease in rate at the start, before a lower limit is reached.

Movement along a gaussian (fig. 1a), parabolic (fig. 1b) and linear absolute value (fig. 1c) gradient shows that as the *E.coli* approach the maximum that the learning rate diminishes and the run magnitude decreases. Simulating multiple *E.coli* shows that despite each random path being different, each *E.coli* is able to locate the gradient maximum after enough iterations (fig. 1d). In each case upon reaching the gradient maximum, *E.coli* randomly tumble and run around the gradient maximum for the remaining iterations. The run magnitude is very limited at this point due to the adaptive learning rate.

Several different *E.coli* memory lengths were also assessed for their impact on finding the gradient maximum (fig 2). Interestingly, shorter memory has little impact on how quickly the *E.coli* find their target in this simulation. The run magnitude is directly proportional to the partial derivatives with respect to the x and y position of the *E.coli*. One would expect that if more tumble steps were taken into account, a larger change in concentration would be observed and a larger run would occur. In reality, when very far from the food source, the change in slope is quite small and additional tumble steps do little to affect these values. As an *E.coli* approaches the maximum, the change in slope becomes much larger; however, at this point the adaptive learning rate has greatly reduced the magnitude of the run step despite the larger change. For this reason, very similar run magnitudes are observed regardless of memory. It would be expected that if the learning rate was static that longer memory would lead to much longer run magnitudes near the gradient maximum. With this simulation, the most efficient path to the source is actually with a memory of one tumble step (fig. 2d) where a nearly straight path to the source is observed, along with less random variation around the maximum once the source is found.

Lastly, a large population of 1000 *E.coli* was simulated. The distance of each *E.coli* from the food source was plotted as a histogram that shows the population distribution moving towards the gradient maximum over the course of the simulation. Two simulations were performed where all *E.coli* were seeded at a single origin point (fig. 3a) and at random points in the lower left quadrant of the simulation bounds (fig. 3b). As expected, the random seeds show a more broadly distributed population compared to the single origin point. In both cases by the 1000th iteration, a narrow population distribution is observed close to the food source. This result confirms that regardless of seeding location, given enough time a large *E.coli* population will all reach the gradient maximum.

| **a** | **b** |
| --- | --- |
| **c** | **d** |
| **Figure 1: Biased random walk of E.coli along different gradient types.** The biased random walk of a single *E.coli* is shown for three different gradient types, gaussian (1a), parabolic (1b) and linear absolute value (1c). In each case, the learning rate over time is plotted to show that this rate is adaptive and decreases as the *E.coli* reaches the gradient maximum. Upon reaching the maximum, the *E.coli* randomly tumbles around the center point for the remaining iterations. An additional walk is shown for 10 *E.coli* (1d) which shows that different random paths will still ascend towards the gradient maximum over enough iterations. | |

| **a** | **b** |
| --- | --- |
| **c** | **d** |
| **Figure 2: Biased random walk of E.coli with varying memory.** The biased random walk of a single *E.coli* is shown with a tumble step memory of four steps (2a), three steps (2b), two steps (2c) and one step (2d). In all cases, due to the adaptive learning rate the run magnitudes are very similar despite a shorter or longer memory. For a short memory, the path to the center is more direct and once at the maximum the *E.coli* stays more centered around the maximum because fewer random tumble steps are performed before each run. | |

| **a** | **b** |
| --- | --- |
| **Figure 3: Distance from food source for a large E.coli population.** Two different simulations of 1000 *E.coli* were run, one where all *E.coli* are seeded at the same origin point (3a) and the other at random points in the lower left quadrant of the simulation bounds (3b). When seeded together the distribution of *E.coli* is more narrow than when seeded randomly. At the 1000th iteration, both histograms show a similar distribution around the food source indicating that starting location ultimately has no impact on finding the gradient maximum. | |

## 5. Discussion

# 5.1 Limits of Simulation

This simulation captures key aspects of *E.coli* chemotaxis through biased random walks, however, it simplifies many of the biological and environmental complexities seen in the real world. In the real world, bacterial populations exhibit behaviors such as quorum sensing and biofilm formation, altering their movement patterns and dynamics.6

Additionally, the chemical gradients used in this simulation are static. In reality, concentration gradients diffuse outward over time. Furthermore, food would be consumed which would impact concentration and subsequent diffusion. *E.coli* would also interact with one another which would affect their movement during the tumble and run phase of the simulation. The current simulation model fails to account for these interactions which should be addressed in future models.

Individual *E.coli* are assigned identical parameters in this simulation. Each has the same memory, learning rate, and responsiveness. Naturally, this rejects natural genetic variability. Different gene expression, mutations, and natural factors would lead to differences in their movement patterns. In addition to this, the fixed timestep and discrete memory introduce an artificial norm that does not fully reflect the decision-making processes bacteria employ in natural environments. For example, real *E.coli* have movement patterns that are more fluid and their response to gradients and environmental signals is continuous and biologically dynamic, not segmented and static.

# 5.2 Areas of Improvement

As discussed there are currently several limitations to this simulation model and improvements could be made to make it more biologically representative. To tackle the static concentration gradients, Fick’s Second Law of Diffusion equation could be introduced to allow for the gradient to dynamically change as the bacteria moves and potentially consumes the attractant.7 Another possible improvement would be to introduce a repulsion vector to try and simulate individual bacteria interactions. A repulsion vector analogous to the short-range repulsion in the Lennard-Jones potential can stimulate bacterial movement when inter-bacterial distance falls below a certain threshold.8

To better capture natural variation in behavioral patterns, we could modify our model to randomize parameters such as memory and learning rate. This would simulate genetic variation, giving a more real world representation of *E.coli* fitness.

AdaGrad was used to dynamically adjust the learning rate of *E.coli*. It is a straightforward and effective approach used in a variety of applications like machine learning.3 However, there are many other adaptive learning rate approaches available such as RMSProp.9 RMSProp uses the exponentially decaying average of squared gradients per parameter. This approach would prevent the learning rate from shrinking too much and allow the bacteria to continue to adjust their learning rates based on recent gradients, even after a large number of iterations.

In conclusion, this simulation serves as a foundational model for illustrating the basic concepts that govern *E.coli’s* random biased walk. It demonstrates well for us the stochasticity of our subjects’ tumble phases, seen in the gaussian curve that emerges in the histogram plot of *E.coli* distances to the gradient center. Future iterations should look to build upon this rudimentary model by introducing variables that bring the behavior closer to what we see in a natural environment. This simulation could then be used for predictive purposes when considering *E.coli* behavior within various environments, including but not limited to human or animal facing systems.

# References

1. Erban, R., & Othmer, H. G. (2004). From individual to collective behavior in bacterial

chemotaxis. *SIAM J. Appl. Math.*, **65**, 361–391.

1. Sourjik, V., & Wingreen, N. S. (2012). Responding to chemical gradients: bacterial chemotaxis. *Curr. Opin. Cell Biol.*, **24**, 262–268.
2. “Adagrad.” AdaGrad - Cornell University Computational Optimization Open Textbook - Optimization Wiki, 14 Dec. 2021, [optimization.cbe.cornell.edu/index.php?title=AdaGrad](http://optimization.cbe.cornell.edu/index.php?title=AdaGrad).
3. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

*John Duchi, Elad Hazan, Yoram Singer*; 12(61):2121−2159, 2011.

1. Berg, Howard C. *E. Coli in Motion*. Springer, 2011.
2. Preda, Veronica G, and Oana Săndulescu. “Communication Is the Key: Biofilms, Quorum Sensing, Formation and Prevention.” *Discoveries (Craiova, Romania)*, U.S. National Library of Medicine, 30 Sept. 2019, pmc.ncbi.nlm.nih.gov/articles/PMC7086079/.
3. Mehrer, Helmut, and Nicolaas A. Stolwijk. “Heroes and highlights in the history of Diffusion.” *Diffusion Fundamentals*, vol. 11, 31 Dec. 2009, https://doi.org/10.62721/diffusion-fundamentals.11.453.
4. P. Ghosh, J. Mondal, E. Ben-Jacob, & H. Levine, Mechanically-driven phase separation in a growing bacterial colony, Proc. Natl. Acad. Sci. U.S.A. 112 (17) E2166-E2173, https://doi.org/10.1073/pnas.1504948112 (2015).
5. “RMSPROP.” *RMSProp - Cornell University Computational Optimization Open Textbook - Optimization Wiki*, optimization.cbe.cornell.edu/index.php?title=RMSProp. Accessed 14 Aug. 2025.

# 

# Appendix

## Appendix I. Python Code

import numpy as np

import matplotlib.pyplot as plt

class EColi:

def \_\_init\_\_(self, N, num\_ecoli=1, box\_length=20, grid\_res = 500, tumble\_step=0.3,

learning\_rate=1.5, memory = 4, grad\_type='gaussian', origin\_type='random'):

#gradient descent parameters

self.N = N

self.tumble\_step = tumble\_step

self.learning\_rate = learning\_rate

self.memory = memory

self.grad\_type = grad\_type

#setup for grid

self.box\_length = box\_length

self.bounds = box\_length/2

self.grid\_res = grid\_res

self.grid = np.linspace(-self.bounds, self.bounds, grid\_res)

#setup for E.coli seeds

self.num\_ecoli = num\_ecoli

self.X = np.zeros((num\_ecoli,N))

self.Y = np.zeros((num\_ecoli,N))

self.origin\_type = origin\_type

if self.origin\_type == 'random':

self.X[:,0] = np.random.uniform(-self.bounds, 0, (num\_ecoli,))

self.Y[:,0] = np.random.uniform(-self.bounds, 0, (num\_ecoli,))

if self.origin\_type == 'together':

self.X[:,0] = -self.bounds\*0.8

self.Y[:,0] = -self.bounds\*0.8

#setup for seeding gradient

self.X\_grad, self.Y\_grad = np.meshgrid(self.grid, self.grid)

self.grad\_xcenter = self.bounds\*0.8

self.grad\_ycenter = self.bounds\*0.8

self.grad = np.zeros((len(self.grid),len(self.grid)))

#Sum of squared gradients for adaptive learning rate (adagrad)

self.Gt\_x = np.zeros((self.num\_ecoli,self.N))

self.Gt\_y = np.zeros((self.num\_ecoli,self.N))

self.adapt\_lr\_x = np.zeros((self.num\_ecoli,self.N))

self.adapt\_lr\_y = np.zeros((self.num\_ecoli,self.N))

#Distance index for histogram

self.I = [1, 10, 50, 100, 1000]

self.distance = np.zeros((self.num\_ecoli,len(self.I)))

#params for different gradient functions

self.gaus\_sigma = 10

self.lin\_slope = 0.5

self.lin\_height = 10

self.parab\_width = 1

self.parab\_height = 10

def tumble\_ecoli(self, n):

dx = np.random.uniform(-self.tumble\_step,self.tumble\_step,(1, self.num\_ecoli))

dy = np.random.uniform(-self.tumble\_step,self.tumble\_step,(1, self.num\_ecoli))

self.X[:,n+1] = np.clip((self.X[:,n] + dx), a\_min=-self.bounds, a\_max=self.bounds)

self.Y[:,n+1] = np.clip((self.Y[:,n] + dy), a\_min=-self.bounds, a\_max=self.bounds)

def run\_ecoli(self,n):

delta\_x = self.X[:,n] - self.X[:,n-self.memory]

delta\_y = self.Y[:,n] - self.Y[:,n-self.memory]

# these are the finite differences used for calculating the derivatives

# if forward or backward delta is out of bounds then clip

x\_fwd = np.clip((self.X[:,n] + delta\_x), a\_min=-self.bounds, a\_max=self.bounds)

x\_bwd = np.clip((self.X[:,n] - delta\_x), a\_min=-self.bounds, a\_max=self.bounds)

y\_fwd = np.clip((self.Y[:,n] + delta\_y), a\_min=-self.bounds, a\_max=self.bounds)

y\_bwd = np.clip((self.Y[:,n] - delta\_y), a\_min=-self.bounds, a\_max=self.bounds)

# calculate concentration from grad for finite differences

# specific conc\_calc depends on gradient type specified by user

x\_conc\_fwd = self.conc\_calc(x\_fwd, self.grad\_xcenter)

x\_conc\_bwd = self.conc\_calc(x\_bwd, self.grad\_xcenter)

y\_conc\_fwd = self.conc\_calc(y\_fwd, self.grad\_ycenter)

y\_conc\_bwd = self.conc\_calc(y\_bwd, self.grad\_ycenter)

x\_deriv = np.zeros((self.num\_ecoli,))

y\_deriv = np.zeros((self.num\_ecoli,))

# calculate partial derivatives by finite differences

# bool mask to prevent divide by zero error if delta is zero

x\_mask = (delta\_x != 0)

x\_deriv[x\_mask] = (x\_conc\_fwd[x\_mask] - x\_conc\_bwd[x\_mask]) / (2 \* delta\_x[x\_mask])

y\_mask = (delta\_y != 0)

y\_deriv[y\_mask] = (y\_conc\_fwd[y\_mask] - y\_conc\_bwd[y\_mask]) / (2 \* delta\_y[y\_mask])

# adaptive gradient (adagrad) approach for learning rate

self.Gt\_x[:,n] = x\_deriv\*\*2

self.Gt\_y[:,n] = y\_deriv\*\*2

# this is the actual adagrad equation

# the derivative for each previous run step is added to the sum

# this causes the learning rate to reduce over time as the sum of derivatives grows

self.adapt\_lr\_x[:,n] = self.learning\_rate / np.sqrt((np.sum(self.Gt\_x, axis=1)) +

np.exp(-12))

self.adapt\_lr\_y[:,n] = self.learning\_rate / np.sqrt((np.sum(self.Gt\_y, axis=1)) +

np.exp(-12))

x\_run\_exp = self.adapt\_lr\_x[:,n] \* x\_deriv

y\_run\_exp = self.adapt\_lr\_y[:,n] \* y\_deriv

# update the next iteration with new position after gradient "ascent"

self.X[:,n+1] = np.clip((self.X[:,n] + x\_run\_exp), a\_min=-self.bounds, a\_max=self.bounds)

self.Y[:,n+1] = np.clip((self.Y[:,n] + y\_run\_exp), a\_min=-self.bounds, a\_max=self.bounds)

# calculates the distance from source for each E.coli for generating histograms

def calculate\_distance(self):

dist\_i = 0

for i in self.I:

self.distance[:,dist\_i] = np.sqrt((self.grad\_xcenter - self.X[:,i-1])\*\*2 + \

(self.grad\_ycenter - self.Y[:,i-1])\*\*2)

dist\_i += 1

# define each gradient calculation

# ecoli and gradient generation call on conc\_calc which depending on gradient type

# applies the correct function

def gaussian\_grad(self, value, mu):

return (1 / self.gaus\_sigma \* np.sqrt(2 \* np.pi)) \* np.exp(-0.5 \* ((value -

mu)/(self.gaus\_sigma))\*\*2)

def linear\_grad(self, value, loc):

return -self.lin\_slope\*np.abs(value - loc) + self.lin\_height

def parabola\_grad(self, value, loc):

return -((value-loc)\*\*2 / self.parab\_width\*\*2) + self.parab\_height

def conc\_calc(self, val, grad\_center):

match self.grad\_type:

case "gaussian":

result = self.gaussian\_grad(val, grad\_center)

case "linear":

result = self.linear\_grad(val, grad\_center)

case "parabola":

result = self.parabola\_grad(val, grad\_center)

return result

def create\_gradient(self):

#match case is like c++ switch, this is how the correct gradient is calculated

match self.grad\_type:

case "gaussian":

self.grad = (self.conc\_calc(self.X\_grad, self.grad\_xcenter)) \* \

(self.conc\_calc(self.Y\_grad, self.grad\_ycenter))

case "linear":

self.grad = (self.conc\_calc(self.X\_grad, self.grad\_xcenter)) +\

(self.conc\_calc(self.Y\_grad, self.grad\_ycenter))

case "parabola":

self.grad = (self.conc\_calc(self.X\_grad, self.grad\_xcenter)) + \

(self.conc\_calc(self.Y\_grad, self.grad\_ycenter))

def plot\_routine\_track(self):

fig, ax = plt.subplot\_mosaic([['A'],['B']], layout="constrained", figsize=(8,8),

gridspec\_kw={'height\_ratios': [7, 1]})

ax['A'].contourf(self.X\_grad, self.Y\_grad, self.grad, levels=5, cmap='grey\_r')

for n in range(self.num\_ecoli):

ax['A'].plot(self.X[n,:],self.Y[n,:], marker= 'o',

markerfacecolor='none', markeredgecolor='white', markersize=4,

linestyle='-', color='white', alpha =0.5)

for n in range(self.num\_ecoli):

ax['A'].scatter(self.X[n,0], self.Y[n,0], marker= 'x',

s=100, color='blue')

ax['A'].scatter(self.X[n,-1], self.Y[n,-1], marker= 'x',

s=100, color='red')

# mask for iterations where an adaptive learning rate was calculated

# determine average lr for x and y and then normalize

lr\_x\_mask = (self.adapt\_lr\_x != 0)

lr\_y\_mask = (self.adapt\_lr\_y != 0)

average\_lr = (self.adapt\_lr\_x[lr\_x\_mask] + self.adapt\_lr\_y[lr\_y\_mask]) / 2

num\_lr = np.arange(0, len(average\_lr), 1)

min\_lr = np.min(average\_lr)

max\_lr = np.max(average\_lr)

norm\_lr = (average\_lr - min\_lr) / (max\_lr - min\_lr)

ax['A'].set\_xlim([-self.bounds,self.bounds])

ax['A'].set\_ylim([-self.bounds,self.bounds])

ax['A'].set\_title(f"Biased Random Walk\nE.coli = {self.num\_ecoli}, iterations = {self.N},

gradient type = {self.grad\_type}, memory={self.memory}")

ax['A'].set\_xlabel("x position")

ax['A'].set\_ylabel("y position")

# plots the adaptive learning rate over time

# only plots up to 25 iterations because after that there is little change

ax['B'].plot(num\_lr, norm\_lr, color='black')

ax['B'].set\_xlim([0, 25])

ax['B'].set\_ylim([0, None])

ax['B'].tick\_params(axis='x', bottom=False, labelbottom=False)

ax['B'].set\_xlabel("time")

ax['B'].set\_ylabel("Learning Rate")

plt.show()

def plot\_routine\_source\_distance(self):

# each plot will show the same kind of histogram at a different iteration

# create a dict to hold the layout key and number of iterations for more

# concise histogram plotting

layout = ['A', 'B', 'C', 'D', 'E']

fig\_dict = dict(zip(self.I, layout))

fig, ax = plt.subplot\_mosaic([['A'],['B'],['C'],['D'],['E']],

layout="constrained", figsize=(8,12))

dist\_i = 0

for i in self.I:

ax[fig\_dict[i]].hist(self.distance[:,dist\_i], bins=200, range=(0,

np.max(self.distance)), color='black')

ax[fig\_dict[i]].text(0.05,0.85,f"I = {i}", fontsize = 14,

transform=ax[fig\_dict[i]].transAxes)

dist\_i += 1

ax['A'].set\_title(f"Simulation of {self.num\_ecoli} E.coli\nGradient = {self.grad\_type},

Origin = {self.origin\_type}", fontsize=20)

ax['C'].set\_ylabel("number of E.coli", fontsize=16)

ax['E'].set\_xlabel("distance from the source", fontsize=16)

# This is the primary simulation function

# generates an E.coli class with specified parameters and performs specified plotting

# returns the class type for post simulation review if desired

def ecoli\_simulation(num\_ecoli, N=1000, mem = 4, origin='random', grad="gaussian",

track\_plot=False, dist\_plot=False):

"""

Parameters

----------

num\_ecoli : int, required

Number of E.coli to seed.

N : int, optional

Set number of iterations. The default is 1000.

origin : str, optional

E.coli seeding ('random', 'together'). The default is 'random'.

grad : str, optional

Gradient type ('gaussian', 'linear', 'parabola'). The default is "gaussian".

track\_plot : bool, optional

Run tracking plot routine. The default is False.

dist\_plot : bool, optional

Run distance from source plot routine. The default is False.

"""

# construct E.coli class for simulation and generate gradient

ecoli = EColi(N, num\_ecoli, memory=mem, grad\_type=grad, origin\_type=origin)

ecoli.create\_gradient()

# perform N number of iterations and run depending on memory of E.coli

for n in range(N-1):

if (n+1) % (ecoli.memory + 1) ==0:

ecoli.run\_ecoli(n)

else:

ecoli.tumble\_ecoli(n)

# after simulation is complete calculate the distances from the source for

# each iteration for histogram plot

ecoli.calculate\_distance()

# perform plotting depending on what user has specified

if track\_plot:

ecoli.plot\_routine\_track()

if dist\_plot:

ecoli.plot\_routine\_source\_distance()

return ecoli

# enter test group number to run test (1-4)

def run\_test\_cases(test):

match test:

##### TEST 1 ##################################################################

# testing our three different gradients, 1 E.coli each

# 'together' origin starts E.coli at the same fixed point

# generating 2d tracking plots

case 1:

t1\_1 = ecoli\_simulation(num\_ecoli=1, grad='gaussian', origin='together',

track\_plot=True)

t1\_2 = ecoli\_simulation(num\_ecoli=1, grad='parabola', origin='together',

track\_plot=True)

t1\_3 = ecoli\_simulation(num\_ecoli=1, grad='linear', origin='together', track\_plot=True)

##### TEST 2 ##################################################################

# testing 10 E.coli generated on guassian gradient

# comparing starting together vs random locations in lower left quadrant

# generating 2d tracking plots

case 2:

t2\_1 = ecoli\_simulation(num\_ecoli=10, grad='gaussian', origin='together',

track\_plot=True)

t2\_2 = ecoli\_simulation(num\_ecoli=10, grad='gaussian', origin='random',

track\_plot=True)

##### TEST 3 ##################################################################

# testing location from source for 1,000 E.coli

# comparting starting together vs random locations in lower left quadrant

# comparing each gradient type

# generating histograms at specific iterations

case 3:

t3\_1 = ecoli\_simulation(num\_ecoli=1000, grad='gaussian', origin='together',

dist\_plot=True)

t3\_2 = ecoli\_simulation(num\_ecoli=1000, grad='gaussian', origin='random',

dist\_plot=True)

t3\_3 = ecoli\_simulation(num\_ecoli=1000, grad='parabola', origin='together',

dist\_plot=True)

t3\_4 = ecoli\_simulation(num\_ecoli=1000, grad='linear', origin='together',

dist\_plot=True)

##### TEST 4 ##################################################################

# testing impact of memory on E.coli

case 4:

t4\_1 = ecoli\_simulation(num\_ecoli=1, mem=3, grad='gaussian', origin='together',

track\_plot=True)

t4\_2 = ecoli\_simulation(num\_ecoli=1, mem=2, grad='gaussian', origin='together',

track\_plot=True)

t4\_3 = ecoli\_simulation(num\_ecoli=1, mem=1, grad='gaussian', origin='together',

track\_plot=True)

case \_:

print("Not a valid test group")