A NUMERICAL METHOD FOR COMPUTING THE MEAN FIRST PASSAGE TIME OF A RANDOM WALKER ON A LATTICE

EVAN S. ALLHANDS, UNMEELAN CHAKRABARTI, & FRANKLIN SMITH

Contents

Date: November 19th, 2024.

1. Introduction.

Consider a random walker on a lattice with stochastic resetting which places the walker back at the initial position according to a specified resetting rate. If we suppose a high resetting rate such that the walker frequently returns to the initial poistion, and the initial position is sufficiently far from a predefined target, then it is possible that the walker may never hit the target. Similarly, if there is no resetting, and the lattice is sufficiently large, then the walker may never encounter the target. It is, therefore, reasonable to question the interaction between the size of the lattice and the resetting rate. To answer such a question, we will require a thorough understanding of the probability density function governing the walker's position on the lattice over time. Then, by integrating the probability density function, we obtain the cumulative distribution function which yields the expected time to hit the desired target based on the specific parameters. Furthermore, we develop a method for obtaining the optimal resetting rate for specific lattice sizes such that the expected passage time is minimized. These results are compared with the results obtained from the known backward equation which gives the expected passage time for various starting positions. We will first introduce the relevant notation before developing the numerical method.

Using resetting rates and determination of expected passage time has been used in various forms of research focusing on diffusion processes in constrained geometries, random walks, and Brownian motion [Redner, 2001]. Reset mechanisms maximize first passage time in communication networks, enzymatic reactions, and search problems [Blasius and Tönjes, 2009] ;[Evans and Majumdar, 2011]. Gene regulation, protein folding, and latency in complex networks are among the applications.

1.1. Notation.

For our specific purpose, we will suppose that the random walker only walks on the lattice $(-N, N)$, can only "exit" the lattice at $j = N$ such that if the walker is at $j = -N$, it either must stay at $j = -N$ or move to $J = -N+1$. Thus, our target will always be $N + 1$. We designate the initial position as j_0. We will let $p_j(t)$ be the probability that the walker is at position $j \in (-N, N)$ for given time t. We may suppose that the probability density satisfies

$$\frac{d\mathbf{p}}{dt} = \mathbf{A}\mathbf{p} + \gamma\mathbf{U}\mathbf{p} \tag{1}$$

where

$$\mathbf{p}(t) = [p_{-N}(t),\ p_{-N+1}(t), \cdots, p_0(t), \cdots, p_{N-1}(t), p_N(t)]^\top. \tag{2}$$

The matrix $[A]_{i,j}$ is a transition matrix defined as

$$[A]_{i,j} = \begin{cases} (-1+\gamma)\delta_{i,j} + 0.5\delta_{i-1,j} + 0.5\delta_{i+1,j} & \text{for } -N < i < N, \\ -(.5+\gamma)\delta_{-N,j} + 0.5\delta_{-N+1,j} & \text{for } i = -N \end{cases}. \tag{3}$$

Furthermore, we let

$$[U]_{i,j} = \begin{cases} 1 & \text{for } i = j_0, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

We let τ represent the mean passing time for the walker to hit the specified target.

By inspection, we observe that $\mathbf{A}$ is tridiagonal and has the form

$$\mathbf{A} = \begin{bmatrix} -(\frac{1}{2}+\gamma) & \frac{1}{2} & & \\ \frac{1}{2} & -(1+\gamma) & \frac{1}{2} & \\ & \ddots & \ddots & \ddots \\ & & \frac{1}{2} & -(1+\gamma) \end{bmatrix}, \tag{5}$$

while $\mathbf{U}$ is a sparse matrix containing ones in the row which corresponds to the initial position of the walker.

We will let $\mathbf{M} = \mathbf{A} + \gamma\mathbf{U}$ such that the system of differential equations in (1) can be written more succinctly as

$$\frac{d\mathbf{p}}{dt} = \mathbf{M}\mathbf{p}. \tag{6}$$

2. Theory.

Solving equation 6 will provide a probability function solution for different point s along the lattice. This is done using ODE45 algorithm in MATLAB. This is common method used to numerically solve ordinary differential equations of the form:

$$\frac{d\mathbf{p}}{dt} = f(\mathbf{p}),$$

with the initial condition:

$$\mathbf{p}(t_0) = p_0.$$

The solver employs an adaptive Runge-Kutta method of orders 4 and 5 to approximate the solution over a specified time span. At each step, the next value p_{j+1} is calculated as:

$$p_{j+1} = p_j + h\Phi(t_j, p_j, h),$$

where Φ is a weighted combination of intermediate evaluations of f($\mathbf{p}$) at multiple points within the step size h.

The error is estimated by comparing the 4th and 5th order solutions, and the step size h is dynamically adjusted to maintain the desired accuracy. This approach ensures efficient and reliable results for non-stiff and moderately stiff systems.

In order to find the expected passage time (τ) of the walker to hit $j = N + 1$, we compute the following integral

$$\tau = \int_0^\infty t \frac{dp_{N+1}}{dt}\, dt. \tag{7}$$

The evolution of the probability density at p_{N+1} is given by

$$\frac{dp_{N+1}}{dt} = 0.5 p_N \tag{8}$$

since the probability the walker is at p_{N+1} is determined entirely by whether or not the walker is currently at p_N at time t. Thus, we can modify the integral above as follows

$$\int_0^\infty t \frac{dp_{N+1}}{dt}\, dt = \frac{1}{2} \int_0^\infty t p_N(t)\, dt. \tag{9}$$

For any given lattice size (N), the value of γ that enables the lowest time to pass N can be considered as the optimal resetting rate. A set of values of γ between [0,1] are input to yield a respective set of M in equation 6. The ODE45 solver explained earlier is used to obtain the solution set for p_N at each input γ. Each of the p_N solutions are then used to obtain a value of τ from equation 9. The local minima for τ in the range of input γ values provide the minimum passage time for a selected lattice size (N). The corresponding $\gamma = \gamma*$ at this minimal passage time provides the optimal resetting rate for the system.

To validate the calculation of $\gamma*$, we use the backward equation [Bressloff, 2014] as a benchmark, which is given by

$$-\mathbf{1} = \mathbf{M}^\top \tau. \tag{10}$$

Solving the linear system (10) yields a vector, τ, whose components τ_k is the expected passage time to hit the target for initial position j_k. The obtained $\gamma*$ from minimizing equation 9 is used for $\mathbf{M}$ in equation 10 to get a solution for τ. This new computed τ is compared to the minimized τ from equation 9 to validate the current methodology.

3. Numerical Method.

The numerical method developed below leverages 4th and 5th order Runge-Kutta methods to solve the system of differential equations before using numerical quadrature to approximate the integral given in (9). The optimal resetting rate, γ^*, is computed using a γ-discretization method and is compared against results from local search optimization algorithm. The computed passage times are then compared with the results from the backward equation (10).

3.1. **Solving the system.** Let γ and N be fixed, and let the initial position $j_0 = 0$ such that the random walk will begin at the center of the lattice. Using the ODE45 implementation in MATLAB, we define our system of differential functions as given in (6), and pass to the ODE45 solver. ODE45 uses an adaptive method to compute the time-steps based upon the dynamics of the system, resulting in better control over error propagation. Euler methods and matrix exponentiation can also be used to solve the system of differential equations.
After solving for $\mathbf{p}(t)$, we obtain probability density curves for the probability that the walker is at position j for time t, as can be seen in the plots below. We observe

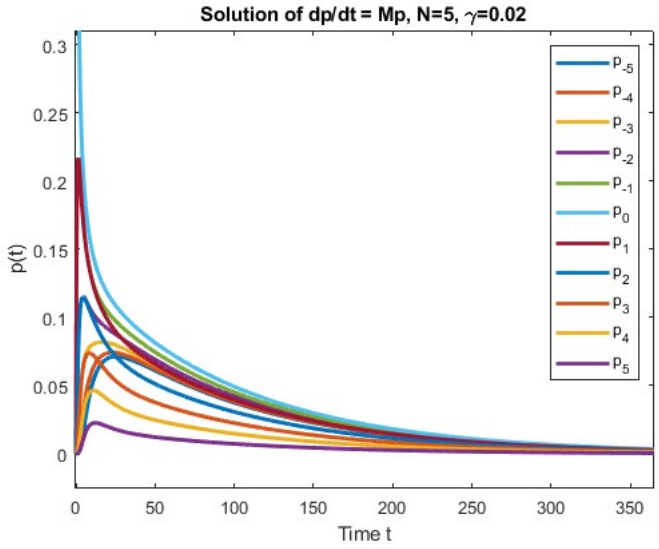

FIGURE 1. $N = 5,\ \gamma = 0.02$

that the probabilities decay as time increases, which indicates that after some time T, the walker has exited the lattice at $j = N$ almost surely.

3.2. **Obtaining the expected time.** Recall that the expected time, τ, is found by computing the integral in (9). To approximate the improper integral, we must compute a stopping T such that

$$\int_0^\infty \frac{1}{2} t p_N(t)\ dt \approx \int_0^T \frac{1}{2} t p_N(t)\ dt. \tag{11}$$

Let θ be the unit round. Then, for $\varepsilon > \theta$, there exists $T \in (0, \infty)$, such that

$$\sum_{j=-N}^{N} p_j(T) < \varepsilon. \tag{12}$$

To approximate the max stopping time numerically, we fix γ, N, and a desired tolerance level, ε. After solving the system as in the previous section, we determine

if (12) is satisfied. If it is not, we increase the maximum time by a pre-determined amount and repeat the steps. If (12) is satisfied, then we take T to be as in (12).

With the ability to truncate the time-interval, we are able to approximate the integral in (11) using Trapezoidal and Simpson's Method. The result of this approximation yields the expected passage time for the walker to reach the target for given resetting rate and lattice size.

3.3. **Computing optimal resetting rate.** To compute the optimal resetting rate, γ^*, a low-fidelity γ-discretization method is deployed. We let $\vec{\gamma}$ be a vector of M evenly-spaced values between 0 and 1. By iteratively solving for the expected passage time for each $\gamma_k \in \vec{\gamma}$, we can then find the minimum of $\vec{\gamma}$. Then, $\gamma^* = \gamma_k$ where k is the index of the minimum of $\vec{\gamma}$, γ_k.

The γ-discretization method–though accurate for sufficiently large M–is incredibly inefficient and computationally expensive. A much faster method is to define an objective function for the computation of τ based on γ and utilize a search algorithm, such as fminbnd() in MATLAB. Comparison of results is available in section 4 of this manuscript.

3.4. **Comparison with backward equation.** The backward equation presented in (10) is dependent only upon on γ and N. Solving the linear system can be accomplished easily in MATLAB by using the linear solve operator, \. We remark that the result of solving this linear system yields a vector of expected passage times corresponding to each initial position–so, we must compare our results by integration to the component of τ that corresponds to our initial position, j_0.

4. Results.

4.1. **N=3.** We let $N = 3$ and consider a range of γ values to explore the effect of resetting on the probability density functions.

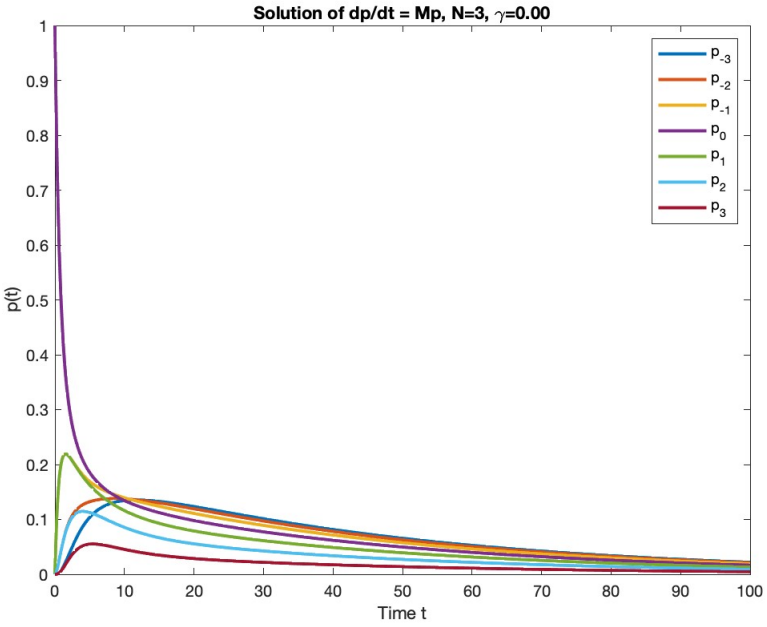

FIGURE 2. $N = 3,\ \gamma = 0.001$

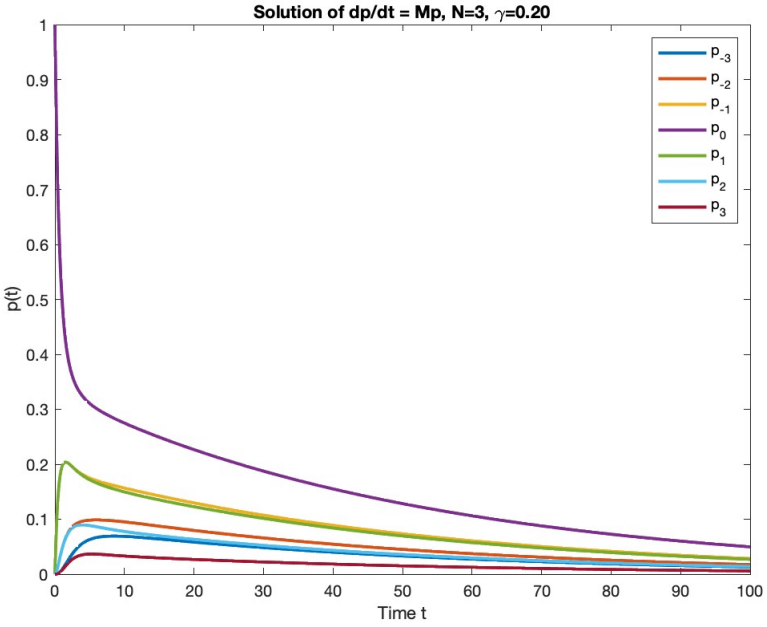

FIGURE 3. $N = 3,\ \gamma = 0.2$

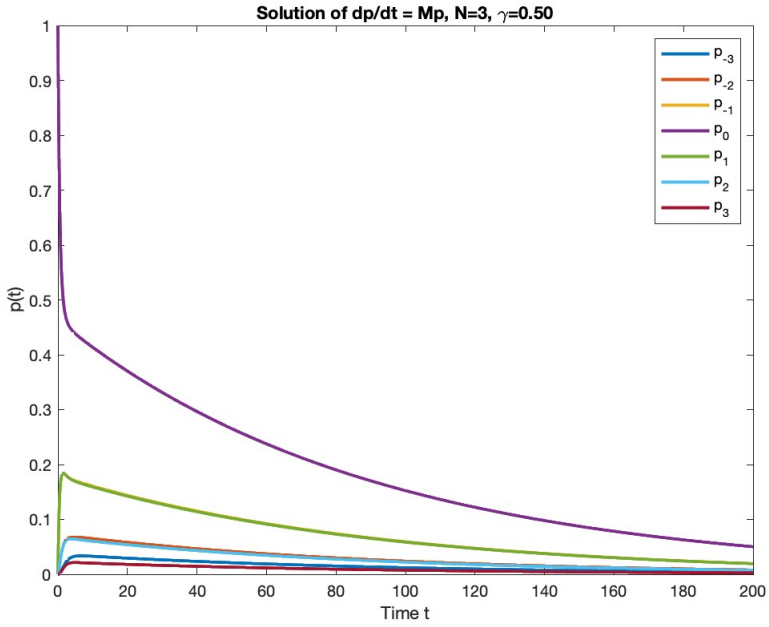

FIGURE 4. $N = 3,\ \gamma = .5$

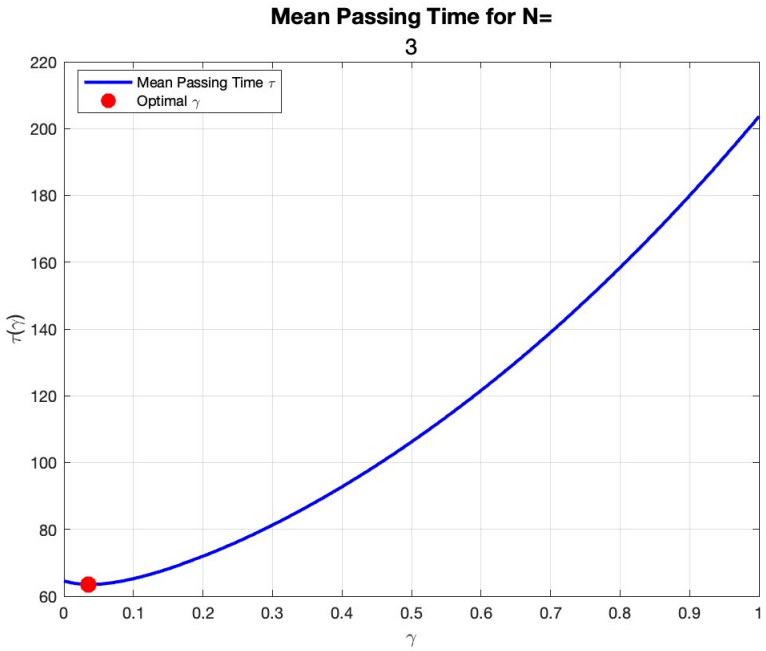

FIGURE 5. Plot of γ^* for $N = 3$

Gamma	0.001	0.1	0.33	0.66	0.99
Tau	64.3501	65.1210	84.3574	131.5019	201.8611

The γ-discretization method yields $\gamma^* = 0.035$, compared with local search algorithm results:

Method	γ^*	τ
Trapezoidal	.034107	63.4594
Simpson's	.037668	63.4594

4.2. **N=5.** We let $N = 5$ and consider a range of γ values to explore the effect of resetting on the probability density functions.

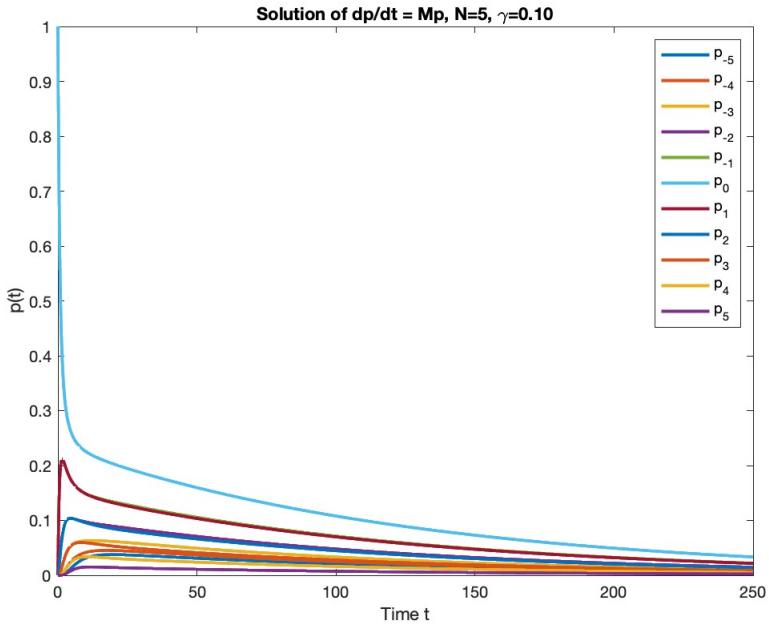

FIGURE 6. $N = 5,\ \gamma = 0.001$

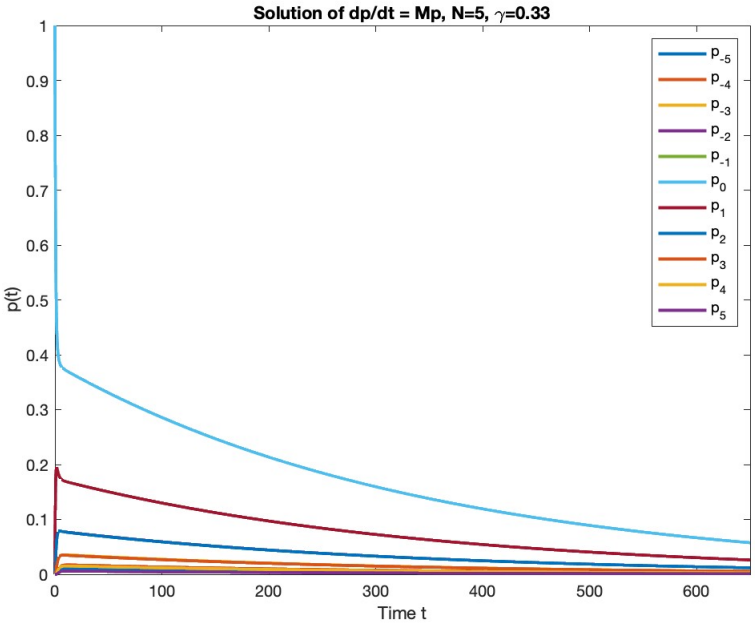

FIGURE 7. $N = 5,\ \gamma = 0.33$

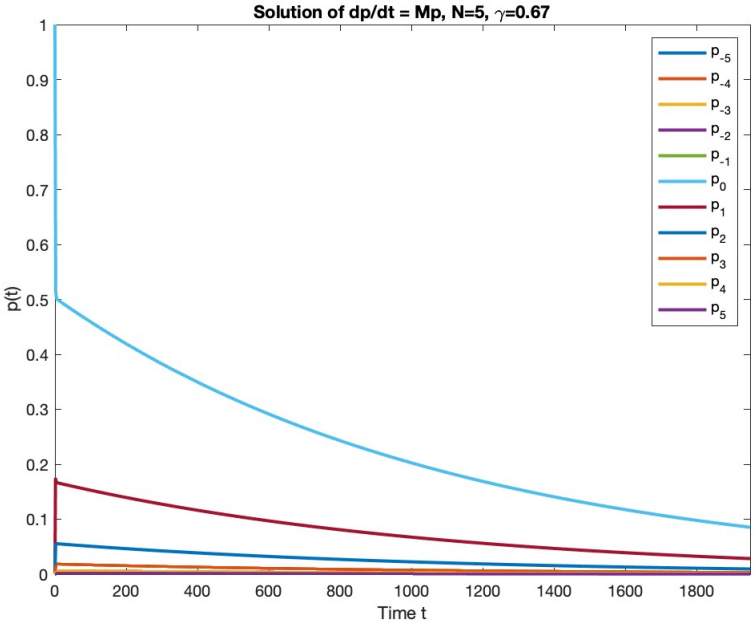

FIGURE 8. $N = 5,\ \gamma = .67$

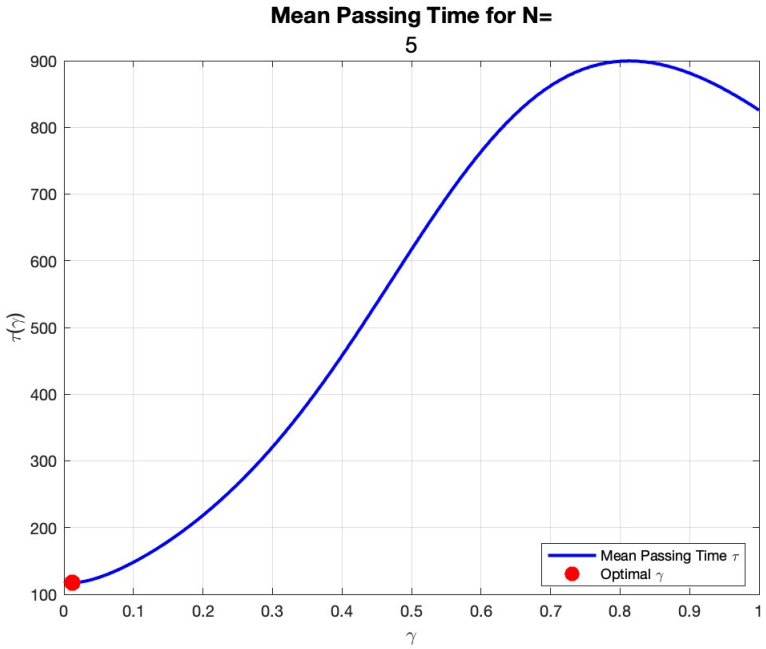

FIGURE 9. Plot of γ^* for $N = 5$

We find the expected passage times to be:

Gamma	0.001	0.1	0.33	0.66	0.99
Tau	118.8344	147.7086	361.3231	1,082.4	2,644.8

The γ-discretization method yields $\gamma^* = 0.0126$, compared with local search algorithm results:

Method	γ^*	τ
Trapezoidal	.012604	117.4509
Simpson's	.012568	117.4509

4.3. **N=8.** We let $N = 8$ and consider a range of γ values to explore the effect of resetting on the probability density functions.

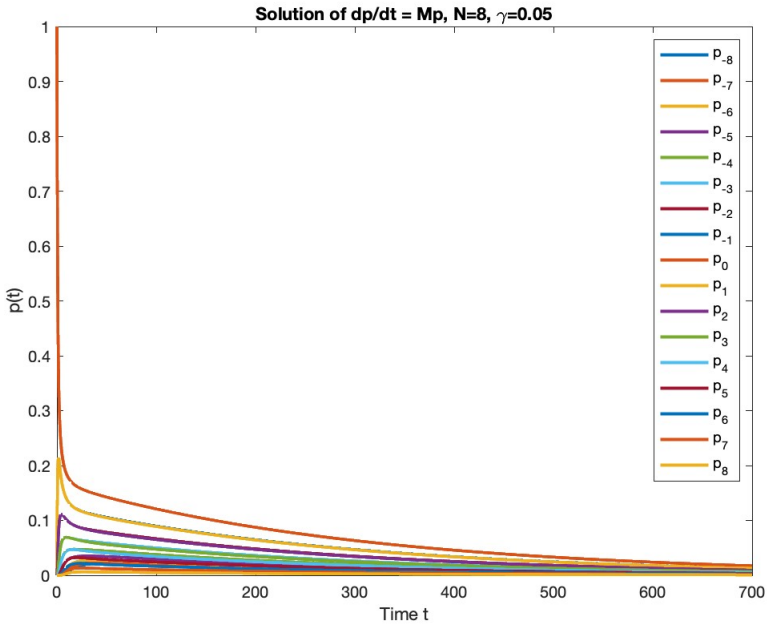

FIGURE 10. $N = 8,\ \gamma = 0.05$

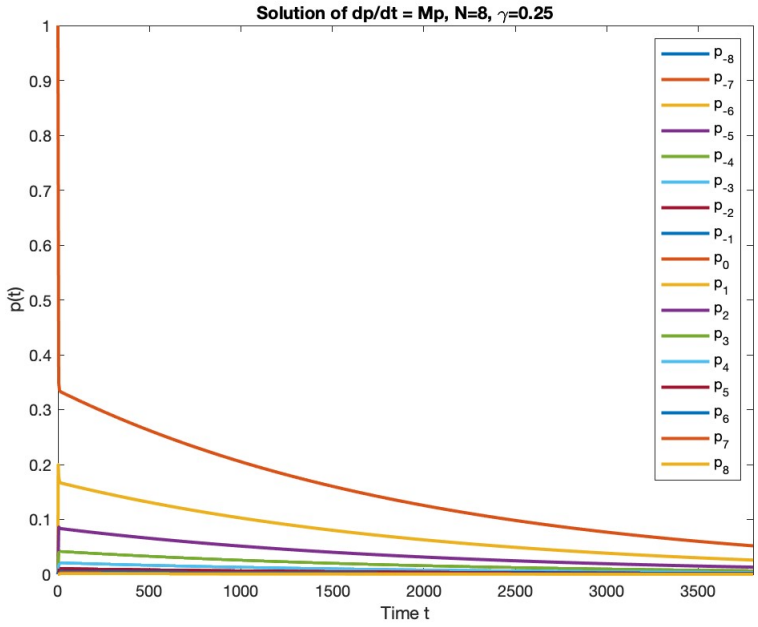

FIGURE 11. $N = 8,\ \gamma = 0.25$

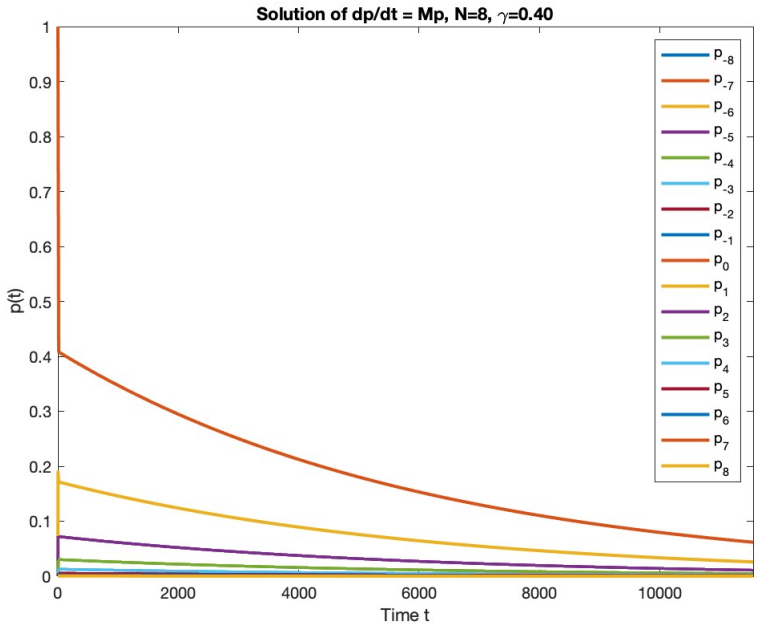

FIGURE 12. $N = 8,\ \gamma = .4$

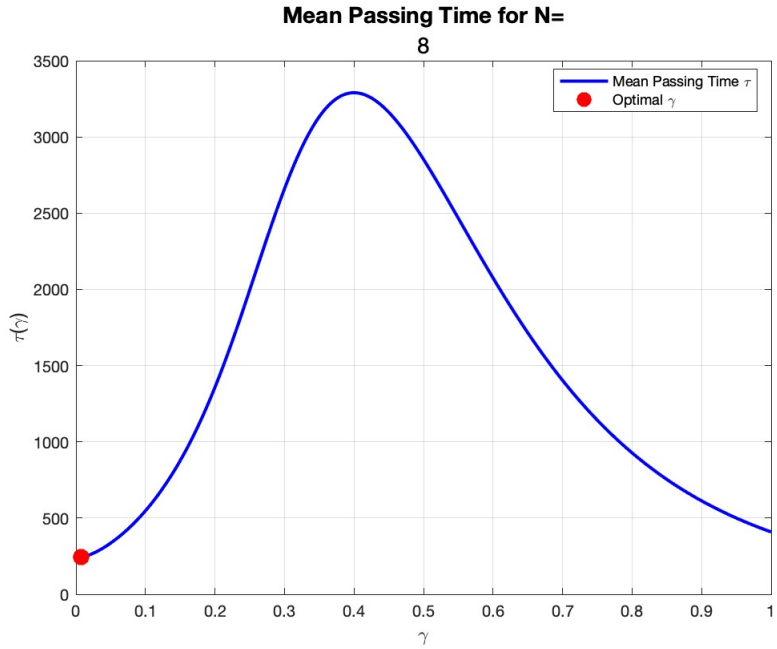

FIGURE 13. Plot of γ^* for $N = 8$

We find the expected passage times to be:

Gamma	0.001	0.1	0.33	0.66	0.99
Tau	249.9	547.8	2,977.7	1,648.8	427.5

The γ-discretization method yields $\gamma^* = 0.0070$, compared with local search algorithm results:

Method	γ^*	τ
Trapezoidal	.0065002	245.8579
Simpson's	.0066884	245.8579

4.4. **N=12.** We let $N = 12$ and consider a range of γ values to explore the effect of resetting on the probability density functions.

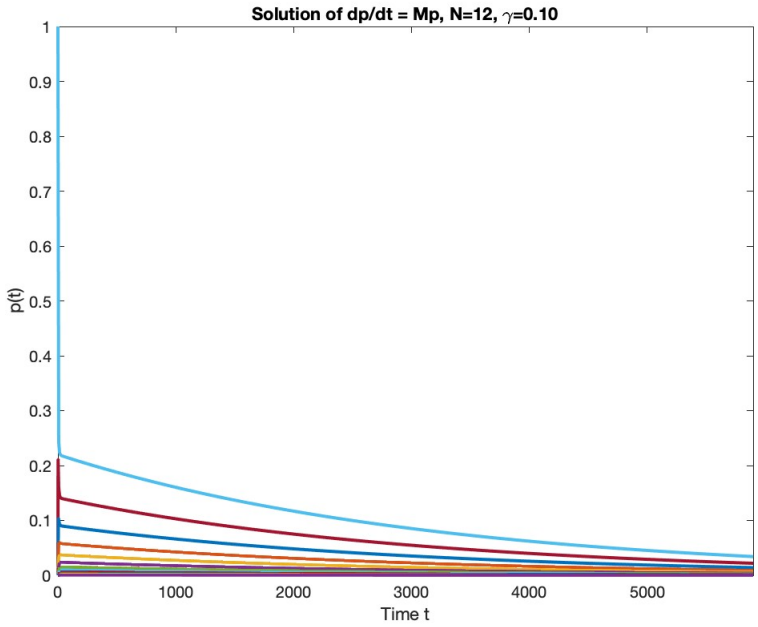

FIGURE 14. $N = 12,\ \gamma = 0.1$

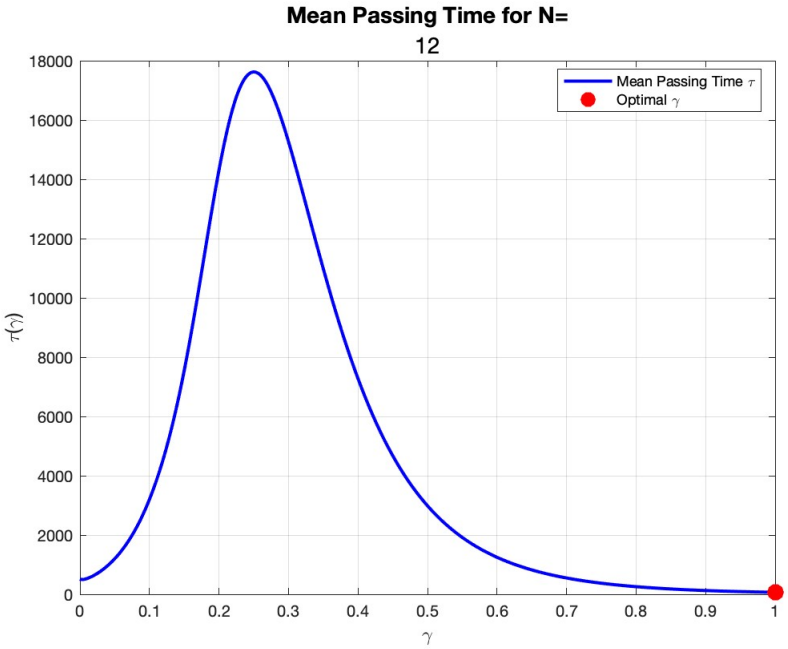

FIGURE 15. Plot of γ^* for $N = 12$

We find the expected passage times to be:

Gamma	0.001	0.1	0.33	0.66	0.99
Tau	505	3,201	12,715	758	70

The γ-discretization method yields $\gamma^* = 1$, compared with local search algorithm results:

Method	γ^*	τ
Trapezoidal	.0032828	498.0643
Simpson's	.0032824	498.0643

4.5. **Discussion of Results.** Solving the system results in $2N+1$ probability density functions which describe the probability that the walker occupies that specific position in the lattice at time t. As a result, the probability density function for the initial position always has initial value of 1, while all other lattice points have initial probability of 0. The effect of a higher resetting rate results in slower decay of the initial and nearby positions, as well as longer time spans required for full capture of system dynamics.

The results for the optimal resetting indicate that less frequent resetting is desirable when the random walk is symmetric, and the resetting rate appears to be inversely correlated with the size of the lattice. An odd behavior is present in the optimal gamma calculations for $N = 8$ and $N = 12$, in which the mean passing time appears decay after a sufficiently large resetting rate. For example, in the case of $N = 8$, the mean passing time decays from a maximum of nearly 18,000 seconds ($\gamma \approx 0.25$) to less than 70 seconds ($\gamma \approx 1$). The interpretation of these results indicates that for $N = 12$, the mean passing time is minimized when the walker resets at each time step–while, intuitively, we should expect a blow up in the mean passing time. More work is needed to address this, but it is our suspicion that the decay in the mean passing time is likely an artifact of the implementation in MATLAB, and not an error with the numerical method. Local search does find a local minimum near zero for $N = 12$, but the global minimum is found to be $\gamma = 1$ even by local search.

5. Code

The following MATLAB code was run on a 2019 MacBook Pro with 2.6 GHz 6-core Intel Core i7 processor.

```
clear; % clear workspace
close all; % close all figures
clc; % clear command window

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Function Declarations %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Build M - This function instantiates the M matrix in the system of
% differential equations
 %Input parameters:
    % gamma: the resetting rate in [0,1]
    % N is the maximum value of the lattice (-N, N)
    % start is the initial position j_0
        % the indexing is handled within the scope of the function such
        % that an initial starting position of j=0 will correspond with the
        % N+1 index of M

function result = Build_M(gamma, N, start)
    % This function builds A and U before adding them together to create M
    A = zeros(2*N+1, 2*N+1); % Initialize matrix A

    % Initialize diagonal vectors for constructing the tridiagonal matrix A
    du = 0.5 * ones(2*N+1, 1); % Upper diagonal
    dl = du; % Lower diagonal is the same size

    d = -1 * ones(2*N+1, 1); % Main diagonal
    d(1) = -1/2; % Adjust 1,1 element

    % Create tridiagonal matrix A
    A = spdiags([dl d du], [-1 0 1], 2*N+1, 2*N+1); % dl for -1, d for 0, du for +1 diagonal
    g = gamma * ones(2*N+1, 1); % Create diagonal matrix for gamma
    G = diag(g); % diagonal matrix for easy matrix arithmetic
    A = A - G; % Modify A with gamma

```

```
36      % Create the U matrix
37      z=N+1; % set zero to N+1 index
38      initial_pos=z+start; % initial position indexing
39      U = zeros(2*N+1, 2*N+1); % initialize empty matrix
40      U(initial_pos, :) = 1; % set the middle row to 1
41
42      % Return M = A + gamma * U
43      M = A + gamma * U;
44      result = M;
45  end
46
47  %Solve System - This function solves the system of differential
        equations
48  % using ODE45
49  % Input parameters:
50      % tol: sum of probabilities must decay to less than or equal
        to tol
51      % time_step: time increment when computing max_time
52      % gamma: resetting rate in [0,1]
53      % N: lattice is (-N,N)
54      % start: initial position
55          % indexing is handled within scope of function--see BuildM
        () for
56          % explanation
57  % Output is a time vector and matrix with function values for each
        p_j
58  function [t,p]= solve_system(tol, time_step, gamma, N, start)
59      M = Build_M(gamma, N, start); % Build M
60      dpdt = @(t, p) M * p; % Define the system
61
62      % Time span for the solution
63      max_t = find_max_time(tol, time_step, gamma, N,start); %
        maximum time for the solution
64      tspan=[0 max_t] % let ODE45 choose timesteps
65
66      % Initial condition for p(t=0)
67      z=N+1; % z gives j=0 on lattice
68      initial_pos=z+start; % index of initial position on lattice
69      p0 = zeros(2*N+1, 1);  % initial condition vector
70      p0(initial_pos) = 1; % Set jth element to 1 for initial cond.
71
72      % Solve using ode45 (RK45)
73      [t, p] = ode45(dpdt, tspan, p0);
74
75  end
```

```
76
77 % Find Maximum Time - this function computes the maximum time
       necessary to
78 % ensure the probabilities decay enough to capture the dynamics of
       the
79 % system
80 % inputs are the same as SolveSystem()
81 function max_time = find_max_time(tol, time_step, gamma, N, start)
82     % Build the system matrix M
83     M = Build_M(gamma, N, start);
84
85     % Define the system
86     dpdt = @(t, p) M * p;
87
88     % Initial condition vector
89     z=N+1; %index of zero
90     initial_pos=z+start; % index of starting position
91     p0 = zeros(2 * N + 1, 1);  % Initial condition vector
92     p0(initial_pos) = 1; % Set jth element to 1 for initial cond.
93
94     % Initialize max_time and sum_last_row
95     max_time = 0; % Start with max_time = 0
96     sum_last_row = Inf; % Initialize to infinity for the while
       loop
97
98     % Iterate until the sum of the last row is less than the
       tolerance level
99     while sum_last_row > tol
100         % Increase max_time by the step
101         max_time = max_time + time_step;
102
103         % Solve the system for time range [0, max_time]
104         [t, p] = ode45(dpdt, [0, max_time], p0);
105
106         % Get the solution at the final time step (last row)
107         last_row = p(end, :);  % Get the last row of matrix p,
       which is the solution at max_time
108
109         % Compute the sum of the last row of p
110         sum_last_row = sum(last_row);  % Sum of the elements in
       the last row
111
112         % If max_time exceeds 150,000, set it to 150,000
113         % comment out if you wish, but it may run for a very long
       time
```

```
        if max_time > 150000
            max_time = 150000;
            fprintf('Max time exceeded 150,000. Setting max time to 150,000.\n')
        return
    end
    end

    % Output the final max_time when the sum of the last row is less than tolerance
    fprintf('Solution converged at max time: %.2f\n', max_time);
end

% Compute Expected Passage Time
% Function to compute tau using trapezoidal method
function result = tau_trapz(gamma, N, start, max_t)
    M = Build_M(gamma, N, start);
    dpdt = @(t, p) M * p;

    % Time span for the solution
    tspan=[0 max_t]

    % Initial condition for p(t=0)
    z=N+1; % index for j=0 on lattice
    initial_pos=z+start; % index of initial position
    p0 = zeros(2*N+1, 1);  % Initial condition vector
    p0(initial_pos) = 1;  % Set the jth element to 1 for intial cond.

    % Solve using ode45 (RK45)
    [t, p] = ode45(dpdt, tspan, p0);

    % Quadrature nodes (time points)
    quad_nodes = linspace(0, max_t, length(p(:,2*N+1)));  % need same length as p(:,j)

    % Compute the integrand as element-wise multiplication
    integrand = 0.5 * quad_nodes' .* p(:, 2*N+1);

    % Compute the integral using the trapezoidal rule
    I = trapz(quad_nodes, integrand);

    result = I;  % Return the computed integral
end
```

```
% Compute Expected Passage Time using Simpson's Rule
function result = tau_simpson(gamma, N, start, max_t)
    M = Build_M(gamma, N, start);  % Build M
    dpdt = @(t, p) M * p;

    % Time span for the solution
    tspan = [0 max_t]%linspace(0, max_t, max_t*3);  % tiny steps

    % Initial condition for p(t=0)
    z = N + 1;  % index for j=0 on lattice
    initial_pos = z + start;  % index of initial position
    p0 = zeros(2*N+1, 1);  % Initial condition vector
    p0(initial_pos) = 1;  % Set the jth element to 1 for initial
    cond.

    % Solve using ode45 (RK45)
    [t, p] = ode45(dpdt, tspan, p0);

    % Quadrature nodes (time points)
    quad_nodes = linspace(0, max_t, length(p(:, 2*N+1)));  % same
    length as p(:, j)

    % Compute the integrand
    integrand = 0.5 * quad_nodes' .* p(:, 2*N+1);

    % Compute the integral using Simpson's Rule
    I = simpson_rule(quad_nodes, integrand);


    result = I;  % Return the computed integral
end

% Simpson's Rule implementation for numerical integration
function I = simpson_rule(x, y)
    n = length(x);
    if mod(n, 2) == 0
        error('Number of points must be odd for Simpson''s rule.')
    ;
    end
    h = (x(end) - x(1)) / (n - 1);
    I = h/3 * (y(1) + 4*sum(y(2:2:end-1)) + 2*sum(y(3:2:end-2)) +
    y(end));
end
```

```
% Backward Equation
% This function returns the expected passage time for a walker beginning in
% the desired start position
function result = BackwardEquation(gamma, N, start)
    M=Build_M(gamma, N, start); % Construct M

    wons=-1*ones(2*N+1,1); % Vector of -1's to solve against

    taus=M'\wons; % Solution to matrix equation
    result=taus(N+1+start) % return the expected passage time for a walker
    % beginning at j
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Desired Parameters for Problem %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% parameters
N = 4; % lattice (-N, N)
gamma=.1; % resetting rate
start=0; % starting position on lattice
tol=10^-8; % tolerance
time_step=500; % time step for computing max_t


%% Solve the system
[t,p]=solve_system(tol, time_step, gamma, N, start);
max_t=t(end)
%% Plot the result
figure;
plot(t, p, 'linewidth', 2); %plot all
% Set x-axis and y-axis limits
xlim([0 max_t*.1]);  % Replace x_min and x_max with your desired range
ylim([0 1]);  % Replace y_min and y_max with your desired range
xlabel('Time t');
ylabel('p(t)');
title(sprintf('Solution of dp/dt = Mp, N=%d, \\gamma=%.2f', N, gamma));
%legend(arrayfun(@(k) sprintf('p_{%d}', k), (-N):N, 'UniformOutput', false));
%% Find expected passage time from start position using backward equation
```

```
other_EP=BackwardEquation(gamma, N, start);

%% compute max time
max_t=find_max_time(tol, time_step, gamma, N, start);

%% compare with solution from integrating
TT=tau_trapz(gamma, N, start, max_t); % Solution by trapz
TS=tau_simpson(gamma, N, start, max_t+1); % Solution by Simpsons
%% difference
diff_bkwrd_trapz=abs(other_EP-TT); % difference between backward
    and trapz
diff_bkwrd_simps=abs(other_EP-TS); % difference between backward
    and simpsons
diff_trapz_simps=abs(TT-TS); % difference between trapz and
    simpsons

%% Optimal Gamma Value using inefficient loop with Trapezoidal
    Method
 %This takes a while to run because its computationally intensive
% compute optimal gamma
l = 1000; % length of gamma vector
gammas = linspace(0, 1, l);  % Define a range of gamma values
tau_vec = zeros(l, 1);       % initialize a vector to store
    results

% Compute tau for each gamma
for i = 1:l
    %max_t=find_max_time(tol, time_step, gammas(i), N, start);
    % line above is commented out because it takes a VERY long
    time to run
    % use only if that level of accuracy is desired
    tau_vec(i) = tau_trapz(gammas(i), N, start, max_t);
end

% Display the optimal gamma value
[~, idx] = min(tau_vec); % find the index of the minimum
gamma_star_trap = gammas(idx); % use the index to find the
    corresponding gamma
disp(['Optimal gamma: ', num2str(gamma_star_trap)]);

%% Optimal Gamma Value using inefficient loop with Simpson's
    Method
% This takes a while to run because its computationally intensive
% compute optimal gamma
l = 1000; % length of gamma vector
```

```
gammas = linspace(0, 1, l);  % Define a range of gamma values
tau_vec = zeros(l, 1);       % initialize a vector to store
    results

% Compute tau for each gamma
for i = 1:l
    %max_t=find_max_time(tol, time_step, gammas(i), N, start);
    % line above is commented out because it takes a VERY long
    time to run
    % use only if that level of accuracy is desired
    tau_vec(i) = tau_trapz(gammas(i), N, start, max_t);
end

% Display the optimal gamma value
[~, idx] = min(tau_vec); % find the index of the minimum
gamma_star_sim = gammas(idx); % use the index to find the
    corresponding gamma
disp(['Optimal gamma: ', num2str(gamma_star_sim)]);

%% plot the optimal gamma
figure;
plot(gammas, tau_vec, 'b-', 'LineWidth', 2);  % Plot tau vs. gamma
    with a blue line
xlabel('\gamma', 'FontSize', 12);  % Label x-axis
ylabel('\tau(\gamma)', 'FontSize', 12);  % Label y-axis
title('Mean Passing Time for N=', num2str(N), 'FontSize', 14);  %
    Title of the plot
grid on;  % Add grid lines to the plot

% Optionally mark the optimal gamma on the plot
hold on;
plot(gamma_star_trap, tau_vec(idx), 'ro', 'MarkerSize', 10, '
    MarkerFaceColor', 'r');  % Red dot at optimal gamma
legend('Mean Passing Time \tau', 'Optimal \gamma', 'Location', '
    Best');
%% recompute using gamma_star
% Find optimal expected passage time from start position using
    backward equation
opt_EP=BackwardEquation(gamma_star_int, N, start);

%% compare with solution from integrating
%max_t_opt=find_max_time(tol, time_step, gamma_star, N, start);
TAU_opt=tau(gamma_star_int, N, start, max_t);

%% Optimization code // trap
```

```
% Define objective function to minimize tau as a function of gamma
objective_function = @(gamma) tau_trapz(gamma, N, start, max_t);
    % Using Trapz Rule

% Use fminbnd to find the optimal gamma value between 0 and
    max_value for search region
[gamma_star_fmin_trapz, tau_min] = fminbnd(objective_function, 0,
    1);

% Define objective function to minimize tau as a function of gamma
objective_function = @(gamma) tau_simpson(gamma, N, start, max_t
    +1);  % Using Simpson's Rule

% Use fminbnd to find the optimal gamma value between 0 and
    max_value for search region
[gamma_star_fmin_simps, tau_min] = fminbnd(objective_function, 0,
    1);
%% Disclaimer: If getting stuck in other local minima, you can
    change max_value to avoid.




%% Difference/Error
delta1=BackwardEquation(gamma_star_fmin_trapz, N, 0);
delta2=BackwardEquation(gamma_star_fmin_simps, N, 0);
Delta=abs(delta1-delta2);


%% Display the optimal gamma value and the corresponding tau
disp(['By Simpsons Rule:'])
disp(['Optimal gamma using fminbnd: ', num2str(
    gamma_star_fmin_simps)]);
disp(['Minimum tau at optimal gamma: ', num2str(tau_min)]);
% Display the optimal gamma value and the corresponding tau
disp(['By Trapezoidal Rule'])
disp(['Optimal gamma using fminbnd: ', num2str(
    gamma_star_fmin_trapz)]);
disp(['Minimum tau at optimal gamma: ', num2str(tau_min)]);
```

References

[Blasius and Tönjes, 2009] Blasius, B. and Tönjes, R. (2009). Zipf's law in the popularity distribution of chess openings. *Phys. Rev. Lett.*, 103:218701.

[Bressloff, 2014] Bressloff, P. (2014). *Stochastic Processes in Cell Biology*. Interdisciplinary Applied Mathematics. Springer International Publishing.

[Evans and Majumdar, 2011] Evans, M. R. and Majumdar, S. N. (2011). Diffusion with stochastic resetting. *Phys. Rev. Lett.*, 106:160601.

[Pal and Prasad, 2019] Pal, A. and Prasad, V. V. (2019). First passage under stochastic resetting in an interval. *Phys. Rev. E*, 99:032123.

[Redner, 2001] Redner, S. (2001). *A Guide to First-Passage Processes*. Cambridge University Press.

[Wang et al., 2021] Wang, S., Chen, H., and Huang, F. (2021). Random walks on complex networks with multiple resetting nodes: A renewal approach. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(9).