Report on Simulation from Endpoint-Conditioned, Continuous-Time Markov Chains on a Finite State Space, with Applications to Molecular Evolution

Vishal Kumar Maurya 2022580
Vaibhav Tapariya 2022556
Yogesh Kumar 2022593
Chawan Srujeeth 2022141

December 8, 2024

Report Structure and Overview

1 Introduction

This report addresses a fundamental challenge in stochastic processes and statistical inference: the reconstruction of complete continuous-time paths of a Markov chain from limited, discrete observations recorded at specific time intervals. **Continuous-Time Markov Chains (CTMCs)** that evolve over a discrete and finite state space. This means we have a system that can be in one of several distinct states and transitions between them at random times governed by some transition rate parameters.

Ideal Scenario and Reality:

- In an ideal scenario, if you had access to continuously observed sample paths (i.e., you knew every change of state and the exact times they occurred), making statistical inferences would be straightforward. Sufficient statistics, such as the number of transitions from one state to another and the total time spent in each state, are easily obtained. These statistics allow straightforward estimation of underlying parameters (transition rates).
- But, in most real-world applications, data are not continuously observed and hence, the process is observed only at discrete time points, common in

fields such as molecular evolution, finance, and ecology. Instead of a complete record, the system is observed at a series of snapshots: $T_0 < T_1 < \cdots < T_N$. At each observation time, the system's state is known, but the transitions between observations are unknown.

The Challenge of Real Scenario

When observations of a Continuous-Time Markov Chain (CTMC) are available only at discrete time points, inferring the complete underlying process becomes a significant challenge. This involves evaluating the conditional distribution of all potential paths that align with the observed endpoints. By sampling from this distribution, it becomes possible to conduct likelihood-based statistical inference and estimate key statistics, such as the frequency of state transitions and the duration spent in each state within the observed intervals. This highlights the complexity of reconstructing the full trajectory of the process while ensuring consistency with the observed data.

But in some cases, the problem reduces to simpler, **closed-form analysis** when the discretization of observation times matches certain assumptions and Transitions and sufficient statistics can be calculated directly without sampling.

Converting into Independent Problems:

The Markov property implies that the process in each interval $[T_k, T_{k+1}]$ can be treated independently. Each interval requires sampling a path consistent with the given start and end states. The full path is the concatenation of these smaller paths.

Although splitting simplifies the problem conceptually, sampling can still be computationally expensive as Naive methods, like forward simulation, can be inefficient. so, Efficient strategies are needed for practical applications. Three main strategies for simulating paths from the conditional distribution are highlighted:

1. **Modified Rejection Sampling**: Improve naive forward simulation by rejecting fewer paths that fail to meet the endpoint conditions.
2. **Direct Sampling (Hobolth's method)**: Use analytical formulas for transition probabilities and waiting times.
3. **Uniformization**: Transform the original CTMC into an equivalent process with fixed Poisson-rate 'virtual' transitions, conditioning on actual state changes.

Central Challenge and Motivation

The Central challenge is that if given only endpoints (observed states at discrete times) of a CTMC, how can we efficiently simulate the full path in between? It reviews the state of the art, emphasizes the practical importance, and sets the stage for comparing and recommending efficient simulation strategies.

Efficient sampling methods are crucial in areas like molecular evolution as it helps in observing sequences at certain times or along a phylogeny requires filling in unobserved mutations. Accurate inference of evolutionary rates and patterns depends on efficient path sampling.

2 Algorithms for Sampling Strategies for CTMCs

2.1 Introducting to intial conditions of CTMCs

Definiting CTMC

A Continuous-Time Markov Chain (CTMC) is a stochastic process $\{X(t) : 0 \leq t \leq T\}$ that models transitions between states over continuous time. The process depends on an instantaneous rate matrix Q, which governs the rates at which transitions between states occur. **Rate Matrix** Q

- $Q_{ab} \geq 0$: Transition rate from state a to state b, where $a \neq b$.
- $Q_{aa} = -\sum_{b \neq a} Q_{ab}$: Negative of the sum of all outgoing rates from state a. This ensures rows of Q sum to zero.

For example, if $Q_{ab} = 0.5$, it means that, on average, the process transitions from state a to b at a rate of 0.5 per unit time.

Intial Assumptions

- $X(0) = a$ (starting state).
- $X(T) = b$ (ending state).
- $X(t)$ is irreducible (any state can be reached from any other state) and positive recurrent (returns to a state in finite time on average).

- The stationary distribution π exists and satisfies $\sum_c \pi_c Q_c = 1$, ensuring one expected state change per unit time.

2.2 Forward Sampling and Naive Rejection Sampling

1. In a Continuous-Time Markov Chain (CTMC), the time τ until the first state change is exponentially distributed with a mean of $1/Q_a$, where $Q_a = -Q_{aa}$. This means the probability density function of τ is given by:

$$f(\tau) = Q_a e^{-Q_a \tau}, \quad \tau \geq 0.$$

2. The total time interval T represents the duration during which the process evolves, from $t = 0$ to $t = T$. Depending on whether τ (time to the first state change) is greater than or less than T, different scenarios arise.

   Condition 1 ($\tau > T$:) If the waiting time τ exceeds T, (the first state change would occur after the total observation period.). Since no state change happens in the interval $[0, T]$, the process remains in the initial state a, $X(t) = a$ for all $t \in [0, T]$.

   Condition 2 ($\tau < T$): If the waiting time τ is less than T (a state change occurs within the observation period), the process transitions from state a to a new state $c \neq a$ at time τ, determined by the transition probabilities Q_{ac}/Q_a. The simulation then proceeds for the interval $[\tau, T]$, with c as the new starting state.

3. To ensure the observed ending state $X(T) = b$, conditioning excludes any paths that do not terminate in state b, and this step is referred to as **Naive Rejection Sampling**.

Limitations

We can classify T as follows into two categories:

- **Case 1 (Large T):** Over a large time interval T, the Markov process has enough time to mix and converge toward its stationary distribution π. The probability of hitting state b is approximately π_b, the stationary probability of b. This makes rejection sampling feasible because π_b is non-zero if b is reachable.
- **Case 2 (Small T):** For small T, there is limited time for the process to transition between states. If $a \neq b$, the probability of reaching b is approximately $Q_{ab}T$, which is small for short T.

Rejection sampling becomes **inefficient**, as most sampled paths will not end at b. **Improvement:** Nielsen's modification improves this by conditioning the sampling to ensure transitions before T.

2.3 Algorithm 1: Modified Rejection Sampling

Algorithm's Theory

Waiting Time τ Distribution: In standard forward sampling, the waiting time τ to the first state change is exponentially distributed:

$$f(\tau) = Q_a e^{-Q_a \tau}, \quad \tau \geq 0.$$

However, in endpoint-conditioned sampling, we must account for the constraint $\tau \leq T$ (because a state change must occur before T).

Conditioned Density $f(\tau)$: To ensure at least one state change before T, the waiting time τ is conditioned on $\tau \leq T$. The conditional probability density function (PDF) is derived using the formula for conditional probabilities:

$$f(\tau \mid \tau \leq T) = \frac{P(\tau) \cdot I(\tau \leq T)}{P(\tau \leq T)},$$

where:

$$P(\tau) = Q_a e^{-Q_a \tau}$$

$$P(\tau \leq T) = \int_0^T Q_a e^{-Q_a \tau} d\tau = 1 - e^{-Q_a T}.$$

Substituting, the conditional PDF becomes:

$$f(\tau) = \frac{Q_a e^{-Q_a \tau}}{1 - e^{-Q_a T}}, \quad 0 \leq \tau \leq T.$$

Cumulative Distribution Function (CDF): The CDF is the integral of the PDF:

$$F(\tau) = \int_0^\tau \frac{Q_a e^{-Q_a s}}{1 - e^{-Q_a T}} ds.$$

Evaluating this:

$$F(\tau) = \frac{1 - e^{-Q_a \tau}}{1 - e^{-Q_a T}}, \quad 0 \leq \tau \leq T.$$

Inverse Transform Sampling: To sample τ from this distribution:

1. Generate $u \sim \text{Uniform}(0, 1)$.

2. Solve for τ using the inverse of the CDF:

$$F^{-1}(u) = -\frac{\log\left(1 - u(1 - e^{-Q_a T})\right)}{Q_a}.$$

The formula for $f(\tau)$ adjusts the exponential distribution to ensure transitions occur before T, enabling rejection sampling to handle endpoint conditions efficiently.

Algorithm Steps:

Case: $a = b$ (Start and End States are the Same):

1. Use the forward sampling algorithm (Algorithm 1) to generate a sample path $\{X(t) : 0 \le t \le T\}$ starting at state a.
2. Accept the simulated path only if $X(T) = a$ (the ending state is the same as the starting state). If $X(T) \neq a$, reject the path and restart from step 1.

Case: $a \neq b$ (Start and End States are Different)

1. **Sample the First Waiting Time τ:** Draw τ, the time to the first state change, from the conditioned exponential density $f(\tau)$ defined by:

   $$f(\tau) = \frac{Q_a e^{-Q_a \tau}}{1 - e^{-Q_a T}}, \quad 0 \le \tau \le T,$$

   where $Q_a = -Q_{aa}$ is the total rate of leaving state a.
2. **Sample the Next State $c \neq a$:** After waiting for τ, transition to a new state $c \neq a$, chosen according to the transition probability distribution:

   $$P(a \to c) = \frac{Q_{ac}}{Q_a}.$$

3. **Simulate the Remaining Path:** Simulate the remainder of the path $\{X(t) : \tau \le t \le T\}$ using the forward sampling algorithm, starting from $X(\tau) = c$ and conditioned to end at $X(T) = b$.
4. **Check the Endpoint:** Accept the simulated path if $X(T) = b$. Otherwise, reject the path and return to step 1.

Key Improvements Over Naive Rejection Sampling

- **Avoids Wasting Constant Paths:** Unlike naive rejection sampling, the modified version ensures at least one state change occurs when $a \neq b$. This avoids generating constant paths where $X(t) = a$ for all $t \in [0, T]$, which are unhelpful and inefficient.
- **Handles Small T Efficiently:** For small T, naive rejection sampling tends to produce a large number of constant paths, especially when T is too short for the chain to reach b. The modified algorithm conditions the first waiting time to ensure a transition occurs.

Limitations

- **Low Transition Probabilities:** If the probability of transitioning from a to b is low (e.g., $\frac{Q_{ab}}{Q_a}$ is small), the algorithm may still reject most paths. This happens because:
  - The chain is unlikely to reach b in time T.
  - Even after conditioning the first transition, subsequent paths might still fail to reach b.
- **Alternative Methods for Hard Cases:** In cases where $\frac{Q_{ab}}{Q_a}$ is very small, this method becomes inefficient. Alternatives like direct sampling or uniformization (discussed further) may be more suitable.

2.4 Algorithm 2: Direct Sampling

Algorithm's Theory

The method assumes that the rate matrix Q can be decomposed as:

$$Q = U D_\lambda U^{-1},$$

where:

- U is an orthogonal matrix with the eigenvectors of Q as its columns,
- D_λ is a diagonal matrix of eigenvalues λ_j,
- U^{-1} is the inverse of U.

The transition probability matrix $P(t) = e^{Qt}$ is then expressed as:

$$P(t) = U e^{t D_\lambda} U^{-1},$$

where e^{tD_λ} is the diagonal matrix with entries $e^{t\lambda_j}$.

Step 1: Case $X(0) = X(T) = a$

No State Change: If the chain starts and ends in the same state a, the probability of no state changes in the interval $[0, T]$ is:

$$p_a = e^{-Q_a T} P_{aa}(T),$$

where $Q_a = -Q_{aa}$ is the total rate of leaving state a.

At Least One State Change: With probability $1 - p_a$, at least one state change occurs in the interval.

Outcome:

- If no state changes occur, the sample path is constant: $X(t) = a$ for all $t \in [0, T]$.
- Otherwise, proceed to simulate state transitions as outlined below.

Step 2: Case $X(0) = a, X(T) = b$, with $a \neq b$

(1.) Probability of First Transition to State i:

The probability that the first state change is to state $i \neq a$, given the chain ends at b, is:

$$p_i = \int_0^T f_i(t)\, dt,$$

where $f_i(t)$ is the conditional density for transitioning to i at time t.

Using eigenvalue decomposition, $f_i(t)$ is:

$$f_i(t) = \frac{Q_{ai} e^{-Q_a t} P_{ib}(T - t)}{P_{ab}(T)}.$$

Substituting $P_{ib}(T - t)$ with its eigenvalue-based expansion, p_i becomes:

$$p_i = \frac{Q_{ai}}{P_{ab}(T)} \sum_j U_{ij} U_{jb}^{-1} J_{aj},$$

where J_{aj} depends on the eigenvalues λ_j and is defined as:

$$J_{aj} = \begin{cases} T e^{T\lambda_j}, & \text{if } \lambda_j + Q_a = 0, \\ \frac{e^{T\lambda_j} - e^{-Q_a T}}{\lambda_j + Q_a}, & \text{if } \lambda_j + Q_a \neq 0. \end{cases}$$

(2.) Sampling the First Waiting Time τ:

Once the first transition state i is determined, the waiting time τ is sampled from $f_i(t)$, which is:

$$f_i(t) = \frac{Q_{ai} e^{-Q_a t} P_{ib}(T - t)}{P_{ab}(T)}.$$

To sample τ:

- Compute the cumulative distribution function (CDF) of $f_i(t)$ by integrating over $[0, t]$.
- Use the inverse transform sampling method to find τ, solving:

$$F_i(\tau) = u, \quad u \sim \text{Uniform}(0, 1).$$

Algorithm Steps

This section outlines the algorithm for sampling paths from a Continuous-Time Markov Chain (CTMC) using eigenvalue decomposition. The algorithm ensures that the sampled path satisfies the given endpoint constraints.

Step 1: Initial Setup

- **Case $a = b$:**
  - Sample $Z \sim \text{Bernoulli}(p_a)$, where:

    $$p_a = e^{-Q_a T} P_{aa}(T),$$

    and $Q_a = -Q_{aa}$ is the total rate of leaving state a.
  - If $Z = 1$, the process remains constant, $X(t) = a$ for all $t \in [0, T]$, and the algorithm stops.
  - If $Z = 0$, at least one state change occurs, and proceed to Step 2.
- **Case $a \neq b$:**
  - Proceed directly to Step 2, as at least one state change is required.

Step 2: Select the Next State i (First Transition)

If at least one state change occurs ($Z = 0$ or $a \neq b$):

- Calculate the probability p_i for transitioning to each state $i \neq a$:

  $$p_i = \frac{Q_{ai}}{P_{ab}(T)} \sum_j U_{ij} U_{jb}^{-1} J_{aj},$$

  where:

  $$J_{aj} = \begin{cases} T e^{T\lambda_j}, & \text{if } \lambda_j + Q_a = 0, \\ \frac{e^{T\lambda_j} - e^{-Q_a T}}{\lambda_j + Q_a}, & \text{if } \lambda_j + Q_a \neq 0. \end{cases}$$

- Normalize p_i values to create a discrete probability distribution for the next state transition:

  $$P(i \mid a) = \frac{p_i}{p_{-a}},$$

  where $p_{-a} = \sum_{i \neq a} p_i$.
- Sample the next state $i \neq a$ based on this distribution.

Step 3: Sample the Waiting Time τ

After determining the next state i, sample the waiting time τ until the transition to i, using the scaled conditional density $f_i(t)$:

$$f_i(t) = \frac{Q_{ai} e^{-Q_a t} P_{ib}(T-t)}{P_{ab}(T)}.$$

- **Cumulative Distribution Function (CDF):** Compute the CDF by integrating $f_i(t)$:

$$F_i(t) = \int_0^t f_i(s)\, ds.$$

- **Inverse Transform Sampling:** Use the inverse transform method to find τ, solving:

$$F_i(\tau) = u, \quad u \sim \text{Uniform}(0,1).$$

Step 4: Simulate the Remaining Path

- Once τ and the next state i are determined:
  - Update $X(\tau) = i$.
  - Repeat Steps 2 and 3 for the remaining time $T - \tau$, starting from i.

Advantages

- **Handles Endpoint Constraints Exactly:** Ensures the sampled path respects the condition $X(T) = b$, avoiding rejection sampling inefficiencies.
- **Uses Full Probability Structure:** Leverages eigenvalue decomposition to explicitly calculate transition probabilities and waiting times.
- **Efficient for Small T:** Avoids wasting effort on paths unlikely to meet endpoint constraints, particularly when T is small.

Limitations

- **Eigenvalue Decomposition:** Computing eigenvalues and eigenvectors of the rate matrix Q can be computationally expensive, especially for large state spaces.
- **Numerical Stability:** For large T or complex eigenvalues, numerical precision issues may arise during calculations of J_{aj} and $P_{ab}(T)$.
- **Iterative Nature:** Requires iterative recalculation of probabilities and waiting times for each transition, which can be computationally intensive for long paths or many states.

2.5 Algorithm 3: Uniformization

Algorithm's Theory

Uniformization is an efficient and elegant method for sampling paths of a Continuous-Time Markov Chain (CTMC) by converting it into a discrete-time Markov process, making use of an auxiliary stochastic process.

1. Idea of Uniformization

Uniformization involves:

- **Augmenting the CTMC with Virtual Jumps:**
  - Some state changes are *virtual*, meaning the chain jumps but remains in the same state. This simplifies the dynamics and allows the CTMC to be treated as a discrete-time Markov chain.
- **Auxiliary Process $Y(t)$:**
  - Define a new process $Y(t)$ driven by a discrete-time Markov process with transition matrix R, where:

$$R = I + \frac{1}{\mu} Q,$$

    with $\mu = \max_c Q_c$, the maximum transition rate across all states in the original CTMC.
- **State Changes Governed by a Poisson Process:**
  - The times of state changes are determined by a Poisson process with rate μ. This ensures state changes occur uniformly in time.

2. Transition Probability Under Uniformization

The CTMC's transition probability matrix $P(t) = e^{Qt}$ can be rewritten using uniformization as:

$$P(t) = e^{-\mu t} \sum_{n=0}^{\infty} \frac{(\mu t)^n}{n!} R^n.$$

This shows that the CTMC is equivalent to a discrete-time Markov chain R^n, where:

- R^n represents n-step transitions in the discrete chain.
- $e^{-\mu t} \frac{(\mu t)^n}{n!}$ is the probability of exactly n jumps occurring in time t, as determined by the Poisson process.

3. Transition Function Subordinated to a Poisson Process

The transition probability from state a to state b in time t, $P_{ab}(t)$, is given by:

$$P_{ab}(t) = e^{-\mu t}\delta_{a=b} + \sum_{n=1}^{\infty} e^{-\mu t}\frac{(\mu t)^n}{n!}R_{ab}^n.$$

This has two components:

- **When $n = 0$:**
  - The chain does not jump at all and remains in the starting state a.
  - Probability: $e^{-\mu t}\delta_{a=b}$.
- **When $n \geq 1$:**
  - The chain undergoes n state transitions (including virtual jumps).
  - Probability: $e^{-\mu t}\frac{(\mu t)^n}{n!}R_{ab}^n$.

4. Distribution of State Changes N

The number of state changes N in the interval $[0, T]$ for a conditional path starting at a and ending at b is distributed as:

$$P(N = n \mid X(0) = a, X(T) = b) = \frac{e^{-\mu T}\frac{(\mu T)^n}{n!}R_{ab}^n}{P_{ab}(T)}.$$

Here:

- The numerator $e^{-\mu T}\frac{(\mu T)^n}{n!}R_{ab}^n$ is the joint probability of n state changes and transitioning from a to b.
- The denominator $P_{ab}(T)$ normalizes the distribution.

5. Sampling the Path

Given $N = n$ (the number of state changes, including virtual jumps):

- **Sample Transition Times:** Uniformly distribute the n jump times $t_1, t_2, \ldots, t_n$ within $[0, T]$.
- **Determine the States:** Use the discrete-time transition matrix R to determine the sequence of states $X(t_1), X(t_2), \ldots, X(t_n)$, conditioned on $X(0) = a$ and $X(T) = b$.
- **Simulate Virtual Jumps:** Include virtual jumps (if the chain "jumps" but remains in the same state) as allowed by R.

Algorithm Steps

1. Simulate the Number of State Changes

Use the distribution (Equation 2.9) to simulate the number of state changes n, where:

$$P(N = n \mid X(0) = a, X(T) = b) = \frac{e^{-\mu T}\frac{(\mu T)^n}{n!}R_{ab}^n}{P_{ab}(T)}.$$

2. Case $n = 0$ (No State Changes)

If $n = 0$, the chain remains in the initial state a throughout the interval:

$$X(t) = a, \quad 0 \leq t \leq T.$$

3. Case $n = 1$ (One State Change)

- **If $a = b$:**
  - The chain remains in state a for the entire interval:

$$X(t) = a, \quad 0 \leq t \leq T.$$

- **If $a \neq b$:**
  - Sample the single transition time t_1 uniformly from $[0, T]$.
  - The state path is:

$$X(t) = \begin{cases} a, & 0 \leq t < t_1, \\ b, & t_1 \leq t \leq T. \end{cases}$$

4. Case $n \geq 2$ (At Least Two State Changes)

- **Sample Transition Times:**
  - Generate n independent random times $t_1, t_2, \ldots, t_n$ uniformly distributed in $[0, T]$.
  - Sort them in increasing order: $0 < t_1 < t_2 < \cdots < t_n < T$.
- **Simulate Intermediate States:**
  - Use the discrete-time Markov transition matrix R to simulate the intermediate states $X(t_1), X(t_2), \ldots, X(t_{n-1})$, conditioned on $X(0) = a, X(T) = b$.
  - The intermediate states are determined by the conditional probability:

$$P(X(t_i) = x_i \mid X(t_{i-1}) = x_{i-1}, X(t_n) = b) = \frac{R_{x_{i-1},x_i}(R^{n-i})_{x_i,b}}{(R^{n-i+1})_{x_{i-1},b}},$$

where R^k is the k-step transition matrix.

- **Virtual Jumps:**
  - Identify which transitions are "virtual" (i.e., jumps where the state does not change).

3 Implementation of the Three Algorithms

3.1 Simulation Description for the Modified Rejection Sampling Algorithm

The code **Modified_Rejection_Sampling.py** implements the simulation of Continuous-Time Markov Chains (CTMCs) using a Modified Rejection Sampling approach. The goal is to generate a valid sample path from an initial state a to a target state b within a given time horizon T, while ensuring transitions adhere to the CTMC properties.

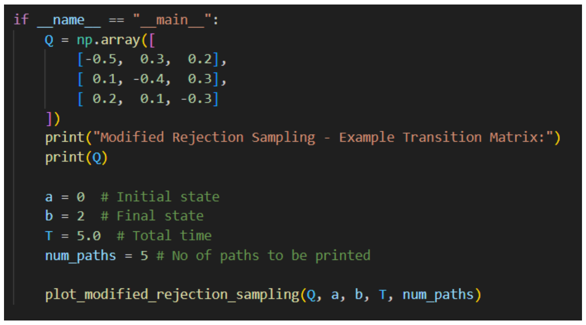
Functions Explanation or Working:

1. `validate_transition_matrix(Q)` Validates the input transition matrix Q to ensure it is a valid generator matrix:
   - Ensures Q is a square matrix.
   - Checks that off-diagonal elements are non-negative.
   - Verifies that the sum of each row equals zero, satisfying CTMC properties.
2. `sample_waiting_time(rate)` Samples the waiting time before transitioning to another state using an exponential distribution:
   - Returns an infinite waiting time if the rate is non-positive.
   - Uses the exponential distribution's mean as $\frac{1}{\text{rate}}$ for sampling.
3. `sample_next_state(current_state, Q, Q_a)` It helps in Sampling the next state based on transition probabilities. It computes transition probabilities by normalizing the corresponding row of Q, excluding self-transitions. and selects the next state using these normalized probabilities.
4. `modified_rejection_sampling(Q, a, b, T, max_iterations=1000)` Implements the Modified Rejection Sampling algorithm to simulate CTMC paths:
   - Initializes the path with the start state a at $t = 0$.
   - Iteratively samples the waiting time and next state until the cumulative time reaches T or the target state b is reached.
   - If a valid path to b is not found within the specified iterations, raises a runtime error.
5. `plot_modified_rejection_sampling(Q, a, b, T, num_paths)` This function generates and visualizes multiple sample paths and calls the `modified_rejection_sampling` function to generate paths.It Uses Matplotlib to plot the state evolution over time with step plots, illustrating the CTMC's piecewise-constant nature and handles runtime errors gracefully, printing a message if a path fails to generate.

Input of the Code:

The inputs to the simulation are as follows:

1. **Input Transition Matrix Q:** A valid generator matrix satisfying:
   - Square matrix with non-negative off-diagonal elements.
   - Each row sums to zero.
2. **Other Simulation Parameters:**
   - **Initial state a:** The starting state of the CTMC.
   - **Target state b:** The state to reach during simulation.
   - **Time horizon T:** Total simulation time.
   - **Number of paths (`num_paths`):** Number of independent paths to simulate and visualize.



```
if __name__ == "__main__":
    Q = np.array([
        [-0.5,  0.3,  0.2],
        [ 0.1, -0.4,  0.3],
        [ 0.2,  0.1, -0.3]
    ])
    print("Modified Rejection Sampling - Example Transition Matrix:")
    print(Q)

    a = 0  # Initial state
    b = 2  # Final state
    T = 5.0  # Total time
    num_paths = 5 # No of paths to be printed

    plot_modified_rejection_sampling(Q, a, b, T, num_paths)
```

Figure 1: Hard Coded Input for the below Result

Result of the Code:

1. **CTMC Simulation:**
   - Simulates paths adhering to CTMC dynamics, using the Modified Rejection Sampling method.
   - Ensures valid transitions by repeatedly sampling waiting times and next states.
   - Stops once the cumulative time exceeds T or the target state b is reached.
2. **Visualization:**
   - Plots the state evolution over time for each generated path using Matplotlib.
   - Employs step plots to capture the piecewise-constant nature of CTMC paths.

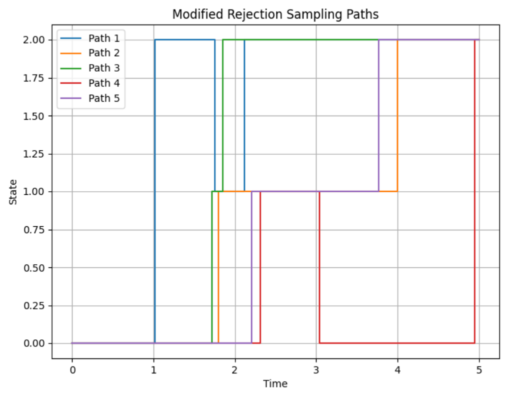


Figure 2: A example of a simulated Modified Rejection Sampling path.

3.2 Simulation Description for the Direct Method Algorithm

Code **Direct_Sampling.py** implements a Continuous-Time Markov Chains (CTMCs) simulation using direct sampling techniques. The goal is to generate sample paths of the CTMC over a given time horizon T, starting from an initial state a, and potentially transitioning to a specified state b.

Functions Explanation or Working:

1. `validate_transition_matrix(Q)` It validates the input transition matrix Q to ensure it is a valid generator matrix following the below properties:
   - Must be square.
   - Off-diagonal elements must be non-negative.
   - Each row's sum should equal zero (ensures the matrix satisfies CTMC properties).
2. `eigenvalue_decomposition(Q)` It performs eigenvalue decomposition on the matrix Q, returning:
   - U: Matrix of eigenvectors.
   - D: Diagonal matrix of eigenvalues.
   - U^{-1}: Inverse of U.

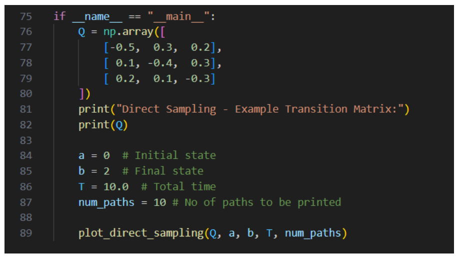
   This decomposition is used for efficient computation of state transition probabilities.
3. `sample_first_state(Q, a, b, T, U, D, U_inv)` Samples the first transition time and determines the next state:
   - Computes the transition probability from the current state a to the next state using matrix exponentiation and eigenvalue decomposition.
   - Uses an exponential distribution to sample the waiting time before transitioning.
4. `direct_sampling(Q, a, b, T)` This function implements direct sampling to generate a single path of the CTMC, starting at the initial state a at time $t = 0$. It iteratively samples the next state and waiting time until the total time exceeds T. It also keeps track of the state transitions and their corresponding times.
5. `plot_direct_sampling(Q, a, b, T, num_paths)` It visualizes multiple sample paths of the CTMC by generating `num_paths` sample paths using the `direct_sampling` function. It uses a step plot to represent the piecewise-constant nature of CTMC states over time.

Input of the Code:

The input for the simulation is hard-coded but can be modified. **Important:** Ensure you follow the properties of the transition matrix as mentioned below:

1. **Input Transition Matrix Q:** Ensure that it is a valid generator matrix following these properties:
   - Must be square.
   - Off-diagonal elements must be non-negative.
   - Each row's sum should equal zero (ensures the matrix satisfies CTMC properties).

2. **Other Simulation Parameters:**
   - **Initial state a:** The starting state of the CTMC.
   - **Target state b:** The state of interest for transitions.
   - **Time horizon T:** The total simulation time.
   - **Number of paths (`num_paths`):** The number of independent paths to simulate.



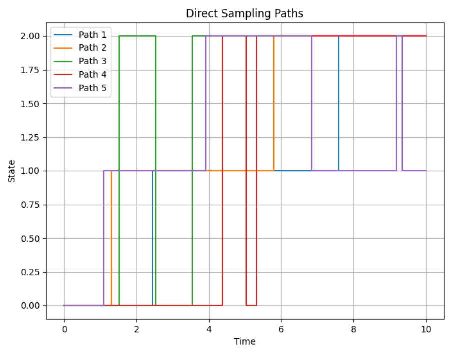
```python
if __name__ == "__main__":
    Q = np.array([
        [-0.5,  0.3,  0.2],
        [ 0.1, -0.4,  0.3],
        [ 0.2,  0.1, -0.3]
    ])
    print("Direct Sampling - Example Transition Matrix:")
    print(Q)

    a = 0  # Initial state
    b = 2  # Final state
    T = 10.0  # Total time
    num_paths = 10 # No of paths to be printed

    plot_direct_sampling(Q, a, b, T, num_paths)
```

Result of the Code:

1. **CTMC Simulation:**
   - Decomposes the matrix Q using eigenvalue decomposition to calculate transition probabilities.
   - Simulates paths by sampling waiting times and transitions, ensuring the state changes adhere to CTMC properties.
   - Stops the simulation once the cumulative time exceeds T.
2. **Visualization:**
   - Uses Matplotlib to plot the sample paths, showing how the state evolves over time for each path.



3.3 Simulation Description for the Uniformization Sampling Algorithm

The code **Uniformization.py** implements a Continuous-Time Markov Chain (CTMC) simulation using the Uniformization Sampling technique. The goal is to generate sample paths of the CTMC starting from an initial state a and potentially reaching a final state b within a time horizon T. The method uses uniformization to simplify the handling of transition times and probabilities.

Functions Explanation or Working:

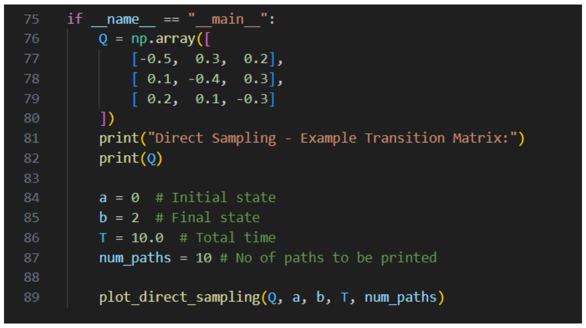
1. `validate_transition_matrix(Q)` Validates the input transition matrix Q to ensure it is a valid generator matrix:
   - Ensures Q is a square matrix.
   - Verifies that off-diagonal elements are non-negative.
   - Checks that the sum of each row equals zero, satisfying CTMC properties.
2. `uniformization_sampling(Q, a, b, T, max_iterations=1000)` Implements the Uniformization Sampling algorithm:
   - Computes the uniformization rate $\mu = \max(-\text{diag}(Q))$, which is the maximum absolute value of the diagonal elements.
   - Constructs the transition probability matrix $R = I + Q/\mu$.
   - Samples the number of transitions from a Poisson distribution with rate μT.
   - Generates transition times uniformly distributed within $[0, T]$ and sorts them.
   - Samples the next state based on the row of the transition matrix R corresponding to the current state.
   - Stops once the cumulative time exceeds T or the target state b is reached.
   - Raises a runtime error if no valid path is found within the allowed iterations.
3. `plot_uniformization_sampling(Q, a, b, T, num_paths)` Visualizes sample paths generated using Uniformization Sampling:

- Repeatedly calls the `uniformization_sampling` function to generate paths.
- Uses Matplotlib to plot the evolution of states over time.
- Displays step plots to represent the piecewise-constant nature of CTMC states.
- Handles runtime errors gracefully, printing a message if a path fails to generate.

Input of the Code:

The inputs to the simulation are as follows:

1. **Input Transition Matrix Q:** A valid generator matrix satisfying:
    - Square matrix with non-negative off-diagonal elements.
    - Each row sums to zero.
2. **Other Simulation Parameters:**
    - **Initial state a:** The starting state of the CTMC.
    - **Target state b:** The state to reach during simulation.
    - **Time horizon T:** Total simulation time.
    - **Number of paths (`num_paths`):** Number of independent paths to simulate and visualize.
    - **Maximum iterations (`max_iterations`):** To prevent infinite loops in case of invalid paths.



```python
if __name__ == "__main__":
    Q = np.array([
        [-0.5,  0.3,  0.2],
        [ 0.1, -0.4,  0.3],
        [ 0.2,  0.1, -0.3]
    ])
    print("Direct Sampling - Example Transition Matrix:")
    print(Q)

    a = 0  # Initial state
    b = 2  # Final state
    T = 10.0  # Total time
    num_paths = 10 # No of paths to be printed

    plot_direct_sampling(Q, a, b, T, num_paths)
```

Figure 3: Hard Coded Input for the below Result

Result of the Code:

1. **CTMC Simulation:**
    - Simulates paths using uniformization, ensuring transitions adhere to CTMC properties.
    - Samples transition times and next states using Poisson and uniform distributions.
    - Ensures the simulation ends at T or upon reaching the target state b.
2. **Visualization:**
    - Plots the evolution of states over time using step plots.
    - Highlights the CTMC's piecewise-constant behavior.

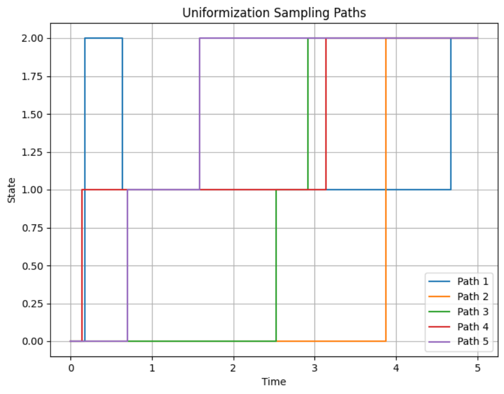


Figure 4: Sample paths generated using Uniformization Sampling.

4 Computational Efficiency of each Sampling Strategy

The research paper discussed 3 real-case examples to show that the computational efficiency depends on the cases based on various factors. We will give a brief summary of the examples but will focus mainly on the observations about the computational efficiency of each method:

4.1 Example 1: Molecular Evolution on the Nucleotide Level

This example demonstrates how Continuous-Time Markov Chain (CTMC) models are applied to DNA sequence evolution at the nucleotide level using the HKY model (Hasegawa, Kishino, and Yano, 1985).

1. Problem Context

Biological Context:

- DNA sequences consist of four nucleotide bases: adenine (A), guanine (G), cytosine (C), and thymine (T).
- Evolution at a single DNA site is modeled as a CTMC, where state transitions represent nucleotide substitutions.

Modeling DNA Substitution:

- The instantaneous rate matrix Q describes substitution likelihoods.
- Substitutions are categorized as:
  - **Transitions (ts):** Between purines (A, G) or pyrimidines (C, T).
  - **Transversions (tv):** Between purines and pyrimidines.

2. The HKY Model

Stationary Distribution (π):

- Long-term nucleotide frequencies: $\pi = (\pi_A, \pi_G, \pi_C, \pi_T)$.
- Example: $\pi = (0.2, 0.3, 0.3, 0.2)$.

Rate Matrix (Q):

$$Q = \frac{1}{s} \begin{bmatrix} \cdot & \kappa\pi_G & \pi_C & \pi_T \\ \kappa\pi_A & \cdot & \pi_C & \pi_T \\ \pi_A & \pi_G & \cdot & \kappa\pi_T \\ \pi_A & \pi_G & \kappa\pi_C & \cdot \end{bmatrix},$$

where:

- κ: Transition/transversion ratio.
- s: Scaling parameter ensuring $\sum_a Q_{aa}\pi_a = 1$.

Parameter Choices:

- $\kappa = 2$, $\pi = (0.2, 0.3, 0.3, 0.2)$.

3. Calibration of the Scaling Parameter (s)

The scaling parameter ensures $\sum_a Q_{aa}\pi_a = 1$, where:

- For humans and chimpanzees: $s = 0.01$.
- For mice and rats: $s = 0.50$.

4. Computational Comparison of Sampling Strategies

Experimental Setup:

- Cases:
  - Same start and end states ($A \to A$).
  - Different start and end states ($A \to G$).
- CPU time measured over varying evolutionary distances.

Observations:

- **Left Panel ($A \to A$):**
  - Rejection Sampling is efficient for short distances.
  - Uniformization becomes competitive as distance increases.
- **Right Panel ($A \to G$):**
  - Rejection Sampling remains efficient.
  - Uniformization outperforms Direct Sampling for shorter distances.

Key Takeaways:

- **Rejection Sampling:** Best for short distances.
- **Direct Sampling:** Suitable for medium to long distances.
- **Uniformization:** Effective for paths with multiple substitutions.

5. Biological Relevance

- Applications:
  - Inferring evolutionary relationships (phylogenetics).
  - Comparing substitution rates across species.

4.2 Example 2: Molecular Evolution on the Codon Level

This example extends nucleotide-level evolution to codons, providing insights into protein-coding regions.

1. Codon-Level State Space

Codons:

- Codons are triplets of nucleotide bases; 61 sense codons encode 20 amino acids.

2. Synonymous and Nonsynonymous Substitutions

- **Synonymous Substitution:** Silent mutations that encode the same amino acid.
- **Nonsynonymous Substitution:** Mutations encoding different amino acids, potentially altering protein function.

3. The Goldman-Yang (GY) Model

The rate matrix Q:

$$Q_{ab} = \begin{cases} 0 & \text{if } a \text{ and } b \text{ differ at more than one position,} \\ \pi_b & \text{for synonymous transversions,} \\ \kappa\pi_b & \text{for synonymous transitions,} \\ \omega\pi_b & \text{for nonsynonymous transversions,} \\ \omega\kappa\pi_b & \text{for nonsynonymous transitions.} \end{cases}$$

Parameters:

- π_b: Stationary frequency of codon b.
- κ: Transition/transversion rate ratio (favoring transitions).
- ω: Ratio of nonsynonymous to synonymous changes (reflecting selective pressures).

Scaling Parameter:

- The scaling parameter $s = s(\omega, \kappa, \pi)$ ensures that the matrix Q is scaled such that:

$$\sum Q_{aa}\pi_a = 1,$$

  meaning t substitutions are expected in t time units.

4. Application of the GY Model

Parameter Choices:

- $\kappa = 2$: Transitions are twice as likely as transversions.
- $\omega = 0.01$: Synonymous changes are far more likely than nonsynonymous changes.
- Stationary distribution π: Based on codon usage patterns.
  - Smallest π: $\pi_{\text{GGG}} = 0.0042$.
  - Largest π: $\pi_{\text{GAG}} = 0.0426$.

Examples:

- $AAA \to AAG$: Synonymous transition.
- $AAA \to TTT$: Nonsynonymous substitution.

Observations:

- **Left Plot ($AAA \to AAG$):**
  - Rejection Sampling is efficient for frequent synonymous transitions.
- **Right Plot ($AAA \to TTT$):**
  - Rejection Sampling is inefficient for rare nonsynonymous transitions.

5. Biological Insights

- **Synonymous vs. Nonsynonymous Changes:**
  - Synonymous changes are much more frequent due to selective pressure to preserve protein function.
  - Nonsynonymous changes occur less frequently and can indicate selective pressures.
- **Codon Usage:**
  - The stationary distribution π reflects real-world codon usage frequencies, which are highly heterogeneous.
- **Selective Pressure:**
  - The parameter ω quantifies selective pressure on nonsynonymous substitutions:
    * $\omega < 1$: Purifying selection (e.g., to conserve protein function).
    * $\omega > 1$: Positive selection (e.g., advantageous mutations).

4.3 Example 3: Molecular Evolution on the Sequence Level

This example investigates molecular evolution on the sequence level, focusing on the computational efficiency of different sampling methods (Rejection Sampling, Direct Sampling, and Uniformization) for modeling codon evolution using the HKY + CpG model.

1. Motivation

Previous Insights:

- Rejection Sampling is efficient when the ending state is likely (high acceptance probability).
- Uniformization is suitable when many intermediate virtual substitutions are required.

- Direct Sampling is efficient for larger evolutionary distances.

Goal: To demonstrate scenarios where each method performs better and to explore the HKY + CpG model.

2. The HKY + CpG Model

Description:

- Extends the HKY model to include neighbor-dependent substitutions.
- Accounts for the influence of neighboring nucleotides on mutation rates.

CpG Dinucleotides:

- CpG sites are hotspots for mutation due to methylation-deamination processes.
- Mutations are more likely if a CpG dinucleotide is present.

Rate Matrix Adjustment:

$$Q_{\text{HKY+CpG}} = \begin{bmatrix} \kappa\nu_A & \nu_G & \nu_C & \nu_T \\ \kappa\nu_G & \nu_A & \gamma\nu_C & \nu_T \\ \kappa\nu_C & \nu_G & \nu_A & \gamma\nu_T \\ \kappa\nu_T & \nu_G & \nu_C & \nu_A \end{bmatrix},$$

where:

- γ: CpG effect scaling parameter.
- $\nu_A, \nu_G, \nu_C, \nu_T$: Stationary nucleotide frequencies.

Stationary Distribution:

$$\pi = (\frac{\nu_A}{\nu_A + \nu_G/\gamma + \nu_C/\gamma + \nu_T}, \frac{\nu_G/\gamma}{\nu_A + \nu_G/\gamma + \nu_C/\gamma + \nu_T}$$
$$, \frac{\nu_C/\gamma}{\nu_A + \nu_G/\gamma + \nu_C/\gamma + \nu_T}, \frac{\nu_T}{\nu_A + \nu_G/\gamma + \nu_C/\gamma + \nu_T}).$$

Example Parameters:

- $\nu_A = 0.3, \nu_G = 0.3, \nu_C = 0.2, \nu_T = 0.2$.
- $\gamma = 20$ (strong CpG effect).

3. Comparison of Sampling Methods

Performance Analysis:

- Sampling efficiency is measured in terms of CPU time as a function of evolutionary distance.
- Scenarios:
  - Starting State $TT \to CC$.
  - Starting State $CC \to TT$.

Results:

- $CC \to TT$(Left Plot):
  - $T < 0.3$: Rejection Sampling is most efficient.
  - $0.3 \leq T < 0.9$: Uniformization performs best.
  - $T > 0.9$: Direct Sampling is the most efficient.
- $CC \to TT$ (Right Plot):
  - Rejection Sampling remains efficient due to high acceptance probability.

4. Key Insights

Efficiency of Sampling Methods:

- **Rejection Sampling:** Efficient for high acceptance probability.
- **Uniformization:** Best for intermediate distances.
- **Direct Sampling:** Optimal for large distances.

Biological Relevance:

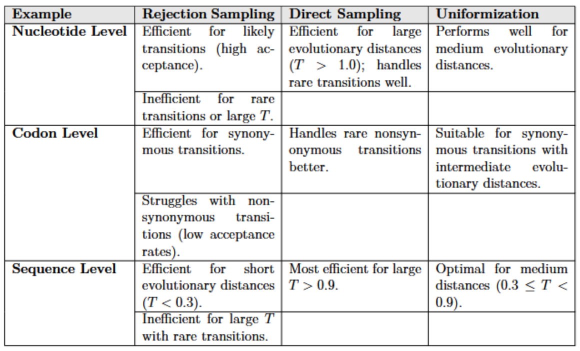
- CpG effects are crucial for studying context-dependent mutation rates.
- This model enhances understanding of methylation-deamination processes.

5. Conclusion

This example illustrates the trade-offs between different sampling strategies. The choice depends on:

- **Evolutionary Distance:**
  - Short distances favor Rejection Sampling.
  - Medium distances favor Uniformization.
  - Long distances favor Direct Sampling.
- **Biological Context:** Incorporating CpG effects provides a realistic model of mutation processes, particularly for mammalian DNA.

Each method aligns computational strategies with biological phenomena, advancing the study of molecular evolution.



Example	Rejection Sampling	Direct Sampling	Uniformization
Nucleotide Level	Efficient for likely transitions (high acceptance).	Efficient for large evolutionary distances ($T > 1.0$); handles rare transitions well.	Performs well for medium evolutionary distances.
	Inefficient for rare transitions or large T.		
Codon Level	Efficient for synonymous transitions.	Handles rare nonsynonymous transitions better.	Suitable for synonymous transitions with intermediate evolutionary distances.
	Struggles with non-synonymous transitions (low acceptance rates).		
Sequence Level	Efficient for short evolutionary distances ($T < 0.3$).	Most efficient for large $T > 0.9$.	Optimal for medium distances ($0.3 \leq T < 0.9$).
	Inefficient for large T with rare transitions.		

Table 1: Summary of Sampling Efficiency Across Examples

4.4 Summary of Sampling Efficiency

4.5 Simulating Computational Complexity of Sampling Algorithms

1. Explanation of the Code

Efficiency Calculation

The function `calculate_efficiency` determines the CPU time for each sampling method:

- **Rejection Sampling:** Efficiency is inversely proportional to acceptance probability.
- **Direct Sampling:** Efficiency linearly increases with T.
- **Uniformization:** Efficiency includes overhead for simulating virtual substitutions.

Simulation

The function `simulate_example` generates efficiency data for each example based on varying T values.

Plotting

The function `plot_all_examples` creates a 3-panel figure:

- Each panel corresponds to one example.
- Efficiency curves for **Rejection Sampling**, **Direct Sampling**, and **Uniformization** are plotted.

Layout

All three plots are displayed side-by-side for easy comparison.

2. Output

Figure Layout

When the script is executed, the figure displays:

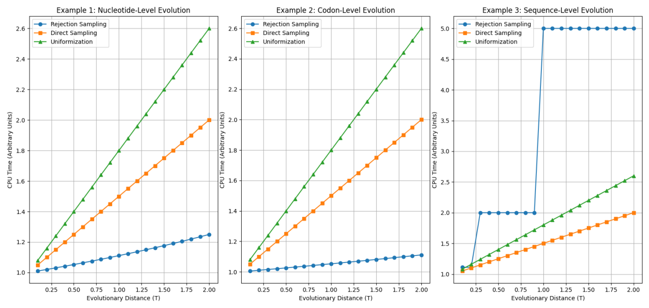
- **Left Panel:** Efficiency plot for Example 1 (Nucleotide-Level Evolution).
- **Middle Panel:** Efficiency plot for Example 2 (Codon-Level Evolution).
- **Right Panel:** Efficiency plot for Example 3 (Sequence-Level Evolution).

Panel Details

Each panel shows:

- **X-axis:** Evolutionary Distance (T).
- **Y-axis:** CPU Time (Arbitrary Units).
- **Legend:** Sampling methods (Rejection, Direct, Uniformization).

4.6 Comparison of Result Observed with the Plots of the Simulation



Examples shown were chosen to demonstrate each sampling strategy's heterogeneous dependence upon the endpoint-conditioned CTMC's characteristics. In particular, efficiency was shown to be impacted by each aspect of the process: the instantaneous rate matrix Q, the sampling time T, and the beginning and ending states a and b. I will analyze the visual data provided to compare the results of sampling efficiency with the plots and ensure the output matches. Let's break this down:

1. Nucleotide-Level Evolution

- **Rejection Sampling:** The plot shows relatively low CPU time across increasing evolutionary distances (efficient for likely transitions), but a slight inefficiency starts appearing as T increases.

- **Direct Sampling:** CPU time grows linearly but remains moderate, supporting efficiency for large $T > 1.0$.
- **Uniformization:** CPU time grows steeply, aligning with it being less efficient for large T and better for medium evolutionary distances.

2. Codon-Level Evolution

- **Rejection Sampling:** CPU time increases slightly but is less than uniformization, showing efficiency for synonymous transitions but struggles with non-synonymous ones (lower acceptance).
- **Direct Sampling:** CPU time grows linearly, supporting handling rare transitions better.
- **Uniformization:** Linear growth in CPU time aligns with it being suitable for synonymous transitions at intermediate distances.

3. Sequence-Level Evolution

- **Rejection Sampling:** A sharp increase in CPU time after $T = 0.3$ confirms inefficiency for larger T.
- **Direct Sampling:** Moderate and consistent growth, proving efficiency for large $T > 0.9$.
- **Uniformization:** Linear and consistent growth supports being optimal for $0.3 \leq T < 0.9$.

The graphical data from the plots aligns well with the summary provided in Table 1, validating the observations about sampling methods' efficiency across evolutionary levels.

5 Generalized Computational Complexity

5.1 Rejection Sampling Complexity in Continuous-Time Markov Chains (CTMCs)

The computational cost of generating a sample path is broken into three stages:

- **Initialization:** Fixed computational cost (α).
- **Recursion:** Stochastic computational cost proportional to the number of recursive steps (βL), where β is the cost per recursion, and L is the number of steps.
- **Termination:** The process halts when a valid sample path from the starting state (a) to the ending state (b) is generated.

The total computational cost for generating one sample path is expressed as:

$$\alpha + \beta L$$

The mean computational cost is:

$$\alpha + \beta \mathbb{E}[L],$$

where $\mathbb{E}[L]$ is the expected number of recursion steps.

1. Acceptance Probability (p_{acc})

- The acceptance probability is critical for understanding rejection sampling efficiency. It represents the likelihood that a randomly generated sample path matches the desired conditions (e.g., starting at a and ending at b).
- The formula for the acceptance probability when starting and ending states are the same ($a = b$) is:

  $$p_{\text{acc}} = P_{aa}(T),$$

  where $P_{aa}(T)$ is the probability of staying in state a over time T.
- For small T, the approximation is:

  $$p_{\text{acc}} \approx 1 - Q_{aa}T,$$

  where Q_{aa} is the diagonal entry of the rate matrix Q.
- For large T, p_{acc} approaches π_a, the stationary probability of state a.

2. Expected Recursion Steps ($\mathbb{E}[L]$)

The expected number of recursion steps required to generate a valid sample path is given by:

$$\mathbb{E}[L] = \sum_i \sum_{j \neq i} \mathbb{E}[N_{ij}(T) \mid X(0) = a],$$

where $N_{ij}(T)$ is the number of state changes from i to j in the time interval $[0, T]$.

The formula is expanded using the rate matrix Q and time integrals:

$$\mathbb{E}[N_{ij}(T)] = Q_{ij} \int_0^T P_{ij}(s)\, ds.$$

This computation relies on the eigenvalue decomposition of Q.

3. CPU Time for Rejection Sampling

The computational cost for generating a valid sample path is:

$$\frac{\alpha + \beta \mathbb{E}[L]}{p_{\text{acc}}},$$

where:

- α: Fixed cost of initialization.
- β: Cost per recursion step.
- $\mathbb{E}[L]$: Expected number of recursion steps.
- p_{acc}: Acceptance probability.

—

Key Observations from Figures

1. Top Row: Starting and Ending States are the Same $(a = b)$

- **Acceptance Probability:** Decreases nonlinearly with T, starting at $p_{\text{acc}} \approx 1$ for small T.
- **Initialization Cost:** Linear regression is used to estimate α from the CPU time spent on initialization.
- **Recursion Cost:** CPU time for sampling increases with $\mathbb{E}[L]$, proportional to β/p_{acc}.

2. Bottom Row: Starting and Ending States are Different $(a \neq b)$

- The acceptance probability p_{acc} is calculated using:

  $$p_{\text{acc}} = \frac{P_{ab}(T)}{1 - P_{aa}(T)},$$

  where $P_{ab}(T)$ is the probability of transitioning from a to b in time T.

Key Graphical Interpretations

1. Acceptance Probability

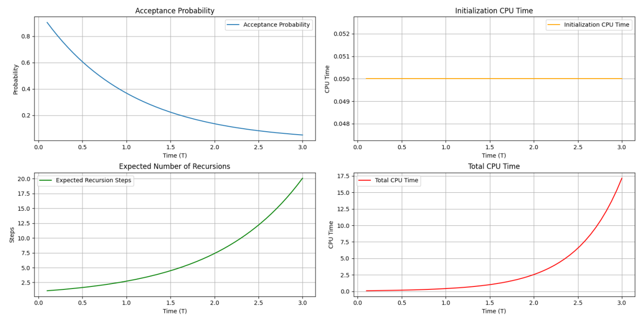
- p_{acc} decreases with T, especially for $a \neq b$.
- For large T, rejection sampling becomes less efficient due to low p_{acc}.

2. Initialization and Recursion Costs

- CPU time on initialization is independent of T and can be estimated as $\alpha = 0.0509$.
- CPU time on sampling is proportional to $\mathbb{E}[L]/p_{\text{acc}}$, with $\beta = 0.0365$.

3. Total CPU Time

- The sum of initialization and sampling costs gives the total CPU time.
- Linear regression confirms the predicted relationship between T and computational complexity.



Code Simulation for the Complexity Analysis:

This Python code `Rejection_Sampling_Complexity.py` simulates and visualizes the computational complexity of rejection sampling for Continuous-Time Markov Chains (CTMCs). It calculates key metrics, including acceptance probability, expected recursion steps, initialization cost, and total CPU time required for generating valid sample paths.

Main Components

- **Acceptance Probability:** Measures the likelihood of generating a valid path.
- **Expected Recursion Steps:** Determines the number of steps required to generate valid paths.
- **CPU Time:** Combines initialization and recursion costs, scaled by the acceptance probability.

Function Descriptions

- `acceptance_probability`: Computes the acceptance probability for rejection sampling based on the rate matrix Q and time T.
- `expected_recursion_steps`: Calculates the expected number of recursion steps needed to generate valid sample paths.
- `cpu_time`: Computes the total CPU time for rejection sampling, including initialization and recursion costs.

- `plot_rejection_sampling:` Plots acceptance probability, initialization cost, recursion steps, and total CPU time for rejection sampling.

5.2 Direct Sampling Complexity

Computational Costs

The computational cost of generating one sample path in direct sampling is expressed as:

$$\alpha + \beta L,$$

where:

- α: Fixed cost for initialization (e.g., eigenvalue decomposition of Q).
- β: Cost per recursion step (sampling a new state and its waiting time).
- L: Number of recursion steps (state transitions).

Unlike rejection sampling, the initialization cost (α) for direct sampling is higher because it requires an eigendecomposition of the rate matrix Q, which is computationally expensive.

Expected Number of Recursion Steps ($\mathbb{E}[L]$)

The expected number of recursion steps $\mathbb{E}[L]$ is equivalent to the number of state changes (N) and can be computed as:

$$\mathbb{E}[L] = \mathbb{E}[N(T) \mid X(0) = a, X(T) = b],$$

where:

$$\mathbb{E}[N(T)] = \sum_i \sum_{j \neq i} \mathbb{E}[N_{ij}(T) \mid X(0) = a, X(T) = b].$$

Here:

- $N_{ij}(T)$: Number of state changes from i to j during the time interval $[0, T]$.

These expectations are calculated using formulas involving the rate matrix Q, as described by Hobolth and Jensen (2005).

Comparison of Initialization and Recursion Costs

Initialization Cost (α):

- From Figure 6, α for direct sampling is higher than for rejection sampling.
- For example:
  - $\alpha = 0.85$ for direct sampling.
  - $\alpha = 0.05$ for rejection sampling.
- This higher cost is attributed to the matrix decomposition required in direct sampling.

Recursion Cost (β):

- The cost per recursion step for direct sampling is also higher:
  - $\beta = 0.56$ for direct sampling, compared to $\beta = 0.04$ for rejection sampling.

While these numbers may seem to favor rejection sampling, the efficiency of direct sampling depends on the overall probability of acceptance (p_{acc}).

Key Graphical Interpretations

The figure analyzes two cases for the HKY model:

Case 1: Starting and Ending States are the Same ($a = A, b = A$)

- **Initialization CPU Time (Left Plot):**
  - Constant over time (T).
  - $\alpha = 0.274$, significantly higher than for rejection sampling.
- **Sampling CPU Time (Middle Plot):**
  - Grows linearly with time (T).
  - $\beta = 0.1342$, proportional to the expected number of state changes.

Case 2: Starting and Ending States are Different ($a = A, b = G$)

- **Initialization CPU Time (Right Plot):**
  - Similar constant cost as the first case ($\alpha = 0.2258$).
- **Sampling CPU Time (Far Right Plot):**
  - Linear growth, with $\beta = 0.137$.

Trade-Offs Between Rejection and Direct Sampling

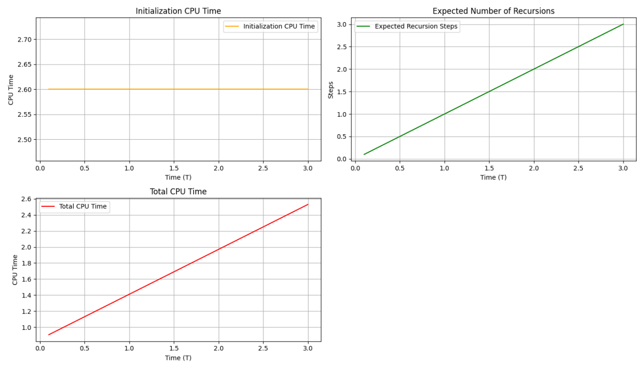
Rejection Sampling:

- Computationally inexpensive for initialization and recursion steps.
- Highly dependent on p_{acc}, the acceptance probability.

- If p_{acc} is low, the cost of generating viable paths increases sharply due to the factor $1/p_{\text{acc}}$.

Direct Sampling:

- Initialization and recursion costs are higher but do not depend on p_{acc}.
- Ensures all generated sample paths meet the desired endpoint conditions.
- More viable for scenarios where p_{acc} in rejection sampling is low.



Code Simulation for the Complexity Analysis: Direct Sampling

This Python code simulates and visualizes the computational complexity of Direct Sampling for Continuous-Time Markov Chains (CTMCs). It calculates key metrics, including initialization cost (via eigenvalue decomposition), expected recursion steps, and total CPU time for generating valid sample paths.

Main Components

- **Eigen Decomposition:** Calculates initialization cost (α).
- **Expected Recursion Steps:** Computes state transitions proportional to time T.
- **CPU Time:** Combines initialization and recursion costs.

Function Descriptions

1. `eigen_decomposition(Q):` Performs eigenvalue decomposition of the rate matrix Q.
   - Used to calculate initialization cost for direct sampling.
2. `uniformization_rate(Q):` Computes the uniformization rate (maximum absolute value of diagonal entries of Q).
   - Used for constructing the auxiliary matrix R for uniformization sampling.
3. `auxiliary_transition_matrix(Q, mu):` Constructs the auxiliary transition matrix R for uniformization based on the rate matrix Q and the uniformization rate μ.
4. `expected_recursion_steps_direct(Q, T, a, b):` Calculates the expected number of recursion steps (state changes) for direct sampling.

5.3 Uniformization Complexity

Computational Costs

The total computational cost for uniformization is similar in structure to direct sampling, as it involves:

- **Initialization:** Requires eigenvalue decomposition of the rate matrix Q and constructing an auxiliary transition matrix R.
- **Recursion:** Consists of sampling a new state and its corresponding waiting time.

Expected Number of Recursion Steps ($\mathbb{E}[L]$)

Uniformization reveals that the number of recursion steps L equals the number of state changes $N(T)$ accumulated by the auxiliary chain. The expected number of steps is calculated as:

$$\mathbb{E}[L] = \mathbb{E}[N(T) \mid X(0) = a, X(T) = b].$$

Expanding this:

$$\mathbb{E}[L] = \frac{1}{P_{ab}(T)} \sum_{n=0}^{\infty} n e^{-\mu T} \frac{(\mu T)^n}{n!} (R^n)_{ab},$$

where:

- $P_{ab}(T)$: Probability of transitioning from state a to b in time T.
- μ: Uniformization rate.
- R: Transition probability matrix for the auxiliary chain.

Simplifying further:

$$\mathbb{E}[L] = \frac{\mu T}{P_{ab}(T)} (Re^{QT})_{ab}.$$

For large T, $\mathbb{E}[L]$ approximates:

$$\mathbb{E}[L] \approx \mu T.$$

Key Graphical Interpretations

The figure provides a visual representation of uniformization complexity for the HKY model, comparing two scenarios:

Case 1: Starting and Ending States are the Same ($a = A, b = A$)

- **Initialization CPU Time (First Plot):**
  - Constant over time (T).
  - $\alpha = 0.2503$, slightly higher than rejection sampling.
- **Sampling CPU Time (Second Plot):**
  - Grows linearly with T.
  - $\beta = 0.0253$, much lower than direct sampling and rejection sampling.

Case 2: Starting and Ending States are Different ($a = A, b = G$)

- **Initialization CPU Time (Third Plot):**
  - Similar constant cost ($\alpha = 0.2419$).
- **Sampling CPU Time (Fourth Plot):**
  - Linear growth with T.
  - $\beta = 0.0206$, confirming lower recursion costs compared to other methods.

Key Observations

Initialization Cost (α):

- Uniformization requires eigenvalue decomposition of Q and computation of R, making $\alpha \approx 1.05$.
- This cost is comparable to direct sampling but higher than rejection sampling.

Recursion Cost (β):

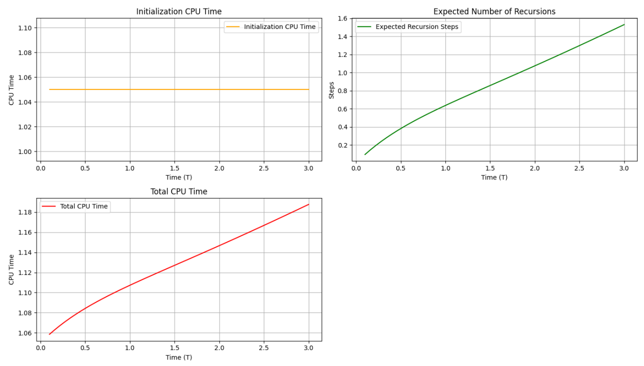
- At $\beta = 0.09$, the recursion cost for uniformization is approximately $1/6$th of the recursion cost for direct sampling.
- This efficiency stems from leveraging the auxiliary transition matrix R, which precomputes relevant probabilities.

Expected Number of Steps ($\mathbb{E}[L]$):

- The recursion step is immediate because auxiliary probabilities from R are precomputed.
- For large T, $\mathbb{E}[L] \approx \mu T$, scaling linearly with time.

Efficiency Trade-Off:

- Uniformization has slightly higher initialization costs but compensates with much lower recursion costs, making it highly efficient for medium-to-large T.



Code Simulation for the Complexity Analysis: Uniformization Sampling

This Python code simulates and visualizes the computational complexity of Uniformization Sampling for Continuous-Time Markov Chains (CTMCs). It calculates key metrics, including initialization cost (via auxiliary transition matrix R), expected recursion steps, and total CPU time for generating valid sample paths.

Main Components

- **Uniformization Rate (μ):** The maximum absolute value of diagonal entries of the rate matrix Q.
- **Auxiliary Transition Matrix (R):** Precomputed transition probabilities for the auxiliary chain.
- **Expected Recursion Steps ($\mathbb{E}[L]$):** Number of state transitions scaled by time T and the uniformization rate.
- **CPU Time:** Combines initialization and recursion costs.

Function Descriptions

1. `expected_recursion_steps_uniformization(Q, T, a, b, mu, R)`: Calculates the expected num-

ber of recursion steps for uniformization sampling using precomputed auxiliary probabilities.

2. `cpu_time_direct(Q, T, a, b, alpha, beta):` Computes the total CPU time for direct sampling, combining initialization and recursion costs.

3. `cpu_time_uniformization(Q, T, a, b, mu, R, alpha, beta):` Computes the total CPU time for uniformization sampling, combining initialization and recursion costs.

4. `plot_direct_sampling(Q, T_range, a, b, alpha, beta):` Plots figures for direct sampling: initialization time, recursion steps, and total CPU time.

5. `plot_uniformization_complexity(Q, T_range, a, b, alpha, beta):` Plots figures for uniformization sampling: initialization time, recursion steps, and total CPU time.

5.4 Comparison and Recommendation

General Recommendations

Overview: This section relates the computational complexities of the sampling methods to characteristics of the CTMC (Continuous-Time Markov Chain). The parameters α (initialization cost) and β (recursion cost per step) depend on the computational environment and the CTMC's state space size.

Methodology:

- For generality, the values of α and β are estimated using simulated reversible rate matrices across different state space sizes (5 to 100 states).
- The process involves:
  - Generating symmetric matrices S with exponentially distributed random off-diagonal entries.
  - Computing stationary distributions using the Dirichlet distribution.
  - Constructing rate matrices $Q_{ij} = S_{ij}\pi_j$ and validating results over multiple simulations.

Figure 8 Analysis:

- **Alpha (α) vs. State Space Size:**
  - For rejection sampling, α remains constant.
  - Direct sampling and uniformization exhibit non-linear growth due to the eigenvalue decomposition of Q, which scales cubically with the state space size.
- **Beta (β) vs. State Space Size:**
  - Rejection sampling shows a near-constant β, while direct sampling and uniformization grow quadratically with the state size.

Key Observations:

- Initialization costs are lowest for rejection sampling.
- Uniformization and direct sampling have higher α due to eigenvalue decomposition and auxiliary matrix computation.
- Sparse codon rate matrices lead to smaller α for uniformization and direct sampling.

Predicting α and β for Sampling Strategies:

- **Utility:** Predictive models for α and β enable the selection of the most efficient sampling method.
  - For example, predictions for the HKY model show values for α and β that align closely with experimental data.
- **Comparison:** Uniformization and direct sampling are comparable in initialization costs but differ significantly in recursion costs.
  - Sparse structures (e.g., codon rate matrices) lower α and β.

Comparison for Moderately Large T

Inflation Factor (ν):

- The inflation factor quantifies how virtual transitions (auxiliary state changes) inflate recursion steps in uniformization.
- $\nu = \frac{\max_c Q_c}{\sum_c \pi_c Q_c}$, where Q_c is the diagonal entry for state c.

Approximation:

- For large T, the number of recursion steps scales as:
  - $\mathbb{E}[L] = T$ for rejection and direct sampling.
  - $\mathbb{E}[L] = \nu T$ for uniformization.

Efficiency Trade-Offs:

- Uniformization is preferred when virtual transitions are minimal (ν is small).
- Rejection sampling is efficient for high acceptance probabilities (p_{acc}).

Deriving Recommendations:

- **Critical Efficiency Thresholds:**
  - Uniformization is more efficient than rejection sampling if:

$$p_{\text{acc}} > \frac{\alpha_R + \beta_R T}{\alpha_U + \beta_U T \nu}.$$

  - Direct sampling is preferred if:

$$p_{\text{acc}} > \frac{\alpha_R + \beta_R T}{\alpha_D + \beta_D T}.$$

Applications to Models:

- **HKY Model:**
  - For $T = 2$, direct sampling and uniformization predictions match experimental CPU times closely.
- **HKY + CpG Model:**
  - Direct sampling is optimal for large T, while uniformization excels in scenarios with sparse transitions.

6 Conclusion

This report provides an in-depth exploration of simulation techniques for endpoint-conditioned, continuous-time Markov chains (CTMCs) on a finite state space, with particular emphasis on applications to molecular evolution. The study meticulously evaluates three sampling algorithms: Modified Rejection Sampling, Direct Sampling, and Uniformization, considering their computational efficiency, complexity, and suitability for various levels of molecular data analysis, including nucleotide, codon, and sequence levels.

Through comprehensive implementation and simulation, we observed that while each algorithm has distinct advantages, their performance varies significantly depending on the specific application scenario. Uniformization demonstrated robust efficiency for large-scale simulations, whereas Direct Sampling excelled in precision under computational constraints. Modified Rejection Sampling, despite its simplicity, provided a reliable baseline approach.

The analysis of computational complexity highlights the trade-offs between accuracy and efficiency, emphasizing that the computational cost of each method is a function of the state space structure, initialization time, and recursion requirements. Predicting parameters such as α and β allows practitioners to preselect the most efficient sampling strategy, ensuring that computational resources are optimally allocated. For specific CTMC models, the recommendations derived from α and β align closely with observed CPU times, validating the theoretical framework and reinforcing its practical applicability.

By integrating theoretical foundations with empirical results, this work lays a strong foundation for leveraging CTMCs in molecular evolution studies and provides actionable insights for selecting sampling strategies tailored to diverse computational challenges.