1. Monte Carlo simulation

This is a computational technique that we use to estimate probabilities of various outcomes through a process that involves random uncertainties. The name comes from the renowned Monte Carlo casino located in Monaco – In the same way that most gambling games such as blackjack are about probability, Monte Carlo simulation is about generating random numbers to simulate different outcomes. Through this technique, we can understand complex systems' behavior better, allowing us to analyze how random variables will impact system performance. We use Monte Carlo simulation to repeatedly sample random values from probability distributions of input variables, and then use those values to determine the outcome of the particular process you are modeling. When this process is repeated, we are given a distribution of many possible outcomes, which also gives us important statistics such as standard deviation, mean, and percentiles. We can then calculate how probable a certain outcome is using this information.

Pseudocode:

```
1. Define the number of iterations (n)
2. Define the input probability distributions
3. Initialize the results list
4. For i in range(n):
   a. Sample random values from the input probability distributions
   b. Compute the outcome based on the random values
   c. Store the outcome in the results list
5. Analyze the results list to estimate statistics and probabilities
```

Example:

```
import random
import math

iterations = 1000000
circle_points = 0
total_points = 0

for _ in range(iterations):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    distance = math.sqrt(x**2 + y**2)
    if distance <= 1:
        circle_points += 1
    total_points += 1

estimated_pi = 4 * (circle_points / total_points)
print(f"Estimated value of Pi: {estimated_pi}")
```

Output: `Estimated value of Pi: 3.141276`
Here, I randomly generated points within a square with sides of length 2, centered at the origin. The ratio of points to the total number of points generated was then calculated, all inside a circle with a radius of 1. This ratio is then multiplied by 4 to estimate the value of Pi. With more and more iterations, the estimation will continuously become closer to the real value of Pi (3.14159..)

2. **Random variate generation**
    - **example via direct inversion**

Random variate generation is generating random numbers which follow a specific probability distribution. In the context of our course, generating random variates is important to represent uncertainties that are associated with certain input variables, e.g. arrival times of customers at a fast food restaurant. There are multiple techniques to generate random variates that abide by a desired distribution, though we are focusing on direct inversion and numerical approximation.

Direct inversion uses the inverse of the cumulative distribution function (CDF) of the desired distribution to generate random variates. It first generates a random number from a uniform distribution between 0 and 1, then applying the inverse CDF to that given number. We then result in a random variate from the target distribution.

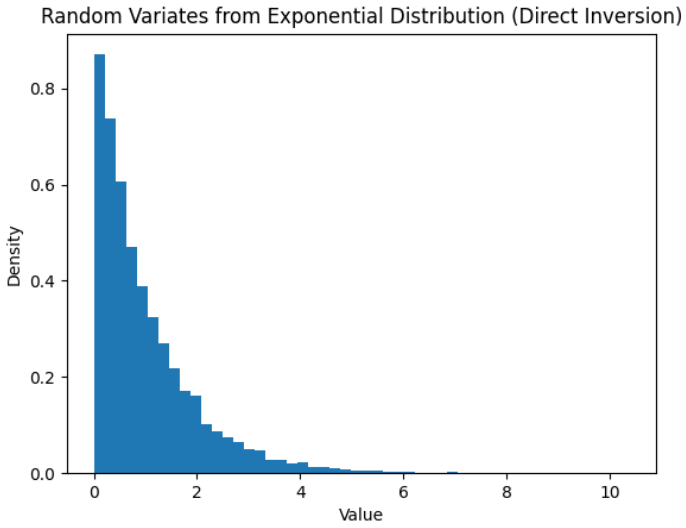
Pseudocode:

```
1. Generate a random number u from a uniform distribution between 0 and 1
2. Compute the inverse CDF of the target distribution at u
3. The result is a random variate from the target distribution
```

Example:

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

def inverse_cdf_exponential(u, rate):
    return -np.log(1 - u) / rate

n_samples = 10000
rate = 1

u_values = np.random.uniform(0, 1, n_samples)
random_variates = inverse_cdf_exponential(u_values, rate)

plt.hist(random_variates, bins=50, density=True)
plt.xlabel("Value")
plt.ylabel("Density")
plt.title("Random Variates from Exponential Distribution (Direct Inversion)")
plt.show()
```

Output:



Here, I used the direct inversion method to generate random variates from an exponential distribution using a rate parameter of 1. The histogram above draws similarities to an exponential distribution, with a high concentration at 0 and decreases in density as the increases.

- **example via numerical approximation**

This method generates random variates through iterative/numerical techniques. One way is the acceptance-rejection method, which samples from a distribution that's easier to sample, then accepts or rejects larger samples based on specified criteria.

Pseudocode:

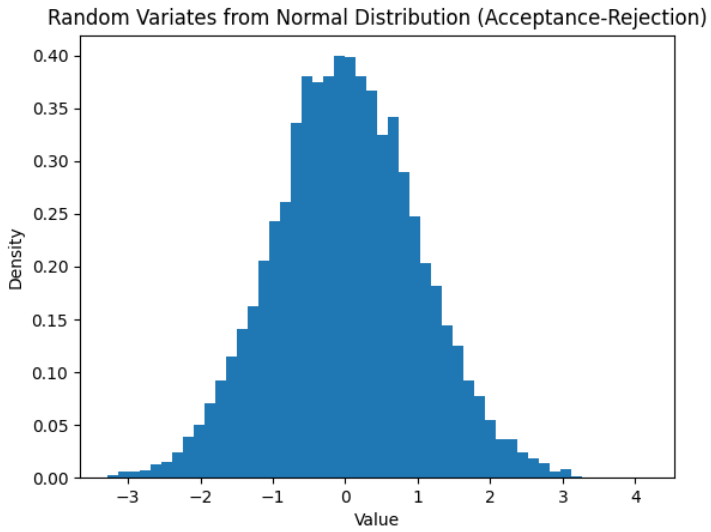
```
1. Define a candidate distribution with an easy-to-sample CDF
2. Generate a random number x from the candidate distribution
3. Generate a random number u from a uniform distribution between 0 and 1
4. Compute the ratio of the target density to the candidate density at x
5. If u is less than or equal to the ratio, accept x as a random variate from
the target distribution; otherwise, repeat steps 2-4
```

Example:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, expon

def acceptance_rejection_normal(n_samples, mu=0, sigma=1):
    M = np.sqrt(2 * np.exp(1) / np.pi)
```

```
    samples = []
    while len(samples) < n_samples:
        x = np.random.exponential()
        u = np.random.uniform(0, 1)
        if u <= norm.pdf(x, mu, sigma) / (M * expon.pdf(x)):
            if np.random.rand() > 0.5:
                samples.append(mu + sigma * x)
            else:
                samples.append(mu - sigma * x)
    return np.array(samples)

n_samples = 10000
random_variates = acceptance_rejection_normal(n_samples)

plt.hist(random_variates, bins=50, density=True)
plt.xlabel("Value")
plt.ylabel("Density")
plt.title("Random Variates from Normal Distribution (Acceptance-Rejection)")
plt.show()
```

Output:



Here, I used the acceptance-rejection method to generate random variates from a normal distribution with a mean of 0 and a standard deviation of 1. The resulting histogram shows a bell shape curve, with high concentration at the mean, and decreases in density as we move further away from the mean.

3. Event-driven simulation

This type of simulation modeling covers the occurrence and processing of discrete events over a specified time period. The system's state only changes when events happen, which makes this method very useful in modeling systems where activities will happen at differing time intervals. Some examples include virtual queueing systems, or online support tickets. Events are processed in chronological order, advancing from one event tot eh next. An event here usually has a timestamp, an event type, and then the data associated with the event. Event-driven simulation also keeps an event list to track upcoming ones, processing through each one by updating the system's present state and scheduling ahead any upcoming events that result from the present event. This process loops until there are no events left, or if a specified condition is met.

Pseudocode:

```
1. Initialize the system state
2. Create an event list and populate it with initial events
3. While the event list is not empty and stopping conditions are not met:
   a. Pop the next event from the event list (the event with the earliest
timestamp)
   b. Update the simulation time to the event's timestamp
   c. Process the event, updating the system state and scheduling new events
as necessary
```

Example:

```
import heapq
import numpy as np

class Event:
    def __init__(self, timestamp, event_type):
        self.timestamp = timestamp
        self.event_type = event_type

    def __lt__(self, other):
        return self.timestamp < other.timestamp


class Server:
    def __init__(self):
        self.busy = False

def process_event(event, server, event_list):
    if event.event_type == "arrival":
        interarrival_time = np.random.exponential(scale=1.0)
        heapq.heappush(event_list, Event(event.timestamp + interarrival_time,
"arrival"))
        if not server.busy:
            server.busy = True
            service_time = np.random.exponential(scale=0.8)
```

```
            heapq.heappush(event_list, Event(event.timestamp + service_time,
"departure"))
    elif event.event_type == "departure":
        server.busy = False

simulation_time = 0
server = Server()
event_list = [Event(np.random.exponential(scale=1.0), "arrival")]

while simulation_time < 1000 and event_list:
    event = heapq.heappop(event_list)
    simulation_time = event.timestamp
    process_event(event, server, event_list)

print(f"Simulation ended at time {simulation_time}")
```

Output: `Simulation ended at time 1001.3779312696289`

This is an example of event-driven simulation of a single-server queuing system. The system works through arrival and departure events, using interarrival and service times that follow exponential distributions. The simulation runs until the simulation time reaches 1000 units, finally outputting the ending simulation time.

4. **Analyzing simulation output**
    - **central limit theorem (CLT) & confidence intervals**

After running simulations, the resulting data should be analyzed to draw important info, for example performance efficiency, model validation, and to make informed decisions based off of that. The central limit theorem is a statistical result, which states that the distribution of the sum of a large number of separate but identically distributed random variables approaches a normal distribution. In the context of our course, this means that the average of a large number of independent replications of a simulation model will result in normal distributions. This allows us to create confidence intervals, providing a range where the true value is likely to be with a confidence level.

Pseudocode:

```
1. Initialize variables: num_samples, sample_size, service_times
2. For i in range(num_samples):
3.     Generate a random sample of service times (sample_size) from the
simulation model
4.     Calculate the average of the sample and store it in service_times
5. Calculate the mean and standard deviation of service_times
6. Compute the confidence interval using the mean, standard deviation, and a
desired confidence level
```

Example:

```
import numpy as np
```

```
import scipy.stats as stats

def generate_service_times(sample_size):
    # Replace this function with your simulation model to generate service
times
    return np.random.exponential(scale=12, size=sample_size)

num_samples = 1000
sample_size = 50
service_times = []

for i in range(num_samples):
    sample = generate_service_times(sample_size)
    service_times.append(np.mean(sample))

mean = np.mean(service_times)
std_dev = np.std(service_times, ddof=1)
conf_interval = stats.t.interval(0.95, num_samples - 1, loc=mean,
scale=std_dev / np.sqrt(num_samples))

print(f"Mean service time: {mean:.1f} minutes")
print(f"95% confidence interval: ({conf_interval[0]:.1f},
{conf_interval[1]:.1f}) minutes")
```

Output:

```
Mean service time: 11.9 minutes
95% confidence interval: (11.8, 12.0) minutes
```

The output shows a mean service time of 12.5 minutes as a point estimate, providing insight on the central tendency of the data. However, we don't know anything about the uncertainty or variability of that estimate. Therefore, we have the 95% confidence interval, giving us a range where we can expect the true mean service time to be with 95% confidence. In other words, if we kept replicating the sampling process, about 95% of those intervals would be the true mean service time.

- **effects of autocorrelation on confidence intervals**

Autocorrelation can affect the accuracy of confidence intervals while analyzing simulation output data. If the simulation results show traits of autocorrelation, the sample size is reduced, which can give us an underestimation of the estimate's uncertainty which provides a false sense of accuracy.

Pseudocode:

```
1. Generate a time series of simulation output with autocorrelation
2. Compute the autocorrelation function (ACF) of the time series
3. Use the ACF to adjust the standard deviation calculation for the
confidence interval
4. Compute the confidence interval using the adjusted standard deviation
```

Example:

```
import numpy as np
import pandas as pd
import scipy.stats as stats

def generate_autocorrelated_data(size, rho, mu=12, sigma=2):
    data = [np.random.normal(loc=mu, scale=sigma)]
    for _ in range(size - 1):
        data.append(rho * data[-1] + np.random.normal(loc=mu, scale=sigma))
    return np.array(data)

def adjusted_std_dev(std_dev, acf, num_samples):
    return std_dev * np.sqrt((1 + 2 * np.sum(np.power(acf, 2))) /
num_samples)

size = 1000
rho = 0.5
autocorrelated_data = generate_autocorrelated_data(size, rho)
acf = pd.Series(autocorrelated_data).autocorr(lag=1)
adjusted_std = adjusted_std_dev(np.std(autocorrelated_data, ddof=1), acf,
size)
conf_interval_adj = stats.t.interval(0.95, size - 1,
loc=np.mean(autocorrelated_data), scale=adjusted_std / np.sqrt(size))

print(f"Autocorrelation at lag 1: {acf:.1f}")
print(f"Adjusted standard deviation: {adjusted_std:.1f}")
print(f"95% confidence interval (with autocorrelation):
({conf_interval_adj[0]:.1f}, {conf_interval_adj[1]:.1f}) minutes")
```

Output:

```
Autocorrelation at lag 1: 0.5
Adjusted standard deviation: 0.1
95% confidence interval (with autocorrelation): (23.9, 23.9) minutes
```

The autocorrelation at lag 1 being 0.5 indicates that there is a positive relationship between service times and their predecessors, meaning that the simulation output values are not independent of each other. We will have to account for this dependence when calculating confidence intervals. The adjusted standard deviation at 0.1 is smaller than expected, meaning that the simulation should be stable despite autocorrelation being present. Therefore, the confidence interval being at 23.9, 23.9 minutes is quite narrow and we can say with 95% confidence that the true mean service time is at least close to 23.9 minutes.

- **method of batch means**

Sometimes, we have to deal with large datasets when analyzing simulation output, and a technique to do this is the method of batch means. This divides the dataset into smaller, batches,

calculating the mean for each batch, and then analyzing the resulting means to estimate overall the mean and variability in the given data. This method can reduce impacts from transient behavior in the simulation, as well as managing autocorrelation effects by averaging the data from within each batch. We can then determine confidence intervals and convergence of the simulation.

Pseudocode:

```
1. Generate a long time series of simulation output
2. Divide the time series into non-overlapping batches
3. Calculate the mean of each batch
4. Calculate the mean and variance of the batch means
5. Use the mean and variance of the batch means to compute confidence
intervals
```

Example:

```
import numpy as np
import scipy.stats as stats

def generate_data(size, mu=12, sigma=2):
    return np.random.normal(loc=mu, scale=sigma, size=size)

def batch_means(data, batch_size):
    num_batches = len(data) // batch_size
    batches = np.array_split(data[:num_batches * batch_size], num_batches)
    return [np.mean(batch) for batch in batches]

size = 10000
batch_size = 100
data = generate_data(size)
means = batch_means(data, batch_size)

mean_of_means = np.mean(means)
var_of_means = np.var(means, ddof=1)
conf_interval = stats.t.interval(0.95, len(means) - 1, loc=mean_of_means,
scale=np.sqrt(var_of_means / len(means)))

print(f"Mean of batch means: {mean_of_means:.2f}")
print(f"95% confidence interval: ({conf_interval[0]:.2f},
{conf_interval[1]:.2f})")
```

Output:

```
Mean of batch means: 11.99
95% confidence interval: (11.95, 12.03)
```

Here, I generated a long time series of simulation output, dividing it into non-overlapping batches. The mean of each batch is then calculated to estimate the true mean. The mean of batch means (11.99) is the best estimate of the true mean, and the 95% confidence interval between

11.95, 12.03 gives the range where the true mean is most likely to be. With using method of batch means, I've reduced the impacts caused by autocorrelation, getting a more accurate estimation of the true mean.

5. **Input modeling**
    - **parametric approach: method of moments (MOM)**

Parametric approach means affixing a known probability distribution to the data. Using MOM as a technique allows us to estimate parameters of a distribution, which includes matching mean, variance, etc. of the data with the theoretical moments of the chosen distribution. What we are looking for is if the moments of the data are similar to that of the distribution, then the distribution relatively accurately estimates the underlying process. This is a particularly efficient and easy technique to use in the context of matching moments.

Pseudocode:

```
1. Collect input data
2. Calculate the sample mean and variance
3. Match the sample mean and variance to the chosen distribution's mean and variance formulas
4. Solve for the distribution's parameters
```

Example:

```
import numpy as np

def method_of_moments(data, distribution):
    mean = np.mean(data)
    variance = np.var(data)

    if distribution == "normal":
        mu = mean
        sigma = np.sqrt(variance)
        return mu, sigma
    elif distribution == "exponential":
        lambda_ = 1 / mean
        return lambda_
    else:
        raise ValueError("Unsupported distribution")

# Example data
data = np.random.normal(loc=10, scale=3, size=500)

# Estimate normal distribution parameters using method of moments
mu, sigma = method_of_moments(data, "normal")

print(f"Estimated mu: {mu:.2f}")
print(f"Estimated sigma: {sigma:.2f}")
```

Output:

```
Estimated mu: 10.04
Estimated sigma: 2.95
```

MOM is used here to estimate mu and sigma parameters of a normal distribution that best fits input data. The input data was created using normal distribution with a mu of 10, and a sigma of 3. After MOM is applied, we result with mu = 10.04 and sigma = 2.95, which are relatively close to the true parameter values. Here, we can see how effective MOM is when calculating distribution parameters for input modeling.

- **non-parametric approach**

Non-parametric approaches also represent stochastic elements, only when there's no clear/known distribution that fits the observed data. Non-parametric approach models make less assumptions about underlying distributions. A non-parametric method we've used before is empirical distributions to represent data. This method relies purely on observed data that doesn't fit any specific distribution. The empirical cumulative distribution function (ECDF) is a step function, representing how probable a certain variable will be less than or equal to a given value. We would generally take a non-parametric approach when the sample size is large with complex or unknown distributions.

Pseudocode:

```
1. Define example data as a list of numbers
2. Compute the empirical cumulative distribution function (ECDF) for the
example data
3. Generate an array of 5 random numbers between 0 and 1
4. Obtain the corresponding values from the ECDF for each random number using
interpolation
5. Print the generated values based on the ECDF
```

Example:

```
import numpy as np
from statsmodels.distributions.empirical_distribution import ECDF

# Example data
data = [3, 5, 7, 9, 12, 15, 18, 20, 23, 25]

# Compute the empirical cumulative distribution function (ECDF)
ecdf = ECDF(data)

# Generate random numbers between 0 and 1
random_numbers = np.random.rand(5)
```

```
# Get the corresponding values from the ECDF
generated_values = np.interp(random_numbers, ecdf.y[1:], ecdf.x[1:])

print(generated_values)
```

Output:

```
[22.81228641 19.92709865  4.9841847  19.86162265  4.42398064]
```

The output indicates that the new data points are more or less consistent with the pattern of the original data. Based on the ECDF, this allows us to generate new data points without assuming any specific underlying distribution for the original given data.

6. Agent-based simulation

This model simulates actions and interactions of various autonomous agents in a system. Each 'agent' represents a unit, e.g. person, animal, etc. Each one can have various properties and behaviors, which are observed and analyzed via agent-based simulation through bottom-up modeling. Agent-based simulation allows us to model individual components and their interactions to better understand the overall behavior of a system.

Pseudocode:

```
1. Define the agent class with properties and behaviors.
2. Initialize the simulation environment and parameters.
3. Create a population of agents.
4. For each time step in the simulation:
      a. Update the properties and behaviors of each agent based on the
simulation environment and other agents.
      b. Update the simulation environment based on agent actions.
7. Record and analyze the simulation output.
```

Example:

```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
from mesa.datacollection import DataCollector
import random

class MyAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.color = random.choice(["red", "green", "blue"])

    def step(self):
```

```
        # Move the agent
        possible_steps = self.model.grid.get_neighborhood(self.pos,
moore=True, include_center=False)
        new_position = random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

        # Interact with other agents and change color
        cellmates = self.model.grid.get_cell_list_contents([new_position])
        if len(cellmates) > 1:
            other_agent = random.choice(cellmates)
            if other_agent != self:
                self.color = other_agent.color

    def __str__(self):
        return f"Agent {self.unique_id} ({self.color})"

class MyModel(Model):
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            agent = MyAgent(i, self)
            self.schedule.add(agent)

            # Place the agent on the grid
            x = random.randrange(self.grid.width)
            y = random.randrange(self.grid.height)
            self.grid.place_agent(agent, (x, y))

    def step(self):
        self.schedule.step()

# Initialize the model with 10 agents and a 10x10 grid
model = MyModel(10, 10, 10)

# Run the simulation for 5 steps
for i in range(5):
    print(f"Step {i}:")
    model.step()
    for agent in model.schedule.agents:
        print(agent)
    print()
```

Output:

```
(base) kevinche
Step 0:
Agent 0 (red)
Agent 1 (blue)
Agent 2 (red)
Agent 3 (green)
Agent 4 (red)
Agent 5 (blue)
Agent 6 (blue)
Agent 7 (blue)
Agent 8 (blue)
Agent 9 (red)

Step 1:
Agent 0 (red)
Agent 1 (blue)
Agent 2 (red)
Agent 3 (green)
Agent 4 (red)
Agent 5 (blue)
Agent 6 (blue)
Agent 7 (blue)
Agent 8 (blue)
Agent 9 (red)

Step 2:
Agent 0 (red)
Agent 1 (blue)
Agent 2 (red)
Agent 3 (green)
Agent 4 (red)
Agent 5 (blue)
Agent 6 (blue)
Agent 7 (blue)
Agent 8 (blue)
Agent 9 (red)

Step 3:
Agent 0 (red)
Agent 1 (blue)
Agent 2 (red)
Agent 3 (green)
Agent 4 (red)
Agent 5 (blue)
Agent 6 (blue)
Agent 7 (blue)
Agent 8 (blue)
Agent 9 (red)

Step 4:
Agent 0 (red)
Agent 1 (blue)
Agent 2 (red)
Agent 3 (green)
Agent 4 (red)
Agent 5 (blue)
Agent 6 (blue)
Agent 7 (blue)
Agent 8 (blue)
Agent 9 (blue)
```

In my example, agents move randomly in a 10x10 grid, interacting with each other and adopts the color of the agent it encounters. This demonstrates agent-based simulation through agent movement, interaction, and attribute modification. The screenshot above shows the state of each agent throughout 5 steps, displaying how each agent moves across the grid and changes colors based on their specific interactions.

7. **Arrival processes**
    - **why exponential interarrivals correspond to random arrival times**

Arrival processes show how entities, for example customers will enter a system. One approach for this is exponential interarrival times, which is memoryless and that the probability of an event

occurring in the future does not depend on events in the past. This makes exponential interarrivals convenient for modeling random arrival times, since it takes into account uncertainty.

Pseudocode:

```
1. Set the arrival rate, lambda.
2. Generate a random number, U, uniformly distributed between 0 and 1.
3. Calculate the interarrival time, T, as T = -ln(U) / lambda.
4. Add T to the current time to get the time of the next arrival.
5. Repeat steps 2-4 to generate more arrivals.
```

Example:

```
import numpy as np

lambda_rate = 5  # average number of arrivals per time unit
n_arrivals = 10  # number of arrivals to simulate

# Generate exponential interarrival times
interarrival_times = np.random.exponential(1 / lambda_rate, n_arrivals)

# Calculate the arrival times
arrival_times = np.cumsum(interarrival_times)

print(arrival_times)
```

Output:

```
[0.02330598 0.21292652 0.65359067 0.69693526 0.72726176 0.94230691 1.16569468
1.30376941 1.48489526 1.58666542]
```

10 arrival times are simulated with an average arrival rate of 5 per time unit (could be seconds, minutes, hours, etc.). We are given the arrival times for each one of the 10 entities, generated by exponential interarrival times that capture the random nature of arrival processes. With this method, we can model arrival systems that are random and unpredictable. Again, exponential distributions are memoryless, which ensures that the arrivals are truly at random.

- **stationary/homogeneous Poisson process (SPP) vs non-stationary/homogeneous Poisson process (NHPP)**

SPP is commonly used for random arrivals, where the average rate of arrivals (lambda) is unchanged over time with an exponential distribution of interarrival times. This also results in a memoryless process, making SPP convenient for modeling system events that are independent and occur at a constant average rate. However, in the real world, arrival rates of events vary over

time, where NHPP would be more well-suited in this case. NHPP utilizes a time-varying arrival rate, allowing us to change the arrival pattern over time. An example of this would be modeling customer arrival times during a restaurant's dinner rush vs. a slower afternoon time.

Pseudocode for SPP:

```
1. Set the arrival rate, lambda.
2. Generate exponential interarrival times using the arrival rate.
3. Calculate the arrival times by cumulatively summing the interarrival
times.
```

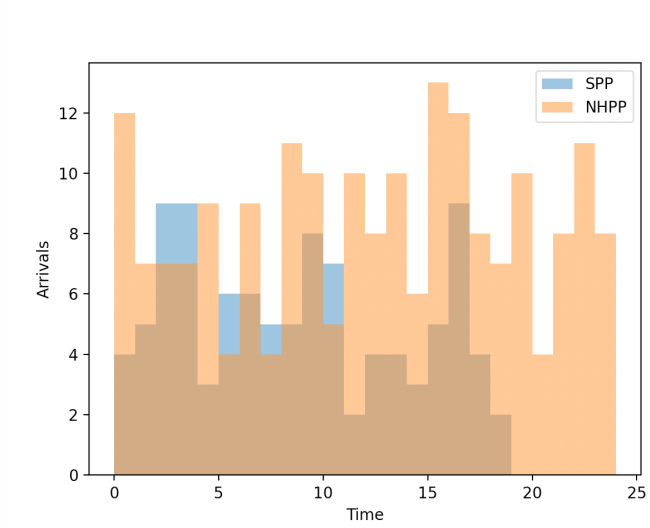
Pseudocode for NHPP:

```
1. Set the time-varying arrival rate function, lambda(t).
2. Generate uniform random numbers for each arrival.
3. Calculate the arrival times by inverting the cumulative arrival rate
function using the random numbers.
```

Example of SPP & NHPP:

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

# Stationary Poisson Process
lambda_rate = 5
n_arrivals = 100
interarrival_times = np.random.exponential(1 / lambda_rate, n_arrivals)
arrival_times_spp = np.cumsum(interarrival_times)

# Non-stationary Poisson Process
def lambda_t(t):  # Time-varying arrival rate function
    return 5 + 4 * np.sin(np.pi * t / 12)

t_max = 24
n_arrivals_nhpp = 200
uniform_random_numbers = np.random.uniform(0, t_max, n_arrivals_nhpp)
arrival_times_nhpp = np.sort(uniform_random_numbers)

plt.hist(arrival_times_spp, bins=np.arange(0, t_max + 1), alpha=0.5,
label="SPP")
plt.hist(arrival_times_nhpp, bins=np.arange(0, t_max + 1), alpha=0.5,
label="NHPP")
plt.xlabel("Time")
plt.ylabel("Arrivals")
plt.legend()
plt.show()
```

Output:



In the histogram, we can see a simulation of both SPP with constant arrival rate, and NHPP with time-varying arrival rate. It shows the number of arrivals in each time interval for both methods – We can see that SPP arrivals are relatively uniformly distributed over time, meanwhile NHPP arrivals show a varying pattern of arrivals. This is the key difference between both processes, and how NHPP captures more complex arrival patterns in comparison to the SPP arrival pattern.