Members: Andrei Shchapaniak (shchaand), Maksim Spiridonov (spirimak)

Members: Andrei Shchapaniak (shchaand), Maksim Spiridonov (spirimak)

`Problem description`

Consider a a queuing model $M|G|\infty$

Requests arrive according to a Poisson process with intensity $\lambda = 10s^{-1}$

The service time for each request (in seconds) follows a Gamma distribution $S \sim Ga(4, 2)$, i.e., Gamma with parameters $a = 4$ and $p = 2$

The inter-arrival times and service times are independent.

The system has (theoretically) infinitely parallel service channels (each incoming request is immediately serviced).

Let N_t denote the number of customers in the system at time t. Assume that the system is initially empty, i.e., $N_0 = 0$

1. `Simulate one trajectory` $N_t(\omega)|t \in (0, 10s)$

`Graphically represent the trajectory. Describe the principle of generating this trajectory properly!`

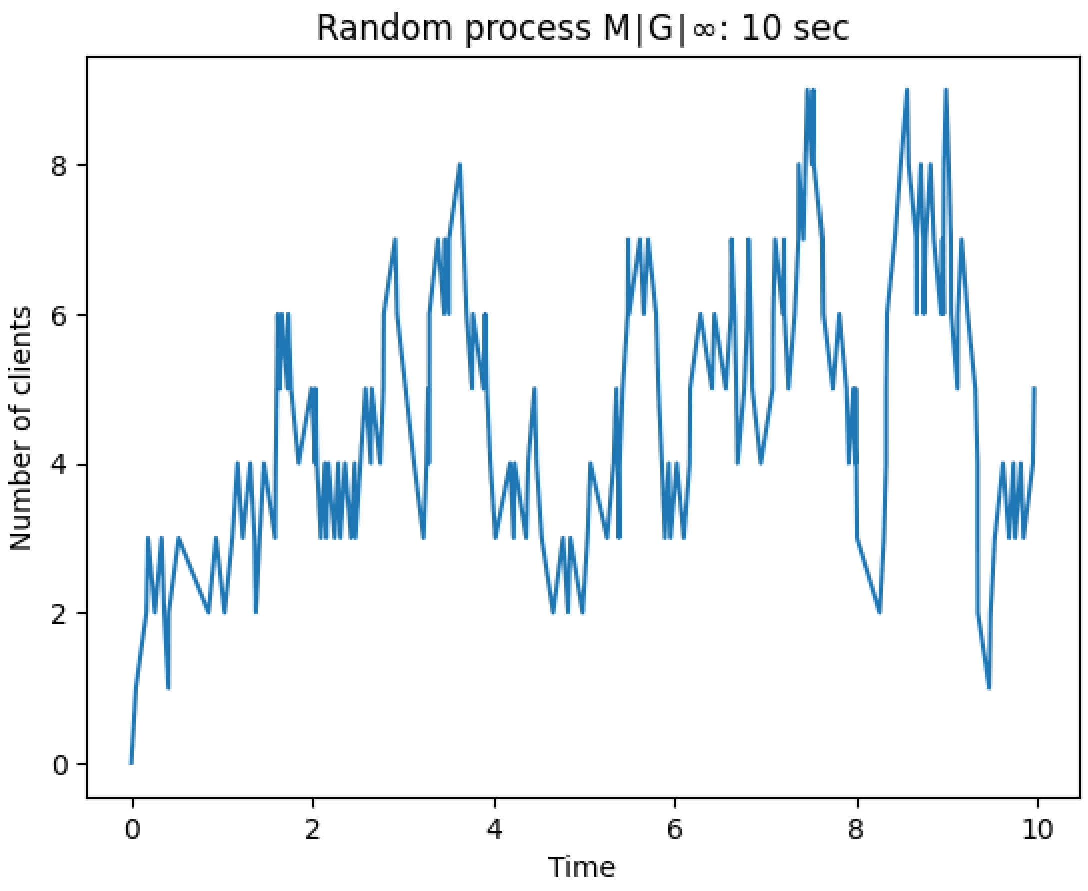
The system $M|G|\infty$ corresponds to a nonhomogeneous Poisson process (lecture 23)

Now with the use of function **create_trajectory** we can create a trajectory of a Poisson process with service, where the inter-arrival intervals are exponentially distributed with a rate parameter **lambda**, and the service lengths follow a gamma distribution with shape parameter **shape_** and scale parameter **scale_**. There we utilize parameter **T** which is the whole duration of observing the process.

The function generates a trajectory of a Poisson process with service, ensuring it doesn't exceed the total time T. It starts at time 0, adds arrivals and their service times until T is reached. Arrival times are exponential with rate lambda_, service times are gamma with parameters shape_ and scale_. The result is a sorted list of timestamp-value pairs, where arrivals have value 1 and service ends have value -1.

```
def create_trajectory(T):
  process = [[0, 0]]
  t = 0
```

```
while t < T:
  t_delta = expon.rvs(scale=1/lambda_)
  if t + t_delta > T:
    break

  process.append([t + t_delta, 1])
  t_service = gamma.rvs(shape_, scale=scale_)
  if t + t_delta + t_service <= T:
    process.append([t + t_delta + t_service, -1])
  t += t_delta

process = np.array(process)
return process[np.argsort(process[:, 0])]
```



On the following graph, created with the plot function, we can see the number of clients on the y axis and the time on x axis. The y axis describes the changes in the Poisson process.

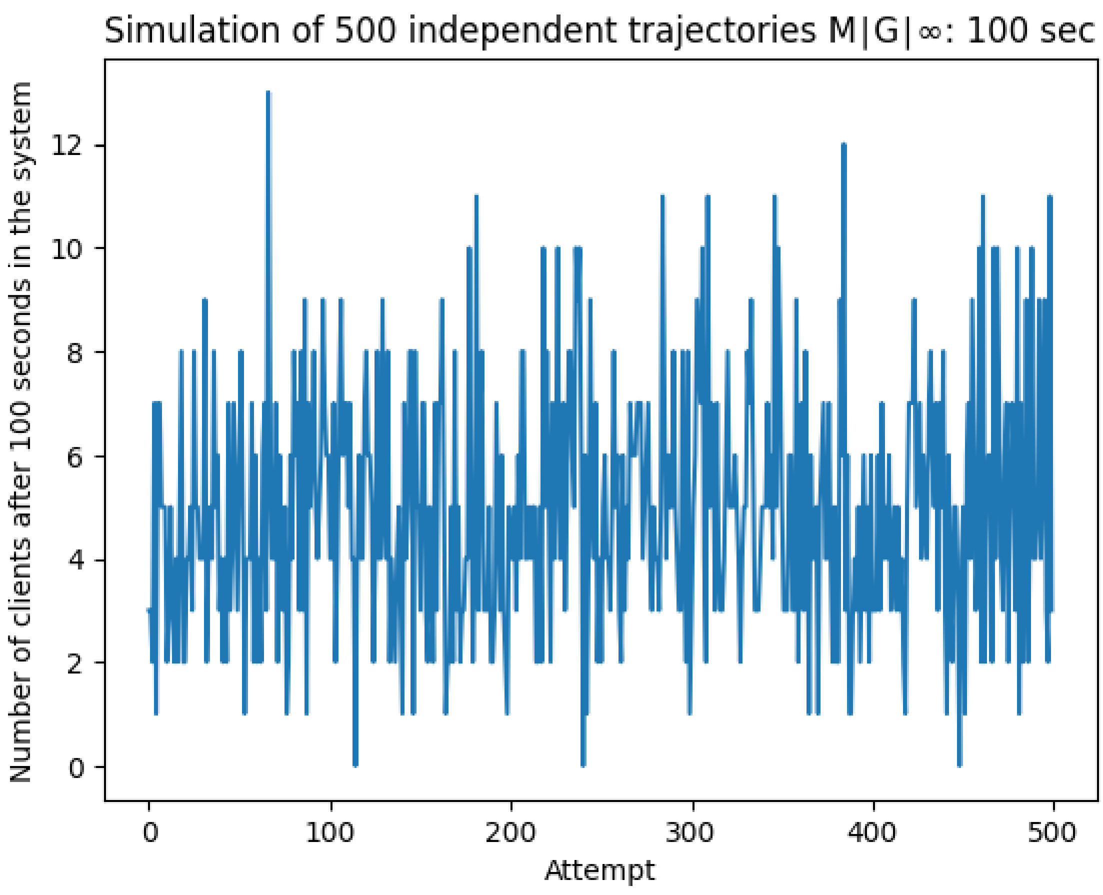
2. `Simulate n = 500 independent trajectories for`$ t∈(0,100)$

`Based on these simulations, estimate the distribution of the random variable N_100.`

With the use of the followinng **simulate_trajectories** function we can simulate **N** random trajectories based on the given parameters N=500, T=100.

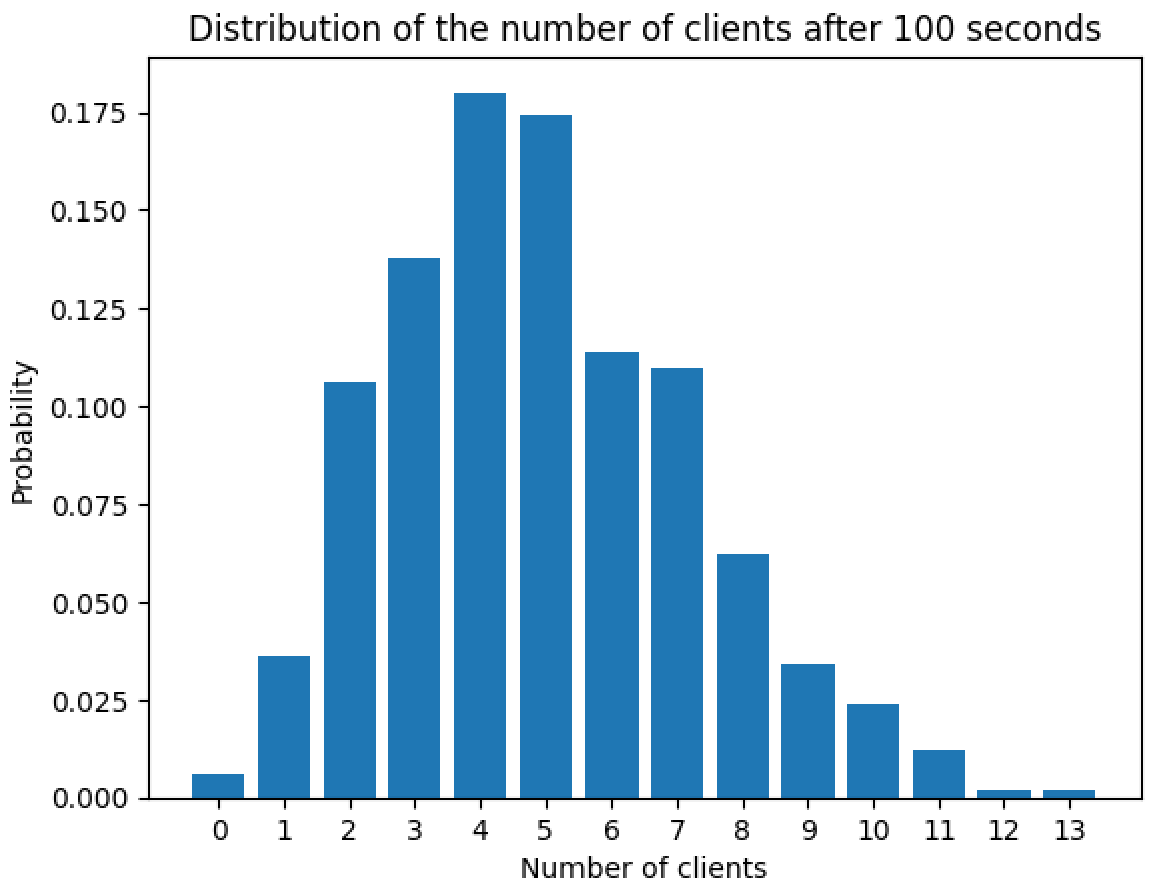
```
def simulate_trajectories(N=500, T=100):
  processes_results = np.zeros(N)
  for i in np.arange(N):
    trajectory = create_trajectory(T)
    processes_results[i] = np.sum(trajectory[:, 1])

  return processes_results
```

In the similar manner to the previous task, we can plot the resultig trajectories.



We can also use a bar plot to show the resulting trajectories, plotting the distribution of the number of clients after 100 seconds. It can be seen that the resulting plot is very similar to Normal distribution.

```
def plot_clients_distr(distr, T=100):
  plt.bar(np.arange(len(distr)), distr)
  plt.title(f"Distribution of the number of clients after {T} seconds")
  plt.xticks(np.arange(len(distr)))
  plt.xlabel('Number of clients')
  plt.ylabel('Probability')
```



3. `Discuss the limiting distribution of this system for` $t \to \infty$ `(see lecture 23).Using an appropriate test at the 5% significance level, check whether the simulation results for N_100 correspond to this distribution.`

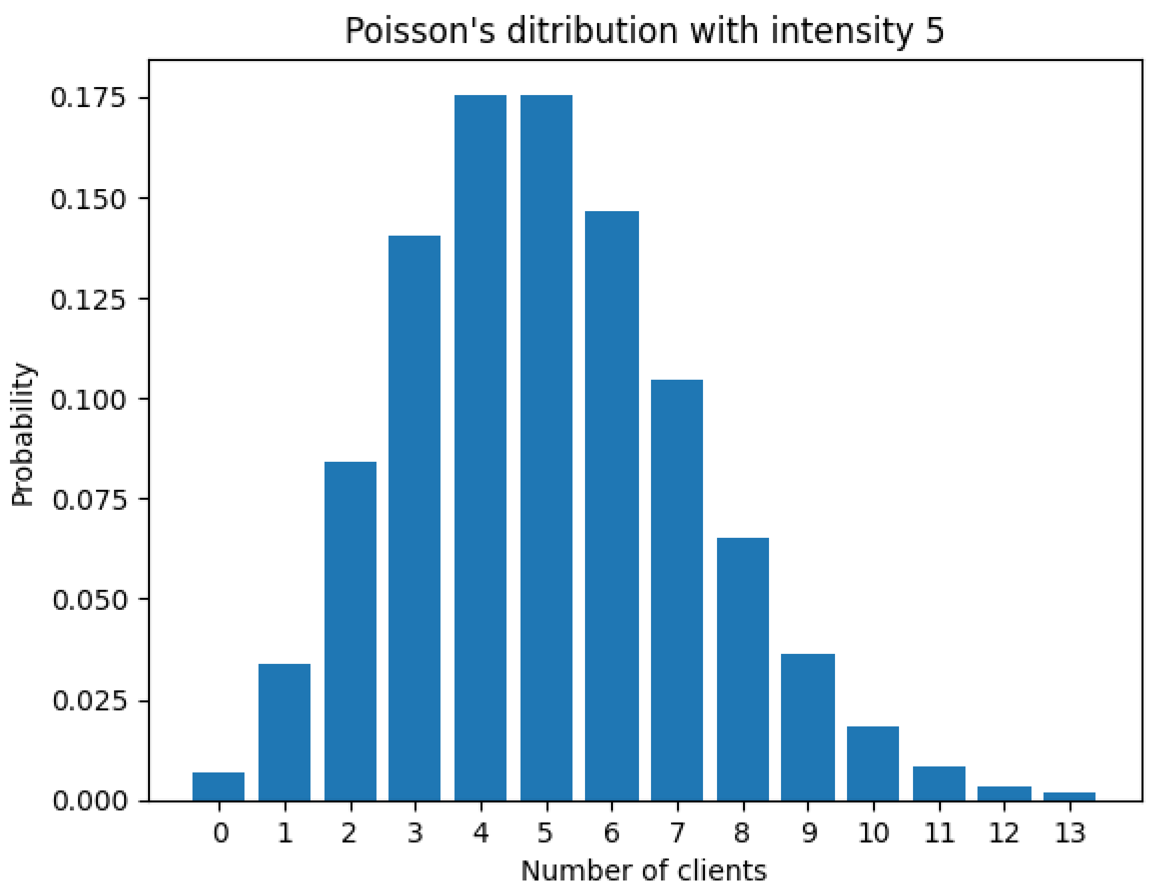
At fist we need to calculate the intensity. The arrivals λ follow a Poisson distribution. Service time itself follows a Gamma distribution $Ga(shape, scale)$ - in the case we have - $Ga(4, 2)$.
We have $\lambda = 10$
$ES = 1\mu = p/a$
$\to \mu = 1/ES = a/p$
$\mu = 4/2 = 2$
$\lambda/\mu = 5$

The **poisson_distr** function generates Poisson probabilities based on a given intensity parameter. It iterates through possible occurrences, calculates their probabilities, and returns an array of these probabilities, ensuring their sum equals 1.

```
def poisson_distr(intensity, len_poisson_probs, size_processes_results):
  poisson_probs = np.zeros(size_processes_results, dtype=float)
  for i in np.arange(len_poisson_probs):
    poisson_probs[i] = poisson.pmf(i, 5.0)
  poisson_probs[-1] = 1 - np.sum(poisson_probs[:-1])

  return poisson_probs
```

We can obtain the following result using $\lambda = 5$:
[0.00673795 0.03368973 0.08422434 0.1403739 0.17546737 0.17546737
0.14622281 0.10444486 0.06527804 0.03626558 0.01813279 0.00824218
0.00343424 0.00201885]

We can also plot it:



In order to complete the task №3, we utilize the class **PearsonsTest**.

It implements Pearson's chi-squared test, a statistical method for evaluating the goodness of fit between observed and expected frequency distributions. It allows for the comparison of observed and expected distributions by calculating the test statistic, degrees of freedom, critical value, and p-value. Additionally, it includes a normalization step to ensure the validity of the test. Finally, it provides a comparison with the chisquare function from scipy.stats for validation purposes.

```
class PearsonsTest:
  def __init__(self, clients_distr, poisson_distr):
    self.clients_distr = clients_distr
    self.poisson_distr = poisson_distr
    self.sets_array = []

  def normalize_table(self):
    while any(poisson_prob < 5 for _, _, poisson_prob in self.sets_array):
      sorted_array = sorted(self.sets_array, key=lambda x: x[1])
      smallest = sorted_array[:2]
      combined_letter = smallest[0][0] + smallest[1][0]
      combined_expected = smallest[0][1] + smallest[1][1]
      combined_actual = smallest[0][2] + smallest[1][2]

      self.sets_array.append((combined_letter, combined_expected,
combined_actual))

      self.sets_array.remove(smallest[0])
      self.sets_array.remove(smallest[1])

    print(self.sets_array)

  def test(self, alpha, normalize_flag):
    for num_client, (client_p, possion_p) in
enumerate(zip(self.clients_distr, self.poisson_distr)):
      self.sets_array.append((num_client, client_p, possion_p))

    print(self.sets_array)
    if normalize_flag:
      self.normalize_table()

    test_val = 0.0
    for _, np, N in self.sets_array:
      test_val += (N - np)**2 / np

    ddf = len(self.sets_array) - 1
    critical_value= chi2.isf(alpha, ddf)
    p_value = chi2.sf(test_val, ddf)
    print(f'Manual values: ddf: {ddf}, p_value: {p_value}, test_val:
{test_val}, critical_value: {critical_value}')

    print(chisquare([s[2] for s in self.sets_array], [s[1] for s in
self.sets_array]))
```

We test the null hypothesis H_0 that **the results of the simulation are equal to the limiting distribution** and the alternative hypothesis H_A that **it is not equal to the limiting distribution**.

The obtained distribution is the following:
[(0, 3.0, 3.3689734995427334),
(1, 18.0, 16.84486749771367),
(2, 53.0, 42.11216874428416),
(3, 69.0, 70.18694790714025),
(4, 90.0, 87.73368488392532),
(5, 87.0, 87.73368488392533),

```
(6, 57.0, 73.1114040699377),
(7, 55.0, 52.22243147852697),
(8, 31.0, 32.63901967407933),
(9, 17.0, 18.132788707821856),
(10, 12.0, 9.066394353910926),
(11, 6.0, 4.1210883426867815),
(12, 1.0, 1.7171201427861613),
(13, 1.0, 1.0094258137187673)]
```

After normalization, we can get:

```
[(1, 18.0, 16.84486749771367),
(2, 53.0, 42.11216874428416),
(3, 69.0, 70.18694790714025),
(4, 90.0, 87.73368488392532),
(5, 87.0, 87.73368488392533),
(6, 57.0, 73.1114040699377),
(7, 55.0, 52.22243147852697),
(8, 31.0, 32.63901967407933),
(9, 17.0, 18.132788707821856),
(10, 12.0, 9.066394353910926),
(36, 11.0, 10.216607798734444)]
```

N is the whole number of events (second column) $n * p$ is sample range multiplied by its probability (third column).

The resultinng test statistic is the following:
$\chi^2 = \sum_{i=1}^{k} \frac{(N_i - np_i)^2}{(np_i}$

The obtained test statictic value is the following:
8.023860249608395

Manual values:

- Degrees of freedom: 10
- p_value: 0.62650619849802
- test_val: 8.023860249608395
- critical_value: 18.307038053275146

As a result, **8.024** $\geq$ **18.307**, so null hypothesis H_0 (that the results of the simulation are equal to the limiting distribution) is hereby accepted at the at the 5% significance level.

As the resulting $p - value$ is larger, than 0.05, we can thereby accept the null hypothesis H_0 at the at the 5% significance level.