Trabalho 1
Filas e Ruína do Apostador

Modelagem e Avaliação de Desempenho

Daniel Corcino de Albuquerque - DRE: 118188457
Letícia Freire Carvalho de Sousa - DRE: 118025324
Lucas Favilla Ferreira Alves da Silva - DRE: 119156518
Roberto Leonie Ferreira Moreira - DRE: 116062192

Prof. Daniel Sadoc Menasché

UFRJ - Universidade Federal do Rio de Janeiro

22 de setembro de 2023

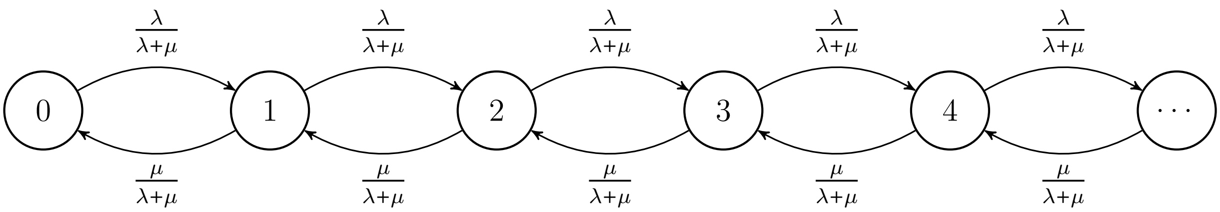
1 Introdução e Objetivos

Preliminarmente, o grupo implementou um simulador de fila $M/M/1$ com disciplina de atendimento *FIFO — First In, First Out* — e com chegadas de clientes obedecendo um processo *Poisson* com taxa λ e tempo entre chegadas exponencialmente distribuído com média $1/\lambda$.

Além disso, o atendimento do servidor do sistema possui uma taxa exponencial μ que leva a sua capacidade a ser modelada como uma variável aleatória (v.a.) exponencial com taxa μ e tempo médio de serviço $E(X) = 1/\mu$.

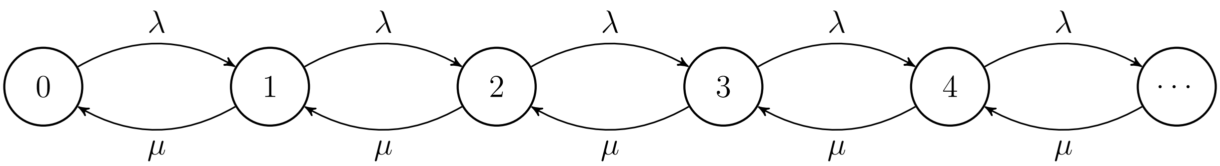
2 Objetivos

- De certa forma, a **ruína do apostador** se relaciona com uma fila $M/M/1$ simples. Seja a cadeia de Markov



que representa com probabilidade $\lambda/(\lambda+\mu)$ que **uma chegada ocorreu antes de uma partida** e com probabilidade $\mu/(\lambda+\mu)$ que **uma partida ocorreu antes de uma chegada**. Associamos a taxa λ às **chegadas** *Poisson* e μ às **partidas** do servidor com tempo de atendimento exponencialmente distribuído. Observe que esta cadeia possui estados **recorrentes** que podem ser visitados no futuro com probabilidade positiva.

Estamos interessados em obter a **distribuição estacionária** deste modelo para filas $M/M/1$. Multiplicando cada taxa por $(\lambda+\mu)$, obtemos a cadeia



e o sistema de equações

$$\begin{cases} \lambda\pi_k = \mu\pi_{k+1}, & k = 0, 1, 2, \ldots \\ \sum_{k=0}^{+\infty} \pi_k = 1 \end{cases} . \tag{1}$$

Seja $\rho = \lambda/\mu$. Resolvendo o sistema acima, temos

$$\pi_k = \rho^k \pi_0, \qquad k = 0, 1, 2, \ldots. \tag{2}$$

Substituindo no somatório do sistema, temos

$$\sum_{k=0}^{+\infty} \rho^k \pi_0 = 1$$

$$\pi_0 \sum_{k=0}^{+\infty} \rho^k = 1$$

$$\pi_0 \frac{1}{1-\rho} = 1, \qquad |\rho| < 1$$

$$\frac{\pi_0}{1-\rho} = 1, \qquad |\rho| < 1$$

$$\pi_0 = 1 - \rho, \qquad |\rho| < 1. \tag{3}$$

Para $|\rho| < 1$, temos que $\pi_k = \rho^k(1-\rho), \quad k = 0, 1, 2, \ldots$, assemelhando-se muito com uma densidade de uma **distribuição geométrica**! De fato, os estados associados à capacidade da fila $M/M/1$ comportam-se como uma distribuição geométrica com parâmetro ρ!

- Desejamos obter a **probabilidade da fila atingir o seu tamanho máximo** K, resultando em um transbordamento (*overflow*) do sistema antes de seu completo esvaziamento. Ou seja, deseja-se obter $P\{N \geq K\}$. Sabemos, para distribuições discretas, que

$$P\{N \geq K\} = 1 - P\{N < k\}.$$

Para $\rho < 1$, temos

$$P\{N \geq K\} = 1 - \sum_{i=0}^{k-1} \rho^k (1-\rho)$$

$$P\{N \geq K\} = 1 - (1-\rho) \sum_{i=0}^{k-1} \rho^k$$

$$P\{N \geq K\} = 1 - (1-\rho) \cdot (1 + \rho^1 + \rho^2 + \rho^3 + \cdots + \rho^{k-1})$$

$$P\{N \geq K\} = 1 - (1-\rho) \cdot \frac{(\rho^{k-1} \cdot \rho) - 1}{\rho - 1}$$

$$P\{N \geq K\} = 1 - (1-\rho) \cdot \frac{\rho^k - 1}{\rho - 1}$$

$$P\{N \geq K\} = 1 - \frac{\rho^k - \rho^{k+1} - \rho - 1}{\rho - 1}$$

$$P\{N \geq K\} = -\frac{2 + \rho^{k+1} + \rho^k}{\rho - 1}$$

$$P\{N \geq K\} = \frac{\rho^k(\rho + 1) + 2}{1 - \rho}, \rho < 1 \tag{4}$$

3 Desafios e suas Soluções

3.1 Desafios

3.1.1 Modelagem da Simulação

Nosso primeiro problema encontrado foi o dilema de como simular um evento contínuo de forma discreta. Assim, a ideia inicial se deu na utilização do método `poisson()` da biblioteca `numpy`, que por sua vez aceita os parâmetros de entrada taxa λ e um número inteiro x que dita o tamanho do retorno, que é um array de tamanho x contendo uma distribuição de *Poisson*. Entretanto, após realizarmos algumas simulações, encontramos resultados pouco satisfatórios, assim nos levando a outro desafio.

Uma vez percebido o problema anterior, nos vimos em um novo dilema de como simular eventos contínuos de chegada e saída de uma fila, já que, à época da problemática, não vislumbrávamos uma maneira de criar as chegadas sem a utilização do método `poisson` mencionado anteriormente, que por sua vez não trazia o dinamismo necessário para a execução do simulador.

3.1.2 Organização do código e retorno

Outra notável questão foi o contínuo e intenso rearranjo do código para se adaptar às novas necessidades do problema a cada funcionalidade que adicionávamos.

A cada refatoração, novos métodos e métricas foram surgindo, aumentando a complexidade do código, tornando-o um tanto quanto confuso de compreender, além da necessidade de corrigir o retorno do simulador, a fim de clarificar o acompanhamento da execução e do resultado final.

3.1.3 Condição de parada da simulação

Uma vez realizada a maior parte da programação necessária do trabalho, nos deparamos com um novo problema fundamental para a execução do código: **qual seria a condição de parada do programa**, uma vez que, dada natureza do problema, a simulação poderia executar por tempo indeterminado.

3.1.4 Validação dos resultados

Após a confecção do código e da correção de *bugs*, entramos na fase de **verificação e validação dos resultados** a fim de determinar a corretude da simulação, surgindo a dúvida de como fazer para garantir a funcionalidade e confiabilidade das respostas dadas pelo programa.

3.2 Soluções

3.2.1 Solução para a modelagem

A resolução do desafio se deu após intensa pesquisa, utilizando os vídeos recomendados como fonte e conversando com o professor doutor Daniel Sadoc Menasché, de forma a percebermos que o erro estava no chamamento do método `poisson()`.

Nos passou despercebido que o **tempo entre chegadas *Poisson* é exponencialmente distribuído**. Dessa forma, para o próximo evento de chegada, bastava apenas somar uma variável exponencial dada pelo método `random.expovariate()` ao tempo da última chegada. O mesmo problema ocorreu para o tempo de serviço, bastando somar ao tempo corrente uma variável aleatória (v.a.) exponencial para a obtenção do momento de encerramento do serviço de um cliente.

3.2.2 Solução de organização e correção

Utilizamos de conceitos aprendidos durante a graduação para encapsularmos para o código de maneira a deixá-lo mais legível, de boa confecção e manutenível, deixando assim as sessões de programação mais fluidas. As estruturas utilizadas neste tópicos serão exploradas a fundo durante este relatório.

3.2.3 Solução para a condição de parada

O grupo decidiu por estabelecer um **tempo limite** para o simulador. Desta maneira, uma vez passado como parâmetro, este tempo cria uma barreira para a criação de novos eventos de chegada (`arrivals`) no simulador.

Dessa forma, o programa irá terminar de servir eventos ainda na fila e encerrará sua execução assim que não houver mais eventos pendentes, retornando as **métricas** calculadas durante a simulação.

3.2.4 Solução para as métricas e resultados obtidos

A fim de nos certificarmos que as métricas retornadas estão corretas, utilizamos as expressões matemáticas fornecidas na bibliografia e apresentadas em aula, assim verificando que a maioria dos resultados estão seguindo o esperado.

Enfatizamos a palavra *"maioria"* já que até a **entrega preliminar** deste relatório, não estamos convencidos de como computar a **média de clientes no sistema** a cada instante, uma vez que os valores obtidos até então divergem dos valores esperados para essa métrica em específico.

4 Desenvolvimento

A função principal de nosso simulador chama-se `simulation`, essa função recebe 4 parâmetros de entrada, sendo 3 deles obrigatórios para o funcionamento da simulação e 1 opcional.

Os parâmetros são:

1. `LAMBDA`:

   Essa variável representa a taxa λ do fluxo de *Poisson* que modela os eventos de chegada de um cliente na fila, aceitando números reais maiores que 0.

2. `MU`:

   Este representa a taxa μ da distribuição exponencial que define o tempo de serviço de um cliente da fila, por isso aqui também são aceitos números reais maiores que 0.

3. `LIM_TIME`:

   Aqui temos a variável que limita o tempo de criação dos eventos de chegada na fila, podendo aceitar qualquer valor de tempo, desde que maior que 0.

4. `VERBOSE` (opcional):

   Essa variável do tipo *boolean* altera o modo de apresentação do `simulation`. Por padrão seu valor é `False`, e só são mostradas as métricas obtidas ao fim da simulação. Porém, uma vez indicado o valor 1 na entrada será imprimido no terminal todas as informações relevantes de cada turno de processamento de eventos na função.

```
# simulação,
def simuation(LAMBDA, MU, LIM_TIME, VERBOSE = False)
```

Figura 1: Definição da função simulation

A função possui 3 classes principais, sendo elas `Measures`, `Event` e `Simulation`, além de alguns métodos auxiliares que atuam no decorrer do funcionamento da simulação. Abaixo explicaremos a fundo o funcionamento de cada um destes componentes.

4.1 Funções de Modelagem do Tempo dos Eventos

Temos duas funções para a modelagem do tempo eventos, a `next_arrival()` e a `residual_life`. Ambas utilizam a função do `python` `random.expovariate` que retorna o valor de uma variável aleatória exponencial com uma taxa passada como parâmetro.

```
# o tempo de vida residual é modelado por uma variavel aleatória exponencial com taxa MU
def residual_life():
  return random.expovariate(MU)

# o momento da proxima chegada é modelado pela soma de variaveis aleatórias exponenciais iid com taxa LAMBDA
ARRIVAL_TIME = 0
def next_arrival():
  return ARRIVAL_TIME + random.expovariate(LAMBDA)
```

Figura 2: Funções de Modelagem do Tempo dos Eventos

A função `residual_life` retorna o tempo que o cliente no servidor atualmente passará sendo servido. Ele é dado pelo valor de uma variável aleatória exponencial com taxa `MU`.

A função `next_arrival` retorna o momento do próximo evento de chegada de cliente, que é dado pelo momento da última chegada mais o valor de uma variável aleatória exponencial com taxa `LAMBDA`.

4.2 Classe Measures

A classe `Measures` é responsável por guardar as variáveis utilizadas para os cálculos das métricas de retorno interessantes à simulação, sua composição é ilustrada a seguir.

```
# Medidas interessantes a serem analisadas
class Measures:
  def __init__(self):
    #variaveis auxiliares para as medidas
    self.TotalClients = 0 # número total de clientes que passaram pelo sistema
    self.sum_Nq = 0 #  somatorio do número de clientes na fila de espera a cada vez que um cliente chega
    self.N = 0 # total medio de clientes no sistema a cada momento
    self.T = 0 # somatorio do tempo total gasto no sistema por cada cliente
    self.W = 0 # somatorio do tempo total gasto na fila de espera por cada cliente
    self.X = 0 # somatorio do tempo total gasto em serviço por cada cliente

    self.count_idle_cycles = 0 # numero de ciclos ociosos
    self.count_birth_and_death_processes = 0 # numero de ciclos de vida e morte
    self.sum_idle_cycles_time = 0 # somatorio da duração de cada ciclo ocioso
    self.sum_birth_and_death_processes_time = 0 # somatorio da duração de cada ciclo de vida e morte
    self.sum_time = 0 # tempo total de simulação
```

Figura 3: Composição da classe Measures

Podemos averiguar que essa classe apresenta 12 variáveis utilizadas para encontrar as médias de cada um dos aspectos relevantes ao problema, sendo elas:

- `TotalClients`:

  Quantidade total de clientes que passaram pelo sistema durante a simulação.

- `sum_Nq`:

  Somatório do número de clientes na fila de espera. Essa variável é computada a cada evento de chegada de um novo membro na fila de espera do sistema.

- `N`:

  Variável que guarda a acumulo do total de clientes no sistema a cada momento, representando a área embaixo do gráfico de (tempo) $\times$ (número de clientes no sistema).

- `T`:

  Acumulador do tempo total que cada cliente gasta no sistema.

- `W`:

  Somatório do tempo gasto por cada cliente na fila de espera do sistema.

- `X`:

  Somatório do tempo total gasto em serviço por cada cliente.

- `count_idle_cycles`:

  Contador do número de ciclos ociosos, isto é, ciclos onde não havia ninguém tanto na fila de espera quanto sendo no servidor.

- `count_birth_and_death_processes`:

  Número de ciclos de vida e morte.

- `sum_idle_cicles_time`:

  Variável que guarda a quantidade de tempo total em que o servidor ficou ocioso.

- `sum_birth_and_death_processes_time`

  Somatório do tempo de cada ciclo de vida do servidor.

- `sum_time`:

  Tempo total da simulação

Utilizamos as variáveis explicitadas acima para calcular as médias através dos métodos ilustrados na Figura 4 fixada na próxima página.

```
# valor esperado do número de clientes no sistema, a cada momento
def Expected_Value_N(self):
  return self.N / (self.sum_time)

# valor esperado do número de clientes na fila de espera, a cada chegada de novo cliente
def Expected_Value_Nq(self):
  return self.sum_Nq / self.TotalClients

# valor esperado da porcentagem do tempo em que o sistema está ativo
def Expected_Value_Rho(self):
  return self.sum_birth_and_death_processes_time / (self.sum_time)

# valor esperado do tempo de serviço prestado a cada cliente
def Expected_Value_X(self):
  return self.X / self.TotalClients

# valor esperado do tempo na fila de espera de cada cliente
def Expected_Value_W(self):
  return self.W / self.TotalClients

# valor esperado do tempo de cada cliente no sistema
def Expected_Value_T(self):
  return self.T / self.TotalClients
```

Figura 4: Definição dos métodos da classe Measures

São calculados 6 métricas na classe Measures, sendo elas:

- `Expected_value_N`:

  Calculo do valor esperado do número de clientes no sistema a cada momento, feito através da conta `N/sum_time`.

- `Expected_value_Nq`:

  Calculo do valor esperado do número de clientes na fila de espera no momento que um novo cliente chega. É calculado a partir do evento de chegada de novos clientes, feito através da conta `sum_Nq/TotalClients`.

- `Expected_value_Rho`:

  Calculo do valor esperado da porcentagem de tempo em que o sistema está ativo, feito através da conta `sum_birth_and_death_processes_time/sum_time`.

- `Expected_value_X`:

  Calculo do valor esperado do tempo em que o serviço é prestado para cada cliente, feito através da conta `X/TotalClients`.

- `Expected_value_W`:

  Calculo do valor esperado do tempo na fila de espera de cada cliente, feito através da conta `W/TotalClients`.

- `Expected_value_T`:

  Calculo do valor esperado do tempo de cada cliente no sistema, feito através da conta `T/TotalClients`.

Com isso explicamos a totalidade da classe `Measures` e seu funcionamento em nosso trabalho.

4.3 Classe Event

A classe `Event` é responsável pela modelagem dos eventos de chegada e saída da simulação, é necessário passar para ela 3 parâmetros de inicialização, sendo eles: `event_type` que indica qual é o tipo de evento sendo tratado, podendo assumir 2 valores, A significando eventos de chegada ou D significando eventos de saída, `time` indicando o momento em que aquele evento deve acontecer e por último `id` representando o id do cliente que aquele evento se refere.

```
class Event:
  def __init__(self, event_type, time, id):
    self.event_type = event_type
    self.time = time
    self.service_start = -1
    if(event_type == 'A'):
      self.arrival_time = self.time
      self.departure_time = -1
    else:
      self.departure_time = self.time
    self.id = id
```

Figura 5: Inicializador da classe Event

Na imagem podemos ver que os parâmetros de entrada `time`, `event_type` e `id` são guardados em variáveis internas a classe `Event` de mesmo nome, além de vermos outras 2 variáveis, sendo elas `arrival_time` e `departure_time`, elas são variáveis só inicializadas dependendo do tipo de evento sendo tratado, caso seja do tipo A inicializamos a variável `arrival_time` com valor de `time` e caso contrário inicializamos `departure_time` com o valor de `time`. Outra variável da classe á a `service_start` que guarda o momento em que o cliente entra no sistema. Vale ressaltar que como preservamos o valor do `arrival_time` e do `service_start` já atribuídos quando temos um evento de saída, quando o cliente sair do sistema na simulação teremos os dados de todos os instantes de tempo relacionados a ele.

Essa classe é constituída por outros 2 métodos, sendo eles o método `__str__`, responsável por imprimir na tela um texto com dados do cliente e do evento em questão, e `__lt__`, função de comparação entre os valores da variável `time` de 2 instâncias da classe `Event`.

```
# para imprimir de forma expressiva os dados do cliente
def __str__(self):
  if(self.event_type == "A"):
    return str(self.id) + "\t|\t"+str(f"{self.arrival_time:.3f}").format()+ "\t|"
  return str(self.id) + "\t|\t" + str(f"{self.arrival_time:.3f}") + "\t|\t"
  + str(f"{self.service_start:.3f}") + "\t|\t" + str(f"{self.departure_time:.3f}") + "\t|"

# comparador, utilizado para sabermos que evento vem primeiro
# em caso de empate damos prioridade para a saída do sistema
def __lt__(self, other):
  if(self.time<other.time):
      return True
  if(self.time>other.time):
      return False
  else:
      return (self.event_type == 'D')
```

Figura 6: Métodos da Classe Event

4.4 Classe System

A classe `System` é responsável pelo controle da nossa fila, o que inclui a inserção de novos clientes na fila de espera, a saída do cliente do servidor e também a passagem do cliente no começo da fila de espera para o servidor. Ela possui duas variáveis, o `server` que é do tipo `Event` e guarda o cliente que está sendo servido e a variável `waiting_queue` que é uma lista de objetos do tipo `Event`, representando os clientes na fila de espera.

```python
# classe para modelar o sistema
# waiting_queue -- lista que representa a fila de espera (células são do tipo Event)
# server -- variável do tipo Event que representa o cliente que está sendo servido
class System:
  def __init__(self):
    self.waiting_queue = []
    self.server = None

  # cliente e entra na lista de espera
  def arrival(self, e):
    self.waiting_queue.append(e)

  # cliente que está sendo servido sai do sistema
  # garantidamente só é chamado se o sistema estava ocupado
  def departure(self):
    server = copy.deepcopy(self.server)
    self.server = None
    return server

  # o próximo cliente sai da fila de espera e é servido
  def next(self, T):
    if(self.busy() or not(self.waiting_queue)):
      return None
    self.server = self.waiting_queue.pop(0)
    x = residual_life()
    measures.X += x
    self.server.departure_time = T + x
    self.server.time = self.server.departure_time
    return self.server
```

Figura 7: Inicializador da classe System e métodos de controle da fila

Na figura acima percebemos que a inicialização do `System` é bem simples, começamos com a fila vazia e com o servidor ocioso. Note que apesar de na inicialização da classe o servidor estar ocioso, isso é mudado imediatamente quando iniciamos a simulação.

O método `arrival` simplesmente insere um novo cliente no fim da fila de espera, sem nenhum tipo de tratamento já que o seu tamanho não é limitado. O método `departure` retira o cliente que estava sendo servido do servidor e retorna uma cópia dele para que seja mais fácil calcularmos algumas métricas na simulação.

O método `next` simula o processo do cliente no início da fila de espera entrar no servidor. Inicialmente é feita a verificação de se o servidor está livre e se há algum cliente na fila de espera, se essas duas condições forem atendidas fazemos esse processo. Neste momento, além de passarmos o cliente do começo da fila para o servidor calculamos seu `departure_time` que é dado pelo momento em que ele entra no servidor mais a sua vida residual.

Nesta classe há também métodos auxiliares que retornam informações úteis sobre o sistema:

```
# verifica se o sistema está ocupado
def busy(self):
  return not(self.server == None)

# tamanho da fila de espera
def waiting_queue_size(self):
  return len(self.waiting_queue)

# tamanho da fila (fila de espera + servidor)
def full_queue_size(self):
  s = len(self.waiting_queue)
  if(self.busy()):
    s += 1
  return s
```

Figura 8: Inicializador da classe System e métodos de controle da fila

Os métodos auxiliares são:

- `busy` - retorna valor booleano indicando se o sistema está ocupado.
- `waiting_queue_size` - retorna o tamanho da fila de espera.
- `full_queue_size` - retorna o tamanho da fila, incluindo o tamanho da fila de espera e o cliente que está sendo servido caso o servidor esteja ativo.

Por fim, temos o método `__str__` que é usado para imprimir informações sobre a fila do servidor de forma estilizada:

```python
# para imprimir de forma expressiva os dados dos clientes na fila
# imprimimos os dados de no máximo 5 clientes para não poluir a simulação
def __str__(self):
  ans = "\tServidor: \n"
  ans += "\033[1m"+"\tid\t|\tchegada\t|\tservidor|\tsaída\t|\n"+"\033[0m"
  ans += "\t" + str(self.server) + "\n"


  ans += "\n \tFila de Espera: ("+ str(self.waiting_queue_size())+ ") \n"

  ans += "\033[1m"+"\tid\t|\tchegada\t|\n"+"\033[0m"

  lim = 5
  for w in self.waiting_queue:
    ans += "\t" + str(w) + " \n"
    lim -= 1
    if(lim == 0):
      ans += "\t...\n"
      break
  ans += "---------------------------------------\n"
  return ans
```

Figura 9: Método de impressão da Classe System

5 Simulação

A simulação utiliza todas as classes e métodos mencionados acima.

Iremos explicar primeiro sua inicialização e, após, seu laço principal. Por fim, mostraremos as métricas que a simulação apresenta para o usuário.

5.1 Inicialização da Simulação

As variáveis principais da simulação são o `system` e o `measures`, duas instâncias das classes que já explicamos detalhadamente em 4.4 e 4.2 respectivamente.

Além dessas, temos as variáveis `id`, `ARRIVAL_TIME` e `next_arrival_event`, referentes ao próximo evento de chegada de cliente. Note que esses valores são inicializados fora do laço principal para já começarmos a simulação com a chegada de um cliente.

A variável `next_departure_event` é referente ao próximo evento de saída do cliente do servidor, ela começa com o valor `None` pois só definimos o momento de saída do primeiro cliente no laço da simulação.

Também temos a variável `T` que indica o tempo atual da simulação, e algumas variáveis auxiliares para coletarmos métricas, sendo elas: `birth`, `death` e `count_clients`.

```
# variáveis principais
system = System()
measures = Measures()

id = 0 # id do próximo cliente que chegará

ARRIVAL_TIME = 0
# evento de chegada do próximo cliente
# é inicializado fora do laço principal para que a simulção comece com um cliente no sistema
next_arrival_event = Event('A', ARRIVAL_TIME, id)

# evento de saída do cliente que está no servidor
# None se o sistema está ocioso, ou antes da primeira iteração do laço principal
next_departure_event = None

T = -1 # tempo atual
birth = -1 # tempo do último nascimento, -1 se ocioso
death = -1 # tempo da última morte, -1 se é o primeiro ciclo de vida e morte
count_clients = 0 # número total de clientes que passaram pelo sistema no ciclo de vida e morte atual
```

Figura 10: Inicialização da simulação

5.2 Laço da Simulação

Apesar do nosso esforço para encapsular o código em classes, o laço da simulação principal ficou maior do que gostaríamos.

A condição de parada do laço da simulação é dada por:

```
# laço principal da simulação
# Quando a simulação chega ao tempo limite paramos de aceitar a chegada de novos clientes
# e, após, a simulação para depois de atender todos os clientes que já estavam na fila
while(next_departure_event or next_arrival_event):
```

Figura 11: Condição de Parada do Laço da Simulação

Isso quer dizer que continuamos a simulação enquanto houver eventos de chegada ou de saída de clientes ainda não processados.

Claro, se não tivéssemos nenhum controle de parada da criação de eventos novos nosso programa rodaria para sempre. Então, esse controle é feito da seguinte forma: quando o tempo da simulação (`T`) chega ao tempo limite passado como parâmetro da função principal (`LIM_TIME`) paramos de criar novos eventos de chegada de clientes. Podemos pensar nisso como o funcionamento da fila do bandejão, quando dá o horário de fechamento do bandejão os alunos que já estavam na fila continuam entrando, mas novos alunos não podem entrar na fila.

A parte interna do laço da simulação pode ser dividida em 4 partes principais:

- Descoberta do evento atual
- Tratamento do evento atual
- Criação de novo evento de `departure`
- Tratamento do fim do ciclo de nascimento e morte

Abaixo explicaremos com mais detalhes cada parte.

5.2.1 Descoberta do Evento Atual

A cada iteração do laço temos pelo menos um entre dois eventos não processados. Um evento de chegada de cliente e um evento de saída de cliente.

Atribuímos ao `current_event` o evento não processado de menor tempo para assegurar a ordem cronológica dos eventos.

Se o evento for de chegada e não tivermos atingido o tempo limite da simulação, criamos um novo evento de chegada. Além disso, atualizamos a métrica `sum_Nq`.

Se o evento for de saída, atualizamos a métrica `T`.

```
# evento atual, pode ser de chegada ou de saída
current_event = []

# se o evento atual é de chegada
if(not(next_departure_event) or (not(next_arrival_event == None) and next_arrival_event.time < next_departure_event.time) ):
  if(VERBOSE):
    print("Arrival event: ")
  # atualizamos a métrica do número de clientes na fila de espera no momento de chegada do cliente
  measures.sum_Nq += system.waiting_queue_size()

  current_event = next_arrival_event
  id += 1
  next_arrival_event = None
  # aceitamos nova chegada na sila de espera se não terminou o tempo da simulação
  if(T < LIM_TIME):
    ARRIVAL_TIME = next_arrival()
    next_arrival_event = Event('A', ARRIVAL_TIME, id)

# se o evento atual é de partida
else:
  if(VERBOSE):
    print("Departure event: ")
  current_event = next_departure_event

  # atualizamos a métrica do tempo total do cliente no sistema
  measures.T += next_departure_event.departure_time - next_departure_event.arrival_time

  next_departure_event = None

if(VERBOSE):
  print("\033[1m"+"id\t|\tchegada\t|\tservidor|\tsaída\t|\n"+"\033[0m")
  print(current_event)
```

Figura 12: Laço da Simulação - Descoberta do evento atual

5.2.2 Tratamento do Evento Atual

Na parte de tratamento, atualizamos o tempo da simulação para o tempo do evento atual.

Caso o evento seja de chegada após um ciclo ocioso, atualizamos as métricas referentes aos ciclos ociosos do sistema, além do tempo total de simulação.

Caso seja de saída, atualizamos a métrica `N` e chamamos a função `departure` do `system` para realizar o procedimento de saída da fila.

```
# atualizamos tempo atual da simulação
last_T = T
T = current_event.time

if(VERBOSE):
  print(f"\nCurrent event time: {T:.3f}\n")

# se o evento atual é de chegada
if(current_event.event_type == 'A'):
  count_clients += 1
  if(birth == -1):
    birth = T

    # se estamos começando novo ciclo de vida e morte
    if(not(death == -1)):
      # atualizamos várias métricas:

      # somatório do número de clientes no sistema a cada momento
      measures.N += system.full_queue_size() * (T - last_T)

      # número de ciclos ociosos
      measures.count_idle_cycles += 1

      # somatorio da duração de cada ciclo ocioso
      measures.sum_idle_cycles_time += birth - death

      # tempo total de simulação
      measures.sum_time += birth - death

      if(VERBOSE):
        print(f"servidor ocioso de {death:.3f} ate {birth:.3f}")
  system.arrival(current_event)

# se o evento atual é de saída
else:
  # somatório do número de clientes no sistema a cada momento
  measures.N += system.full_queue_size() * (T - last_T)

  # cliente termina de ser servido
  system.departure()
```

Figura 13: Laço da Simulação - Tratamento do Evento Atual

5.2.3 Criação de novo evento de departure

Ao final do tratamento do evento atual, verificamos se um cliente começou a ser servido nesta iteração do laço. Se isso ocorre, criamos o evento de saída desse cliente.

```python
serving = system.next(T)
# se um cliente começou a ser servido neste momento
if(serving):
  # somatório do tempo que cada cliente passa na fila de espera
  measures.W += T - serving.arrival_time

  # gerando o evento de saída do cliente do sistema
  serving.service_start = T
  serving.event_type = 'D'
  next_departure_event = serving
```

Figura 14: Laço da Simulação - Criação de novo evento de departure

5.2.4 Tratamento do fim do ciclo de nascimento e morte

Se o sistema está ocioso ao fim da iteração, então finalizamos um ciclo de vida e morte e atualizamos as métricas relacionadas a ele, sendo elas: `count_birth_and_death_processes`, `sum_birth_and_death_processes_time` e `TotalClients`, além da métrica de tempo total da simulação `sum_time`.

```python
# se o sistema está ocioso
if(not(system.busy())):
  death = T
  # atualizamos várias métricas:

  # número de ciclos de vida e morte
  measures.count_birth_and_death_processes += 1

  # somatorio da duração de cada ciclo de vida e morte
  measures.sum_birth_and_death_processes_time += death - birth

  # tempo total de simulação
  measures.sum_time += death - birth

  # numero total de clientes que passaram pelo sistema
  measures.TotalClients += count_clients

  if(VERBOSE):
    print(f"servidor ficou ativo de {birth:.3f} ate {death:.3f} e atendeu {count_clients} clientes")
  count_clients = 0
  birth = -1

  if(VERBOSE):
    print("")
if(VERBOSE):
  print(system)
```

Figura 15: Laço da Simulação - Tratamento do fim do ciclo de nascimento e morte

5.3 Apresentação das Métricas Obtidas

Por fim, a função da simulação mostra para o usuário as métricas obtidas. Se quiséssemos retornar essas métricas para serem utilizadas fora da função poderíamos simplesmente retornar o objeto `measures` em vez de imprimir seus resultados.

```python
print("Métricas obtidas:")
print(f"E[T] =  {measures.Expected_Value_T():.3f}")
print(f"E[W] =  {measures.Expected_Value_W():.3f}")
print(f"E[X] =  {measures.Expected_Value_X():.3f}")
print("")
print(f"E[N] =  {measures.Expected_Value_N():.3f}")
print(f"E[Nq] =  {measures.Expected_Value_Nq():.3f}")
print(f"E[Rho] =  {measures.Expected_Value_Rho():.3f}")
```

Figura 16: Apresentação das Métricas Obtidas pela Simulação

6 Decisões de projeto

Como esta primeira etapa é apenas uma entrega preliminar e dado o conhecimento técnico da equipe, entendemos que a melhor ferramenta a ser utilizada seria a linguagem de programação *Python 3*.

Graças à sua simplicidade e poder de implementação simples, conseguimos facilmente traduzir nossas ideais iniciais para a simulação em um projeto que trouxe suas primeiras saídas em poucos minutos.

Durante o desenvolvimento do simulador, percebemos que diversos trechos de código possuíam manipulações de dados que seriam melhor organizadas em métodos ou funções.

Além disso, boa parte destes métodos e funções possuíam objetivos pertinentes a um mesmo contexto. Dessa maneira, foi utilizada a abordagem *OOP* ou de orientação à objetos para a simplificação e coerência do código.

Nesse ínterim, uma das **classes mais relevantes no projeto** foi a classe `Event`, responsável pelo tratamento das chegadas (`arrivals`) e das partidas (`departures`) do sistema. De forma a contemplar a condição de parada do simulador, um laço `while` também foi utilizado, indicando que a execução do programa simulador encerra apenas quando o **tempo limite** previamente estabelecido fosse atingido.

É de conhecimento do grupo que, por ser uma linguagem de programação **interpretada** e não compilada, a linguagem *Python* não é a mais rápida. Todavia, como um de nossos objetivos centrais é compreender os fundamentos da disciplina de Avaliação e Desempenho e observar os conceitos chave estudados em aula, percebemos que o tempo de execução gasto para os **casos de teste** selecionados não é um impeditivo em nosso estudo.

Para a versão preliminar, visando a otimização do tempo de desenvolvimento do programa, consideramos apenas métricas numéricas simples, ainda não sendo desenvolvido nenhum gráfico complementar para a visualização dos resultados obtidos.

Outrossim, desenvolvemos o simulador utilizando o **Google *Colaboratory*** devido aos ajustes e refinos do ambiente de programação para desenvolvimento serem realizados de forma praticamente automática. Para mais, o ambiente facilitou o **intercambiamento** de blocos de código e comentários, além de ter possibilitado a boa manutenção do trabalho.

7 Resultados Obtidos

Antes de iniciar, é importante ressaltar que valores de ρ maiores do que 1 indicam que a fila está com **gargalo.** Isto é, o servidor de atendimento não está conseguindo atender às demandas de todos os processos *Poisson* que chegam, congestionando todo o sistema, uma vez que para $\rho > 1$, **ocorrem mais chegadas do que partidas** em todo o sistema (fila de espera + servidor).

A seguir apresentamos os resultados obtidos pela simulação com diferentes taxas, mas com o tempo limite da simulação fixo em $100,000$. Optamos por colocar os resultados de somente uma execução com cada conjunto de taxas.

7.1 Taxas $\lambda = 1, \mu = 2$

```
[2]  simuation(1, 2, 100000)

     Métricas obtidas:
     E[T] =  0.989
     E[W] =  0.490
     E[X] =  0.499

     E[N] =  0.658
     E[Nq] =  0.486
     E[Rho] =  0.497
```

Figura 17: Resultado da Simulação com Taxas $\lambda = 1$, $\mu = 2$

7.2 Taxas $\lambda = 2, \mu = 4$

```
[3]  simuation(2, 4, 100000)

     Métricas obtidas:
     E[T] =  0.503
     E[W] =  0.253
     E[X] =  0.250

     E[N] =  0.673
     E[Nq] =  0.505
     E[Rho] =  0.500
```

Figura 18: Resultado da Simulação com Taxas $\lambda = 2$, $\mu = 4$

7.3 Taxas $\lambda = 1.05, \mu = 1$

```
[4]  simuation(1.05, 1, 100000)

     Métricas obtidas:
     E[T] =  2165.305
     E[W] =  2164.303
     E[X] =  1.002

     E[N] =  1109.856
     E[Nq] =  2148.775
     E[Rho] =  1.000
```

Figura 19: Resultado da Simulação com Taxas $\lambda = 1.05$, $\mu = 1$

7.4 Taxas $\lambda = 1.10, \mu = 1$

```
[5]  simuation(1.10, 1, 100000)

     Métricas obtidas:
     E[T] =  5745.134
     E[W] =  5744.133
     E[X] =  1.001

     E[N] =  3004.642
     E[Nq] =  5734.055
     E[Rho] =  1.000
```

Figura 20: Resultado da Simulação com Taxas $\lambda = 1.10$, $\mu = 1$

Durante as fases de desenvolvimento e de obtenção das métricas para posterior análise, a equipe percebeu que a métrica $E(N)$ está **divergindo** do esperado analiticamente, tanto violando a **Lei de Little** que diz que $E(N) = \lambda E(T)$ para um sistema homogêneo e em equilíbrio, quanto pelo número total de clientes no sistema ser igual ao tamanho da fila de espera somado ao número de clientes em servidor, ou simplesmente a expressão $E(N) = E(Nq) + \rho$. Ao invés disso, **o simulador está retornando valores de $E(N)$ menores do que $E(Nq)$**, um absurdo!

Acreditamos que este fenômeno se deve pela definição do tempo limite da simulação como parâmetro de entrada impedir que este seja longo o suficiente para uma convergência ou pela condição de parada estabelecida pela equipe. Quando o simulador chega ao tempo limite de execução, é permitido que o servidor termine o atendimento pendente dos clientes que aguardavam na fila de espera e que impeça novos eventos de chegada.

8 Anexos

```
import numpy as np
import random
import copy
import time

# A simulacao em si
def simuation(LAMBDA, MU, LIM_TIME, VERBOSE = False):

  # Medidas interessantes a serem analisadas
  class Measures:
    def __init__(self):
      #variaveis auxiliares para as medidas
      self.TotalClients = 0 # número total de clientes que passaram pelo sistema
      self.sum_Nq = 0 #  somatorio do número de clientes na fila de espera a cada vez que um cliente chega
      self.N = 0 # total medio de clientes no sistema a cada momento
      self.T = 0 # somatorio do tempo total gasto no sistema por cada cliente
      self.W = 0 # somatorio do tempo total gasto na fila de espera por cada cliente
      self.X = 0 # somatorio do tempo total gasto em serviço por cada cliente

      self.count_idle_cycles = 0 # numero de ciclos ociosos
      self.count_birth_and_death_processes = 0 # numero de ciclos de vida e morte
      self.sum_idle_cycles_time = 0 # somatorio da duração de cada ciclo ocioso
      self.sum_birth_and_death_processes_time = 0 # somatorio da duração de cada ciclo de vida e morte
      self.sum_time = 0 # tempo total de simulação

    # valor esperado do número de clientes no sistema, a cada momento
    def Expected_Value_N(self):
      return self.N / (self.sum_time)

    # valor esperado do número de clientes na fila de espera, a cada chegada de novo cliente
    def Expected_Value_Nq(self):
      return self.sum_Nq / self.TotalClients

    # valor esperado da porcentagem do tempo em que o sistema está ativo
    def Expected_Value_Rho(self):
      return self.sum_birth_and_death_processes_time / (self.sum_time)

    # valor esperado do tempo de serviço prestado a cada cliente
    def Expected_Value_X(self):
      return self.X / self.TotalClients

    # valor esperado do tempo na fila de espera de cada cliente
    def Expected_Value_W(self):
      return self.W / self.TotalClients

    # valor esperado do tempo de cada cliente no sistema
    def Expected_Value_T(self):
      return self.T / self.TotalClients

  # classe que modela os eventos de chegada e saída de cada cliente no sistema
  # id -- id do cliente -- incremental a partir de 1
  # event_type -- tipo do evento ("A" - chegada, "D" - saída)
  # arrival_time -- momento da chegada no sistema
```

```
# departure_time -- momento de saída do sistema
 # service_start -- momento em que começa a ser serviço
# time -- momento atual, pode ser igual a arrival_time ou departure_time, a depender
#         de qual evento está sendo modelado

class Event:
  def __init__(self, event_type, time, id):
    self.event_type = event_type
    self.time = time
    self.service_start = -1
    if(event_type == 'A'):
      self.arrival_time = self.time
      self.departure_time = -1
    else:
      self.departure_time = self.time
    self.id = id

  # para imprimir de forma expressiva os dados do cliente
  def __str__(self):
    if(self.event_type == "A"):
      return str(self.id) + "\t|\t"+str(f"{self.arrival_time:.3f}").format()+"\t|"
    return str(self.id) + "\t|\t" + str(f"{self.arrival_time:.3f}") + "\t|\t"
    + str(f"{self.service_start:.3f}") + "\t|\t" + str(f"{self.departure_time:.3f}") + "\t|"

  # comparador, utilizado para sabermos que evento vem primeiro
  # em caso de empate damos prioridade para a saída do sistema
  def __lt__(self, other):
    if(self.time<other.time):
        return True
    if(self.time>other.time):
        return False
    else:
        return (self.event_type == 'D')

# o tempo de vida residual é modelado por uma variavel aleatória exponencial com taxa MU
def residual_life():
  return random.expovariate(MU)

# o momento da proxima chegada é modelado pela soma de variaveis aleatórias exponenciais iid com taxa LAMBDA
ARRIVAL_TIME = 0
def next_arrival():
  return ARRIVAL_TIME + random.expovariate(LAMBDA)

# classe para modelar o sistema
# waiting_queue -- lista que representa a fila de espera (células são do tipo Event)
# server -- variável do tipo Event que representa o cliente que está sendo servido
class System:
  def __init__(self):
    self.waiting_queue = []
    self.server = None

  # cliente e entra na lista de espera
  def arrival(self, e):
```

```
      self.waiting_queue.append(e)

    # cliente que está sendo servido sai do sistema
    # garantidamente só é chamado se o sistema estava ocupado
    def departure(self):
      server = copy.deepcopy(self.server)
      self.server = None
      return server

    # o próximo cliente sai da fila de espera e é servido
    def next(self, T):
      if(self.busy() or not(self.waiting_queue)):
        return None
      self.server = self.waiting_queue.pop(0)
      x = residual_life()
      measures.X += x
      self.server.departure_time = T + x
      self.server.time = self.server.departure_time
      return self.server

    # verifica se o sistema está ocupado
    def busy(self):
      return not(self.server == None)

    # tamanho da fila de espera
    def waiting_queue_size(self):
      return len(self.waiting_queue)

    # tamanho da fila (fila de espera + servidor)
    def full_queue_size(self):
      s = len(self.waiting_queue)
      if(self.busy()):
        s += 1
      return s

    # para imprimir de forma expressiva os dados dos clientes na fila
    # imprimimos os dados de no máximo 5 clientes para não poluir a simulação
    def __str__(self):
      ans = "\tServidor: \n"
      ans += "\033[1m"+"\tid\t|\tchegada\t|\tservidor|\tsaída\t|\n"+"\033[0m"
      ans += "\t" + str(self.server) + "\n"


      ans += "\n \tFila de Espera: ("+ str(self.waiting_queue_size())+ ") \n"

      ans += "\033[1m"+"\tid\t|\tchegada\t|\n"+"\033[0m"

      lim = 5
      for w in self.waiting_queue:
        ans += "\t" + str(w) + " \n"
        lim -= 1
        if(lim == 0):
          ans += "\t...\n"
          break
      ans += "---------------------------------------\n"
      return ans


  # variáveis principais
  system = System()
```

```
measures = Measures()

id = 0 # id do próximo cliente que chegará

ARRIVAL_TIME = 0
# evento de chegada do próximo cliente
# é inicializado fora do laço principal para que a simulção comece com um cliente no sistema
next_arrival_event = Event('A', ARRIVAL_TIME, id)

# evento de saída do cliente que está no servidor
# None se o sistema está ocioso, ou antes da primeira iteração do laço principal
next_departure_event = None

T = -1 # tempo atual
birth = -1 # tempo do último nascimento, -1 se ocioso
death = -1 # tempo da última morte, -1 se é o primeiro ciclo de vida e morte
count_clients = 0 # número total de clientes que passaram pelo sistema no ciclo de vida e morte atual

# laço principal da simulação
# Quando a simulação chega ao tempo limite paramos de aceitar a chegada de novos clientes
# e, após, a simulação para depois de atender todos os clientes que já estavam na fila
while(next_departure_event or next_arrival_event):
  # evento atual, pode ser de chegada ou de saída
  current_event = []

  # se o evento atual é de chegada
  if(not(next_departure_event) or (not(next_arrival_event == None) and next_arrival_event.time < next_departure_event.time) ):
    if(VERBOSE):
      print("Arrival event: ")
    # atualizamos a métrica do número de clientes na fila de espera no momento de chegada do cliente
    measures.sum_Nq += system.waiting_queue_size()

    current_event = next_arrival_event
    id += 1
    next_arrival_event = None
    # aceitamos nova chegada na sila de espera se não terminou o tempo da simulação
    if(T < LIM_TIME):
      ARRIVAL_TIME = next_arrival()
      next_arrival_event = Event('A', ARRIVAL_TIME, id)

  # se o evento atual é de partida
  else:
    if(VERBOSE):
      print("Departure event: ")
    current_event = next_departure_event

    # atualizamos a métrica do tempo total do cliente no sistema
    measures.T += next_departure_event.departure_time - next_departure_event.arrival_time

    next_departure_event = None

```

```
    if(VERBOSE):
      print("\033[1m"+"id\t|\tchegada\t|\tservidor|\tsaída\t|\n"+"\033[0m")
      print(current_event)

    # atualizamos tempo atual da simulação
    last_T = T
    T = current_event.time

    if(VERBOSE):
      print(f"\nCurrent event time: {T:.3f}\n")

    # se o evento atual é de chegada
    if(current_event.event_type == 'A'):
      count_clients += 1
      if(birth == -1):
        birth = T

        # se estamos começando novo ciclo de vida e morte
        if(not(death == -1)):
          # atualizamos várias métricas:

          # somatório do número de clientes no sistema a cada momento
          measures.N += system.full_queue_size() * (T - last_T)

          # número de ciclos ociosos
          measures.count_idle_cycles += 1

          # somatorio da duração de cada ciclo ocioso
          measures.sum_idle_cycles_time += birth - death

          # tempo total de simulação
          measures.sum_time += birth - death

          if(VERBOSE):
            print(f"servidor ocioso de {death:.3f} ate {birth:.3f}")
      system.arrival(current_event)

    # se o evento atual é de saída
    else:
      # somatório do número de clientes no sistema a cada momento
      measures.N += system.full_queue_size() * (T - last_T)

      # cliente termina de ser servido
      system.departure()

    serving = system.next(T)
    # se um cliente começou a ser servido neste momento
    if(serving):
      # somatório do tempo que cada cliente passa na fila de espera
      measures.W += T - serving.arrival_time

      # gerando o evento de saída do cliente do sistema
      serving.service_start = T
      serving.event_type = 'D'
      next_departure_event = serving

    # se o sistema está ocioso
    if(not(system.busy())):
      death = T
      # atualizamos várias métricas:
```

```python

      # número de ciclos de vida e morte
      measures.count_birth_and_death_processes += 1

      # somatorio da duração de cada ciclo de vida e morte
      measures.sum_birth_and_death_processes_time += death - birth

      # tempo total de simulação
      measures.sum_time += death - birth

      # numero total de clientes que passaram pelo sistema
      measures.TotalClients += count_clients

      if(VERBOSE):
        print(f"servidor ficou ativo de {birth:.3f} ate {death:.3f} e atendeu {count_clients} clientes")
      count_clients = 0
      birth = -1

      if(VERBOSE):
        print("")
    if(VERBOSE):
      print(system)

  print("Métricas obtidas:")
  print(f"E[T] =  {measures.Expected_Value_T():.3f}")
  print(f"E[W] =  {measures.Expected_Value_W():.3f}")
  print(f"E[X] =  {measures.Expected_Value_X():.3f}")
  print("")
  print(f"E[N] =  {measures.Expected_Value_N():.3f}")
  print(f"E[Nq] =  {measures.Expected_Value_Nq():.3f}")
  print(f"E[Rho] =  {measures.Expected_Value_Rho():.3f}")
```

Listing 1: Código-fonte do simulador com tempo limite variável para posterior obtenção de métricas e estatísticas do sistema durante sua execução. O código pode ser acessado em: https://colab.research.google.com/drive/1TYhnW9Jsmq_qBT44OYYVq4YuSVp3Dhpb?usp=sharing.