FACULTAD DE INGENIERIA
Universidad de Buenos Aires

U.B.A. FACULTAD DE INGENIERÍA

Departamento de Computación

Modelos y Simulación 7526 - 9519

TRABAJO PRÁCTICO #2

Procesos de Poisson, Cadenas de Markov, Sistemas dinámicos, Simpy

Curso: 2019 - 1er Cuatrimestre

Turno: Miércoles

GRUPO N° 1	
Integrantes	**Padrón**
Amurrio, Gastón	93584
Gamarra Silva, Cynthia Marlene	92702
Pinto, Tomás	98757
Fecha de Entrega:	19-06-2019
Fecha de aprobación:	
Calificación:	
Firma de aprobación:	

Observaciones:

Índice

Índice **1**

1. Enunciado del trabajo práctico **2**

2. Implementación y resultados **5**
2.1. Ejercicio 1 5
2.2. Ejercicio 2 6
2.3. Ejercicio 3 8
2.4. Ejercicio 4 10
2.5. Ejercicio 5 10

3. Conclusiones **10**

Referencias **10**

A. Código fuente **11**
A.1. Resolución ejercicio 1 11
A.1.1. instruccion 11
A.1.2. main 11
A.2. Resolución ejercicio 2 13
A.2.1. ejercicio2.py 13
A.3. Resolución ejercicio 3 16
A.3.1. ejercicio3.py 16
A.4. Resolución ejercicio 4 17
A.4.1. ejercicio4.py 17
A.5. Resolución ejercicio 5 19
A.5.1. ejercicio5.py 19

1. Enunciado del trabajo práctico

FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

Simulación - 75.26 – 95.19

Consideraciones generales

La entrega de este TP debe contar con un informe explicando el procedimiento utilizado para resolver cada ejercicio y justificando las conclusiones a las que se arriba en cada punto. Se debe incluir también el código fuente utilizado para resolver cada ejercicio en uno o más archivo(s) por ejercicio.

Procesos de Poisson

Ejercicio 1

Utilizando Matlab, Octave o Python (sin utilizar Simpy) se debe determinar con cuál de las siguientes 2 arquitecturas nos permite tener un menor tiempo de procesamiento.

A un procesador arriban instrucciones siguiendo una distribución exponencial con una tasa de 250 instrucciones por microsegundo.

Las instrucciones se pueden agrupar en las siguientes categorías, y en función de ella es el tiempo que demoran en ser ejecutadas el cual sigue una distribución exponencial con parámetro lambda especificado a continuación.

Tipo de instrucción	Probabilidad	Tasa del tiempo de ejecución (instrucciones por µseg)
Simple	0,6	60
Compleja	0,4	10

Se sabe que el 65% de las instrucciones requieren buscar datos en memoria para antes de ser ejecutadas.

En este punto se plantean dos alternativas:

Alternativa 1: Cada vez que se requiera un dato de memoria se lo busca en memoria principal. Tarea cuyo tiempo corresponde a una distribución exponencial con una tasa de 2000 instrucciones por microsegundo.

Alternativa 2: Implementar un Caché, con el cual existiría una probabilidad de .6 de encontrar el dato requerido y no tener la necesidad de buscarlo en memoria principal.

El tiempo de acceso a este caché se puede modelar siguiendo una distribución exponencial con una tasa de 500 instrucciones por microsegundo.

Cuando el dato no es encontrado en el caché y debe buscarse en memoria principal donde su demora también sigue una distribución exponencial con una tasa de 1500 instrucciones por microsegundo.

FACULTAD DE INGENIERIA Universidad de Buenos Aires

Trabajo Práctico 2

Simulación - 75.26 – 95.19

Cadenas de Markov

Ejercicio 2

Una entidad bancaria se encuentra analizando cuantos clientes en simultáneo están conectados a su home banking.
Para ello comenzó a monitorear cada minuto la cantidad de clientes activos, y analizó como se modifica esta cantidad de una observación a la siguiente.
Se determinó que la probabilidad que la cantidad de clientes conectados aumentara, o se mantenga igual, podía modelarse utilizando una distribución de probabilidad Binomial, mientras que la probabilidad de tener menos clientes conectados respondía a una distribución uniforme.
Siendo i y j dos cantidades de usuarios conectados al sistema.

$$P\left(i \to j\right) = \begin{cases} \binom{n}{j-i} p^{j-i}(1-p)^{n-j-i} & si\ j \geq i \\ \dfrac{1 - \sum_{j=i}^{j=n} P\left(i \to j\right)}{i-1} & si\ j < i \end{cases}$$

Se sabe además que la probabilidad que un cliente se conecte al sitio es p=0,7
(Por simplicidad supondremos que el servidor de este home banking permite cómo máximo 50 clientes en simultáneo)

a) Determinar la matriz de transición de estados explicando cómo se obtiene la misma.
b) Utilizando Matlab, Octave o Python simular una posible evolución del sistema a lo largo de 100 observaciones graficando cómo se modifica la cantidad de clientes conectados en cada momento.
c) Simulando 100.000 observaciones realizar un histograma mostrando cuantas veces el sistema estuvo en cada estado.
(Recomendación: utilizar tantas categorías en el histograma como estados tiene el sistema).
d) Determinar el % de tiempo que en el home banking no tuvo clientes conectados.
e) Se está evaluando migrar el home banking a un servidor que permitiría cómo máximo 40 clientes conectados en simultáneo. La migración sólo se realizaría si la probabilidad de tener más de 40 clientes es menor a 10%. Indique si se puede recomendar realizar la migración.

Sistemas dinámicos

Ejercicio 3

El siguiente sistema dinámico es conocido como “Modelo de la telaraña”.
Mediante el mismo se puede modelar la formación de precios de productos cuya oferta se establece en función del precio de mercado en el período de tiempo anterior.

$$\begin{cases} Q_t^{demanda} = a - bP_t \\ Q_t^{oferta} = dP_{t-1} - c \\ Q_t^{demanda} = Q_t^{oferta} = Q_t \end{cases}$$

Con:
a=9
b=1,1
c=0,4
d=1
P_0=8

Se pide:

1. Determinar el tipo de sistema (discreto/continuo, lineal/ no lineal, autónomo/no autónomo, de primer orden / orden superior)
2. Hallar el o los puntos de equilibrio.
3. Realizar un análisis asintótico del sistema.
   ¿Cómo afectan las condiciones iniciales al estado asintótico del sistema? Justifique.
   Simule el sistema para distintas condiciones iniciales e interprete los resultados.
4. Graficar la variable precio en función del período (t) para 100 períodos.
5. Graficar el espacio de faces del sistema a través de 100 períodos para pares ordenados (Q_t , P_t) (mostrar los puntos vinculados mediante una recta).

Simpy

Ejercicio 4

Resolver el Ejercicio 1 utilizando Simpy y comparar los resultados obtenidos con los obtenidos en el Ejercicio1.

Ejercicio 5

Siguiendo una distribución Exponencial negativa de media 45 ms arriban solicitudes a un cluster de base de datos, compuesta por 6 servidores y un balanceador de carga.
El tiempo de procesamiento de cada solicitud dependerá del tipo de solicitud que se trate:

Tipo	Probabilidad	Tiempo de proceso (mseg)
A	.6	75 ± 10
B	.25	60 ± 15
C	.15	90 ± 20

Se necesita determinar la mejor política de asignación de procesos a utilizar en el balanceador entre las siguientes:

a. Utilizando una política Round Robin (la primer solicitud se asigna al servidor 1, la segunda al 2, etc).
b. La solicitud es asignada al azar entre los 6 servidores.

(Justifique la respuesta midiendo todos los indicadores que considere necesarios)

2. Implementación y resultados

Para cada uno de los ejercicios pedidos se realiza una explicación de cada uno de ellos. Se toma como base teórica lo explicado en clase tanto teórica como clase práctica.

2.1. Ejercicio 1

Luego de ejecutar 5 veces el programa OS.py se obtuvieron los siguientes resultados (promedios en microsegundos de 20 iteraciones):

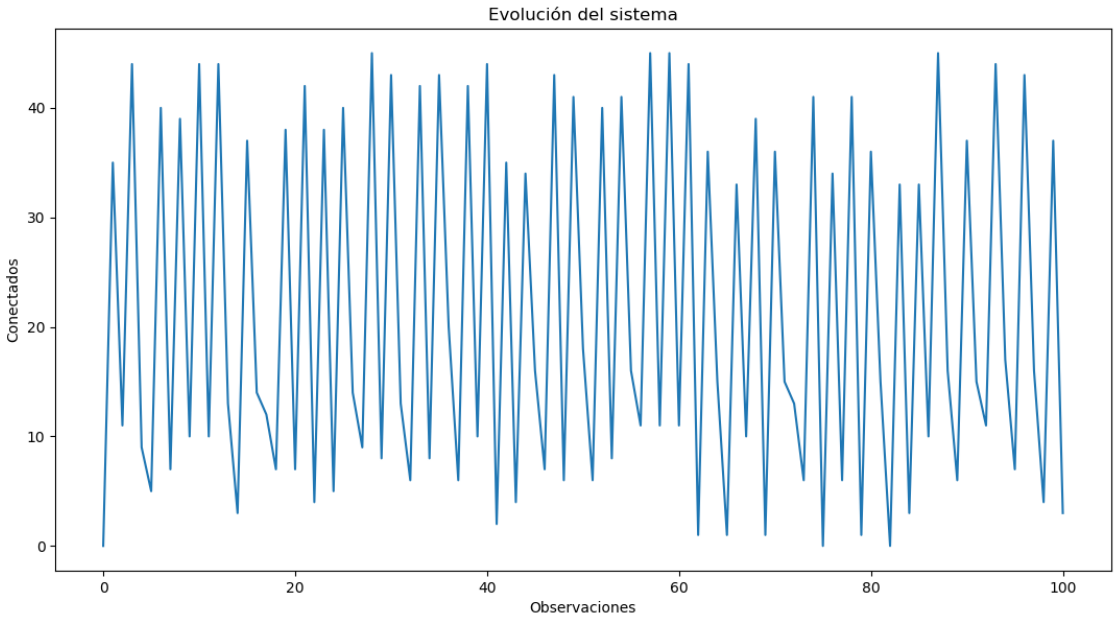
ejecucion	costo alt 1	costo alt 2
1	5434.09	5495.7
2	5436.64	5495.11
3	5436.59	5495.32
4	5424.47	5495.64
5	5430.06	5494.5

La alternativa 1 es mejor.

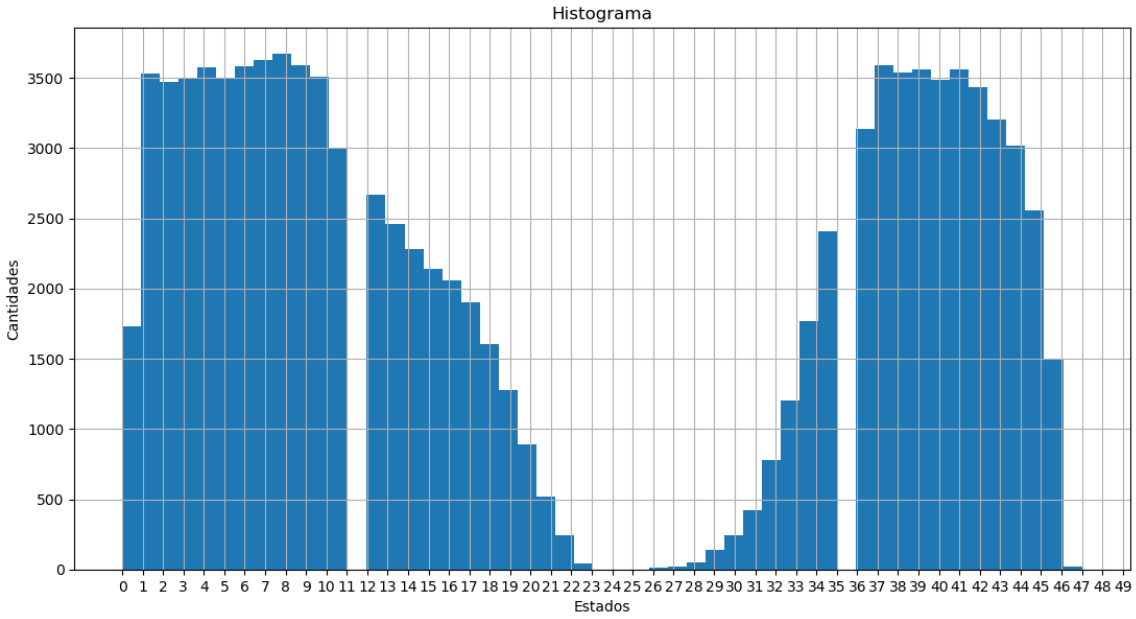
2.2. Ejercicio 2

a) La matriz de transición de estados se obtiene de la siguiente manera, dado que el ejercicio impone por simplicidad que la cantidad máxima de clientes que se permite en simultaneo en el Homebanking es 50, entonces la matriz de transición es de 51 x 51 (Tambien se incluye la cantidad de 0 clientes). Cada elemento de la matriz es el cambio de estado al siguiente, es decir el paso del estado i al j, por ejemplo si el elemento de la matriz es el (1,2) entonces es la probabilidad de la cantidad de clientes conectados sea 2 sabiendo que en el minuto anterior estaba conectado solo 1.

b)



c)



d) Porcentaje de tiempo que el sistema no tuvo clientes conectados: 2,24531353 %

e) Es recomendable hacer la migración a otro servidor, la probabilidad de tener mas de 40 clientes es: 2,15 %

2.3. Ejercicio 3

El "modelo de la telaraña"tiene la siguiente estructura:

$$\begin{aligned} Q_t^{demanda} &= a - bP_t \\ Q_t^{oferta} &= dP_{t-1} - c \\ Q_t^{demanda} &= Q_t^{oferta} = Q_t \end{aligned} \tag{1}$$

En donde tienen los parámetros siguientes :

- a = 9
- b = 1,1
- c = 0,4
- d = 1
- $P_0 = 8$

Respondiendo lo que se pide:

1. El sistema es dicreto, lineal, autónomo y de primer orden
2. Resolviendo el sistema de ecuaciones se obtuvo el punto de equilibrio:

   $$P = \frac{a+c}{d+b}$$

   Utilizando los datos del ejercicio se obtuvo:

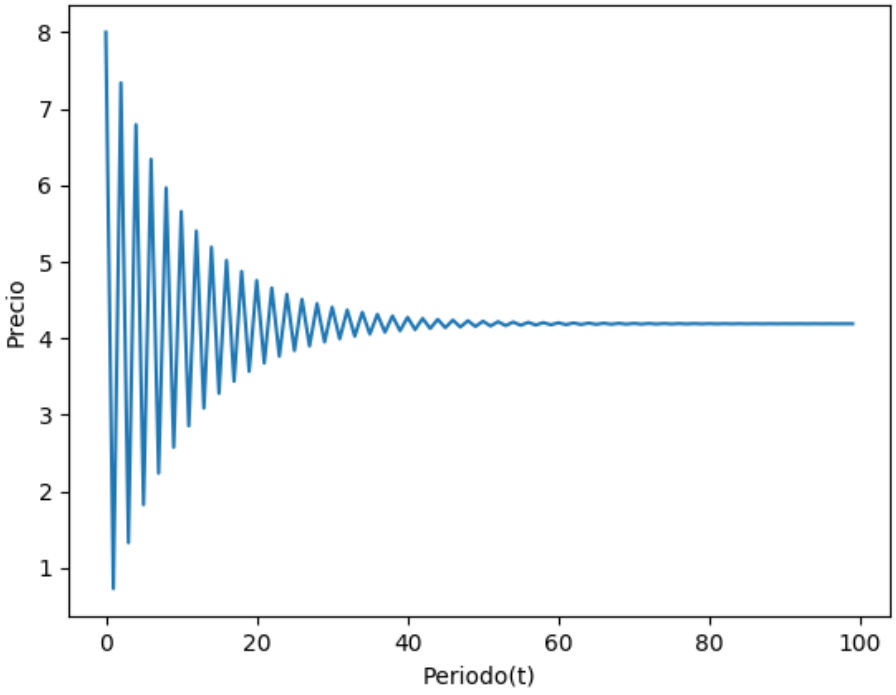
   $$P = \frac{94}{21}$$

   $$Q = \frac{428}{105}$$
3. Realizando un análisis asintótico obtuvimos que:

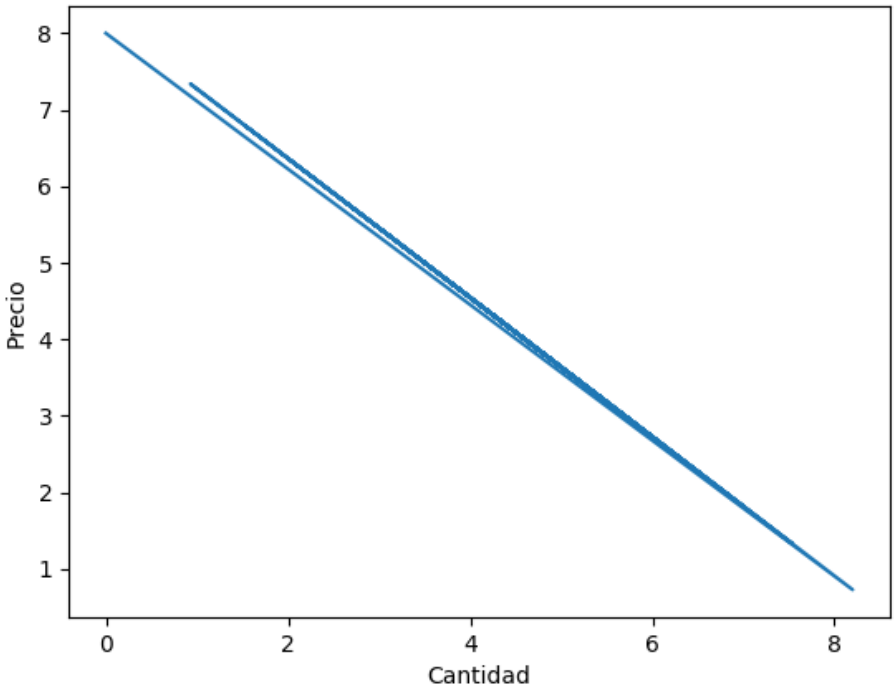
   $$\lambda_1 = 0$$

   $$\lambda_2 = \frac{10}{11}$$

   En donde se llega a que el sistema es un sistema estable.
4. Graficando la variable precio en función del período(t):



5. Graficando el espacio de fases del sistema a través de 100 períodos para pares ordenados (Q_t , P_t):



2.4. Ejercicio 4

La conclusion es consistente con el ejercicio 1, es mejor la alternativa 2. Sin embargo por como esta implementado el ejercicio 1 nos dio distinto la cantidad de tiempo que lleva ejecutar 200000 instrucciones, para la alternativa 1 nos dio 21837124 microsegundos (alrededor de 20 segundos) y para la alternativa 2 7422610 microsegundos (alrededor de 7 segundos)

2.5. Ejercicio 5

Utilizando los datos del ejercicio procesamos nuestro archivo creado usando *Simpy* y obtuvimos los siguientes resultados:

- Con política Round Robin(en segundos):
  - Tiempo espera: 73.5
  - Tiempo max: 140.6
  - Tiempo min: 45.0
  - Tiempo total: 453896.8
- Las solicitudes son asignadas al azar entre los servidores:
  - Tiempo espera: 73.7
  - Tiempo max: 148.9
  - Tiempo min: 45.0
  - Tiempo total: 449154.5

Podemos observar que la mejor política de asignación es la de Round Robin. Hay una diferencia significativa en el tiempo total de procesamiento y en el tiempo máximo entre ambos.

3. Conclusiones

El trabajo práctico nos permitió conocer y realizar simulaciones teniendo como base teórica los conceptos explicados en clase . Además, nos permitió conocer herramientas que permiten realizar simulaciones que son muy utilizadas en el campo científico como Simpy que un framwork de simulación discreta orientada a eventos.

Referencias

[1] Python, Generación de números con distintas distribuciones de probabilidad, https://docs.python.org/3/library/random.html.

[2] Simpy, Apunte de la cátedra.

[3] Simpy, Documentación, https://simpy.readthedocs.io/en/latest/

A. Código fuente

A.1. Resolución ejercicio 1

A.1.1. instruccion

```
import random
import numpy

PROBA_ACCEDER_MEMORIA = 0.65
PROBA_SER_SIMPLE = 0.6
PROBA_ACCEDER_CACHE = 0.6
TASA_ACCESO_CACHE = 500
TASA_ACCESO_MEMORIA_CON_CACHE = 1500
TASA_ACCESO_MEMORIA_SIN_CACHE = 2000
TASA_EJECUCION_INSTRUCCION_SIMPLE = 60
TASA_EJECUCION_INSTRUCCION_COMPLEJA = 10

class InstruccionFactory(object):

  @staticmethod
  def nueva_instruccion():
    tipo = random.uniform(0,1)
    memoria = random.uniform(0,1)
    lee_memoria = False
    if (memoria <= PROBA_ACCEDER_MEMORIA):
      lee_memoria = True
    if (tipo <= PROBA_SER_SIMPLE):
      return InstruccionSimple(lee_memoria)
    else:
      return InstruccionCompleja(lee_memoria)

class Instruccion(object):

  def __init__(self, lee_memoria, tasa):
    self.lee_memoria = lee_memoria
    self.tasa_tiempo_ejecucion = tasa

  def costo(self, modo):
    costo = numpy.random.exponential((1/self.tasa_tiempo_ejecucion))
    if self.lee_memoria and modo == "CACHE":
      u = random.uniform(0,1)
      if u <= PROBA_ACCEDER_CACHE:
        costo += numpy.random.exponential((1/TASA_ACCESO_CACHE))
      else:
        costo += numpy.random.exponential((1/TASA_ACCESO_MEMORIA_CON_CACHE))
    elif self.lee_memoria: #Alt 1
      costo += numpy.random.exponential((1/TASA_ACCESO_MEMORIA_SIN_CACHE))
    return costo

class InstruccionCompleja(Instruccion):

  def __init__(self, lee_memoria):
    super().__init__(lee_memoria, TASA_EJECUCION_INSTRUCCION_COMPLEJA)

class InstruccionSimple(Instruccion):

  def __init__(self, lee_memoria):
    super().__init__(lee_memoria, TASA_EJECUCION_INSTRUCCION_SIMPLE)
```

A.1.2. main

```
1
2  TASA_ARRIBO_INSTRUC = 250
3  n = 100000 #cantidad de instrucciones que genero
4
5  iteraciones = 20
6
7  def run(modo):
8    import instruccion
9    import numpy
10   if modo is not "CACHE" and modo is not "RAM":
11     print("MODO INCORRECTO")
12     return
13
14   generador_instrucciones = instruccion.InstruccionFactory()
15   tiempo_proceso = float(0)
16   for x in range(n):
17     tiempo_de_espera_arribo_instruc = numpy.random.exponential((1/TASA_ARRIBO_INSTRUC
       ))
18     instruccion = generador_instrucciones.nueva_instruccion()
19     tiempo_proceso += tiempo_de_espera_arribo_instruc + instruccion.costo(modo)
20   return tiempo_proceso
21
22 print("Corriendo alternativa 1 20 veces")
23 tiempo_total_alt1 = 0
24 for x in range(iteraciones):
25   tiempo = run("RAM")
26   tiempo_total_alt1 += tiempo
27 print("Tiempo promedio alternativa 1:" + str(round(tiempo_total_alt1/iteraciones, 2))
     )
28
29 print("Corriendo alternativa 2 20 veces")
30 tiempo_total_alt2 = 0
31 for x in range(iteraciones):
32   tiempo = run("CACHE")
33   tiempo_total_alt2 += tiempo
34 print("Tiempo promedio alternativa 2:" + str(round(tiempo_total_alt2/iteraciones, 2))
     )
```

A.2. Resolución ejercicio 2

A.2.1. ejercicio2.py

```
import os
import numpy
import random
import matplotlib.pyplot as plt

cant=51
p = 0.7

def factorial(n):
        if (n==0):
                return 1
        else:
                return n* factorial(n-1)

def combinatoria(n,r):
        return factorial(n)/(factorial(r) * factorial(n-r))


def jMayorIgualAi(i,j):
        return combinatoria(cant,j-i) * p**(j-i) * (1-p) **(cant-(j-i))

def jMenorAi(i,j):
        sumatoria =0
        for k in range(i,cant):
                 sumatoria = sumatoria + jMayorIgualAi(i,k)
        return (1-sumatoria)/i

def funcionProbabilidad(i,j):
        if (i<=j):
                probabilidad = jMayorIgualAi(i,j)
        else:
                probabilidad = jMenorAi(i,j)
        return round(probabilidad,4)

def generarMatriz():
        archivo = open("matrizDeTransicion.txt","w")
        cadena = ''
        matriz=[]
        for m in range (0,cant):
                fila=[]
                for n in range (0,cant):
                        valor=funcionProbabilidad(m,n)
                        fila.append(valor)
                        cadena = cadena + str(valor) + ','
                archivo.write(cadena + os.linesep)
                matriz.append(fila)
                cadena=''
        archivo.close()
        return matriz

#a) genero la matriz de transición
matrizDeTransicion = generarMatriz()

# La matriz de transición de estados se obtiene de la siguiente manera, dado que el
    ejercicio impone por simplicidad que la cantidad máxima
# de clientes que se permite en simultaneo en el Homebanking es 50, entonces la
    matriz de transición es de 51 x 51 (Tambien se incluye la
# cantidad de 0 clientes).
# Cada elemento de la matriz es el cambio de estado al siguiente, es decir el paso
```

```
    del estado i al j, por ejemplo si el elemento de la
# matriz es el (1,2) entonces es la probabilidad de la cantidad de clientes
    conectados sea 2 sabiendo que en el minuto anterior estaba
# conectado solo 1.

#b)

def generarOtroEstado(Lista):
        aleatorio = random.random()
        sumaTotal = Lista[0]
        i=0
        while(aleatorio > sumaTotal ):
                if(i<50):
                        sumaTotal = sumaTotal + Lista[i]
                        i=i+1
                sumaTotal = sumaTotal + Lista[i]
        return i

estados = []
estadoActual = 0
estados.append(estadoActual)
observaciones = 100

#simulo los valores de los estados en cada momento
for i in range(observaciones):
        estadoActual = generarOtroEstado(matrizDeTransicion[estadoActual])
        estados.append(estadoActual)

#grafico de como se modifican los clientes a lo largo de las observaciones.
plt.plot(estados)
plt.title("Evolución del sistema")
plt.xlabel('Observaciones')
plt.ylabel('Conectados')
plt.show()

#c)
estados = []
estadoActual = 0
estados.append(estadoActual)
observaciones = 100000

#simulo los valores de los estados en cada momento
for i in range(0,observaciones):
        estadoActual = generarOtroEstado(matrizDeTransicion[estadoActual])
        estados.append(estadoActual)

#Histograma
plt.title('Histograma')
plt.xlabel('Estados')
plt.ylabel('Cantidades')
plt.xticks(numpy.arange(50))
plt.hist(estados, bins =51)
plt.grid(True)
plt.show()

#d)

# funcion que devuelve el estado estacionario del sistema
def steady_state_prop(matriz):
        dimension = matriz.shape[0]
        matrizMenosIdentidad = (matriz-numpy.eye(dimension))
        vectorDeUnos = numpy.ones(dimension)
        matrizMenosIdentidad = numpy.c_[matrizMenosIdentidad,vectorDeUnos]
```

```
119         matrizXTranspuesta = numpy.dot(matrizMenosIdentidad, matrizMenosIdentidad.T)
120         return numpy.linalg.solve(matrizXTranspuesta,vectorDeUnos)
121
122 matrizDeTransicion = numpy.matrix(matrizDeTransicion)
123 estadoEstacionario = steady_state_prop(matrizDeTransicion)
124
125 # porcentaje de tiempo en que el sistema no tuvo clientes conectados
126 print ('Porcentaje de tiempo que el sistema no tuvo clientes conectados: %.8f%%' % (
        float(estadoEstacionario[0] * 100)))
127
128
129 #e)
130 #P(i-->j >=40) <= 0.1
131
132 probabilidad=0
133 divisor=0
134 for i in range(0,51):
135         for j in range(40,51):
136                 probabilidad = probabilidad + funcionProbabilidad(i,j)
137                 divisor=divisor+1
138 probabilidad = probabilidad/divisor
139
140 if (probabilidad < 0.1):
141         print("Es recomendable hacer la migración a otro servidor, la probabilidad de
        tener mas de 40 clientes es:" , probabilidad)
142 else:
143         print("No es recomendable hacer la migración a otro servidor, la probabilidad
        de tener mas de 40 clientes es:" , probabilidad)
```

A.3. Resolución ejercicio 3

A.3.1. ejercicio3.py

```
#!/usr/bin/env/ python

import matplotlib.pyplot as plt
from matplotlib.pyplot import plot
import matplotlib.colors
from mpl_toolkits.mplot3d import Axes3D

Pt = [8]
Qt = [1/5]

def calculoPt(valor_pt):
        return ((-0.909*valor_pt) + (94/11))

def fully_price(Pt):
        for i in range(1,100):
                Pt.append(calculoPt(Pt[i-1]))
        return Pt

#Calculamos los valores de Qt
def calculoQt(valor_pt):
        return (9 - (1.1*valor_pt))

def cantidadQt(Pt):
        for i in range(1,100):
                Qt.append(calculoQt(Pt[i]))
        return Qt

prices = fully_price(Pt)
#Graficamos el precio en funcion del tiempo
t = range(0,100)
plt.xlabel('Periodo(t)')
plt.ylabel('Precio')
plt.plot(t, [ prices[i] for i in t ])
plt.show()

#Hacemos el diagrama de fases del sistema
cantidad = cantidadQt(prices)
plt.xlabel('Cantidad')
plt.ylabel('Precio')
plt.plot(cantidad,prices)
plt.show()
```

A.4. Resolución ejercicio 4

A.4.1. ejercicio4.py

```
1
2  import simpy
3  import random as rnd
4  import numpy
5  import instruccion
6  generador_instrucciones = instruccion.InstruccionFactory()
7
8  tiempos = []
9  SECOND_TO_MICROSECOND = 1000000
10
11 class Procesamiento:
12         def __init__(self, env, tiempos_procesamientos ,instruc, cache ):
13
14                 self.env = env
15                 self.instruc = instruc
16                 self.tiempo_ejecucion = self.calcular_tiempo_ejecucion(cache)
17                 self.tiempo_procesamiento = None
18                 self.registro_procesamientos= tiempos_procesamientos
19
20
21
22         def ejecutar(self, procesador):
23                 with procesador.request() as req:
24                         self.tiempo_procesamiento = env.now
25                         yield req
26                         yield self.env.timeout(self.tiempo_ejecucion)
27                         self.tiempo_procesamiento = env.now - self.
       tiempo_procesamiento
28                         self.registro_procesamientos.append(self.tiempo_procesamiento
       )
29
30         def calcular_tiempo_ejecucion(self,cache):
31                 tiempo_ejecucion = 0
32
33                 if self.instruc.lee_memoria:
34
35                         if not cache:
36                                 tiempo_ejecucion += numpy.random.exponential(float(
       SECOND_TO_MICROSECOND)/2000)
37
38
39                         elif cache:
40
41                                 en_cache = (rnd.uniform(0,1) <= 0.6)
42
43                                 if not en_cache:
44                                         tiempo_ejecucion += numpy.random.exponential(
       float(SECOND_TO_MICROSECOND)/1500) #busqueda en memoria
45                                 else:
46                                         tiempo_ejecucion += numpy.random.exponential(
       float(SECOND_TO_MICROSECOND)/500) #busqueda en cache
47
48
49                 tiempo_ejecucion += numpy.random.exponential(self.instruc.costo())
50
51                 return tiempo_ejecucion
52
53
54
```

```
55 def procesar_instrucciones(environment, cantidad, cache):
56         procesador = simpy.Resource(env, capacity = 1)
57         for i in range(cantidad):
58                 instruc = generador_instrucciones.nueva_instruccion()
59                 proc = Procesamiento(env, tiempos,instruc, cache)
60                 environment.process(proc.ejecutar(procesador))
61                 tiempo_prox_instruc = numpy.random.exponential(float(
    SECOND_TO_MICROSECOND)/250)
62                 yield environment.timeout(tiempo_prox_instruc)
63
64
65
66 cantidad = 200000
67
68 env = simpy.Environment()
69 env.process(procesar_instrucciones(env, cantidad, False))
70 env.run()
71 print(sum(tiempos))
72 print("Tiempo total de ejecucion alternativa 1 (microsegundos) 200000 instrucciones:
    ", round(sum(tiempos)))
73
74
75 tiempos = []
76 env = simpy.Environment()
77 env.process(procesar_instrucciones(env, cantidad, True))
78 env.run()
79 print("Tiempo total de ejecucion alternativa 2 (microsegundos) 200000 instrucciones:
    ", round(sum(tiempos)))
```

A.5. Resolución ejercicio 5

A.5.1. ejercicio5.py

```
import random
from random import randint
import simpy
import numpy
from numpy import mean


client_count = 10000
arrival_rate = 45
tiempo_esperas = []

class Client(object):

    def process_duration(self):
        if (self.type == 'A'):
            return (75 + randint(-10, 10))
        if (self.type == 'B'):
            return (60 + randint(-15, 15))
        return (90 + randint(-20, 20))


    def __init__(self, env):
        self.tiempo_entrada = env.now
        self.env = env
        self.type = numpy.random.choice(['A', 'B', 'C'], p=[0.6, 0.25, 0.15])

    def process(self, cashier):
        global tiempo_esperas
        with cashier.request() as req:
            process_duration = self.get_process_duration()
            yield req
            yield self.env.timeout(process_duration)

        # print("%.2f Client type %s attended" % (self.env.now, self.type))
        tiempo_esperas.append(env.now - self.tiempo_entrada)

    def get_process_duration(self):
        return self.process_duration()

class Balanceador(object):

    def __init__(self, round_robin):
        self.resources = []
        self.round_robin = round_robin
        self.last_assigned = 0

        for i in range(6):
            self.resources.append(simpy.Resource(env, capacity = 1))
            self.cantidad_colas = 6


    def get_resource(self):
        #If we use Round Robin politic
        if self.round_robin:
            self.last_assigned = (self.last_assigned + 1) % 6
            return self.resources[self.last_assigned]
        else:
            # If we assign randomly
            random_value = randint(0,5)
```

```
60              return_value = self.resources[random_value]
61              return return_value
62
63
64  def generate_clients(environment, count, interval):
65      #Select one the options in the exercise
66      balanceador_round_robin = Balanceador(False)
67      balanceador_2 = Balanceador(True)
68
69      balanceador = balanceador_2
70
71      for i in range(count):
72          client = Client(env)
73          environment.process(client.process(balanceador.get_resource()))
74          t = random.expovariate(1.0 / interval)
75          yield environment.timeout(t)
76
77
78  env = simpy.Environment(
79  env.process(generate_clients(env, client_count, arrival_rate))
80  env.run()
81
82  print("Tiempo espera: %s" % mean(tiempo_esperas))
83  print("Tiempo max: %s" % max(tiempo_esperas))
84  print("Tiempo min: %s" % min(tiempo_esperas))
85  print("Tiempo total: %s" % env.now)
```