Simulación

Conceptos básicos

Moisés Saavedra Cáceres
mmsaavedra1@ing.puc.cl

¿Qué es la simulación?

Simulación puede ser definido como un proceso numérico diseñado para experimentar el comportamiento de cualquier sistema en una computadora digital, a lo largo de la dimensión tiempo.

En los procesos de simulación de cualquier sistema se deben definir los siguientes parámetros:

Componente: Cualquier parte importante del sistema, su símil en Python son las clases.

Atributo: Se refiere a las propiedades de cualquier componente del sistema, su símil en Python son los atributos de las clases.

Actividad: Cualquier proceso que causa cambios en el sistema, su símil en Python son las funciones y métodos de clase.

Estado del sistema: Descripción de las componentes, sus atributos y actividades de un sistema, en un determinado periodo de tiempo.

Un ejemplo práctico puede ser lo siguiente:

Sistema	Componentes	Atributos	Actividades
Comercio	Clientes	Lista de compras	Comprar
	Productos	Inventario	Pagar
			Reembolso

Pasos a seguir en un proceso de simulación

Planificar un proceso de simulación requiere los siguientes pasos:

a) Formulación del problema.

b) Recolección y procesamiento de la información requerida.

c) Formulación del modelo matemático.

d) Evaluación de las características de la información procesada.

e) Formulación de un programa de computadora.

f) Validación del programa de computadora.

g) Diseño de experimentos de simulación.

h) Análisis de resultados y validación de la simulación.

Generación de números aleatorios

Los números aleatorios se utilizan para introducir el comportamiento estocástico en el sistema bajo estudio. Estos números pueden encontrarse en tablas o generar con computadoras. Para efectos práticos de simulación tendremos que generarlos en cada programa de simulación que implementemos en nuestras computadoras. Estos números **no son del todo aleatorios** porque se obtienen de fórmulas establecidas que cumplen una serie de pruebas estadísticas de aleatoriedad. Por esta razón a estos números se les llama *pseudoaleatorios*.

La mayoría de los métodos para generar números aleatorios son iterativos, donde un número pseudoaleatorio se genera del anterior.

Método de la transformada inversa

Para cada variable aleatoria x con una distribución cualquiera $F(x)$ existe una variable aleatoria r, única, distribuida uniformemente, tal que:

$$F(x) = r$$

De hecho, r es la probabilidad de que una variable aleatoria con distribución cualquiera $F(\dot{})$, tenga un valor menor a x.

Cuando es posible encontrar la función inversa $F^{-1}(r) = x$, se pueden generar variables aleatorias con la distribución $F(\dot{})$, a partir de varibales aleatorias r, distribuidas uniformemente en el intervalor cerrado [0,1].

En palabras más simples: se debe conocer $F(x)$, que es la función de distribución deseada y se genera un número aleatorio r_o de una **distribución uniforme** en el intervalo [0,1]. Finalmente, el número aleatorio deseado simplemente se calcula como: $x_0 = F^{-1}(r_0)$.

Ejemplo: Se quieren generar números aleatorios t distribuidos exponencialmente con media $1/\lambda$. Se tiene entonces que:

$$F(x) = 1 - e^{-\lambda x} = r$$

Luego calculamos su función inversa, quedando:

$$\begin{aligned} 1 - r &= e^{-\lambda x} \\ log(1-r) &= -\lambda x \\ F^{-1}(r) &= -\frac{1}{\lambda} log(1-r) \end{aligned}$$

Como sabemos que:

$$\begin{aligned} x &= F^{-1}(r) \\ x &= -\frac{1}{\lambda} log(1-r) \end{aligned}$$

Por último se genera un número aleatorio distribuido uniformemente entre el intervalo [0,1] el que será almacena en r, luego dicho r se evalua como $F^{-1}(r)$. Lo anterior se puede adaptar a algún software computacional y de esta forma se obtendrían valores de la variables aleatoria x que tiene como función de distribución a $F()$.

Método de aceptación y rechazo

¿Qué pasa si la función de distribución no puede ser invertida de forma eficiente? Esta es la pregunta que abre debate en formular métodos de generación alternativos al anterior visto, ya que muchas veces la inversa es un tanto difícil de hayar o despejar.

El ḿetodo de aceptación y rechazo tiene como fin la generación de variables aleatorias a partir de una **función "parecida"**, de los que solamente se aceptarán algunos valores para representar los buscados efectivamente.

Hay que partir de la base que tenemos una variables aleatoria X cuya **función de densidad** es $f : I \to \mathrm{R}$ (I es el dominio de valores que toma f). De aquí supongamos que $f(x)$ puede se descompuesta de la siguiente forma:

$$f(x) = Cg(x)h(x)$$

$$\begin{aligned} C &: \text{ constante} \geq 1 \\ g(x) &: \ 0 < g(x) \leq 1 \\ h(x) &: \text{ función densidad en conjunto } I \end{aligned}$$

Ahora que hemos definido a $f(x)$ como la ponderación de la multiplicación de las funciones $g(x)$ y $h(x)$ vamos a definir un nuevo aliado que llamaremos $t(x)$. Esta función tiene la característica de aproximar (por arriba) a nuestra función $f(x)$ de estudio, por lo que $t(x)$ debemos elegirla siendo cuidadosos de que sea de una composición fácil para los cálculos y que cumpla con:

$$t(x) : I \to R$$
$$f(x) \leq t(x)$$
$$1 = \int_I f(x) \leq \int_I t(x) = C < \infty$$

Ahora que hemos definido las funciones $g(x)$, $h(x)$ y $t(x9$, junto con la constante C. Definimos lo siguiente:

$$g(x) = \frac{f(x)}{t(x)}$$
$$h(x) = \frac{t(x)}{C}$$

Recordamos las propiedades definidas arriba:

$$C: \text{ constante} \geq 1$$
$$g(x): \ 0 < g(x) \leq 1$$
$$h(x): \text{ función densidad en conjunto } I$$

Teorema 1 (von Neumann, 1951) *Sea la función $g(x)$ (notar que $0 < g(x) \leq 1$). Sean U, Y dos varibles aleatorias independientes tales que $U \sim U(0,1)$ e Y posee función densidad $h(y)$.*
La variables aleatoria definida como Y condicionada a que $U \leq g(Y)$ distribuye según la funcionón de densidad $f(y)$.

Lo anterior puede resultar un tanto enredado, pero simplemente se resume (matemáticamente hablando) a la siguiente expresión:

$$f(x) = h(x/U \leq g(Y))$$

Entonces, el algoritmo que se debe implementar en las simulaciones es el siguiente:

1. Generar una instancia de $U \sim U(0,1)$.
2. Generar una instacia de $Y \sim h(y)$
3. Si $U \leq g(Y)$, entonces hacer $X = Y$. Si no, volver al paso 1.

Cabe destacar que en lo anterior nuestras funciones auxiliares g(y) y h(y) las conocemos.

Ahora hay tres factores muy importante que se debe tener en mente al momento de ocupar este método:

- Se debe poder generar fácilmente instancias a partir de $h(y)$, es decir la función $t(x)$ (aproximante por arriba) debe ser "sencilla".
- El tamaño de C esta relacionado en que tan cercana es la aproximación de $t(x)$ a $f(x)$. Por lo que se desea que su valor sea pequeño.
- El número esperado de iteraciones hasta obtener una instancia que nos sirva es el mismo valor de C. Por lo tanto se busca el mínimo valor de C lo que se traduce en hayar el máximo valor de $f(x)$, ya que s ecumple que $f(x) \leq Ch(x)$

Ejemplo: Sea la función $60x^3(1-x)^2, 0 \leq x1$. Genera una instancia de una v.a cuya función de densidad es $f(x)$. Nota: si debe generar instancias de $U(0,1)$, asuma que se tienen dos a priori: 0.13 y 0.25.

Paso 1: Debemos encontar el máximo de la función $f(x)$ (derivar e igualar a cero)

$$x_{max} = 0.6$$
$$f(0.6) = 2.0736$$

Paso 2: Definimos la función que acote por arriba como $t(x) = 2.0736$ y obviamente en el mismo dominio de $f(x)$, es decir $0 \leq x \leq 1$.

Paso 3: Calculamos el valor de C y como se mencionó arriba es simplemente la integral de $t(x)$ sobre todo su dominio:

$$c = \int_0^1 t(x)dx = 2.0736$$

Paso 4: Definimos la función ponderante $h(x)$ queda como:

$$h(x) = \frac{t(x)}{C} = 1$$

Paso 5: Definimos la otra función ponderante $g(x)$ queda como:

$$g(x) = \frac{f(x)}{t(x)} = \frac{60x^3(1-x)^2}{2.0736}$$

Paso 6: Ahora se ocupan las instancias (generadas o entregadas, según sea el caso) y comparamos:

$$U = 0.13$$
$$Y \sim h(y) = 0.25 \text{ instancias¡ obtenida a partir de } h(y)$$
$$U \leq g(Y) = \frac{60(0.25)^3(1-(0.25))^2}{2.0736} = 0.254 \text{ se cumple!}$$

Pao 7: Aceptamos el valor $Y = 0.25$, por lo que tenemos una instancia para nuestra variable aleatorias X que es: $X = 0.25$.

Método aceptación y rechazo: Caso especial distribución normal

Si queremos generar instancias de una variable aleatoria X con distribución normal $N(\mu, \sigma^2)$, primero tenemos que estandarizar quedando la variable aleatoria:

$$Z = \frac{X-\mu}{\sigma}$$
$$X = Z\sigma + \mu$$
$$Z \sim N(0,1)$$

Es por lo anterior que simplemente debemos pensar en un algoritmo para la variable aleatoria Z. Lo anterior se puede simpliciar aún más, ya que si consideramos solamente la parte derecha de la función, es decir, solo valores positivos se tiene que generar instancias para $\|Z\|$. Luego, ocupando el argmuento de simetría de la función densidad se ocupa una variables aleatoria auxiliar S que tiene las siguientes caráctirísticas:

- Toma el valor 1 con probabilidad 1/2
- Toma el valor -1 con probabilidad 1/2

De esta forma tenemos que:

$$Z = S\|Z\| \sim N(0,1)$$

Ahora tenemos que parra $\|Z\|$ su función densidad es:

$$f(x) = \frac{2}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} \ , \ x \geq 0$$

El factor 2 viene dado por la condición de valor absoluto y la paridad de la función.

Ahora definimos la función $t(x)$ que acota superiormente como:

$$t(x) = \sqrt{\frac{2e}{\pi}} e^{-x}$$

Lo anterior acota superiormente ya que podemos notar que:

$$\begin{aligned} \sqrt{\frac{2}{\pi}} e^{\frac{-x^2}{2}} &\leq \sqrt{\frac{2e}{\pi}} e^{-x} \\ e^{\frac{-x^2}{2}} &\leq e^{\frac{1}{2}} e^{-x} \\ -x^2 &\leq 1 - 2x \\ x^2 &\geq 2x - 1 \text{ , lo que se cumple } \forall x \geq 0 \end{aligned}$$

Con esto tenemos que la funcion $h(x)$ es de la forma:

$$h(x) = e^{-x} \ , \ x \geq 0$$

Como nos resulta una exponencial en $h(x)$ aquí podemos ocupar el método de la funcón inversa para generar las instancias necesarias para hacer el análisis de m^o'etodo y rechazo (ocupo un méotodo dentro de otro).

Y como es trivial notar por fórmula $h(x) = \frac{t(x)}{C}$ tenemos que el valor de la constante es:

$$C = \sqrt{\frac{2e}{\pi}} \approx 1.32$$

Así de esta forma tenemos que la función $g(y)$ que ocupamos para hacer la comparación $U \leq g(Y)$ queda como:

$$\begin{aligned} g(y) &= \frac{f(y)}{t(y)} \\ g(y) &= e^{\frac{-(y-1)^2}{2}} \end{aligned}$$

Si se cumple el criterio de comparación $U \leq g(Y)$, tenemos que $\|Z\| = Y$, luego de esto generamos una instancia de una variable aleatoria auxiliar $V \sim U(0,1)$ y la ocupamos para darle valor a la variable S y definir o bien Z o $-Z$ para finalmente reemplazar en:

$$X = Z\sigma + \mu$$

Entonces, el algoritmo que se debe implementar en las simulaciones es el siguiente:

1. Generar una instancia de $U \sim U(0,1)$.
2. Generar Y con una distribución exponencial de tasa 1 ocupando el método de la inversa.
3. Si $U \leq e^{\frac{-(Y-1)^2}{2}}$, sea $\|Z\| = Y$, si no volver a 1.
4. Generar instancias $V \sim U(0,1)$. Sea $Z = \|Z\|$ si $V \leq 0.5$, sea $Z = -\|Z\|$ si $V > 0.5$

Implementación en Python: Exponencial ocpuando el método de la función inversa

Para trabajar esto de forma computacional se puede ocupar el software Python en el que debemos importar dos módulos matemáticos: *numpy* y *matplotlib.pyplot*.

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

Una vez importados debemos definir una función que sea la generadora de nuestras variables aleatorias uniformes la que queda de la siguiente manera:

```
def instancias_de_exponenciales(_lambda):
    """Funcion que genera variables aleatoias y returna su valor
    evaluado en la funcion inversa de la densidad"""
    u = np.random.uniform(0, 1)
    f_inversa = -np.log(1-u)/_lambda

    return f_inversa
```

Luego hay que generar y almacenar todas las instancias deseadas, lo que se puede traducir a lo siguiente:

```
# Ahora se crean las cantidades de instancias que se desean obtener
# junto con los parametros necesario: tasa lambda y vector donde almacenar
# la informacion

instancias_deseadas = 10000  # Mientas + grande el numero, mejor la aproximacion
tasa = 0.1  # Corresponde a la tasa pedida por la formula densidad
x = np.zeros(instancias_deseadas)  # Vector de numeros que se va actualizando por iteracion


# Ahora se ejecuta el programa
for i in range(instancias_deseadas):
    x[i] = instancias_de_exponenciales(tasa)
```

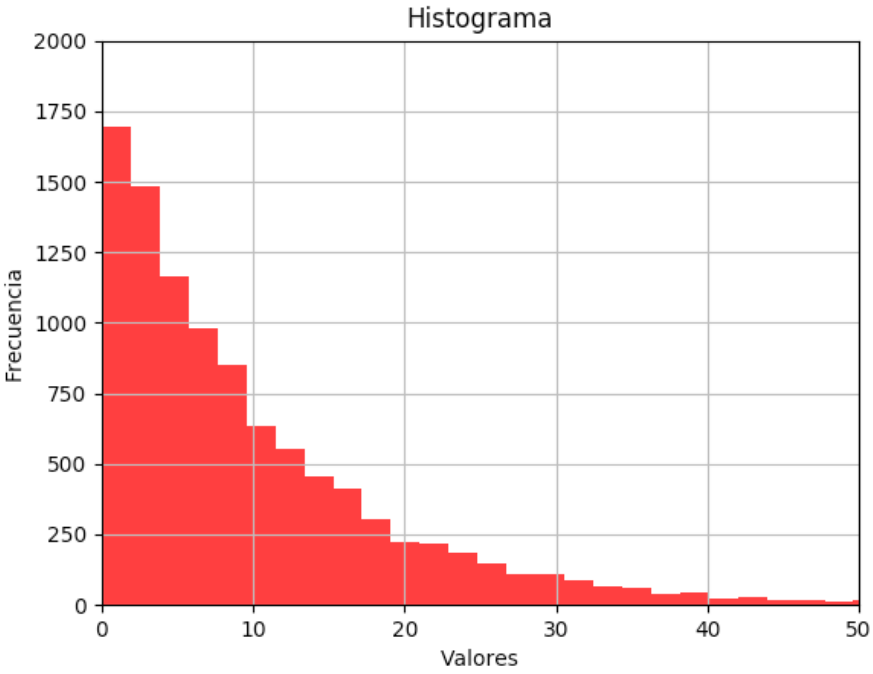
Finalmente si queremos analizar gráficamente una buena estrategia es ocupar un histograma para analizar la frecuencia de repetición de cada dato según un intervalo estipulado, la implementación en Python es la siguiente:

```
# Ahora se debe generar un histograma que refleje la frecuencia
# de cada dato que se necesita.

plt.hist(x, 50, facecolor="r", alpha=0.75)  # Se genera el objeto a graficar

plt.xlabel("Valores")  # Titulo eje x
plt.ylabel("Frecuencia")  # Titulo eje Y
plt.title("Histograma")  # Titulo del histograma
plt.axis([0,50,0,2000])  # Largo del eje x, largo del eje Y
plt.grid(True)  # Se setea la grilla
plt.show()  # Mostramos en pantalla el histograma
```

Obteniendo los siguiente resultados en dicho histograma al momento de correr el programa:



Podemos notar que mientras mayor sea la cantidad de datos, mejor es el "ajuste" a la función densidad que conocemos, esa es la lógica detrás de la generación de instancias de variables aleatorias.

Claramente lo anterior se puede extrapolar a otras funciones simplemente cambiando la inversa y los parámetros de largo del eje x e y.

Funciones de probabilidad con sus respectivas inversas

Como es de esperar hay funciones de probabilidades que no se puede encontrar su inversa de forma tan trivial, para lo anterior se deja el siguiente listado de funciones para que sea de utilidad en caso de que se deba generar números aleatorios usando algunas de ellas

Distribución uniforme

Sea t una variable con densidad uniforme en [a,b], por lo que:

$$F(t) = \begin{cases} \frac{t-a}{b-a} & \text{si } a \leq t \leq b \\ 0 & \text{si no} \end{cases}$$

$$F^{-1}(r) = a + (b-a)r \quad \text{donde } r \sim U(0,1)$$

Distribución exponencial

Sea t una variable aleatoria que distribuye exponencialmente con media $1/\lambda$, por lo que:

$$F(t) = 1 - e^{-\lambda t}$$

$$F^{-1}(r) = \tfrac{1}{\lambda} log(1-r) \quad \text{donde } r \sim U(0,1)$$

Distribución de Erlang

Sea t una variable aleatoria con distribución de Erlang, con media k/λ y varianza k/λ^2, por lo que:

$$F(t,k) = \frac{\lambda^k t^{(k-1)} e^{-\lambda t}}{(k-1)!} \quad k = 1, 2, \ldots$$

$$F^{-1}(r_1, r_2, \ldots, r_k) = -\tfrac{1}{\lambda} \log(\textstyle\prod_{i=1}^{k} r_i) \quad \text{donde } r_i \sim U(0,1)$$

Distribución normal

Sea t una variables aleatoria con distribución normal con media μ y varianza σ^2, por lo que:

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(t-\mu)^2}{\sigma^2}}$$

Entonces si $\mu = 0$ y $\sigma^2 = 1$, la función inversa queda de la forma

$$F^{-1}(r_1, r_2, \ldots, r_n) = \textstyle\sum_{i=1}^{n} \frac{r_i - \frac{n}{2}}{\sqrt{n(\frac{1}{\sqrt{12}})}} \quad \text{donde } r_i \sim U(0,1)$$

Es recomendable que n ≥ 10

Aunque si se desea analizar para un μ y σ cualquiera queda como:

$$F^{-1}(r_1, r_2, \ldots, r_n) = (\textstyle\sum_{i=1}^{12} r_i - 6)\sigma + \mu \quad \text{donde } r_i \sim U(0,1)$$

Distribución de ji-cuadrado

Sea t una variable aleatoria con distribución ji-cuadrada con n grados de libertad y densidad dada por:

$$f(t) = \frac{1}{2^{\frac{n}{2}}(\frac{n}{2}-1)!} t^{(\frac{n}{2}-1)} e^{(\frac{-t}{2})}$$

Si fijamos $k = n/2$ y $\lambda = 1/2$, su inversa queda como:

$$F^{-1}(r_1, r_2, \ldots, r_n) = -2\log(\prod_{i=1}^{n/2} r_i) \quad \text{donde } r_i \sim U(0,1)$$

Distribución de Poisson

Sea t una variable aleatoria discreta, con distribución de Poisson con media *lambda*:

$$P_r(t = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

La función inversa se obtiene formando productos de variables aleatorias uniformemente distribuidas en [0,1], denotadas como r_i, hasta que este producto sea menor que $e^{-\lambda}$, es decir, hasta que satisfaga la desigualdad:

$$\prod_{i=1}^{t} r_i \geq e^{-\lambda} > \prod_{i=1}^{t+1} r_i$$

Al cumplirse lo anterior, encontramos el valor de t en la generación de variables aleatorias.

Simulacion por eventos discretos (DES) en Python

Ahora que se tiene en mente los conceptos básicos de generación de instancias de variables aleatorias se puede generar un programa en Python para simular comportamientos de determinados sistemas, como objeto de estudio en este caso se utilizará un modelo de simulación de inventario.

Problema de Inventario

Para el siguiente modelo se tendrá un sistema que consiste en una tienda que posee un cierto stock de productos que vende a un precio r por unidad. Los clientes que demandan este producto llegan al local de acuerdo a un proceso Poisson con tasa λ y la cantidad demandada por cada una variable aleatoria que tiene distribución Uniforme(a, b). Para poder cubrir esta demanda el productor posee un inventario en el que almacena sus productos, pero este almacenaje tiene dos restricciones asociadas: La cantidad máxima de stock es S y si se tiene menos de s productos en stock ($s < S$) se pide un orden de delivery de $S - x$, donde x corresponde al número actual de productos en inventario.

Ahora se presentan los costos asociados a este sistema. Primero, tenemos $c(y)$ que corresponde al costo total que incurre pedir y productos al delivery para reponer inventario, lo anterior toma L unidades de tiempo para ser recibido en el local una vez hecha la orden de delivery. Segundo, el otro costo asociado es el de inventario, donde la empresa debe costear en h por cada producto que se mantiene en el inventario por cada día guardado.

Es importante considerar que si un cliente desea comprar más de lo disponible en el inventario, se le vende todo hasta que el inventario quede vacío, independiente si se cumple su demanda o no. Y en caso de que el inventario esté vacío y alguien desee comprar simplemente no se le vende nada.

A continuación se mostrará como simular el beneficio esperado por la tienda para un rango T de tiempo. Para comenzar se arma una estructura para modelar la situación:

Variables temporales del sistema:

Tiempo actual de simulación

Componentes

Clientes

Delivery

Atributos del sistema

Tiempo de llegada del cliente al sistema

Tiempo de llegada del delivery al sistema

Inventario actual del sistema

Cantidad ordenada de delivery

Costo total de mantención en inventario

Cosoto total de orden de delivery

Ingresos generados por la demanda de productos de cada cliente

Actividades (eventos) Aquí hay que ser muy rigurosos, ya que cada actividad depende de que que tiempo está por ocurrir primero, para esto nos fijamos entonces en:

Caso 1 Tiempo de llegada próximo cliente al sistema < Tiempo de llegada del delivery al sistema

- Actualizar costos de inventario sumando todo el tiempo que ha pasado desde el útlimo evento hasta que ocurra este.
- Actualizar el Tiempo actual como el Tiempo de llegada del próximo cleinte.
- Generar una **instancia de variable aleatoria** para la demanda (según sea su distribución de probabilidades).
- Quitar del inventario los elementos que compra efectivamente la demanda.
- Actualizar ingresos según la cantidad comprada.
- Generar un condicional:

  Si Inventario actual < s y la orden de delivery es cero: hay que definir la cantidad ordenada al delivery como (S - inventario actual), y el Tiempo de llegada del delivery al sistema como (tiempo actual + L).
- Generar un nuevo Tiempo de llegada del cliente al sistema a partir de una instancia de variable aleatoria.

Caso 2 Tiempo de llegada del delivery al sistema ≤ Tiempo de llegada próximo cliente al sistema

- Actualizar costos de inventario sumando todo el tiempo que ha pasado desde el útlimo evento hasta que ocurra este.
- Actualizar el Tiempo actual como el Tiempo de llegada del próximo cleinte.
- Actualizar costos de delivery según la cantidad ordenada.
- Actualizar el inventario como (inventario actual + cantidad ordenada al delivery).
- Actualizar la cantidad ordenada de delivery como cero (ya que se entegó el pedido)
- Actualizar el TIempo de llegada del delivery al sistema como esto con el fin de decir al sistema, que no hay un tiempo definido para un futuro delivery y solo ocurren eventos de llegada de clientes al sistema (a menos que se quede con menos de s en inventario y sea necesasrio realizar una orden de delivery).

Una vez realizado el diagrama de acciones y restricciones por realizar de forma secuencial y causal, hay que traducir lo anterior en Python.

Para comenzar se debe descargar e instalar la librería *numpy*, ya que este contiene la generación de instacias de distribuciones uniformes (como se discutió arriba esta es la base para generar instancias aleatorias de distribuciones más complejas). En caso de ser necesario se deben crear funciones que ocupen o bien el método de la inversa o el de aceptación y rechazo (y en algunos casos ambos). Para este ejemplo de simulación de inventario no hay que crear ninguna función auxiliar.

El primer paso a seguir es importar el módulo:

```
import numpy as np
```

Aquí **as np** sirve para abreviar el módulo *numpy* cada vez que este sea llamado.

Para este caso solamente se ocupará una clase que será la del sistema a simular. En su método __init__ se definen todos los atributos asociados, ya sean temporales, endógenos o exógenos al sistema por simular quedando:

```python
class Simulacion:
    def __init__(self, inventario_minimo, orden_de_destino):
        """
        Aqui se crean los atributos del sistema a simular.
        """
        self.inventario = 0
        self.cantidad_ordenada = 0

        self.tiempo_actual = 0
        self.tiempo_cliente = self.generar_llegadas()
        self.tiempo_delivery = float("inf")

        self.ingresos = 0
        self.costo_ordenes = 0
        self.costo_inventario = 0

        self.cota_de_inventario = inventario_minimo
        self.orden_de_destino = orden_de_destino
```

Luego se debe definir uno de los métodos más importante de la simulación al que se llamará *avanzar el tiempo*. La importancia de este método radica en que es el encargado de llamar a los otros métodos que actualizan los atributos de la simulación según sea el próximo evento por ocurrir. Esto se define en Python de la siguiente manera:

```python
def avanzar_el_tiempo(self):
    """
    Metodo que actualiza el tiempo de los proximos eventos por ocurrir
    que para este caso son solo dos opciones:
    1º --> LLegada de un nuevo cliente
    2º --> Entrega de un delivery
    """

    # Mensaje amigable
    print("[INVENTARIO] Tiempo actual {} e inventario {}\n".format(self.tiempo_actual, self.inventario))

    # Se calcula el tiempo del proximo evento por ocurrir
    tiempo_evento = min(self.tiempo_cliente, self.tiempo_delivery)

    # Se calcula el costo de inventario asociado a la cantidad de
    # dias que han pasado y la cantida de elementos
    # en el inventario durante estos dias.
    self.costo_inventario += 2*self.inventario*(tiempo_evento - self.tiempo_actual)


    # Se analiza que evento realizar primero segun los tiempo del proximo
    # evento por ocurrir y se ejecuta dicho metodo.
    if self.tiempo_delivery <= self.tiempo_cliente:
        self.manejo_evento_delivery()

    else:
        self.manejo_evento_de_cliente()

    # Actualiza el tiempo
    self.tiempo_actual = tiempo_evento
```

Luego de haber definido la parte anterior , se deben definir métodos según las cantidad de eventos por ocurrir, como en este caso solo se tienen los eventos: *llegada de cliente al sistema* y *orden de delivery* se deben definir y armar dos métodos que actualicen las condiciones del sistema según lo descrito en el pauteo anterior. De esta forma los métodos por escribir son:

```python
def manejo_evento_de_cliente(self):
    """Metodo que simula la demanda de productos por cliente"""

    # Genera una demanda por el cliente actual en el sistema.
    demanda = self.generar_demanda()

    # Mensaje amigable
    print("[CLIENTE] Ha llegada un cliente en {} y demanda {} de productos\n".format(self.tiempo_actual, demanda))

    # Analiza si el inventario ACTUAL es mayor que la demanda
    # o no para ver que acciones realizar segun sea el caso.
    if self.inventario > demanda:
        self.ingresos += 100*demanda
        self.inventario -= demanda
    else:
        self.ingresos += 100*demanda
        self.inventario = 0

    # Ahora analiza si una vez realiazada la compra por el usuario
    # se debe realizar un pedido segun la restriccion de cantidad
    # minima en inventario: INVENTARIO < s.

    if (self.inventario < self.cota_de_inventario) and self.cantidad_ordenada == 0:
        self.cantidad_ordenada = self.cota_de_inventario - self.inventario
        self.costo_ordenes += 5*self.cantidad_ordenada
        self.tiempo_delivery = self.tiempo_actual + 2
        self.tiempo_cliente = self.tiempo_actual + self.generar_llegadas()

    else:
        self.tiempo_cliente = self.tiempo_actual + self.generar_llegadas()
```

```python
def manejo_evento_delivery(self):
    """
    Metodo que simula el delivery de productos hacia la empresa
    para abastecer el inventario.
    """

    # Se actualiza el inventario actual con la cantidad ordenada 2 dias atras
    self.inventario += self.cantidad_ordenada

    # Mensaje amigable
    print("[ORDEN] Ha llegado a inventario {} en el tiempo {}\n".format(self.cantidad_ordenada, self.tiempo_actual))

    # Se indica que la cantidad ordenada fue recibida.
    self.cantidad_ordenada = 0

    # Tiempo de ocurrencia del proximo delivery se define como "infinito"
    # esto es con el fin de indicar que solamente ocurriran llegadas
    # de clientes en el corto plazo a menos que se quede sin inventario
    # y se pida una nueva orden de delivery fijando un nuevo
    # tiempo de delivery.
    self.tiempo_delivery = float("inf")
```

Como se mencionó, es muy importante tener funciones que generen instancias de variables aleatorias, para este contexto aprovechamos las funciones *np.random* y *np.poisson* que tiene implementadas el módulo *numpy*. Como no es necesario (en este caso) ocupar uno de los dos métodos de generación de instancias de variables aleatorias, simplemente se debe definir los métodos de la siguiente manera:

```python
def generar_llegadas(self):
    """
    Metodo que genera tiempos aleatorios de llegadas al sistema
    """
    return np.random.exponential(5)
```

```python
def generar_demanda(self):
    """
    Metodo que genera una demanda aleatoria de productos
    """
    return np.random.randint(1, 5)
```

Es importante tener estadísticas de lo que se desee simular, para este conexto se define un método que calcule e imprima en pantalla las estadísticas finales de los beneficios, ingresos y costos asociados a la tienda y su inventario. Esto queda como:

```python
def imprimir_estadisticas(self):
    """
    Metodo en el que se imprimen las estadisiticas al final del sistema
    """
    print("\n------ ESTADISTICAS ------")
    print("Los ingresos totales son: {}".format(self.ingresos))
    print("Los costos de mantencion de inventario son: {}".format(self.costo_inventario))
    print("Los costos de ordenes de inventario son: {}".format(self.costo_ordenes))
    print("\n")
    print("Los beneficios totales para la empresa son: {}".format(self.ingresos - self.costo_inventario - self.costo_ordenes))
```

Finalmente, hay que definir la instancia de la clase Simulación dentro del programa principal junto con la cantidad temporal que se desee simular y los parámetros que uno desee definir, todo esto de la siguiente forma:

```python
if __name__ == '__main__':
    s = Simulacion(10, 30)  # Cuando se tiene menos de 10 en inventario se piden 30

    # Se setea la cantidad de tiempo que se desea operar
    limite_tiempo = int(input("Ingrese cuanto tiempo desea simular: "))

    while s.tiempo_actual < limite_tiempo:
        # Se actualiza el tiempo en cada iteracion
        s.avanzar_el_tiempo()

    # Al terminar se imprimen las estadisticas
    s.imprimir_estadisticas()
```