MÁSTER EN INTELIGENCIA ARTIFICIAL

MÉTODOS DE SIMULACIÓN

Práctica Grupo 3

Autores
LUIS COUTO SELLER
IRENE MARBÁN ÁLVAREZ
AÍDA MUÑOZ MONJAS

December 6, 2022

Contenidos

1 Generación de números y variables aleatorias

1.1 Enunciado

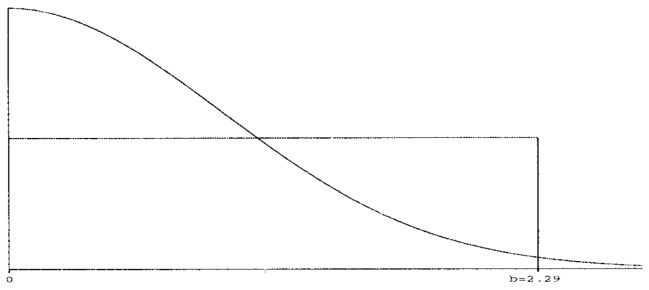
Describir el método *Monty Phyton* para la distribución normal y compararlo con otros métodos para la generación de valores de la normal.

1.2 El método Monty Python

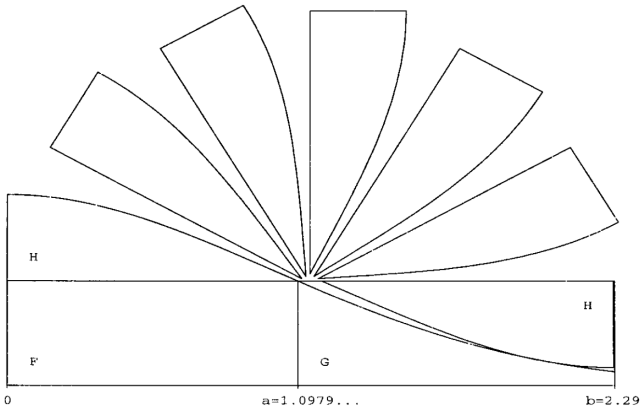
El objetivo del método Monty Python es lograr generar números aleatorios cuya frecuencia se aproxime a aquella de una distribución normal. Para ello, emplea la similitud entre la expresión de la función de densidad de la distribución normal y una sigmoide decreciente, así como se aprovecha del bajo coste computacional de la generación de números aleatorios en un rectángulo.

$$N(0,1) \sim f(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{-x^2/2}$$

Tomando $x > 0$, se traza un rectángulo de base b y altura $1/b$ sobre la sigmoide. Con un giro de π radianes y un desplazamiento sobre la región exterior al rectángulo se pretende representar la mayor superficie posible de la sigmoide en el interior del rectángulo de área 1. En las imágenes siguientes, podemos observar el procedimiento descrito tomando un $b = 2.29$.



(a) Sigmoide con el rectángulo de base b. [1]



(b) Giro y desplazamiento gráficamente. [1]

Una vez descrito el área bajo la función de densidad en el interior del rectángulo, todo número aleatorio generado en ese rectángulo pertenecerá a las regiones F, G, H o a la región comprendida entre G y H correspondiente a la cola de la sigmoide. La probabilidad de obtener un número en la cola es de un 0.022 [1].

Siendo (x, y) el valor aleatorio generado, (x', y') será su valor correspondiente en la sigmoide.

$$(x', y') = \begin{cases} (x, y) & \text{si} \quad x \in F \text{ ó } G \\ (b - x, 2/b - y) & \text{si} \quad x \in H \end{cases}$$

Si el valor aleatorio generado no pertenece a F, G ó H, pertenecerá a la cola de la distribución, por lo que basta con devolver una variante de la cola normal mediante el método de Marsaglia o el método de la cola general de Marsaglia y Tsang.

La elección del valor de b no es crítica, pero elegir un valor de b demasiado grande implica que habrá solapamiento al realizar el giro y desplazamiento descrito, mientras que si el valor de b es demasiado pequeño, será necesario realizar frecuentemente el método para hallar los valores de la cola. En caso de utilizar el método anterior para "ajustar" la sigmoide al rectángulo, la elección de $b = 2.29$ es prácticamente la máxima posible [1].

En lugar de una rotación y un desplazamiento de la región exterior al rectángulo, "estirar" la región H permite un ajuste mejor, manteniendo constante el área de la sigmoide, y por tanto siendo este un método más complejo pero más eficiente para realizar dicha aproximación.

Definiendo el factor de estiramiento s y la función de densidad de la región H, $f_H(x)$,

$$s = \frac{a}{b-a} \qquad f_H(x) = f(x) - \frac{1}{b} \text{ con } 0 < x < a$$

se tiene que la región H girada y estirada tiene una función de densidad $f_{H'}(x)$.

$$f_{H'}(x) = \frac{1}{b} - s\left[f(s(b-x)) - \frac{1}{b}\right]$$

La siguiente figura representa gráficamente las transformaciones matemáticas descritas compuestas con la rotación aplicada a la sección H.

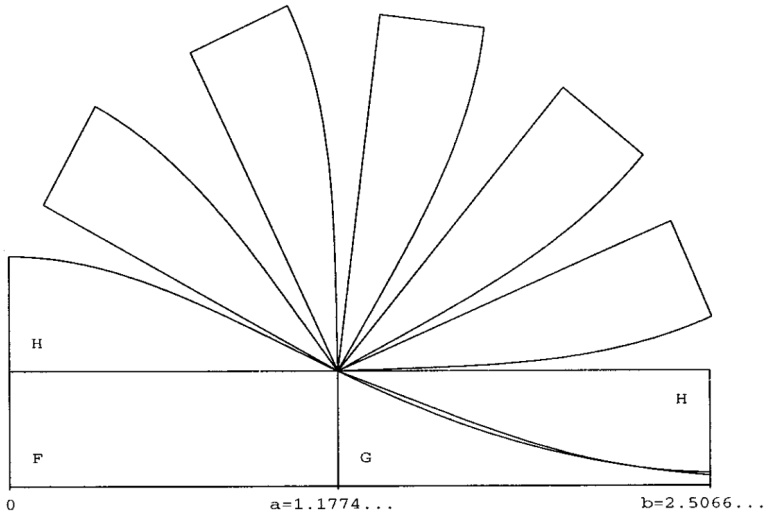


Figure 2: Descripción gráfica del estiramiento descrito. [1]

Para este segundo caso, la elección habitual para el valor de b es $b = \sqrt{2\pi}$. Esta elección no es crítica, cualquier valor entre 2.506 y 2.5074 es válido [1], pero el par $b = \sqrt{2\pi}$, $a = \sqrt{\ln 4}$ son opciones fácilmente identificables con una precisión ilimitada.

1.3 Comparación de métodos

A continuación, se compararán diferentes métodos para la generación de valores de la normal con el método Monty Python. Para ello, nos basaremos en la publicación citada [2]. Teniendo en cuenta que tanto el método generador de las secciones F, G y H, como el método generador de las colas de la sigmoide son relevantes, el artículo divide la evaluación en cuatro secciones.

- Tests de bondad de ajuste. Para evaluar los algoritmos utilizados, se utiliza el test χ^2. El número de intervalos en los que se divide la distribución para realizar el test es k, que se define mediante la siguiente expresión, siendo n el número de muestras.
$$k = \lceil n^{3/5} \rceil$$
- Test para valores altos de sigma. Este test mide principalmente el comportamiento de los generadores en las colas de la distribución. El principal reto es la baja probabilidad de obtener muestras que se encuentren en las colas. Se utiliza por tanto un algoritmo para "forzar" la generación de muestras que sean grandes múltiplos de sigma.
- Test de aleatoriedad. Para este test, se utiliza la batería Crush, que forma parte de los *TestU01*.
- Test de correlación interbloque. Este test mide como afecta la distribución de cada una de las muestras a la muestra siguiente, considerándose un buen resultado si la distribución de cada una de las muestras es independiente de las anteriores.

También se evalúa el número de operaciones realizadas y la velocidad de ejecución.

1.3.1 Resultados en velocidad.

La velocidad se mide relativa a la velocidad del método de Rechazo Polar. El método Monty Python es el cuarto más rápido en comparación con los 16 otros métodos de generación de variables aleatorias estudiados [2], siendo 1.61 veces más rápido que el método de Rechazo Polar. Cabe destacar que el número de operaciones realizadas es bajo en comparación con el método Wallace, que está por encima del método Monty Python en rapidez. Este último tiene menos cantidad de constantes que el método Ziggurat, que está también por encima del Monty Python en cuestión de velocidad.

Nombre del método	Velocidad
Wallace (qual=1)	6.41
Ziggurat	4.29
Wallace (qual=4)	2.48
Monty Python	**1.61**
PPND7	1.16
Mixture-of-triangles	1.14
Polar	1.00

1.3.2 Resultados en el test de bondad de ajuste χ^2.

En el test de bondad de ajuste χ^2, el método Monty Python no pasa el test al utilizar muestras de valores mayores a 2^{34}. De nuevo, los métodos Wallace y Ziggurat obtienen mejores resultados, ya que no fallan al utilizar muestras de valor igual o mayor a 2^{36}. El algoritmo PPND7 falla en la misma prueba que el método Monty Python, que a su vez es 3 veces más lento.

1.3.3 Resultados en el test de valores altos de sigma.

Los resultados de este test muestran el múltiplo más alto de sigma donde el test se realiza con éxito. El método Monty Python obtiene un valor de 8.27, siendo en este caso mejor que los métodos Ziggurat y Wallace. En comparación con el resto de generadores, el método Monty Python obtiene muy buenos resultados, situándose el cuarto mejor de todos los generadores comparados.

1.3.4 Resultados del test de aleatoriedad

El generador Monty Python no tuvo fallos al realizarse el test Crush.

1.3.5 Resultados del test de correlación interbloque.

Este test solo se realizó en los generadores Ziggurat, Wallace y Monty Pythhon, dado que fueron los más rápidos y en este test se necesitan realizar pruebas con muestras muy grandes. Los generadores de Ziggurat y Monty Python pasaron todos los tests con éxito, mientras que el generador Wallace (tanto el de baja calidad como el de alta), falló con un número inferior a 8 iteraciones. Esto se debe a que el método Wallace es recursivo.

1.3.6 Conclusiones

Si bien es cierto que el método Wallace es el más rápido, presenta desventajas evidentes como la correlación. El método Ziggurat, que es más lento que el Wallace, no tiene problemas de correlación, pero si que falla en el test Crush con una colisión identificada en los tests de doble precisión. Además, utiliza un total de 388 constantes, lo que puede ser problemático en algunos entornos. El método Monty Python es el tercero más rápido después del Wallace y el Ziggurat, y presenta ventajas evidentes, como los buenos resultados en los tests de aleatoriedad y correlación, así como el bajo número de operaciones y constantes necesarias. La mayor limitación del método Monty Python está en el uso de muestras cuya n sea mayor que 2^{36}.

2 Simulación de sucesos discretos y optimización

2.1 Enunciado

Consideremos un almacén de dos productos cuyos precios de venta al público son de 2.5 y 3.5 euros la unidad, respectivamente. La llegada de clientes al almacén se distribuye según un proceso de Poisson de parámetro $\lambda = 1.5$ clientes por hora y la cantidad de productos demandados por cada uno de ellos tiene la siguiente distribución:

Demanda	1 unidad	2 unidades	3 unidades	4 unidades
Producto 1	0.3	0.4	0.2	0.1
Producto 2	0.2	0.2	0.4	0.2

Para satisfacer la demanda de sus clientes el dueño del almacén mantiene un stock de productos. La política de pedidos al distribuidor es periódica, es decir todos los viernes a primera hora realiza un pedido, tomando como referencia el nivel de inventario de los productos en ese. Para satisfacer la demanda de sus clientes el dueño del almacén mantiene un stock de productos. La política de pedidos al distribuidor es periódica, es decir todos los viernes a primera hora realiza un pedido en el que tomando como referencia el nivel de inventario de los productos en ese momento, se solicitan las unidades necesarias para que el nivel del inventario de cada producto llegue a 1000 y 1500 unidades, respectivamente en cada producto.

Asociado a cada pedido que realizamos al proveedor existe un coste fijo (coste de preparación) de 100 euros, independientemente de las unidades demandadas. Adicionalmente, el coste por unidad incluida en el pedido depende de la cantidad solicitada habiendo descuentos por cantidad. Si el número de unidades demandadas del primer producto es menor o igual a 600 el precio es de 1 euro la unidad, mientras que si se piden más de 600 el precio desciende a 75 céntimos. Para el segundo producto, si el número de unidades demandadas es menor que 800 el precio es de 1.5 euros la unidad, mientras que si se piden más de 800 el precio desciende a 1.25 euros.

El tiempo que tarda en ser servido el pedido por los proveedores (tiempo líder), sigue una distribución normal de media 48 horas y desviación típica 3.5, pagándose en ese momento.

Se ha llegado a un acuerdo con el proveedor de forma que, tomando como referencia las 48 horas que tarda en media un pedido en ser servido, si el pedido llega con 3 horas de retraso en la entrega del mismo se realiza un descuento del 0.03% del valor del pedido, encareciéndose en la misma cantidad en el caso de que el pedido llegue con al menos 3 horas de adelanto.

El dueño del almacén debe pagar 0.0002 euros por unidad del producto y unidad de tiempo, asociado al almacenamiento físico de los productos (alquiler del local, refrigeración...). En el caso de que al llegar un cliente éste solicite una cantidad mayor que la que hay en inventario, se le sirve lo que queda, perdiendo la venta restante.

a) Simular el comportamiento de almacén durante un periodo de tiempo de 5 meses para estimar el beneficio esperado, la proporción de clientes cuya demanda se satisface completamente y el porcentaje de tiempo que el nivel del inventario permanece a cero. Para ello, supondremos que el nivel del inventario inicial es de 70 unidades de ambos productos.

b) Representar gráficamente la evolución del nivel del inventario durante los 5 meses y durante los 5 primeros días.

c) Mediante el uso de la metaheurística recocido simulado identificar cuál será la política de pedidos óptima, es decir, identificar cada cuánto tiempo se deberían realizar los pedidos y el valor de referencia del inventario para identificar el número de unidades a solicitar en la política de pedidos periódica.

Nota. El almacén permanece abierto las 24 horas al día de lunes a domingo. El proveedor y la empresa transportista (que sirve los pedidos) también trabajan las 24 horas, no cerrando en ningún momento.

2.2 Modelado del problema

El problema descrito se puede representar según el diagrama a continuación.

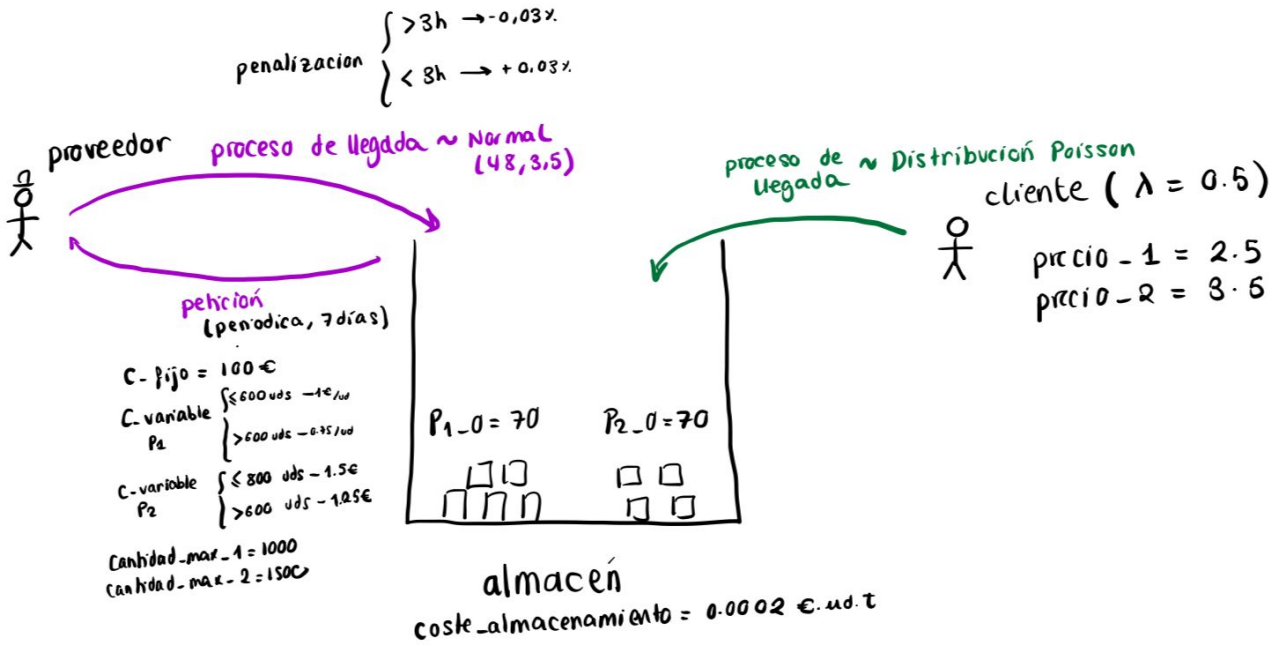


Figure 3: Representación gráfica del modelado del problema

El almacén guarda productos de dos tipos, siendo x_1 y x_2 la cantidad de cada uno de los productos almacenados en un instante. Los productos se guardan por separado, teniendo una cantidad máxima de cada producto en stock.

Una de las principales características de este problema es su política de pedidos. Cada 7 días el propietario realiza un pedido al proveedor, independientemente del estado del almacén durante los días anteriores.

La llegada de clientes al sistema sigue una distribución de Poission con $\lambda = 0.5$, mientras que la llegada de pedidos sigue una normal $N(48, 3.5)$.

El coste total del sistema se calcula como las ganancias por las ventas de productos (R) menos los costes de almacenamiento (H) y costes de compra de productos al proveedor (C).

2.3 Simulación del problema. Estimación de beneficios y satisfacción de clientes.

Según lo indicado en el apartado a), el objetivo de esta sección es obtener los siguientes valores.

- Estimación el beneficio esperado.
- Proporción de clientes cuya demanda se satisface completamente.
- Porcentaje de tiempo que el nivel del inventario permanece a cero.

Se simulará el comportamiento del sistema durante 5 meses, teniendo inicialmente 70 unidades de ambos productos en el almacén.

2.3.1 Implementación

La implementación de esta simulación se realizó utilizando el lenguaje python, definiendo de manera separada las rutinas de llegada de un cliente, llegada de un pedido y compra de un pedido.

Las variables utilizadas se definen como globales, permitiendo que la entrada de cada uno de los métodos sea únicamente el tiempo de simulación (*ts*). No existen problemas de acceso concurrente a las variables, ya que todo el proceso de simulación sucede de manera lineal.

A continuación se destacarán las rutinas utilizadas para cada tipo de evento:

- **Llegada de un cliente**.

  Esta rutina se ejecuta con la llegada de un nuevo cliente al sistema. Se actualizan los costes de almacenamiento y el estado del almacén, así como los beneficios en caso de poder satisfacer las necesidades del cliente. Cabe resaltar que un cliente no se considera satisfecho si recibe el pedido deseado de uno de los productos, pero no del otro.

  En este método se genera también la llegada del siguiente cliente.

```
def rutina_llegada_cliente(ts):
  global H, h, t_real, x_1, x_2, Nc, Nnc, var_aux, R, Y, y_1, y_2
  global r_1, r_2, var_aux, T_simulacion

  #Aumenta el coste de almacenamiento
  H += (ts-t_real)*h*(x_1+x_2)
  t_real = ts


  #Generamos demanda del cliente
  demanda_1 = np.random.choice(demanda, 1, p=probab_1)[0]
  demanda_2 = np.random.choice(demanda, 1, p=probab_2)[0]

  #Si hay suficiente almacenado, esta satisfecho
  if demanda_1<=x_1 and demanda_2<=x_2 :
    R += demanda_1*r_1 + demanda_2*r_2 #sube el beneficio
    x_1 -= demanda_1    #baja el inventario
    x_2 -= demanda_2
    Nc += 1 #cliente satisfecho
```

```
  #Si no hay suficiente almacenado de algun producto, no esta satisfecho
  else:
    if(demanda_1<=x_1):
      R += demanda_1*r_1
      x_1 -= demanda_1
    elif(demanda_2<=x_2):
      R += demanda_2*r_2
      x_2 -= demanda_2
    Nnc += 1 #cliente no satisfecho

  # Si se ha vaciado del todo (y antes no estaba vacio) guardamos cuando
  if x_2 == 0 and x_1 == 0 and var_aux == 0 :
    var_aux = t_real


  #Generamos el tiempo que tarda en llegar el siguiente cliente
  Y = stats.poisson.rvs(lambda_poisson, size=1)[0]

  # si el cliente llega antes de acabar la simulacion, se simula
  if Y+t_real < T_simulacion:
    lista['tc'] = t_real+Y
```

- **Llegada de un pedido**.

  Esta rutina se ejecuta con la llegada de un pedido que se compró en el pasado al sistema. Se actualizan los costes de almacenamiento, así como el nivel de inventario posterior a la recogida del pedido. Según el tiempo de llegada del pedido, se calcula el precio de la compra.

```
def rutina_llegada_pedido(ts):
  global H, K, h, t_real, C, t0, var_aux, x_1, x_2, y_1, y_2
  global p1_1, p1_2, p2_1, p2_1, penal, var_aux

  #Aumenta el coste de almacenamiento
  H += (ts-t_real)*h*(x_1+x_2)
  t_real = ts

  #Aumenta el nivel de inventario
  x_1 += y_1
  x_2 += y_2

  #Si son muchas unidades, descuento en el precio
  Ci_1 = K + y_1 * p1_1 if y_1<=n_descuent_1 else K + y_1 * p1_2
  Ci_2 = K + y_2 * p2_1 if y_2<=n_descuent_2 else K + y_2 * p2_2

  #Si llega pronto es mas caro, si llega tarde, es mas barato
  if L < Lref-lim_penal:
    C += (Ci_1 + Ci_2) * (1 + penal)
  elif L > Lref+lim_penal:
    C += (Ci_1 + Ci_2) * (1 - penal)
  else:
    C += Ci_1 + Ci_2

  #Ya no quedan productos por llegar
  y_1 = 0
```

```
  y_2 = 0

  # Si estaba vacio el inventario (se vacio en el instante var_aux),
  # aumenta el tiempo que ha estado vacio.
  if var_aux > 0:
    t0 += t_real - var_aux
    var_aux = 0
```

- **Compra de un pedido**.

  Esta rutina se ejecuta en la compra de un nuevo pedido para el almacén. De acuerdo con las indicaciones del enunciado, cada siete días se realiza una compra de la cantidad restante para completar el almacén de cada uno de los productos.

```
def rutina_compra_pedido(ts):

  global H, x_1, x_2, t_real, y_1, y_2, h, t_real
  global P1, P2, lista, T_simulacion

  # Aumenta el coste de almacenamiento
  H += (ts-t_real)*h*(x_1+x_2)
  t_real = ts

  #Cantidad a pedir es lo que falta para llenar el almacen
  y_1 = P1 - x_1
  y_2 = P2 - x_2

  #Generamos cuanto va a tardar en llegar el pedido
  L = np.random.normal(mu, sigma, 1)[0]

  # actualizamos el tiempo de llegada del pedido y el tiempo de siguiente
                                          compra
  if L+t_real < T_simulacion:
    lista['tp'] = t_real + L

  if t_real+Tp < T_simulacion:
    lista['tpc'] = t_real + Tp
```

2.3.2 Resultados y representación gráfica

En esta sección se muestran los resultados de la ejecución de la simulación descrita. Tal y como se ha descrito en la sección de implementación del código, se ha llevado un seguimiento de la simulación gracias a las sentencias *print()* que se han colocado en las distintas funciones del script. Ejemplos de las salidas para cada uno de los eventos posibles son:

- **Llegada de un cliente.** Podemos hacer un seguimiento del momento en el que se produce la llegada del cliente, cuántas unidades demanda de cada producto y el estado del almacén antes y después del servicio.

```
------------------------------------------------------------
Llega nuevo cliente
  - demanda producto 1: 1
  - demanda producto 2: 2
Estado del almacen: x_1:495, x_2:899
> Cliente satisfecho!
Estado del almacen: x_1:494, x_2:897
tiempo actual: 3460
------------------------------------------------------------
```

- **Realización de un pedido.** Se muestra el instante de tiempo en el que se realiza el pedido, el estado del almacén en ese momento y el tiempo que tardará en llegar dicho pedido.

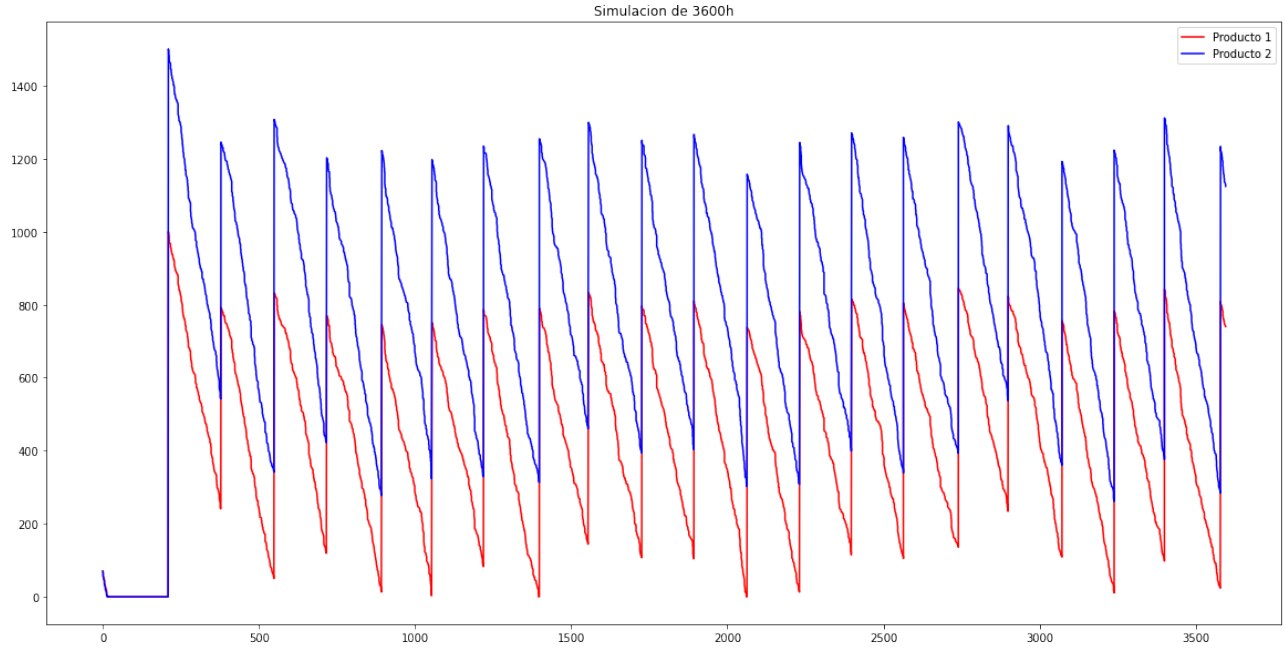
```
------------------------------------------------------------
Realizamos pedido al proveedor
Estado del almacen: x_1:215, x_2:551
> El pedido tardara este tiempo: 51.9811619786948
> Se ha pedido: x_1:785 x_2:949
tiempo actual: 3528
------------------------------------------------------------
```

- **Llegada del pedido.** Se indica el instante en el que llega el pedido, el nivel de inventario antes y después de la llegada del pedido, y el tiempo que el almacén estuvo vacío en caso de haber sido así.

```
------------------------------------------------------------
El pedido ha llegado
Estado del almacen (antes): x_1:24, x_2:284
Estado del almacen (desp.): x_1:809, x_2:1233

tiempo actual: 3579.9811619786947
------------------------------------------------------------
```

En la siguiente gráfica podemos ver la evolución de los niveles de inventario de los productos 1 y 2 a lo largo del tiempo de simulación (5 meses), equivalente a 3600 horas. Podemos observar cómo se repite para cada tipo de producto la estructura de picos a lo largo del tiempo con unas alturas variables pero muy cercanas.



- Durante el tiempo de simulación el total de ingresos recibidos gracias a las compras de los clientes es de $99\ 460$ euros.
- En total se han realizado 21 pedidos que suman un gasto de $40\ 315.34$ euros.
- El coste de almacenamiento de los productos ha sido de 864.34 euros.

Para calcular el beneficio obtenido basta con restar a los ingresos los gastos de pedido y los gastos de almacenamiento:

$$beneficio = 99\ 460 - 40\ 315.34 - 864.34 = 58\ 280.32$$

Partiendo de un nivel de inventario de 70 unidades de cada tipo de producto, y con la política de pedidos descrita en el enunciado, la proporción de tiempo que el almacén está completamente vacío es de un 5.33% del tiempo, lo que en horas equivale a 191.3 horas.

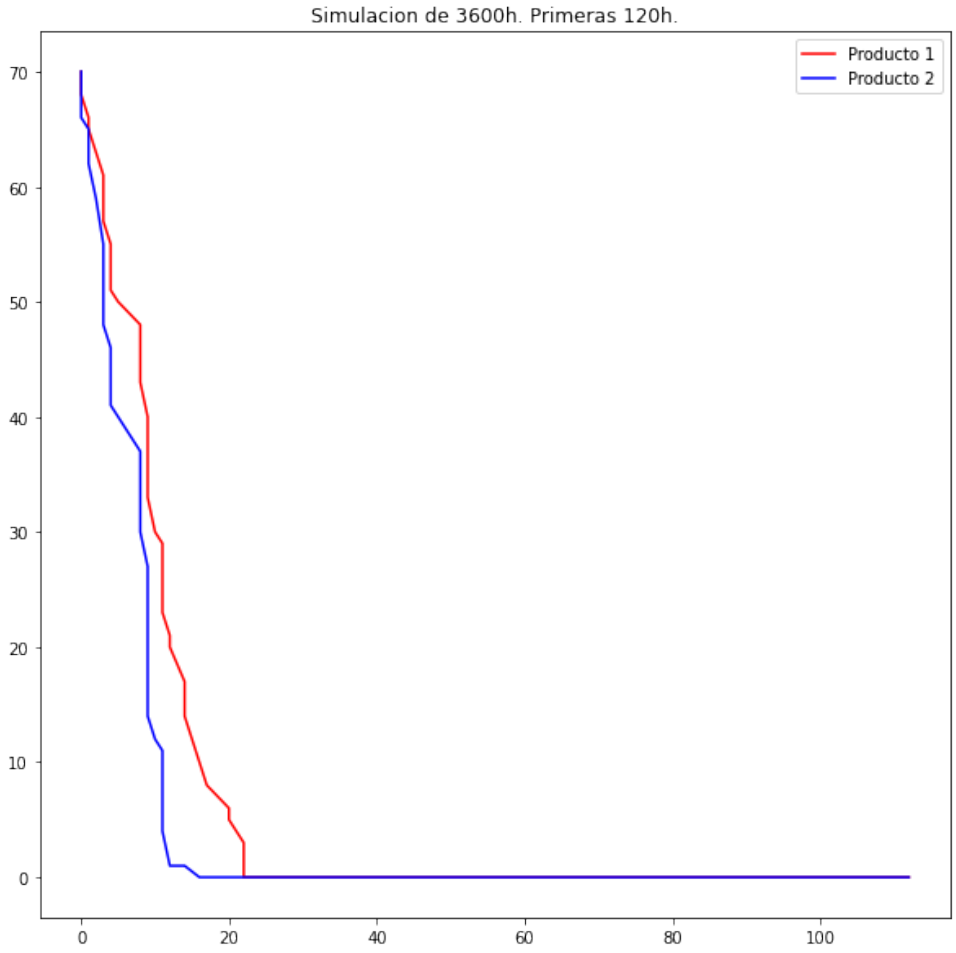
En $3\ 600$ horas han llegado un total de $7\ 298$ clientes de los cuales:

- $6\ 941$ clientes su demanda ha sido completamente satisfecha.
- 357 clientes su demanda no ha sido satisfecha.

Eso supone que la demanda de un 95.11% de los clientes que llegaron a la tienda en este periodo

de 5 meses fue totalmente satisfecha.

Analizando los resultados, si observamos más de cerca en la gráfica el periodo de tiempo que comprende los 5 primeros días de la simulación (120 horas), podemos observar que debido al nivel inicial de inventario y a que la política de pedidos establece que los pedidos se hacen de forma periódica cada 168 horas (cada 7 días), hay una gran franja en la que el inventario de ambos productos está a 0 sin poder satisfacerse la demanda de los clientes que llegan en ese periodo.



Como se observa, partiendo únicamente con 70 unidades de cada producto, alrededor de las 16 horas se agotaron las unidades del producto 2 y a las 22 horas de simulación ya se habían acabado las existencias de ambos productos quedando completamente vacío el inventario, teniendo que esperar a que se realizara el pedido y a que este llegara.

2.4 Simulación del problema con metaheurísticas. Identificación de política de pedidos óptima.

2.4.1 Introducción al Recocido Simulado

Debido a la complejidad de los problemas de optimización que se busca resolver en el mundo, la búsqueda local y los métodos clásicos de optimización no son suficientes para acoplarse al gran número de factores de decisión y restricciones. Es por ello que se desarrollan las metaheurísticas, dando un paso más allá de las heurísticas clásicas que se centraban únicamente en un tipo de problema, y podían quedarse atrapadas en un mínimo local.

Las metaheurísticas utilizan conocimiento de distintas áreas de la ciencia, como puede ser la inteligencia artificial, la estadística, la genética... para proponer algoritmos que devuelven una solución eficiente y satisfactoria de un problema de optimización.

En esta práctica, se nos ha propuesto utilizar la metaheurística del tipo de búsqueda global, conocida como *recocido simulado*. Esta metaheurística se basa en el fénomeno de enfriamiento de metales que se estudia en el ámbito de metalurgia y termodinámica.

Para aplicar esta idea a un problema de optimización, se plantean tres principales etapas:

- En la primera etapa, las soluciones elegidas en el entorno $E(x)$ de manera aleatoria no son necesariamente óptimas. Si la solución generada es mejor que la anterior, la exploramos directamente, pero si es peor, podemos explorarla según una probabilidad determinada por una variable aleatoria. En esta primera fase, la probabilidad de elegir soluciones peores es más alta, evitando así caer en mínimos locales y permitiendo explorar un entorno más amplio.
- En la segunda etapa, la probabilidad de que elijamos una solución peor ya no es tan alta, pero aun así hay posibilidad de volver atrás.
- En la última etapa, sólo aceptaremos movimientos que mejoren la solución actual.

En este algoritmo es muy importante la elección del parámetro *Temperatura*. Este parámetro, ira decreciendo cada L iteraciones, hasta llegar a una temperatura final u otro punto de parada que se haya especificado. La forma en la que la temperatura influye en la búsqueda del algoritmo es mediante la expresión que define la distribución de Boltzman.

$$p(i) = e^{-\frac{f(y) - f(x_i)}{T_i}}$$

Esta es la distribución que va a tomar la probabilidad de elegir una peor solución, de manera que si la temperatura es muy alta, la probabilidad de que elijamos una peor solución será mayor. La temperatura decrece según la expresión

$$T_{hL} = \alpha^h T_1$$

El valor alfa (α) determinará cuanto disminuye la temperatura.

En general, los hiperparámetros que vamos a tener que controlar para la ejecución del algoritmo son los siguientes:

- Temperatura inicial
- L, iteraciones que deben pasar para actualizar la temperatura
- α, velocidad de disminución de la temperatura
- Criterio de parada
- Radio del entorno para elegir una solución vecina

El problema que buscamos resolver en esta práctica con este algoritmo es encontrar el valor de periodicidad con el que se deben realizar los pedidos, así como la cantidad que se debe pedir de cada uno de los productos para maximizar el beneficio.

2.4.2 Implementación

El proceso de recocido simulado genera N iteraciones de la simulación, modificando algunos de los valores en cada iteración.

Para implementar en python el recocido simulado, hemos definido el método `recocido_simulado`, que toma como parámetro un estado inicial, con las variables de decisión inicializadas. A continuación, se inicializan las variables L, Temperatura Inicial, Temperatura final, α, Temperatura Final, el contador de iteraciones totales y el contador de pasos para cambiar la temperatura.

```
def recocido_simulado(initial_state):
  L = 0 #Num de iteraciones donde mantenemos la temperatura
  initial_temp = 10000
  final_temp =.1
  alpha = 0.95 #Ritmo de decrecimiento de la temperatura
  iteraciones = 1 #Contador de iteraciones
```

En la primera iteracion, damos a la variable `current_state`, que va a ir guardando el estado donde nos encontramos, el valor del estado inicial. Generamos los beneficios utilizando el método `get_benef`, que ejecuta la simulación desarrollada en el primer apartado, y los guardamos en la clave "benef" del current_state. Igualamos la variable `best_state` al current_state.

```
  # Primera iteracion
  current_temp = initial_temp
  current_state = initial_state
  current_state["benef"] = get_benef(current_state)
  best_state = current_state
```

Una vez finalizada esta primera iteración, entramos en un bucle que terminará cuando la temperatura alcance el valor mínimo. Este criterio de parada es uno de los más utilizados (referencia a tfm), si bien es cierto que existen otros.

```
  while current_temp>final_temp:
    # Comprobar si el vecino es mejor
    neighbor = get_neighbor(current_state)
    neighbor["benef"] = get_benef(neighbor)

    benef_diff = neighbor["benef"] - current_state["benef"]
    prob = math.exp(-abs(benef_diff) / current_temp)

    # Si es mejor, aceptamos vecino
    # Si es peor, aceptamos si p>u
    if benef_diff > 0 or random.uniform(0, 1) < prob:
      current_state = neighbor
```

Al entrar en el bucle, utilizamos el método `get_neighbors` para obtener una solucion del entorno del estado actual, y de nuevo utilizando el método `get_benef`, obtenemos el beneficio que genera el vecino, y igualamos `prob` al valor de la probabilidad siguiendo la distribución de Boltzmann para la temperatura actual.

Si la solución es mejor, la variable `current_state` pasa a ser el estado del vecino. Si la solución es peor, generamos una probabilidad aleatoria utilizando el generador `random.uniform`. Si su valor es mayor que la probabilidad inicializada previamente, nos quedamos con el `current_state` actual. Si es menor, actualizamos current_state al valor del vecino.

```
def get_benef(state):
      return simul_main(state["periodicidad"],
                        state["nivel_1"], state["nivel_2"])

def get_neighbor(state):
  """Devuelve vecinos en un radio 1"""
  new_periodicidad = state["periodicidad"] +  np.random.normal(0,2,1)[0]
  new_nivel_1 = state["nivel_1"] +  round(np.random.normal(0,2,1)[0])
  new_nivel_2 = state["nivel_2"] +  round(np.random.normal(0,2,1)[0])

  neighbor = {'periodicidad':new_periodicidad ,
          'nivel_1': new_nivel_1,
          'nivel_2': new_nivel_2 }

  return neighbor
```

Dentro del mismo bucle, una vez se ha hecho el cambio de estado, se pasan por dos condicionales. En primer lugar, si el contador de pasos ha llegado al valor L, entonces actualizamos la temperatura. En segundo lugar, si la solución almacenada en `best_state` es peor que la del estado actual, actualizamos `best_state`. Esta modificación la hemos añadido para evitar que el algoritmo haya pasado en las primeras iteraciones por el mínimo global, y al rectificar haya terminado en un mínimo local.

```
  # Bajamos temperatura
  if L>10:
      current_temp = current_temp * alpha
      L=0

  if best_state["benef"] < current_state["benef"]:
    best_state = current_state

  iteraciones+=1
  L+=1
```

Cuando se alcanza la temperatura mínima, el algoritmo se detiene y el método devuelve el `best_state`.

2.4.3 Resultados

Los mejores resultados se han obtenido al inicializar los hiperparámetros identificados previamente a los siguientes valores:

- La temperatura inicial es de $10\,000$ grados. Probando con valores menores, como $9\,000$, no se alcanzaba el mínimo global.
- La temperatura final es de 0.1 grados
- α es 0.95, siguiendo las recomendaciones (Hajek, 1988).

- El valor de L es de 10 pasos
- El radio del entorno es igual a 1.

Para estos hiperparámetros, hemos conseguido un beneficio de $59\,306.66$, siendo $4\,291.35$ unidades mayor que el del estado inicial.

Cabe destacar que, al inspeccionar los *logs* que hemos impreso, el mínimo global se alcanza 150 iteraciones antes del final. Hemos probado a optimizar este resultado alterando el valor de L y la temperatura final, pero ninguna de las dos modificaciones ha dado resultados satisfactorios.

También hemos probado a cambiar la distribución que usamos para obtener vecinos del entorno. Para elegir los vecinos, hemos optado por una normal de radio 1, ya que al probar con distribuciones uniformes y mayor radio, el algoritmo se quedaba atrapado en mínimos locales.

Bibliografía

References

[1] G. Marsaglia and W. W. Tsang, "The monty python method for generating random variables," vol. 24, no. 3, p. 341–350, sep 1998. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/292395.292453

[2] D. Thomas, W. Luk, P. Leong, and J. Villasenor, "Gaussian random number generators," *ACM Comput. Surv.*, vol. 39, 11 2007.