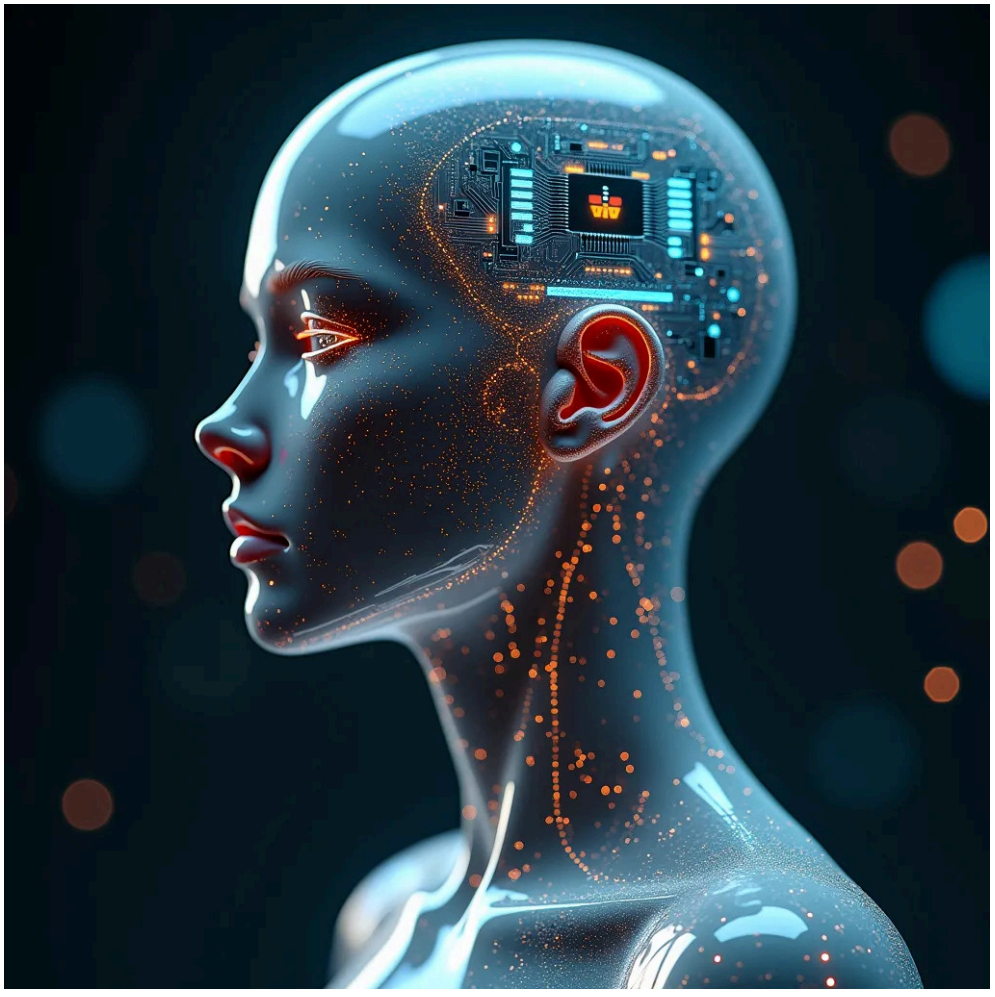
UD02_03 Práctica02: Lógica Difusa



1. Lógica difusa. Funciones de pertenencia

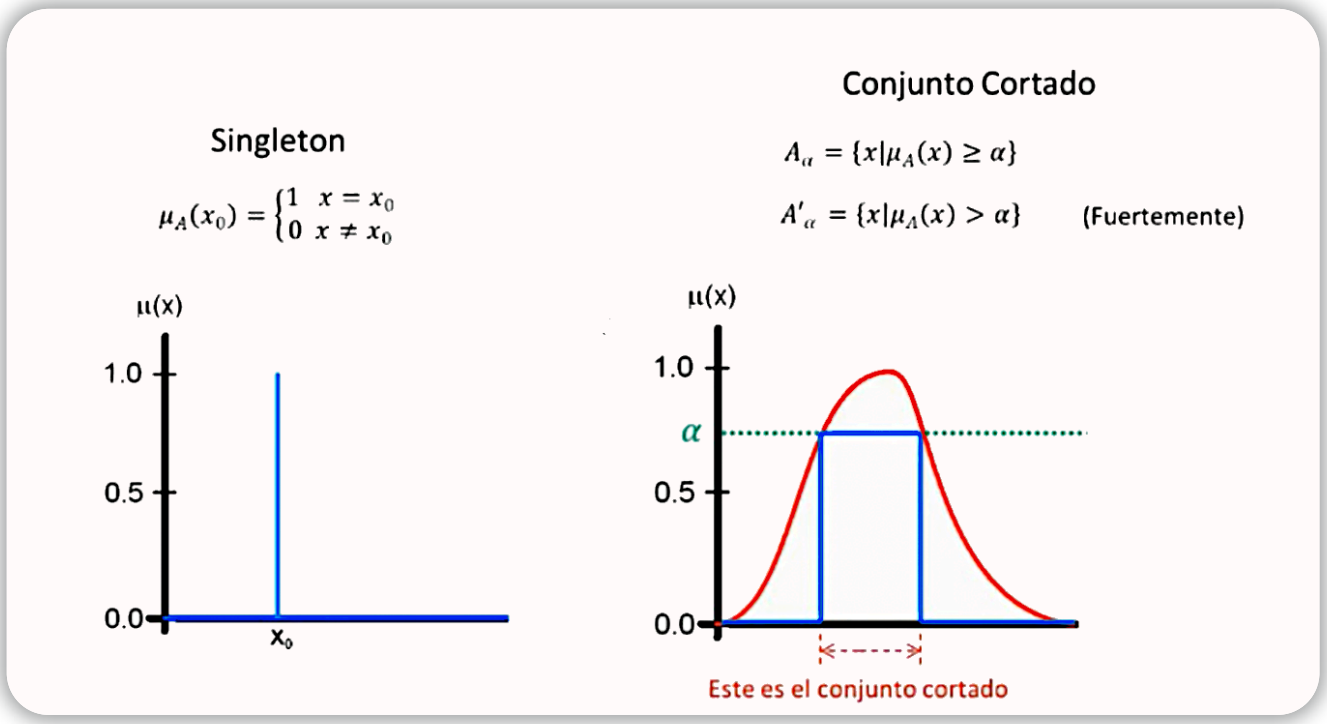
En esta práctica se presentan diferentes funciones difusas.

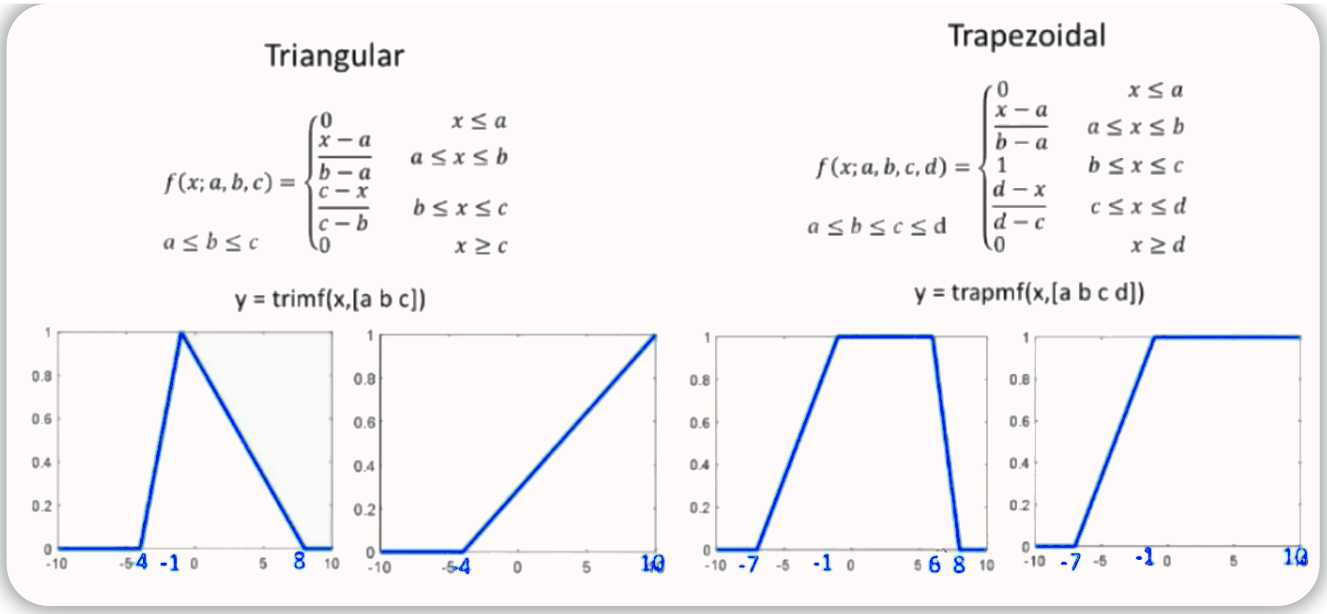
1.1. Funciones difusas de pertenencia básicas

En la primera parte de la práctica vamos a completar un cuaderno de júpiter en el que vamos a realizar en python las siguientes funciones de pertenencia:

- Función **singleton**(x, x0): función de pertenencia singleton.
  - x0: valor de referencia (int, float)
- Función **trimf**(x, param): función de pertenencia triangular.
  - param = [a, b, c]: debe cumplirse a <= b <= c
- Función **trapmf**(x, [a, b, c, d]): función de pertenencia trapezoidal.
  - [a, b, c, d]: debe cumplirse a <= b <= c <= d

Funciones de pertencias básicas



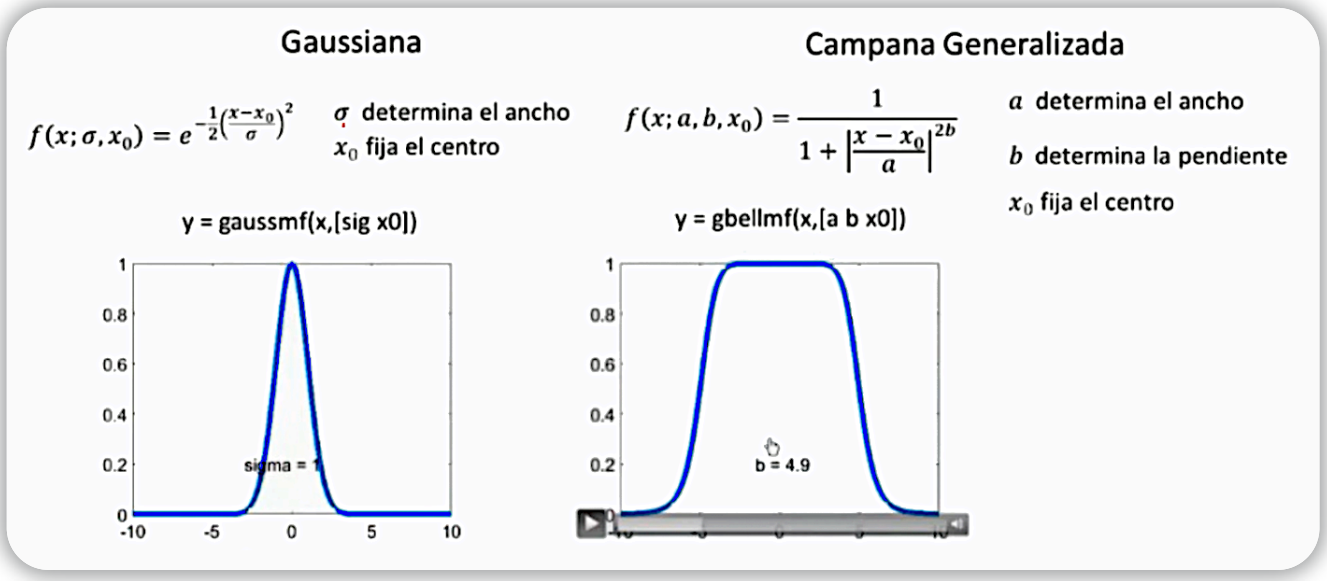


Tarea: **Debes terminar las funciones si alguna tiene pendiente parte de su implementación y verificar el funcionamiento de las mismas**, no solo con el ejemplo suministrado, y **explicar brevemente el significado de sus parámetros.**

1.2. Otras funciones difusas de pertenencia

En la segunda parte de la práctica vamos a ampliar nuestras funciones de pertenencia con:

- Función **gaussmf**(x, param): función de pertenencia gaussiana.
  - sig: desviación estándar (> 0); x0: media de la función
- Función **gbellmf**(x, param): función de pertenencia campana generalizada.
  - param = [a, b, X0]: debe cumplirse a, b > 0, con a= ancho, b=pendiente y x0=centro
- Función **sigmf**(): función de pertenencia sigmoidal.
  - param = [a, x0]: con a pendiente y x0 punto de cruce



Sigmoidal

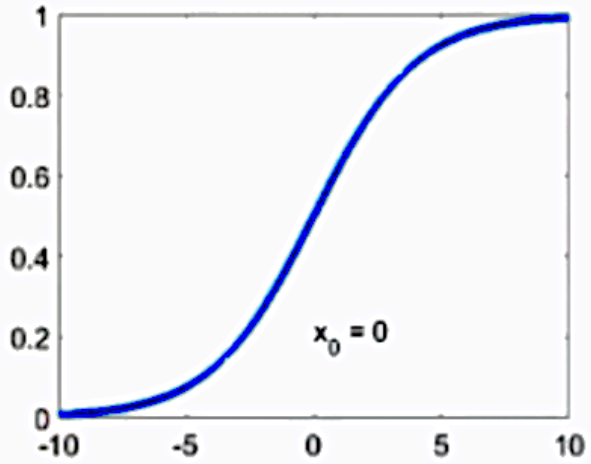
$$f(x; a, x_0) = \frac{1}{1 + e^{-a(x-x_0)}}$$

a determina la pendiente

x_0 fija el punto de cruce

y = sigmf(x,[a x0])

Si $a > 0$, abre a la derecha
Si $a < 0$, abre a la izquierda



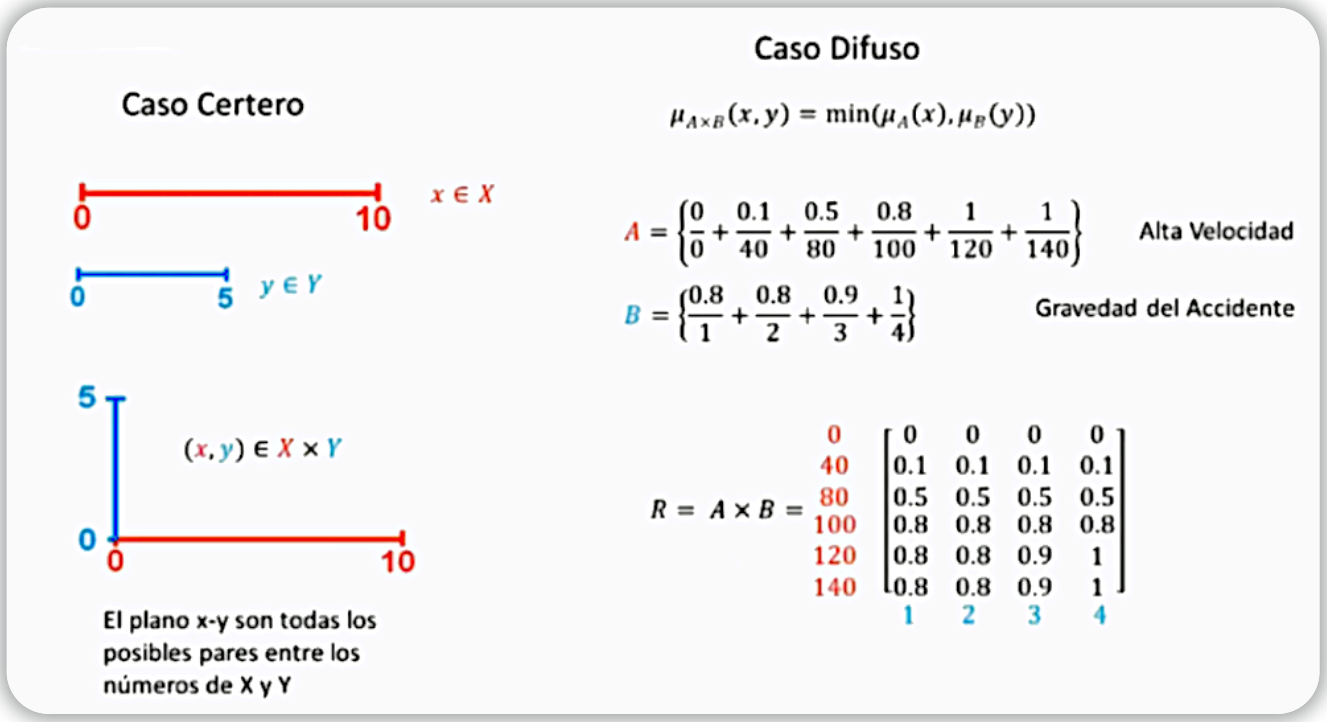
Tarea: **Debes terminar las funciones si alguna tiene pendiente parte de su implementación y verificar el funcionamiento de las mismas**, no solo con el ejemplo suministrado, y **explicar brevemente el significado de sus parámetros.**

2. Relaciones de funciones difusas

En esta parte de la práctica vamos a estudiar cómo podemos construir relaciones entre funciones difusas.

2.1. Producto cartesiano

Podemos establecer relaciones difusas entre x e y, de forma que un valor de x se va a relacionar con un valor de y, $f(x) = y$ con un grado de pertenencia. Para ello utilizamos una matriz de relación, que no es más que un **producto cartesiano**.



En el ejemplo anterior se muestra la **relación entre los vehículos que circulan a alta velocidad y la gravedad de los accidentes**, definida por el producto cartesiano. Antes de determinar el producto cartesiano es necesario ordenar el universo del discurso de A y de B de los conjuntos. Cada conjunto aporta información por separado, pero al establecer la relación aportamos nueva información.

> **Definición** Para realizar el producto cartesiano entre conjuntos difusos se aplica la función de mínimo: $\mu_{A\times B}(x,y) = \min(\mu_A(x), \mu_B(x))$

Tarea: Implementar en python la función `cartesian(mA, mB)` para determinar el producto cartesiano de dos conjuntos

```
def cartesian(mA, mB):
#   mA: numpy.ndarray, Vector con valores de pertenencia de xA
#   mB: numpy.ndarray, Vector con valores de pertenencia de xB

# Realizar
# Alta velocidad
x = np.array([0, 0.1, 0.5, 0.8, 1, 1])
# Gravedad del accidente
y = np.array([0.8, 0.8, 0.9, 1])

# Producto cartesiano de funciones discretas
cartesian(x, y)
```

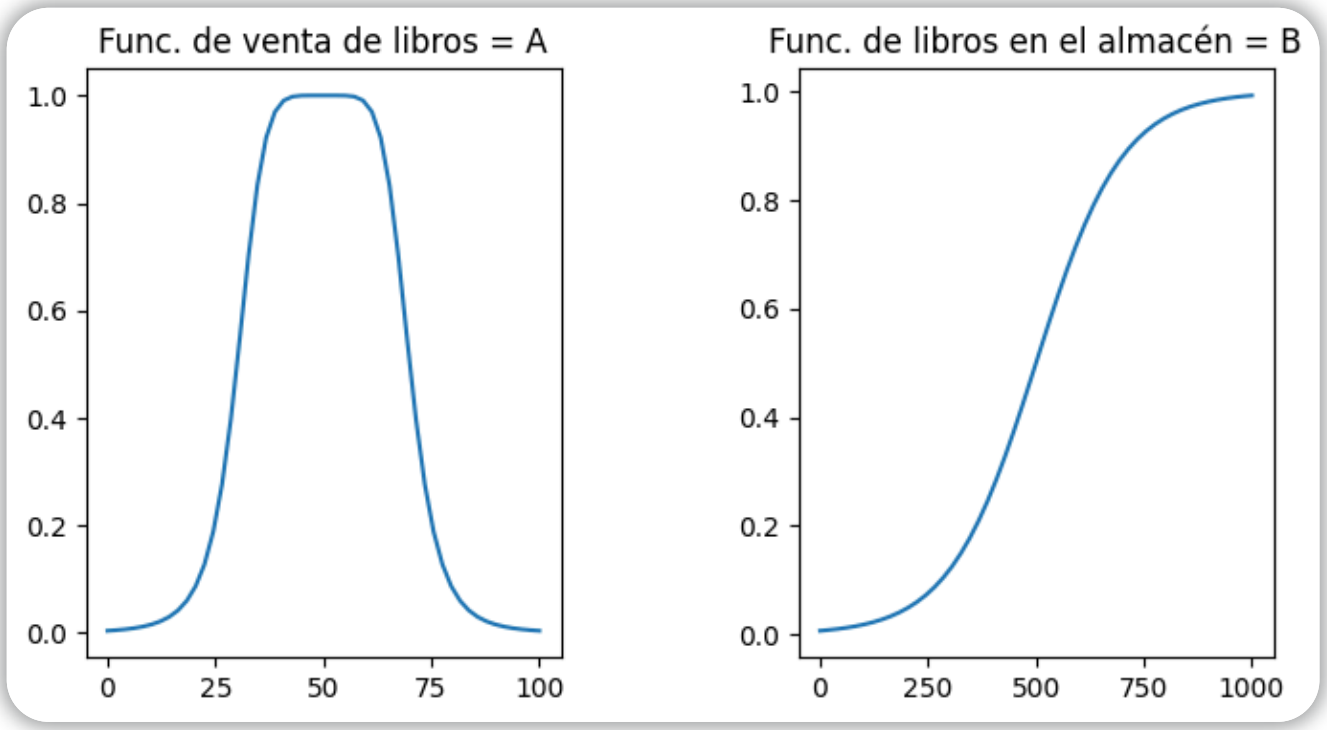
Igualmente podemos relacionar 2 funciones continuas:

- Sea A un conjunto que representa la **venta de libros** con un universo de [0,100]
- Sea B un conjunto que representa el número de libros en el almacén con un universo de [0,1000]
- Podemos definir la **función de venta de libros** como una campana de gaus, con un punto central en 50, un ancho de 20 y una pendiente de 3.
- Igualmente podemos definir la **función de libros en el almacén**, como una sigmoide con pendiente de 0.1 y punto de inflexión en 500.

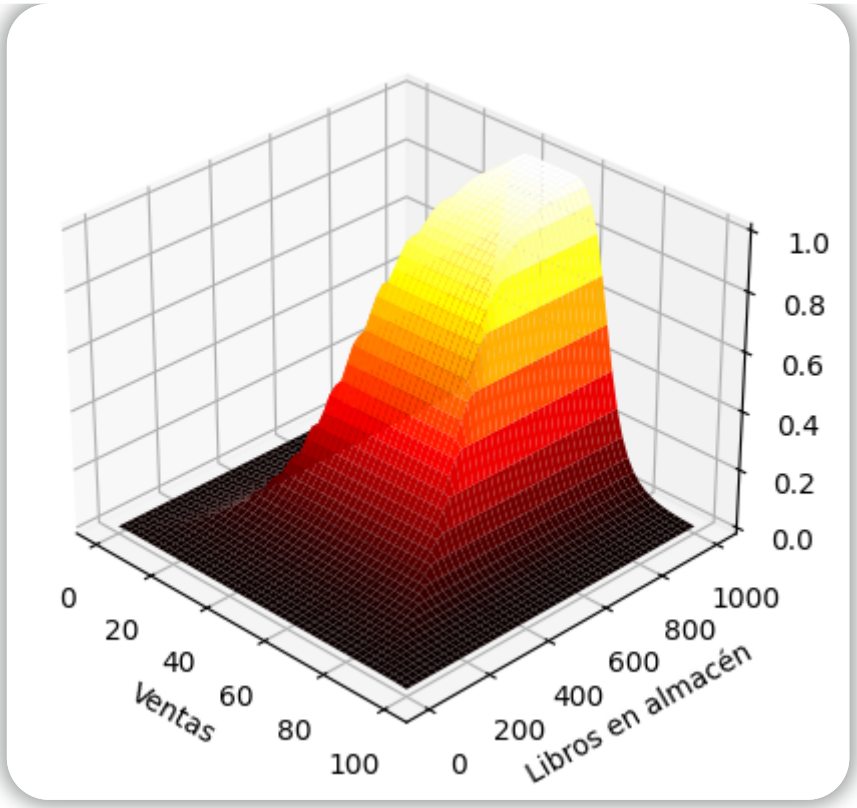
```
# Definir el tamaño de la figura
plt.figure(figsize=(8, 4))  # Ajustar el tamaño de la figura

# Generar los datos
# array de 50 elemntos de 0 1 100
# que representa la cantidad de libros vendidos mensualmente
# [0,100] libros vendidos al mes
xA = np.linspace(0, 100, 50)

# Conjunto A = función de Ventas
# campana con punto central en 50,
# ancho de a= 20
# pendiente de b= 3
mA = gbellmf(xA, [20, 3, 50])

# Libros en el almacén [0,1000]
xB = np.linspace(0, 1000, 50)

# Conjunto B = Función de libros en el almacén
mB = sigmf(xB, [0.01, 500])
```

```
21
22  # Crear los subplots
23  plt.subplot(121)
24  plt.plot(xA, mA)
25  plt.title("Func. de venta de libros = A")
26
27  plt.subplot(122)
28  plt.plot(xB, mB)
29  plt.title("Func. de libros en el almacén = B")
30
31  # Ajustar el espacio entre los subplots
32  plt.subplots_adjust(wspace=0.5)
33
34  # Mostrar la figura
35  plt.show()
```

Representamos estas funciones:



Ahora podemos calcular el producto cartesiano, tal y como lo hemos definido anteriormente mediante $\mu_{A\times B}(x,y) = \min(\mu_A(x), \mu_B(x))$ y plotearlo en 3D. Donde uno de los ejes representa las ventas (campana) y en el otro los libros almacenados (sigmoide). De la figura podemos deducir que **el máximo de ventas coincide con la mayor cantidad de libros disponibles en el almacén**. Esta información se deduce de la relación entre ambas funciones.

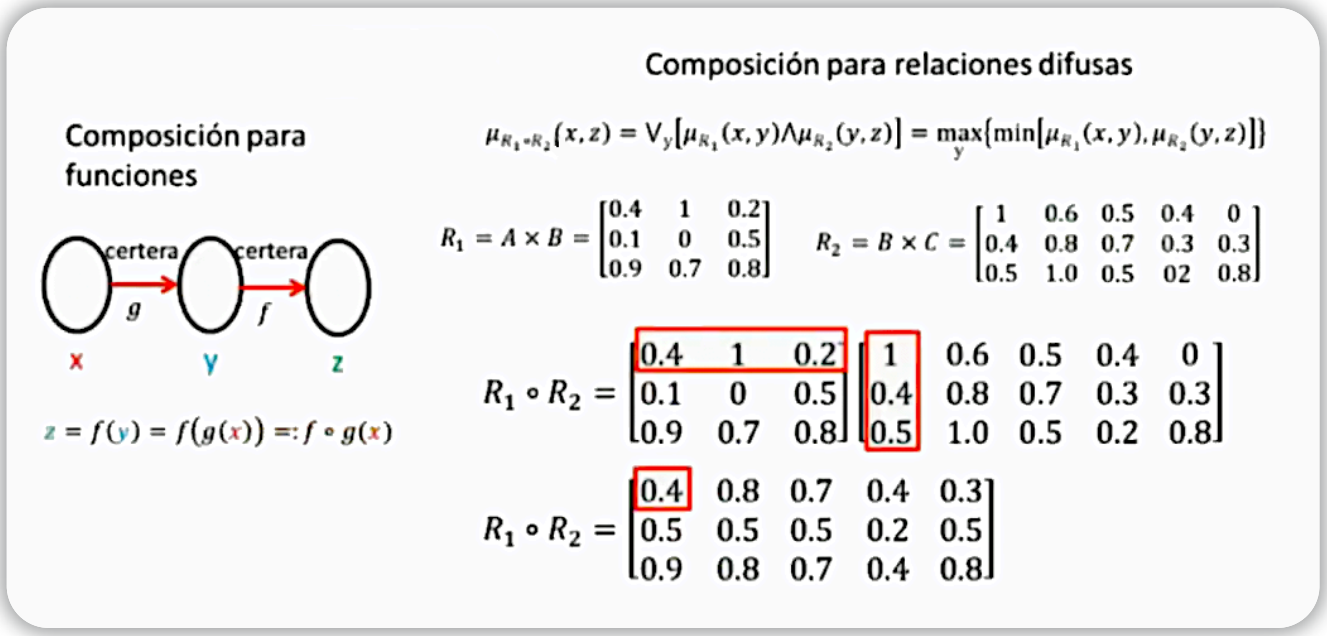


Tarea: Calcular el producto cartesiano `R_AB = cartesian(mA, mB)` y representar la figura en python.

2.2. Composición de funciones

La composición de relaciones, es similar al producto matricial, pero la operación que se realiza es la siguiente:

$$\mu_{R_1 \circ R_2}(x,z) = \bigvee_y [\mu_{R_1}(x,y) \wedge \mu_{R_2}(y,z)] = \max_y \{\min [\mu_{R_1}(x,y), \mu_{R_2}(y,z)]\}$$



Tarea: Basándonos en la función cartesian() vamos a realizar en python la función compose(), mediante la composición difusa max-min.

```
# Función compose(): Composición difusa max-min.
# Argumentos:
#   mRA: numpy.ndarray, vector o matriz de relación A
#   mRB: numpy.ndarray, Matriz de relación B
# Retorna:
#   AoB: Matriz de composición max-min
# Para este cálculo es necesario una matriz auxiliar aux[] donde se
# guardarán los mínimos

x = np.array([[0.4, 1.0, 0.2],
              [0.1, 0.0, 0.5],
              [0.9, 0.7, 0.8]])

y = np.array([[1.0, 0.6, 0.5, 0.4, 0.0],
              [0.4, 0.8, 0.7, 0.3, 0.3],
              [0.5, 1.0, 0.5, 0.2, 0.8]])


```