UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

CARRERA PROFESIONAL DE INGENIERIA INFORMATICA Y DE SISTEMAS

COMPUTACION GRAFICA 2

DOCENTE: M.SC. HECTOR E. UGARTE R.

HOJA DE EJERCICIOS 5: SPLINES Y SUPERFICIES BEZIER

1. COMPETENCIAS

- El estudiante conoce el concepto de splines.
- El estudiante calcula y grafica curvas Superficies Bezier.

2. MARCO TEORICO

Splines

El término "spline" hace referencia a una amplia clase de funciones que son utilizadas en aplicaciones que requieren la interpolación de datos, o un suavizado de curvas. Los splines son utilizados para trabajar tanto en una como en varias dimensiones.

para calcularlo, debe cumplen ciertas condiciones:

El trazador cúbico o *spline* es un conjunto de polinomios de tercer grado que se genera a partir de un conjunto de puntos y, para calcularlo, debe cumplen ciertas condiciones:

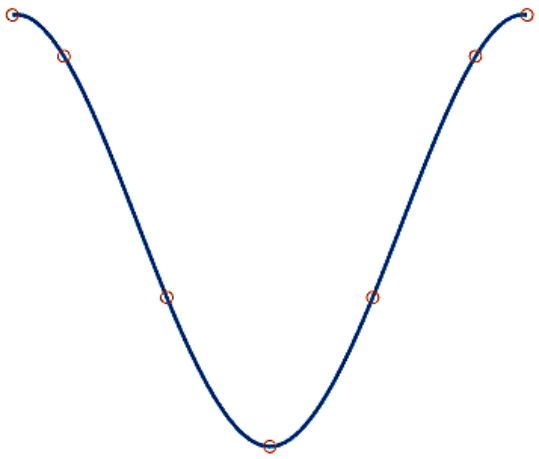
I Los polinomios pasan por los puntos: $P_k(x_k) = f(x_k)$ con $k = \{0, 1, \dots\}$ y $P_{n-1}(x_n) = f(x_n)$.

II Continuidad en los nodos interiores: $P_k(x_{k+1}) = P_{k+1}(x_{k+1})$ con $k = \{0, 1, \dots\}$.

III Derivabilidad en los nodos interiores: $P_k{}'(x_{k+1}) = P_{k+1}{}'(x_{k+1})$ con $k = \{0, 1, \dots\}$.

IV Continuidad de la primera derivada para conservar la concavidad en la vecindad de los nodos interiores: $P_k{}''(x_{k+1}) = P_{k+1}{}''(x_{k+1})$ con $k = \{0, 1, \dots\}$.

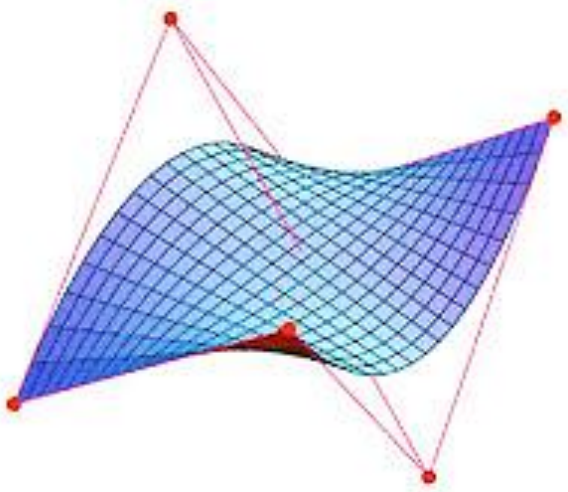
V Condición de frontera natural ($P_0{}''(x_0) = 0$ y $P_{n-1}{}''(x_n) = 0$) o frontera sujeta ($P_0{}'(x_0) = f'(x_0)$ y $P_{n-1}{}'(x_n) = f'(x_n)$)



Superficies Bezier

Tienen las mismas propiedades que las curvas de Bézier y proporcionan un método conveniente para aplicaciones interactivas de diseño. Para especificar las posiciones de los puntos de control con coordenadas tridimensionales, podríamos en primer lugar construir una cuadrícula rectangular en el plano xy «de tierra»

Una superficie de Bézier compuesta construida con dos secciones de dos secciones de Bézier, unidas por la línea indicada, las líneas discontinuas unen los puntos de control, la continuidad de primer orden se establecen haciendo que la relación entre la longitud y la longitud z.



Una superficie de Bézier bidimensional se puede definir como una superficie paramétrica donde la posición de un punto **p** en función de las coordenadas paramétricas *u* , *v* viene dada por:

$$\mathbf{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)\, B_j^m(v)\, \mathbf{k}_{i,j}$$

evaluado sobre el cuadrado unitario, donde

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

es un polinomio base de Bernstein , y

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

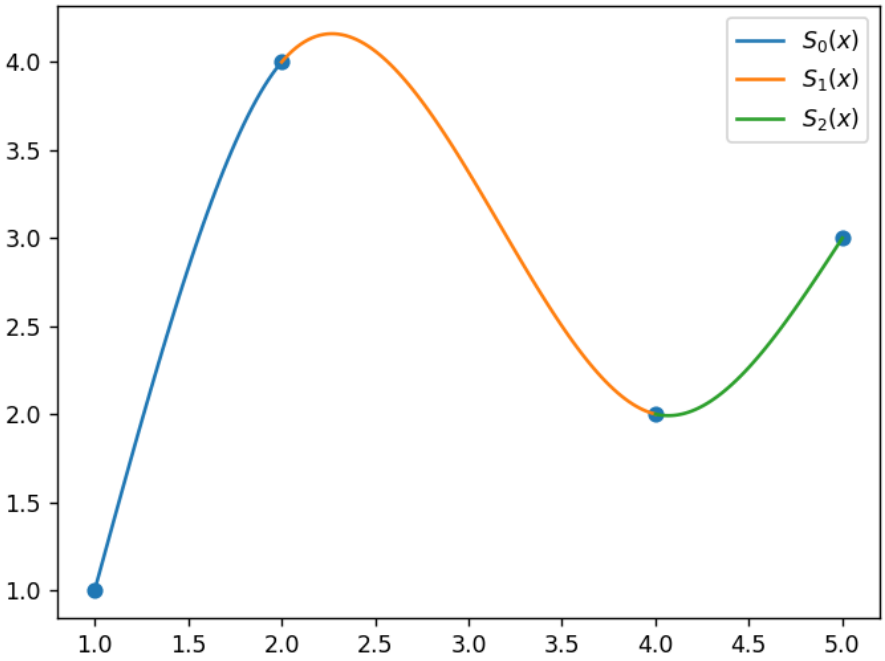
es un coeficiente binomial .

3. PRACTICA

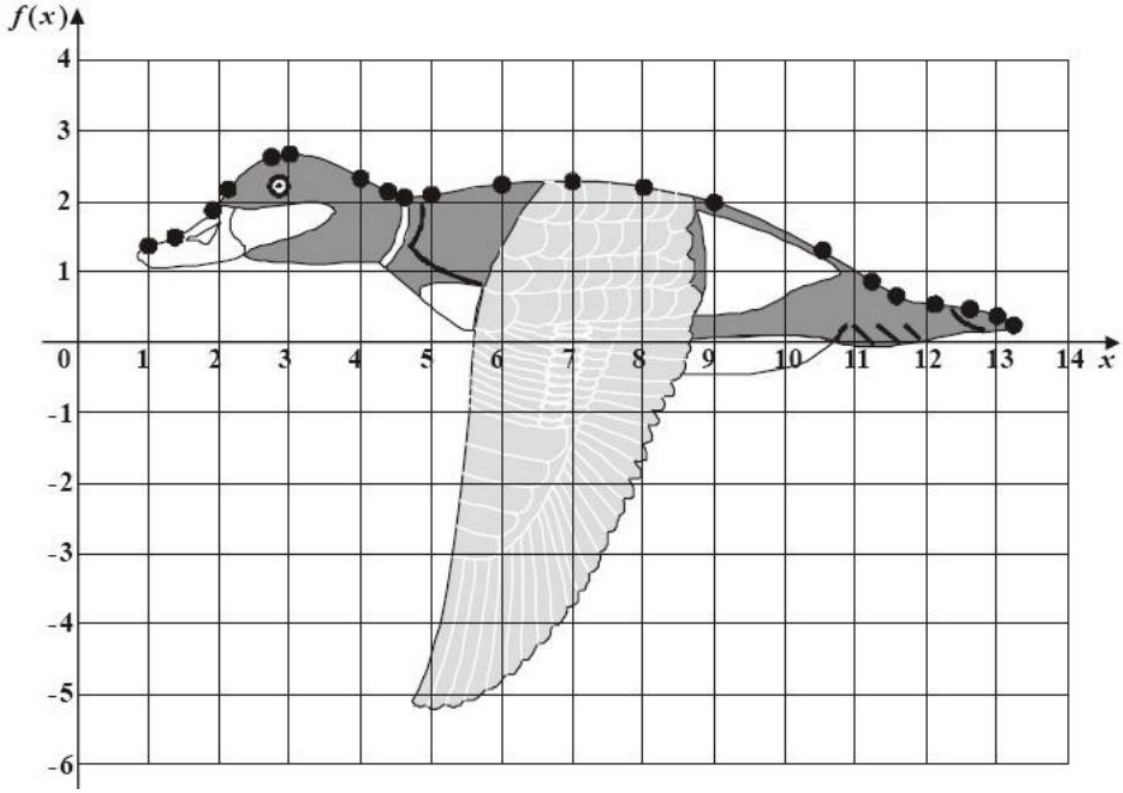
A. El siguiente código implementa un spline cubico natural. Entiéndelo y coméntalo.

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

def spline(x, y):
    n=len(x)
    a = {k: v for k, v in enumerate(y)}
    h = {k: x[k+1] - x[k] for k in range(n - 1)}

    A = [[1] + [0] * (n - 1)]
    for i in range(1, n-1):
        row = [0] * n
        row[i - 1] = h[i - 1]
        row[i] = 2*(h[i-1] + h[i])
        row[i+1] = h[i]
        A.append(row)
    A.append([0] * (n-1) + [1])

    B=[0]
    for k in range(1,n-1):
        row = 3 * (a[k+1] - a[k]) / h[k] - 3 * (a[k] - a[k-1])/h[k-1]
        B.append(row)
    B.append(0)

    c = dict(zip(range(n), np.linalg.solve(A,B)))
    b = {}
    d = {}

    for k in range(n-1):
        b[k] = (1/h[k]) * (a[k+1] - a[k]) - (h[k]/3) * (2*c[k]+c[k+1])
        d[k] = (c[k+1] - c[k])/(3*h[k])

    s = {}
    for k in range(n-1):
        eq = f'{a[k]}{b[k]:+}*(x-{x[k]}){c[k]:+}*(x-{x[k]})**2{d[k]:+}*(x-{x[k]})**3'
        s[k] = {'eq': eq, 'domain': [x[k],x[k+1]]}
    return s

x = [1,2,4,5]
y = [1,4,2,3]

eqs = spline(x,y)
print(eqs)

for key, value in eqs.items():
    def p(x):
        return eval(value['eq'])
    t = np.linspace(*value['domain'],100)
    plt.plot(t,p(t),label=f"$S_{key}(x)$")

plt.scatter(x,y)
plt.legend()
plt.savefig('spline.png')
plt.show()
```

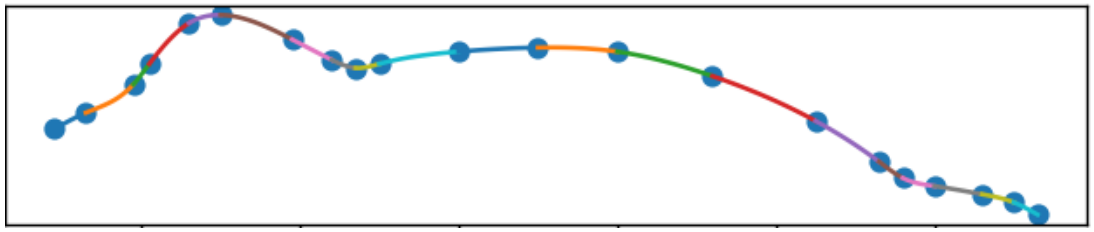
Salida:



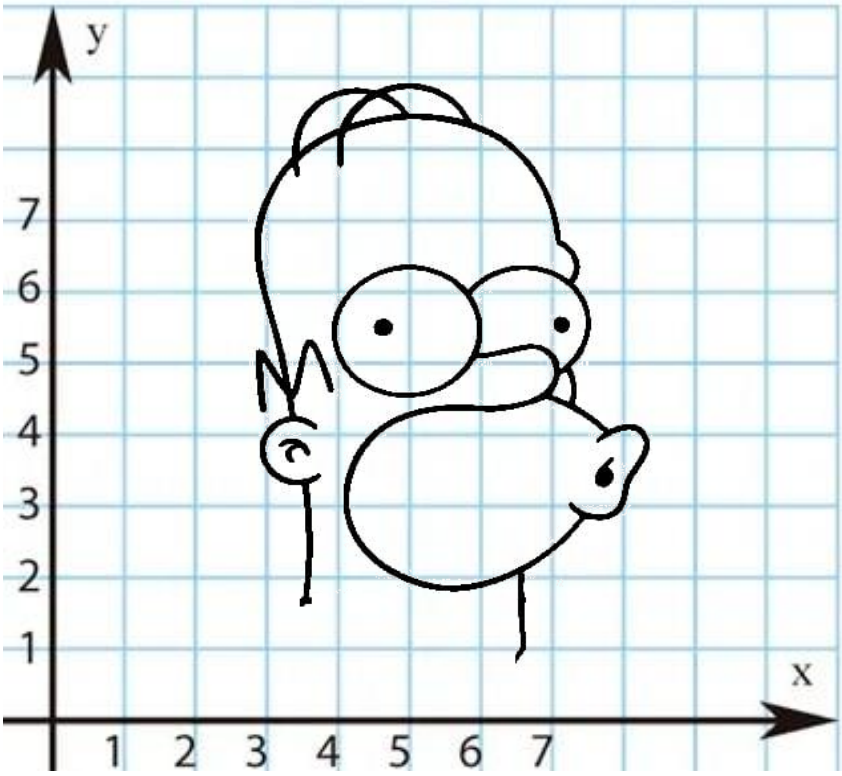
B. Implementa el ejemplo visto en clases teóricas:



Deberías obtener una salida similar a:



C. Implementa y obtén una salida lo más aproximado al siguiente modelo (posiblemente usando múltiples curvas):



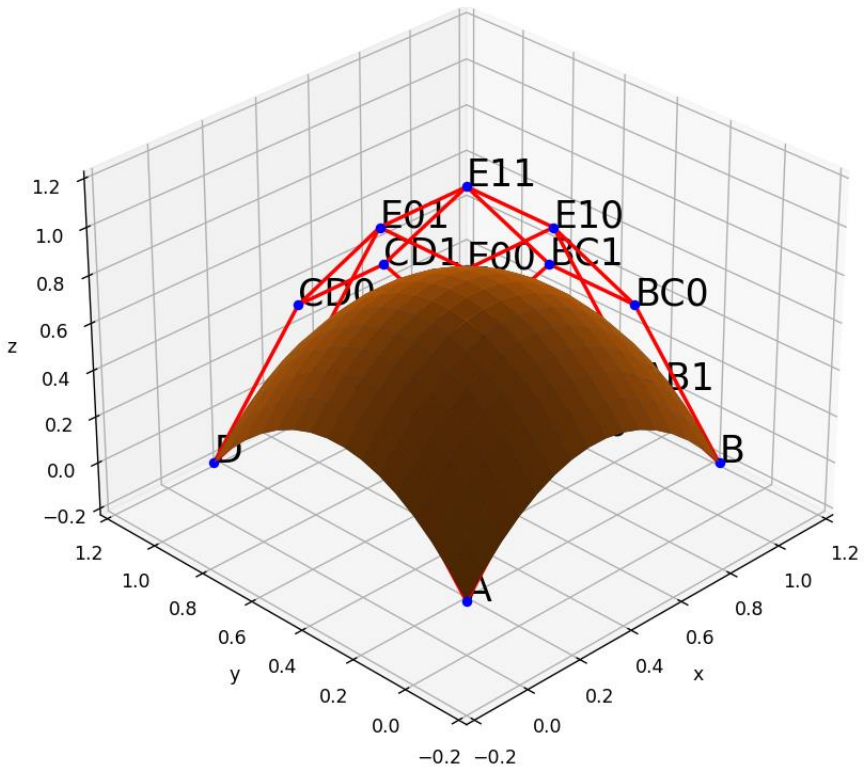
D. El siguiente código implementa una superficie Bezier bicubica. Entiéndelo y coméntalo.

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
from matplotlib.animation import FuncAnimation

# superficie bezier con 16 puntos

# puntos de borde
A = np.array([0, 0, 0])
B = np.array([1, 0, 0])
C = np.array([1, 1, 0])
D = np.array([0, 1, 0])

# puntos centrales de control
E00 = np.array([1/3, 1/3, 1])
E10 = np.array([2/3, 1/3, 1])
E11 = np.array([2/3, 2/3, 1])
E01 = np.array([1/3, 2/3, 1])
```

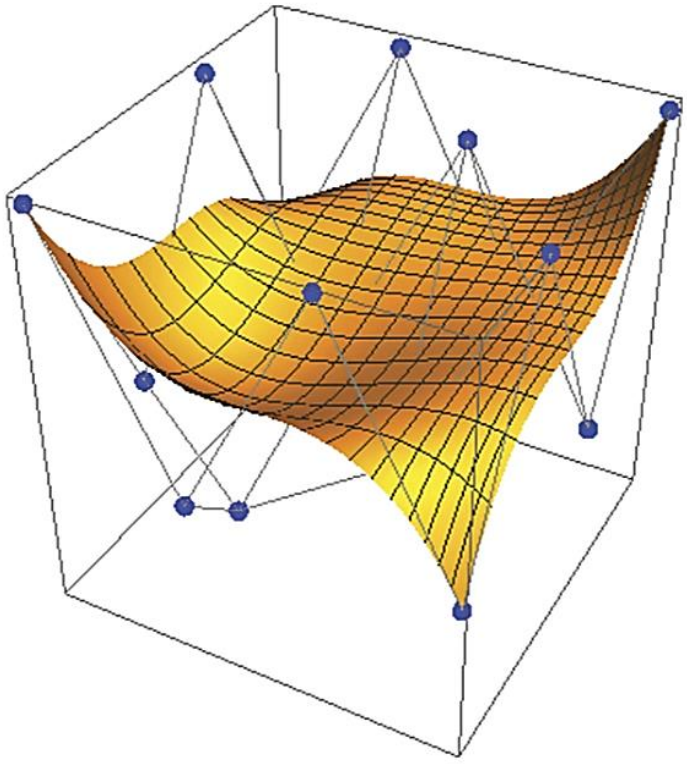
```
# puntos de control
AB0 = np.array([1/3, 0, 1/2])
AB1 = np.array([2/3, 0, 1/2])
BC0 = np.array([1, 1/3, 1/2])
BC1 = np.array([1, 2/3, 1/2])
CD1 = np.array([2/3, 1, 1/2])
CD0 = np.array([1/3, 1, 1/2])
DA1 = np.array([0, 2/3, 1/2])
DA0 = np.array([0, 1/3, 1/2])

# interpolacion lineal
n = 16
ss = np.linspace(0, 1, n)
ts = np.linspace(0, 1, n)
shape = (n, n, 3)
def interpolar0(A, B, C, D):
    return np.array([
        (1-t)*(1-s)*A + (1-t)*s*B + t*(1-s)*D + t*s*C
        for s in ss for t in ts]).reshape(shape)
def interpolar(A, B, C, D):
    return np.array([
        (1-t)*(1-s)*A[i, j] + (1-t)*s*B[i, j] + t*(1-s)*D[i, j] + t*s*C[i, j]
        for i, s in enumerate(ss) for j, t in enumerate(ts)]).reshape(shape)

# primera interpolacion
P00 = interpolar0(A, AB0, E00, DA0)
P01 = interpolar0(AB0, AB1, E10, E00)
P02 = interpolar0(AB1, B, BC0, E10)
P10 = interpolar0(DA0, E00, E01, DA1)
P11 = interpolar0(E00, E10, E11, E01)
P12 = interpolar0(E10, BC0, BC1, E11)
P20 = interpolar0(DA1, E01, CD0, D)
P21 = interpolar0(E01, E11, CD1, CD0)
P22 = interpolar0(E11, BC1, C, CD1)

# segunda interpolacion
Q00 = interpolar(P00, P01, P11, P10)
Q01 = interpolar(P01, P02, P12, P11)
Q10 = interpolar(P10, P11, P21, P20)
Q11 = interpolar(P11, P12, P22, P21)

# ultima interpolacion
R = interpolar(Q00, Q01, Q11, Q10)

xs = R[:,:,0]
ys = R[:,:,1]
zs = R[:,:,2]

# dibujar
fig = plt.figure(figsize=(8, 8))
s = fig.add_subplot(111, projection="3d")

# dibujar superficie
s.plot_surface(xs, ys, zs)

# dibujar superficie
s.plot_surface(xs, ys, zs)

# dibujar alambres
def alambre(A, B):
    xyz = list(zip(A, B))
    s.plot(xyz[0], xyz[1], xyz[2], color="r", lw=2)
    pass
def alambres(A, B, C, D):
    alambre(A, B); alambre(B, C); alambre(C, D)
    pass
alambres(A, AB0, AB1, B)
alambres(D, CD0, CD1, C)
alambres(A, DA0, DA1, D)
```

```
alambres(B, BC0, BC1, C)
alambres(DA0, E00, E10, BC0)
alambres(DA1, E01, E11, BC1)
alambres(AB0, E00, E01, CD0)
alambres(AB1, E10, E11, CD1)

# dibujar puntos
def poner_punto(p, label):
    s.scatter(p[0], p[1], p[2], color="b")
    s.text(p[0], p[1], p[2], label, size=20, color="k", zorder=1)
    pass
poner_punto(A, "A")
poner_punto(B, "B")
poner_punto(C, "C")
poner_punto(D, "D")
poner_punto(E00, "E00")
poner_punto(E01, "E01")
poner_punto(E11, "E11")
poner_punto(E10, "E10")
poner_punto(AB0, "AB0")
poner_punto(AB1, "AB1")
poner_punto(BC0, "BC0")
poner_punto(BC1, "BC1")
poner_punto(CD0, "CD0")
poner_punto(CD1, "CD1")
poner_punto(DA0, "DA0")
poner_punto(DA1, "DA1")

# axis
s.set_xlim(-0.2, 1.2)
s.set_ylim(-0.2, 1.2)
s.set_zlim(-0.2, 1.2)
s.set_xlabel("x")
s.set_ylabel("y")
s.set_zlabel("z")

s.view_init(elev=30., azim=-135)
plt.show()
```

Salida:



E. Implementa y obtén una salida lo más aproximado a la siguiente superficie.



Los 16 puntos de control son: