



**Universidad Autónoma del Estado de México**

**Centro Universitario UAEM Zumpango**

**Ingeniería en Computación**

**Graficación Computacional**

Periodo 2022B

Matrícula: 1925320

**Keren Mitsue Ramírez Vergara**

Profesor:

**Manuel Almeida Vázquez**

Zumpango, Estado de México

# Introducción

La Graficación, también está relacionada con el diseño de nuevos dispositivos o nuevas técnicas de interacción humano-computadora. Define las técnicas y los fundamentos que son aplicados a la Animación por Computadora, la Visualización, así como la creación de Ambientes Virtuales o Realidad Virtual.

De ésta manera, la Graficación por Computadora permite establecer un diálogo o comunicación muy diferente entre el hombre y la computadora que es llevado a cabo a través de imágenes y con las técnicas interactivas más novedosas. Las áreas de la aplicación de la Graficación son: la arquitectura, el diseño industrial, el CAD/CAM, desarrollo de aplicaciones en el WEB, laboratorios virtuales, la robótica, entrenamiento virtual, simulaciones, aplicaciones forenses, recrear situaciones especiales, rehabilitación, los sistemas geográficos, la meteorología, visualización científica, ingeniería en todas sus áreas, medicina, desarrollo de interfaces humano-computadora, reconstrucción de sitios arqueológicos, arte, educación, entretenimiento (video-juegos, películas, televisión, publicidad), entre otras. La graficación computacional es una área de las Ciencias de la computación, su principal objetivo es establecer los principios, técnicas y algoritmos para la generación y manipulación de las imágenes mediante una computadora. Las imágenes pueden ser bidimensionales o tridimensionales. De esta manera la graficación computacional permite establecer una interacción especial entre una computadora y el hombre.

Gracias al gran avance tecnológico que ha experimentado el hardware, se utilizan métodos de graficación de forma generalizada, con el propósito de producir películas, programas de televisión, modelos de sistemas financieros, fisiológicos, económicos, entre otros.

# Contenido

<b>Introducción</b>	<b>ii</b>
<b>1 Transformaciones</b>	<b>1</b>
1.1 Traslación . . . . .	1
1.2 Reflexión . . . . .	4
1.3 Escalamiento sobre el origen . . . . .	5
1.4 Rotación en el plano . . . . .	8
<b>2 Coordenadas Homogéneas</b>	<b>10</b>
2.1 Transformaciones en coordenadas homogéneas . . . . .	10
2.1.1 Traslación . . . . .	10
2.1.2 Reflexión . . . . .	11
2.1.3 Escalamiento . . . . .	11
2.1.4 Rotación . . . . .	11
2.2 Concatenación de transformaciones . . . . .	11
<b>3 Transformaciones inversas</b>	<b>17</b>
3.1 Escalamiento . . . . .	17
3.2 Rotación . . . . .	17
3.3 Traslación . . . . .	17
<b>4 Interpolación lineal</b>	<b>19</b>
<b>5 Morphing</b>	<b>24</b>
<b>6 Transformaciones en 3D</b>	<b>28</b>
6.1 Coordenadas homogéneas . . . . .	28
6.2 Transformaciones en el espacio . . . . .	29
6.2.1 Traslación . . . . .	29
6.2.2 Escalamiento . . . . .	31
6.2.3 Reflexiones . . . . .	34
6.2.4 Rotación . . . . .	35
<b>7 Curvas</b>	<b>39</b>
7.1 Curvas de b茅zier . . . . .	40
7.2 Curvas lineales de B茅zier . . . . .	41
7.3 Curvas cuadr谩ticas de B茅zier . . . . .	41
7.4 Estructura general de las curvas de B茅zier . . . . .	42
7.5 Forma matricial de curvas de B茅zier . . . . .	44
7.6 Combinaciones de curvas de B茅zier . . . . .	45
7.7 Parametrizaci茅n de curvas de B茅zier . . . . .	48

<b>8 Superficies</b>	<b>49</b>
8.1 Superficies paramétricas . . . . .	49
8.2 Superficies por deslizamiento(swapping) . . . . .	55
8.3 Superficies por rotación (superficies de revolución) . . . . .	55
8.4 Curvas sobre superficies . . . . .	57
8.5 Superficies de b茅zier . . . . .	58
8.5.1 Productos cartesianos . . . . .	59
8.6 Curvatura . . . . .	62
<b>9 Superficies minimales</b>	<b>64</b>
9.1 Introducción . . . . .	64
9.2 Pel铆culas de jab贸n y superficies m铆nimas . . . . .	64
9.3 El problema de Plateau . . . . .	65
9.3.1 Leyes de Plateau . . . . .	65
9.4 El problema de Steiner . . . . .	65
9.5 Superficies minimales . . . . .	66
9.5.1 Catenoide . . . . .	68
9.5.2 Helicoide . . . . .	68
9.5.3 Superficie de Enneper . . . . .	69
<b>10 Triangulaci贸n de Delaunay</b>	<b>70</b>
10.1 Introducción . . . . .	70
10.2 Triangulaci贸n de Delaunay . . . . .	70
10.3 Condici贸n de Delaunay . . . . .	71
10.4 Propiedades de la triangulaci贸n de Delaunay . . . . .	71
<b>11 C谩culo</b>	<b>76</b>
11.1 Recta tangente a una curva en un punto . . . . .	76
11.2 Integral definida con sumas de Riemann . . . . .	78
<b>12 Conclusiones</b>	<b>80</b>

# Capítulo 1

## Transformaciones

Una transformación  $M : R^n \rightarrow R^n$  se llama movimiento, isometría o transformación congruente de  $R^n$  si

$$|M(p)M(q)| = |pq|$$

para cada par de puntos  $(p, q) \in R^n$

### 1.1 Traslación

Una traslación  $T(h, k)$  es una transformación que mapea un punto  $P(x, y)$  a un punto  $P'(x', y')$  añadiendo una cantidad constante a cada coordenada

$$\begin{aligned}x' &= x + h \\y' &= y + k\end{aligned}$$

Donde,  $h$  es la cantidad de unidades que se mueve el punto  $P(x, y)$  en dirección al eje  $x$ , y  $k$  es la cantidad de unidades que se mueve el punto  $P(x, y)$  en dirección al eje  $y$ . El punto  $(x', y')$  expresado como un vector fila es:

$$(x', y') = (x, y) + (h, k)$$

Para transladar un objeto o figura es necesario añadir un vector  $(h, k)$  a cada punto de ese objeto, considerando la adición de matrices si  $(x, y)$  se representa como una matriz de filas.

#### Ejercicio 1.1.1

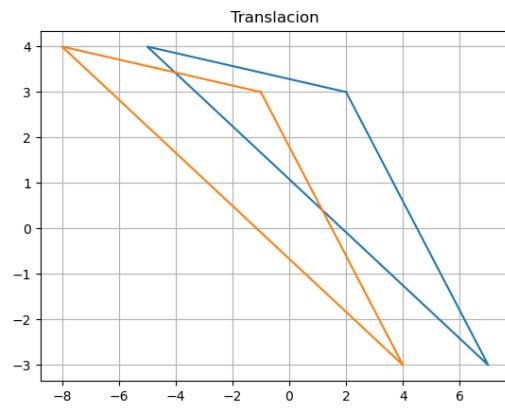
Considerando una figura que esta compuesta por los puntos  $f = \{(2, 3), (-5, 4), (7, -3)\} \in R^2$ , se desea trasladar en el eje  $x$  3 unidades a la izquierda. Como se observa en la figura 1.1a, a los puntos que corresponden al eje  $x$  serán sumados -3 unidades para que se trasladen de derecha a izquierda. (línea 26 del código).

```

3 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4 CU UAEU ZUMPANGO
5 UA: Graficación computacional
6 Tema: Translacion
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Mostrar diferentes tipos de translacion
10 (en eje x, en eje y, en ambos)
11 Created on Tue Aug 9 12:45:07 2022
12
13 @author: KerenMitsue
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19
20 P = np.array([[2,3], [-5,4], [7,-3], [2,3]])
21 h = -3
22 k = 4
23
24 figure, ax = plt.subplots()
25 ax.plot(P[:,0], P[:,1])
26 ax.plot(P[:,0]+h, P[:,1])
27 ax.grid()
28 ax.set_title('Translacion')
29 plt.show()

```

(a) Implementación en código



(b) Ejecución de código

Figura 1.1: Translación en eje X

### Ejercicio 1.1.2

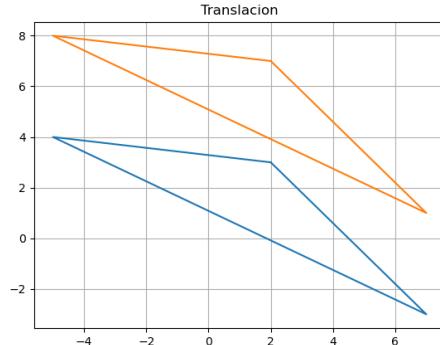
Considerando la misma figura del ejercicio 1.1, compuesta por los puntos  $f = \{(2, 3), (-5, 4), (7, -3)\} \in R^2$ , trasladar sobre el eje  $y$  4 unidades hacia arriba. Como se observa en la figura 1.2a, a los puntos que corresponden al eje  $y$  serán sumados 4 unidades para que la figura se traslade hacia arriba. (línea 26 del código).

```

6 Tema: Translacion
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Mostrar diferentes tipos de translacion
10 (en eje x, en eje y, en ambos)
11 Created on Tue Aug 9 12:45:07 2022
12
13 @author: computerappliedresearch
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19
20 P = np.array([[2,3], [-5,4], [7,-3], [2,3]])
21 h = -3
22 k = 4
23
24 figure, ax = plt.subplots()
25 ax.plot(P[:,0], P[:,1])
26 ax.plot(P[:,0], P[:,1]+k)
27 ax.grid()
28 ax.set_title('Translacion')
29 plt.show()

```

(a) Implementación en código



(b) Ejecución del código

Figura 1.2: Translación en eje Y

### Ejercicio 1.1.3

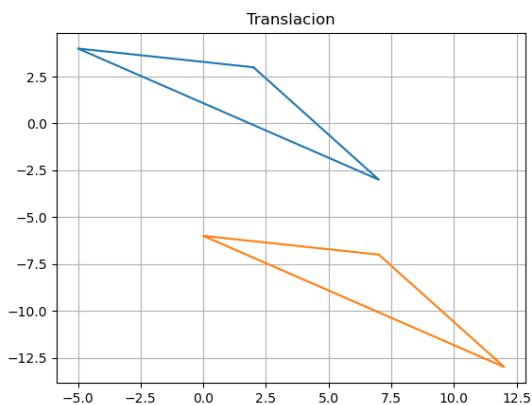
Dado un triángulo, conformado por los puntos  $f = \{(2, 3), (-5, 4), (7, -3)\} \in R^2$ , trasladar sobre el eje  $y$  10 unidades hacia abajo y en el eje  $x$  5 unidades a la derecha. Los valores que se consideran para realizar esta translación son  $h = 5$  y  $k = -10$ , como se ilustra en la figura 1.3

```

5 UA: Graficación computacional
6 Tema: Translacion
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Mostrar diferentes tipos de translacion
10      (en eje x, en eje y, en ambos)
11 Created on Tue Aug  9 12:45:07 2022
12
13 @author: computerappliedresearch
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19
20 P = np.array([[2,3], [-5,4],[7,-3], [2,3]])
21 h = 5
22 k = -10
23
24 figure, ax = plt.subplots()
25 ax.plot(P[:,0], P[:,1])
26 ax.plot(P[:,0]+h, P[:,1]+k)
27 ax.grid()
28 ax.set_title('Translacion')
29 plt.show()

```

(a) Código



(b) Ejecución del código

Figura 1.3: Translación en eje X y Y

#### Ejercicio 1.1.4

Generar una animación donde la figura que está conformada por los puntos  $f = \{(2, 3), (-5, 4), (7, -3)\} \in R^2$ , se traslade en el eje horizontal. Considere mostrar cada una de las translaciones.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4 CU UAEIM ZUMPANGO
5 UA: Graficación computacional
6 Tema: Translacion
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Generar una animación de la translación de una figura
10    Consideré mostrar cada una de las translaciones.
11 """
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15 import time as tm
16
17
18 P = np.array([ [2,3],[-5,7], [1,-5],[2,3] ], dtype=float)
19 h = 0.5
20 k= -3
21 plt.axis([-5,12,-6,10]) #mantener ejes fijos
22
23 plt.plot(P[:,0], P[:,1])
24
25
26 for i in range(1,10):
27     plt.plot(P[:,0] + i*h, P[:,1])
28     plt.axis('off')
29     plt.pause(0.5)
30
31 tm.sleep(2)

```

Three screenshots of a Jupyter Notebook cell showing the progression of the animation. The first shows a single blue triangle. The second shows several overlapping triangles in various colors. The third shows many overlapping triangles filling the plot area.

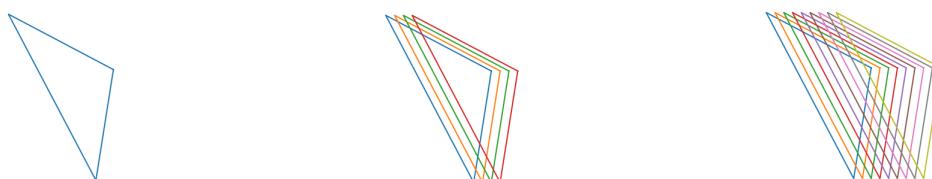


Figura 1.4: Animación de translación

## 1.2 Reflexión

La reflexión se utiliza habitualmente en paquetes de CAD o de dibujo por ordenador, estos son efectos de “voltero” o de “espejo” horizontal y vertical. La reflexión  $R_x$  en el eje  $x$  es la transformación  $L(x, y) = (x, -y)$ . La reflexión  $R_y$  en el eje  $y$  es la transformación  $L(x, y) = (-x, y)$ .

La reflexión  $R_x$  puede calcularse mediante la multiplicación matricial:

$$R_X(x, y) = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

La reflexión  $R_x$  puede calcularse mediante la multiplicación matricial:

$$R_Y(x, y) = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

### Ejercicio 1.2.1

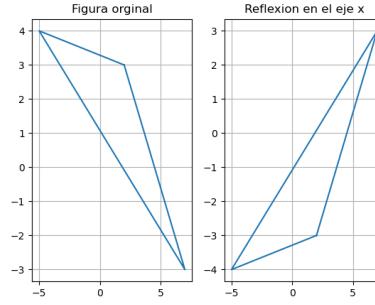
Aplica una reflexión  $R_x$  en un triángulo conformado por los puntos  $(2, 3), (-5, 4)$  y  $(7, -3)$

```

3 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4 CU UAEH ZUMPANGO
5 UA: Gráfificación computacional
6 Tema: Reflexión
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Aplicar la reflexión a una figura en el eje x
10 @author: computerappliedresearch
11 """
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15
16 P = np.array([[2,3], [-5,4], [7,-3], [2,3]])
17 Rx = np.array([[1,0], [0,-1]])
18 T = P@Rx #Producto entre matrices
19
20
21 fig = plt.figure()
22 ax = fig.add_subplot(1,2,1)
23 ax.plot(P[:,0], P[:,1])
24 ax.grid()
25 ax.set_title('Figura original')
26 ax1 = fig.add_subplot(1,2,2)
27 ax1.plot(T[:,0], T[:,1])
28 ax1.grid()
29 ax1.set_title('Reflexión en el eje x')

```

(a) Implementación en código



(b) Reflexión de un triángulo en eje X

Figura 1.5: Reflexión en  $R_x$

### Ejercicio 1.2.2

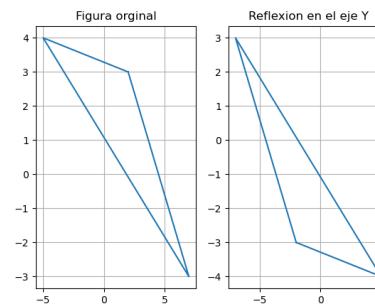
Ahora, aplica una reflexión  $R_y$  con el mismo triángulo conformado por los puntos  $(2, 3), (-5, 4)$  y  $(7, -3)$

```

3 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4 CU UAEH ZUMPANGO
5 UA: Gráfificación computacional
6 Tema: Reflexión
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Aplicar la reflexión a una figura en el eje y
10 @author: computerappliedresearch
11 """
12
13 import numpy as np
14 import matplotlib.pyplot as plt
15
16 P = np.array([[2,3], [-5,4], [7,-3], [2,3]])
17 Ry = np.array([-1,0])
18 T = P@Ry #Producto entre matrices
19
20 fig = plt.figure()
21 ax = fig.add_subplot(1,2,1)
22 ax.plot(P[:,0], P[:,1])
23 ax.grid()
24 ax.set_title('Figura original')
25 ax1 = fig.add_subplot(1,2,2)
26 ax1.plot(T[:,0], T[:,1])
27 ax1.grid()
28 ax1.set_title('Reflexión en el eje Y')

```

(a) Implementación de código



(b) Triángulo con reflexión en eje Y

Figura 1.6: Reflexión en  $R_y$

### 1.3 Escalamiento sobre el origen

Una escala sobre el origen es una transformación que mapea un punto  $P(x, y)$  a un punto  $P'(x', y')$  multiplicando las coordenadas  $x$ ,  $y$  por factores de escalas constantes diferentes de cero.  $S_x$  y  $S_y$  respectivamente, para obtener:

$$x' = S_x x \text{ y } y' = S_y y$$

Por lo que, un punto  $P'(x', y')$  tiene la estructura  $S(P) = (S_x x, S_y y)$ .

Si se desea aplicar esta transformación a una figura, existen cinco posibilidades, en las que se puede aplicar el escalamiento:

1. Si  $S_x$  y  $S_y > 1$ , se tiene una expansión en ambos ejes. Si solo es en el eje  $S_x$  entonces se amplia (ensanchar) la figura, si sólo es en el eje  $S_y$  entonces se alarga.
2. Si  $S_x < 0$  y  $S_y \leq 1$ , se tiene una reducción, considerando en que eje se aplica el escalamiento.
3. Si  $S_x$  y  $S_y = 0$ , la figura colapsa.
4. Si  $S_x$  y  $S_y$  son negativos, para los casos 1 y dos, se expanden o contraen inversamente.
5. Si  $S_x = S_y$ , se dice que es una transformación de escala uniforme

Para representar un punto  $P(x, y)$  como una matriz de filas  $(x \ y)$  la matriz de transformación de escala puede realizarse mediante una multiplicación matricial.

$$P' = (x \ y) \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} = (s_x x \ s_y y)$$

Como se observa en la ecuación, la matriz de transformación de escala es:

$$S(s_x x, s_y y) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

#### Ejercicio 1.3.1

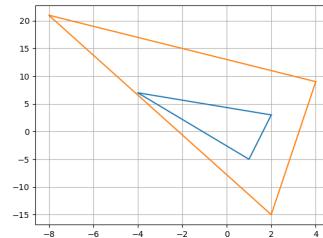
Aplica las transformaciones de escala  $S(2, 3)$  a un triángulo conformado por los puntos  $(2, 3)$ ,  $(-5, 4)$  y  $(7, -3)$ , muestra las coordenadas  $P'(x', y')$  obtenidas al aplicar dicho escalamiento.

```

3  """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4  CU UAEM ZUMPANGO
5  UA: Graficación computacional
6  Tema: Transformación: escalado
7  Alumno: Keren Mitsue Ramírez Vergara
8  Profesor: Manuel Almeida Vázquez
9  Descripción:
10    Aplicar una translación a un trángulo
11 Created on Fri Aug 12 12:44:09 2022
12
13 @author: KerenMitsue
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19 P = np.array([ [2,3], [-4,7], [1,-5], [2,3] ], dtype=float)
20 sx = float(input("escriba el factor de escala sx: "))
21 sy = float(input("escriba el factor de escala sy: "))
22 S = np.array([ [sx,0], [0,sy] ])
23 T = P@S
24
25 #plt.axis([-10,10,-10,10])
26 plt.plot(P[:,0], P[:,1])
27
28 plt.plot(T[:,0], T[:,1])
29 plt.grid()
30 plt.show()
31
32 print(T)

```

(a) Implementación de código



(b) Escalamiento  $S(2, 3)$

Figura 1.7: Escalamiento de un triángulo

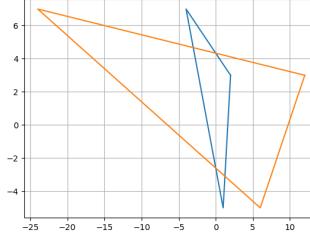
Las coordenadas para cada punto  $P'(x', y')$ , se muestran en la figura 1.7a.

#### Ejercicio 1.3.2

Ejecutando el código anterior, comprueba que si solo se aplica un escalamiento en  $S_x$ , la figura conformada por los puntos  $(2, 3), (-5, 4)$  y  $(7, -3)$  se ensancha.

```
escriba el factor de escala sx: 6
escriba el factor de escala sy: 1
[[ 12.   3.]
 [-24.   7.]
 [  6.  -5.]
 [ 12.   3.]]
```

(a) Factores de escalamiento



(b) Triángulo ensanchado

Figura 1.8: Escalamiento  $S(6, 1)$  (ensanchar triángulo)

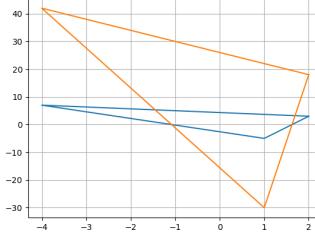
Las coordenadas para cada punto  $P'(x', y')$  son  $\begin{pmatrix} 12 & 3 \\ -24 & 7 \\ 6 & -5 \\ 12 & 3 \end{pmatrix}$ , como se muestra en la figura 1.8a

#### Ejercicio 1.3.3

Ejecutando el código anterior, comprueba que si solo se aplica un escalamiento en  $S_y$ , la figura conformada por los puntos  $(2, 3), (-5, 4)$  y  $(7, -3)$  se alarga

```
escriba el factor de escala sx: 1
escriba el factor de escala sy: 6
[[ 2.  18.]
 [-4.  42.]
 [ 1. -30.]
 [ 2.  18.]]
```

(a) Factores de escalamiento



(b) Triángulo alargado

Figura 1.9: Escalamiento  $S(6, 1)$  (alargar figura)

Las coordenadas para cada punto  $P'(x', y')$  son  $\begin{pmatrix} 2 & 18 \\ -4 & 42 \\ 1 & -30 \\ 2 & 18 \end{pmatrix}$ , como se muestra en la figura 1.9a

#### Ejercicio 1.3.4

Implementando el mismo código, aplica las transformaciones de escala  $S(-1, 4)$  a un triángulo conformado por los puntos  $(2, 3), (-5, 4)$  y  $(7, -3)$ , muestra las coordenadas  $P'(x', y')$  obtenidas al aplicar dicho escalamiento.

```

escriba el factor de escala sx: -1
escriba el factor de escala sy: 4
[[-2. 12.]
 [ 4. 28.]
 [-1. -20.]
 [-2. 12.]]

```

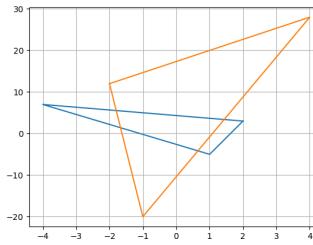


Figura 1.10: Escalamiento  $S(-1,4)$  a un triángulo

#### Ejercicio 1.3.5

En la figura 1.11 se demuestra que si se aplica una escala  $S(0,0)$  a una figura, este colapsa.

```

escriba el factor de escala sx: 0
escriba el factor de escala sy: 0
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]

```

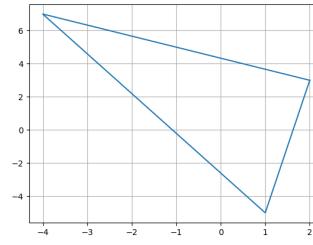


Figura 1.11: Escalamiento  $S(0,0)$

#### Ejercicio 1.3.6

En la figura 1.12, se muestra que si  $S_x$  y  $S_y$  son negativos, se expande o se contraen inversamente. En la figura se aplica una transformación de escala a  $S(-2, -2)$

```

escriba el factor de escala sx: -2
escriba el factor de escala sy: -2
[[-4. -6.]
 [ 8. -14.]
 [-2. 10.]
 [-4. -6.]]

```

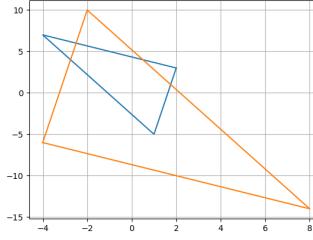


Figura 1.12: Escalamiento  $S(-2,-2)$  a una figura

#### Ejercicio 1.3.7

Generar una animación donde la figura que está formado por los puntos  $(2, 3)$ ,  $(-5, 4)$  y  $(7, -3)$ , se escale en el eje  $x$  4 unidades y en el eje  $y$  3 unidades. Considere mostrar cada una de los escalamientos.

```

3 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4 CU UAEM ZUMPANGO
5 UA: Graficación computacional
6 Tema: Escalamiento
7 Alumno: Keren Mitsue Ramírez Vergara
8 Profesor: Manuel Almeida Vázquez
9 Descripción: Animacion del escalamiento en una figura (triangulo)
10
11 Created on Tue Aug 23 03:40:28 2022
12
13 @author: KerenMitsue
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19 P = np.array([ [2,3],[-4,7], [1,-5], [2,3] ], dtype=float)
20 sx = float(input("escriba el factor de escala sx: "))
21 sy = float(input("escriba el factor de escala sy: "))
22 #plt.axis([-20,20,-20,20])
23 for i in range(1,20):
24     S = np.array([ [sx*i,0], [0,sy*i] ])
25     T = P@S
26     plt.plot(P[:,0], P[:,1])
27     plt.plot(T[:,0], T[:,1])
28     plt.axis('off')
29     plt.pause(0.5)
30
31 print(T)

```

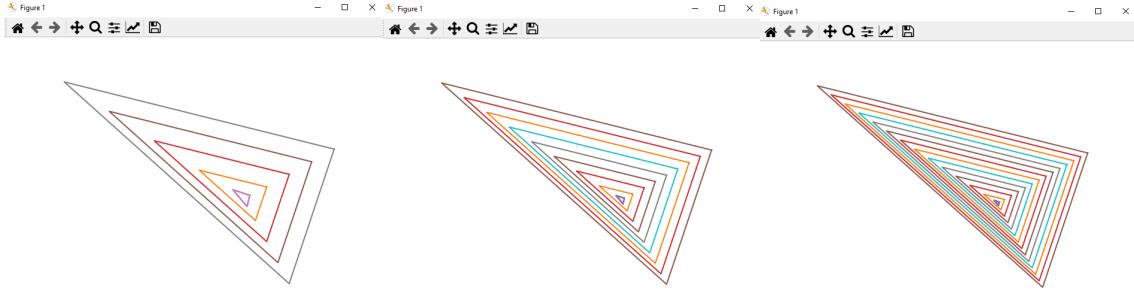


Figura 1.13: Animación de escalamiento  $S(4, 3)$

## 1.4 Rotación en el plano

Una rotación alrededor del origen con un ángulo  $\theta$  tiene como efecto que un punto  $P(x, y)$  se mapea a un punto  $P'(x', y')$  de manera que el punto inicial  $P$  y su imagen  $P'$  están a la misma distancia del origen, y el ángulo entre las líneas  $\overline{OP}$  y  $\overline{OP'}$  es  $\theta$ . Hay dos posibles puntos de la imagen que satisfacen estas propiedades dependiendo de si la rotación se realiza en el sentido de las agujas del reloj o en sentido contrario a las agujas del reloj.

Sea  $\theta \in R$ . Un mapa  $R : R^2 \rightarrow R^2$  de la forma  $R(r, \alpha) = (r, \alpha + \theta)$  donde los puntos se han expresado en coordenadas polares, se denomina rotación alrededor del origen por un ángulo  $\theta$ . Usando coordenadas polares, es una forma fácil de definir las rotaciones sobre el origen, sin embargo, no es conveniente desde el punto de vista computacional. Para derivar las ecuaciones de una rotación  $R$  en coordenadas cartesianas, utilizamos la correspondencia básica entre las coordenadas polares  $(r, \alpha)$  y las cartesianas  $(x, y)$  para un punto  $P$ :

$$x' = r \cos \alpha$$

$$y' = r \sin \alpha$$

Las ecuaciones para una rotación  $R$  alrededor del origen a través de un ángulo  $\theta$  son:

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin\theta + y \cos\theta$$

En particular, dicha rotación es una transformación lineal con matriz. La matriz de transformación es

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

Las coordenadas  $(x', y')$  pueden obtenerse de  $(x, y)$  por una multiplicación de matriz

$$P' \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta & x\sin\theta + y\cos\theta \end{pmatrix}$$

#### Ejercicio 1.4.1

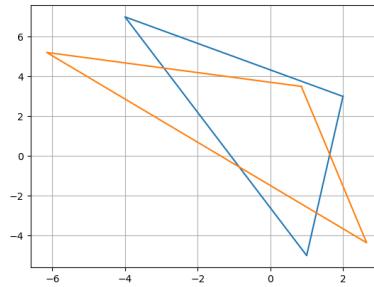
En la figura 1.14 se aplica una transformación de rotación donde  $\theta = 20^\circ$  a un triángulo compuesto por los puntos  $(2, 3)$ ,  $(-5, 4)$  y  $(7, -3)$ .

```

3  """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4  CU UAEM ZUMPANGO
5  UA: Gráfificación computacional
6  Tema: Rotación
7  Alumno: Keren Mitsue Ramírez Vergara
8  Profesor: Manuel Almeida Vázquez
9  Descripción: Aplicar la rotación a una figura
10 @author: KerenMitsue
11 """
12
13 import numpy as np
14 from numpy import pi, cos, sin
15 import matplotlib.pyplot as plt
16
17
18 P = np.array([ [2,3],[-4,7], [1,-5], [2,3] ], dtype=float)
19 gr = float(input("Ingresa el angulo de rotacion: "))
20 th = gr*(pi/180)
21 R = np.array([[cos(th), sin(th)], [-sin(th), cos(th)] ])
22
23
24 T = P@R
25 plt.plot(P[:,0], P[:,1])
26 plt.plot(T[:,0], T[:,1])
27 plt.grid()

```

(a) Implementación de código



(b) Rotación de un triángulo

Figura 1.14: Rotación a  $20^\circ$

## Capítulo 2

# Coordenadas Homogéneas

“La representación mediante coordenadas homogéneas de la localización de sólidos en un espacio dimensional se realiza a través de coordenadas de un espacio  $(n+1)$ . Es decir, un espacio  $n$  dimensional se encuentra representado en coordenadas homogéneas por  $(n+1)$  dimensiones, de tal forma que un punto  $P(x, y, z)$  vendrá representado por  $P(wx, wy, wz, w)$ , donde  $w$  tiene un valor arbitrario y representan un factor de escala”. [3]

Se puede definir formalmente las coordenadas homogéneas en términos de la relación de equivalencia. Una relación  $\sim$  sobre un conjunto  $S$  es una regla que determina si dos conjuntos están relacionados o no. Si el conjunto  $S_1$  está relacionado con el conjunto  $S_2$ , entonces se expresa:  $S_1 \sim S_2$ .

Cualquier relación  $\sim$  sobre el conjunto  $S$  cumple con las siguientes características:

1. Reflexiva, si  $s \sim s$  para todo  $s$  en el subconjunto  $S$ .
2. Simétrica, si siempre que  $s_1 \sim s_2$ , entonces  $s_2 \sim s_1$ .
3. Transitiva, si siempre que  $s_1 \sim s_2$  y  $s_2 \sim s_3$ , entonces  $s_1 \sim s_3$ .
4. Una relación de equivalencia si  $\sim$  es reflexiva, simétrica y transitiva.

## 2.1 Transformaciones en coordenadas homogéneas

A partir de la definición de las coordenadas homogéneas surge inmediatamente el concepto de matriz de transformación homogénea. Se define como matriz de transformación homogénea  $T$  a una matriz de dimensión  $n * n$  que representa la transformación de un vector de coordenadas homogéneas de un sistema de coordenadas a otro.

### 2.1.1 Traslación

La matriz de traslación homogénea para la traslación  $T(h, k)$  es

$$T(h, k) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix}$$

entonces:

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix} = \begin{pmatrix} x + h & y + k & 1 \end{pmatrix}$$

### 2.1.2 Reflexión

La matriz de reflexión homogénea para la reflexión  $R_x$  es  $\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ ,

y para la reflexión  $R_y$  es  $\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ , por lo que

$$R_x : P(x, y, 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x, -y, 1)$$

$$R_y : P(x, y, 1) \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (-x, y, 1)$$

### 2.1.3 Escalamiento

La matriz de escalamiento homogénea es

$$S(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

entonces

$$P(x, y, 1) \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = (s_x x, s_y y, 1)$$

### 2.1.4 Rotación

En coordenadas homogéneas la matriz de transformación para una rotación  $Rot(\theta)$  sobre el origen a través de un ángulo  $\theta$  es

$$Rot(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Donde un ángulo positivo denota una rotación en sentido contrario a las manecillas del reloj. Por lo tanto,

$$R : P(x, y, 1) \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, 1)$$

## 2.2 Concatenación de transformaciones

En coordenadas homogéneas, la concatenación de las transformaciones  $T_1$  y  $T_2$ , denotada  $T_1 \circ T_2$ , puede realizarse solo con multiplicaciones matriciales. La concatenación de transformaciones son una cadena de transformaciones que para calcularlas hacemos los productos de sus matrices de transformación. Como el producto de matrices es asociativo  $ABC = A(BC) = (AB)C$  y no commutativa  $AB \neq BA$

**Ejercicio 2.2.1**

A una figura compuesta por los puntos  $(2, 3), (3, 5), (4, 4), (6, 5), (7, 4), (4, 2), (2, 3)$  y  $(2, 7)$  se aplican las transformaciones de  $TSR_0$ ,  $STR_0$  y  $R_0ST$  para comprobar que el producto de las matrices no son conmutativas. Los valores aplicados para cada transformación son: traslación  $T(2, 3)$ , escalamiento  $S(2, 3)$  y rotación  $R_0(45^\circ)$

En la figura 2.1, se muestra una clase que contiene las matrices de transformaciones; cada función de la clase corresponde al producto de matrices de las transformaciones.

```

4 UA: Graficacion computacional
5 Tema: Transformadas Homogeneas
6 Alumno: Keren Mitsue Ramirez Vergara
7 Descripcion:
8     Clase que contiene las matrices de las transformaciones homogéneas
9
10 Created on Thu Aug 18 21:32:48 2022
11
12 @author: KerenMitsue
13 """
14
15 import numpy as np
16 from numpy import sin, cos, pi
17
18 class Homogeneas:
19
20     def translacion(self,P,h,k):
21         #Matriz de translacion
22         T = np.array([[1,0,0], [0,1,0], [h,k,1]])
23         return P@T
24
25     def reflexion(self,P,t='x'):
26         if t=='x':
27             R = np.array([[1,0,0], [0,-1,0], [0,0,1]])
28         else:
29             R = np.array([[ -1,0,0], [0,1,0], [0,0,1]])
30         return P@R
31
32     def escalamiento(self, P,sx,sy):
33         S = np.array([[sx,0,0], [0,sy,0], [0,0,1]])
34         if sx<0 or sy<0:
35             print("La figura ha colapsado")
36         return P@S
37
38     def rotacion(self,P,gr):
39         th = gr*(pi/180)
40         R = np.array([[cos(th),sin(th), 1], [-sin(th), cos(th), 1], [0,0,1]])
41         return P@R

```

Figura 2.1: Clase Homogéneas

En la figura 2.2, se muestra las concatenaciones de las transformaciones.

```

16 from Homogeneas import Homogeneas
17 import matplotlib.pyplot as plt
18 import numpy as np
19
20 homo = Homogeneas()
21 plt.figure()
22
23 P = np.array([[2,3,1], [3,5,1], [4,4,1], [6,5,1], [7,4,1], [4,2,1], [2,3,1], [2,7,1]])
24
25 #original
26 plt.subplot(221)
27 plt.plot(P[0:-1,0], P[0:-1,1])
28 plt.title('Imagen original')
29
30 #TSR0
31 plt.subplot(222)
32 T = homo.translacion(P,2,3)
33 S = homo.escalamiento(T,2,3)
34 Ro = homo.rotacion(S,45)
35 plt.plot(Ro[0:-1,0], Ro[0:-1,1])
36 plt.title('TSR0')
37
38 #STR0
39 plt.subplot(223)
40 S = homo.escalamiento(P,2,3)
41 T = homo.translacion(S,2,3)
42 Ro = homo.rotacion(T,45)
43 plt.plot(Ro[0:-1,0], Ro[0:-1,1])
44 plt.title('STR0')
45
46
47 #ROST
48 plt.subplot(224)
49 Ro = homo.rotacion(P,45)
50 S = homo.escalamiento(Ro,2,3)
51 T = homo.translacion(S,2,3)
52 plt.plot(T[0:-1,0], T[0:-1,1])
53 plt.title('ROST')
54

```

Figura 2.2: Concatenación de transformaciones

Como se ilustra en la figura 2.3, el orden de la concatenación de las transformaciones, si afecta el producto. Nótese que cada figura resultante es diferente.

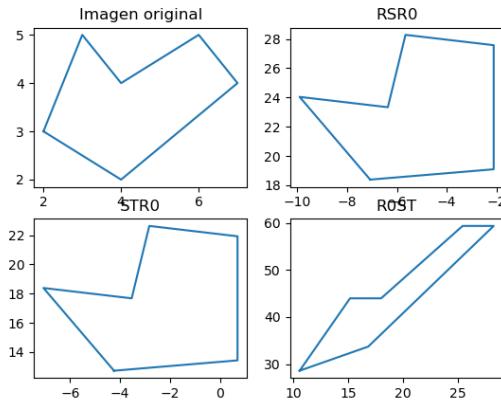


Figura 2.3: Matrices no comutativas

#### Ejercicio 2.2.2

Realizar una animación donde una figura compuesta por 8 puntos: (2, 3),(3, 5),(4,4),(6.5),(7, 4),(4, 2),(2, 3) y (2, 7) se expanda y contraiga hasta colapsar, aplicando concatenación de transformaciones.

```

3  ***UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4  CU UAEM ZUMPANGO
5  UA: Graficación computacional
6  Tema: Transformadas homogéneas
7  Alumno: Keren Mitsue Ramírez Vergara
8  Descripción: Expandir y contraer una figura hasta colapsar
9
10 Created on Fri Aug 19 00:40:52 2022
11
12 @author: KerenMitsue
13 ...
14 from Homogeneous import Homogeneous
15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 homo = Homogeneous()
19 P = np.array([ [2,3,1], [3,5,1], [4,4,1], [6,5,1], [7,4,1], [4,2,1], [2,3,1], [2,7,1] ])
20 for i in range(1,10):
21     plt.clf()
22     plt.axis([0,40,0,40])
23     S = np.array([ [1*i,0,0], [0,1*i,0], [0,0,1] ])
24     M = P*S
25     plt.plot(M[0:-1,0], M[0:-1,1])
26     plt.axis("off")
27     plt.pause(0.5)
28     print (M)
29
30 for j in range(10,-2, -1):
31     plt.clf()
32     plt.axis([0,40,0,40])
33     S = np.array([ [1*j,0,0], [0,1*j,0], [0,0,1] ])
34     P = P*S
35     M = P*S
36     plt.plot(M[0:-1,0], M[0:-1,1])
37     plt.axis("off")
38     plt.pause(0.5)
```

Figura 2.4: Código de animación de expandir y contraer

En la imagen 2.5 se muestran diferentes capturas de pantalla respecto a la expansión de la figura de 8 puntos. Una vez que termina la expansión, en la imagen 2.6 se muestra como la figura de 8 puntos se contrae hasta colapsar.

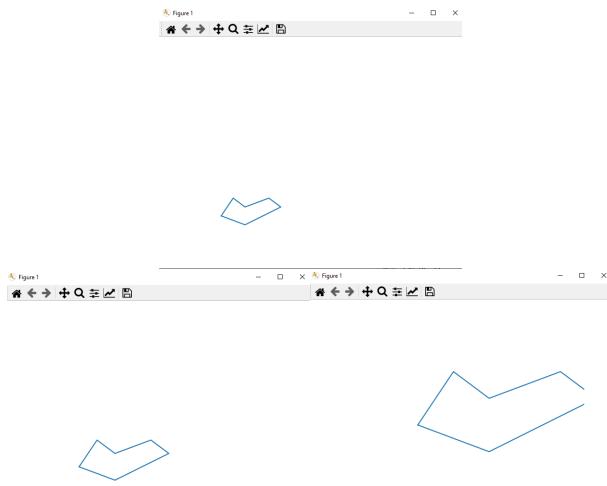


Figura 2.5: Figura expandida

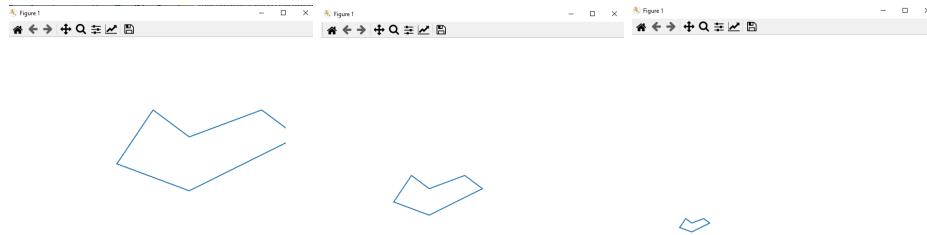


Figura 2.6: Figura contraida

### Ejercicio 2.2.3

Se realiza una animación a una figura compuesta por los puntos  $(2, 3), (3, 5), (4, 4), (6, 5), (7, 4), (4, 2), (2, 3)$  y  $(2, 7)$ , donde se muestre la concatenación de transformaciones con el objetivo de hacer girar  $360^\circ$  la figura. En la imagen 2.7, se muestra el código fuente de la animación.

```

2  """
3  #UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
4  CU UAEM ZUMPANGO
5  UA: Graficacion computacional
6  Tema: Transformadas Homogeneas
7  Alumno: Keren Mitsue Ramirez Vergara
8  Descripción:
9      Animación de figura que rota 360°
10 
11 
12 @author: KerenMitsue
13 """
14 
15 from Homogeneas import Homogeneas
16 import matplotlib.pyplot as plt
17 import numpy as np
18 
19 homo = Homogeneas()
20 plt.figure()
21 
22 P = np.array([ [2,3,1], [3,5,1], [4,4,1],[6,5,1],[7,4,1],[4,2,1],[2,3,1], [2,7,1]])
23 
24 plt.clf()
25 #Girar una figura 360
26 for i in range(1,360):
27     plt.clf()
28     plt.axis("off")
29     Ro = homo.rotacion(P,i)
30     plt.plot(Ro[0:-1,0], Ro[0:-1,1])
31     plt.title('Gira una figura a 360')
32     plt.pause(0.01)

```

Figura 2.7: Implementación en código de animación  $360^\circ$

A continuación se muestran imágenes de la figura girando a  $360^\circ$  (véase en la figura 2.8)

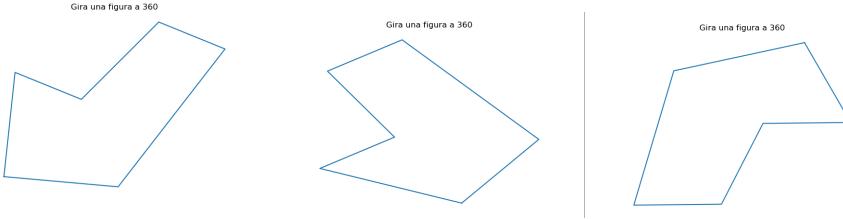


Figura 2.8: Animación de rotar figura a  $360^\circ$

#### Ejercicio 2.2.4

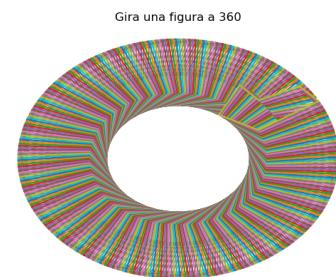
Realiza una animación a la figura compuesta por los puntos  $(2, 3), (3, 5), (4, 4), (6, 5), (7, 4), (4, 2), (2, 3)$  y  $(2, 7)$ , donde se muestre la concatenación de transformaciones con el objetivo de hacer girar  $360^\circ$  la figura. Muestra en pantalla cada iteración y observa el resultado.

```

1 # -*- coding: utf-8 -*-
2 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
3 UADE IMPANGO
4 UAC-Grafificación computacional
5 Tema: Transformadas Homogéneas
6 Alumno: Keren Mitsue Ramírez Vergara
7 Descripción:
8 Animación de figura que rota 360°
9
10 Created on Thu Aug 18 21:32:48 2022
11
12 @author: KerenMitsue
13 """
14
15 from Homogeneous import Homogeneous
16 import matplotlib.pyplot as plt
17 import numpy as np
18
19 homo = Homogeneous()
20 plt.figure()
21 P = np.array([[2,3,1], [3,5,1], [4,4,1],[6,5,1],[7,4,1],[4,2,1],[2,3,1], [2,7,1]])
22
23 #Girar una figura 360
24 for i in range(1,360):
25     P = homo.rotacion(P,i)
26     plt.axis("off")
27     plt.plot(Ro[0:-1,0], Ro[0:-1,1])
28     plt.title('Gira una figura a 360°')
29     plt.pause(0.01)
30
31

```

(a) Implementación de código



(b) Animación de rotación

Figura 2.9: Figura rotada  $360^\circ$

#### Ejercicio 2.2.5

Consideremos un brazo manipulador robótico plano 2R que consta de dos eslabones. El primer eslabón está unido a la base mediante una junta revolutiva J1. Una junta revolutiva permite que el eslabón gire alrededor de un punto. El segundo eslabón está unido al segundo eslabón está unido al primero mediante una segunda junta de revolución J2. La mano del robot o el efecto final está unido al segundo eslabón. La posición y orientación de la mano robótica controla la posición y la orientación de la mano del robot se controlan girando los eslabones alrededor de las dos articulaciones. Implementar en Python la animación del movimiento de dicho brazo manipulador.

```
1 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
```

```
2 CU UAEM ZUMPANGO
```

```
3 UA: Graficación computacional
```

```
4 Tema: Transformaciones homogéneas
```

```
5 Alumno: Keren Mitsue Ramírez Vergara
```

```
6 Descripción:
```

```
7 Animación de brazo manipulador
```

```
8 Created on Sat Sep 10 18:04:49 2022
```

```
9
```

```
10 @author: KerenMitsue
```

```
11 """
```

```
12 from matplotlib import pyplot as plt
```

```
13 import numpy as np
```

```
14 B1 = np.array([[0,0,1],[0, 0.28,1],[7, 0.11, [0, 0,1]])
```

```
15 B2 = np.array([[7, 0,1],[7, 0.28,1],[11, 0.11,[7, 0,1]])
```

```
16 M = np.array([[11,-0.25,1],[12,-0.25,1],[12,0,1],[11.5,0,1],[11.5,0.25,1],[12,0,25,1],
```

```
17 [12,0.5,1],[11,0.5,1],[11,-0.25,1]])
```

```
18
```

```
19 #Figura de robot
```

```
20 fig = plt.figure()
```

```
21 ax = fig.add_subplot()
```

```
22
```

```
23 #Primer brazo
```

```
24 for i in range(20):
```

```
25 plt.cla()
```

```
26 ax.axis([-0.5,15,-0.5,15])
```

```
27 th = 0.01*np.pi*i
```

```
28 R = np.array([[np.cos(th),np.sin(th),0], [-np.sin(th), np.cos(th),0], [0,0,1]])
```

```
29
```

```
30 T1 = B1@R
```

```
31 T2 = T1@R
```

```
32 M2 = M@R
```

```
33 ax.plot(T1[:,0],T1[:,1], T2[:,0],T2[:,1],M2[:,0],M2[:,1] )
```

```
34 plt.pause(0.2)
```

(a) Parte 1

```
36 T2 = np.array([[-6.04838710e-03, -2.39026396e-03,1],[-1.49697581e-01, 2.27379459e-01,1],  
37 [ 3.13155242e+00, 2.46380476e+00,1], [ 3.27520161e+00, 2.24935302e+00,1],  
38 [-6.04838710e-03, -2.39026396e-03,1]])  
39  
40 T = np.array([ [1,0,0], [0,1,0], [-5.78956402, 3.93458364, 1]])  
41 W = M2@T  
42  
#Segundo brazo  
43 for i in range(20):  
44 plt.cla()  
45 ax.axis([-0.5,15,-0.5,15])  
46 th = 0.01*np.pi*i  
47 T1 = np.array([ [1,0,0], [0,1,0], [5.78956402, 3.93458364, 1]])  
48 R = np.array([[np.cos(th),np.sin(th),0], [-np.sin(th), np.cos(th),0], [0,0,1]])  
49 X1 = T1@R  
50 X2 = T2@R  
51 M = W@R@T  
52 ax.plot(T1[:,0],T1[:,1],X2[:,0],X2[:,1],M[:,0],M[:,1] )  
53 plt.pause(0.1)
```

(b) Parte 2

Figura 2.10: Implementación de brazo manipulador

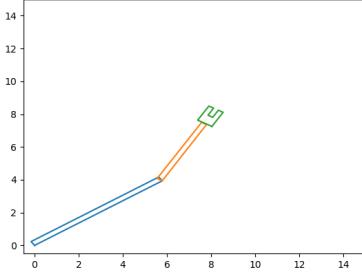
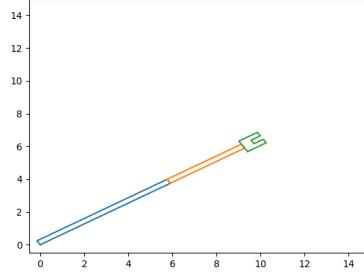
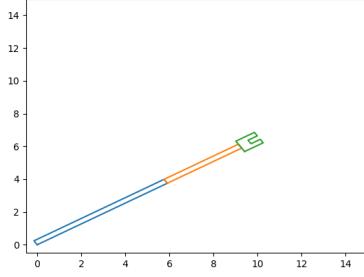


Figura 2.11: Brazo manipulador

# Capítulo 3

## Transformaciones inversas

La transformación de identidad  $L$  es una transformación que tiene el efecto de dejar todos los puntos del plano sin cambios, es decir, una transformación inversa regresa un conjunto de puntos a su estado inicial. La inversa de una transformación  $L$ , denominada  $L^{-1}$ , tiene el efecto de devolver las imágenes de la transformación  $L$  a sus puntos originales. Estas transformaciones pueden tener una definición más precisa en términos de concatenación de transformaciones.

La transformación identidad del plano es la transformación para  $L^{-1} * L = L * L^{-1} = I$ . Para las transformaciones no singulares se tiene que  $(A * B)^{-1} = B^{-1} * A^{-1}$ . La matriz de transformación es de  $3 \times 3$ , es decir, la matriz con valores de 1 en la diagonal principal y 0 en el resto.

### 3.1 Escalamiento

La matriz identidad para realizar un transformación inversa del escalamiento es:

$$S^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

### 3.2 Rotación

La matriz identidad para realizar un transformación inversa de rotación es:

$$R^{-1} = \begin{pmatrix} \cos(-\theta) & \sin(-\theta) & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

### 3.3 Traslación

La matriz identidad para realizar un transformación inversa de traslación es:

$$T = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ -h & -k & 1 \end{pmatrix}$$

#### Ejercicio 3.1

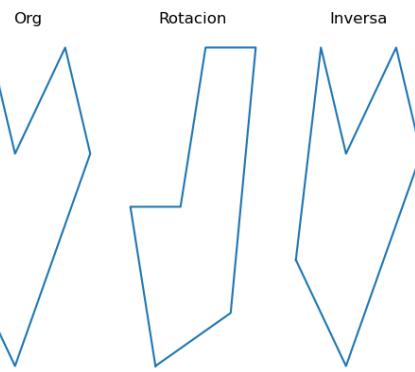
Se aplica una transformación inversa de rotación a una figura para demostrar que las transformaciones inversas permiten regresar a un conjunto de puntos a su estado inicial. En la figura se muestra la implementación en código y en la figura \* se muestra la transformación de rotación y su inversa.

```

14 import matplotlib.pyplot as plt
15 import numpy as np
16 from numpy import sin, cos, pi
17
18 P = np.array([[2,3,1], [3,5,1], [4,4,1],[6,5,1],[7,4,1],[4,2,1],[2,3,1], [2,7,1]])
19
20 plt.figure()
21 plt.subplot(131)
22 plt.plot(P[0:-1,0], P[0:-1,1])
23 plt.axis('off')
24 plt.title("Org")
25
26
27 gr = 45
28 th = gr*(pi/180)
29 R = np.array([[cos(th),sin(th), 1], [-sin(th), cos(th), 1], [0,0,1]])
30 M = P@R
31 plt.subplot(132)
32 plt.plot(M[0:-1,0], M[0:-1,1])
33 plt.axis('off')
34 plt.title("Rotacion")
35
36
37 gr = 45
38 th = gr*(pi/180)
39 Rinv= np.array([[cos(-th),sin(-th), 1], [-sin(-th), cos(-th), 1], [0,0,1]])
40 Minv = M@Rinv
41 plt.subplot(133)
42 plt.axis('off')
43 plt.plot(M[0:-1,0], M[0:-1,1])
44 plt.title("Inversa")
45

```

(a) Implementación de código



(b) Transformación inversa de rotación

Figura 3.1: Transformación inversa

## Capítulo 4

# Interpolación lineal

“La interpolación lineal permite estimar una función  $f(x)$  para un  $x$  arbitrario, a partir de la construcción de una curva o superficie que une los puntos donde se han realizado mediciones y cuyo valor se conoce. Se asume que el punto arbitrario  $x$  se encuentra dentro de los límites de los puntos de medición, en caso contrario se llamaría extrapolación.”[4]

La interpolación lineal es el método más simple utilizado actualmente, ya que lo usan los programas de generación de gráficas, donde se interpola con líneas rectas una serie de puntos que el usuario quiere graficar.

La interpolación consta de conectar dos puntos dados en  $x_i$ , es decir  $(x_0, y_0)$  y  $(x_1, y_1)$ . La función interpolante es una línea recta entre dos puntos, que siguen la siguiente ecuación

$$\frac{y-x_0}{y_1-y_0} = \frac{x-x_0}{x_1-x_0}$$

Como se conoce únicamente el valor conocido de  $y$ , se despeja:

$$y = y_0 + (x - x_0) \frac{y_1 - y_0}{x_1 - x_0}$$

Si se tienen más de dos puntos para la interpolación, es decir  $N > 2$ , con puntos  $x_0, x_1, x_2, \dots, x_N$ , simplemente se concatena la interpolación lineal entre pares de puntos continuos.

Dado un parámetro  $t \in [0, 1]$ , para obtener la interpolación de dos puntos  $P(x, y)$  y  $Q(x_1, y_1)$  se realiza

$$P(x', y') = (1 - t)P + tQ = (1 - t)P(x, y) + tQ(x_1, y_1)$$

### Ejercicio 4.1

Realizar una animación sobre una figura en forma de casa, simulando un desplazamiento vertical. Considere utilizar utilizar la interpolación en los puntos y el parámetro  $t \in [0, 1]$ .

En la figura 4.2a se muestra el código fuente de la animación y en la figura 4.2b se muestra como se realiza el desplazamiento.

```

17 import numpy as np
18 import matplotlib.pyplot as plt
19 P = np.array([[0.0],[0.4],[1.5,5],[3.4],[3.0],[2.0],[2.1],[1.1],[1.0],[0.0]])
21
22 B = np.zeros((10,2))
23
24 h = 0
25 k = 15
26
27 B[:, 0] = P[:, 0] + h
28 B[:, 1] = P[:, 1] + k
29
30 T = np.zeros((10,2))
31
32 for i in range(20):
33     plt.clf()
34     plt.axis([0,20,0,20])
35
36     t = float(i/20);
37
38     T[:, 0] = (1-t) * P[:, 0] + t * B[:, 0]
39     T[:, 1] = (1-t) * P[:, 1] + t * B[:, 1]
40
41     plt.plot(T[:,0], T[:,1])
42     plt.axis('off')
43     plt.pause(0.5)

```

Figura 4.1: Implementación de código



(a) Interpolación vertical

Figura 4.2: Animación de interpolación vertical

#### Ejercicio 4.2

Realizar una animación sobre una figura en forma de casa, simulando un desplazamiento horizontalmente. Considere utilizar utilizar la interpolación en los puntos y el parámetro  $t \in [0, 1]$ .

En la figura 4.4a se muestra el código fuente de la animación y en la figura 4.4b se muestra como se realiza el desplazamiento.

```

13 # Horizontal
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 P = np.array([[0,0],[0,4],[1.5,5],[3,4],[3,0],[2,0],[2,1],[1,1],[1,0],[0,0]])
19
20 B = np.zeros((10,2))
21
22 h = 10
23 k = 0
24
25 B[:, 0] = P[:, 0] + h
26 B[:, 1] = P[:, 1] + k
27
28 T = np.zeros((10,2))
29
30 for i in range(50):
31     plt.clf()
32     plt.axis([0,20,0,20])
33
34     t = float(i/50);
35
36     T[:, 0] = (1-t) * P[:, 0] + t * B[:, 0]
37     T[:, 1] = (1-t) * P[:, 1] + t * B[:, 1]
38
39     plt.plot(T[:,0], T[:,1])
40     plt.axis('off')
41     plt.pause(0.1)

```

Figura 4.3: Implementación de código



(a) Interpolación horizontal

Figura 4.4: Animación de interpolación horizontal

### Ejercicio 4.3

Realizar una animación sobre una figura en forma de casa, simulando un desplazamiento en diagonal. Considere utilizar utilizar la interpolación en los puntos y el parámetro  $t \in [0, 1]$ .

En la figura 4.6a se muestra el código fuente de la animación y en la figura 4.6b se muestra como se realiza el desplazamiento.

```

13 import numpy as np
14 import matplotlib.pyplot as plt
15
16 P = np.array([[0,0],[0,4],[1.5,5],[3,4],[3,0],[2,0],[2,1],[1,1],[1,0],[0,0]])
17
18 B = np.zeros((10,2))
19
20 h = 10
21 k = 15
22
23 B[:, 0] = P[:, 0] + h
24 B[:, 1] = P[:, 1] + k
25
26 T = np.zeros((10,2))
27
28 for i in range(20):
29     plt.clf()
30     plt.axis([0,20,0,20])
31
32     t = float(i/20);
33
34     T[:, 0] = (1-t) * P[:, 0] + t * B[:, 0]
35     T[:, 1] = (1-t) * P[:, 1] + t * B[:, 1]
36
37     plt.plot(T[:,0], T[:,1])
38     plt.axis('off')
39     plt.pause(0.5)

```

Figura 4.5: Implementación de código



(a) Interpolación horizontal

Figura 4.6: Animación de interpolación horizontal

#### Ejercicio 4.4

Implementar una figura cualquiera, que simule el movimiento de una función seno implementando la interpolación lineal, mostrar esta implementación simulando una animación de la función

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17
18 P = np.array([-20., -0.9],[-20., -0.55],[-6., -0.55],[-6., -0.8],
19 [-3., -0.8],[-3., -0.6],[2., -0.6],[4., -0.65],
20 [4., -0.9],[1., -0.9],[1., -0.95],[0., -1.],
21 [-1., -1.],[2., -0.95],[-2., -0.9],[-3., -0.9],
22 [-3., -0.85],[-6., -0.85],[-6., -0.9],
23 [-8., -0.95],[-8., -0.95],[-9., -1.],[-10., -1.],
24 [-11., -0.95],[-11., -0.95],[-15., -0.9],
25 [-15., -0.95],[-16., -1.],[-17., -1.],
26 [-18., -0.95],[-18., -0.9],[-20., -0.9]])
27
28 T = np.zeros((32,2))
29 x = np.arange(0,4*np.pi,0.01)
30
31 for i in range(500):
32     plt.clf()
33
34     plt.axis([-20,100,-2,1.5])
35
36     t = (i*np.pi)/180
37
38     T[:, 0] = t*8 + P[:, 0]
39     T[:, 1] = P[:, 1] + np.sin(t)
40
41     plt.plot(T[:,0], T[:,1])
42     plt.axis('off')
43     plt.pause(0.000001)
44
45

```

Figura 4.7: Implementación de código



Figura 4.8: Animación de carrito

# Capítulo 5

## Morphing

El morphing es un efecto especial que consiste en una transformación de una imagen o figura gradualmente en otra. Esta pensado de forma que el observador no aprecia en que momento una imagen cambio en la otra. Es un cambio suave entre las dos aún cuando este cambio se hace en pocos segundos.

Antiguamente los morphings sólo podían realizarse en un par de grandes empresas de efectos especiales (como la Industrial Light & Magic de George Lucas), debido a la necesidad de potentes ordenadores. Sin embargo, actualmente, con la mejora de las prestaciones de los PCs y la simplificación de los métodos utilizados para producir el morphing, cualquier persona ya puede tener acceso a los medios técnicos necesarios y, aunque no podrá llegar a producir efectos tan espectaculares como los anteriores, sí será capaz de crear morphings de notable valor.

### Ejercicio 5.1

Empleando el efecto Morphing, elaborar una programa capaz de transformar una casa compuesta por los siguientes puntos, a una 'T' conformada por los siguientes puntos (0, 0), (0, 4), (1.5, 5), (3, 4), (3, 0), (2, 0), (2, 1), (1, 1) y (1, 0) transformalos en una letra 'T' con las coordenadas (8, 2), (8, 4), (6, 4), (6, 5), (8.5, 5), (11, 5), (11, 4), (9, 4) y (9, 2)

```
9 import numpy as np
10 import matplotlib.pyplot as plt
11
12 A = np.array([[0,0],[0,4],[1.5,5],[3,4],[3,0],[2,0],[2,1],[1,1],[1,0],[0,0]])
13 B = np.array([[8,2],[8,4],[6,4],[6,5],[8.5,5],[11,5],[9,4],[9,2],[8,2]])
14
15
16 T = np.zeros((10,2))
17
18 plt.axis([-1,20,-1,20])
19
20 for i in range(11):
21     plt.axis([0,12,0,12])
22
23     t = float(i/10);
24
25     T[:, 0] = (1-t) * A[:, 0] + t * B[:, 0]
26     T[:, 1] = (1-t) * A[:, 1] + t * B[:, 1]
27
28     plt.plot(T[:,0], T[:,1])
29     plt.axis('off')
30     plt.pause(1)
31
32 plt.clf()
```

Figura 5.1: Implementación de código



Figura 5.2: Morphing de casita a letra 'T'

### Ejercicio 5.2

Aplicar morphing a una figura de más de 6 puntos, implementar el código y los resultados.

```

2  """UNIVERSIDAD AUTONOMA DEL ESTADO DE MEXICO
3  CU UAEM ZUMPANGO
4  UA: Graficación computacional
5  Tema: Morphing
6  Alumno: Keren Mitsue Ramirez Vergara
7  Profesor: Manuel Almeida Vazquez
8  Descripción: Aplicar Morphing a una figura de más de 6 puntos
9
10 Created on Sat Aug 27 19:28:59 2022
11
12 @author: KerenMitsue
13
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 A = np.array([
19     [5,8],[7,10],[7,8],[9,8],[7,6],[9,4],[7,4],[7,2],[5,4],[3,2],
20     [3,4],[1,4],[3,6],[1,8],[3,8],[3,10],[5,8]])
21 B = np.array([
22     [16,28],[18,26],[19,28],[20,26],[21,28],[22,26],[23,28],
23     [24,26],[24,23],[22,22],[21,22],[21,16],[19,16],[19,22],
24     [18,22],[16,23],[16,28]])
25
26 T = np.zeros((17,2))
27
28 for i in range(11):
29     plt.clf()
30     plt.axis([0,30,0,30])
31     t = float(i/10);
32     T[:, 0] = (1-t) * A[:, 0] + t * B[:, 0]
33     T[:, 1] = (1-t) * A[:, 1] + t * B[:, 1]
34     plt.plot(T[:,0], T[:,1])
35     plt.axis('off')
36     plt.pause(1)

```

Figura 5.3: Implementación de código



Figura 5.4: Morphing de una figura de más de 6 puntos

### Ejercicio 5.3

Aplicar morphing a una figura entre 40-60 puntos, implementar el código y los resultados.

```

2  """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
3  CU UAEM ZUMPANGO
4  UA: Graficación computacional
5  Tema: Morphing
6  Alumno: Keren Mitsue Ramírez Vergara
7  Profesor: Manuel Almeida Vazquez
8  Descripción: Aplicar Morphing a una figura de entre 40-60 puntos
9  Created on Sat Aug 27 19:28:59 2022
10
11 @author: KerenMitsue
12 """
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16
17
18 P = np.array([[4.5, 10], [3.9, 5], [1.5, 8.5], [0.5, 7], [0.5, 5], [1.5, 6.5],
19 [1.5, 7], [3.9, 1], [4.5, 10], [4.7], [1.5, 6.5], [0.5, 5], [3.5, 5], 36
20 [0.5], [0.5, 4.5], [4, 4], [4.7], [4.5, 7], [5.7], [5, 4], [5, 1], [4, 0], 37
21 [2.5, -0.5], [1.5, 0.5], [1.5, 1], [2, 1.5], [2.5, 1.5], [3, 1], [3.5, 1], 38
22 [4, 1.5], [4.7], [4.5, 7], [4.5, 10], [6, 9.5], [7.5, 8.5], [8.5, 7], 39
23 [9, 5.5], [8.5, 6], [7.5, 6.5], [7.5, 7], [6, 9], [4.5, 10], [4.5, 7], 40
24 [5, 7], [7.5, 6.5], [8, 5.6], [9, 5.5], [9, 5], [8, 4.5], [5, 4] ] ) 41
25
26 B = np.array([[18.5, 29], [19, 29], [19, 5, 28.5], [22, 22], [23, 21], [23, 19.5], 42
27 [22, 18.5], [20, 5, 18], [19, 5, 18], [20, 5, 15], [21, 16.5], [21, 15], 43
28 [21, 5, 16], [21, 5, 15], [22, 5, 15.5], [22, 15], [23, 15], [21, 14], 44
29 [17, 14], [15, 15], [16, 15], [15, 16], [16, 15.5], [16, 16.5], 45
30 [16, 5, 15], [16, 5, 16.5], [17, 15], [18, 18], [17, 5, 19.5], [18, 5, 20], 46
31 [19, 20], [20, 19.5], [19, 5, 18], [20, 5, 18.5], [21, 19], [22, 5, 20], 47
32 [21.5, 5.5], [21, 21.5], [16, 21], [15, 20], [16, 19], [18, 18], [17, 18], 47
33 [15, 5, 18.5], [14, 5, 19.5], [14, 5, 21], [15, 5, 22], [18, 20.5], 48
34 [18, 5, 29], [19, 29]] ) 49

```

Figura 5.5: Implementación de código



Figura 5.6: Morphing de una figura entre 40-60 puntos

#### Ejercicio 5.4

Aplicar morphing en 3D a una figura entre 40-60 puntos, implementar el código y los resultados.

```

1  # -*- coding: utf-8 -*-
2  """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
3  CU UAEM ZUMPANGO
4  UA: Graficación computacional
5  Tema: Morphing
6  Alumno: Keren Mitsue Ramírez Vergara
7  Profesor: Manuel Almeida Vazquez
8  Descripción: Aplicar Morphing a una figura de entre 40-60 puntos
9  en 3D
10 Created on Sat Aug 27 19:28:59 2022
11 @author: KerenMitsue
12 """
13
14 from matplotlib import pyplot as plt
15 import numpy as np
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 B1 = np.array([
21 [-1.33, 3.24, 0], [-2.42, 0, 0], [1.413, -0.53, 2.91], [-2.45, -2.91, 4.7], 54
22 [0.3, -3.84, 2.91], [0, 32, -2.04, 0], [0.3, -3.84, 2.91], [3, 07, -2.97, 4.7], 55
23 [4.81, -0.64, 2.91], [3.11, -0.06, 0], [4.81, -0.64, 2.91], [4.84, 2.27, 4.7], 56
24 [3, 16, 0.99, 2.91], [2, 08, 0, 0], [3, 16, 0.99, 2.91], [3, 07, 4.7, 4.7], 57
25 [-2.37, 4.12, 2.91], [-2.37, 3.24, 0], [-2.37, 4.12, 2.91], [-4, 1.2, 2.97, 4.7], 58
26 [-4.13, -0.53, 2.91], [-2.42, 0, 0], [-4.13, -0.53, 2.91], [-2.45, -2.91, 4.7], 59
27 [-1.38, -1.47, 7.61], [-1.24, -1.47, 8.761], [-1.41, 2.37, 4.7], [-2.4, 1.8, 7.61], 60
28 [0.39, 3.77, 7.61], [0, 41.5, 5.57, 4.7], [0.39, 3.77, 7.61], [3.13, 1.75, 7.61], 61
29 [4.84, 2.27, 4.7], [1.13, 1.75, 7.61], [0.03, -1.51, 7.61], [3, 07, -2.97, 4.7], 62
30 [2.03, -1.51, 7.61], [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], 63
31 [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], 64
32 [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], 65
33 [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61], 66
34 [-1.38, -1.47, 7.61], [-1.38, -1.47, 7.61] ] ) 67

```

Figura 5.7: Implementación de código

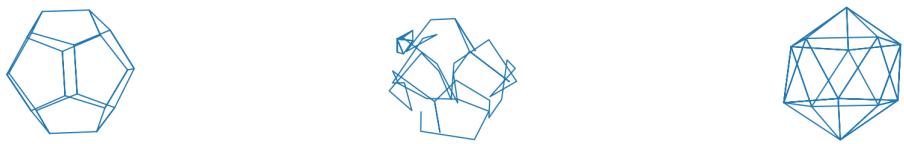


Figura 5.8: Morphing en 2D

# Capítulo 6

## Transformaciones en 3D

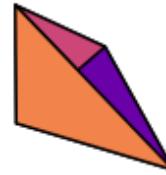
En computación, se utilizan gráficos en 3D para crear animaciones, gráficos, películas, juegos, realidad virtual, diseño, etc. El proceso de creación de los gráficos comienza con un conjunto de fórmulas matemáticas que describen objetos poligonales, tonalidades, texturas, sombras, profundidad, etc. Para la virtualización en 3D se necesita ejes coordenados ( $x, y, z$ ), además de considerar que los objetos están ubicados en un sistema universal de referencia.

### Ejercicio 6.0

Realizar un tetraedro, coloreando cada uno de sus lados. Mostrar la figura en 3D

```
14 import matplotlib as mpl
15 import matplotlib.pyplot as plt
16 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
17 import numpy as np
18
19
20 def patch(ax, x, y, z, v, vmin=0, vmax=100, cmap_name='plasma'):
21     cmap = mpl.cm.get_cmap(cmap_name)
22     c = np.array(mpl.colors.Normalize(vmin, vmax))(v) # Normalize value and get color
23     pc = Poly3DCollection([list(zip(x,y,z))]) # Create PolyCollection from vertices
24     pc.set_facecolor(c) # Set facecolor to mapped value
25     pc.set_edgecolor('k') # Set edgecolor to black
26     ax.add_collection3d(pc)
27     return pc
28
29
30 fig = plt.figure()
31 ax = fig.add_subplot(111, projection='3d')
32 ax.set_xlim(-4,6)
33 ax.set ylim(-4,6)
34 ax.set_zlim(-4,6)
35 ax.axis('off')
36
37
38 P = np.array([[0,0,4],[0,4,0],[4,0,0],[0,0,4],[0,0,0],[4,0,0],[0,0,0],[0,4,0]])
39
40 patch(ax, P[0], P[1], P[2], 50)
41 patch(ax, P[2], P[3], P[4], 20)
42 patch(ax, P[5], P[6], P[7], 70)
43 patch(ax, P[6], P[7], P[0], 100)
44
45 plt.show()
```

(a) Implementación de código



(b) Figura en 3D

Figura 6.1: Tetraedro 3D

### 6.1 Coordenadas homogéneas

Las coordenadas homogéneas de un punto  $(x, y, z)$  en  $R_3$  son una 4-tupla:

$$(X, Y, Z, W) \text{ donde } W \neq 0 \text{ y } x = X/W, y = Y/W \text{ Y } z = Z/W$$

Toda transformación proyectiva de  $R_3$  es una transformación afín o un compuesto de una transformación afín y una perspectiva única. Utilizando coordenadas homogéneas, podemos expresar dicho mapa como un producto de matrices  $4 \times 4$  que corresponden a los mapas en esa composición o mediante una matriz única de  $4 \times 4$ . Un punto de la forma  $(X, Y, Z, 0)$  no corresponde a un punto cartesiano, sino que representa un punto en el infinito en la dirección del vector tridimensional  $(X, Y, Z)$ . El conjunto de todas las coordenadas homogéneas se denomina **espacio proyectivo**.

## 6.2 Transformaciones en el espacio

En las transformaciones en el espacio 3D habitualmente se utilizan sistemas de coordenadas homogéneas, ya que regularmente aconsejable la composición matricial. Al igual que en el plano, las composiciones de las transformaciones tridimensionales se realizan mediante la multiplicación de matrices de transformación.

### 6.2.1 Traslación

Dentro de un espacio de referencia los objetos pueden modificar su tamaño relativo en uno, dos, o los tres ejes. El sistema de referencia homogénea tendrá 4 dimensiones, por lo que, para un punto  $P(x, y, z, 1)$ , la traslación del punto  $P$  a  $P'$  es

$$T(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ h & k & l & 1 \end{pmatrix} = (x + h, y + k, z + l, 1)$$

Cuando  $s_x = s_y = s_z$  el cambio de escala es **uniforme**; en cualquier otro caso el cambio de escala será no uniforme.

#### Ejercicio 6.2.1.1

Realizar una transformación de traslación de 12 unidades en el eje X a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje X ( $T_x = 12$ ).

```

12 from matplotlib import pyplot as plt
13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15 import matplotlib as mpl
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 # Puntos de la figura
21 v = np.array([[-1, -6.13, 4.9, 11, -1, -4.2, 0, 11, [2, 2, 0, 1], [3.85, 7.71, 0, 1],
22 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 11]])
23
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 # Matriz de transformación
27 h = 12
28 k = 0
29 l = 0
30 T = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [h, k, l, 1]])
31
32 M = v@T
33
34 # Lados del polígono
35 verts = [[v[0, 0:-1], v[4, 0:-1], v[5, 0:-1]], [v[4, 0:-1], v[5, 0:-1], v[6, 0:-1]],
36 [v[6, 0:-1], v[1, 0:-1], v[2, 0:-1]], [v[6, 0:-1], v[2, 0:-1], v[3, 0:-1]],
37 [v[3, 0:-1], v[1, 0:-1], v[4, 0:-1]], [v[6, 0:-1], v[1, 0:-1], v[5, 0:-1]],
38 [v[1, 0:-1], v[2, 0:-1], v[3, 0:-1], v[4, 0:-1], v[5, 0:-1]]]
39
40 # Color sides of figure original
41 cmap = mpl.cm.get_cmap('viridis')
42 for i in range(len(verts)):
43     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
44     coll = Poly3DCollection(verts[i], edgecolors='w')
45     coll.set_facecolor(c)
46     coll.set_clim(vmin=0, vmax=100)
47     ax.add_collection(coll)
48
49 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
50 verts = [[M[0, 0:-1], M[4, 0:-1], M[5, 0:-1], M[4, 0:-1], M[5, 0:-1], M[6, 0:-1],
51 [M[0, 0:-1], M[1, 0:-1], M[2, 0:-1], M[0, 0:-1], M[2, 0:-1], M[3, 0:-1]],
52 [M[0, 0:-1], M[3, 0:-1], M[4, 0:-1], M[0, 0:-1], M[1, 0:-1], M[5, 0:-1]],
53 [M[1, 0:-1], M[2, 0:-1], M[3, 0:-1], M[4, 0:-1], M[5, 0:-1]]]
54
55
56 cmap = mpl.cm.get_cmap('plasma')
57 for i in range(len(verts)):
58     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
59     coll = Poly3DCollection(verts[i], edgecolors='w')
60     coll.set_facecolor(c)
61     coll.set_clim(vmin=0, vmax=100)
62     ax.axis("off")
63     ax.add_collection(coll)
64
65 plt.show()

```

Figura 6.2: Implementación de código

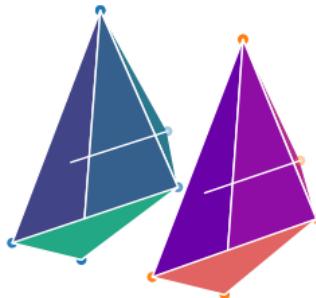


Figura 6.3: Translación en eje X

#### Ejercicio 6.2.1.2

Realizar una transformación de traslación de 12 unidades en el eje Y a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje X ( $T_y = 12$ ).

```

13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15 import matplotlib as mpl
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 # Puntos de la figura
21 v = np.array([[1, 6.13, 4.9], [-4.2, 0.1], [2, 2, 0.1], [3.85, 7.71, 0.1],
22 [-1, 11.23, 0.1], [-5.85, 7.71, 0.1], [-1, 6.13, 4.9, 1]])
23
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 #Matriz de transformacion
27 h = 0
28 k = 12
29 l = 0
30 T = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [h, k, l, 1]])
31
32 M = v@T
33 # Lista de lados del poligono
34 verts = [[v[0,0:-1],v[4,0:-1],v[5,0:-1]], [v[4,0:-1],v[5,0:-1],v[6,0:-1]],
35 [v[0,0:-1],v[1,0:-1],v[2,0:-1]], [v[0,0:-1],v[2,0:-1],v[3,0:-1]],
36 [v[0,0:-1],v[3,0:-1],v[4,0:-1]], [v[0,0:-1],v[1,0:-1],v[5,0:-1]],
37 [v[1,0:-1],v[2,0:-1],v[3,0:-1],v[4,0:-1],v[5,0:-1]]]
38
39 #Color sides of figure original
40 cmap = mpl.cm.get_cmap('viridis')
41 for i in range(len(verts)):
42     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
43     coll = Poly3DCollection(verts[i],edgecolors='w')
44     coll.set_facecolor(c)
45     coll.set_clim(vmin=0, vmax=100)
46     ax.add_collection(coll)
47
48 plt.show()
49 #Color sides of figure original
50 cmap = mpl.cm.get_cmap('viridis')
51 for i in range(len(verts)):
52     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
53     coll = Poly3DCollection(verts[i],edgecolors='w')
54     coll.set_facecolor(c)
55     coll.set_clim(vmin=0, vmax=100)
56     ax.axis("off")
57     ax.add_collection(coll)
58
59 plt.show()
60
61
62
63
64
65 plt.show()

```

Figura 6.4: Implementación de código

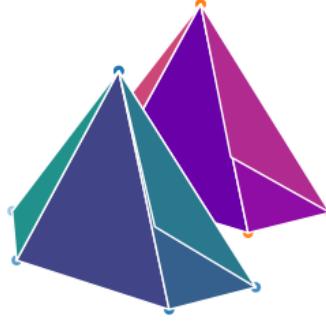


Figura 6.5: Translación en eje Y

### Ejercicio 6.2.1.3

Realizar una transformación de traslación de 12 unidades en el eje Z a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje X ( $T_z = 12$ ).

```

12 from matplotlib import pyplot as plt
13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15 import matplotlib as mpl
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 # Puntos de la figura
21 v = np.array([[1, 6.13, 4.9], [-4.2, 0.1], [2, 2, 0.1], [3.85, 7.71, 0.1],
22 [-1, 11.23, 0.1], [-5.85, 7.71, 0.1], [-1, 6.13, 4.9, 1]])
23
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 #Matriz de transformacion
27 h = 0
28 k = 0
29 l = 12
30 T = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [h, k, l, 1]])
31
32 M = v@T
33 # Lista de lados del poligono
34 verts = [[v[0,0:-1],v[4,0:-1],v[5,0:-1]], [v[4,0:-1],v[5,0:-1],v[6,0:-1]],
35 [v[0,0:-1],v[1,0:-1],v[2,0:-1]], [v[0,0:-1],v[2,0:-1],v[3,0:-1]],
36 [v[0,0:-1],v[3,0:-1],v[4,0:-1]], [v[0,0:-1],v[1,0:-1],v[5,0:-1]],
37 [v[1,0:-1],v[2,0:-1],v[3,0:-1],v[4,0:-1],v[5,0:-1]]]
38
39 #Color sides of figure original
40 cmap = mpl.cm.get_cmap('viridis')
41 for i in range(len(verts)):
42     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
43     coll = Poly3DCollection(verts[i],edgecolors='w')
44     coll.set_facecolor(c)
45     coll.set_clim(vmin=0, vmax=100)
46     ax.add_collection(coll)
47
48 plt.show()
49 #Color sides of figure original
50 cmap = mpl.cm.get_cmap('viridis')
51 for i in range(len(verts)):
52     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
53     coll = Poly3DCollection(verts[i],edgecolors='w')
54     coll.set_facecolor(c)
55     coll.set_clim(vmin=0, vmax=100)
56     ax.axis("off")
57     ax.add_collection(coll)
58
59 plt.show()
60
61
62
63
64
65 plt.show()

```

Figura 6.6: Implementación de código

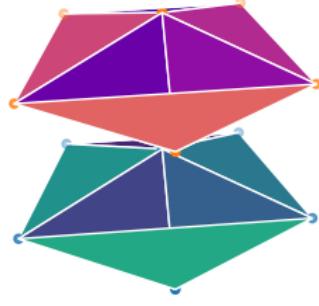


Figura 6.7: Translación en eje Z

#### Ejercicio 6.2.1.4

Realizar una transformación de translación en un prisma pentagonal. La translación para cada eje es  $T_x = 3$ ,  $T_y = 2$  y  $T_z = 5$ .

```

1  """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
2  CU UAEM ZUMPANGO
3  UA: Graficacion computacional
4  Tema: Transformaciones en 3D
5  Alumno: Keren Mitsue Ramírez Vergara
6  Profesor: Manuel Almeida Vazquez
7  Descripción: Translación de una figura en 3D (prisma pentagonal)
8  Created on Sat Aug 27 19:30:42 2022
9
10 #author: KerenMitsue
11 """
12 from matplotlib import pyplot as plt
13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15 import matplotlib as mpl
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 # Puntos de la figura
21 v = np.array([-1, 6.13, 4.9, 1], [-4.2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1],
22 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1])
23
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 #Matriz de transformacion
27 h = 3
28 k = 2
29 l = 5
30 T = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [h, k, l, 1]])
31
32 M = v@T
33 # lista de lados del polígonos
34 verts = [[v[0,:-1],v[4,:-1],v[5,:-1]], [v[4,:-1],v[5,:-1],v[6,:-1]],
35 [v[0,:-1],v[1,:-1],v[2,:-1]], [v[0,:-1],v[2,:-1],v[3,:-1]],
36 [v[0,:-1],v[3,:-1],v[4,:-1]], [v[0,:-1],v[1,:-1],v[5,:-1]],
37 [v[1,:-1],v[2,:-1],v[3,:-1]], [v[4,:-1],v[5,:-1],v[6,:-1]]]
38
39 #color sides of figure original
40 cmap = mpl.cm.get_cmap('plasma')
41 for i in range(len(verts)):
42     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
43     coll = Poly3DCollection(verts[i],edgecolors='w')
44     coll.set_facecolor(c)
45     coll.set_clim(vmin=0, vmax=100)
46     ax.add_collection(coll)
47 plt.show()
48
49 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
50 verts = [[M[0,:-1],M[4,:-1],M[5,:-1]],[M[4,:-1],M[5,:-1],M[6,:-1]],
51 [M[0,:-1],M[1,:-1],M[2,:-1]],[M[0,:-1],M[2,:-1],M[3,:-1]],
52 [M[0,:-1],M[3,:-1],M[4,:-1]],[M[0,:-1],M[1,:-1],M[5,:-1]],
53 [M[1,:-1],M[2,:-1],M[3,:-1]],[M[4,:-1],M[5,:-1],M[6,:-1]]]
54
55
56 cmap = mpl.cm.get_cmap('plasma')
57 for i in range(len(verts)):
58     c = cmap(mpl.colors.Normalize(0, 100)(i*10))
59     coll = Poly3DCollection(verts[i],edgecolors='w')
60     coll.set_facecolor(c)
61     coll.set_clim(vmin=0, vmax=100)
62     ax.add_collection(coll)
63
64 plt.show()

```

Figura 6.8: Implementación de código

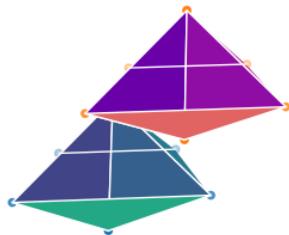


Figura 6.9: Translación en 3D en los tres ejes

#### 6.2.2 Escalamiento

Un escalado alrededor del origen por un factor  $w$  (frecuentemente  $w = 1$ ) en las direcciones  $x, y$  y  $z$ , se obtienen a partir de la siguiente matriz de transformación

$$S(s_x, s_y, s_z, 1) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Por lo tanto, para un punto  $P(x, y, z, 1)$ , el escalamiento es

$$S(x, y, z) = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (s_x x, s_y y, s_z z, 1)$$

#### Ejercicio 6.2.2.1

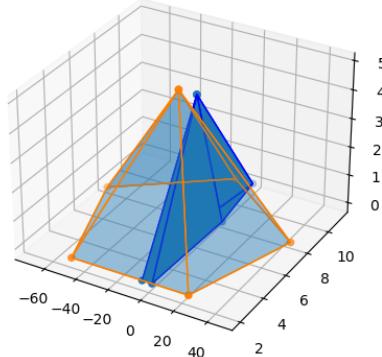
Realizar una transformación de escalamiento de 12 unidades en el eje X a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje X ( $S_x = 12$ ).

```

14 from matplotlib import pyplot as plt
15 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
16 import numpy as np
17
18 fig = plt.figure()
19 ax = fig.add_subplot(111, projection='3d')
20
21 # Puntos de la figura
22 v = np.array([[-1, 6.13, 4.9, 1], [-4.2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1]])
23 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
24
25 # Matriz de transformación
26 sx = 12
27 sy = 0
28 sz = 0
29 T = np.array([[sx, 0, 0, 0], [0, sy, 0, 0], [0, 0, sz, 0], [0, 0, 0, 1]])
30
31 # Lista de lados del polígono
32 vertx = [[v[0, 0:-1], v[4, 0:-1], v[5, 0:-1], v[4, 0:-1], v[5, 0:-1], v[6, 0:-1]],
33           [v[0, 0:-1], v[1, 0:-1], v[2, 0:-1]], [v[0, 0:-1], v[2, 0:-1], v[3, 0:-1]],
34           [v[0, 0:-1], v[3, 0:-1], v[4, 0:-1]],
35           [v[0, 0:-1], v[1, 0:-1], v[5, 0:-1]], [v[1, 0:-1], v[2, 0:-1], v[3, 0:-1], v[4, 0:-1], v[5, 0:-1]]]
36
37 # Plot de los resultados
38 ax.add_collection3d(Poly3DCollection(vertx, linewidths=1, edgecolors='b', alpha=1))
39
40 M = v@T
41 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
42 vertx = [[M[0, 0:-1], M[4, 0:-1], M[5, 0:-1], M[4, 0:-1], M[5, 0:-1], M[6, 0:-1]],
43           [M[0, 0:-1], M[1, 0:-1], M[2, 0:-1]], [M[0, 0:-1], M[2, 0:-1], M[3, 0:-1]],
44           [M[0, 0:-1], M[3, 0:-1], M[4, 0:-1]],
45           [M[0, 0:-1], M[1, 0:-1], M[5, 0:-1]], [M[1, 0:-1], M[2, 0:-1], M[3, 0:-1], M[4, 0:-1], M[5, 0:-1]]]
46 ax.add_collection3d(Poly3DCollection(vertx, linewidths=1, edgecolors='#FF8000', alpha=0.25))
47

```

(a) Implementación de código



(b) Escala en eje X de prisma

Figura 6.10: Escalamiento en eje X

#### Ejercicio 6.2.2.2

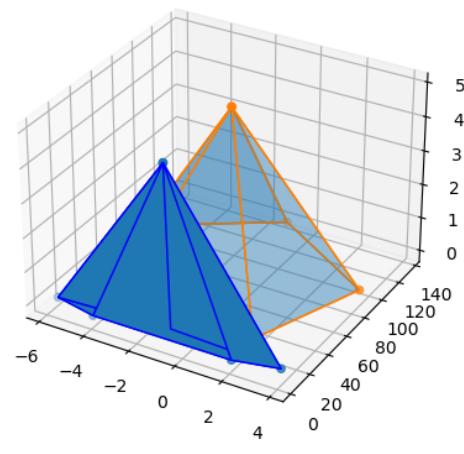
Realizar una transformación de escalamiento de 12 unidades en el eje Y a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje Y ( $S_y = 12$ ).

```

14 from matplotlib import pyplot as plt
15 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
16 import numpy as np
17
18 fig = plt.figure()
19 ax = fig.add_subplot(111, projection='3d')
20
21 # vPuntos de la figura
22 v = np.array([[-1, 6.13, 4.9, 1], [-4.2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1],
23 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1]])
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 #Matriz de transformacion
27 sx = 1
28 sy = 12
29 sz = 1
30 T = np.array([[sx, 0, 0, 0], [0, sy, 0, 0], [0, 0, sz, 0], [0, 0, 0, 1]])
31
32 # lista de lados del poligono
33 verts = [[v[0,0:-1],v[4,0:-1],v[5,0:-1],[v[4,0:-1],v[5,0:-1],v[6,0:-1]],
34 [v[0,0:-1],v[1,0:-1],v[2,0:-1],[v[0,0:-1],v[2,0:-1],v[3,0:-1]],
35 [v[0,0:-1],v[3,0:-1],v[4,0:-1]],
36 [v[0,0:-1],v[1,0:-1],v[4,0:-1],[v[1,0:-1],v[2,0:-1],v[3,0:-1],v[4,0:-1],v[5,0:-1]]]
37 # plot sides
38 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors='b', alpha=1))
39
40 M = v@T
41 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
42 verts = [[M[0,0:-1],M[4,0:-1],M[5,0:-1],[M[4,0:-1],M[5,0:-1],M[6,0:-1]],
43 [M[0,0:-1],M[1,0:-1],M[2,0:-1],[M[0,0:-1],M[2,0:-1],M[3,0:-1]],
44 [M[0,0:-1],M[3,0:-1],M[4,0:-1]],
45 [M[0,0:-1],M[1,0:-1],M[5,0:-1],[M[1,0:-1],M[2,0:-1],M[3,0:-1],M[4,0:-1],M[5,0:-1]]]
46 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors="#FF8000", alpha=0.25))
47
48 plt.show()

```

(a) Implementación de código



(b) Escala en eje Y de prisma

Figura 6.11: Escalamiento en eje Y

**Ejercicio 6.2.2.3**

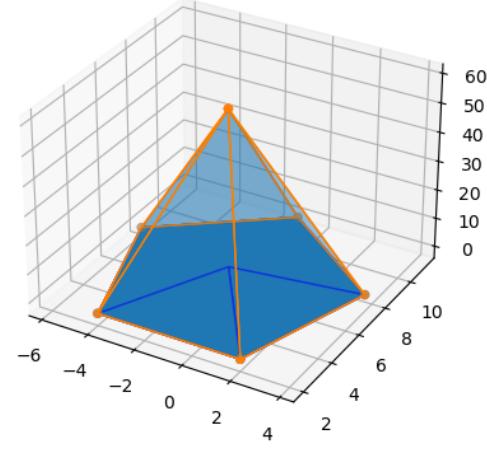
Realizar una transformación de escalamiento de 12 unidades en el eje Z a un prisma pentagonal. Es decir, aplicar una transformación únicamente en el eje Z ( $S_z = 12$ ).

```

14 from matplotlib import pyplot as plt
15 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
16 import numpy as np
17
18 fig = plt.figure()
19 ax = fig.add_subplot(111, projection='3d')
20
21 # vPuntos de la figura
22 v = np.array([[-1, 6.13, 4.9, 1], [-4.2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1],
23 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1]])
24 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
25
26 #Matriz de transformacion
27 sx = 1
28 sy = 1
29 sz = 12
30 T = np.array([[sx, 0, 0, 0], [0, sy, 0, 0], [0, 0, sz, 0], [0, 0, 0, 1]])
31
32 # lista de lados del poligono
33 verts = [[v[0,0:-1],v[4,0:-1],v[5,0:-1],[v[4,0:-1],v[5,0:-1],v[6,0:-1]],
34 [v[0,0:-1],v[1,0:-1],v[2,0:-1],[v[0,0:-1],v[2,0:-1],v[3,0:-1]],
35 [v[0,0:-1],v[3,0:-1],v[4,0:-1]],
36 [v[0,0:-1],v[1,0:-1],v[5,0:-1],[v[1,0:-1],v[2,0:-1],v[3,0:-1],v[4,0:-1],v[5,0:-1]]]
37 # plot sides
38 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors='b', alpha=1))
39
40 M = v@T
41 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
42 verts = [[M[0,0:-1],M[4,0:-1],M[5,0:-1],[M[4,0:-1],M[5,0:-1],M[6,0:-1]],
43 [M[0,0:-1],M[1,0:-1],M[2,0:-1],[M[0,0:-1],M[2,0:-1],M[3,0:-1]],
44 [M[0,0:-1],M[3,0:-1],M[4,0:-1]],
45 [M[0,0:-1],M[1,0:-1],M[5,0:-1],[M[1,0:-1],M[2,0:-1],M[3,0:-1],M[4,0:-1],M[5,0:-1]]]
46 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors="#FF8000", alpha=0.25))
47
48 plt.show()

```

(a) Implementación de código



(b) Escala en eje Z de prisma

Figura 6.12: Escalamiento en eje Z

**Ejercicio 6.2.2.4**

Realizar una expansión a un tetraedro. La expansión para cada eje es  $S_x = 3$ ,  $S_y = 4$  y  $S_z = 5$ .

```

1 # -*-_coding: utf-8 -*-
2 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
3 CU UAEM ZUMPANGO
4 UN: Gráficas computacionales
5 Tema: Transformaciones en 3D
6 Alumno: Keren Mitsue Ramírez Vergara
7 Descripción: escalar un tetraedro
8
9 Created on Sat Aug 27 19:30:42 2022
10
11 @author: KerenMitsue
12 """
13
14 import matplotlib.pyplot as plt
15 import numpy as np
16 from numpy import pi, sin, cos, exp, linspace
17
18 sx = float(input("Dev el factor de escala en el eje x:"))
19 sy = float(input("Dev el factor de escala en el eje y:"))
20 sz = float(input("Dev el factor de escala en el eje z:"))
21
22 fig = plt.figure()
23 ax = fig.add_subplot(111, projection = "3d")
24 P = np.array([[0,0,4], [0,4,0], [4,0,0], [0,0,4], [0,0,0], [4,0,0], [0,0,0], [0,4,0]])
25 Rx = np.array([[sx,0,0], [0,sy,0], [0,0,sz]])
26 T = P@Rx
27
28 ax.plot(P[:,0], P[:,1], P[:,2])
29 ax.plot(T[:,0], T[:,1], T[:,2])
30
31 plt.show()

```

Dev el factor de escala en el eje x:3  
 Dev el factor de escala en el eje y:4  
 Dev el factor de escala en el eje z:5

Figura 6.13: Implementación en código

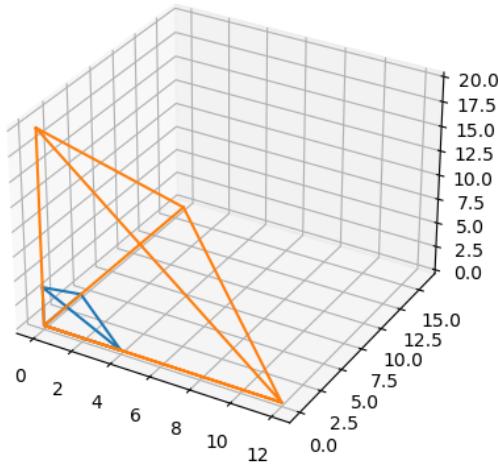


Figura 6.14: Expansión de tetraedro

### 6.2.3 Reflexiones

La reflexiones se aplican en cada uno de los 3 planos. La reflexión  $R_{yz}$  en el plano  $x = 0$ , la reflexión  $R_{xz}$  en el plano  $y = 0$  y  $R_{xy}$  en el plano  $z = 0$ , se obtienen considerando una escala de -1 en las direcciones de coordenadas. El escalamiento de un punto  $P(x, y, z, 1)$  en cada plano es

$$R_{yz} = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -x & y & z & 1 \end{pmatrix}$$

$$R_{xz} = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x & -y & z & 1 \end{pmatrix}$$

$$R_{xy} = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x & y & -z & 1 \end{pmatrix}$$

Ejercicio 6.2.3.1

Realizar una transformación de reflexión a un prisma pentagonal. La reflexión para cada eje es  $x = 3, y = 2, z = 5$ .

```

12 from matplotlib import pyplot as plt
13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15
16 fig = plt.figure()
17 ax = fig.add_subplot(111, projection='3d')
18
19 # vPuntos de la figura
20 v = np.array([[-1, 6.13, 4.9, 1], [-4.2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1],
21 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1]])
22 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
23
24 #Matriz de transformacion
25 h = 3
26 k = 2
27 l = 5
28 Ryz = np.array([[ -1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
29 Rxz = np.array([[ 1, 0, 0, 0], [0, -1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
30 Rxy = np.array([[ 1, 0, 0, 0], [0, 1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
31
32 # lista de lados del poligono
33 verts = [[v[0,0:-1],v[4,0:-1],v[5,0:-1]],[v[4,0:-1],v[5,0:-1],v[6,0:-1]],
34 [v[0,0:-1],v[1,0:-1],v[2,0:-1]],[v[0,0:-1],v[2,0:-1],v[3,0:-1]],
35 [v[0,0:-1],v[3,0:-1],v[4,0:-1]],[v[0,0:-1],v[1,0:-1],v[5,0:-1]],
36 [v[1,0:-1],v[2,0:-1],v[3,0:-1]],[v[4,0:-1],v[5,0:-1]]]
37 # plot sides
38 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors='b',
39 alpha=0.25))
40
41 M = v@Rxy
42 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
43 verts = [[M[0,0:-1],M[4,0:-1],M[5,0:-1]],[M[4,0:-1],M[5,0:-1],M[6,0:-1]],
44 [M[0,0:-1],M[1,0:-1],M[2,0:-1]],[M[0,0:-1],M[2,0:-1],M[3,0:-1]],
45 [M[0,0:-1],M[3,0:-1],M[4,0:-1]],[M[0,0:-1],M[1,0:-1],M[5,0:-1]],
46 [M[1,0:-1],M[2,0:-1],M[3,0:-1]],[M[4,0:-1],M[5,0:-1]]]
47 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors="#FF8000",
48 alpha=0.25))
49
50 plt.show()

```

Figura 6.15: Transformación de reflexión

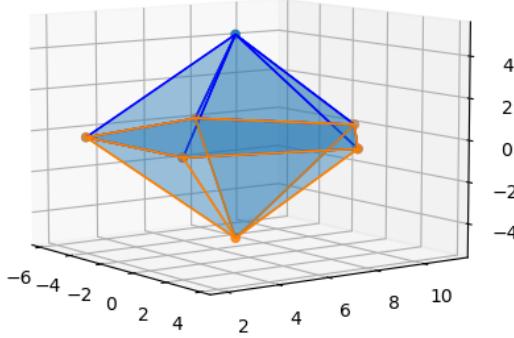


Figura 6.16: Reflección en el prisma

#### 6.2.4 Rotación

Las rotaciones en el espacio tienen lugar alrededor de una línea llamada eje de rotación. Las rotaciones alrededor de los tres ejes de coordenadas se denominan **rotaciones primarias**.

La rotación sobre el eje  $x$  con un ángulo  $\theta_x$  es

$$Rot_x(\theta_x) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La rotación sobre el eje  $y$  con un ángulo  $\theta_y$  es

$$Rot_y(\theta_y) = \begin{pmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La rotación sobre el eje  $z$  con un ángulo  $\theta_z$  es

$$Rot_z(\theta_z) = \begin{pmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

#### Ejercicio 6.2.4.1

Realizar una transformación de rotación a un prima pentagonal, La rotación a aplicar es de  $45^\circ$ . Implementa el código para rotar en cada eje, además de mostrara el código y sus diferentes opciones.

```

12 from matplotlib import pyplot as plt
13 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
14 import numpy as np
15
16 fig = plt.figure()
17 ax = fig.add_subplot(111, projection='3d')
18
19 # VPuntos de la figura
20 v = np.array([[-1.6, 13, 4.9, 1], [-4, 2, 0, 1], [2, 2, 0, 1], [3.85, 7.71, 0, 1],
21 [-1, 11.23, 0, 1], [-5.85, 7.71, 0, 1], [-1, 6.13, 4.9, 1]])
22 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
23
24 #Matriz de transformacion
25 gr = 45
26 th = gr * (np.pi/180)
27 Rox = np.array([[1, 0, 0, 0], [0, np.cos(th), np.sin(th), 0], [0, -np.sin(th),
28 | np.cos(th), 0], [0, 0, 0, 1]])
29 Roy = np.array([[np.cos(th), 0, -np.sin(th), 0], [0, 1, 0, 0],
30 | [np.sin(th), 0, np.cos(th), 0], [0, 0, 0, 1]])
31 Roz = np.array([[np.cos(th), np.sin(th), 0, 0], [-np.sin(th), 0, np.cos(th), 0],
32 | [0, 0, 1, 0], [0, 0, 0, 1]])
33
34
35 # lista de lados del polígono
36 verts = [[v[0,:-1],v[4,:-1],v[5,:-1],[v[4,:-1],v[5,:-1],v[6,:-1]],
37 | [v[0,:-1],v[1,:-1],v[2,:-1],[v[0,:-1],v[2,:-1],v[3,:-1]],
38 | [v[0,:-1],v[3,:-1],v[4,:-1],[v[0,:-1],v[1,:-1],v[5,:-1]],
39 | [v[1,:-1],v[2,:-1],v[3,:-1],[v[4,:-1],v[5,:-1]]]
40
41 # plot sides
42 ax.add_collection3d(Poly3DCollection(verts,linewidths=1,edgecolors='b',
43 | alpha=1))
44
45
46
47 M = v@Rox
48 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
49 verts = [[M[0,:-1],M[4,:-1],M[5,:-1],[M[4,:-1],M[5,:-1],M[6,:-1]],
50 | [M[0,:-1],M[1,:-1],M[2,:-1],[M[0,:-1],M[2,:-1],M[3,:-1]],
51 | [M[0,:-1],M[3,:-1],M[4,:-1],[M[0,:-1],M[1,:-1],M[5,:-1]],
52 | [M[1,:-1],M[2,:-1],M[3,:-1],[M[4,:-1],M[5,:-1]]]
53
54 ax.add_collection3d(Poly3DCollection(verts,linewidths=1,edgecolors="#FF8000",
55 | alpha=1))
56
57 plt.show()

```

Figura 6.17: Rotación de un tetraedro

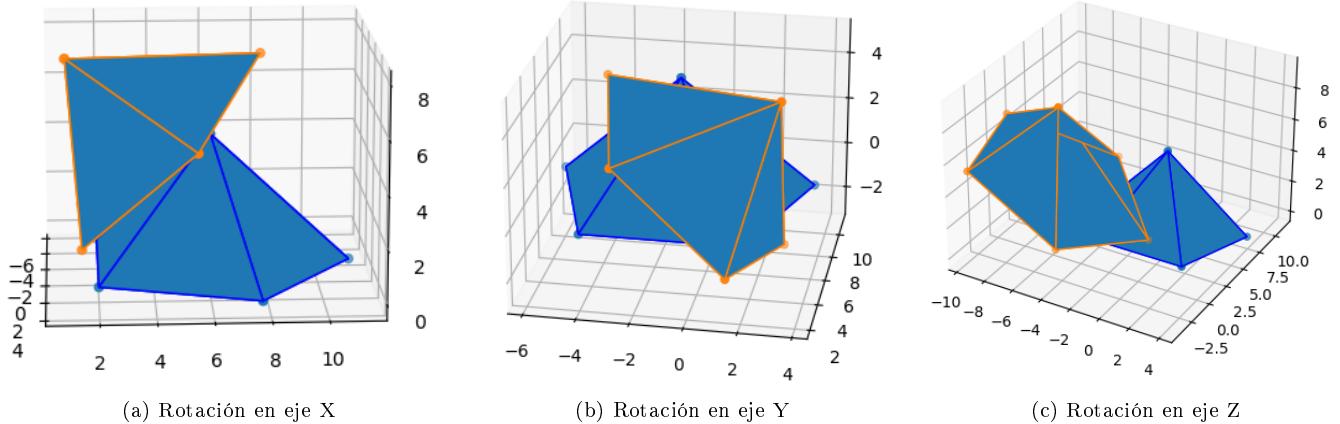


Figura 6.18: Rotaciones en cada eje

#### Ejercicio 6.2.4.2

Realizar una animación donde una figura de tetraedro, sea rotada varias veces, hasta formar un cono, considere mostrar acada una de las transformaciones aplicadas.

#### Ejercicio 6.1

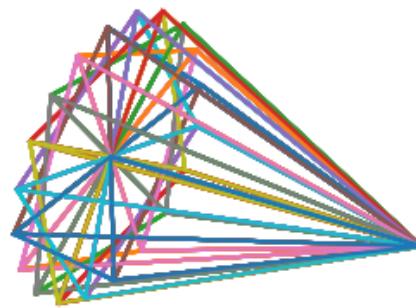
Realizar una transformación de traslación y rotación a un tetraedro. La traslación para cada eje es  $T_x = 3$ ,  $T_y = 2$  y  $T_z = 8$ . El grado de rotación para la figura es de  $45^\circ$ .

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17 from matplotlib.collections import PolyCollection
18 from numpy import pi, sin, cos, exp, linspace
19 import matplotlib.patches as mpatches
20 import matplotlib.path as mpath
21
22 Path = mpath.Path
23 path_data = [(Path.MOVETO,(0, 0, 0)),(Path.LINETO,(0, 4, 0)),
24               (Path.LINETO,(4, 0, 0)),(Path.CLOSEPOLY,(0, 0, 0))]
25
26 #Rotacion en eje x con translacion en xy
27 fig = plt.figure()
28 ax = fig.add_subplot(111, projection = "3d")
29 P = np.array([[0, 0, 4], [0, 4, 0], [4, 0, 0], [0, 0, 4], [0, 0, 0],
30               [4, 0, 0], [0, 0, 0], [0, 4, 0]])
31
32 for i in range(280):
33     Rox = np.array([ [1,0,0,0],[0,np.cos(0.1*np.pi*i),np.sin(0.1*np.pi*i),0],
34                     [0,-np.sin(0.1*np.pi*i),np.cos(0.1*np.pi*i),0],[0,0,0,1]])
35     T = P@Rox
36     ax.plot(T[:, 0], T[:, 1], T[:, 2])
37     #ax.add_collection3d(PolyCollection(P),zs=T[:, 2])
38     ax.axis("off")
39     plt.pause(0.5)
40

```

(a) Implementación de código



(b) Animación

Figura 6.19: Animación de tetraedro

```

13 from matplotlib import pyplot as plt
14 from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection
15 import numpy as np
16
17 fig = plt.figure()
18 ax = fig.add_subplot(111, projection='3d')
19
20 # vertices of a pyramid
21 v = np.array([[3, 1, 0, 1],[2, 1, 0, 1],[-0.5, 5.33, 0, 1],[-0.5, 2.44, 4.08, 1],[-3, 1, 0, 1]])
22 ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
23
24 # generate list of sides' polygons of our pyramid
25 verts = [[v[0,0:-1],v[1,0:-1],v[4,0:-1],[v[0,0:-1],v[3,0:-1],v[4,0:-1]],[v[1,0:-1],
26           v[2,0:-1],v[3,0:-1],v[4,0:-1]],[v[0,0:-1],v[1,0:-1],v[2,0:-1],v[3,0:-1]],
27           v[3,0:-1],v[4,0:-1],[v[0,0:-1],v[1,0:-1],v[2,0:-1],v[3,0:-1]]]
28
29 # plot sides
30 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors='b', alpha=1))
31
32 #Translacion en xy y Rotacion en eje x
33 h = 3
34 k = 2
35 l = 2
36
37 T = np.array([ [1,0,0,0],[0,1,0,0],[0,0,1,0],[h,k,l,1]])
38 gr = 45
39 th = gr * (np.pi/180)
40 Rox = np.array([[1,0,0,0],[0,np.cos(th),np.sin(th),0],[0,-np.sin(th),np.cos(th),0],[0,0,0,1]])
41
42 R = Rox @ T
43 M = R @ T
44
45 T = T@Rox
46 M = v@T
47
48 ax.scatter3D(M[:, 0], M[:, 1], M[:, 2])
49 verts = [[M[0,0:-1],M[1,0:-1],M[4,0:-1]],[M[0,0:-1],M[3,0:-1],M[4,0:-1]],[M[1,0:-1],
50           M[2,0:-1],M[3,0:-1]],[M[2,0:-1],M[1,0:-1],M[4,0:-1]],[M[2,0:-1],M[3,0:-1]],[M[4,0:-1],
51           M[0,0:-1],M[1,0:-1],M[2,0:-1],M[3,0:-1],[M[0,0:-1],M[1,0:-1],M[3,0:-1]]]
52 ax.add_collection3d(Poly3DCollection(verts, linewidths=1, edgecolors="#FFB000", alpha=1))
53
54 plt.show()

```

(a) Implementación de código

Figura 6.20: Transformación concatenada

### Ejercicio Extra

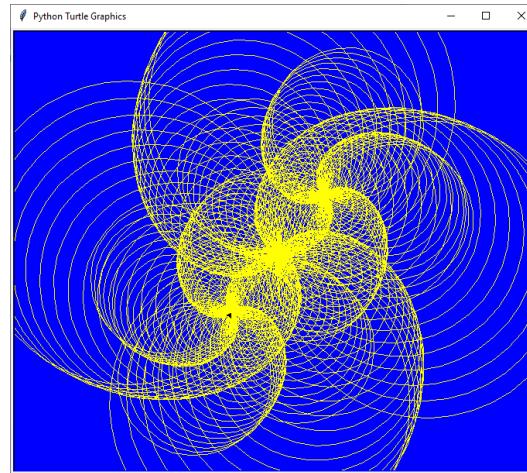
Utiliza la librería **turtle** de python para realizar círculos, considere que el centro debe ser desplazado.

```

1 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
2 CU UAEM ZUMPANGO
3 UA: Graficación computacional
4 Tema: Transformaciones en 3D
5 Alumno: Keren Mitsue Ramírez Vergara
6 Profesor: Manuel Almeida Vázquez
7 Descripción: Uso de la librería turtle
8 Created on Sat Aug 27 19:30:42 2022
9
10 @author: KerenMitsue
11 """
12
13 import turtle as t
14
15 t.speed(0)
16 t.bgcolor("blue")
17 t.pencolor("yellow")
18
19 for x in range(100):
20     t.circle(x*2)
21     t.right(91)
22 t.setpos(60,75) #desplazamiento del pencolor a coordenada (60,75)
23
24 for x in range(100):
25     t.circle(x)
26     t.right(91)
27
28 t.setpos(-60,-75)
29 for x in range(100):
30     t.circle(x)
31     t.right(91)

```

(a) Implementación de código



(b) Dibujo de círculos

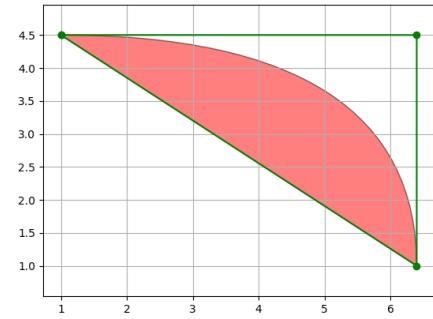
Figura 6.21: Dibujar círculos

## Ejercicio Extra

Como se muestra en la figura 6.22, se ha implementado el uso de la librería **path** en python, la librería path permite dibujar figuras a partir de las manipulación de coordenadas.

```
15 import matplotlib.patches as mpatches
16 import matplotlib.path as mpath
17 import matplotlib.pyplot as plt
18
19 fig, ax = plt.subplots()
20 Path = mpath.Path
21 path_data = [(Path.MOVETO,(1.0,4.5)),(Path.CURVE3,(6.4,4.5)),
22               (Path.LINETO,(6.4,1.0)),(Path.CLOSEPOLY,(1.0,4.5))]
23
24 codes , verts = zip(*path_data)
25 path = mpath.Path(verts,codes)
26 patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
27 ax.add_patch(patch)
28 x, y = zip(*path.vertices)
29 line, = ax.plot(x,y,'go-')
30 ax.grid()
31 ax.axis('equal')
32 plt.show()
```

(a) código



(b) Figura

Figura 6.22: figura realizada con librería path

# Capítulo 7

## Curvas

Las curvas surgen en muchas aplicaciones como el arte, el diseño industrial, las matemáticas arquitectura e ingeniería, y se han desarrollado numerosos paquetes de dibujo por ordenador y paquetes de dibujo por ordenador y de diseño asistido por ordenador han sido desarrollados para facilitar la creación de curvas. Una aplicación especialmente ilustrativa es la de las fuentes informáticas, que se definen mediante curvas que especifican el contorno de cada carácter de la fuente. Los diferentes tamaños de las fuentes se obtienen aplicando transformaciones de escala. Los efectos especiales de los tipos de letra de efectos especiales de los tipos de letra se obtienen aplicando otras transformaciones, como cizallas y rotaciones. Asimismo, en otras aplicaciones es necesario realizar diversas tareas como modificar, analizar y visualizar las curvas. Para poder ejecutar dichas operaciones se requiere una representación matemática de las curvas.

El proceso de dibujar una curva se llama **renderizado**. Las curvas paramétricas son las más utilizadas en los gráficos por ordenador y en el modelado geométrico, ya que los puntos en la curva se calculan fácilmente.

Una curva con la siguiente estructura  $C(t) = (x(t), y(t))$  definida en un intervalo de  $[a, b]$  es considerable evaluar  $n + 1$  puntos, donde  $t_0 < \dots < t_n$ ,  $t_0 = a$  y  $t_n = b$ . Los puntos se unen en secuencia por segmentos de línea para dar una aproximación lineal a la curva, como se muestra en la figura 7.1. Si la aproximación resultante es demasiado irregular, se puede obtener una curva más suave aumentando el número de puntos evaluados.

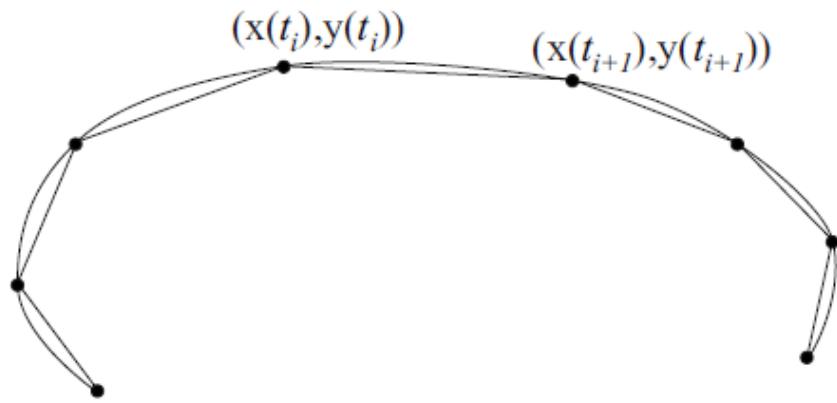


Figura 7.1: Línea de aproximación a una curva paramétrica.

### Ejercicio 7.1

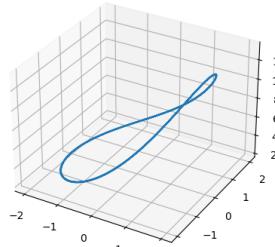
Realizar la implementación en Python de una curva en 3D

```

15 import matplotlib.pyplot as plt
16 from numpy import *
17
18 fig = plt.figure()
19 ax = fig.add_subplot(111, projection='3d')
20 t = linspace(0,8*pi,100)
21 x = 2*cos(t)
22 y = 2*sin(t)
23 z = 5*exp(cos(t))
24 plt.plot(x,y,z)
25 plt.grid()
26 #plt.axis("off")
27 plt.show()

```

(a) Implementación de código



(b) Curva en 3D

Figura 7.2: Curva en 3D en Python

## 7.1 Curvas de b茅zier

Las curvas de B茅zier fueron publicadas por primera vez en 1962 por el ingeniero franc茅s Pierre B茅zier y posteriormente, trabajando en la Renault, las us贸 con abundancia en el dise帽o de las diferentes partes del autom贸vil.

Las curvas fueron desarrolladas por Paul de Casteljau usando en el algoritmo que lleva su nombre los polinomios de Bernstein. Se trata de un m茅todo num茅ricamente estable para evaluar las curvas de B茅zier.

Posteriormente, los inventores de PostScript, lenguaje que permiti贸 el desarrollo de sistemas de impresi贸n de alta calidad desde el ordenador, introdujeron en ese c芒digo de las curvas y los trazados.

El lenguaje PostScript sigue emple谩ndose ampliamente y se ha convertido en un est谩ndar de calidad universal; por ello los programas de dise帽o vectorial como Adobe Illustrator e Inkspace, denominan <b茅zier> a algunas de sus herramientas de dibujo, y se habla de <Trazados b茅zier>, <pluma b茅zier>, <l脿piz b茅zier>.

Para trazar una recta entre un punto A con coordenadas  $(x_1, y_1)$  y un punto B con coordenadas  $(x_2, y_2)$  basta con conocer su posici贸n. Si en lugar de unir dos puntos con una recta se unen con una curva, surgen los elementos esenciales de una curva B茅zier; los puntos se denominan: **puntos de anclaje o nodos**. La forma de la curva se define por los puntos invisibles en el dibujo denominados: **puntos de control, manejadores o manecillas**. Como se ilustra en la figura 7.3



Figura 7.3: Curva de B茅zier

Se denomina curva de B茅zier a un m茅todo de definici贸n de una curva en serie de potencias. El m茅todo consiste en definir algunos puntos de control, a partir de los cuales se calculan los puntos de la curva. Describiremos el m茅todo de construcci贸n recursivo conocido como algoritmo de Casteljau. Para dos puntos, la curva es un segmento recto, denido en forma param茅trica por interpolaci贸n de los puntos extremos:

$$P = (1 - t)P_0 + tP_1$$

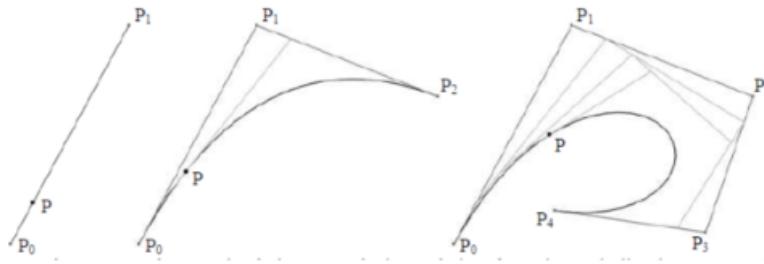


Figura 7.4: Proceso iterado de la curva de Bézier

Agregando un punto de control más la curva adquiere más interés: se interpola linealmente en cada uno de los segmentos y luego entre los puntos resultantes, siempre con el mismo valor del parámetro. Para más puntos de control, el proceso se repite en forma iterativa. La poligonal que forman los puntos de control se conoce como **polígono de control**. Por razones históricas se llama orden de una curva de Bézier a la cantidad de puntos de control. Para dos puntos de control, la curva es de segundo orden y primer grado, por cada punto de control que se agrega, se agrega además un paso de interpolación, en donde los términos en  $t$  quedan multiplicados por  $t$  o  $(1 - t)$  y por lo tanto se aumenta en uno el grado del polinomio. Nosotros no utilizaremos el orden sino el grado pero es necesario definirlo.  $o = n + 1$  ( $o := \text{Orden} = \text{grado} + 1$ )

## 7.2 Curvas lineales de Bézier

Una curva lineal de b茅zier es un segmento de l铆nea unidos por dos puntos de control  $P_0$  y  $P_1$ , una curva lineal de B茅zier es una l铆nea recta entre los dos puntos. La curva est谩 dada por la expresi贸n.

$$B_1(t) = (1 - t)P_0 + tP_1$$

La curva est谩 definida en el intervalo  $t \in [0, 1]$ , por lo que el punto inicial de la curva es  $B(0) = P_0$  y el punto final  $B(1) = P_1$ , es decir, la curva de b茅zier interpola el primer y el ltimo punto de control.

## 7.3 Curvas cuadr谩ticas de B茅zier

Una curva cuadr谩tica de B茅zier es el camino trazado por la funci贸n  $B(t)$ , dado los puntos:  $P_0$ ,  $P_1$  y  $P_2$ . Por lo que la curva de b茅zier se define como

$$B_2(t) = (1 - t)^2 P_0 + 2(1 - t)tP_1 + t^2 P_2$$

El punto inicial de la curva es  $B(0) = P_0$  y el punto final es  $B(1) = P_2$ . El tri脿ngulo  $P_0P_1P_2$  obtenido al unir los puntos de control con los segmentos de l铆nea en su orden prescrito, se le llama **polígono de control**.

### Ejercicio 7.3.1

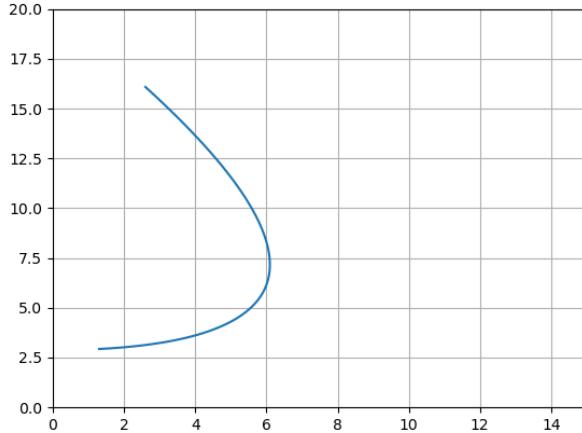
En la figura 7.5 se muestra la implementaci贸n de un c髍digo que genera una curva de b茅zier cuadr谩tica, a partir de puntos que son seleccionados por el usuario.

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17
18 P = np.array([[4,0], [-5,2], [-2,8]])
19
20 t = np.arange(0,1.01,0.01)
21 t2= t*t
22 U = np.ones((len(t),1), dtype=float)
23 B = np.zeros((len(t),2), dtype=float)
24 B1 = np.zeros((3,2), dtype=float)
25 M = np.array([ [1,-2,1],[-2,2,0],[1,0,0] ])
26 P = np.zeros((3,2), dtype = float)
27 W = np.zeros((len(t),3), dtype=float)
28
29 W[:,0] = t2[:,]
30 W[:,1] = t[:,]
31 W[:,2] = U[:,0]
32
33 plt.plot()
34 plt.grid(True)
35 plt.axis([0,15,0,20])
36 P = np.array(plt.ginput(3, mouse_stop=3))
37 print(P)
38 Bl= M@P
39 B = W@Bl
40 plt.plot(B[:,0], B[:,1])
41 plt.show()
42

```

(a) Implementación de código



(b) Curva de Bézier

Figura 7.5: Curva de bézier cuadrática

## 7.4 Estructura general de las curvas de Bézier

Una curva de Bézier es una curva polinomial que aproxima a una serie de puntos llamados “puntos de control”. Esta curva puede ser de cualquier grado, y podemos decir que de una curva de grado  $n$  aproxima a  $n + 1$  puntos de control.

Utilizando una forma paramétrica, las curvas de Bézier se pueden construir utilizando la expresión:

$$B_u(t) = \sum \binom{u}{k} (1-t)^{2-k} t^k P_k$$

donde:

$$\binom{u}{k} = \frac{h!}{(h-k)!k!} \text{ combinaciones}$$

Desarrollando la ecuación, para diferentes grados, se tiene lo siguiente:

$$B_2(t) = \binom{2}{0} (1-t)^2 t^0 P_0 + \binom{2}{1} (1-t)^1 t P_1 + \binom{2}{2} (1-t)^0 t^2 P_2$$

$$B_3(t) = \binom{3}{0} (1-t)^3 t^0 P_0 + \binom{3}{1} (1-t)^2 t P_1 + \binom{3}{2} (1-t)^1 t^2 P_2 + \binom{3}{3} (1-t)^0 t^3 P_3$$

$$\binom{4}{0} (1-t)^4 t^0 P_0 + \binom{4}{1} (1-t)^3 t P_1 + \binom{4}{2} (1-t)^2 t^2 P_2 + \binom{4}{3} (1-t)^1 t^3 P_3 + \binom{4}{4} (1-t)^0 t^4 P_4$$

$$B_5(t) = \binom{5}{0} (1-t)^5 t^0 P_0 + \binom{5}{1} (1-t)^4 t P_1 + \binom{5}{2} (1-t)^3 t^2 P_2 + \binom{5}{3} (1-t)^2 t^3 P_3 + \\ \binom{5}{4} (1-t)^1 t^4 P_4 + \binom{5}{5} (1-t)^0 t^5 P_5$$

### Ejercicio 7.4.1

Seleccione 3 puntos en el plano 2D y calcule 4 puntos de la curva de Bézier correspondiente, y realice su implementación en python. Los puntos a seleccionar son:  $(4,0)$ ,  $(-5,2)$  y  $(-2,8)$  cuando  $t \in [0, 1/4, 1/2, 3/4, 1]$ .

Utilizando un Bézier cuadrático de la forma:  $(1-t)^2 P_0 + 2(1-t)(t)P_1 + t^2 P_2$ , se realiza el cálculo para cada uno de los puntos.

$$\begin{aligned} t = 0 &: (1-0)^2(4,0) + 2(1-0)(0)(-5,2) + (0)^2(-2,8) = (4,0) \\ t = 1/4 &: (1-1/4)^2(4,0) + 2(1-1/4)(1/4)(-5,2) + (1/4)^2(-2,8) = (-23/16, 5/4) \\ t = 1/2 &: (1-1/2)^2(4,0) + 2(1-1/2)(1/2)(-5,2) + (1/2)^2(-2,8) = (-11/4, 3) \\ t = 3/4 &: (1-3/4)^2(4,0) + 2(1-3/4)(3/4)(-5,2) + (3/4)^2(-2,8) = (-47/16, 39/9) \\ t = 1 &: (1-1)^2(4,0) + 2(1-1)(1)(-5,2) + (1)(-2,8) = (-2,8) \end{aligned}$$

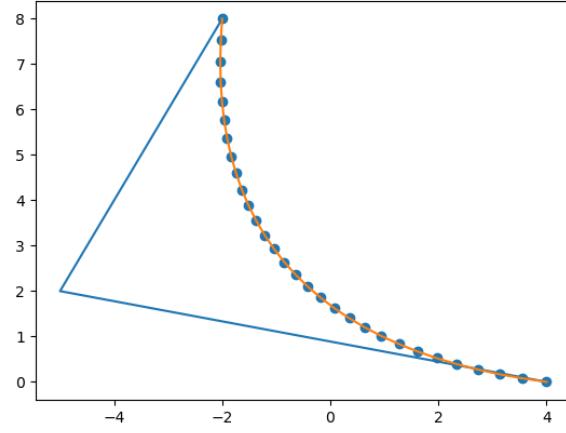
A continuación, en la figura 7.6, se muestra la implementación en código del ejercicio, además de su resultado.

```

1 """UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
2 CU UAEU ZUMPANGO
3 UA: Graficacion computacional
4 Tema: Curvas de Bézier
5 Alumno: Keren Mitsue Ramirez Vergara
6 Profesor: Manuel Almeida Vazquez
7 Descripción: Seleccionar tres puntos en el plano 2D y calcular 4 puntos
8 de la curva de Bézier correspondiente.
9
10 Created on Sat Aug 27 19:30:42 2022
11
12 @author: KerenMitsue
13
14
15 import matplotlib.pyplot as plt
16 import numpy as np
17
18
19 #P = np.array([[1,0], [4,2], [6,4], ])
20 P = np.array([[4,0], [-5,2], [-2,8]])
21 plt.plot(P[:,0], P[:,1])
22
23 t = np.linspace(0,1,30)
24 #t = np.array([ 0, 0.25, 0.5 , 0.75, 1 ])
25 T = np.zeros((30,2))
26 for i in range(0,t.size):
27     T[i,0] = ((1-t[i])**2) * (P[0,0]) + (1-t[i])*t[i]*P[1,0] + (t[i]**2)*P[2,0]
28     T[i,1] = ((1-t[i])**2) * (P[0,1]) + (1-t[i])*t[i]*P[1,1] + (t[i]**2)*P[2,1]
29
30 plt.scatter(T[:,0], T[:,1])
31 plt.plot(T[:,0], T[:,1])
32 plt.show()
33

```

(a) Implementación del código



(b) Curva de Bézier

Figura 7.6: Curva de Bézier de segundo orden

#### Ejercicio 7.4.2

Seleccione 4 puntos en el plano 2D y calcule 4 puntos de la curva de Bézier correspondiente y realice su implementación en python. Los puntos a seleccionar son: (4,0), (-5,2), (-2,8) y , (4,9) cuando  $t \in [0, 1/4, 1/2, 3/4, 1]$

Utilizando un Bézier cuadrático de la forma:  $, (1-t)^3 P_0 + 3(1-t)^2(t)P_1 + 3(1-t)t^2 P_2 + t^3 P_3$ , se realiza el cálculo para cada uno de los puntos.

$$\begin{aligned} t = 0 &: (1-0)^3(4,0) + 3(1-0)^2(0)(-5,2) + 3(1-0)(0)^2(-2,8) + (0)^3(4,9) = (4,0) \\ t = 1/4 &: (1-1/4)^3(4,0) + 3(1-1/4)^2(1/4)(-5,2) + 3(1-1/4)(1/4)^2(-2,8) + (1/4)^3(4,9) = (4,0) \\ t = 1/2 &: (1-1/2)^3(4,0) + 3(1-1/2)^2(1/2)(-5,2) + 3(1-1/2)(1/2)^2(-2,8) + (1/2)^3(4,9) = (-2.37, 7.8) \\ t = 3/4 &: (1-3/4)^3(4,0) + 3(1-3/4)^2(3/4)(-5,2) + 3(1-3/4)(3/4)^2(-2,8) + (3/4)^3(4,9) = (13/64, 477/64) \\ t = 1 &: (1-1)^3(4,0) + 3(1-1)^2(1)(-5,2) + 3(1-1)(1)^2(-2,8) + (1)^3(4,9) = (4,9) \end{aligned}$$

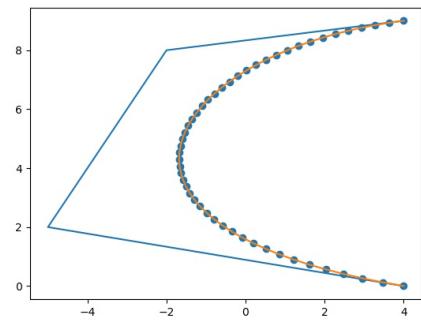
A continuación, en la figura 7.7 se muestra la implementación en código del ejercicio, además de su resultado.

```

1  """UNIVERSIDAD AUTONOMA DEL ESTADO DE MEXICO
2  CU UAEM ZUMPANGO
3  UA: Graficacion computacional
4  Tema: Curvas de Bézier
5  Alumno: Keren Mitsue Ramirez Vergara
6  Profesor: Manuel Almeida Vazquez
7  Descripción: Seleccionar cuatro puntos en el plano 2D y calcular 4 puntos
8  de la curva de bézier correspondiente.
9  """
10 Created on Sat Aug 27 19:30:42 2022
11
12 @author: KerenMitsue
13 """
14
15 import matplotlib.pyplot as plt
16 import numpy as np
17
18
19 #P = np.array([[1,0], [4,2],[6,4],])
20 P = np.array([[4,0], [-5,2],[-2,8],[4,9]])
21 plt.plot(P[:,0], P[:,1])
22
23 t = np.linspace(0,1,50)
24 et = np.arange(0,0.25,0.5 ,0.75,1 )
25 T = np.zeros((50,2))
26 for i in range(0,t.size):
27     T[i,0] = ((1-t[i])**3) * (P[0,0]) + 3*(1-t[i])**2*t[i]*P[1,0] + 3*(1-t[i])*t[i]**2*P[2,0] + ((t[i])**3)*P[3,0]
28     T[i,1] = ((1-t[i])**3) * (P[0,1]) + 3*(1-t[i])**2*t[i]*P[1,1] + 3*(1-t[i])*t[i]**2*P[2,1] + ((t[i])**3)*P[3,1]
29 plt.scatter(T[:,0], T[:,1])
30 plt.plot(T[:,0], T[:,1])
31 plt.show()

```

(a) Implementación del código



(b) Curva de Bézier

Figura 7.7: Curva de Bézier de tercer orden

#### Ejercicio 7.4.3

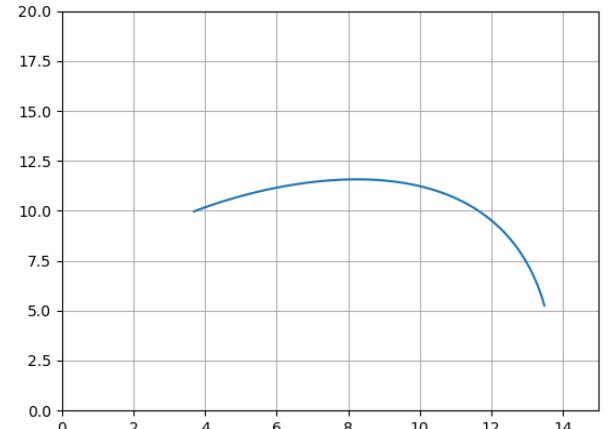
En la figura 7.8 se muestra la implementación de un código que genera una curva de bézier de cuarto grado, a partir de puntos que son seleccionados por el usuario.

```

19 t = np.arange(0,1.01,0.01)
20 t2= t*t
21 t3 = t*t*t
22 U = np.ones((len(t),1), dtype=float)
23 B = np.zeros((len(t),2), dtype=float)
24 B1= np.zeros((4,2), dtype=float)
25 M = np.array( [[-1,3,-3,1],[0,-6,3,0],[-3,3,0,0],[1,0,0,0]])
26 P = np.zeros((4,2), dtype = float)
27 W = np.zeros((len(t),4), dtype=float)
28
29 W[:,0] = t3[:,]
30 W[:,1] = t2[:,]
31 W[:,2] = t[:,]
32 W[:,3] = U[:,0]
33
34 plt.plot()
35 plt.grid(True)
36 plt.axis([0,15,0,20])
37 P = np.array(plt.ginput(4, mouse_stop=4))
38 print(P)
39 B1= M@P
40 B = W@B1
41 plt.plot(B[:,0], B[:,1])
42 plt.show()

```

(a) Implementación de código



(b) Curva de Bézier

Figura 7.8: Curva de bézier cuadrática

## 7.5 Forma matricial de curvas de Bézier

Al desarrollar las ecuaciones de las curvas de bézier se obtienen las siguientes matrices

$$B_2(t) = \begin{bmatrix} t^2 & t & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$B_3(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

## 7.6 Combinaciones de curvas de Bézier

Las curvas simples de Bézier no son suficientes para el diseño, por lo que es necesario hacer combinaciones de curvas Bézier de distintos grados, estas combinaciones no son por lo general suaves, por lo que es necesario una forma de suavizarlas, por lo que se debe calcular una nueva curva que satisfaga la condición de ser suave en su unión. Además se puede deformar nuevas curvas formadas por partes de varias curvas de Bézier.

Dadas dos curvas de Bézier  $B_1$  de puntos de control; por ejemplo  $P_0, P_1, P_2$ , y  $Q_0, Q_1, Q_2$  de  $B_2$ , que se unen en un punto común, el último de  $B_1$  y el primero de  $B_2$ . Esto es  $P_2 = Q_0$ , construimos un nuevo punto común que será una combinación lineal de los puntos  $P_1$  y  $Q_1$ , dada por

$$(1 - \alpha)P_0 + \alpha Q_1$$

cuando  $0 < \alpha < 1$ .

En la figura 7.9 se muestra la concatenación de dos curvas de b茅zier cuadr谩ticas, adem谩s, aplicando una variable de suavizado, se observa como cambia la direcci髇 de la curva.

```
 9 import numpy as np
10 import matplotlib.pyplot as plt
11
12 def segOrd(P, M, W):
13     B1 = M @ P
14     B = W @ B1
15     return B
16
17 # Puntos
18 P = np.array([[2,1],[4,7],[-3,5]], dtype=float)
19 Q = np.array([[-3,5],[8,-2],[7,2]], dtype=float)
20
21 # Bezier
22 t = np.arange(0,1.01, 0.01)
23 t2 = t * t
24
25 U = np.ones((len(t), 1))
26 M = np.array([[1,-2,1],[-2,2,0],[1,0,0]])
27 W = np.zeros((len(t), 3))
28 W[:,0] = t2[:,]
29 W[:,1] = t[:,]
30 W[:,2] = U[:,0]
31
32 B1 = segOrd(P, M, W)
33 B2 = segOrd(Q, M, W)
34
35 # Primer plot
36 fig, ax = plt.subplots(1,2);
37 ax[0].plot(B1[:,0], B1[:,1])
38 ax[0].plot(B2[:,0], B2[:,1])
39 ax[0].set_title('Unión')
40 ax[0].grid()
41
42 # Suavizado
43 alph = 0.3
44 PQ = np.zeros([1,2])
45 PQ[0,0] = (1 - alph) * P[1,0] + alph * Q[1,0]
46 PQ[0,1] = (1 - alph) * P[1,1] + alph * Q[1,1]
47
48 P[2,:] = PQ[:,]
49 Q[0,:] = PQ[:,]
50
51 B3 = segOrd(P, M, W)
52 B4 = segOrd(Q, M, W)
53
54 # Segundo Plot
55 ax[1].plot(B3[:,0], B3[:,1])
56 ax[1].plot(B4[:,0], B4[:,1])
57 ax[1].set_title('Suavizado')
58 ax[1].grid()
59
60
```

Figura 7.9: Implementaci髇 de concatenaci髇 de curvas de b茅zier

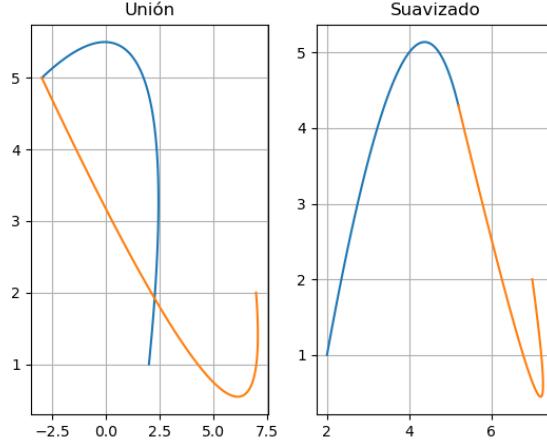


Figura 7.10: Concatenaci髇 de curva de b茅zier de segundo orden

En la figura 7.11 se muestra la concatenaci髇 de dos curvas de b茅zier. La primer curva es cuadr谩tica y la segunda curva es c煤bica, adem醩, aplicando una variable de suavizado, se observa como cambia la direcci髇 de la curva.

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17
18
19 def tercerOrden():
20     M2 = np.array([[-1,3,-3,1],[3,-6,3,0],[-3,3,0,0],[1,0,0,0]])
21     W2 = np.zeros((len(t),4), dtype=float)
22     U = np.ones((len(t),1), dtype=float)
23     W2[:,0] = t*t*t
24     W2[:,1] = t*t
25     W2[:,2] = t*1
26     W2[:,3] = U[:,0]
27     return M2, W2
28
29 #Puntos para la curva
30 P = np.array([[2,1],[4,7],[-3,5],[6,12]],dtype=float)
31 Q = np.array([[6,12],[8,-2],[7,2],[5,4]],dtype=float)
32 t = np.arange(0,1.01,0.01)
33
34
35 #Puntos de P
36 M1,W1 = tercerOrden()
37 B1 = M1@P
38 B = W1@B1
39
40 B2 = M1@Q
41 C = W1@B2
42 plt.figure()
43 plt.subplot(121)
44 plt.plot(B[:,0], B[:,1])
45 plt.plot(C[:,0], C[:,1])
46
47 #Concatenaci髇 de curvas
48 alpha = 0.8
49 PQ0 = (1-alpha)*P[1,0] + alpha*Q[1,0]
50 PQ1 = (1-alpha)*P[1,1] + alpha*Q[1,1]
51
52 #Actualizaci髇
53 P[2,0] = PQ0
54 P[2,1] = PQ1
55 Q[0,0] = PQ0
56 Q[0,1] = PQ1
57
58
59 B1 = M1@P
60 B = W1@B1
61
62 B2 = M1@Q
63 C = W1@B2
64
65
66 plt.subplot(122)
67 plt.plot(B[:,0], B[:,1])
68 plt.plot(C[:,0], C[:,1])
69
70
71 plt.show()

```

Figura 7.11: Implementaci髇 de concatenaci髇 de curvas de b茅zier

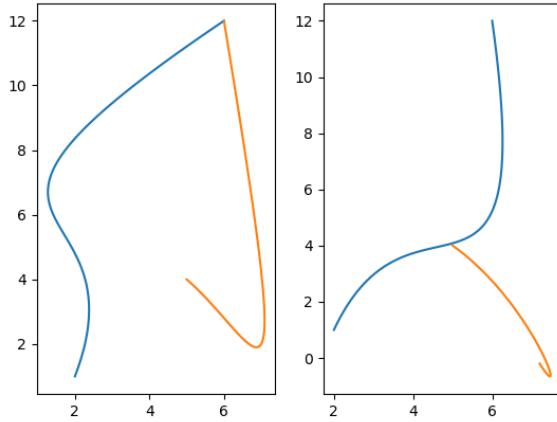


Figura 7.12: Concatenaci髇 de curva de b茅zier de segundo y tercero orden

En la figura 7.13 se muestra la concatenaci髇 de dos curvas de b茅zier c醙icas, adem醘s de aplicar una variable de suavizado que permite observar como cambia la direcci髇 de la curva.

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17
18 def segundoOrden():
19     M1 = np.array([[1,-2,1],[-2,2,0],[1,0,0]],dtype=float)
20     U = np.ones((len(t),1), dtype=float)
21     W1 = np.zeros((len(t),3), dtype=float)
22     W1[:,0] = t
23     W1[:,1] = t**2
24     W1[:,2] = U[:,0]
25     return M1, W1
26
27 def tercerOrden():
28     M2 = np.array([-1,3,-3,1],[3,-6,3,0],[-3,3,0,0],[1,0,0,0])
29     W2 = np.zeros((len(t),4), dtype=float)
30     U = np.ones((len(t),1), dtype=float)
31     W2[:,0] = t***t
32     W2[:,1] = t**2
33     W2[:,2] = t**1
34     W2[:,3] = U[:,0]
35     return M2, W2
36
37 #Puntos para la curva
38 P = np.array([-2,1],[4,7],[-3,5]), dtype=float)
39 Q = np.array([-3,5],[8,-2],[7,2],[5,4]), dtype=float)
40 t = np.arange(0,1.01,0.01)
41
42 #Puntos de P
43 M1,W1 = segundoOrden()
44 B1 = M1@P
45
46 B = W1@B1
47 #Puntos de Q
48 M2,W2 = tercerOrden()
49 B2 = M2@Q
50 C = W2@B2
51 plt.figure()
52 plt.subplot(121)
53 plt.plot(B[:,0], B[:,1], C[:,0], C[:,1])
54
55 #Concatenaci髇 de curvas
56 alpha = 0.8
57 PQ0 = (1-alpha)*P[1,0] + alpha*Q[1,0]
58 PQ1 = (1-alpha)*P[1,1] + alpha*Q[1,1]
59
60 #Actualizaci髇
61 P[2,0] = PQ0
62 P[2,1] = PQ1
63 Q[0,0] = PQ0
64 Q[0,1] = PQ1
65
66 B1 = M1@P
67 B = W1@B1
68
69 B2 = M2@Q
70 C = W2@B2
71 plt.plot(B[:,0], B[:,1], C[:,0], C[:,1])
72
73 plt.show()

```

Figura 7.13: Implementaci髇 de concatenaci髇 de curvas de b茅zier

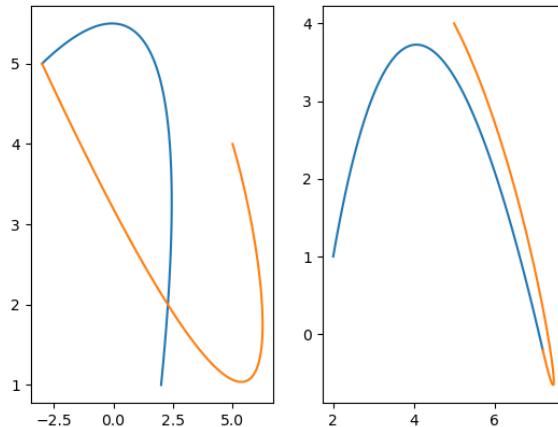


Figura 7.14: Concatenaci髇 de curva de b茅zier de tercero orden

## 7.7 Parametrización de curvas de Bézier

Las curvas de b茅zier son definidas por partes de la siguiente manera:

$$B(t) = \left\{ \begin{array}{l} B_0(t) : t_0 \leq t \leq t_1 \\ B_1(t) : t_1 \leq t \leq t_2 \\ \dots \\ B_n(t) : t_{n-1} \leq t \leq t_n \end{array} \right\}$$

Por lo que, se obtiene la siguiente estructura:

$$B_n(t) = \sum b_i B_{ij} \left( \frac{t - t_{min}}{t_{max} - t_{min}} \right)$$

### Ejercicio 7.6.1

Dadas las curvas de b茅zier

$$B_0(t) \rightarrow t \in [-2, 0] : b_{00}(2, -1), b_{01}(5, 2), b_{02}(7, 3), b_{03}(8, -1)$$

$$B_1(t) \rightarrow t \in [0, 3] : b_{10}(8, -1), b_{11}(8, -3), b_{12}(7, -4), b_{13}(5, -4)$$

$$B_2(t) \rightarrow t \in [3, 4] : b_{00}(5, -4), b_{01}(3, -4), b_{02}(4, -2), b_{03}(6, -2)$$

La curva de b茅zier es:

$$B(t) = \left\{ \begin{array}{l} B_0(t) : -2 \leq t \leq 0 \\ B_1(t) : 0 \leq t \leq 3 \\ B_2(t) : 3 \leq t \leq 4 \end{array} \right\}$$

Reparametrizando la variable  $t$ , se obtiene lo siguiente:

$$B_0(t) \rightarrow \frac{t - (-2)}{0 - (-2)} = \frac{t + 2}{2} = \frac{t}{2} + 1$$

$$B_0(t) \rightarrow \frac{t - (-0)}{3 - (0)} = \frac{t}{3}$$

$$B_0(t) \rightarrow \frac{t - (-3)}{4 - (-3)} = t - 3$$

Para calcular los puntos de las curvas de b茅zier

$$\begin{aligned} & \binom{3}{0} \left(\frac{t}{2}\right)^3 b_{00} + \binom{3}{1} \left(\frac{t}{2}\right)^2 \left(\frac{t}{2} + 1\right) b_{01} + \binom{3}{2} \left(\frac{t}{2}\right) \left(\frac{t}{2} + 1\right)^2 b_{02} + \binom{3}{3} \left(\frac{t}{2} + 1\right)^3 b_{03} \\ & \binom{3}{0} \left(1 - \left(\frac{t}{3}\right)\right)^3 b_{10} + \binom{3}{1} \left(1 - \left(\frac{t}{3}\right)\right)^2 \left(\frac{t}{3}\right) b_{11} + \binom{3}{2} \left(1 - \left(\frac{t}{3}\right)\right) \left(\frac{t}{3}\right)^2 b_{12} + \binom{3}{3} \left(\frac{t}{3}\right)^3 b_{13} \\ & \binom{3}{0} (4+t)^3 b_{20} + \binom{3}{1} (4+t)^2 (t-3) b_{21} + \binom{3}{2} (4+t)(t-3) b_{22}^2 + \binom{3}{3} (t-3)^3 b_{23} \end{aligned}$$

# Capítulo 8

## Superficies

Las superficies tienen un papel fundamental en aplicaciones como los gráficos por ordenador la realidad virtual, los juegos de ordenador y el diseño asistido por ordenador de coches, barcos aviones y edificios. Un subconjunto de  $\mathbb{R}^3$  de la forma  $\{(x, y, z) : F(x, y, z) = 0\}$  para alguna función  $F : \mathbb{R}^3 \rightarrow \mathbb{R}$  se llama función implícita. Cuando  $F$  es un polinomio en  $x, y$  y  $z$ , la superficie se llama superficie algebraica. Si las derivadas parciales de  $F$  existen entonces los puntos de la superficie satisfacen

$$F(x, y, z) = \frac{\partial F}{\partial x}(x, y, z) = \frac{\partial F}{\partial y}(x, y, z) = \frac{\partial F}{\partial z}(x, y, z) = 0$$

Se denominan puntos singulares, y todos los demás puntos se denominan no singulares o regulares puntos.

### 8.1 Superficies paramétricas

Sea  $U$  un subconjunto de  $\mathbb{R}^2$ , una superficie paramétrica es mapeada  $S : U \rightarrow \mathbb{R}^3$ . Un mapeo  $S : V \rightarrow \mathbb{R}^3$ , definida sobre un subconjunto cerrado  $V$  de  $\mathbb{R}^2$  se dice que es una superficie paramétrica siempre que exista un subconjunto abierto  $U$  que contenga a  $V$ , y una superficie parámetrica  $S : U \rightarrow \mathbb{R}^3$ , tal que  $S(s, t) = S_1(s, t)$  para todo  $(s, t) \in V$ .  $S_1$  se dice que extiende de  $S$ . El subconjunto  $S = S(U)$  o  $S = S(V)$  de  $\mathbb{R}^3$  se denomina como la superficie  $S$ .

Las coordenadas de un punto arbitrario de una superficie paramétrica  $S$  pueden ser expresarse como funciones de dos variables, por ejemplo,

$$S(s, t) = (x(s, t), y(s, t), z(s, t))$$

Un ejemplo, es la parametrización de una esfera, La ecuación de la esfera de radio  $a$  es  $x^2 + y^2 + z^2 = a^2$ , puede parametrizarse usando los ángulos  $\phi$  y  $\theta$ . Donde  $0 \leq \theta \leq \pi$ ,  $0 \leq \phi \leq 2\pi$ .

$$x = a \sin(\theta) \cos(\phi)$$

$$y = a \sin(\theta) \sin(\phi)$$

$$z = a \cos(\theta)$$

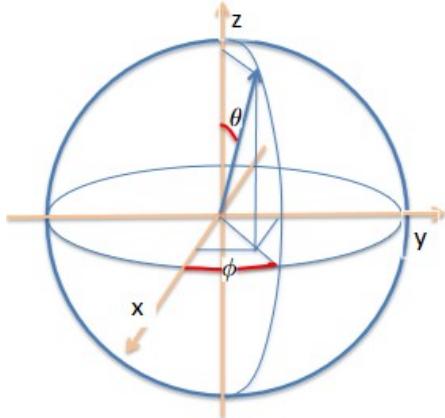


Figura 8.1: Parametrización de esfera

### Ejercicio

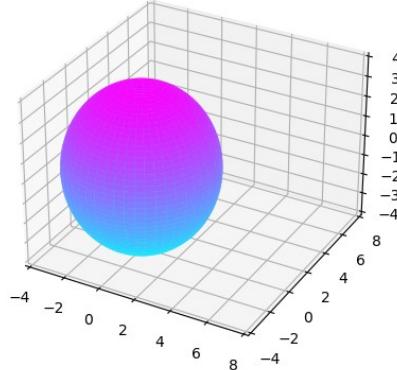
Implementar un código en python que muestre una esfera, de acuerdo a la ecuación parametrizada anteriormente.

```

16  from numpy import *
17  import matplotlib.pyplot as plt
18  from matplotlib import cm
19
20  u = linspace(0,(2*pi)+0.1, 100)
21  v = linspace(-pi/2, pi/2,100)
22
23 U,V = meshgrid(u,v);
24
25 R = 4
26 fig = plt.figure()
27 ax = fig.add_subplot(111, projection='3d')
28 ax.axis([-4,8,-4,8])
29 X = R* cos(U) * cos(V)
30 Y = R* sin(U) * cos(V)
31 Z = R* sin(V)
32 ax.plot_surface(X,Y,Z, cmap=cm.cool)
33 plt.pause(0.1)
34 plt.show()

```

(a) Implementación de código



(b) Esfera parametrizada

Figura 8.2: Parametrización de esfera

### Ejercicio

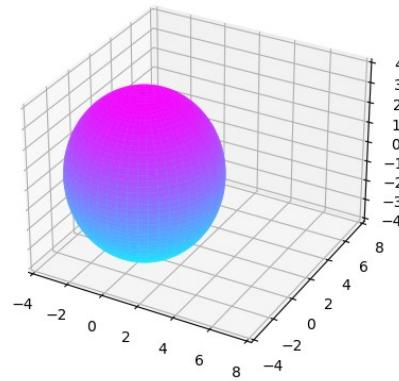
Graficar una esfera con radio de 4 unidades, utilizando sus ecuaciones paramétrica.

```

16 from numpy import *
17 import matplotlib.pyplot as plt
18 from matplotlib import cm
19
20 u = linspace(0,(2*pi)+0.1, 100)
21 v = linspace(-pi/2, pi/2,100)
22
23 U,V = meshgrid(u,v);
24
25 R = 4
26 fig = plt.figure()
27 ax = fig.add_subplot(111, projection='3d')
28 ax.axis([-4,8,-4,8])
29 X = R* cos(U) * cos(V)
30 Y = R* sin(U) * cos(V)
31 Z = R* sin(V)
32 ax.plot_surface(X,Y,Z, cmap=cm.cool)
33 plt.pause(0.1)
34 plt.show()

```

(a) Implementación de código



(b) Esfera

Figura 8.3: Esfera

### Ejercicio

Realizar una animación donde la esfera se translade sobre un eje.

```

14 from numpy import *
15 import matplotlib.pyplot as plt
16 from mpl_toolkits.mplot3d import axes3d
17 from matplotlib import cm
18
19 u = linspace(0,(2*pi)+0.1, 100)
20 v = linspace(-pi/2, pi/2,100)
21
22 U,V = meshgrid(u,v);
23
24 R = 3
25 fig = plt.figure()
26 ax = fig.add_subplot(111, projection='3d')
27 ax.axis([-4,8,-4,8])
28
29 for i in range(1,10):
30     X = R* cos(U) * cos(V)*(i/10)
31     Y = R* sin(U) * cos(V)*(i/10)
32     Z = R* sin(V)
33     ax.plot_surface(X,Y,Z, cmap=cm.cool)
34     plt.pause(0.1)
35     plt.show()

```

Figura 8.4: Animación de la esfera

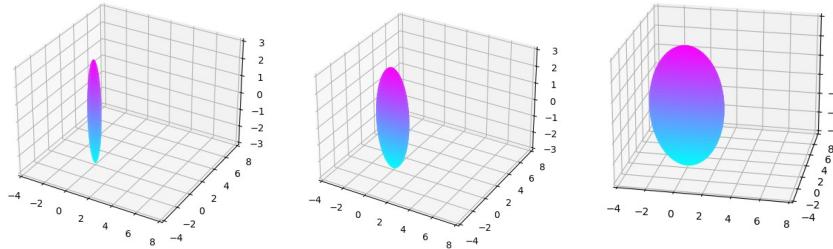


Figura 8.5: Animación de la esfera

### Ejercicio

Graficar helicoide, utilizando sus ecuaciones paramétricas.

$$x = R * v * \cos(u)$$

$$y = R * v * \sin(u)$$

$$z = b * u$$

```

15 from numpy import *
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 fig = plt.figure()
20 ax = fig.add_subplot(111, projection='3d')
21 v = arange(-pi/2, pi/2, 0.1)
22 u = arange(0, 2*pi, 0.1)
23 R = 3
24 b = 0.5
25 U,V = meshgrid(u,v)
26
27 ax.plot_surface(R * V* cos(U), R * V* sin(U), b * U, rstride=1,
28                  cstride=1, cmap=cm.PuRd)
29 ax.axis('off')
30 ax.set_title("Helicoide")
31 plt.show()
32

```

(a) Implementación de código

Helicoide



(b) Helicoide

Figura 8.6: Helicoide

### Ejercicio

Realizar una animación sobre un helicoide, donde una o varias de sus ecuaciones parámetricas sea modificada.

```

15 from numpy import *
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 fig = plt.figure()
20 ax = fig.add_subplot(111, projection='3d')
21 v = arange(-pi/2, pi/2, 0.1)
22 u = arange(0, 2*pi, 0.1)
23 R = 3
24 b = 0.5
25 U,V = meshgrid(u,v)
26 ax.axis([-10,10,-10,10])
27 for i in range(20):
28     ax.plot_surface(R * V* cos(U), R * V* sin(U*i),
29                      b * U*1, rstride=1, cstride=1, cmap=cm.cool)
30     plt.axis('off')
31     plt.pause(0.01)
32     plt.show()

```

Figura 8.7: Animación del helicoide



Figura 8.8: Animación del helicoide

### Ejercicio

Graficar un toroide, utilizando sus ecuaciones paramétricas.

$$\begin{aligned}x &= (R + r \cos(u)) \cos(v) \\y &= (R + r \cos(u)) \sin(v) \\z &= r \sin(u)\end{aligned}$$

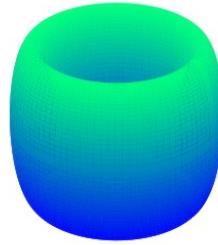
```

16 from numpy import *
17 from mpl_toolkits.mplot3d import axes3d
18 import matplotlib.pyplot as plt
19 from matplotlib import cm
20
21 fig = plt.figure()
22 ax = fig.add_subplot(111, projection='3d')
23 v = linspace(0,(2*pi)+0.1,100)
24 u = linspace(0,(2*pi)+0.1,100)
25 R = 15
26 r = 4
27 U,V = meshgrid(u,v)
28 X = (R+r*cos(U))*cos(V)
29 Y = (R+r*cos(U))*sin(V)
30 Z = R*sin(U)
31 ax.plot_surface(X,Y,Z, rstride=1, cstride=1, cmap=cm.winter)
32 ax.axis('off')
33 ax.set_title('Toroide')
34 plt.show()

```

(a) Implementación de código

Toroide



(b) Toroide

Figura 8.9: Toroide

**Ejercicio**

Realizar una animación sobre un toroide, donde una o varias de sus ecuaciones parámetricas sea modificada.

```

14 from numpy import *
15 from mpl_toolkits.mplot3d import axes3d
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 fig = plt.figure()
20 ax = fig.add_subplot(111, projection='3d')
21 v = linspace(0,(2*pi)+0.1,100)
22 u = linspace(0,(2*pi)+0.1,100)
23 R = 15
24 r = 4
25 U,V = meshgrid(u,v)
26 ax.axis([-20,20,-10,20])
27 for i in range(20):
28     X = (R+r*cos(U*i/2))*cos(V)
29     Y = (R+r*cos(U*i/5))*sin(V)
30     Z = R*sin(U)
31     ax.plot_surface(X,Y,Z, rstride=1, cstride=1, cmap=cm.cool)
32     plt.axis('off')
33     plt.show()
34     plt.pause(0.01)

```

Figura 8.10: Animación del toroide

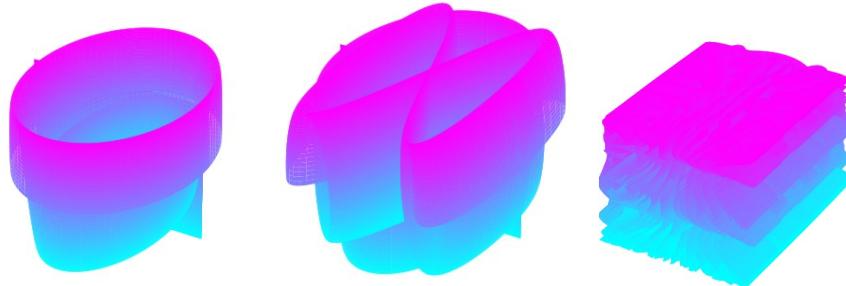


Figura 8.11: Animación del toroide

**Ejercicio**

Graficar la banda de Möbius, utilizando sus ecuaciones paramétricas.

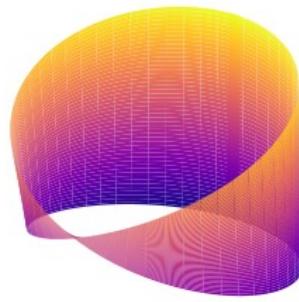
$$\begin{aligned}x &= (1 + v \sin(\frac{u}{2})) \sin(u) \\y &= (1 + v \sin(\frac{u}{2})) \cos(u) \\z &= v \cos(\frac{u}{2})\end{aligned}$$

```

9 import matplotlib.pyplot as plt
10 import numpy as np
11 from matplotlib import cm
12
13 u = np.linspace(0, (2 * np.pi) + 0.1, 100)
14 v = np.linspace(-1, 1, 100)
15
16 R = 18
17
18 U, V = np.meshgrid(u, v)
19
20 fig = plt.figure()
21 ax = fig.add_subplot(111, projection='3d')
22
23
24 X = (R - V * np.sin(U/2)) * np.sin(U)
25 Y = (R - V * np.sin(U/2)) * np.cos(U)
26 Z = V * np.cos(U/2)
27 plt.axis("off")
28
29 ax.plot_surface(X,Y,Z, cmap = cm.plasma)
30 ax.set_title("Möbius")

```

(a) Implementación de código



(b) banda de Möbius

Figura 8.12: banda de Möbius

### Ejercicio

Realizar una animación sobre la banda de Möbius, donde una o varias de sus ecuaciones parámetricas sea modificada.

```

9 import matplotlib.pyplot as plt
10 import numpy as np
11 from matplotlib import cm
12
13 u = np.linspace(0, (2 * np.pi) + 0.1, 100)
14 v = np.linspace(-1, 1, 100)
15
16 R = 18
17
18 U, V = np.meshgrid(u, v)
19
20 fig = plt.figure()
21 ax = fig.add_subplot(111, projection='3d')
22
23
24 X = (R - V * np.sin(U/2)) * np.sin(U)
25 Y = (R - V * np.sin(U/2)) * np.cos(U)
26 Z = V * np.cos(U/2)
27 plt.axis("off")
28
29 ax.plot_surface(X,Y,Z, cmap = cm.plasma)
30 ax.set_title("Möbius")
31 plt.pause(1)
32 #plt.clf()
33
34 for i in range(10):
35     X = (R - V * np.sin(U/2)) * np.sin(U) * i
36     Y = (R - V * np.sin(U/2)) * np.cos(U)
37     Z = V * np.cos(U/2)
38
39
40     ax.plot_surface(X,Y,Z, cmap = cm.plasma)
41     ax.set_title("Möbius")
42     plt.pause(1)

```

Figura 8.13: Animación de la banda de Möbius

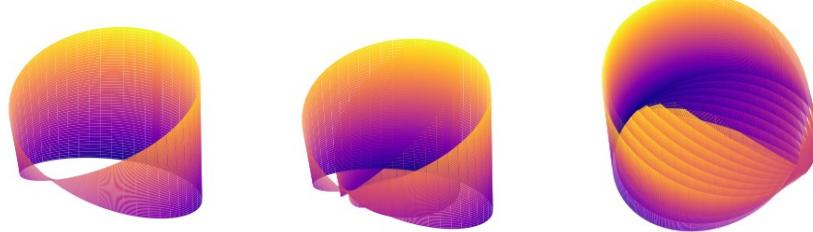


Figura 8.14: Animación de la banda de Möbius

## 8.2 Superficies por deslizamiento(swapping)

Consideremos una transformación tridimensional no singular  $T(t)$  que depende de un parámetro  $t$ , es decir, las entradas de la matriz de transformación son funciones continuas de  $t$ . La construcción de una superficie se obtiene aplicando  $T(t)$  a una curva generadora especificada de modo que al variar el parámetro  $t$ , la curva se desplaza por el espacio y, por lo tanto, “barre” una superficie. En general, una superficie construida de esta manera es retorcida y se auto-interseca, y no sirve en la práctica. Sin embargo, algunas elecciones particulares de la curva generadora y de la transformación de la curva generadora y de la transformación dan lugar a formas de superficie útiles.

La construcción de la superficie puede generalizarse para dar una superficie de barrido traslacional obtenida mediante la translación de una curva generadora  $B(s)$  y  $C(t)$  donde ambas son curvas de Bézier.

### Ejercicio

Realiza una superficie aplicando desplazamiento a una curva de Bézier de segundo orden, observa la superficie que se forma.

```

15 import matplotlib.pyplot as plt
16 import numpy as np
17
18 print("===== Curvas de Bezier =====")
19 print("===== SWAPPING =====")
20 fig = plt.figure(111, projection='3d')
21 ax = fig.add_subplot(111, projection='3d')
22 plt.axis([0,25,-10,35])
23 t = np.arange(0,1,0.01)
24 U = np.ones((len(t),1), dtype=float)
25 B = np.zeros((len(t),3), dtype=float)
26 Bl = np.zeros((4,3), dtype=float)
27 M = np.array([-1,1,-3,1],[3,-6,3,0],[-3,3,0,0],[1,0,0,0]) 
28 P = np.array([3,7,4],[3,-6,8],[12,5,4],[4,2,9])
29
30 W = np.zeros((len(t),4), dtype=float)
31 W[:,0] = t**t
32 W[:,1] = t**t
33 W[:,2] = t[:,]
34 W[:,3] = U[:,0]
35
36 plt.axis([-3,20,0,20])
37 print(P)
38 Bl = M@P
39 B = W@Bl
40
41 for i in range(50):
42     ax.plot( Bl[:,0]+(i/2), Bl[:,1]+(i/3), Bl[:,2]+(i/3) )
43     plt.axis('off')
44     plt.pause(0.5)

```

Figura 8.15: Implementación de código

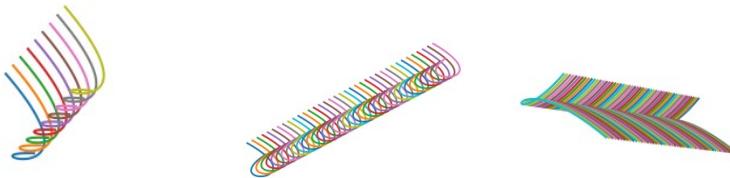


Figura 8.16: Superficie de desplazamiento

## 8.3 Superficies por rotación (superficies de revolución)

La superficie obtenida al girar una curva generadora  $B(s)$  alrededor de un eje fijo se denomina superficie de revolución. Se propone que la curva se encuentra en un plano que contiene el eje y, para evitar las auto-intersecciones de la superficie que el eje no intersecta la curva.

$$\text{Sea } B(s) = \sum_{i=0}^n b_i u_i N_{i,d}(s)$$

Las superficies de revolución son figuras que se forman al girar  $360^\circ$  una línea recta o una curva contenida en un plano, llamada generatriz, alrededor de un eje de rotación, contenido también en el mismo plano. Las superficies regladas de curvatura simple se generan cuando su generatriz se desliza

siempre en contacto con otra línea curva, llamada directriz cumpliendo unas condiciones. Una de ellas es que cualquier par de generatrices contiguas pertenezcan al mismo plano.

Cuando la directriz es una circunferencia perpendicular a las generatrices estaremos en el caso particular de las superficies de revolución.

Ejemplos básicos de superficies de revolución son las superficies cilíndricas, cónicas, esféricas y toroidales.

### Ejercicio

Realiza una superficie aplicando rotación a una curva de Bézier, observa la superficie que se forma.

```

15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 print("===== Curvas de Bezier =====")
19 print("===== POR ROTACION =====")
20 fig = plt.figure()
21 ax = fig.add_subplot(111, projection='3d')
22 plt.axis([-25, 25, -35, 35])
23 t = np.arange(0, 1, 0.01)
24 U = np.ones((len(t),1), dtype=float)
25 U = np.ones((len(t),1), dtype=float)
26 B = np.zeros((len(t),3), dtype=float)
27 B1 = np.zeros((4,3), dtype=float)
28 P = np.array([[-3,0,0],[3,0,0],[0,3,0],[0,-3,0]])
29 P = np.array([[2,5,3],[10,4,8],[9,1,11],[4,2,9]]).T
30
31 W = np.zeros((len(t),4), dtype=float)
32 W[:,0] = t**t
33 W[:,1] = t**t
34 W[:,2] = t**t
35 W[:,3]= U[:,0]
36
37 B1 = M@P
38 B = W@B1
39
40 for i in range(360):
41     th = float((np.pi/180)*i)
42     T = np.array([[ np.sin(th),np.cos(th),0, [-np.sin(th), np.cos(th),0, [1,0,0] ] ] )
43     B2 = B@T
44     ax.plot(B2[:,0], B2[:,1], B2[:,2])
45     plt.axis([-3,20,0,15])
46     plt.pause(0.5)
47
48

```

Figura 8.17: Implementación de código



Figura 8.18: Superficie por rotación

### Ejercicio

Realiza una superficie aplicando rotación a una curva de Bézier de segundo orden, observa la superficie que se forma.

```

15 import numpy as np
16 import matplotlib.pyplot as plt
17
18 print("===== Curvas de Bezier =====")
19 print("===== POR ROTACION =====")
20 fig = plt.figure()
21 ax = fig.add_subplot(111, projection='3d')
22 plt.axis([-25, 25, -35, 35])
23 t = np.arange(0, 1, 0.01)
24 U = np.ones((len(t),1), dtype=float)
25 U = np.ones((len(t),1), dtype=float)
26 B = np.zeros((len(t),3), dtype=float)
27 B1 = np.zeros((4,3), dtype=float)
28 P = np.array([[-3,0,0],[3,0,0],[0,3,0],[0,-3,0]])
29 P = np.array([[2,5,3],[10,4,8],[9,1,11],[4,2,9]]).T
30
31 W = np.zeros((len(t),4), dtype=float)
32 W[:,0] = t**t
33 W[:,1] = t**t
34 W[:,2] = t**t
35 W[:,3]= U[:,0]
36
37 B1 = M@P
38 B = W@B1
39
40 for i in range(360):
41     th = float((np.pi/180)*i)
42     T = np.array([[ np.sin(th),np.cos(th),0, [-np.sin(th), np.cos(th),0, [1,0,0] ] ] )
43     B2 = B@T
44     ax.plot(B2[:,0], B2[:,1], B2[:,2])
45     plt.axis([-3,20,0,15])
46     plt.pause(0.5)
47
48

```

Figura 8.19: Implementación de código



Figura 8.20: Superficie por rotación en curva de segundo Orden

## 8.4 Curvas sobre superficies

Una forma habitual de representar gráficamente los valores de una función de 2 variables, es dibujando sus líneas o curvas de nivel. Para trazar la gráfica de una función de dos variables, en el espacio, es de gran ayuda construir un mapa compuesto por un conjunto de curvas, que permiten imaginar de alguna manera, la superficie. Dado una superficie  $S(u, w)$  con  $a \leq u \leq b$  y  $c \leq w \leq d$ . Si fijamos  $u = u_0$  y  $w = w_0$ , tanto  $S(u_0, w)$  y  $S(u, w_0)$  son curvas sobre  $S(u, w)$ .

Cada curva de nivel  $N_k(f)$  determina una curva en la superficie  $z = f(x, y)$ , paralela a la curva de nivel, a una distancia  $k$  del plano  $XY$ .

### Ejercicio

Implementar un programa donde se fija uno de los puntos de la esfera  $S(u, w)$ , para obtener una curva sobre esta superficie.

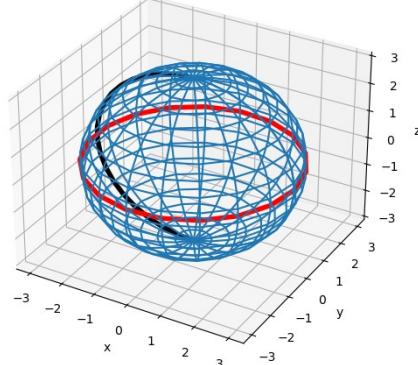
```

14 from numpy import cos, sin, linspace, pi, meshgrid
15 import matplotlib.pyplot as plt
16
17 u = linspace(0, 2*pi, 20)
18 v = linspace(0, pi, 20)
19 a = 3
20
21 U,V = meshgrid(u,v)
22 X = a*cos(U)*sin(V)
23 Y = a*sin(U)*sin(V)
24 Z = a*cos(V)
25 v0 = pi/2
26 u0 = pi
27
28 x = a*cos(u0)*sin(v)
29 y = a*sin(u0)*sin(v)
30 z = a*cos(v)
31
32 x1 = a*cos(u)*sin(v0)
33 y1 = a*sin(u)*sin(v0)
34 z1 = a*cos(v0)
35
36 fig = plt.figure(figsize=[6,6])
37 ax = fig.add_subplot(projection='3d')
38 ax.set_title('Esfera')
39 ax.set_xlabel('x')
40 ax.set_ylabel('y')
41 ax.set_zlabel('z')
42
43 ax.plot3D(x,y,z,linewidth=4, color='black')
44 ax.plot3D(x1,y1,z1,linewidth=4, color='red')
45 ax.plot_wireframe(X,Y,Z)
46 plt.show()

```

(a) Implementación de código

Esféra



(b) Curva sobre la esfera

Figura 8.21: Curva sobre la esfera

### Ejercicio

Implementar un programa donde se fija uno de los puntos de la superficie helicoide  $S(u, w)$ , para obtener una curva sobre esta superficie.

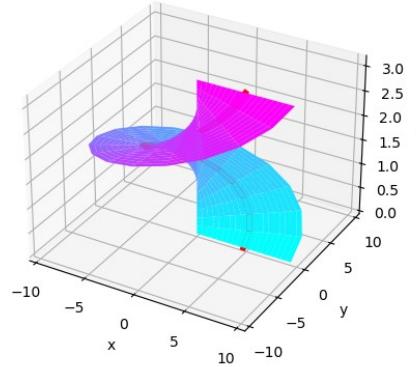
```

15 from numpy import sin, cos, linspace, pi, meshgrid
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 u = linspace(0,2*pi,20)
20 v = linspace(0,pi,20)
21 R = 3
22 b = 0.5
23 v0 = pi/2
24 u0 = pi
25
26 U,V = meshgrid(u,v)
27
28 X = R * V**cos(U)
29 Y = R * V**sin(U)
30 Z = b * U
31
32 x = R* u**cos(u0)
33 y = R* v**sin(u0)
34 z = b* u0
35
36 xl = R* v0**cos(u)
37 yl = R* v0**sin(u)
38 zl = b* u
39
40 fig = plt.figure()
41 ax = fig.add_subplot(projection='3d')
42 ax.set_title('Esfera')
43 ax.set_xlabel('x')
44 ax.set_ylabel('y')
45 ax.set_zlabel('z')
46
47 ax.plot3D(x,y,z,linewidth=4, color='black')
48 ax.plot3D(xl,yl,zl,linewidth=4, color='red')
49 ax.plot_surface(X,Y,Z, rstride=1, cstride=1, cmap=cm.cool)
50 plt.show()

```

(a) Implementación de código

Curva sobre helicoide



(b) Curva sobre helicoide

Figura 8.22: Curva sobre helicoide

### Ejercicio

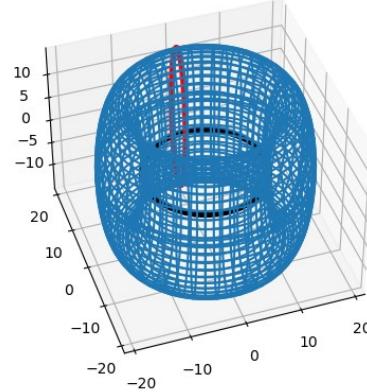
Implementar un programa donde se fija uno de los puntos del toroide  $S(u, w)$ , para obtener una curva sobre esta superficie.

```

16 from numpy import linspace, meshgrid, cos, sin, pi
17 import matplotlib.pyplot as plt
18
19 fig = plt.figure()
20 ax = fig.add_subplot(projection='3d')
21 v = linspace(0,(2*pi)+0.1,100)
22 u = linspace(0,(2*pi)+0.1,100)
23 R = 15
24 r = 4
25 U,V = meshgrid(u,v)
26 X = (R+r*cos(U))*cos(V)
27 Y = (R+r*cos(U))*sin(V)
28 Z = R*sin(U)
29
30 v0 = pi/2
31 u0 = pi
32
33 x = (R+r*cos(u0))*cos(v)
34 y = (R+r*cos(u0))*sin(v)
35 z = R*sin(u0)
36
37 xl = (R+r*cos(u))*cos(v0)
38 yl = (R+r*cos(u))*sin(v0)
39 zl = R*sin(u)
40
41 #ax.plot_surface(X,Y,Z, rstride=1, cstride=1, cmap=cm.cool)
42 ax.plot3D(v,y,z,linewidth=4, color='black')
43 ax.plot_wireframe(X,Y,Z)
44 ax.plot3D(xl,yl,zl,linewidth=4, color='red')
45 plt.show()

```

(a) Implementación de código



(b) Curva sobre el toroide

Figura 8.23: Curva sobre el toroide

## 8.5 Superficies de Bézier

La manera más sencilla de construir una superficie consiste en barrer una curva en el espacio, en la representación de Bézier. Los puntos de control de esta curva se mueven a su vez siguiendo curvas de Bézier, cuyos puntos de control definen la superficie. La representación de la superficie por medio de estos puntos de control tiene propiedades similares a la representación de Bézier univariada. Por esta razón se puede trabajar con estas superficies aplicando los algoritmos para curvas. También, se pueden construir volúmenes multidimensionales barriendo una superficie o un volumen en el espacio de manera que sus puntos de control se mueven a lo largo de diversas curvas. Análogamente se obtienen mallas de control con propiedades similares de las de representaciones de curvas.

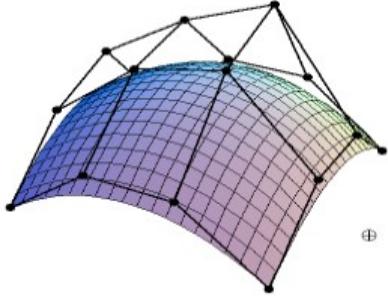


Figura 8.24: Ejemplo de superficie de Bézier

### 8.5.1 Productos cartesianos

Desarrollando el producto cartesiano de dos curvas de b茅zier de segundo orden, se obtiene lo siguiente:

Siendo  $B(u) = \sum_{i=0}^2 \binom{2}{i} (1-u)^{2-i} u^i$  y  $B(w) = \sum_{j=0}^2 \binom{2}{j} (1-w)^{2-j}$ , el producto cartesiano es

$$B(u) \times B(w) = \left( \sum_{i=0}^2 \binom{2}{i} (1-u)^{2-i} u^i \right) \left( \sum_{j=0}^2 \binom{2}{j} (1-w)^{2-j} \right)$$

$$B(u) \times B(w) = \left[ \binom{2}{0} (1-u)^2 + \binom{2}{1} (1-u)u + \binom{2}{2} u^2 \right] \left[ \binom{2}{0} (1-w)^2 + \binom{2}{1} (1-w)w + \binom{2}{2} w^2 \right]$$

$$B(u) \times B(w) = [(1-u)^2 + 2(1-u)u + u^2][(1-w)^2 + 2(1-w)w + w^2]$$

$$\begin{aligned} B(u) \times B(w) &= (1-u)^2 [(1-w)^2 + 2(1-w)w + w^2] + 2(1-u)u [(1-w)^2 + 2(1-w)w + w^2] \\ &\quad + u^2 [(1-w)^2 + 2(1-w)w + w^2] \end{aligned}$$

$$\begin{aligned} B(u) \times B(w) &= (1-u)^2 (1-w)^2 + 2w(1-u)^2 (1-w) + w^2 (1-u)^2 + 2u(1-u)(1-w)^2 \\ &\quad + 4uw(1-u)(1-w) + 2uw^2(1-u) + u^2(1-w)^2 + 2wu^2(1-w) + w^2u^2 \end{aligned}$$

Evaluando en  $\begin{bmatrix} (0,0,0) & (0,0,1) & (0,0,2) \\ (1,0,0) & (1,1,1) & (1,2,2) \\ (2,0,0) & (2,1,0) & (2,2,2) \end{bmatrix}$  se obtiene finalmente.

$$\begin{aligned} B(u) \times B(w) &= (1-u)^2 (1-w)^2 P_{00} + 2w(1-u)^2 (1-w) P_{01} + w^2 (1-u)^2 P_{02} + 2u(1-u)(1-w)^2 P_{10} \\ &\quad + 4uw(1-u)(1-w) P_{11} + 2uw^2(1-u) P_{12} + u^2(1-w)^2 P_{20} \\ &\quad + 2wu^2(1-w) P_{21} + w^2u^2 P_{22} \end{aligned}$$

Implementando el producto cartesiano en python, se obtiene el resultado de la figura 8.26

```

15 from numpy import *
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 P = array([[0,0,0],[1,0,1],[0,0,1],
20           [1,1,0],[1,1,1],[1,1,2],
21           [0,2,0],[1,2,1],[0,2,2]] , dtype=float)
22
23 U = linspace(0,1,50)
24 W = linspace(0,1,50)
25
26 B0 = zeros((50,50), dtype = float)
27 B1 = zeros((50,50), dtype = float)
28 B2 = zeros((50,50), dtype = float)
29
30 u,w = meshgrid(U,W)
31
32 B0 = ((1-u)**3)*(1-w)**3*B
33 B0 = ((1-u)**2)*(1-w)**2*(w**2)*B[1,0] + ( 2*w*(1-u)**2*(1-w)*w*B[1,0] ) + ((1-u)**2*(w**2)*B[2,0]) + ( 2*(1-u)*u*(1-w)**2*B[3,0]* 4*w*(1-u)*u*(1-w)*B[4,0] ) +
34 B1 = ((1-u)**2)*(1-w)**2*(w**2)*B[0,1] + ( 2*w*(1-u)**2*(1-w)*w*B[1,1] ) + ((1-u)**2*(w**2)*B[2,1]) + ( 2*(1-u)*u*(1-w)**2*B[3,1]* 4*w*(1-u)*u*(1-w)*B[4,1] ) +
35 B2 = ((1-u)**2)*(1-w)**2*B[0,2] + ((1-u)**2*(w**2)*B[1,2]) + ( 2*(1-u)*u*(1-w)**2*B[2,2]) + ( 2*(1-u)*u*(1-w)**2*B[3,2]* 4*w*(1-u)*u*(1-w)*B[4,2] ) +
36
37
38 fig = plt.figure()
39 ax = fig.add_subplot(projection='3d')
40 ax.set_title("Curva Bezier")
41 ax.set_xlabel('x')
42 ax.set_ylabel('y')
43 ax.set_zlabel('z')
44
45 ax.plot_surface(B0,B1,B2, cmap = cm.cool)
46 ax.set_xlim(0,1)
47 plt.show()

```

)

Figura 8.25: Implementación de código

### Curva Bezier

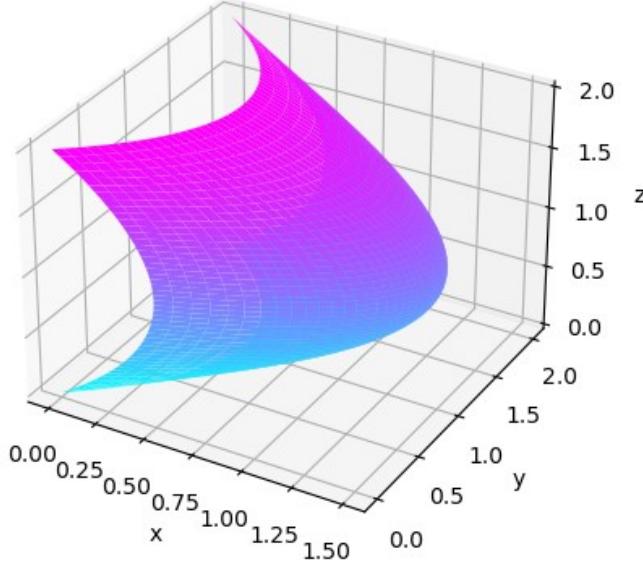


Figura 8.26: Producto cartesiano de curvas de Bézier (2x2)

Desarrollando el producto cartesiano de dos curvas de Bézier, la primera curva de tercer orden y la segunda curva de segundo orden se obtiene lo siguiente:

Siendo  $B(w) = \sum_{j=0}^3 \binom{3}{j} (1-w)^{3-j}$  y  $B(u) = \sum_{i=0}^2 \binom{2}{i} (1-u)^{2-i} u^i$ , el producto cartesiano es

$$B(u) \times B(w) = \left( \sum_{i=0}^3 \binom{3}{i} (1-w)^{3-i} u^i \right) \left( \sum_{j=0}^2 \binom{2}{j} (1-u)^{2-j} \right)$$

$$B(u) \times B(w) = \left[ \begin{pmatrix} 3 \\ 0 \end{pmatrix} (1-w)^3 + \begin{pmatrix} 3 \\ 1 \end{pmatrix} (1-w)^2 (w) + \begin{pmatrix} 3 \\ 2 \end{pmatrix} (1-w) (w^2) + \begin{pmatrix} 3 \\ 0 \end{pmatrix} w^3 \right]$$

$$\left[ \begin{pmatrix} 2 \\ 0 \end{pmatrix} (1-u)^2 + \begin{pmatrix} 2 \\ 1 \end{pmatrix} (1-u) (u) + \begin{pmatrix} 2 \\ 1 \end{pmatrix} u^2 \right]$$

$$B(u) \times B(w) = \frac{[(1-u)^2 + 2(1-u)u + u^2]}{[(1-w)^3 + 3(1-w)^2w + 3(1-w)w^2 + w^3]}$$

$$B(u) \times B(w) = (1-u)^2 [(1-w)^3 + 3(1-w)^2w + 3(1-w)w^2 + w^3]$$

$$+ 2(1-u)u [(1-w)^3 + 3(1-w)^2w + 3(1-w)w^2 + w^3]$$

$$+ u^2 [(1-w)^3 + 3(1-w)^2w + 3(1-w)w^2 + w^3]$$

$$B(u) \times B(w) = (1-u)^2 (1-w)^3 + 3w(1-u)^2 (1-w)^2 + 3w^2(1-u)^2 (1-w) + (1-u)^2 w^3$$

$$+ 2u(1-u)(1-w)^3 + 6uw(1-u)(1-w)^2 + 6uw^2(1-u)(1-w)$$

$$+ 2uw^3(1-u) + u^2(1-w)^3 + 3wu^2(1-w)^2 + 3w^2u^2(1-w) + u^2w^3$$

Evaluando en  $\begin{bmatrix} (0,0,0) & (1,0,1) & (0,0,2) & (1,1,0) \\ (4,1,1) & (1,1,2) & (0,2,0) & (1,1,0) \\ (0,2,2) & (2,0,2) & (0,0,4) & (2,2,4) \end{bmatrix}$  se obtiene finalmente.

$$B(u) \times B(w) = (1-u)^2 (1-w)^3 P_{00} + 3w(1-u)^2 (1-w)^2 P_{01} + 3w^2(1-u)^2 (1-w) P_{02} + (1-u)^2 w^3 P_{03}$$

$$+ 2u(1-u)(1-w)^3 P_{10} + 6uw(1-u)(1-w)^2 P_{11} + 6uw^2(1-u)(1-w) P_{12}$$

$$+ 2uw^3(1-u) P_{13} + u^2(1-w)^3 P_{20} + 3wu^2(1-w)^2 P_{21} + 3w^2u^2(1-w) P_{22} + u^2w^3 P_{23}$$

Implementando el producto cartesiano en python, se obtiene el resultado de la figura 8.28

```

14
15 from numpy import *
16 from mpl_toolkits.mplot3d import axes3d
17 import matplotlib.pyplot as plt
18 from matplotlib import cm
19
20
21 P = array([[0,0,0],[1,0,1],[0,0,2],[1,1,0],
22 [4,1,1],[1,1,2],[0,2,0],[1,1,0],
23 [0,2,2],[2,0,2],[0,0,4],[2,2,4]]).T
24
25 U = linspace(0,1,50)
26 W = linspace(0,1,50)
27
28 B0 = zeros((50,50), dtype = float)
29 B1 = zeros((50,50), dtype = float)
30 B2 = zeros((50,50), dtype = float)
31
32
33 U,W = meshgrid(U,W)
34
35 B0 = ((1-u)**2*(1-w)**3)*P[0,0] + ((3*u**((1-u)**2)*(1-w)**2)*P[1,0]) + ((3*(u**2)*(1-u)**2)*(1-w)**2)*P[2,0] + (((1-u)**2)*(u**3)*(1-w)**2)*P[3,0]
36 B1 = (((1-u)**2)*(1-w)**3)*P[0,1] + ((3*u**((1-u)**2)*(1-w)**2)*P[1,1]) + ((3*(u**2)*(1-u)**2)*(1-w)**2)*P[2,1] + (((1-u)**2)*(u**3)*P[3,1])
37 B2 = (((1-u)**2)*(1-w)**3)*P[0,2] + ((3*u**((1-u)**2)*(1-w)**2)*P[1,2]) + ((3*(u**2)*(1-u)**2)*(1-w)**2)*P[2,2] + (((1-u)**2)*(u**3)*P[3,2])
38
39
40 ax = fig.add_subplot(projection='3d')
41 ax.set_title('Curva Bezier')
42 ax.set_xlabel('x')
43 ax.set_ylabel('y')
44 ax.set_zlabel('z')
45
46 ax.plot_surface(B0,B1,B2, cmap = cm.cool)
47
48 plt.show()

```

+ ( 2\*u\*\*((1-u)\*\*2\*((1-w)\*\*3)\*P[4,0]) + ( 6\*u\*\*((1-u)\*\*2)\*(1-w)\*\*2\*P[5,0] ) + ( 6\*u\*\*((u\*\*2)\*(1-u)\*\*2)\*P[6,0] ) + ( 2\*u\*\*((u\*\*3)\*(1-u)\*\*2)\*P[7,0] ) + ( (u\*\*2)\*(1-w)\*\*3)\*P[8,0] )
+ ( 2\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[9,0] ) + ( 2\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[10,0] ) + ( (u\*\*2)\*(u\*\*3)\*P[11,0] )
+ ( 3\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[9,1] ) + ( 3\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[10,1] ) + ( (u\*\*2)\*(u\*\*3)\*P[11,1] )
+ ( 3\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[9,2] ) + ( 3\*u\*\*((u\*\*2)\*(1-w)\*\*2)\*P[10,2] ) + ( (u\*\*2)\*(u\*\*3)\*P[11,2] )

Figura 8.27: Implementación de código

Curva Bezier

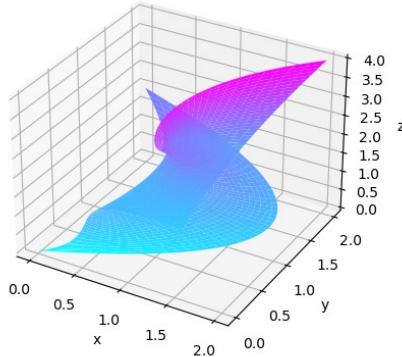


Figura 8.28: Producto cartesiano de curvas de Bézier (2x3)

## 8.6 Curvatura

### Ejercicio

Calcula la curvatura  $k$  para la curva  $r(t) = \cos(t)\hat{i}, \sin(t)\hat{j}$ .

$$r'(t) = -\sin(t)\hat{i} + \cos(t)\hat{j}$$

$$\|r'(t)\| = \sqrt{(-\sin(t))^2 + (\cos(t))^2} = \sqrt{1} = 1$$

$$T(t) = \frac{r'(t)}{\|r'(t)\|} = \frac{-\sin(t)\hat{i} + \cos(t)\hat{j}}{1} = -\sin(t)\hat{i} + \cos(t)\hat{j}$$

$$T'(t) = \frac{-\cos(t)\hat{i} + \sin(t)\hat{j}}{1}$$

$$\|T'(t)\| = \sqrt{(-\cos(t))^2 + (\sin(t))^2} = 1$$

$$k(t) = \frac{\|T'(t)\|}{\|r'(t)\|} = 1$$

∴ La curvatura es de 1.

### Ejercicio

Calcula la curvatura  $k$  para la curva  $r(t) = \langle 2\cos(t), 2\sin(t) \rangle$

$$r'(t) = \langle -2\sin(t), 2\cos(t) \rangle$$

$$\|r'(t)\| = \sqrt{(-2\sin(t))^2 + (2\cos(t))^2} = \sqrt{2^2(\cos^2(t) + \sin^2(t))} = 2$$

$$T(t) = \frac{r'(t)}{\|r'(t)\|} = \frac{\langle -2\sin(t), 2\cos(t) \rangle}{2} = \langle -\sin(t), \cos(t) \rangle$$

$$T'(t) = \langle -\cos(t), -\sin(t) \rangle$$

$$\|T'(t)\| = \sqrt{(-\cos(t))^2 + (-\sin(t))^2} = 1$$

$$N(t) = \frac{T'(t)}{\|T'(t)\|} = \frac{\langle -\cos(t), -\sin(t) \rangle}{1}$$

$$k(t) = \frac{\|T'(t)\|}{\|r'(t)\|} = \frac{1}{2}$$

∴ La curvatura es de  $\frac{1}{2}$

### Ejercicio

Calcula la curvatura  $k$  para la curva  $r(t) = \langle 2\cos(t), 2\sin(t), tk \rangle$

$$r'(t) = \langle -2\sin(t), 2\cos(t), 1 \rangle$$

$$\|r'(t)\| = \sqrt{(-2\sin(t))^2 + (2\cos(t))^2 + 1^2} = \sqrt{4+1} = \sqrt{5}$$

$$T(t) = \frac{r'(t)}{\|r'(t)\|} = \frac{\langle -2\sin(t), 2\cos(t), 1 \rangle}{\sqrt{5}}$$

$$T'(t) = \left\langle -\frac{2}{\sqrt{5}}\cos(t), -\frac{2}{\sqrt{5}}\sin(t), 0 \right\rangle = \frac{2}{\sqrt{5}}$$

$$k(t) = \frac{\|T'(t)\|}{\|r'(t)\|} = \frac{2}{5}$$

∴ La curvatura es de  $\frac{2}{5}$

# Capítulo 9

## Superficies minimales

### 9.1 Introducción

Las estructuras del arquitecto Frei Otto son contorneados tan graciosamente como pelícuas de jabón sobre telarañas. Las membranas translúcidas, soportadas por redes de alambre de acero, levantadas por altos mástiles, además de contar con anclas que fijan las bases de esas formas estereas al suelo. El arquitecto buscaba usar el mínimo posible de material de construcción, para crear estructuras ligeras que fueran fácilmente erigidas, movidas y desmanteladas. Sus modelos fueron dibujados en la naturaleza, de la elegancia y economía desplegada por las pompas de película de jabón.

### 9.2 Películas de jabón y superficies mínimas

La tensión superficial de los líquidos es la tendencia que tienen sus moléculas a juntarse y es el causante de que la superficie que muestran al exterior sea la mínima posible y actúe como una membrana tensa con propiedades elásticas. Para resolver los problemas de máximos y mínimos se requiere unos conocimientos de cálculo infinitesimal. El objetivo de realizar superficies mínimas con películas de jabón es comprobar que basta con una simple disolución jabonosa para que podamos llegar a la comprobación visual y experimental de problemas como el cálculo de recorridos mínimos y las superficies mínimas que a la física y a las matemáticas les ha llevado tanto tiempo dar respuesta y que todavía hoy en día no son definitivos.

Los materiales necesarios fueron:

1. Una mezcla compuesta por el 50 % de agua, 40 % de shampoo y 10 % de glicerina.
2. Poliedros que se pueden construir de acero.
3. Pajas de tomar refrescos o tubos

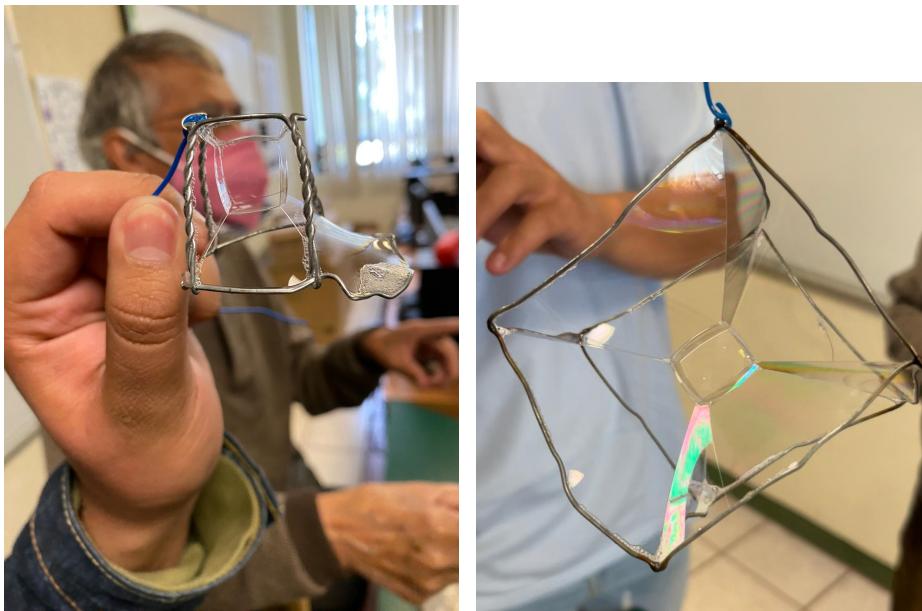


Figura 9.1: Películas de jabón

Fermat demostró que hay un punto, denominado **punto de Fermat**, cuya suma de distancias a los vértices del triángulo es mínima. Los segmentos que unen el punto de Fermat con los vértices del triángulo forman siempre entre sí ángulos de  $120^\circ$ .

### 9.3 El problema de Plateau

Plateau físico belga (1801-1883), encontró la superficie de área mínima acotada por un contorno cerrado en el espacio. Matemáticamente, el problema de Plateau está relacionado con la solución de una ecuación diferencial parcial o un sistema de tales ecuaciones. Euler mostró que todas las superficies mínimas (no planas) deben de tener forma de silla de montar y que la curvatura media en todo punto debe ser cero.

Los experimentos de Plateau proporcionan de inmediato soluciones físicas para contornos muy generales. Si uno sumerge un contorno cerrado hecho de alambre a un líquido con baja tensión superficial y luego lo saca, quedará adherido al contorno una película en forma de una superficie de área mínima. Con este método es muy fácil 'resolver' el problema de Plateau simplemente haciendo un armazón de alambre con la forma deseada.

#### 9.3.1 Leyes de Plateau

1. Primera ley: Si varias láminas de jabón se intersectan, lo hacen de tres en tres a lo largo de una línea y formando entre sí ángulos de  $120^\circ$ .
2. Segunda ley: Cuatro de las líneas, todas formadas por la intersección de tres superficies, se intersectan en un punto y el ángulo formado por cada par de ellas es de  $109^\circ 47'$
3. Tercera ley: Una película que puede moverse libremente sobre una superficie la interseca en un ángulo de  $90^\circ$

### 9.4 El problema de Steiner

Un problema sencillo pero instructivo fue investigado a principios del siglo XIX por Jacob Steiner famoso geómetra de la universidad de Berlín. El problema es . Tres pueblos A, B y C, tiene que

conectarse mediante un sistema de caminos de longitud mínima, en un plano, y se busca un cuarto punto  $P$  en el plano de manera que la suma  $a + b + c$  sea mínima, donde  $a, b, c$  denotan las distancias de  $P$  a  $A, B$  y  $C$ , respectivamente.

La respuesta al problema es. Si el triángulo  $ABC$  todos los ángulos son menores de  $120^\circ$ , entonces el punto  $P$  es el punto en el que cada uno de los tres lados  $AB$ ,  $BC$  y  $CA$ , subtienen un ángulo de  $120$  grados.

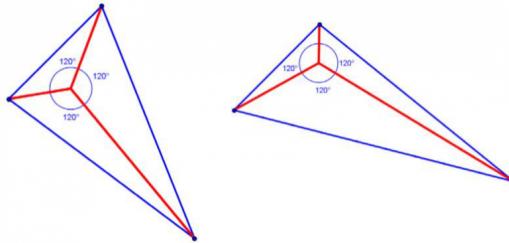


Figura 9.2: Ángulos menores de  $120^\circ$

Sin embargo, si uno de los ángulos de  $ABC$ , por ejemplo  $C$ , es igual o mayor a  $120$  grados, entonces el punto  $P$  coincide con el vértice  $C$ .

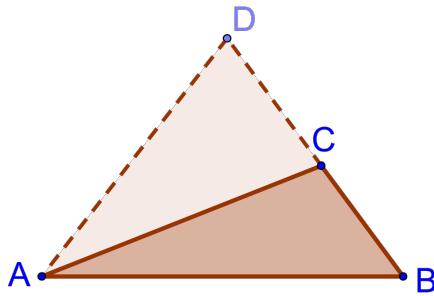


Figura 9.3: Ángulos menores de  $120^\circ$

En una versión más importante del problema de Steiner, buscamos una red de caminos de longitud mínima. Matemáticamente, dados  $n$  puntos  $A_1, A_2, \dots, A_n$ , encontrar un sistema conectado de segmentos de recta de longitud total mínima, tal que cualesquiera dos de los puntos dados se puedan unir mediante un poligonal que consta de segmentos del sistema.

## 9.5 Superficies minimales

En la obtención de la ecuación de Laplace-Young podemos elegir los bordes del entorno rectangular utilizando las direcciones principales que, como ya comentamos, son perpendiculares. En un entorno lo suficientemente pequeño estas curvas son arcos de circunferencias de radios  $R_1$  y  $R_2$ . Así, la fórmula siguiente puede ser reescrita en términos de la curvatura media  $H$ .

$$p = \sigma(k_1 + k_2) = \sigma 2H$$

En el caso de una superficie jabonosa delimitada por una curva, la presión que ejerce el aire sobre la película es cero. Así

$$p = 0 \leftrightarrow H = 0$$

Si una película de jabón encierra un volumen formando una burbuja, esto generaría una presión constante de dentro a fuera, por lo que se tiene que  $H$  es constante y positiva.

El comportamiento de las películas de jabón en relación con la presión del ambiente sugiere la siguiente definición.

Una superficie regular  $M$  se dirá que es minimal si su curvatura media es idénticamente cero.

El objetivo de realizar superficies mínimas con películas de jabón es comprobar que basta con una simple disolución jabonosa para que podamos llegar a la comprobación visual y experimental de problemas como el cálculo de recorridos mínimos y las superficies mínimas que a la física y a las matemáticas les ha llevado tanto tiempo dar respuesta y que todavía hoy en día no son definitivos.

Se dice que una superficie regular es mínima si su curvatura media se anula en todos los puntos. Una superficie  $S \subset \mathbb{R}^3$  es mínima si cada una de sus parametrizaciones es mínima, lo anterior equivale a que  $H = 0$  para todo punto de  $S$ . El adjetivo minimal es dado, pues en definitiva, las superficies minimales son aquellas que, de alguna manera, “minimizan” el área. El problema de minimizar el área se aborda también desde el punto de vista del Cálculo Variacional, rama de las Matemáticas desarrollada principalmente por Lagrange en 1760. Lagrange estudia grafos, esto es, superficies que globalmente pueden expresarse como gráfica de una función diferenciable  $Z = f(x,y)$ , e introduce una nueva definición de superficial minimal.

La superficie como la silla de montar tiene curvatura media cero, ya que en cada punto la superficie se curva suavemente tanto hacia arriba como hacia abajo. El concepto de curvatura auxilia a caracterizar superficies minimales. Dado que un punto sobre una superficie minimal deberá tener una curvatura medial igual a cero. Tales superficies deben ser ser planos, cilindros o como sillas de montar. Euler estableció que una característica de las superficies minimales es que su forma debería de ser como la silla de montar, y que su curvatura promedio debería ser igual a cero.

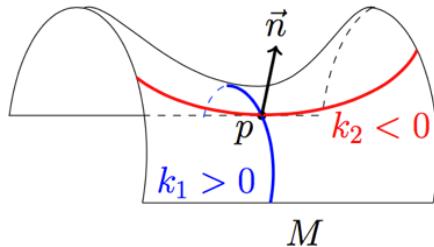


Figura 9.4: Silla de montar

Para calcular la curvatura media de una superficie, se realiza lo siguiente:

Sea una superficie  $S$  con una parametrización  $r(u,v)$  regular simple en  $\mathbb{R}^3$ . Llamaremos primera forma fundamental a la expresión:

$$I = (r_u \bullet r_v)du^2 + 2(r_u \bullet r_v)dudv + (r_u \bullet r_v)dvu^2$$

Se acostumbra descomponer para el cálculo en

$$E = r_u \bullet r_u = \|r_u\|^2$$

$$F = r_v \bullet r_u$$

$$G = r_v \bullet r_v = \|r_v\|^2$$

De esta manera la primera forma fundamental quedaría

$$I = Edu^2 + 2Fdudv + Gdv^2$$

Para obtener  $H$ , se necesita calcular el vector normal unitario  $N = (r_u \times r_v) / |r_u \times r_v|$  y las cantidades  $I = r_{uu} \bullet N$ ,  $m = r_{uv} \bullet N$  y  $n = r_{vv} \bullet N$ .

Finalmente, calculamos  $H$  con la siguiente fórmula:

$$H = \frac{GI - En - 2Fm}{2(EG - F^2)}$$

### 9.5.1 Catenoide

La catenoide es la superficie de revolución generada por la curva catenaria cuando rota alrededor de un eje coplanario rectilíneo, perpendicular al eje de simetría de la catenaria, además se considera que es una superficie minimal. Fijado el sistema de referencia en  $R^3$ , la catenoide se parametriza de la siguiente manera

$$r(u,v) = \langle u, \cosh(u) \cos(v), \cosh(u) \sin(v) \rangle \quad 0 < u < 2\pi, -\infty < v < \infty.$$

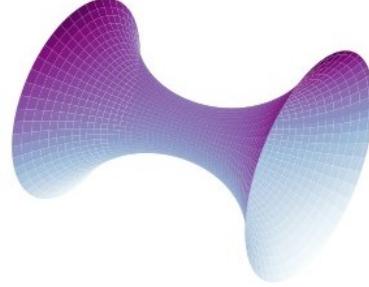
En la figura 9.5, se muestra la implementación de código para graficar un catenoide.

```

15 import numpy
16 from numpy import *
17 import matplotlib.pyplot as mp
18 from mpl_toolkits.mplot3d import axes3d
19 from matplotlib import cm
20
21 v = linspace(0,(2*pi)+0.1,100)
22 u = linspace(-2,2+0.1,100)
23 U,V = meshgrid(u,v)
24 X = U
25 Y = cosh(U) * cos(V)
26 Z = cosh(U) * sin(V)
27
28 fig = mp.figure()
29
30 ax = fig.add_subplot(111,projection="3d")
31 ax.axis('off')
32 ax.set_title("Catenoide")
33 ax.plot_surface(X,Y,Z, cmap = cm.BuPu)
34 mp.show()

```

(a) Implementación de código



(b) Catenoide

Figura 9.5: Superficie minimal Catenoide

### 9.5.2 Helicoide

El Helicoide es una superficie reglada y tiene la topología de un plano. Es el tornillo de Arquímedes. Lejos de su eje vertical, el Helicoide es prácticamente plano. Además tiene un gran número de simetrías y en particular es invariante por una traslación vertical. Esta propiedad es común a los ejemplos clásicos y a muchos de los modernos. El helicoide se parametriza de la siguiente manera:

$$r(u,v) = \langle R * v * \cos(u), R * v * \sin(u), b * u \rangle \quad 0 < u < 2\pi, -\infty < v < \infty.$$

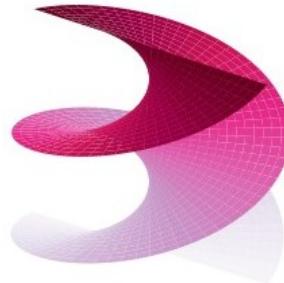
En la figura 9.6, se muestra la implementación de código para graficar un helicoide.

```

15 from numpy import *
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18
19 fig = plt.figure()
20 ax = fig.add_subplot(111, projection='3d')
21 v = arange(-pi/2, pi/2, 0.1)
22 u = arange(0, 2*pi, 0.1)
23 R = 0.5
24 b = 0.5
25 U,V = meshgrid(u,v)
26
27 ax.plot_surface(R * V* cos(U), R * V* sin(U), b * U, rstride=1,
28                  cstride=1, cmap=cm.PuRd)
29 ax.axis('off')
30 ax.set_title("Helicoide")
31 plt.show()

```

(a) Implementación de código



(b) Helicoide

Figura 9.6: Superficie minimal Helicoide

### 9.5.3 Superficie de Enneper

La superficie de Enneper fue introducido por Alfred Enneper en relación con la teoría de superficies mínimas. La superficie de Enneper es la superficie parametrizada:

$$r(u,v) = \langle u - \frac{1}{3}u^3 + uv^2, v - \frac{1}{3}v^3 + vu^2, u^2 - v^2 \rangle \quad (u, v) \in \mathbb{R}^2$$

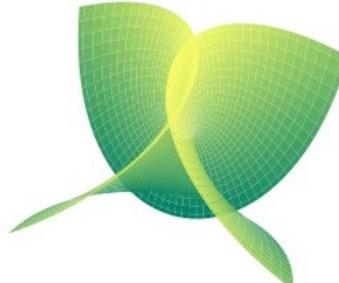
En la figura 9.7, se muestra la implementación de código para graficar una superficie de Enneper.

```

15 from numpy import *
16 import matplotlib.pyplot as mp
17 from matplotlib import cm
18
19 v = linspace(-2,2,100)
20 u = linspace(-2,2,100)
21
22 U,V = meshgrid(u,v)
23
24 X = U - (U**3) /3 + (U*V**2)
25 Y = V - (V**3 /3) + (V*U**2)
26 Z = U**2 - V**2
27
28 fig = mp.figure()
29
30 ax = fig.add_subplot(111,projection="3d")
31 ax.set_title("Superficie de Enneper")
32 ax.plot_surface(X,Y,Z, cmap = cm.summer)
33 ax.axis('off')
34 mp.show()

```

(a) Implementación de código



(b) Superficie de Enneper

Figura 9.7: Superficie minimal Enneper

# Capítulo 10

## Triangulación de Delaunay

### 10.1 Introducción

Podemos modelar un pedazo de la tierra, como un terreno. Un terreno es una superficie de dos dimensiones en el espacio, con una propiedad especial: cada línea vertical intersecta en un punto al terreno. En otras palabras, esto es la gráfica de una función  $f : A \subset \mathbb{R}^2 \rightarrow \mathbb{R}$  que asigna una altura  $f(p)$  a cada punto  $p$  en el dominio  $A$  del terreno. Se puede visualizar como la siguiente imagen.

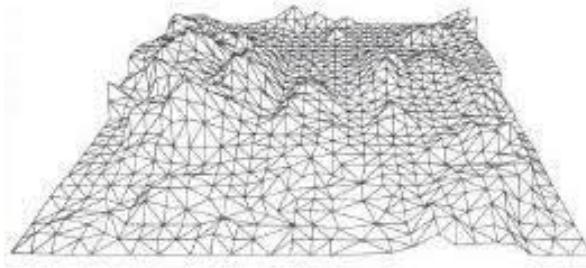


Figura 10.1: Triangulación de Delaunay en terrenos

Con esto se consigue un terreno de forma poliédrica, definido por una función continua. Este terreno poliédrico se puede utilizar como una aproximación del terreno real. Por regla general, interesa que, puntos cuyas proyecciones son puntos próximos, estén conectados por aristas de la superficie poliédrica. Esre da interés al siguiente problema cuya solución es lo que se conoce como la triangulación de Delaunay.

Primero determinamos una triangulación de  $P$ : una subdivisión plana cuyas caras acotadas son triángulos y cuyos vértices son los puntos  $P$ . Después se levanta en los puntos muestra las alturas correspondientes, este modod se mapea cada triángulo en la triangulación a un triángulo en espacio 3D.

Este terreno políedro se puede utilizar como una aproximacion del terreno real.

### 10.2 Triangulación de Delaunay

Dada una nube de puntos del plano, hallar la triangulación en la que los puntos próximos estén conectados entre sí por una arista. O dicho de otro modo, en la que los triángulos sean los más regulares posibles. Se trata de obtener una triangulación óptima, que permita una interpolación coherente entre los valores asociados (cotas o altitudes) a cada uno de los vértices; y así construir una red de triángulos irregulares (TIN), para la generación de modelos digitales de elevación. Una triangulación de Delaunay es una red de triángulos conexa y convexa que cumple la condición de Delaunay. Esta de la red no

debe contener ningún vértice de otro triángulo. Las triangulaciones de Delaunay tienen importante relevancia en el campo de la geometría computacional, especialmente en gráficos 3D por computadora.

Se le denomina así por el matemático ruso Borís Nikolaevich Delone quien ideó en 1934; el mismo Delone usó la forma francesa de su apellido “Delaunay”, como apreciación a sus antecedentes franceses.

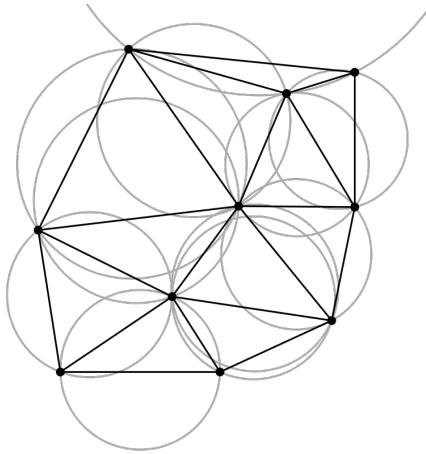


Figura 10.2: Triangulación de Delaunay de 10 puntos

### 10.3 Condición de Delaunay

La condición de Delaunay de un triángulo establece que la circunferencia circunscrita del mismo no debe contener ningún otro vértice de la triangulación en su interior, aunque sí se admiten vértices situados sobre la circunferencia.

Se dice que una red de triángulos es una **triangulación de Delaunay** si todos los triángulos de la misma cumplen la condición de Delaunay. Es decir, que cada circunferencia circunscrita de cada triángulo no contiene vértices de la triangulación en su interior. Esta definición original para espacios bidimensionales se puede ampliar a espacios tridimensionales o incluso dimensiones superiores, usando la esfera circunscrita en vez de la **circunferencia circunscrita**.

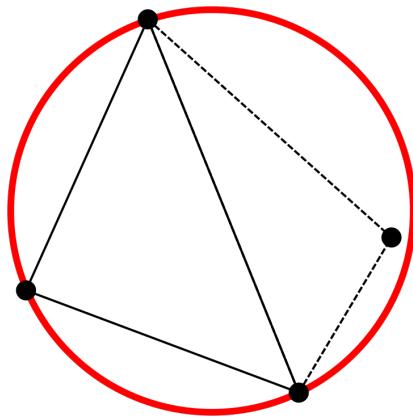


Figura 10.3: Vértice completamente en el interior de la circunferencia circunscrita

### 10.4 Propiedades de la triangulación de Delaunay

Las triangulaciones de Delaunay de un conjunto de puntos cumplen las siguientes propiedades:

- La frontera externa de triangulación forma la envolvente convexa del conjunto de puntos.
- El ángulo mínimo dentro de todos los triángulos está maximizado, es decir, se evita obtener resultados con ángulos demasiado agudos.
- Los triángulos generados en una triangulación de Delaunay tienden a ser lo más equiláteros posible.
- La triangulación de Delaunay es unívoca salvo en casos donde los vértices presentan una alineación perfecta.
- El **grafo de Gabriel** es un subgrafo de las aristas de la triangulación de Delaunay. Es decir, todas las aristas del grafo de Gabriel pertenecen a algún triángulo de la triangulación.
- El **grafo del vecino más cercano** es un subgrafo de las aristas de la triangulación de Delaunay. Es decir, todas las aristas del grafo del vecino más cercano pertenecen a algún triángulo de la triangulación.
- Como consecuencia de lo anterior, cada punto del conjunto de entrada tendrá una arista que lo une con su punto más cercano.
- La triangulación de Delaunay y el **diagrama de Voronoi** de una serie de puntos son grafos duales, por lo que la construcción de uno es trivial a partir del otro. En este sentido, los circuncentros de los triángulos de Delaunay coinciden con los vértices de las regiones del diagrama de Voronoi. Dos vértices del diagrama de Voronoi estarán conectados si sus triángulos de Delaunay correspondientes son vecinos entre sí.

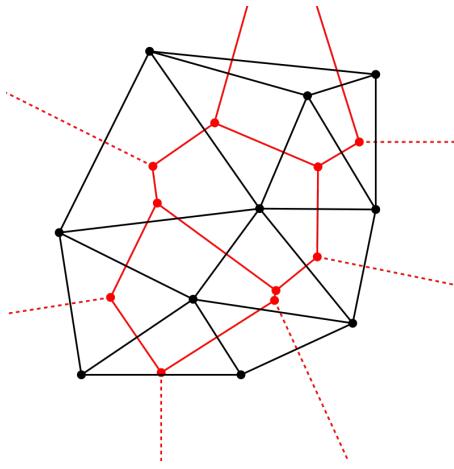


Figura 10.4: Diagrama de voronoi

- En un grafo construido a partir de las aristas de la triangulación de Delaunay, el camino más corto entre dos puntos nunca será mayor que  $\frac{4\pi}{3\sqrt{3}} \approx 2.418$  veces la distancia euclídea entre ellos.

La propiedad de la triangulación de Delaunay de maximizar los ángulos interiores de los triángulos es especialmente práctica en geometría computacional porque evita errores de redondeo que pueden aparecer al realizar cálculos con triangulaciones arbitrarias donde pueden aparecer ángulos demasiado pequeños.

#### Ejercicio

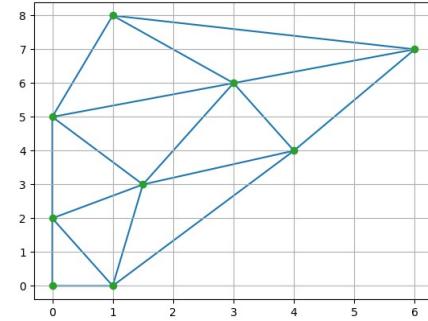
Utiliza la librería Delaunay en python, para obtener las triangulaciones de los siguientes puntos:  $(0, 0), (0, 2), (1, 0), (1.5, 3), (4, 4), (3, 6), (0, 5), (6, 7), (1, 8)$

```

14 import numpy
15 from numpy import *
16 import matplotlib.pyplot as plt
17 from matplotlib import cm
18 from mpl_toolkits.mplot3d import axes3d
19 from scipy.spatial import Delaunay
20
21 points = array([[0,0],[0,2],[1,0],[1.5,3],[4,4],[3,6],[0,5],[6,7],[1,8] ])
22
23 tri = Delaunay(points)
24
25 plt.triplot(points[:,0],points[:,1],tri.simplices)
26
27 plt.plot(points[:,0],points[:,1],'o')
28 plt.grid("on")
29 plt.show()

```

(a) Implementación de código



(b) Tringulación de Delaunay

Figura 10.5: Tringulación de Delaunay

### Ejercicio

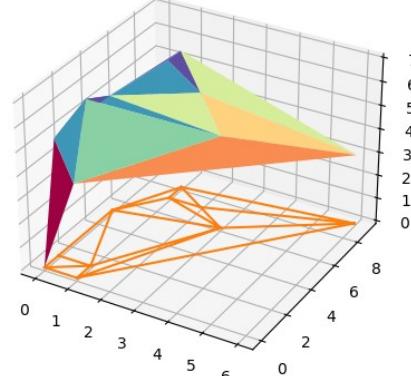
Utiliza la librería Delaunay en python, para obtener las triangulaciones de los siguientes puntos:  
 $(0,0)$ ,  $(0,1)$ ,  $(1,0)$ ,  $(2,7)$ ,  $(3,6)$ ,  $(0,5)$ ,  
 $(1,7)$ ,  $(6,9)$ ,  $(1,8)$  y realiza una superficie con la proyección de las triangulaciones

```

14 from numpy import *
15 import matplotlib.pyplot as plt
16 from matplotlib import cm
17 from scipy.spatial import Delaunay
18 from scipy import spatial as sp_spatial
19
20
21 p = array([[0,0],[0,1],[1,0],[1,1],[2,7],[3,6],[0,5],[1,7],[6,9],[1,8]])
22 P = array([[0,0,0],[0,1,5],[1,0,4],[1,1,7],[2,7,5],[3,6,4],
23 [0,5,5],[1,7,6],[6,9,3],
24 [1,8,6]])
25
26 tri = Delaunay(P)
27 tri2 = Delaunay(p)
28
29
30 fig = plt.figure()
31 ax = fig.add_subplot(111, projection='3d')
32 ax.plot_trisurf(P[:,0],P[:,1], P[:,2],
33 triangles=tri.simplices, cmap=plt.cm.Spectral)
34 #ax.plot_wireframe(P[:,0],P[:,1],P[:,2],triangles=tri.simplices)
35 plt.triplot(P[:,0],P[:,1],tri2.simplices)
36 plt.show()

```

(a) Implementación de código



(b) Tringulación de Delaunay y superficie en 3D

Figura 10.6: Tringulación de Delaunay y superficie en 3D

### Ejercicio

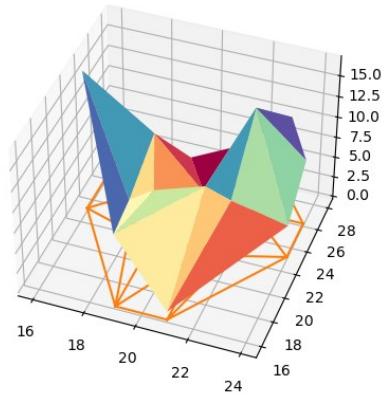
Utiliza la librería Delaunay en python, para obtener las triangulaciones de los siguientes puntos:  
 $(16,28)$ ,  $(18,26)$ ,  $(19,28)$ ,  $(20,26)$ ,  $(21,28)$ ,  $(22,26)$ ,  $(23,28)$ ,  $(24,26)$ ,  $(24,23)$ ,  $(22,22)$ ,  $(21,16)$ ,  $(19,16)$ ,  
 $(19,22)$ ,  $(18,22)$ ,  $(16,23)$  y realiza una superficie con la proyección de las triangulaciones.

```

14 from numpy import *
15 import matplotlib.pyplot as plt
16 from matplotlib import cm
17 from scipy.spatial import Delaunay
18
19
20 P = array([[16,28],[18,26],[19,28],[20,26],[21,28],[22,26],[23,28],
21 [24,26],[24,23],[22,22],[21,22],[21,16],[19,16],[19,22],
22 [18,22],[16,23],[16,28]])
23
24 P = array([[16,28,6],[18,26,7],[19,28,2],[20,26,1],[21,28,5],[22,26,13],[23,28,10],
25 [24,26,8],[24,23,4],[22,22,7],[21,22,8],[21,16,1],[19,16,9],[19,22,6],
26 [18,22,4],[16,23,17],[16,28,15]])
27
28
29 tri = Delaunay(P)
30 tri2 = Delaunay(p)
31
32 fig = plt.figure()
33 ax = fig.add_subplot(111,projection='3d')
34 ax.plot_trisurf(P[:,0],P[:,1],P[:,2],triangles=tri.simplices,cmap=plt.cm.Spectral)
35 #ax.plot_wireframe(P[:,0],P[:,1],P[:,2],triangles=tri.simplices)
36 plt.triplot(P[:,0],P[:,1],tri2.simplices)
37 plt.show()

```

(a) Implementación de código



(b) Tringulación de Delaunay y superficie en 3D

Figura 10.7: Tringulación de Delaunay y superficie en 3D

### Ejercicio

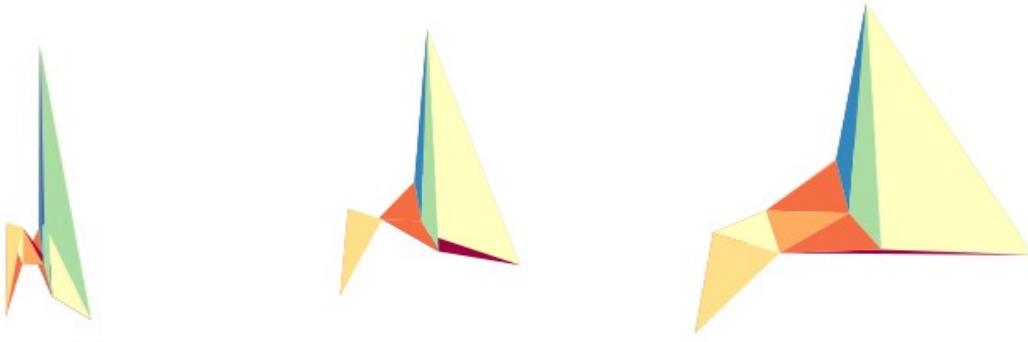
Utiliza la librería Delaunay en python, para realizar una animación de una superficie realizada con las proyecciones de las triangulaciones de Delaunay, de los siguientes puntos:  $(0,0)$ ,  $(0,1)$ ,  $(1,0)$ ,  $(2,7)$ ,  $(3,6)$ ,  $(0,5)$ ,

```

14 from numpy import array
15 import matplotlib.pyplot as plt
16 from scipy.spatial import Delaunay
17
18
19 p = array([[0,0],[0,1],[1,0],[1,1],[2,7],[3,6],[0,5],[1,7],[6,9],[1,8]])
20
21 P = array([[2,6,5],[3,1,7],[1,0,4],[1,1,7],[2,7,12],[3,6,4],
22 [0,5,5],[1,7,6],[6,9,3],
23 [1,8,6]])
24
25 fig = plt.figure()
26
27 for i in range(1,10):
28     ax = fig.add_subplot(111,projection='3d')
29     ax.axis([0,90,0,90])
30     tri = Delaunay(P[:i])
31     tri2 = Delaunay(p[:i])
32     ax.plot_trisurf(P[:,0]*i,P[:,1]*i, P[:,2]*i,triangles=tri.simplices,
33                     cmap=plt.cm.Spectral)
34     ax.axis('off')
35     plt.pause(0.2)

```

Figura 10.8: Implementación de código



(a)

(b)

(c)

Figura 10.9: Animación de una superficie con triangulaciones de Delaunay

## Ejercicio

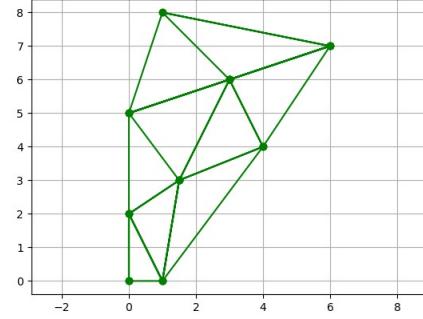
Utiliza la librería Path de python, para dibujar las triangulaciones de Delaunay de los siguientes puntos:  
 $(0,0), (0,2), (1,0), (1.5,3), (4,4), (3,6), (0,5), (6,7), (1,8)$

```

14 from numpy import array
15 import matplotlib.patches as mpatches
16 import matplotlib.path as mpath
17 import matplotlib.pyplot as plt
18
19
20 points = array([[0,0],[0,2],[1,0],[1.5,3],[4,4],[3,6],[0,5],[6,7],[1,8] ])
21
22 fig, ax = plt.subplots()
23 patch = mpath.PathPatch
24
25
26 path_data = [(Path.MOVETO,(0,0)),(Path.MOVETO,(0,2)),(Path.MOVETO,(1,0)),(Path.CLOSEPOLY,(0,0)),
27 (Path.MOVETO,(0,2)),(Path.MOVETO,(1,0)),(Path.MOVETO,(1.5,3)),(Path.CLOSEPOLY,(0,0)),
28 (Path.MOVETO,(1,0)),(Path.MOVETO,(1.5,3)),(Path.MOVETO,(4,4)),(Path.CLOSEPOLY,(0,0)),
29 (Path.MOVETO,(1,0)),(Path.MOVETO,(1.5,3)),(Path.MOVETO,(4,4)),(Path.MOVETO,(3,6)),(Path.CLOSEPOLY,(1.5,3)),
30 (Path.MOVETO,(1.5,3)),(Path.MOVETO,(4,4)),(Path.MOVETO,(3,6)),(Path.CLOSEPOLY,(3,6)),
31 (Path.MOVETO,(3,6)),(Path.MOVETO,(0,5)),(Path.MOVETO,(6,7)),(Path.MOVETO,(1,8)),(Path.CLOSEPOLY,(0,5)),
32 (Path.MOVETO,(0,5)),(Path.MOVETO,(6,7)),(Path.MOVETO,(1,8)),(Path.CLOSEPOLY,(0,5)),
33 (Path.MOVETO,(6,7)),(Path.MOVETO,(1,8)),(Path.MOVETO,(4,4)),(Path.CLOSEPOLY,(0,5)),
34 (Path.MOVETO,(3,6)),(Path.MOVETO,(1,8)),(Path.MOVETO,(6,7)),(Path.CLOSEPOLY,(3,6)),
35 (Path.MOVETO,(4,4)),(Path.MOVETO,(6,7)),(Path.CLOSEPOLY,(3,6))
36 ]
37
38 codes , verts = zip(path_data)
39 path = mpath.Path(verts,codes)
40 patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
41 ax.add_patch(patch)
42 x,y = zip(*patch.vertices)
43 line, = ax.plot(x,y,'go-')
44 ax.grid()
45 ax.axis('equal')
46 plt.show()

```

(a) Implementación de código



(b) Traingulación de Delaunay

Figura 10.10: Tringulación de Delaunay con librería Path

## Ejercicio

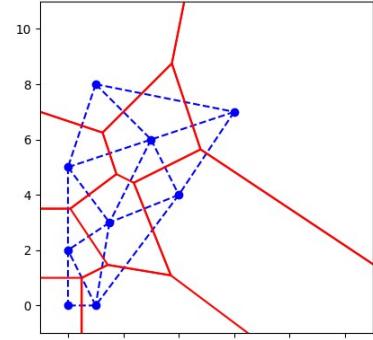
Utiliza el código de Delaunay e implementa las triangulaciones del los puntos anteriores.

```

2
3 Minimal delaunay2D test
4 See: http://github.com/jmespadero/pyDelaunay2D
5
6 import numpy as np
7 from delaunay2D import Delaunay2D
8
9 # Create a random set of 2D points
10 seeds = np.array([[0,0],[0,2],[1,0],[1.5,3],[4,4],[3,6],[0,5],[6,7],[1,8]])
11
12 # Create Delaunay Triangulation and insert points one by one
13 dt = Delaunay2D()
14 for s in seeds:
15     dt.addPoint(s)
16
17 # Dump points and triangles to console
18 print("Input points:\n", seeds)
19 print ("Delaunay triangles:\n", dt.exportTriangles())
20

```

(a) Implementación de código



(b) Traingulación de Delaunay

Figura 10.11: Tringulación de Delaunay

# Capítulo 11

## Cálculo

### 11.1 Recta tangente a una curva en un punto

La pendiente de la recta tangente a una curva en un punto es la derivada de la función en dicho punto.

$$\tan(\theta) = \lim_{h \rightarrow 0} \frac{\Delta y}{h} = f'(a)$$

La recta tangente a una curva en un punto es quella que pasa por el punto  $(a, f(a))$  y cuya pendiente es igual a  $f'(a)$

$$y - f(a) = f'(a)(x - a)$$

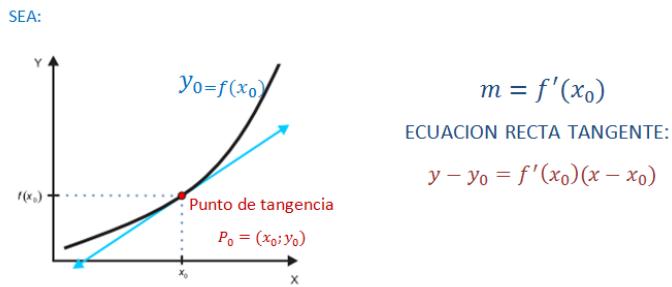


Figura 11.1: Recta tangente a una curva

#### Ejercicio

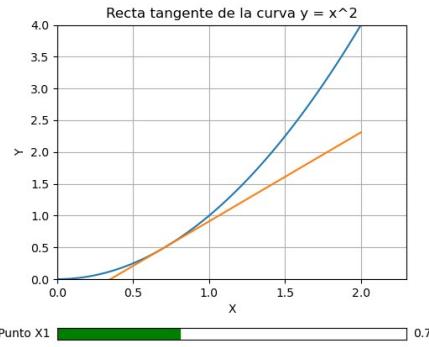
Realiza un programa que muestre la recta tangente en un punto de la función  $x^2$  de  $[0, 2]$

```

15 import numpy as np
16 import matplotlib.pyplot as plt
17 from matplotlib.widgets import Slider
18
19 fig,ax = plt.subplots()
20 plt.subplots_adjust(left=0.25, bottom=0.25)
21
22 #Configuración de slider
23 axslider = plt.axes([0.25, 0.1, 0.65, 0.03])
24 slider = Slider(axslider, 'Punto X1', 0, 2, valinit=0,
25 valstep=0.1,color='green')
26 x = np.linspace(0,2,100)
27 y = x**2
28
29 #completar código de plot
30 ax.plot(x,y)
31 ax.set_xlim([0, 3, 0, 4])
32 ax.set_title("Recta tangente de la curva y = x^2")
33 ax.set_xlabel("x")
34 ax.set_ylabel("y")
35 ax.grid()
36 l, = ax.plot(x,y)
37
38 def update(val):
39     xp = slider.val
40     m = 2*xp #pendiente
41     yp = xp**2 #y coordenada
42     tangente = m*(x-xp) + yp #tangente
43     l.set_ydata(tangente)
44     fig.canvas.draw_idle()
45
46
47 slider.on_changed(update)
48 plt.show()

```

(a) Implementación de código



(b)  $x = 0.7$

Figura 11.2: Recta tangente en un punto

### Ejercicio

Realiza un programa que muestre la recta tangente en un punto a cualquier función.

```

15 import numpy as np
16 import matplotlib.pyplot as plt
17 from matplotlib.widgets import Slider
18
19 fig,ax = plt.subplots()
20 plt.subplots_adjust(left=0.25, bottom=0.25)
21
22 #Configuración de slider
23 axslider = plt.axes([0.25, 0.1, 0.65, 0.03])
24 slider = Slider(axslider, 'Punto X1', 0, 2, valinit=0,
25 valstep=0.1,color='green')
26 x = np.linspace(0,2,100)
27 y = x**2
28
29 #completar código de plot
30 ax.plot(x,y)
31 ax.set_xlim([0, 3, 0, 4])
32 ax.set_title("Recta tangente de la curva y = x^2")
33 ax.set_xlabel("x")
34 ax.set_ylabel("y")
35 ax.grid()
36 l, = ax.plot(x,y)
37
38 def update(val):
39     xp = slider.val
40     m = 2*xp #pendiente
41     yp = xp**2 #y coordenada
42     tangente = m*(x-xp) + yp #tangente
43     l.set_ydata(tangente)
44     fig.canvas.draw_idle()
45
46
47 slider.on_changed(update)
48 plt.show()

```

(a)

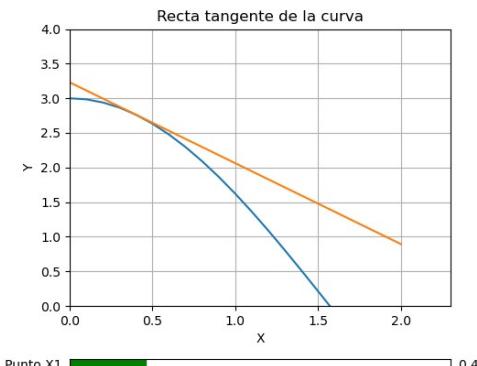
```

15 import numpy as np
16 from sympy import symbols,diff
17 from sympy.parsing.sympy_parser import parse_expr
18 import matplotlib.pyplot as plt
19 from matplotlib.widgets import Slider
20
21 x= symbols("x")
22 f = symbols("f")
23 f = parse_expr(input("Escriba la función f: "))
24 der = diff(f)
25 w = np.arange(0,2.01,0.1)
26
27 fig,ax = plt.subplots()
28 plt.subplots_adjust(left=0.25, bottom=0.25)
29
30 #Configuración de slider
31 axslider = plt.axes([0.25, 0.1, 0.65, 0.03])
32 slider = Slider(axslider, 'Punto X1', 0, 2, valinit=0,
33 valstep=0.1,color='green')
34
35 #calcular la evaluación en los puntos
36 yeval = []
37 for i in w:
38     yeval.append(f.subs(x,i))

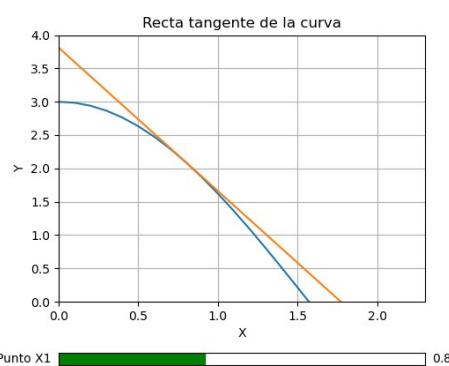
```

(b)

Figura 11.3: Implementación de código



(a)  $x=0.4$



(b)  $x=0.8$

Figura 11.4: Tangente de la función  $3 \cos(x)$

## 11.2 Integral definida con sumas de Riemann

Las sumas de Riemann es un método para aproximar el área total bajo la gráfica de una curva. Estas sumas toman su nombre matemático alemán Bernhard Riemann. Es una operación sobre una función continua y limitada en un intervalo  $[a, b]$ , donde  $a$  y  $b$  son llamados extremos de la integración. La operación consiste en hallar el límite de la suma de productos entre el y la función en un punto  $x_i$  y el ancho  $\Delta x$  del subintervalo contenido al punto.

$$\int_a^b f(x)dx = \lim_{h \rightarrow \infty} \sum_{i=0}^{\infty} f(x_i)\Delta x$$

donde:

$$\Delta x = \frac{b-a}{h}, x_i \in \Delta x$$

La integral de Riemann es una forma simple de definir la integral de una función sobre un intervalo como el área bajo la curva de la función. Sea  $f$  una función con valores reales definida sobre el intervalo  $[a, b]$ , tal que para todo  $x$ ,  $f(x) \geq 0$  (es decir, tal que  $f$  es positiva). Sea  $S = Sf = \{(x, y), 0 \leq y \leq f(x)\}$  la región del plano delimitada por la curva correspondiente a la función  $f$ , el eje de las abscisas  $y$  las rectas verticales de ecuaciones  $x = a$  y  $x = b$ . Estamos interesados en medir el área del dominio  $S$ , si es que se puede medir.

La idea fundamental es la de utilizar aproximaciones del área del dominio  $S$ . Determinando un área aproximada de la que estamos seguros de que son inferiores al área del dominio  $S$ , y buscaremos un área aproximada que sepamos que es mayor al área de  $S$ . Si estas aproximaciones pueden hacerse de forma que la diferencia entre ambas puede hacerse arbitrariamente pequeña, entonces podemos obtener el área del dominio  $S$ . Por lo tanto, el límite del área para infinitos rectángulos es el área comprendida debajo de la curva.

### Ejercicio

Realiza un programa que muestre el área bajo la curva de la función  $x^2$  de  $[0, 2]$

```

2 import numpy as np
3 from sympy import symbols
4 import matplotlib.pyplot as plt
5 from sympy.parsing.sympy_parser import parse_expr
6 from matplotlib.widgets import Slider
7
8 def evalf(f,i):
9     return f.subs(x,i)
10
11 def coorY(f,i):
12     #puntos correspondientes a función
13     yeval = []
14     for i in w:
15         yeval.append(f.subs(x,i))
16
17     return np.array(yeval)
18
19 def Riemann(f,w):
20     A = 0
21     f1 = []
22     f2 = []
23     xbar = []
24     ybar = []
25     a = []
26
27     for i in range (1, len(w)):
28         #Suma de Riemann
29         f1.append(evalf(f,w[i-1]))
30         f2.append(evalf(f,w[i]))
31         fx = np.array([evalf(f,w[i-1]), evalf(f,w[i])])
32         deltax = w[i] - w[i-1]
33         a.append(fx*deltax)
34         A += fx*deltax
35
36     #datos gráficos de Barres:
37     xbar.append(w[1]-1) #Inferior izquierda
38     xbar.append(w[1]) #Esquina superior izquierda
39     ybar.append(f1[1]) #Esquina superior derecha
40     ybar.append(f2[1]) #Esquina inferior derecha
41     ybar.append(0) #Esquina inferior izquierda
42
43     return(a, xbar, ybar)
44
45     #symbols('x')
46     f = symbols('f')
47
48     #Outputs del ejercicio
49     f = parse_expr(input("Función: "))
50     xf = int(input("Final: "))
51     xi = int(input("Incial: "))
52
53     n = np.linspace(xi,xf,100)
54     v = np.linspace(xi,xf,100)
55
56     fig, plt.subplots_adjust(left=0.1, bottom=0.25)
57     plt.subplot(1,1,1)
58     plt.plot(v,f)
59     plt.grid()
60
61     slider = Slider(ax=ax, label="Barres", 0, 100, valinit=0, valstep=1,color="green")
62
63     l, = ax.plot(w,coorY(f,w))
64
65     #Configuración del plot
66     ax.plot(w, coorY(f,w), 'k', label="fx(x)")
67     ax.set_xlim(0,4)
68     ax.set_title("Suma de Riemann")
69     ax.set_xlabel("x")
70     ax.set_ylabel("y")
71     ax.grid()
72
73     def update(val):
74
75         n = slider.val
76         w = np.linspace(xi,xf,n)
77         A, xbar, ybar = Riemann(f,w)
78         l.set_xdata(xbar)
79         l.set_ydata(ybar)
80         print("Área ", A)
81         fig.canvas.draw_idle()
82
83     slider.on_changed(update)
84
85     plt.show()
86

```

(a)

(b)

(c)

Figura 11.5: Implementación de código

Funcion:  $x^{**2}$

Inicial: 0

Final: 2

Area 2.44855967078189

Area 2.59690161075203

Area 2.61494349806038

Area 2.616875000000000

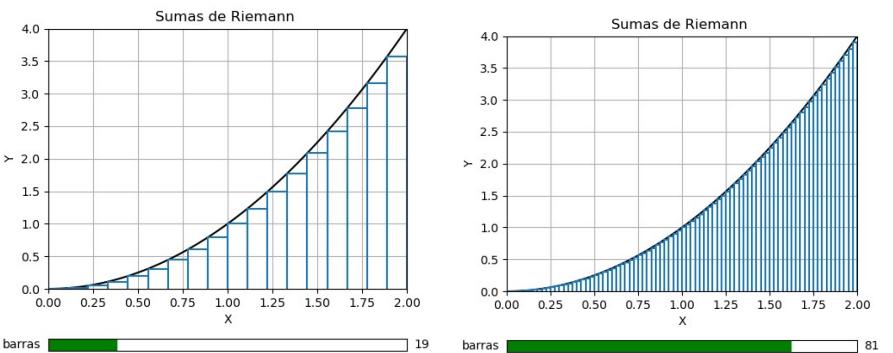


Figura 11.6: Integral definida por sumas de Riemann

# Capítulo 12

## Conclusiones

La graficación computacional es de suma importancia en el diseño computacional, ciencia, industria, tecnología, arte, publicidad, educación, etc. ya que permite graficar figuras, formas, animaciones y otras herramientas que generan una representación visual de un objeto creado a través de distintos métodos.

La graficación por computadora es una de las áreas más importantes de las ciencias de la computación y su principal objetivo es establecer los principios, técnicas y algoritmos para la generación y manipulación de imágenes mediante una computadora.

Además de la sencillez y facilidad que implica el uso de una aplicación en un entorno gráfico, los gráficos e imágenes generados por computadora tienen otra función muy importante y es ser atractivos a la vista, porque podríamos tener una aplicación muy buena pero con un entorno gráfico pobre, esto puede desencadenar que dicha aplicación no tenga el impacto que debería.

La industria cinematográfica es un de las que más se beneficia de los contantes avances de la graficación, pero también está la industria de los videojuegos que es simplemente programas por computadora pero con la peculiaridad que la graficación y la animación toma tanta importancia como la funcionalidad, el mundo de los videojuegos actualmente recauda tanto e incluso más dinero que el cine.

En este trabajo, desarrollaron programas en python para la visualización de superficies en 3D, animaciones, morphing, triangulaciones y otras herramientas que tiene la graficación computacional.

# Referencias

- [1] (2004). Duncan Marsh. Applied Geometry for Computer Graphics and CAD. Segunda Edición. Springer
- [2] (2005). Max K. Agoston. Computer Graphics and Geometric Modeling. Springer. British Library Cataloguing in Publication Data
- [3] (2011). Transformación de Matrices Homogéneas. Online: <http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro39/3>
- [4] (2012). Interpolación y aproximación de funciones. Universidad de Cntabria, Open course Ware Online: <https://ocw.unican.es/pluginfile.php/2926/course/section/2766/Capitulo2.pdf>.
- [5] (2010). Issac Trigo Conde. Transformaciones de imágenes:Morphing. Online: [https://www.acta.es/medios/articulos/disenno\\_y\\_multimedia/008061.pdf](https://www.acta.es/medios/articulos/disenno_y_multimedia/008061.pdf)
- [6] (2006). Daniel Paredes García y David Bernal Martínez. Procesamiento audiovisual: Morphing. Online:<http://dis.um.es/~ginesgm/files/doc/pav/miniproyectos/morphing06/memoria%20morphing.pdf>
- [7] (2009). Cordero Valle, J.M. Curvas y superficies para modelado geométrico. AlfaOmega.
- [8] (2009). Shirlay, T. Ashikhmin, M. Marschner, S., Fundamentals of Computer Graphics (English Edition). Tercera edición, pp. 365-370. A.K.Peters/CRC Press (2009).
- [9] (2011). Ugarte, Francisco & Azabache, Haydee. Superficies mínimas : caso del complejo olímpico de Munich. Pontificia Universidad católica del perú.