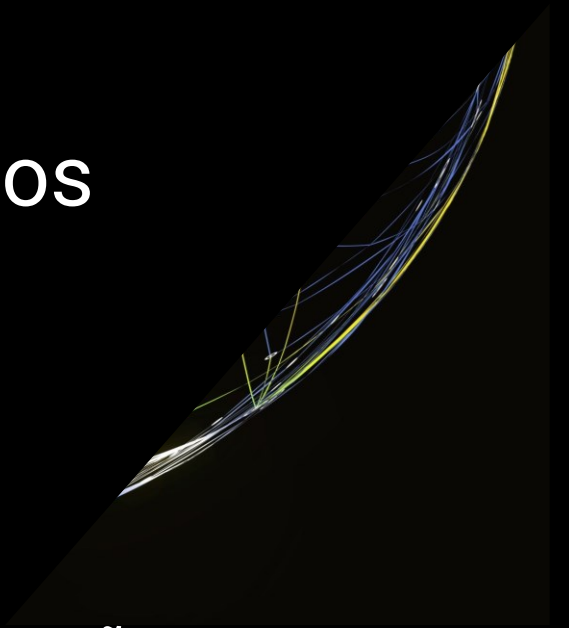
COMPUTAÇÃO GRÁFICA

Aula 5 – Modelos Geométricos



Curso de Ciência da Computação
Dr. Rodrigo Xavier de Almeida Leão
Cientista de Dados

Caro aluno, nesta seção serão apresentadas as formas de representação de objetos tridimensionais por meio de modelos geométricos. A representação de objetos tridimensionais, junto às transformações geométricas, forma a base para a síntese de imagens. Vale lembrar que a síntese de imagens e a computação gráfica estão hoje presentes no cotidiano de todas as pessoas, mesmo as mais leigas em termos de informática. A computação gráfica 3D está presente em filmes, efeitos de televisão, material de ensino, e até mesmo na interação das pessoas com o dispositivo que lhes acompanha a todo momento: o smartphone. Um mundo sem computação gráfica hoje é inimaginável.

Vamos começar pela representação de formas geométricas além das retas e círculos apresentados até o momento. Primeiramente, vamos criar linhas não retas. Uma linha qualquer, desenhada sobre um plano, nem sempre pode ser descrita por uma equação trivial. A Figura 2.7(a) mostra uma linha com esta característica.

Figura 2.7 | Linhas e polilinhas

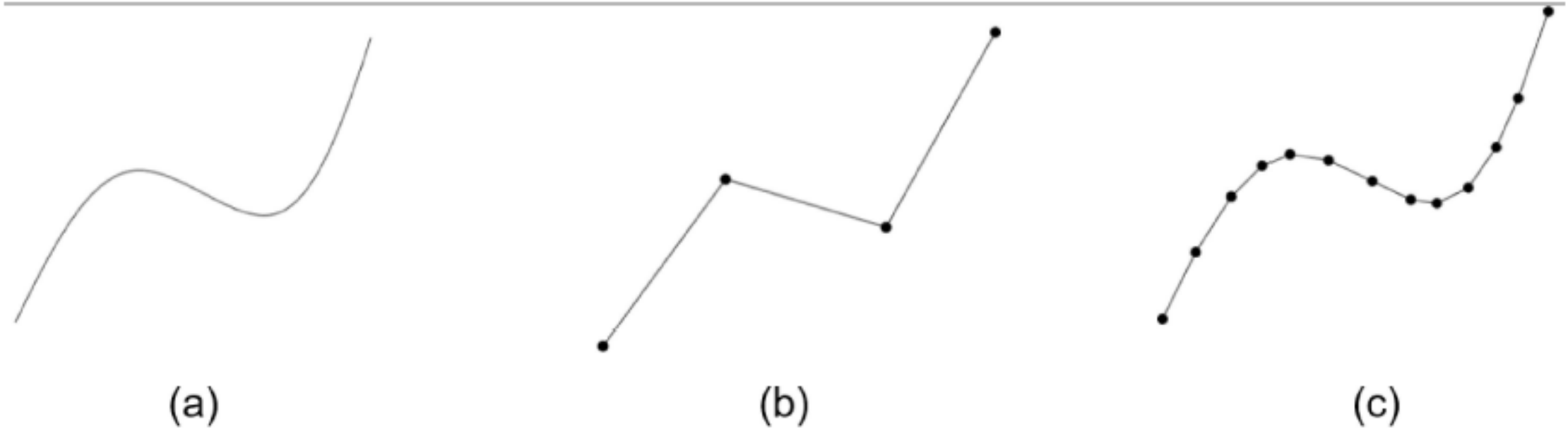
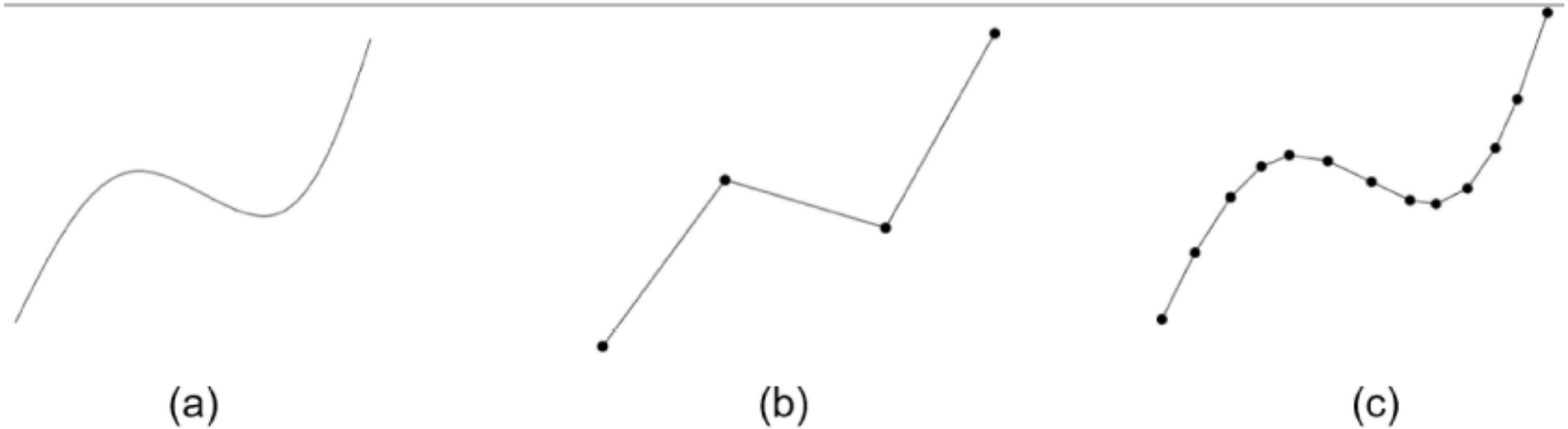


Figura 2.7 | Linhas e polilinhas



Fonte: elaborada pelo autor.

A linha da Figura 2.7(a) pode ser aproximada por uma sequência de segmentos de reta. A Figura 2.7(b) mostra uma aproximação utilizando 3 segmentos de reta e a Figura 2.7(c) mostra uma aproximação utilizando 12 segmentos de reta. É possível observar que a utilização de um número maior de segmentos de reta (Figura 2.7(c)) resulta em uma melhor aproximação da linha contínua original representada na Figura 2.7(a).

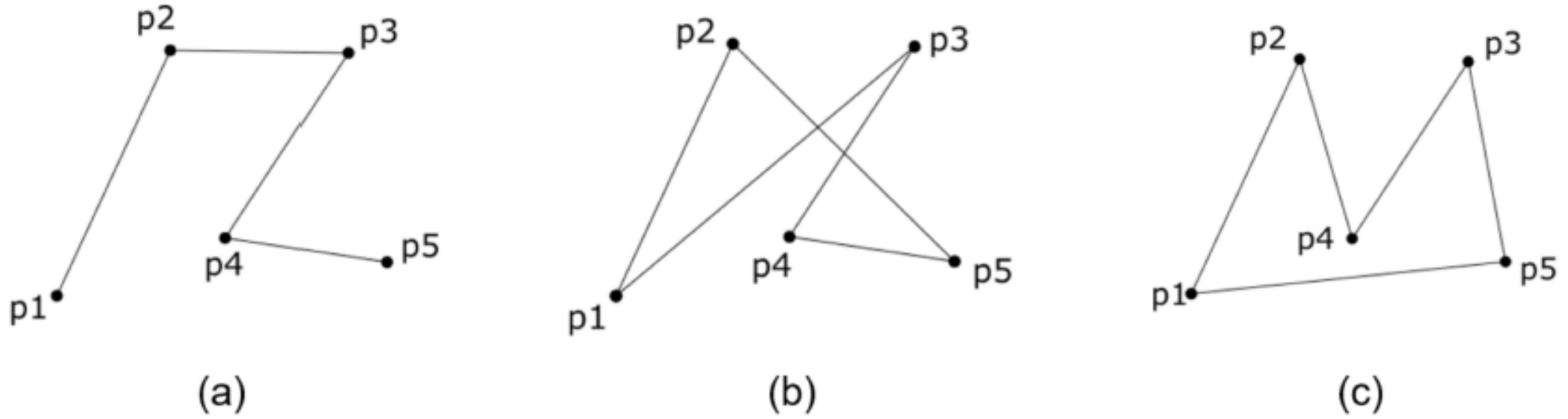
(HUGHES et al., 2013). Uma polilinha é um conjunto de vértices conectados por segmentos de reta (arestas). Com polilinhas é possível desenhar diversas formas geométricas, inclusive polígonos. A Figura 2.8 mostra exemplos de formas geométricas representadas por polilinhas. A Figura 2.8(c) mostra um polígono construído por uma polilinha.

As três formas apresentadas na Figura 2.8 podem ser representadas em linguagem de programação como sequências de vértices, ordenadas. Em Python podemos escrever da seguinte forma (considere já declarados os vértices, por exemplo, como p1=(x1,y1)):

```
polilinhaA = [p1, p2, p3, p4, p5]
polilinhaB = [p1, p2, p5, p4, p3, p1]
polilinhaC = [p1, p2, p4, p3, p5, p1]
```

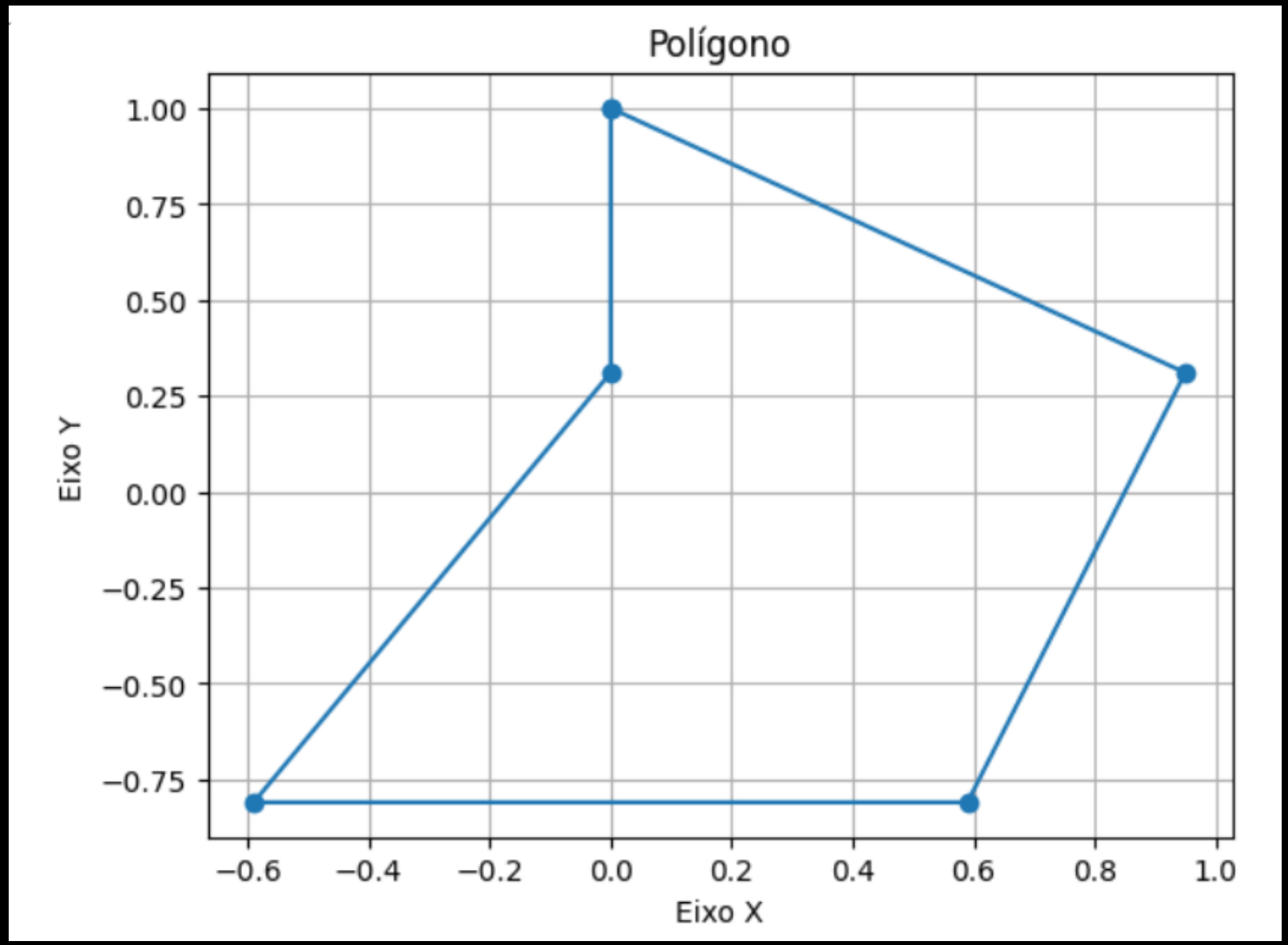
As polilinhas das Figura 2.8(a), (b) e (c) são representadas pela lista *vértices* e pelas listas de arestas *arestasA*, *arestasB* e *arestasC*, respectivamente.

Figura 2.8 | Formas geométricas representadas por polilinhas



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

def plot_polygon_with_lines(points):
    x_coords = [point[0] for point in points]
    y_coords = [point[1] for point in points]

    plt.plot(x_coords + [x_coords[0]], y_coords + [y_coords[0]], marker='o')
    plt.xlabel("Eixo X")
    plt.ylabel("Eixo Y")
    plt.title("Polígono")
    plt.grid(True)
    plt.show()
```

```python
# Coordenadas dos vértices do pentágono
vertices_pentagono = [
        (0, 1),
        (0.95, 0.31),
        (0.59, -0.81),
        (-0.59, -0.81),
        (0, 0.31)
]

# Plotar o pentágono
plot_polygon_with_lines(vertices_pentagono)
```


Polígono
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Eixo Y
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
Eixo X

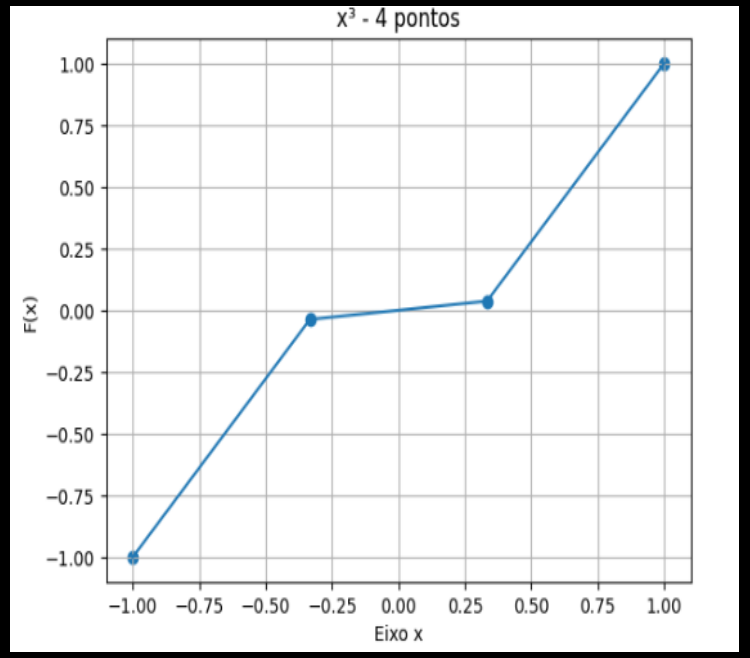
As polilinhas são aproximações de curvas, mas se conhecermos a equação da curva, podemos no espaço contínuo pôr sua equação. Desta forma a curva poderá ser desenhada em qualquer resolução espacial, por algoritmos similares aos de desenhos de reta e círculo.

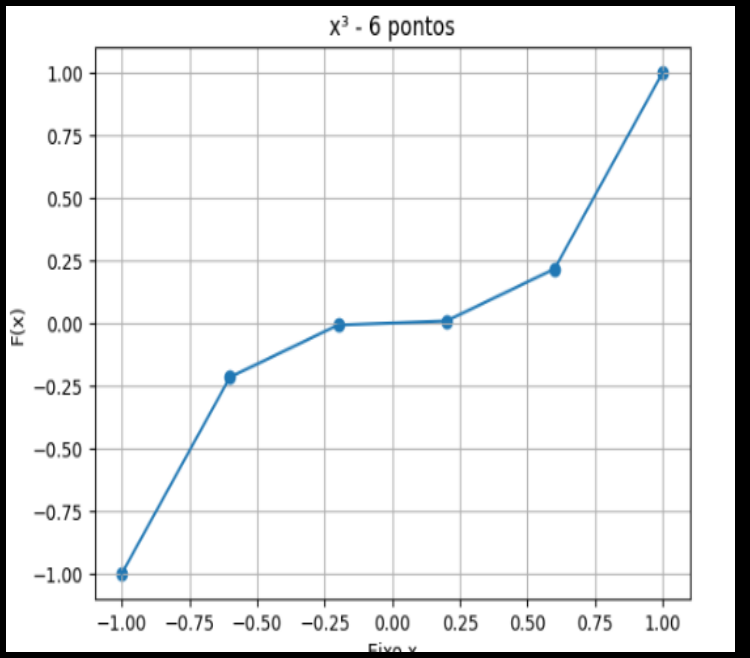
A curva da Figura 2.7(a) é uma curva descrita por equações e por isso pôde ser desenhada pixel a pixel, mantendo a aparência de uma curva contínua. As curvas mais utilizadas são curvas paramétricas denominadas *splines* (HEARN et al., 2013).

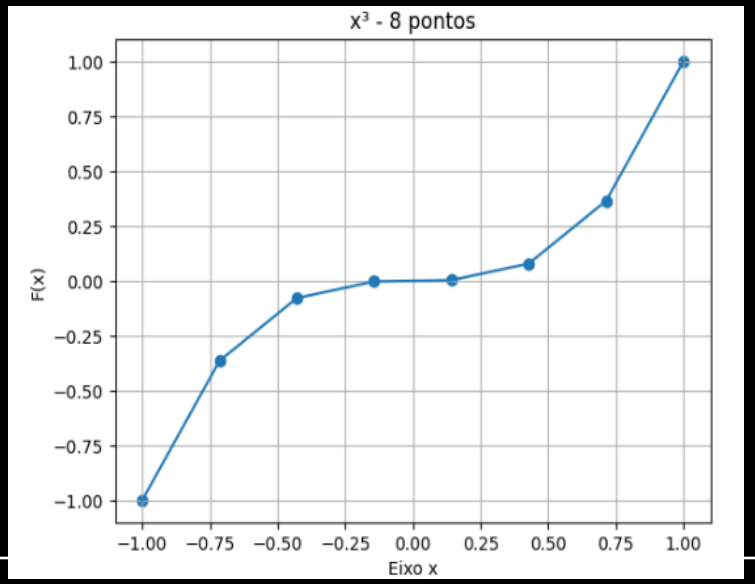
Splines

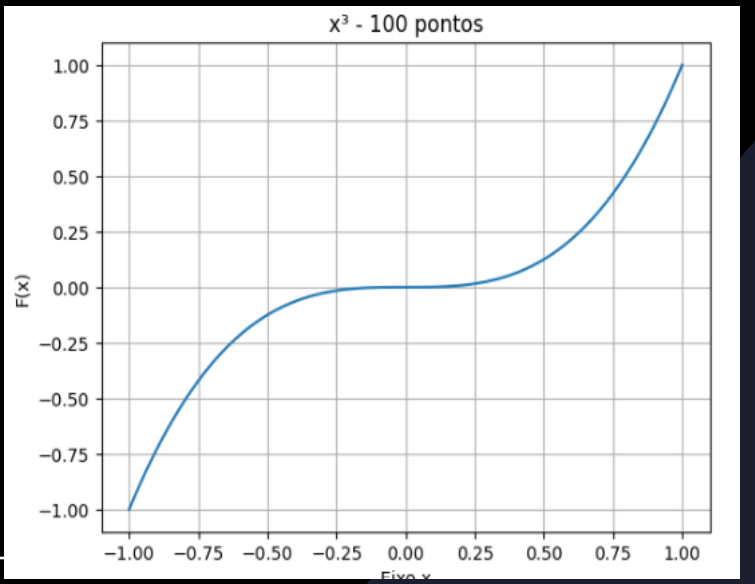
Uma *spline* é qualquer curva composta formada por secções polinomiais satisfazendo algum critério de continuidade nas junções das secções (HEARN et al., 2013). Enquanto as polilinhas são sequências de segmentos de reta, as *splines* são sequências de curvas polinomiais.

```python
# prompt: desenhar grafico de uma função

import numpy as np
import matplotlib.pyplot as plt

# Define a função que você quer plotar
def f(x):
  return x**3  # Exemplo: função quadrática

# Cria um array de valores de x
x = np.linspace(-1,1, 4)  # Valores de x de -

# Calcula os valores correspondentes de y
y = f(x)

# Plota o gráfico
plt.plot(x, y)
plt.scatter(x,y)

plt.xlabel("Eixo x")
plt.ylabel("F(x)")
plt.title("x³ - 4 pontos")
plt.grid(True)

plt.show()
```







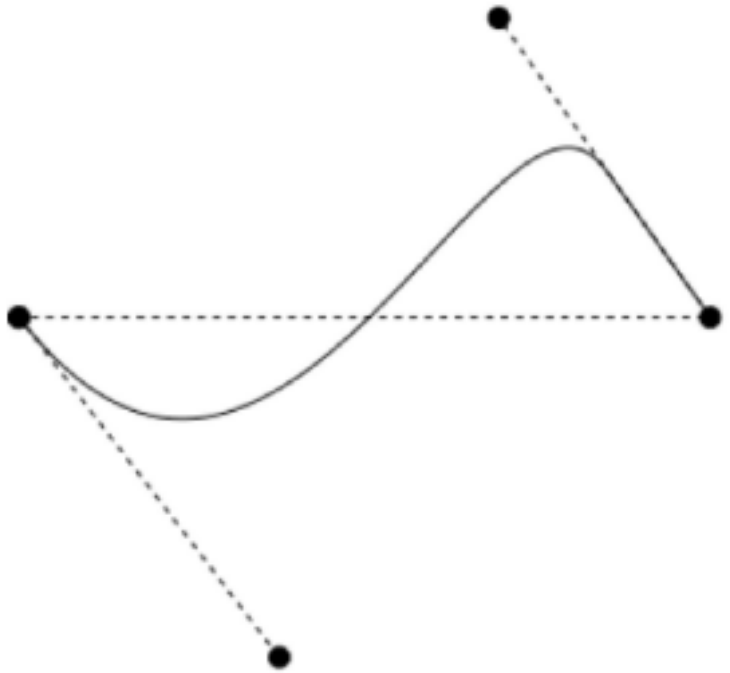


Primeiramente vamos especificar uma única secção polinomial. Um polinômio de ordem três é descrito pela equação $f(t) = at^3 + bt^2 + ct + d$, que pode ser reescrita na forma matricial

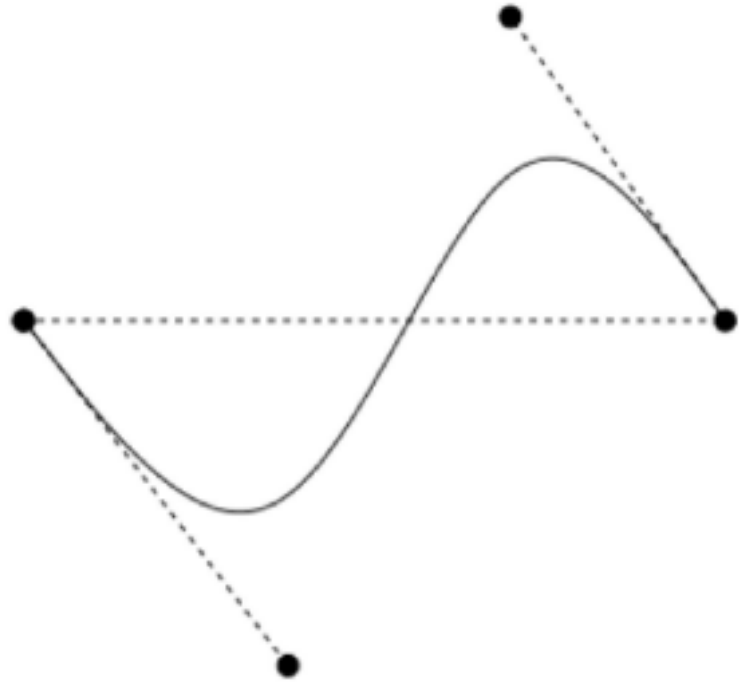
$$f(x) = [d,c,b,a]\begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} = C \cdot T$$

onde T são as potências do parâmetro t e C é a matriz de coeficientes. C pode ser escrita como $C = G \cdot M$, onde G é a matriz geometria, contendo as restrições geométricas da *spline*, e M é a matriz base, uma matriz *4x4* que transforma as restrições geométricas em coeficientes e provê uma caracterização da *spline* (HUGHES et al., 2013). A Figura 2.9 apresenta duas *splines* com as mesmas restrições geométricas (G) e caracterização diferente (M). Finalmente, a equação de uma *spline* é dada por $f(t) = G \cdot M \cdot T$.

Figura 2.9 | Exemplos de ***splines*** de características diferentes e restrições geométricas iguais: (a) curva de Bézier; (b), B-Spline. A linha pontilhada mostra a ordem dos pontos de controle.



(a)



(b)

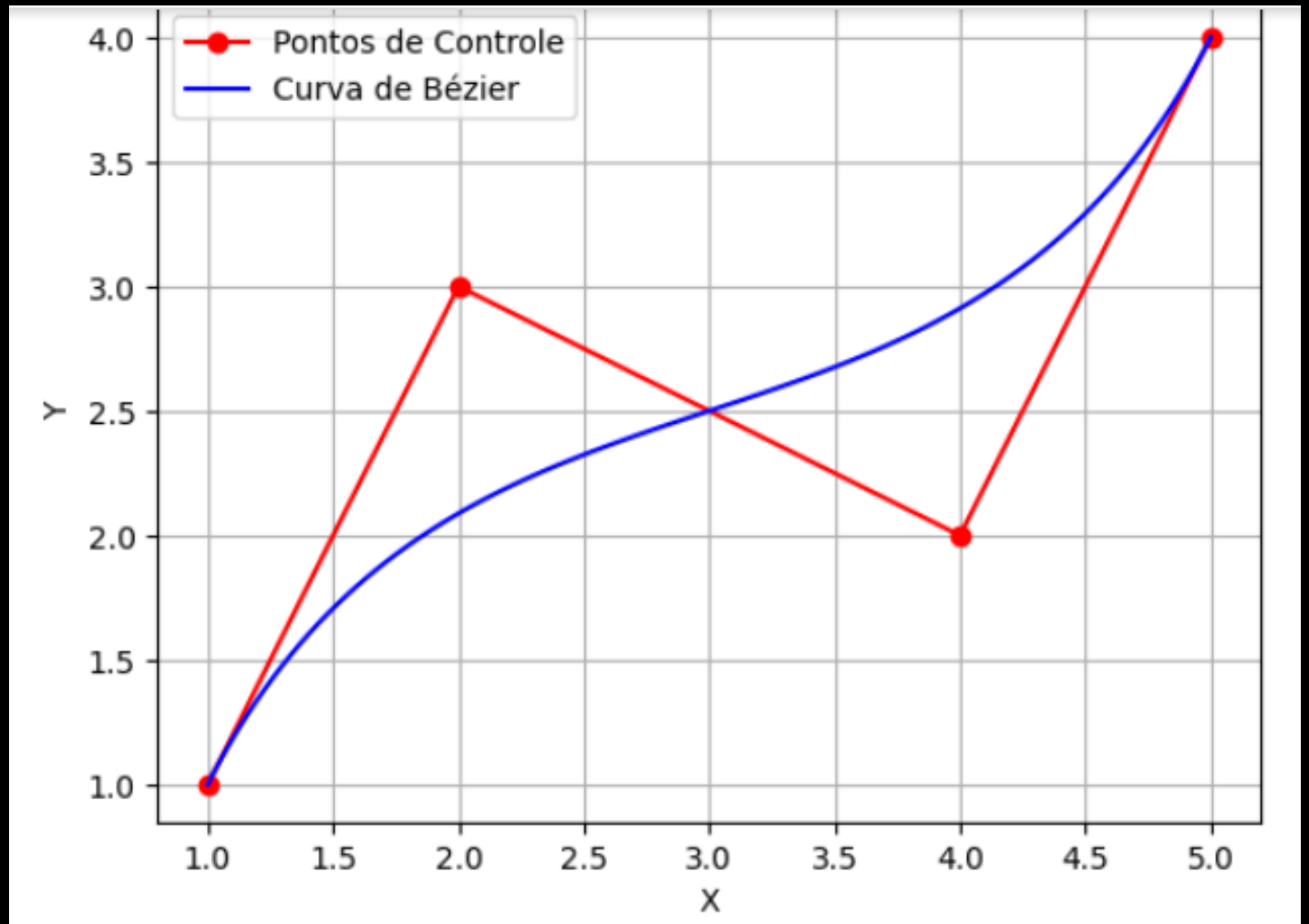
Curvas de Bézier

Curvas de Bézier são *splines* com uma característica específica. Foram propostas por Pierre Bézier para o design de automóveis (HEARN et al., 2013). As curvas de Bézier são muito utilizadas para descrever curvas suaves obtidas a partir de pontos de referência, denominados **pontos de controle**. Os pontos de controle de uma curva de Bézier formam a geometria da *spline* (G), e a matriz base da *spline* (M), que define a característica da curva polinomial, é obtida a partir da definição polinomial dessa *spline*.

A definição polinomial de uma Curva de Bézier é dada por $B(t)=\sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^{i}P_{i},0\leq t\leq 1$, onde $\binom{n}{i}=\frac{n!}{i!(n-i)!}$ é o coeficiente binomial e P_i são os pontos de controle.

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

def binomial_coefficient(n, i):
  return np.math.factorial(n) / (np.math.factorial(i) * np.math.factorial(n - i))

def bernstein_polynomial(n, i, t):
  return binomial_coefficient(n, i) * (t ** i) * ((1 - t) ** (n - i))

def bezier_curve(control_points, num_points=100):
  n = len(control_points) - 1
  curve_points = []
  for t in np.linspace(0, 1, num_points):
    x = 0
    y = 0
    for i in range(n + 1):
      basis = bernstein_polynomial(n, i, t)
      x += basis * control_points[i][0]
      y += basis * control_points[i][1]
    curve_points.append((x, y))
  return curve_points

# Pontos de controle da curva
control_points = [(1, 1), (2, 3), (4, 2), (5, 4)]
# Calcula a curva de Bézier
curve_points = bezier_curve(control_points)
```

```python
# Separa as coordenadas x e y dos pontos da curva
x_curve = [point[0] for point in curve_points]
y_curve = [point[1] for point in curve_points]

# Plota os pontos de controle e a curva
plt.plot([p[0] for p in control_points], [p[1] for p in control_points],
         'ro-', label='Pontos de Controle')
plt.plot(x_curve, y_curve, 'b-', label='Curva de Bézier')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Curva de Bézier')
plt.legend()
plt.grid(True)
plt.show()

# Equação do polinômio de Bézier
n = len(control_points) - 1
print("Equação do polinômio de Bézier:")
for i in range(n + 1):
    print(f"  {binomial_coefficient(n, i):.0f} * t^{i} * (1-t)^{n-i} * P{i}")
```




```
Equação do polinômio de Bézier:
  1 * t^0 * (1-t)^3 * P0
  3 * t^1 * (1-t)^2 * P1
  3 * t^2 * (1-t)^1 * P2
  1 * t^3 * (1-t)^0 * P3
```

```python
# prompt: desenhar uma curva de Bézier 3d

def bezier_curve_3d(control_points, num_points=100):
  n = len(control_points) - 1
  curve_points = []
  for t in np.linspace(0, 1, num_points):
    x = 0
    y = 0
    z = 0
    for i in range(n + 1):
      basis = bernstein_polynomial(n, i, t)
      x += basis * control_points[i][0]
      y += basis * control_points[i][1]
      z += basis * control_points[i][2]
    curve_points.append((x, y, z))
  return curve_points

# Pontos de controle da curva 3D
control_points_3d = [(1, 1, 1), (2, 3, 2), (4, 2, 3), (5, 4, 1)]

# Calcula a curva de Bézier 3D
curve_points_3d = bezier_curve_3d(control_points_3d)

# Separa as coordenadas x, y e z dos pontos da curva
x_curve_3d = [point[0] for point in curve_points_3d]
y_curve_3d = [point[1] for point in curve_points_3d]
z_curve_3d = [point[2] for point in curve_points_3d]
```

```python
# Plota a curva em 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_curve_3d, y_curve_3d, z_curve_3d, label='Curva de Bézier 3D')
ax.scatter([p[0] for p in control_points_3d], [p[1] for p in control_points_3d],
           [p[2] for p in control_points_3d], color='red', label='Pontos de Controle')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Curva de Bézier 3D')
ax.legend()
plt.show()
```

B-Splines

B-Splines são, junto às curvas de Bézier, as *splines* mais usadas. Assim como as curvas de Bézier, B-Splines são geradas pela aproximação de um conjunto de pontos de controle. Curvas B-Splines são mais complexas que curvas de Bézier, mas possuem algumas vantagens. Uma vantagem é que o grau do polinômio da B-Spline pode ser definido independentemente do número de pontos de controle (com algumas limitações). Veremos aqui apenas as B-Splines cúbicas. A função base de uma B-Spline cúbica é

$$b_3(t) = \frac{1}{6}t^3, 0 \leq t \leq 1$$

$$b_3(t) = \frac{1}{6}(-3(t-1)^3 + 3(t-1)^2 + 3(t-1) + 1), 1 \leq t \leq 2$$

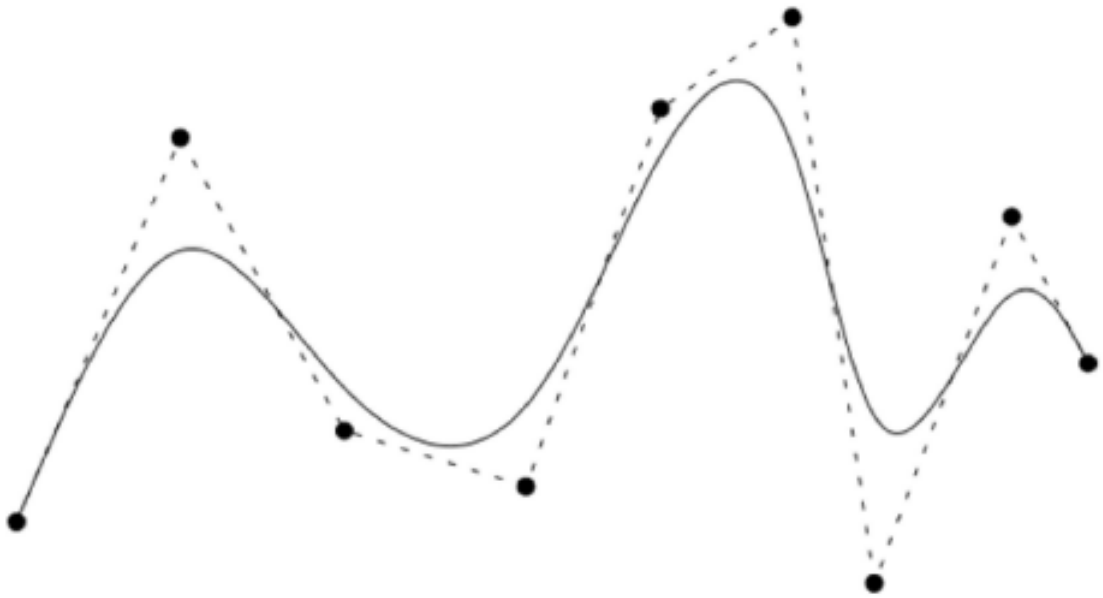
$$b_3(t) = \frac{1}{6}((t-2)^3 - 6(t-2)^2 + 4), 2 \leq t \leq 3$$

$$b_3(t) = \frac{1}{6}(-(t-3)^3 + 3(t-3)^2 - 3(t-3) + 1), 3 \leq t \leq 4$$

$$b_3(t) = 0, \textit{senão}$$

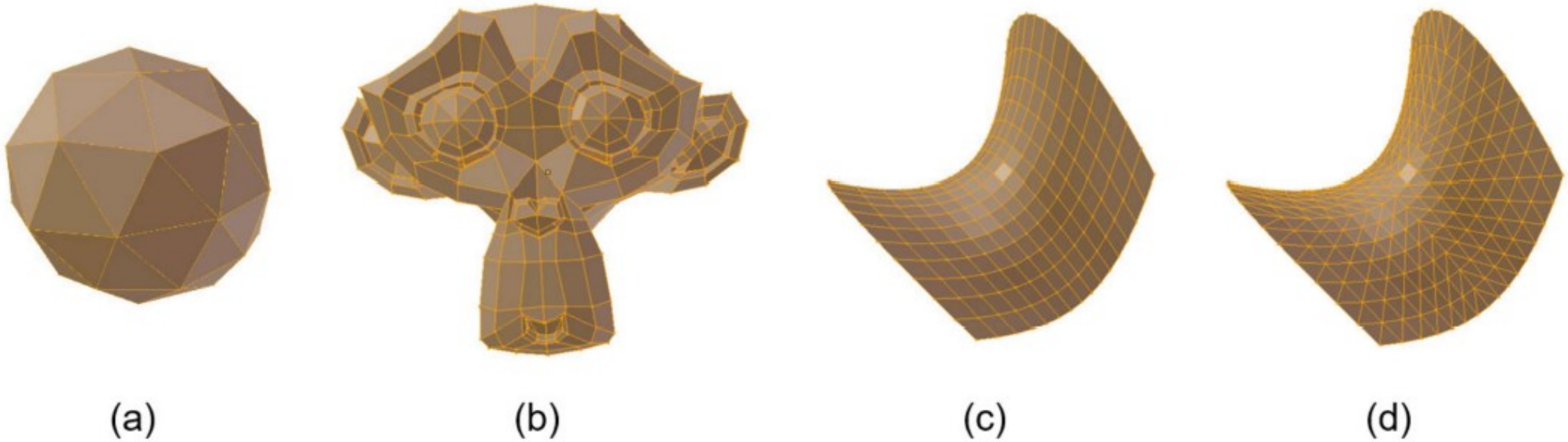
e a equação da B-Spline cúbica com pontos de controle P_0, ..., P_n é $B_s(t) = \sum_{i=0}^{n} P_i b_3(t-i)$. A Figura 2.10 apresenta uma B-Spline cúbica com 9 pontos de controle.

Figura 2.10 | B-Spline cúbica com 9 pontos de controle. A linha pontilhada mostra a sequência dos pontos de controle.



Em 3D temos o interesse de construir sólidos ou superfícies. Em duas dimensões, uma curva contínua é aproximada por uma curva mais simples, a polilinha. Em 3D os sólidos contínuos são aproximados por sólidos mais simples: os **poliedros**. O poliedro é um sólido tridimensional cujas faces são polígonos planares (BERG et al., 2008; ANGEL; SHREINER, 2012). A superfície de um poliedro é uma superfície fechada, que separa o interior do poliedro de seu exterior. Mas uma superfície contínua aberta também pode ser aproximada por um conjunto de polígonos planares. A superfície, aberta ou fechada, composta apenas por polígonos planares é denominada **malha poligonal**. A Figura 2.12 mostra algumas malhas poligonais.

Figura 2.12 | Poliedros e malhas: (a) aproximação da esfera por um poliedro de faces triangulares; (b) macaco desenhado por malha poligonal com diferentes polígonos; (c) cela desenhada por quadriláteros e (d) a mesma cela (c) desenhada por triângulos.

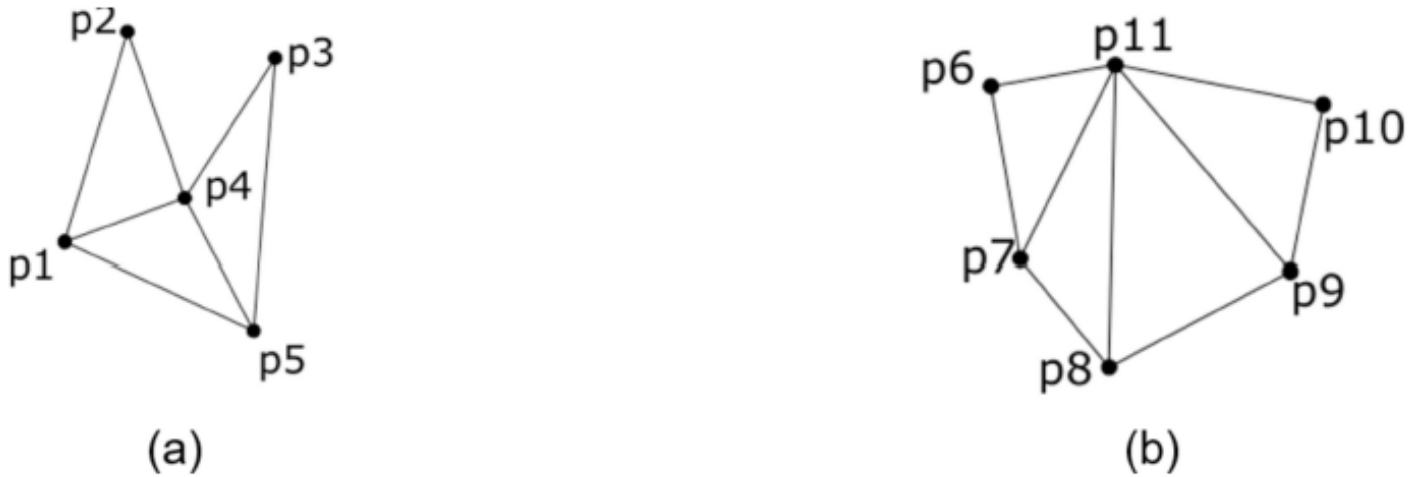


(a) (b) (c) (d)

Figura 2.12(c) para gerar a Figura 2.12(d) (OLIVEIRA, 2015). As malhas triangulares são as mais utilizadas por dois motivos: i) simplicidade na representação da malha em memória e ii) o fato de que quaisquer três pontos diferentes no espaço 3D formam um polígono planar convexo, o triângulo, o que não é garantido para polígono com mais lados. Por esses motivos básicos as malhas triangulares já foram extensivamente estudadas.

Há diversas estruturas de dados possíveis para a representação em memória de uma malha triangular, e a eficiência dos algoritmos de manipulação de malhas depende da estrutura de dados utilizada (OLIVEIRA, 2015). Veremos a seguir a estrutura de dados mais intuitiva, que se assemelha ao que foi apresentado para as polilinhas. Para ilustrar, a Figura 2.13 apresenta triangulações dos polígonos apresentados nas Figuras 2.11(b) e (e).

Figura 2.13 | Malhas triangulares construídas a partir de polígonos da Figura 2.11.

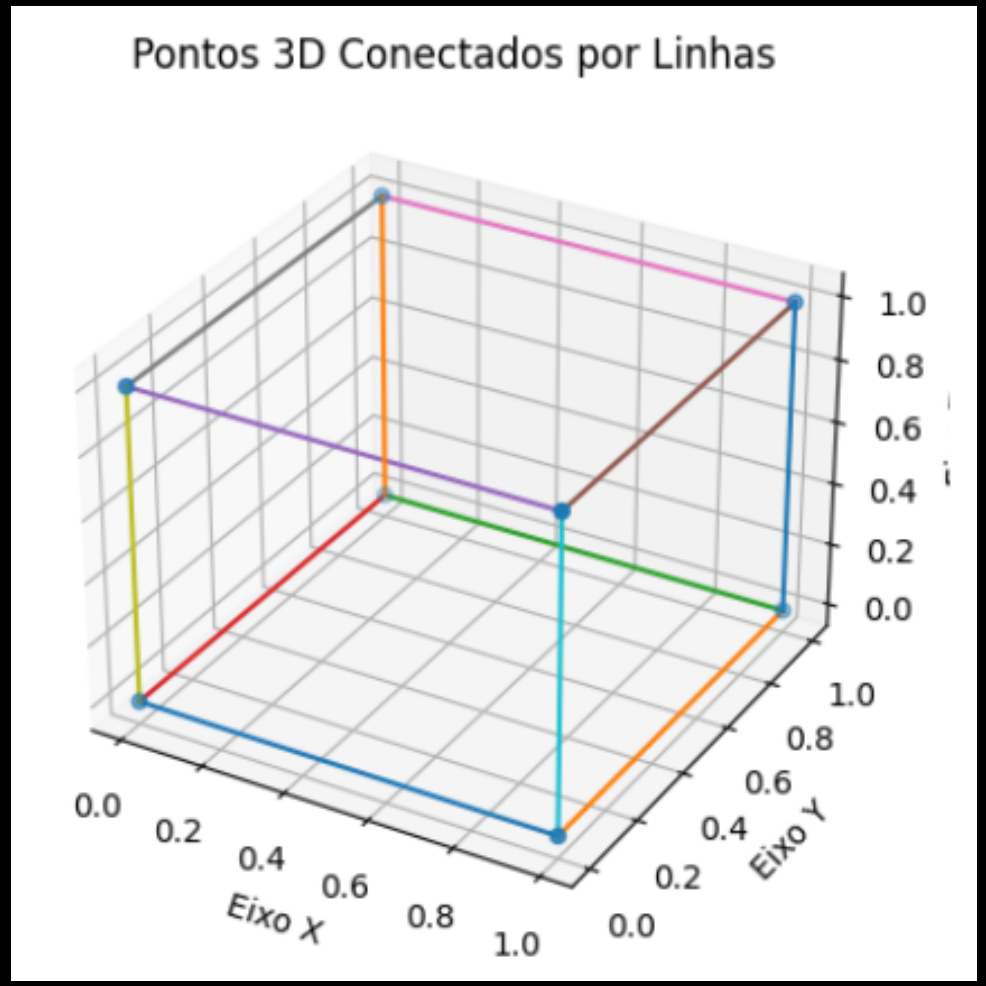


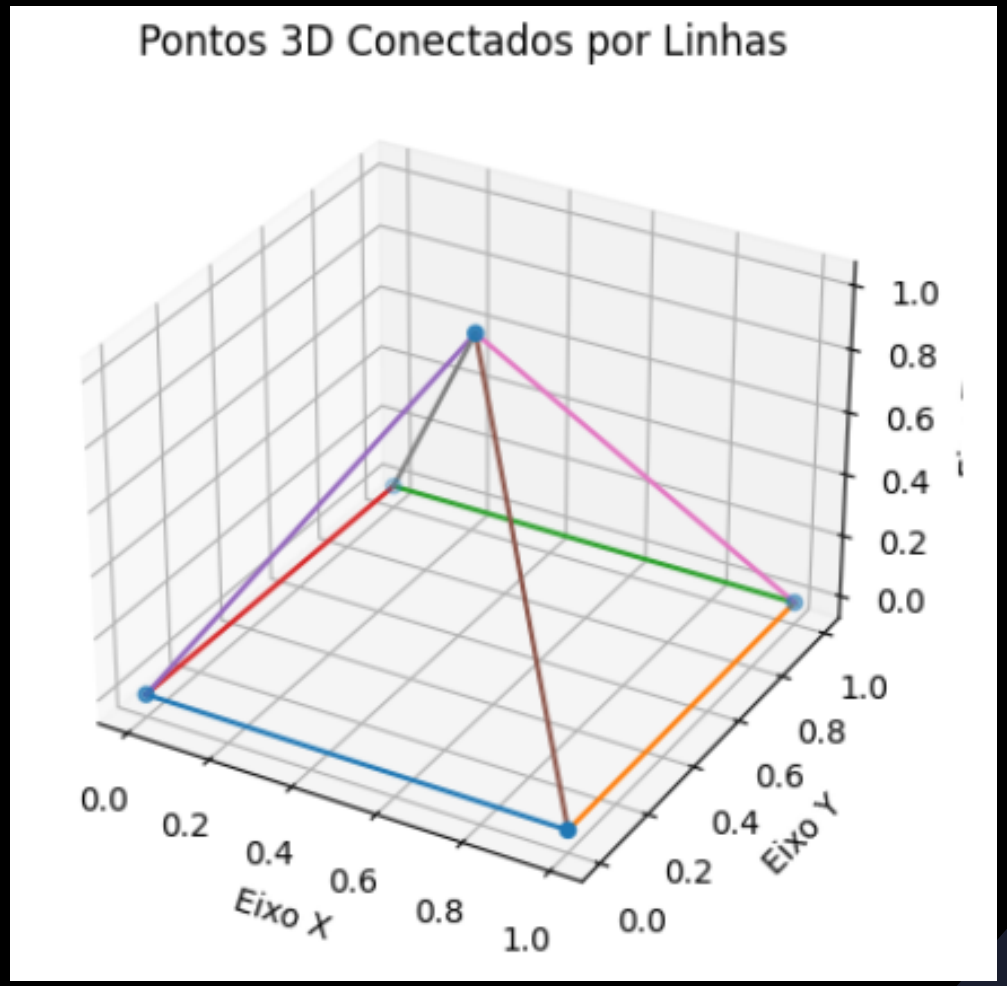
A malha triangular pode ser representada por duas listas: uma lista de vértices e uma lista de triângulos, como mostra o código em Python a seguir:

```python
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

def plot_3d_points_with_lines(points, edges=None, labels=None):

  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')

  x_coords = [point[0] for point in points]
  y_coords = [point[1] for point in points]
  z_coords = [point[2] for point in points]

  # Plotar os pontos
  ax.scatter(x_coords, y_coords, z_coords)

  # Conectar os pontos com linhas (apenas as arestas especificadas)
  if edges:
    for edge in edges:
      i, j = edge
      ax.plot([x_coords[i], x_coords[j]], [y_coords[i], y_coords[j]], [z_coords[i], z_coords[j]])

  if labels:
    for i, label in enumerate(labels):
      ax.text(x_coords[i], y_coords[i], z_coords[i], label)

  ax.set_xlabel("Eixo X")
  ax.set_ylabel("Eixo Y")
  ax.set_zlabel("Eixo Z")
  ax.set_title("Pontos 3D Conectados por Linhas")
  plt.show()
```

```python
# Definir os vértices do cubo
vertices = [
    (0, 0, 0),  # Vértice 1
    (1, 0, 0),  # Vértice 2
    (1, 1, 0),  # Vértice 3
    (0, 1, 0),  # Vértice 4
    (0, 0, 1),  # Vértice 5
    (1, 0, 1),  # Vértice 6
    (1, 1, 1),  # Vértice 7
    (0, 1, 1)   # Vértice 8
]

# Lista de arestas do cubo (conexões entre vértices)
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),  # Face inferior
    (4, 5), (5, 6), (6, 7), (7, 4),  # Face superior
    (0, 4), (1, 5), (2, 6), (3, 7)   # Arestas verticais
]

# Plotar o cubo, mostrando apenas as arestas
plot_3d_points_with_lines(vertices, edges=edges)
```

```python
# Plotar o cubo, mostrando apenas as arestas
plot_3d_points_with_lines(vertices, edges=edges)

# Definir os vértices da pirâmide
vertices = [
    (0, 0, 0),  # Base: Vértice 1
    (1, 0, 0),  # Base: Vértice 2
    (1, 1, 0),  # Base: Vértice 3
    (0, 1, 0),  # Base: Vértice 4
    (0.5, 0.5, 1)  # Topo da pirâmide
]

# Lista de arestas da pirâmide (conexões entre vértices)
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),  # Arestas da base
    (0, 4), (1, 4), (2, 4), (3, 4)   # Arestas laterais
]

# Plotar a pirâmide
plot_3d_points_with_lines(vertices, edges=edges)
```


Pontos 3D Conectados por Linhas
Eixo X
Eixo Y
0.0
0.2
0.4
0.6
0.8
1.0

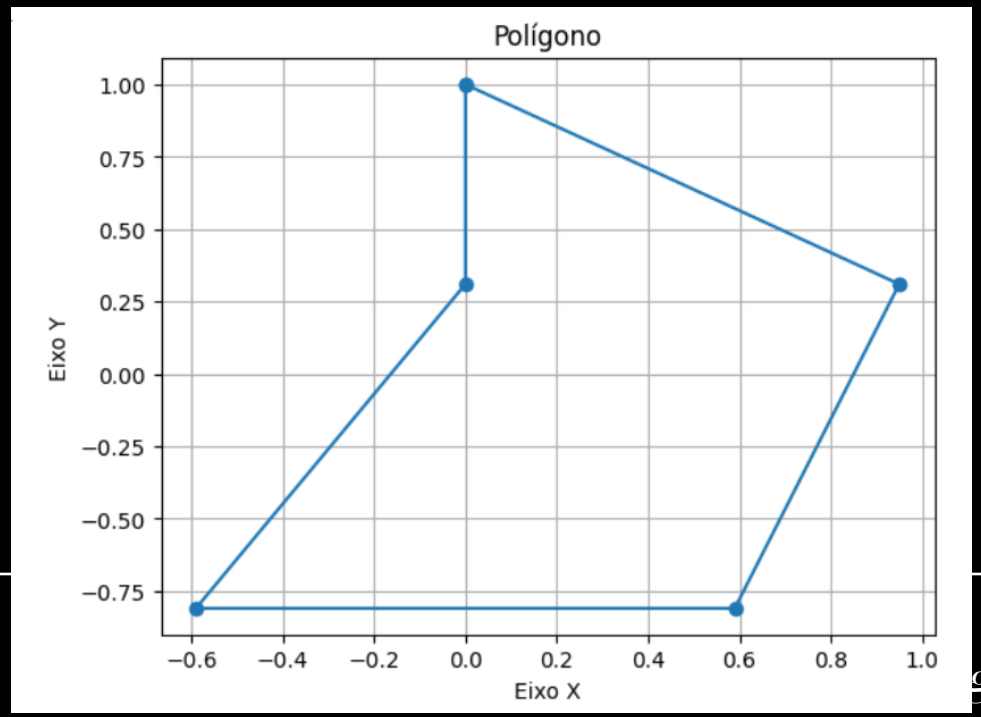

Pontos 3D Conectados por Linhas
Eixo X
Eixo Y
0.0
0.2
0.4
0.6
0.8
1.0

ATIVIDADE

. APLICAR UMA ROTAÇÃO DE 35° SOBRE O POLÍGONO

. DIVIDA O POLÍGONO UTILIZANDO UMA MALA TRIÂNGULAR (DEFINIR OS VÉRTICE E ARESTAS DE CADA TRIÂNGULO)

. ENCONTRE A CURVA DE BÉZIER PARA OS PONTOS DE CONTROLE ABAIXO:



P_0: $(0, 0)$

P_1: $(1, 2)$

P_2: $(2, -1)$

P_3: $(3, 2)$

P_4: $(4, 0)$