Geometric Representation

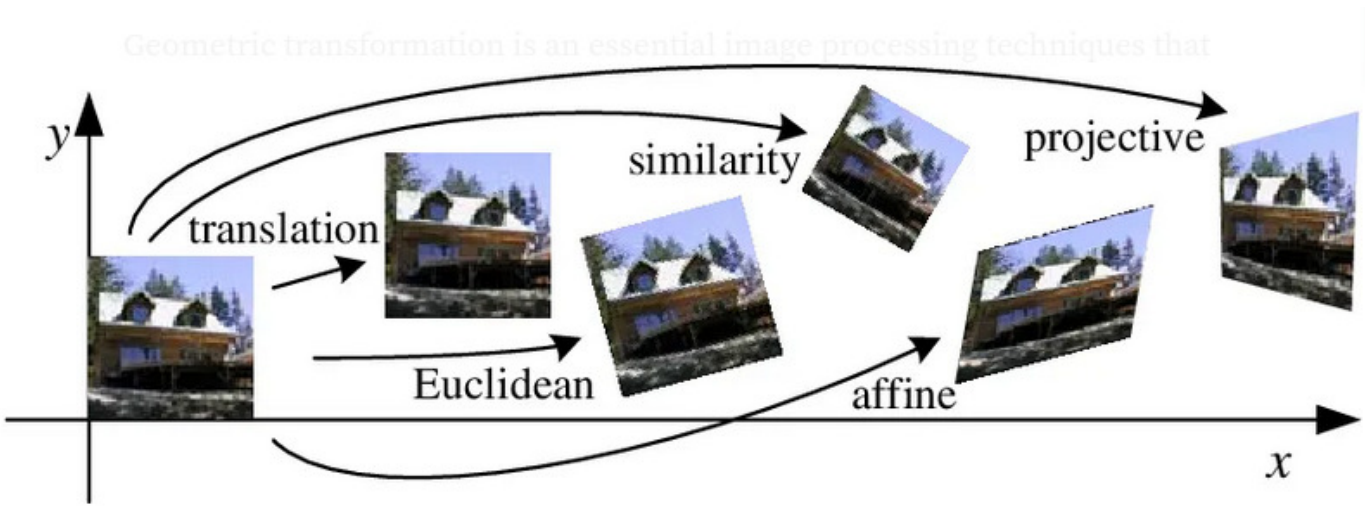
Biswajit Biswas
Ph.D. (C.S.E)

Department of Computer Science
Vivekananda Centenary College
Rahara, Kolkata

March 17, 2025

Geometric Transformations

- One of the major concepts in computer graphics is modeling of **objects**;
- A graphics system typically uses a set of geometric forms (**primitives**) that enough efficiently implement on the computer and easily manipulate **(assembled, deformed)** to a variety of 3D models;
- Geometric forms that are often used as primitives include, **points, lines, polylines, polygons, and polyhedra**;
- More complex geometric forms include **curves, curved surface patches, and quadric surfaces**;



Geometric Transformations

Figure: 3D solid Objectes

Geometric Transformations

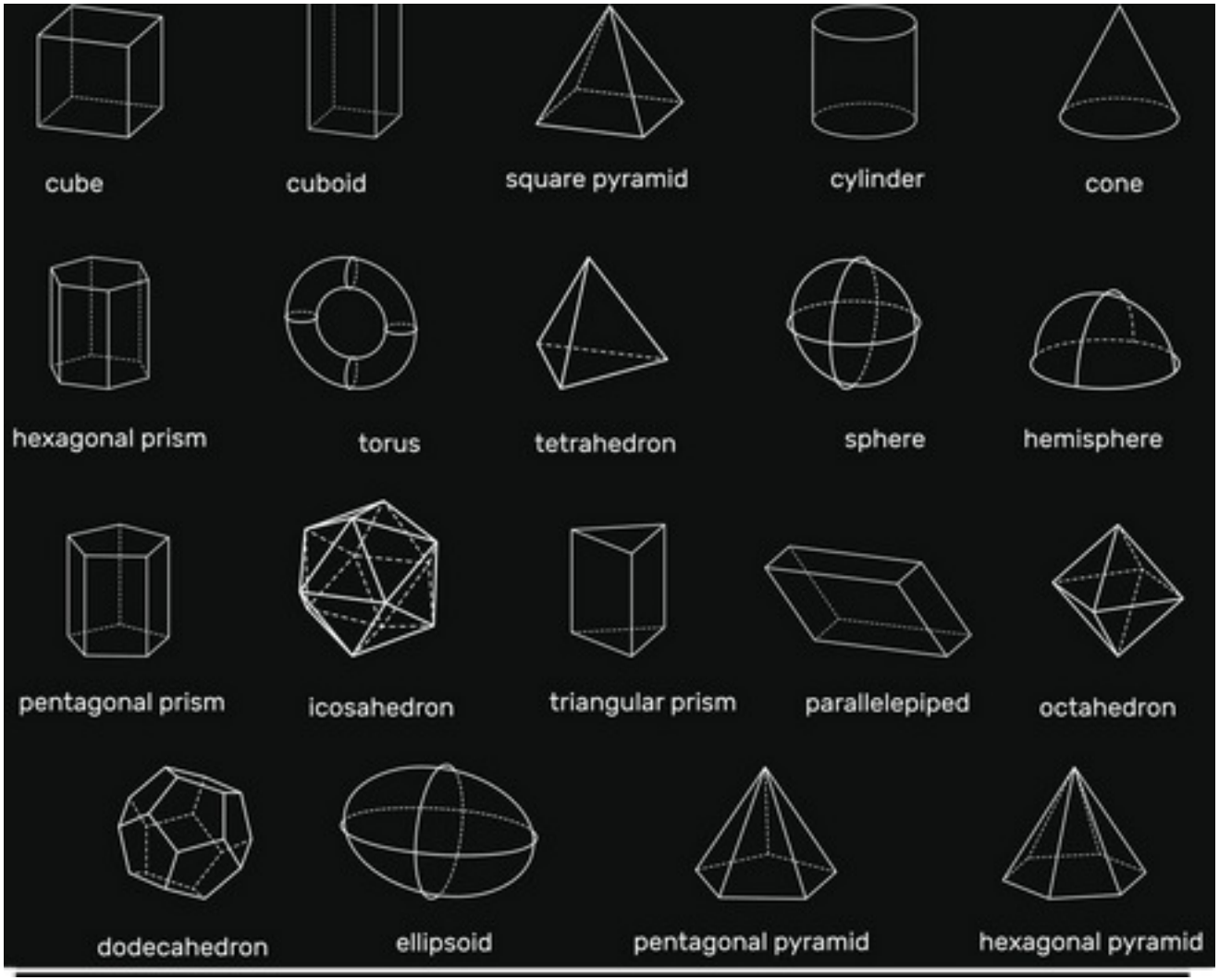
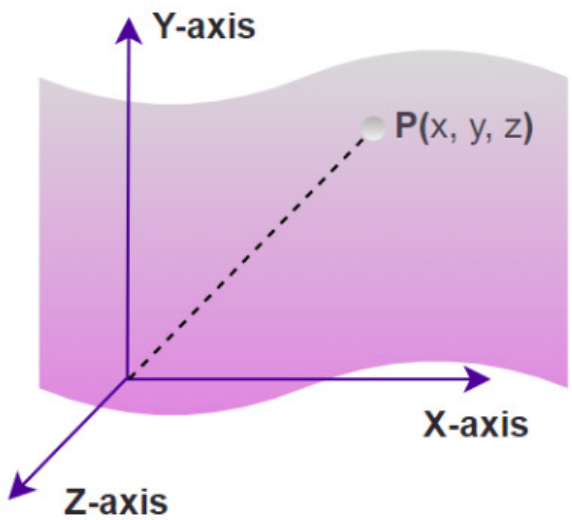


Figure: 3D Wire-frame model

Points and Lines

- **Points** and **Lines** are the basic building blocks of computer graphics;
- We specify a point by giving its coordinates in three- (or two-) dimensional space (3D/2D);
- A line or line segment is specified by giving its endpoints as: $P_1[x_1, y_1, z_1]$ and $P_2[x_2, y_2, z_2]$;



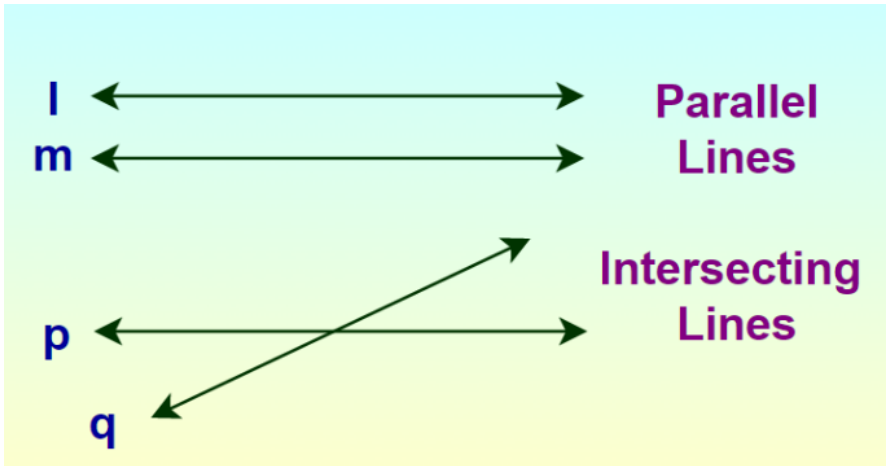


Figure: Points and Lines

Polylines

A polyline is a sequence of connected line segments treated as a single object, useful for representing lines, paths, or shapes in various applications like CAD, GIS, and map.

- A polyline is a chain of connected line segments;
- It is specified by giving the vertices (nodes) $P_0, P_1, \cdots, P_N$ defining the line segments;
- The first vertex is called the initial or starting point and the last vertex, the final or terminal point;

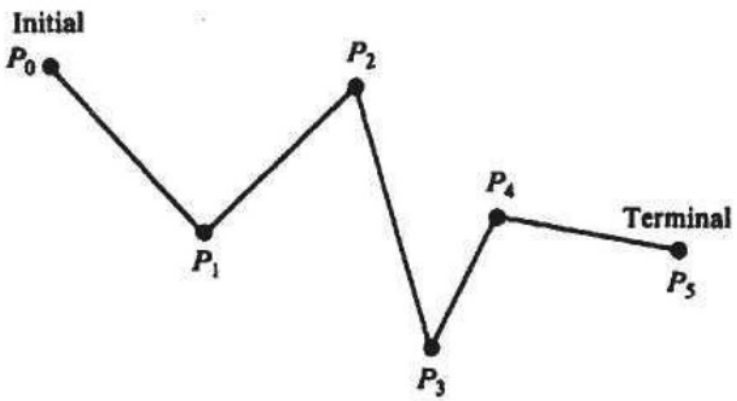


Figure: Polylines

Polygons

A polygon is a two-dimensional, closed shape formed by straight lines, with at least three sides and angles.

- A polygon is a closed polyline, that is, one in which the initial and terminal points coincide;
- A polygon is specified by its vertex list $P_0, P_1, \cdots, P_N, P_0$;
- The line segments $\overline{P_0P_1}, \overline{P_1P_2}, \cdots, \overline{P_NP_0}$ are called the edges of the polygon;
- A planar polygon is a polygon in which all vertices (and thus the entire polygon) lie on the same plane;

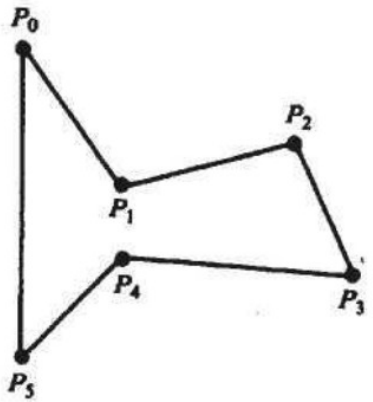


Figure: Polygons

Regular Polygon

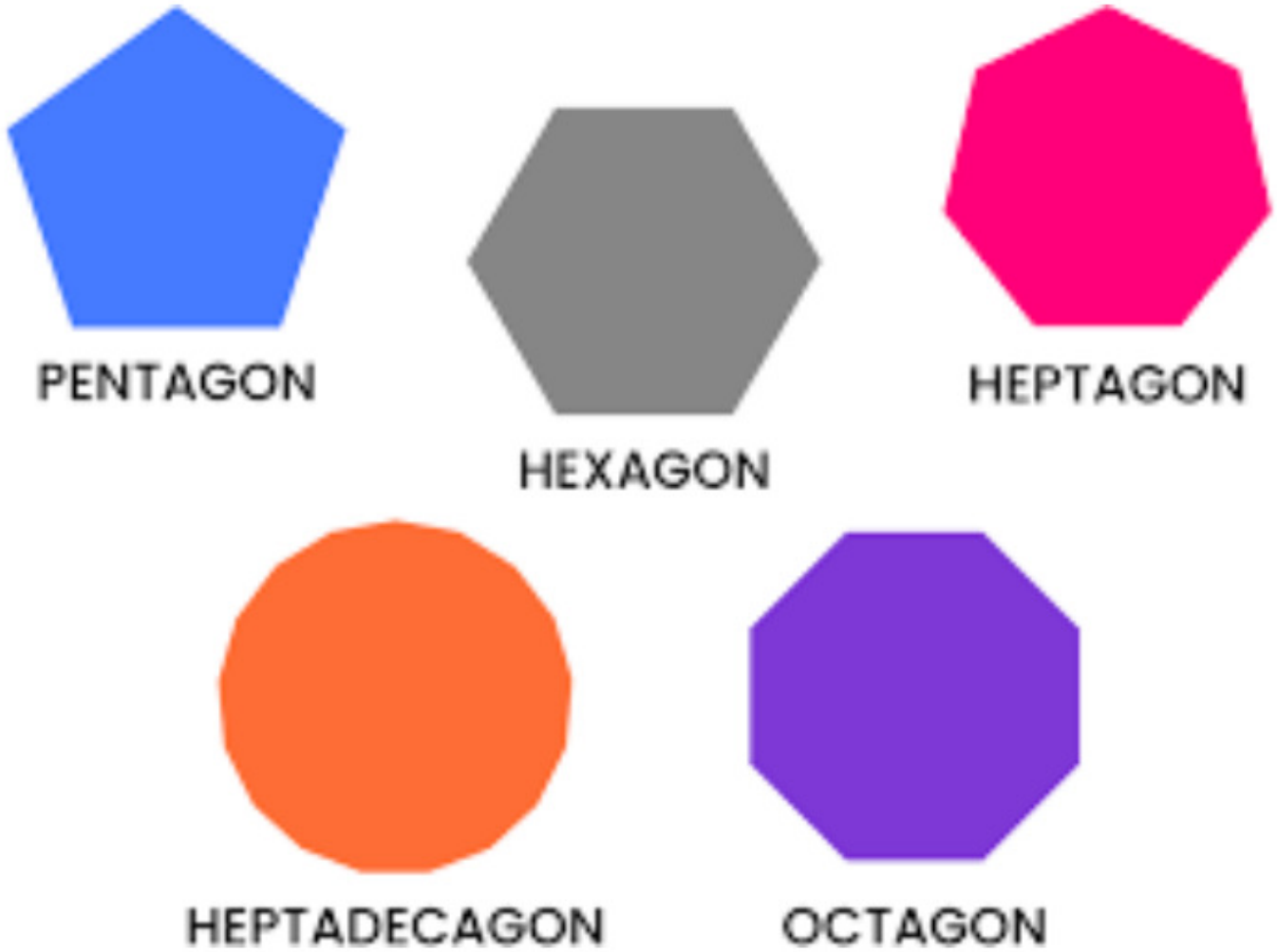


Figure: Regular Polygon

Polygons

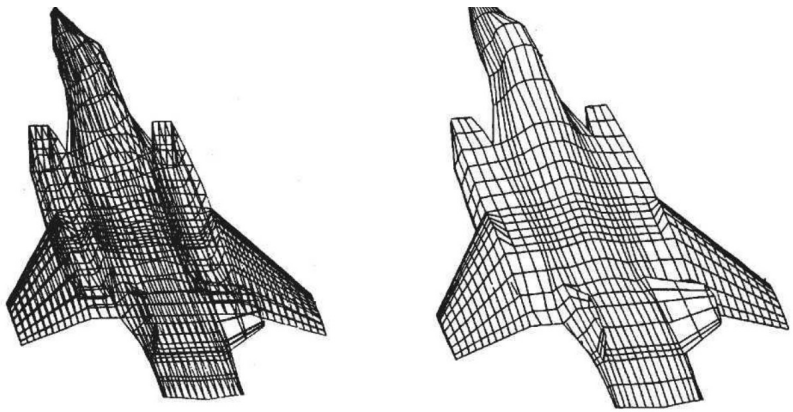
The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

y = np.array([[1,1],[2,1],[2,2],[1,2],[0.5,1.5]])

p = Polygon(y, facecolor='k')

fig,ax = plt.subplots()
ax.add_patch(p)
ax.set_xlim([0,3])
ax.set_ylim([0,3])
plt.show()
```

Wirefram Model

A **wireframe** model is a skeletal, simplified representation of a website, application, or 3D object, focusing on structure, user flow, and functionality using lines, shapes, and basic elements before adding visual details or content.

- A wirefram model consists of edges, vertices, and polygons;
- All vertices are connected by edges, and polygons are sequences of vertices or edges;
- The edges may be curved or straight line segments;
- The wireframe model is called a polygonal net or polygonal mesh;



(*a*) Wire frame model. (*b*) Hidden lines removed.

Wirefram Model

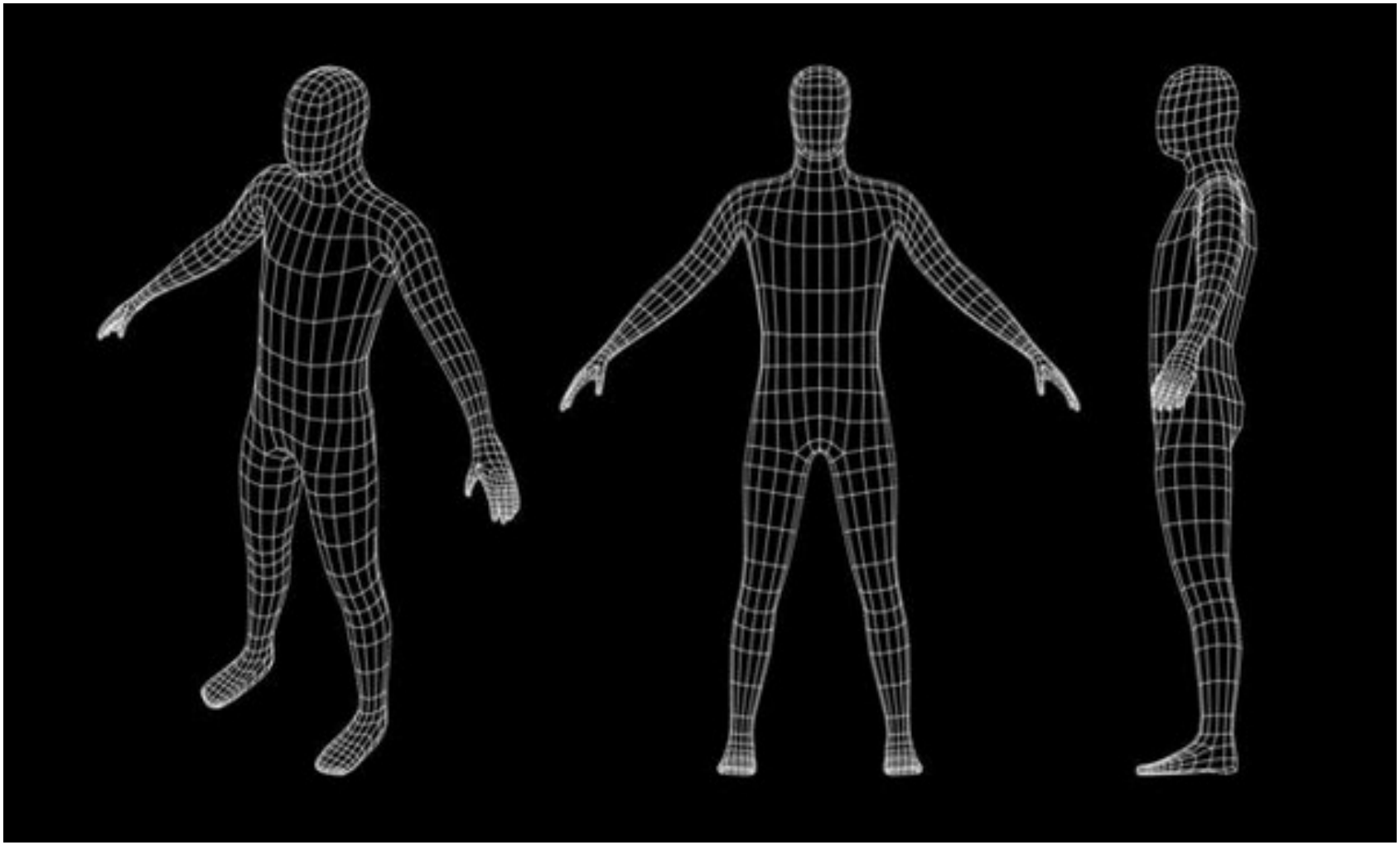
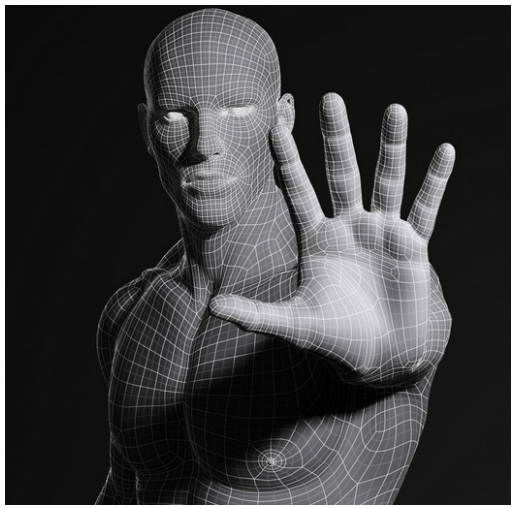


Figure: Wirefram Model

Advantages and Disadvantages of Wireframe Models

- Wireframe models are used in engineering applications.
- They are easy to construct composed of straight lines, clip and manipulate through the use of geometric and coordinate transformations.
- For building realistic models, especially of highly curved objects, the wireframe model is inefficient;
- To achieve the illusions of roundness and smoothness for very large number of polygons, the wireframe model is unable to visualize proper context.



Representing a Polygonal Net Model

There are several different ways of representing a polygonal net model.

- **Explicit vertex list** $\mathrm{V} = \{P_0, P_1, \cdots, P_N\}$: The points $P_i(x_i, y_i, z_i)$ are the vertices of the polygonal net, stored in the order in which they would be encountered by traveling around the model;
- **Polygon listing**: In this form of representation, each vertex is stored exactly once in a vertex list $\mathrm{V} = (P_0, P_1, \cdots, P_N)$, and each polygon is defined by pointing or indexing into this vertex list;
- **Explicit edge listing**: In this form of representation, we keep a vertex list in which each vertex is stored exactly once and an edge list in which each edge is stored exactly once. Each edge in the edge list points to the two vertices in the vertex list which define that edge.

Polyhedron

A polyhedron is a 3D solid made up of polygons and contains flat faces, straight edges, and vertices.

- A polyhedron is a closed polygonal net (i.e., one which encloses a definite volume) in which each polygon is planar;
- The polygons are called the faces of the polyhedron;
- In modeling, polyhedrons are often treated as solid (i.e., block) objects, as opposed to wireframes or two-dimensional surfaces;

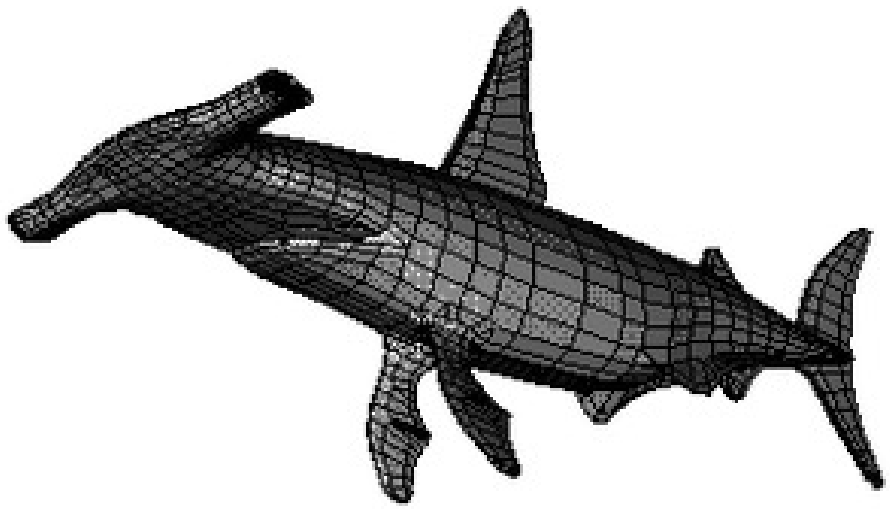


Figure: Polyhedron

Polyhedron

- A polyhedral surface Polyhedron consists of vertices **V**, edges **E**, facets **F** and an incidence relation on them.
- Each edge is represented by two halfedges with opposite orientations.
- Vertices represent points in 3d-space. Edges are straight line segments between two endpoints.

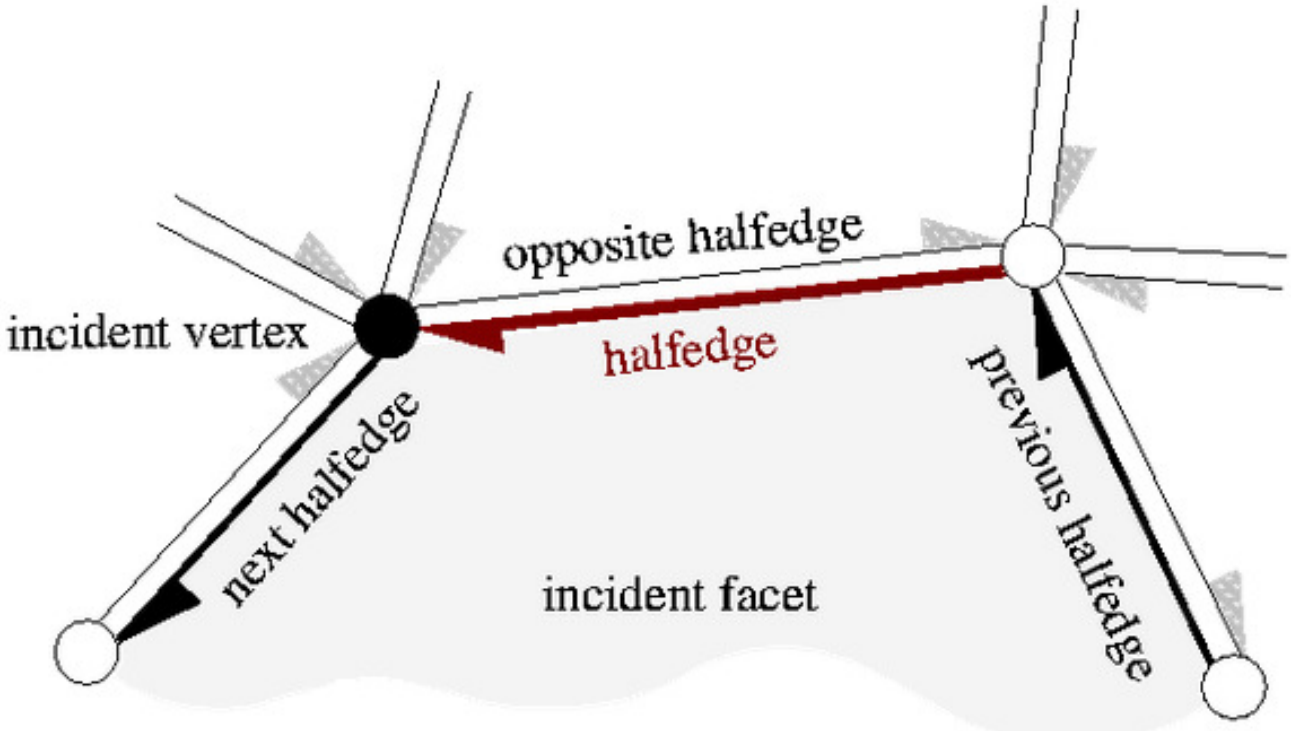


Figure: Polyhedron

Polyhedron

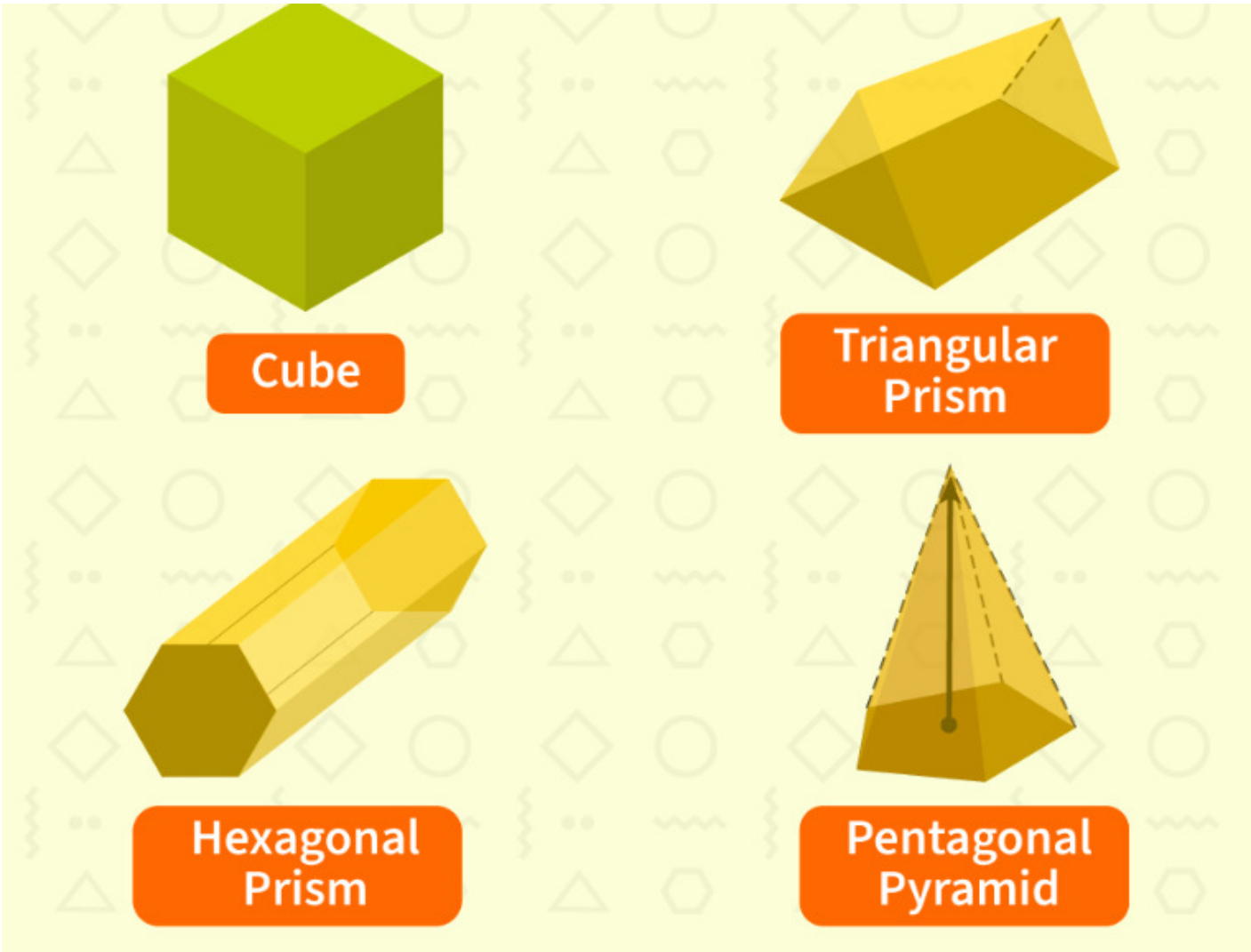


Figure: Polyhedron

Polyhedron

The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

y = np.array([[1,1],[2,1],[2,2],[1,2],[0.5,1.5]])

p = Polygon(y, facecolor='k')

fig,ax = plt.subplots()
ax.add_patch(p)
ax.set_xlim([0,3])
ax.set_ylim([0,3])
plt.show()
```

Curved and Surfaces

- The use of curved surfaces allows for a higher level of realistic modeling;
- There are several approaches to modeling curved surfaces. One is the polyhedral models;
- An object be modeled by using small, curved surface patches placed next to each other without using polygons;
- The surfaces approach be used to defined solid objects, such as polyhedra, spheres, cylinders, and cones. A model can then be constructed with these solid objects used as building blocks as called **solid modeling**;

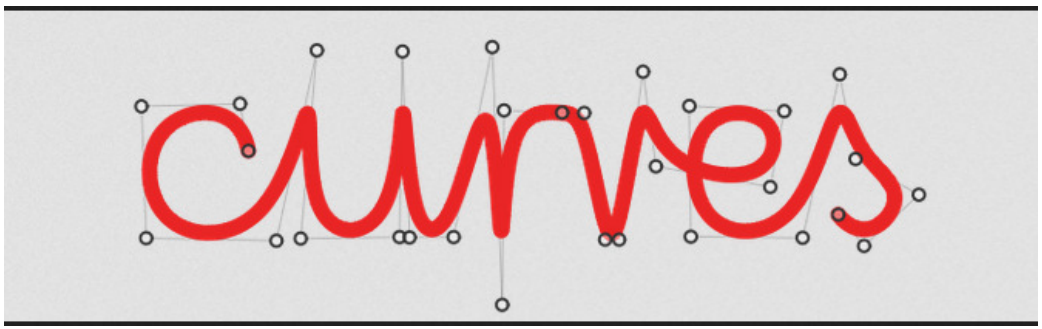
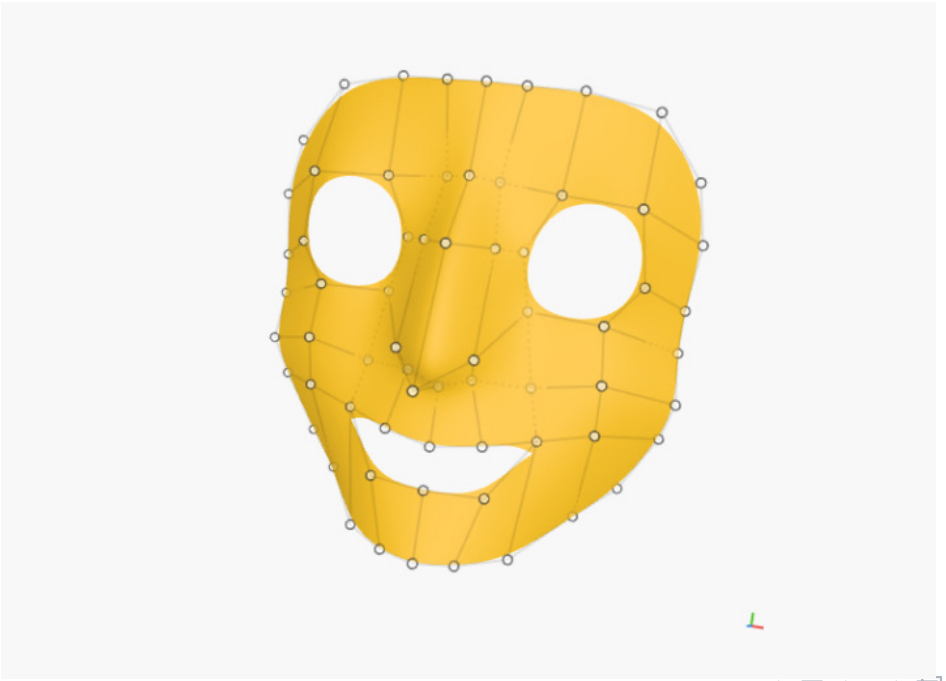


Figure: Curved and Surfaces

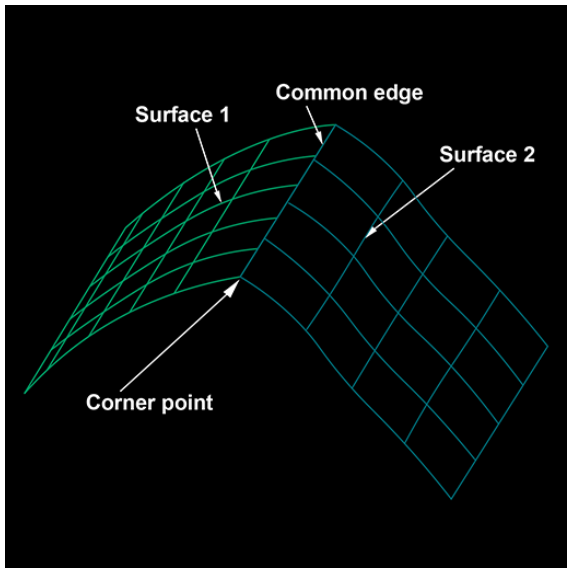
Curved and Surfaces

- There are two ways to construct a **model—additive modeling** and **subtractive modeling**;
- **Additive modeling** is the process of building the model by assembling many simpler objects;
- **Subtractive modeling** is the process of removing pieces from a given object to create a new object, for example, creating a (cylindrical) hole in a sphere or a cube;



How to define a curve

What is a curve in $\mathcal{R}^n$?



A circle (or circumference) with center $(x_0, y_0) \in \mathcal{R}^n, n = 2$ and radius $r > 0$ is the curve having equation:

$$(x - x_0) + (y - y_0) = r^2$$

However, it can be represented as the image of the map $\sigma : \mathcal{R} \to \mathcal{R}^2$ as

$$\sigma(t) = (x_0 + r\ cos(t), y_0 + r\ sin(t))$$

How to define a curve

Definition: Given $k \in \mathcal{N} \cup \{\propto\}$ and $n \leq 2$, a parameterized curve of class $\mathcal{C}^k \in \mathcal{R}^n$ is a map $\sigma : \mathrm{I} \to \mathcal{R}n$ of class $\mathcal{C}^k$, where $\mathrm{I} \subset \mathcal{R}$ is an interval. The image $\sigma(I)$ is often called support of the curve; the variable $t \in I$ is the parameter of the curve. If $I = [a, b]$ and $\sigma(a) = \sigma(b)$, we shall say that the curve is closed.

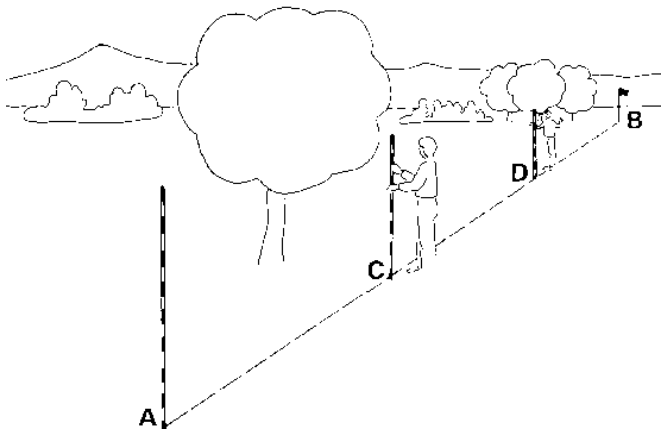


Figure: Geometric curve

Example: Given $v_0, v_1 \in \mathcal{R}^n$ such that $v_1 \neq \mathrm{O}$, the parameterized curve $\sigma : \mathcal{R} \to \mathcal{R}^n$ given by $\sigma(t) = v_0 + tv_1$ has as its image the straight line through v_0 in the direction v_1.

Curve Design

- To find a curve for a given $n+1$ data points, $P_0(x_0, y_0), \cdots, P_n(x_n, y_n)$ that require the curve to pass through all the points, we are faced with the problem of **interpolation**.
- If we require only that the curve be near these points, we are faced with the problem of **approximation**.
- To solve these problems, it is necessary to find the curve segments or smaller pieces of curves, in order to meet the design criteria.
- When modeling a curve $f(x)$ by using curve segments, we try to represent the curve as a sum of smaller segments $\Phi_i(x)$ (called basis functions):

$$f(x) = \Sigma_{i=0}^{N} a_i \Phi_i(x)$$

There are different types of basis function may be used to approximate the curve functions which are given as follows:

Curve Design

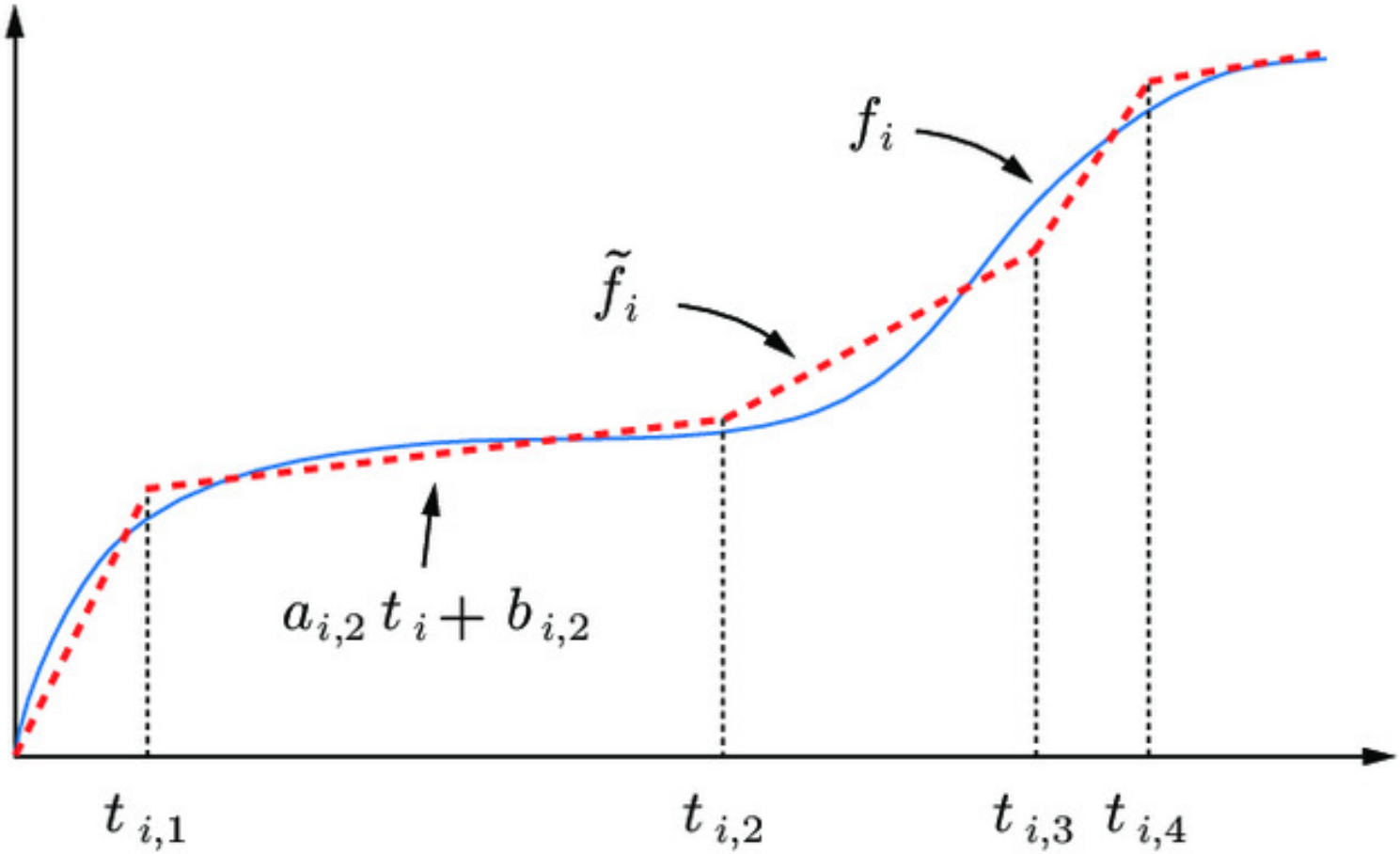


Figure: Continuous polynomial curve

Curve Design

1. A *polynomial of degree n* is a function that has the form:

$$Q(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

This polynomial is determined by its $n+1$ coefficients $[a_n, a_{n-1}, \cdots, a_0]$.

2. A *continuous piecewise polynomial* $Q(x)$ of degree n is a set of k polynomials $q_i(x)$, each of degree n and $k+1$ knots (nodes) $t_0, \cdots, t_k$ so that

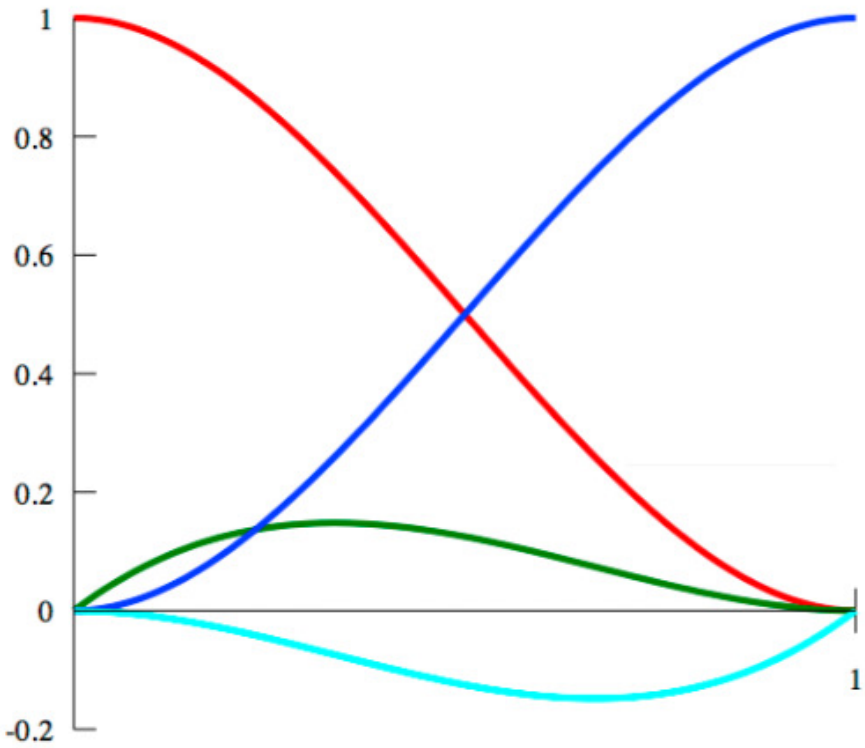
$$Q(x) = q_i(x) \text{ for } t_i \leq x \leq t_{i+1} \text{ and} i = 0, \cdots, k+1$$

This definition requires the polynomials to match or piece together at the knots, that is,

$$q_{i-1}(t_i) = q_i(t_i), i = 1, \cdots, k-1$$

3. Polynomials of high degree are not very useful for curve designing because of their oscillatory nature.

Curve Design



$$b_0(u) = 2u^3 - 3u^2 + 1$$
$$b_1(u) = -2u^3 + 3u^2$$
$$b_2(u) = u^3 - 2u^2 + u$$
$$b_3(u) = u^3 - u^2$$

Figure: Continuous piecewise polynomial curve

Cubics

The most useful piecewise polynomials are those for which the polynomials $q_i(x)$ are cubic (degree 3).

$$\mathbf{f}(u) = \mathbf{a}_0 + \mathbf{a}_1 u + \mathbf{a}_2 u^2 + \mathbf{a}_3 u^3$$

- Allow up to C^2 continuity at knots
- need 4 control points
  - may be 4 points on the curve, combination of points and derivatives, ...
- good smoothness and computational properties

Draw Cubics curve

The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

points = [(3.28,0.00),(4.00,0.50),(4.40,1.0),(4.60,1.52),(5.00,2.5),(5.00,3.34),(4.70,3.8)]
points = points + [(4.50,3.96),(4.20,4.0),(3.70,3.90),(3.00,3.5),(2.00,2.9)]
data = np.array(points)

tck,u = interpolate.splprep(data.transpose(), s=0)
unew = np.arange(0, 1.01, 0.01)
out = interpolate.splev(unew, tck)

plt.figure()
plt.plot(out[0], out[1], color='orange')
plt.plot(data[:,0], data[:,1], 'ob')
plt.show()
```

Curve Design

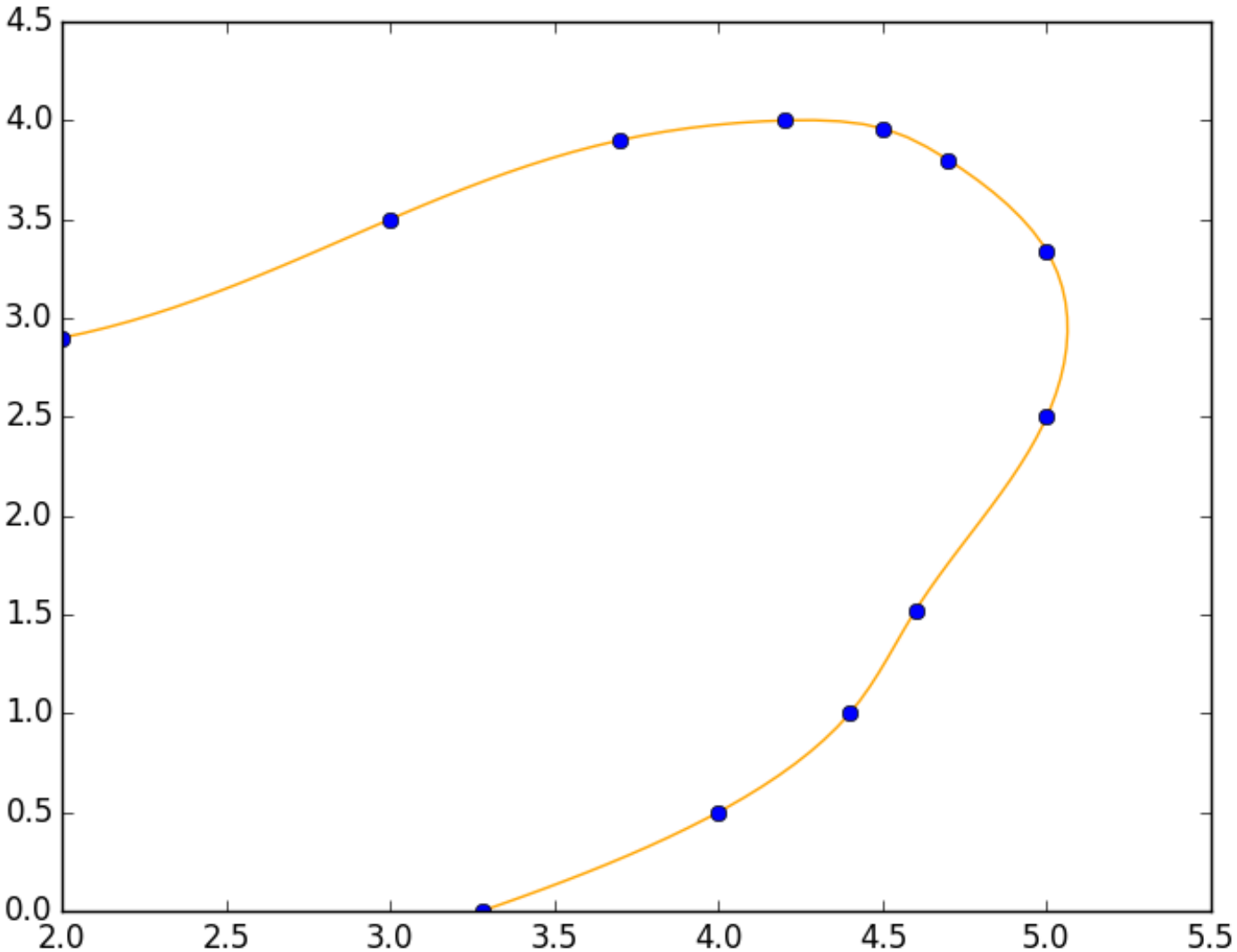


Figure: Cubics curve

Lagrange Polynomials of Degree n

Let $P_0(x_0, y_0), \cdots, P_n(x_n, y_n)$ represent $n+1$ data points and also, consider $t_0, t_1, t_2, \cdots, t_n$, be any numbers (knots or nodes). The following are common choices for basis functions given as follows:

1. **Lagrange Polynomials of Degree n:**

$$L_i(x) = \Pi_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \ \ i = 0, 1, \cdots, n$$

Note that $L_i(x_i) = 1$ and $L_i(x_j) = 0$ for all $j \neq i$.

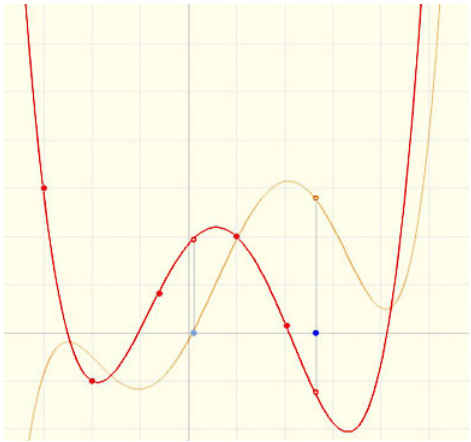


Figure: Lagrange Polynomials of Degree n

Lagrange Polynomials of Degree n

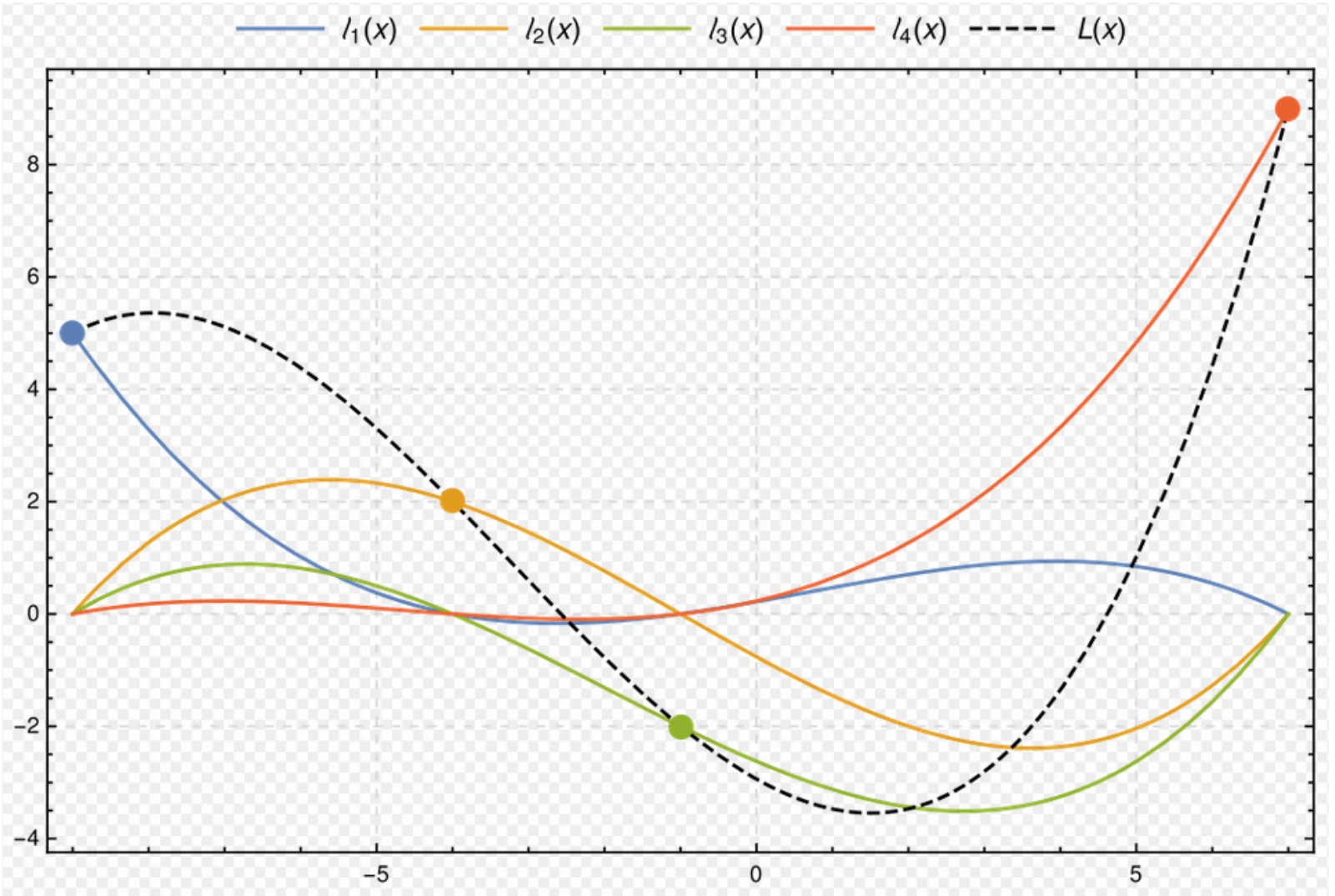


Figure: Lagrange Polynomials of Degree n

Lagrange Polynomials Basis Functions

The following Python code:

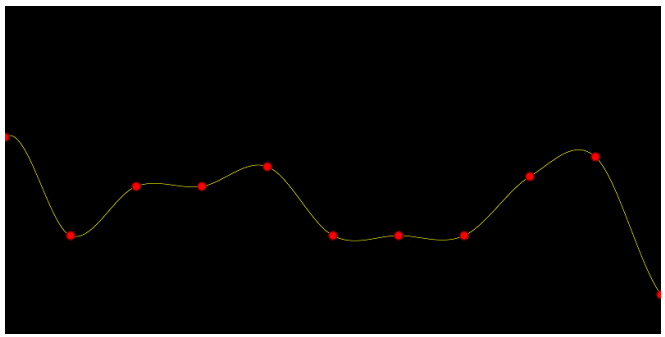
```
import numpy as np
import numpy.polynomial.polynomial as poly
import matplotlib.pyplot as plt

plt.style.use('seaborn-poster')
x = [0, 1, 2]
y = [1, 3, 2]
P1_coeff = [1,-1.5,.5]
P2_coeff = [0, 2,-1]
P3_coeff = [0,-.5,.5]

# get the polynomial function
P1 = poly.Polynomial(P1_coeff)
P2 = poly.Polynomial(P2_coeff)
P3 = poly.Polynomial(P3_coeff)

x_new = np.arange(-1.0, 3.1, 0.1)

fig = plt.figure(figsize = (10,8))
plt.plot(x_new, P1(x_new), 'b', label = 'P1')
plt.plot(x_new, P2(x_new), 'r', label = 'P2')
plt.plot(x_new, P3(x_new), 'g', label = 'P3')

plt.plot(x, np.ones(len(x)), 'ko', x, np.zeros(len(x)), 'ko')
plt.title('Lagrange Basis Polynomials')
plt.xlabel('x'),plt.ylabel('y')
plt.grid(),plt.legend(),plt.show()
```

Hermite Polynomials Basis Functions

A cubic Hermite spline or cubic Hermite interpolator is a spline where each piece is a third-degree polynomial specified in Hermite form, that is, by its values and first derivatives at the end points of the corresponding domain interval.

$$H_i(x) = \begin{cases} -\frac{2(x-t_{i-1})^3}{(t_i-t_{i-1})^3} + \frac{3(x-t_{i-1})^2}{(t_i-t_{i-1})^2} & t_{i-1} \leq x \leq t_i \\ -\frac{2(t_{i+1}-x)^3}{(t_{i+1}-t_i)^3} + \frac{3(t_{i+1}-x)^2}{(t_{i+1}-t_i)^2} & t_i \leq x \leq t_{i+1} \end{cases}$$

$$\bar{H}_i(x) = \begin{cases} \frac{(x-t_{i-1})^2(x-t_i)}{(t_i-t_{i-1})^2} & t_{i-1} \leq x \leq t_i \\ \frac{(x-t_i)(t_{i+1}-x)^2}{(t_{i+1}-t_i)^2} & t_i \leq x \leq t_{i+1} \end{cases}$$



Hermite Polynomials Basis Functions

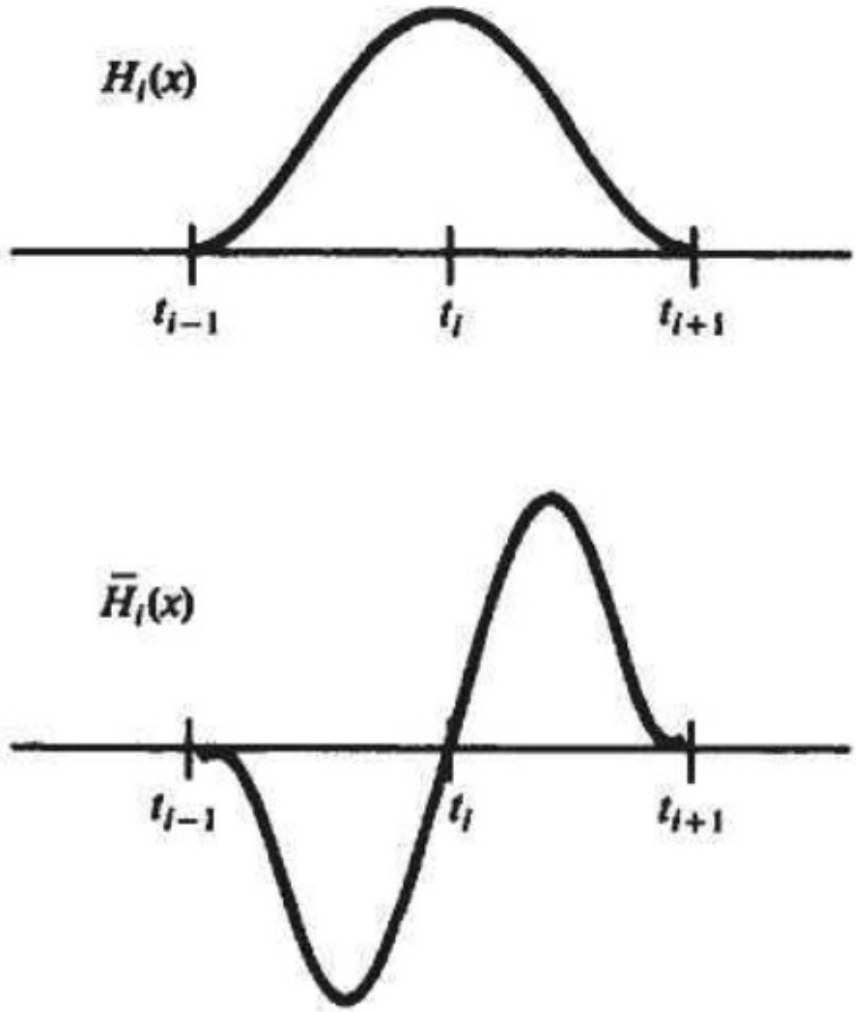


Figure: Hermite Polynomials Basis Functions

Hermite Polynomials Basis Functions

The following Python code:

```
from scipy import special
import matplotlib.pyplot as plt
import numpy as np

p_monic = special.hermite(3, monic=True)
x = np.linspace(-3, 3, 400)
y = p_monic(x)
plt.plot(x, y)
plt.title("Monic Hermite polynomial of degree 3")
plt.xlabel("x")
plt.ylabel("H_3(x)")
plt.show()
```

B-Splines

A **B-spline** curve is a piecewise polynomial curve specified by an arbitrary collection of control points P_j and a non-decreasing sequence of knots t_k, where each individual polynomial segment is defined by the different algorithm (e.g. De Boor). For the knot set $t_0, t_1, t_2, \cdots, t_n$ the n^{th}-degree B-splines $B_{i,n}$ in are defined recursively:

$$B_{i,0}(x) = \begin{cases} 1 & t_i \leq x \leq t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,n}(x) = \frac{(x - t_i)}{(t_{i+n} - t_i)} B_{i,n-1}(x) + \frac{(t_{i+n+1} - x)}{(t_{i+n+1} - t_{i+1})} B_{i+i,n-1}(x)$$

for $t_i \leq x \leq t_{i+n+1}$.

B-Splines

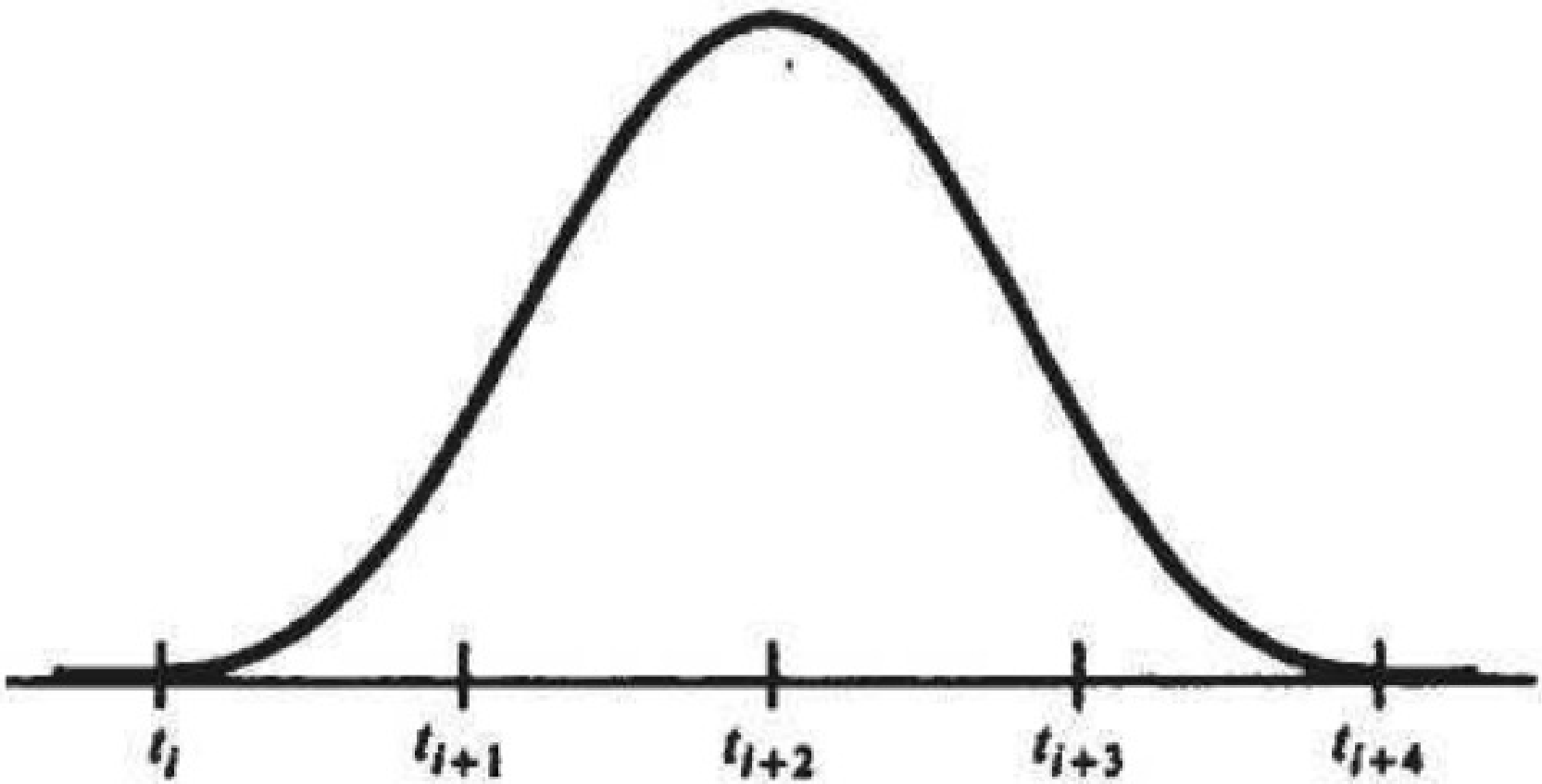


Figure: B-Splines Polynomials Basis Functions

B-Splines

Note that $B_{i,n}(x)$ is nonzero only in the interval $[t_i, t_{i+n+1}$. In particular, the cubic **B-spline** $B_{i,3}$ is nonzero over the interval $[t_i, t_{i+4}]$ (which spans the knots $[t_i, t_{i+1}, t_{i+2}, t_{i+3}, t_{i+4}]$). For non-repeated knots, the **B-spline** is zero at the end knots t_i and t_{i+n+1}, that is,

$$B_{i,n}(t_i) = 0; \;\; B_{i,n}(t_{i+n+1}) = 0, \;\; (n \geq 1)$$

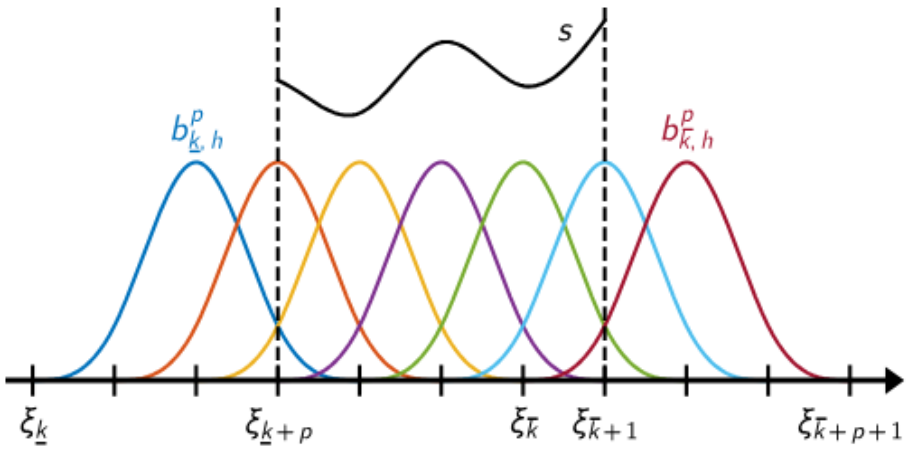


Figure: B-Splines Polynomials Basis Functions

B-Splines

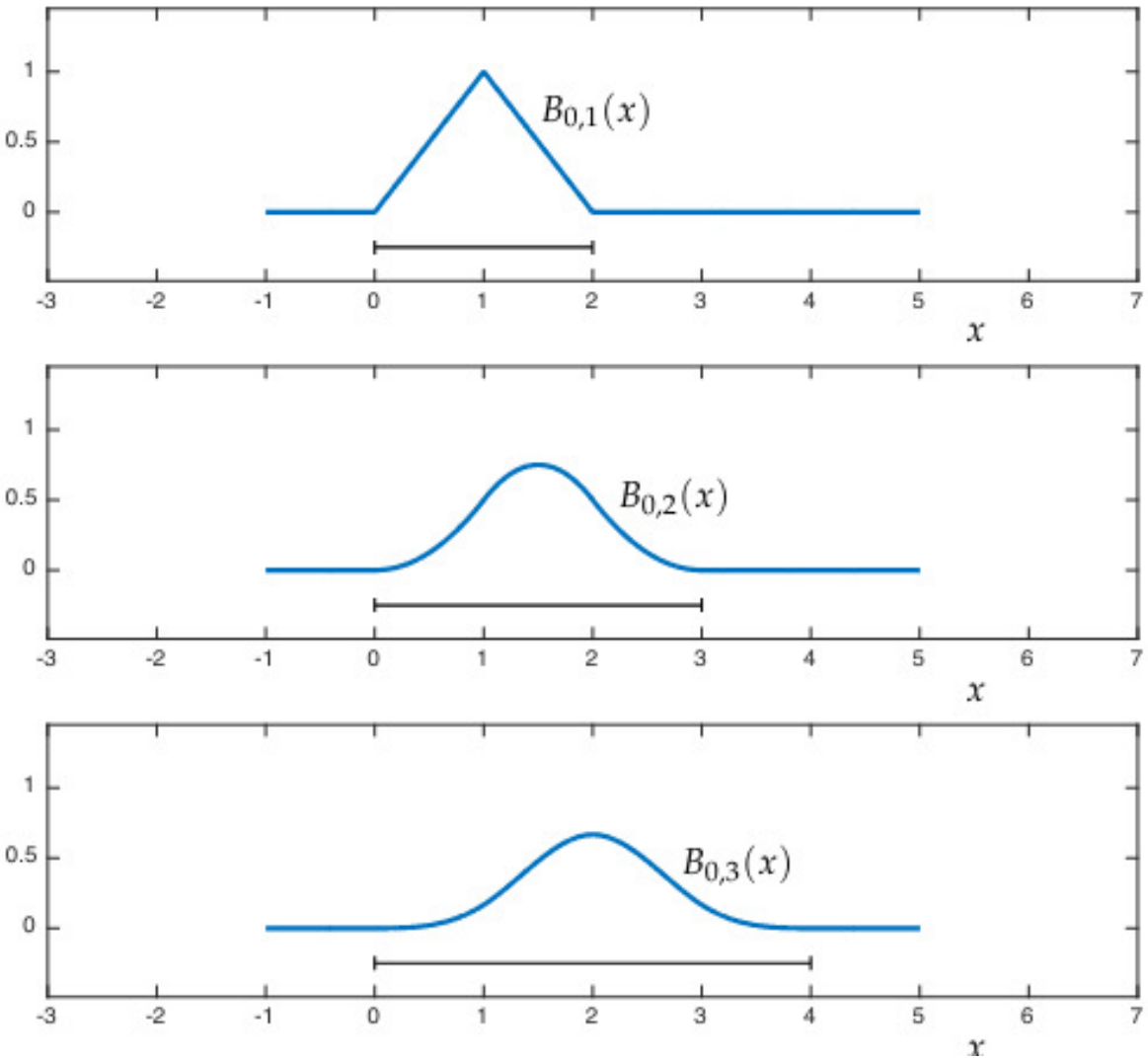


Figure: B-Splines Polynomials Basis Functions

B-Splines

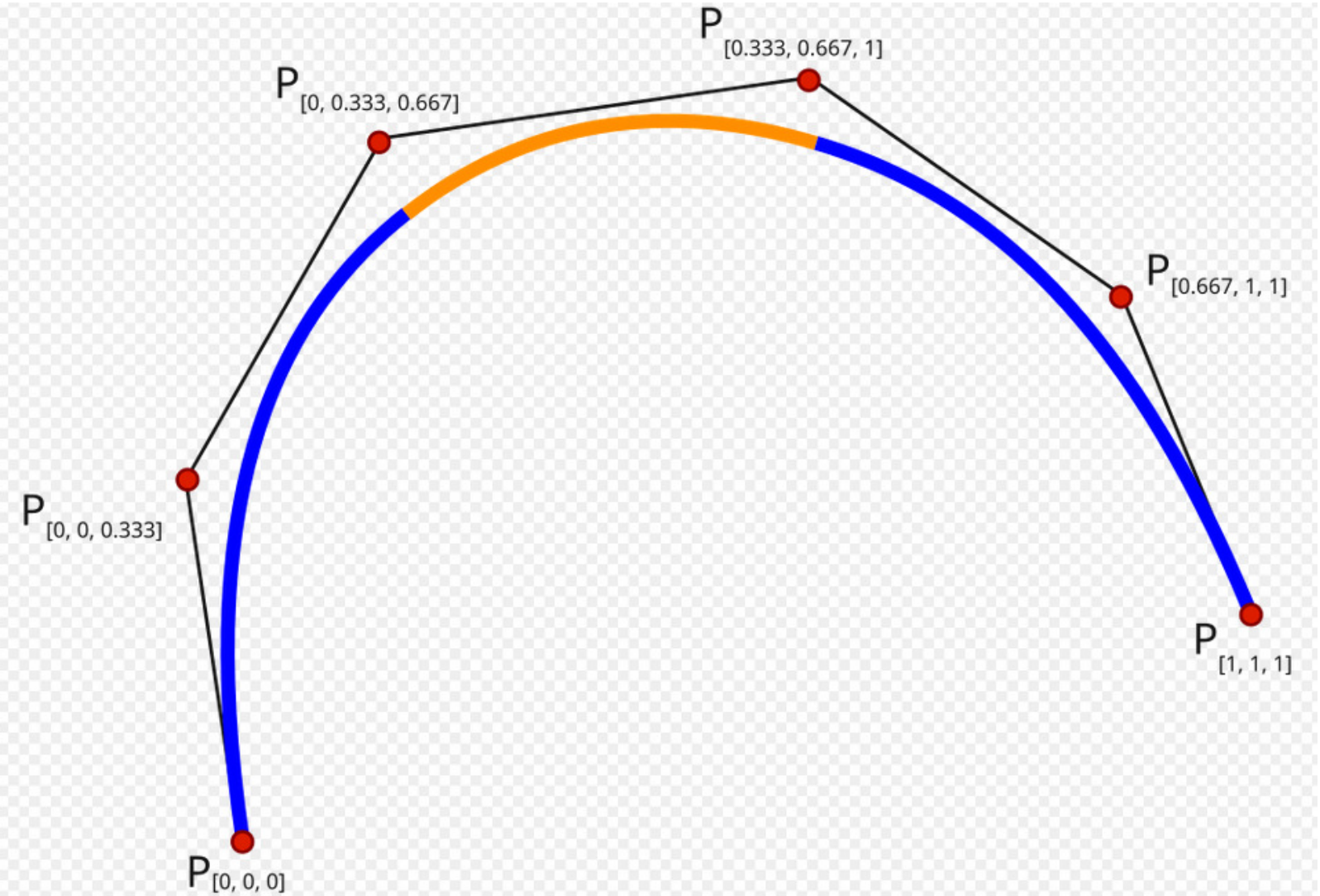


Figure: B-Splines Polynomials Basis Functions

B-spline Polynomials Basis Functions

The following Python code:

```
import numpy as np
from scipy.interpolate import BSpline
import matplotlib.pyplot as plt

# Define knot vector, coefficients, and degree
t = [0, 1, 2, 3, 4, 5]
c = [-1, 2, 0, -1]
k = 2

# Create a BSpline object
spl = BSpline(t, c, k)

# Evaluate the spline at multiple points
x = np.linspace(1.5, 4.5, 50)
y = spl(x)

plt.plot(x, y)
plt.title('B-Spline Curve')
plt.xlabel('x')
plt.ylabel('S(x)')
plt.grid(True)
plt.show()
```

Bernstein Polynomials

A **Bernstein** polynomial is a polynomial expressed as a linear combination of Bernstein basis polynomials. The **Bernstein** polynomials of degree n over the interval $[0, 1]$ are defined as

$$BE_{k,n}(x) = \frac{n!}{k!(n-k)!} x^k (1-x)^{n-k}$$

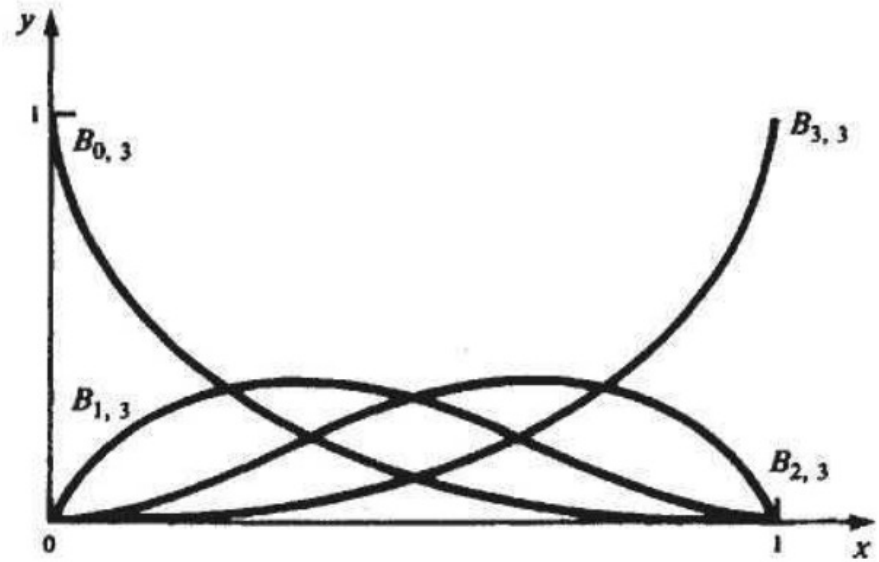
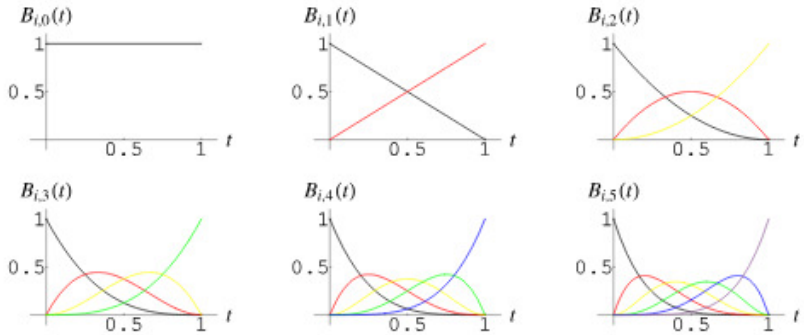


Figure: Bernstein Polynomials Basis Functions

Bernstein Polynomials Basis Functions



The polynomials defined by

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i},$$

where $\binom{n}{k}$ is a binomial coefficient. The Bernstein polynomials of degree n form a basis for the power polynomials of degree n.

first few polynomials are

$$\begin{aligned}
B_{0,0}(t) &= 1 \\
B_{0,1}(t) &= 1 - t \\
B_{1,1}(t) &= t \\
B_{0,2}(t) &= (1-t)^2 \\
B_{1,2}(t) &= 2(1-t)t \\
B_{2,2}(t) &= t^2 \\
B_{0,3}(t) &= (1-t)^3 \\
B_{1,3}(t) &= 3(1-t)^2 t \\
B_{2,3}(t) &= 3(1-t)t^2 \\
B_{3,3}(t) &= t^3.
\end{aligned}$$

Bernstein Polynomials Basis Functions

$$B_{i,n}(t) = B_{n-i,n}(1-t),$$

positivity

$$B_{i,n}(t) \geq 0$$

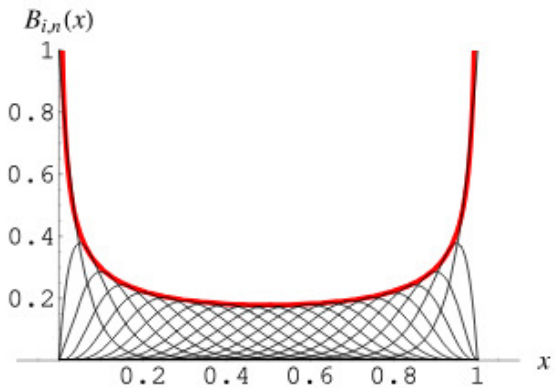
for $0 \leq t \leq 1$, normalization

$$\sum_{i=0}^{n} B_{i,n}(t) = 1,$$

and $B_{i,n}$ with $i \neq 0,\ n$ has a single unique local maximum of

$$i^i\, n^{-n}\, (n-i)^{n-i} \binom{n}{i}$$

occurring at $t = i/n$.



The envelope $f_n(x)$ of the Bernstein polynomials $B_{i,n}(x)$ for $i = 0, 1, ..., n$ (Mabry 2003) is given by

$$f_n(x) = \frac{1}{\sqrt{2\pi n x(1-x)}},$$

Bernstein Polynomials Basis Functions

The following Python code:

```
from scipy.interpolate import BPoly

def main():
        x = [0, 1]
        c = [[1], [2], [3]]
        bp = BPoly(c, x)

if __name__ == "__main__":
        main()
```

Problem of Interpolation

- An interpolation problem refers to the task of finding an estimation function that accurately predicts the value of a dependent variable based on a given set of independent variables;
- The interpolation function is designed to pass through all the data points in the dataset, providing a smooth and continuous estimation;
- Given data points $P_0(x_0, y_0), \cdots, P_n(x_n, y_n)$, we wish to find a curve which passes through these points;

How to solve an interpolation problem:

- Organize the data into a chart;
- Create a graph;
- Select two points;
- Enter the values into the interpolation equation;
- Solve for the missing variable

Lagrange Polynomial Interpolation

Lagrange Interpolation Formula finds a polynomial called Lagrange Polynomial that takes on certain values at an arbitrary point. Here

$$L(x) = \sum_{i=0}^{n} y_i L_i(x)$$

where $L_i(x)$ are the Lagrange polynomials and $L(x)$ is the n^{th}-degree polynomial interpolating the data points.

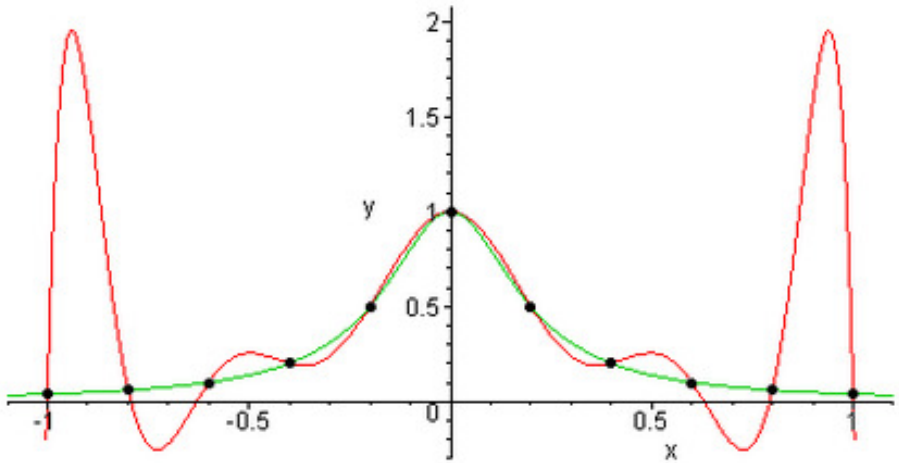


Figure: Lagrange Polynomial Interpolation

Lagrange Polynomial Interpolation

The following Python code:

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

plt.style.use('seaborn-poster')
x = [0, 1, 2], y = [1, 3, 2]
P1_coeff = [1,-1.5,.5]
P2_coeff = [0, 2,-1]
P3_coeff = [0,-.5,.5]

# get the polynomial function
P1 = poly.Polynomial(P1_coeff)
P2 = poly.Polynomial(P2_coeff)
P3 = poly.Polynomial(P3_coeff)

x_new = np.arange(-1.0, 3.1, 0.1)

fig = plt.figure(figsize = (10,8))
plt.plot(x_new, P1(x_new), 'b', label = 'P1')
plt.plot(x_new, P2(x_new), 'r', label = 'P2')
plt.plot(x_new, P3(x_new), 'g', label = 'P3')

plt.plot(x, np.ones(len(x)), 'ko', x, np.zeros(len(x)), 'ko')
plt.title('Lagrange Basis Polynomials')
plt.xlabel('x'), plt.ylabel('y'), plt.grid(), plt.legend(), plt.show()
```

Lagrange Polynomial Interpolation Solution

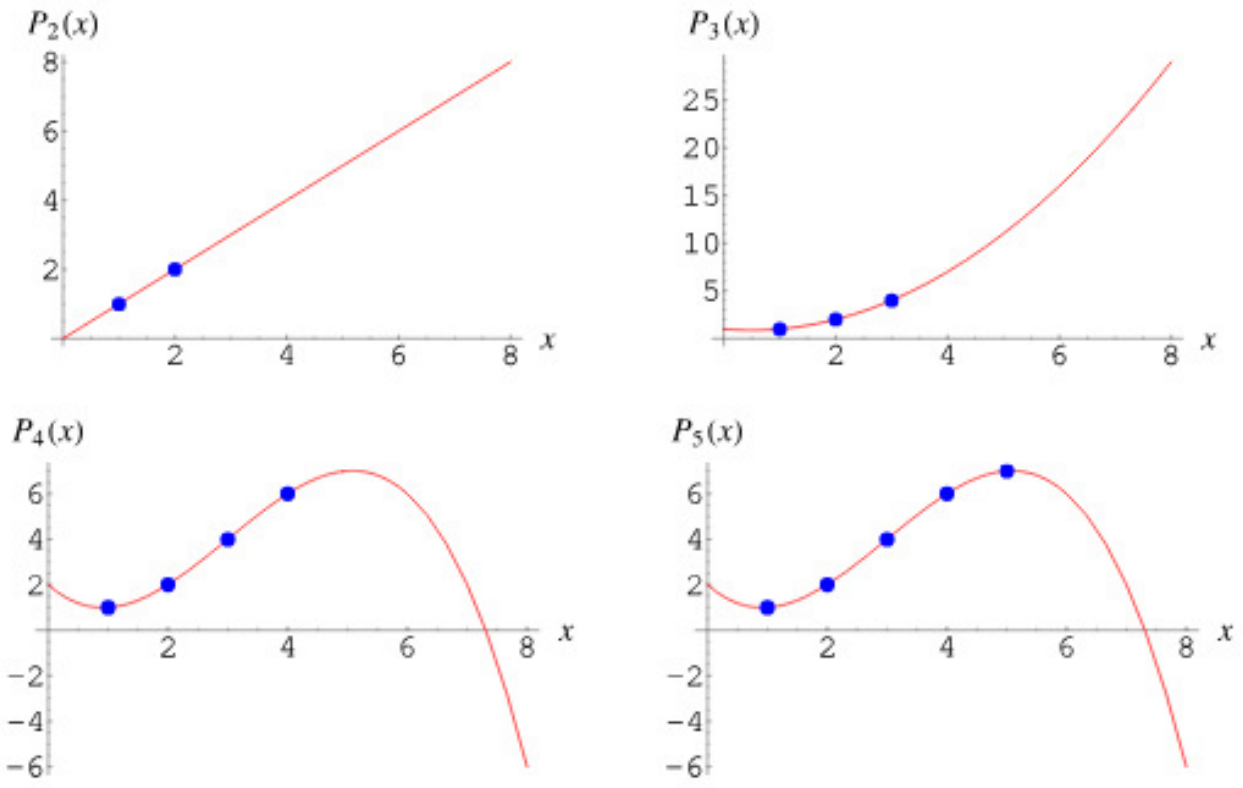


Figure: Lagrange Polynomial Interpolation

Hermitian Cubic Interpolation

- Hermite spline interpolator is a spline where each piece is a third-degree polynomial specified in Hermite form with its values and first derivatives at the end points of the corresponding domain interval.
- To find a piecewise polynomial $H(x)$ of degree 3 which passes through the data points and continuously differentiable at these points.
- The values of the derivatives $\acute{y}$ at the given data points $(x_0\acute{y}_0), \cdots, (x_n\acute{y}_n)$:

$$H(x) = \sum_{i=0}^{n} [y_i H_i(x) + \acute{y}_i \bar{H}_i(x)]$$

where $H_i(x)$ and $\bar{H}_i(x)$ are the Hermitian cubic basis functions and $t_0 = x_0, t_1 = x_1, t_2 = x_2, \cdots, t_n = x_n$ are the choices for the knot set.

Hermitian Cubic Interpolation

The following Python code:

```
from scipy.interpolate import pchip
import numpy as np
from scipy.interpolate import PPoly
x = np.arange(10)
y = [1., 1., 3., 2., 1., 1., 1.5, 2., 8., 1.]
s = pchip(x, y)
pp = PPoly.from_bernstein_basis(s)
pp.roots()
```

Hermitian Cubic Interpolation

The following Python code:

```
from matplotlib.pyplot import subplots
from chspy import CubicHermiteSpline

spline = CubicHermiteSpline(n=3)

spline.add((0,[1,3,0],[0,1,0]))
spline.add((1,[3,2,0],[0,4,0] ))
spline.add((4,[0,1,3],[0,4,0] ))

fig,axes = subplots(figsize=(7,2))
spline.plot(axes)
axes.set_xlabel("time")
axes.set_ylabel("state")
fig.legend(loc="right",labelspacing=1)
fig.subplots_adjust(right=0.7)
```

Hermitian Cubic Interpolation

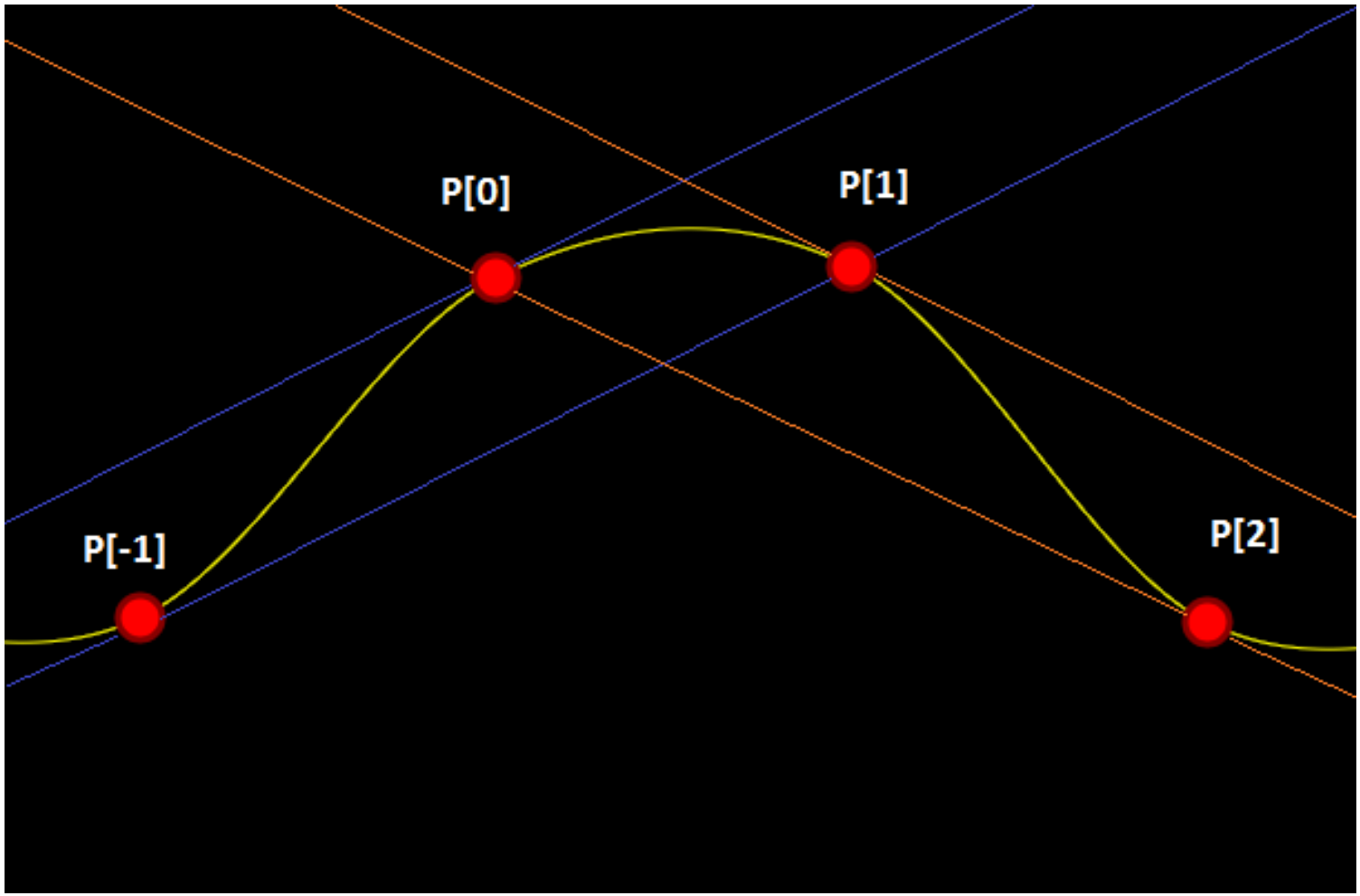


Figure: Hermitian Interpolation

Spline Interpolation

- A spline which interpolating piecewise polynomial be joined as smoothly as possible at the data points;
- A spline of degree m has continuous derivatives up to order $m-1$ at the data points.
- Any m^{th}-degree spline that passes through $n+1$ data points can be represented in terms of the **B-spline basis** functions $B_{i,n}$ in as

$$S_m(x) = \sum_{i=0}^{m+n-1} a_i B_{i,m}(x)$$

- To define the **B-spline** functions $B_{i,m}(x)$ so as to solve the interpolation problem, the knots $t_0, t_1, \cdots, t_{m+n+1}$ must be chosen to satisfy the Shoenberg-Whitney condition:

$$t_i < x_i < t_{m+n+1}, \ \ i = 0, \cdots, n$$

Spline Interpolation

Shoenberg-Whitney condition:

1. Choose $t_0 = \cdots = t_m < x_0, t_{n+1} = \cdots = t_{m+n+1} > x_n$
2. Choose the remaining knots $t_{m+n+1} = \frac{x_{i+1}+\cdots+x_{i+m}}{m}, \ i = 0, \cdots, n-m-1;$
3. For cubic splines $(m = 3)$, Choose $t_{i+4} = x_{i+2}, i = 0, \cdots, n-4$
4. The splines $S_2(x)$ and $S_3(x)$ are called **quadratic** and **cubic** splines, respectively:

$$S_2(x) = \sum_{i=0}^{n+1} a_i B_{i,2}(x), \ S_3(x) = \sum_{i=0}^{n+2} a_i B_{i,3}(x)$$

5. To the cubic spline, there are $n+3$ equations requiring to evaluate $n+3$ coefficients a_i and the interpolation criterion $S_3(x_j) = y_j, j = 0, \cdots, n$ provides $n+1$ equations:

$$y_j = S_3(x_j) = \sum_{i=0}^{n+2} a_i B_{i,3}(x_j)$$

Spline Interpolation

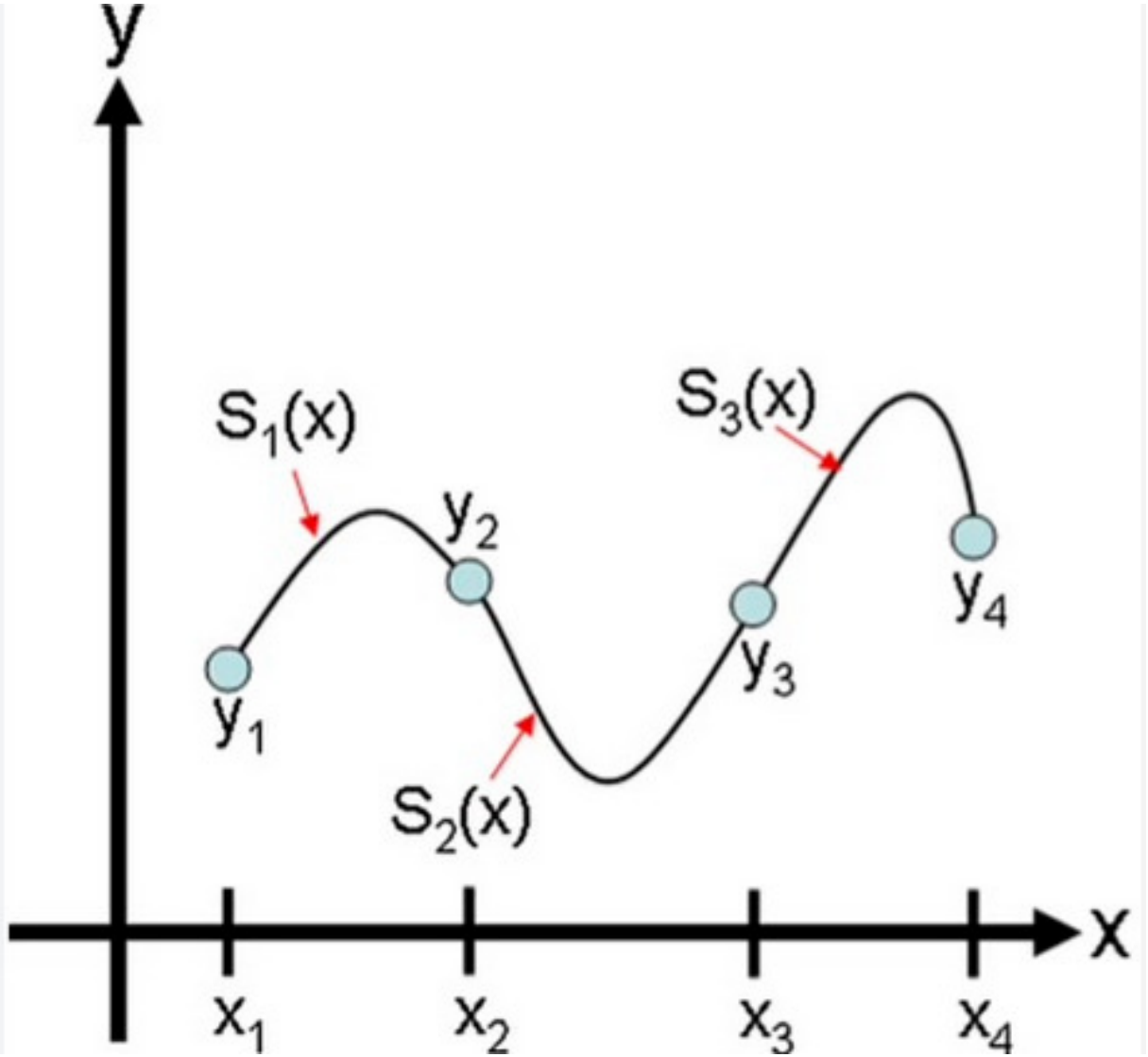


Figure: Spline Interpolation

Spline Interpolation (Continue)

The Two equations are usually specified as boundary conditions at the endpoints x_0 and x_n which are:

1. **Natural spline condition:** $S_3''(x_0) = 0,\ S_3''(x_n) = 0$
2. **Clamped spline condition:** $S_3'(x_0) = y_0',\ S_3'(x_n) = y_n'$ where y_o' and y_n' are fixed derivative values;
3. **Cyclic spline condition:** $S_3'(x_0) = S_3'(x_n),\ S_3''(x_0) = S_3''(x_n)$, This is useful for producing closed curves;
4. **Anticyclic spline condition:** $S_3'(x_0) = -S_3'(x_n),\ S_3''(x_0) = -S_3''(x_n)$, This is useful in producing splines with parallel endings whose tangent vectors are equal in magnitude but opposite in direction;

The boundary condition 1 is known as natural boundary condition.

Spline Interpolation

The following Python code:

```
from scipy.interpolate import CubicSpline
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('seaborn')
x = [0,1,2]
y = [1,3,2]

f = CubicSpline(x, y, bc_type='natural')
x_new = np.linspace(0,2,100)
y_new = f(x_new)
plt.figure(figsize=(10,8))
plt.plot(x_new,y_new,'b')
plt.plot(x,y,'ro')
plt.title('Cubic Spline Interpolation')
plt.xlabel('x'), plt.ylabel('y'),plt.show()
```

Spline Interpolation

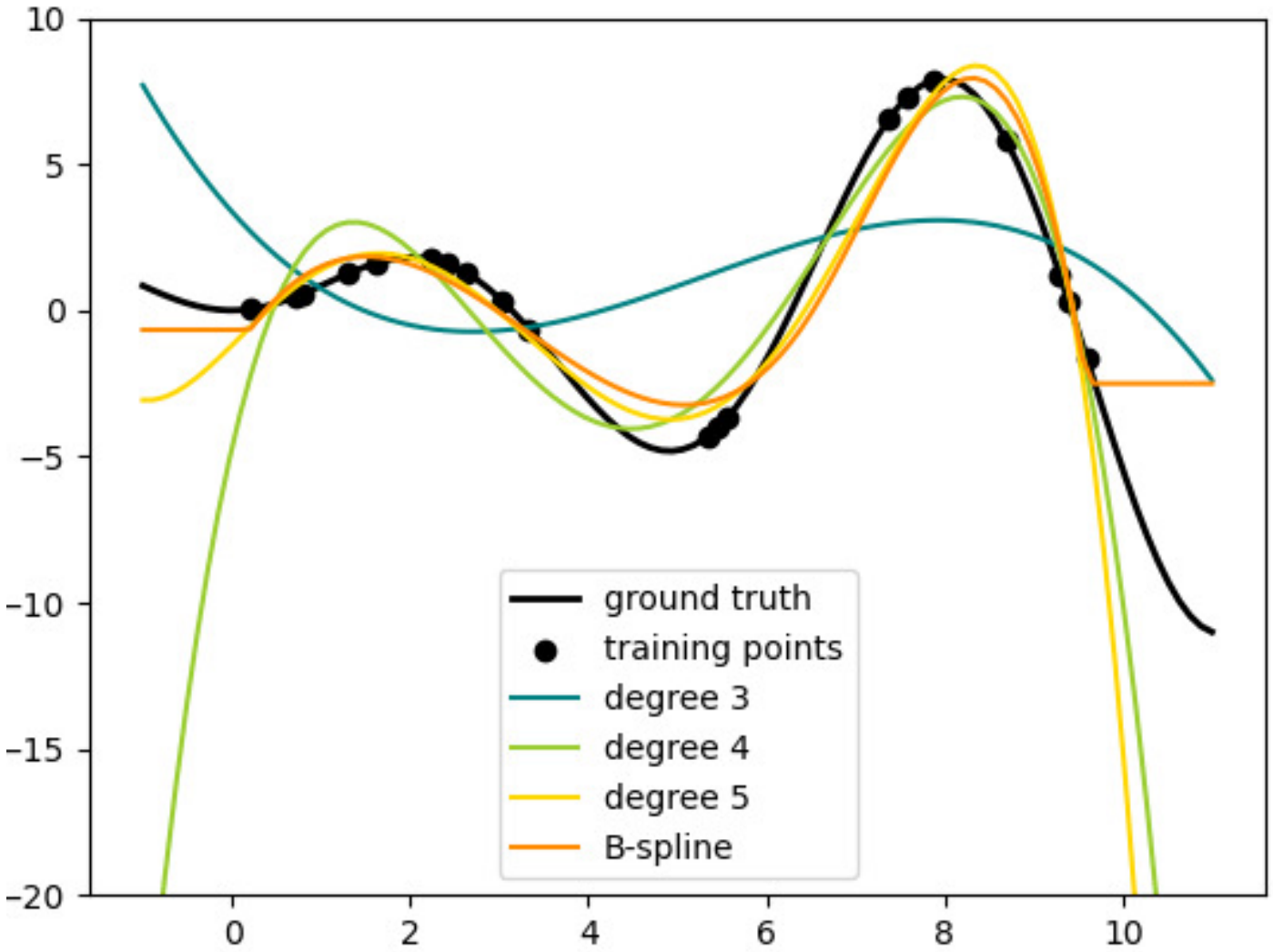


Figure: Spline Interpolation

Spline Interpolation

The following Python code:

```
import numpy as np
import scipy as sp
from scipy.interpolate import interp1d

x1 = [1., 0.88, 0.67, 0.50, 0.35, 0.27, 0.18, 0.11, 0.08, 0.04, 0.04, 0.02]
y1 = [0., 13.99, 27.99, 41.98, 55.98, 69.97, 83.97, 97.97, 111.96, 125.96, 139.95, 153.95]

points = zip(x1, y1)
points = sorted(points, key=lambda point: point[0])
x1, y1 = zip(*points)

new_length = 25
new_x = np.linspace(min(x1), max(x1), new_length)
new_y = sp.interpolate.interp1d(x1, y1, kind='cubic')(new_x)
```

Spline Interpolation

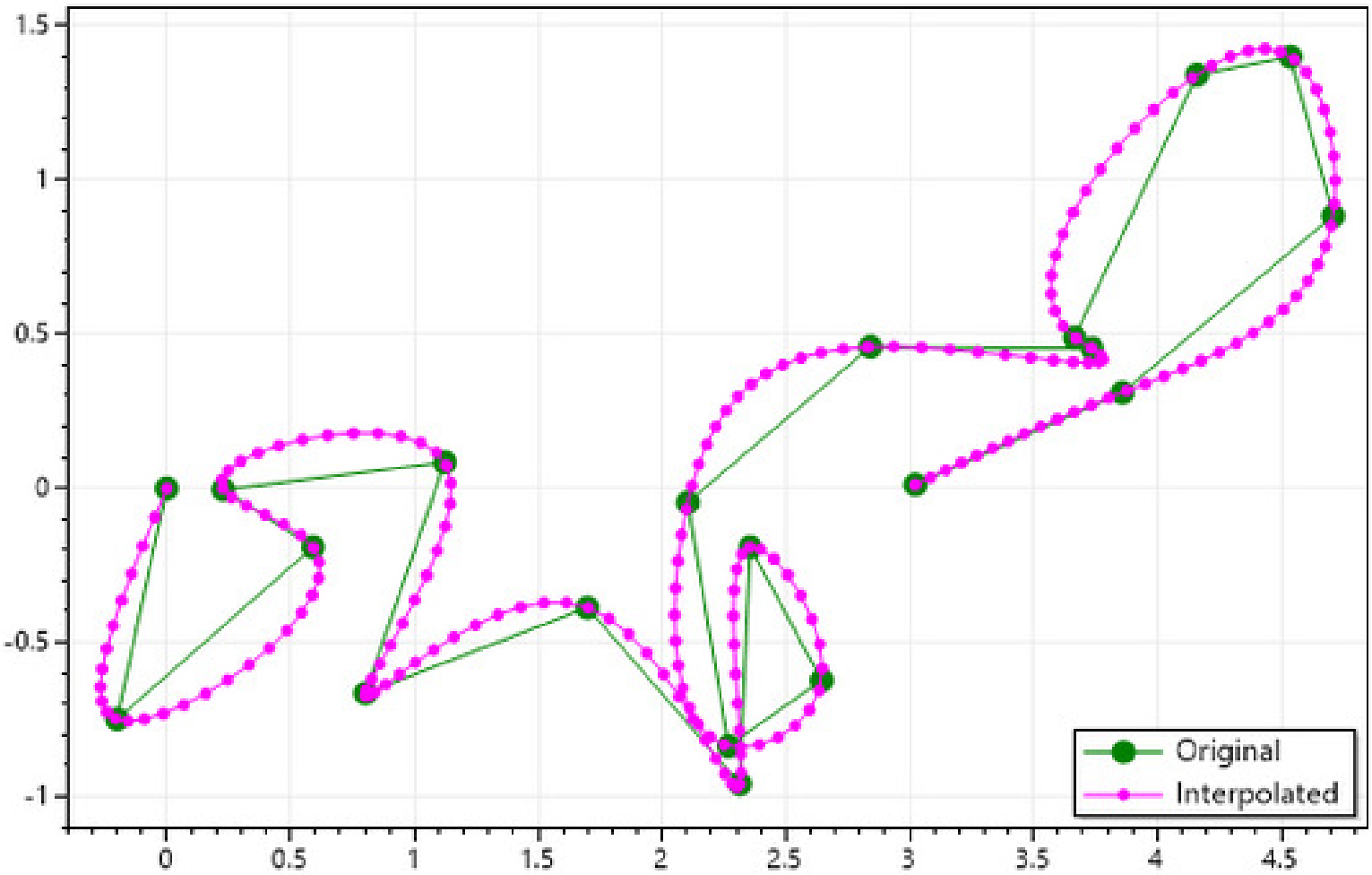


Figure: Spline Interpolation

The Problem of Approximation

1. The problem is to provide a smooth representation of a 3D curve which approximates given data so as to yield a given shape;
2. Usually the data is given interactively in the form of a guiding polyline determined by control points $P_0(x_0, y_0, z_0)$, $P_1(x_1, y_1, z_1)$, $\cdots$, $P_n(x_n, y_n, z_n)$;
3. Find a curve which approximates the shape of this guiding polyline;

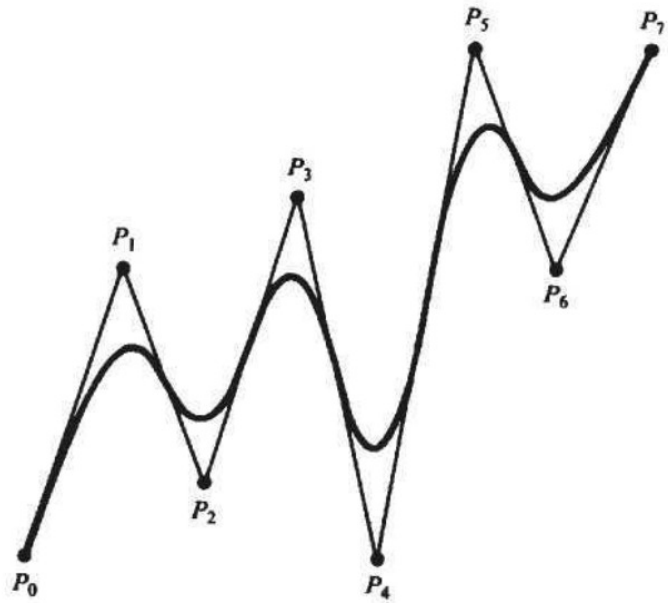


Figure: The Problem of Approximation

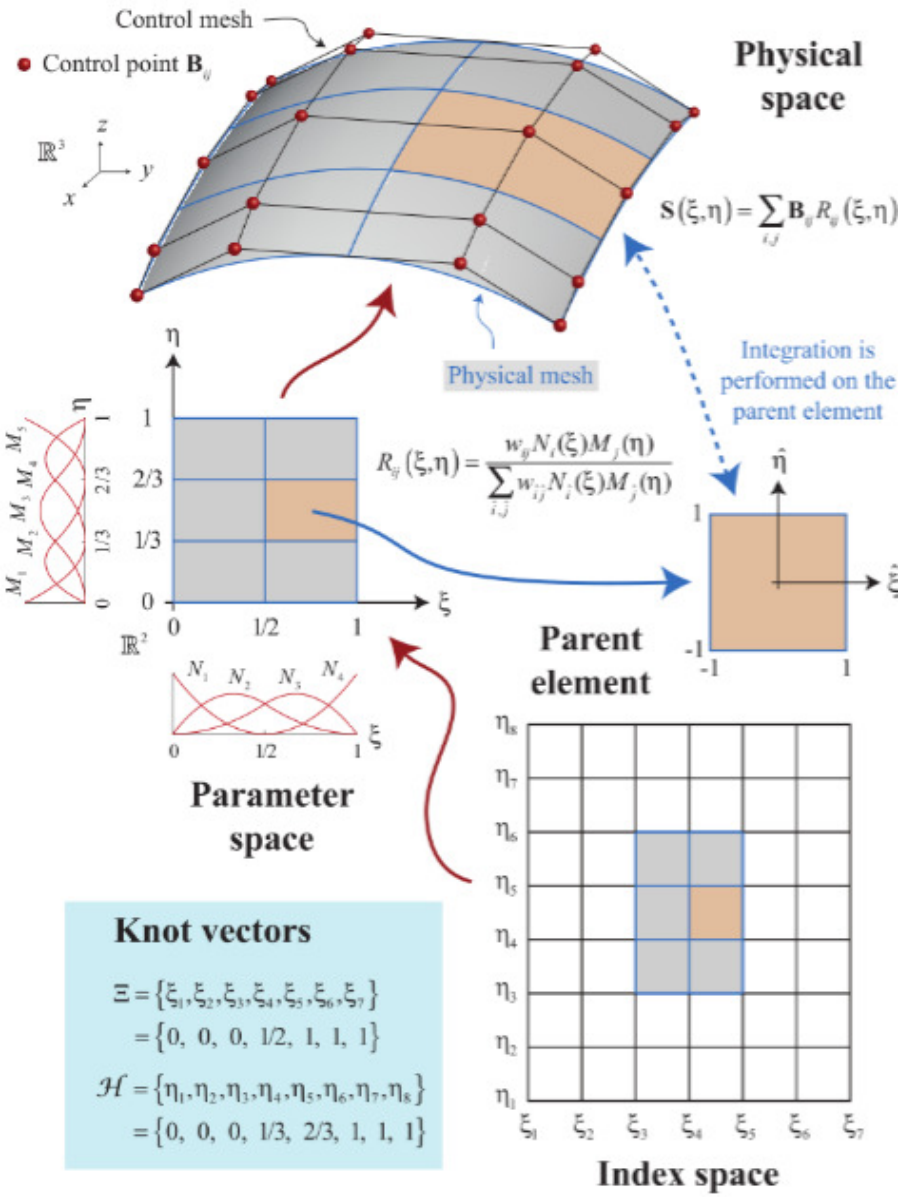
Bezier-Bernstein Approximation

Bezier-Bernstein Approximation: A fundamental method for representing polynomial curves and surfaces in computer-aided geometric design (CAGD), using Bernstein polynomials as a basis. The Bernstein polynomials basis is defined as:

$$P(t) = \begin{cases} x(t) = \sum_{i=0}^{n} x_i BE_{i,n}(t) \\ y(t) = \sum_{i=0}^{n} y_i BE_{i,n}(t) \quad 0 \leq t \leq 1 \\ z(t) = \sum_{i=0}^{n} z_i BE_{i,n}(t) \end{cases}$$

where $P(t)$ is called the **Bezier** curve.

Approximation



Properties of the Bezier-Bernstein Approximation

There are four basic properties:

1. The Bezier curve has the same endpoints as the guiding polyline, that is: $P_0 = P(0) = [x(0), y(0), z(0)]$, $P_n = P(1) = [x(1), y(1), z(1)]$;
2. The direction of the tangent vector at the endpoints P_0 , P_n is the same as that of the vector determined by the first and last segments $\overline{P_0P_1}$, $\overline{P_1P_2}$, $\overline{P_{n-1}P_n}$ of the guiding polyline;
3. The Bezier curve lies entirely within the convex hull of the guiding polyline $(P_0, P_1, \cdots, P_n)$;
4. The Bezier curves can be pieced together so as to ensure continuous differentiability at their juncture by letting the edges of the two different guiding polylines that are adjacent to the common endpoint be collinear;
5. Bezier curves are suited to interactive design;

Bezier-Bernstein Approximation

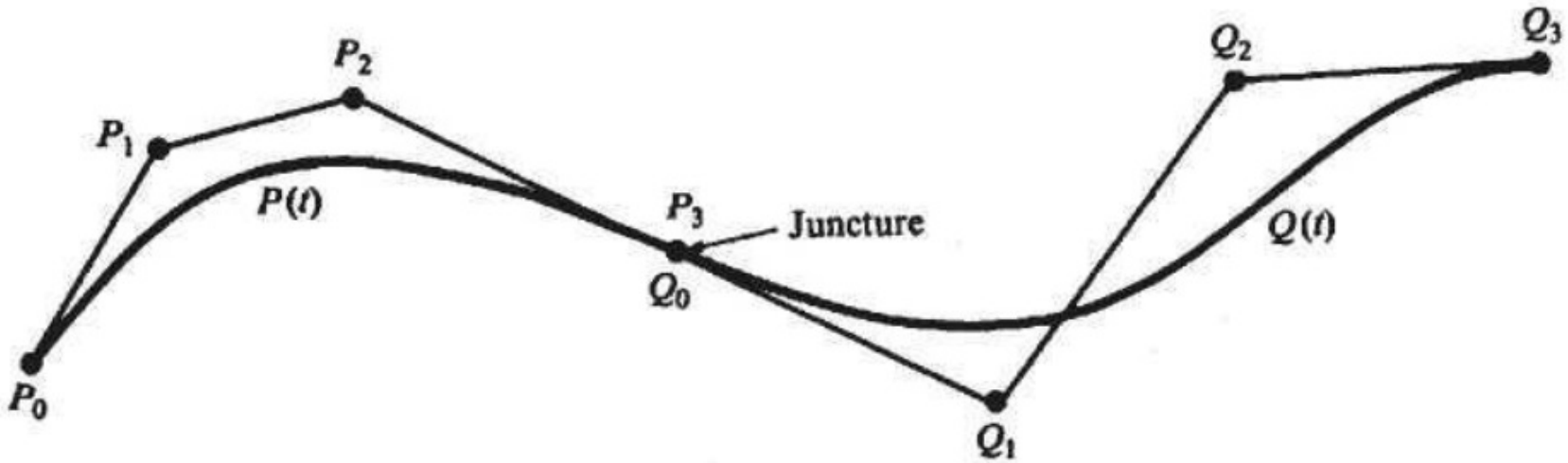


Figure: Bezier-Bernstein Approximation

Bezier-Bernstein Approximation

The following Python code:

```
import bezier
import numpy as np
import seaborn
seaborn.set()

nodes1 = np.asfortranarray([
     [0.0, 0.5, 1.0],
     [0.0, 1.0, 0.0], ])
curve1 = bezier.Curve(nodes1, degree=2)

nodes2 = np.asfortranarray([
     [0.0, 0.25,  0.5, 0.75, 1.0],
     [0.0, 2.0 , -2.0, 2.0 , 0.0], ])
curve2 = bezier.Curve.from_nodes(nodes2)
intersections = curve1.intersect(curve2)
s_vals = np.asfortranarray(intersections[0, :])
points = curve1.evaluate_multi(s_vals)
ax = curve1.plot(num_pts=256)
_ = curve2.plot(num_pts=256, ax=ax)
lines = ax.plot(points[0, :], points[1, :],marker="o",linestyle="None",color="black")
_ = ax.axis("scaled")
_ = ax.set_xlim(-0.125, 1.125)
_ = ax.set_ylim(-0.0625, 0.625)
```

Bezier-Bernstein Approximation

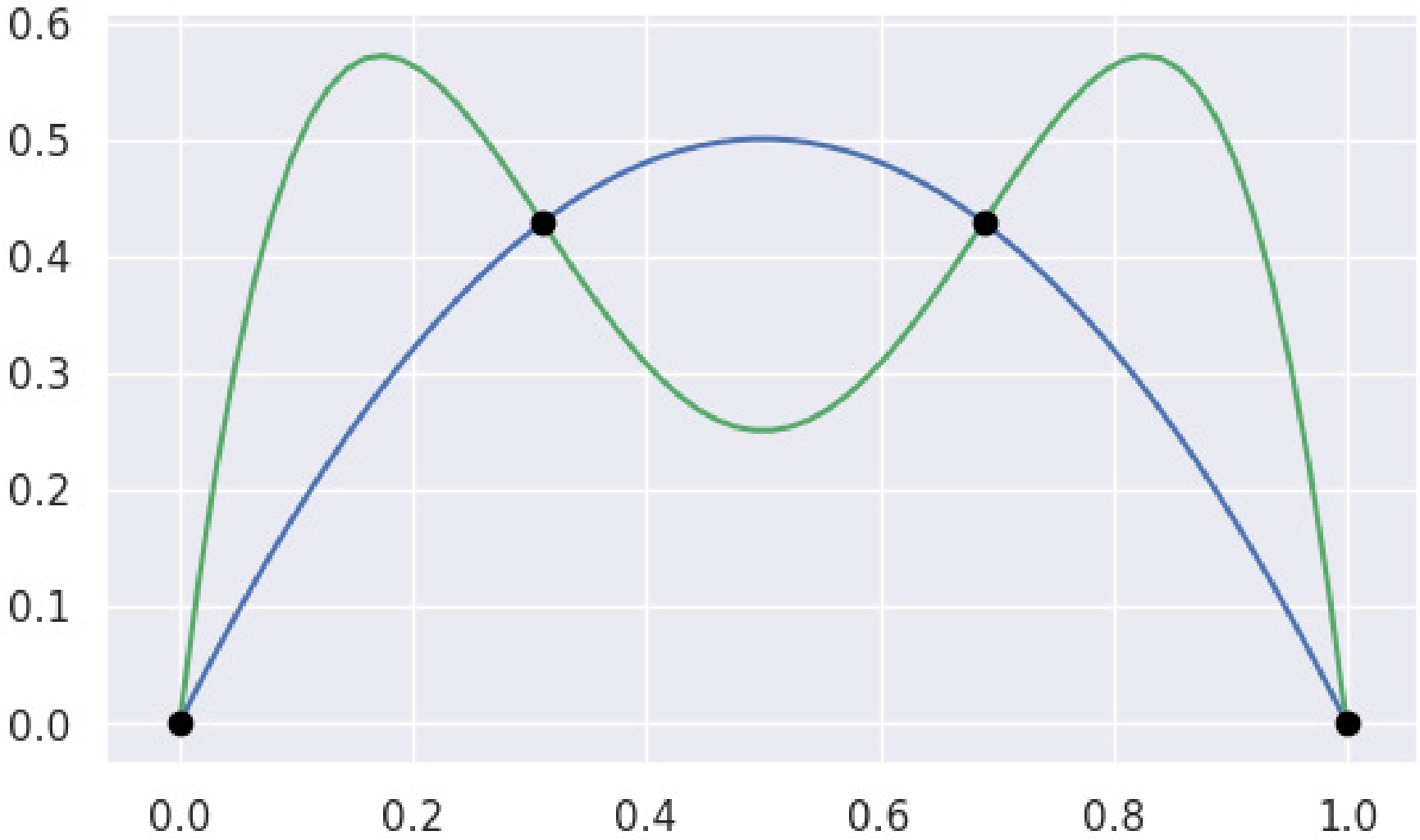


Figure: Bezier-Bernstein Approximation

Bezier-Bernstein Approximation

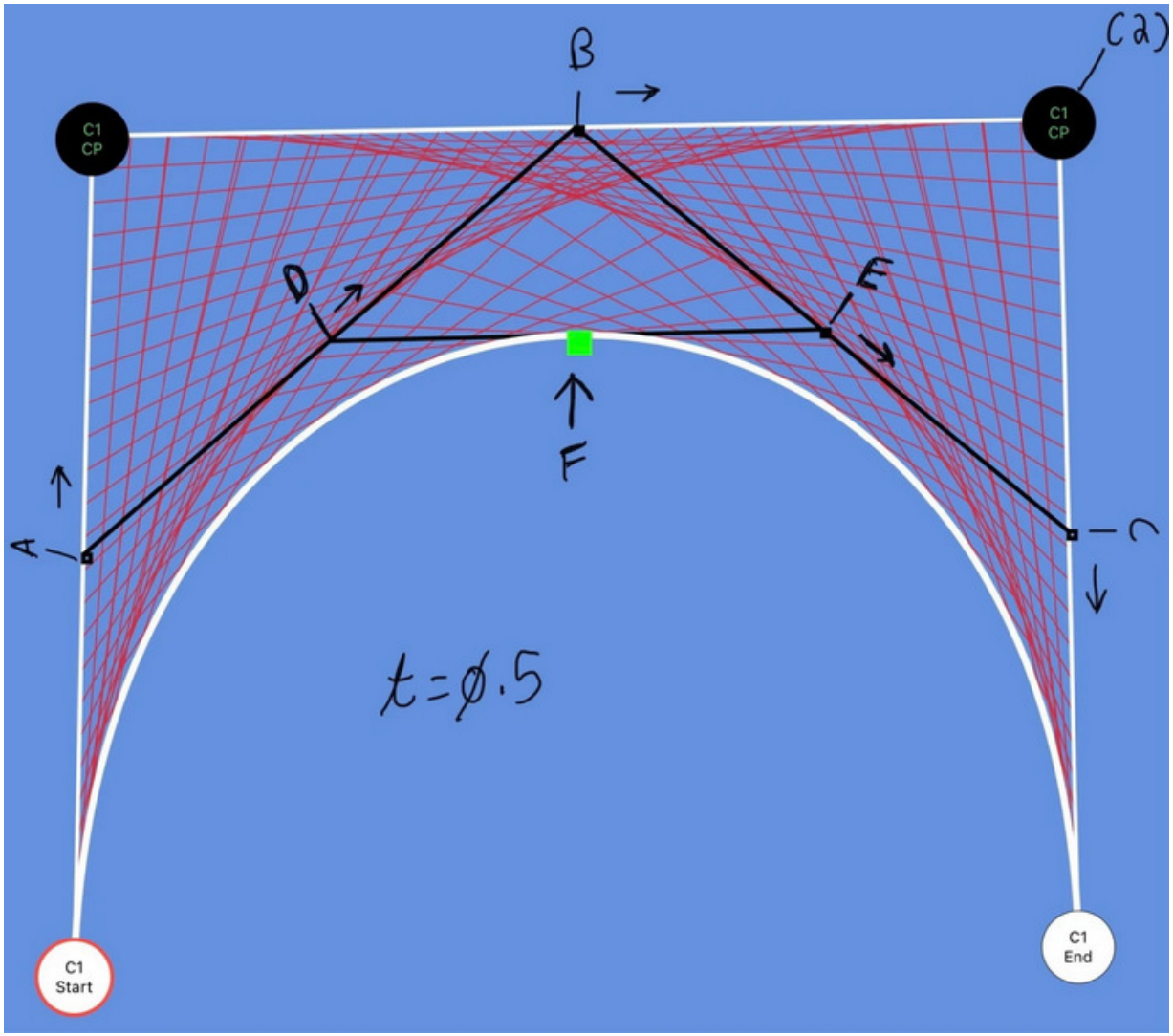


Figure: Bezier-Bernstein Approximation

Bezier-B-Spline Approximation

Bezier and **B-spline** curves are both methods for approximating curves and surfaces, with **B-splines** offering local control and while **Bezier** curves are defined globally. For this approximation, we use **B-splines** as follows:

$$P(t) = \begin{cases} x(t) = \sum_{i=0}^{n} x_i B_{i,m}(t) \\ y(t) = \sum_{i=0}^{n} y_i B_{i,m}(t) \quad 0 \leq t \leq n-m+1 \\ z(t) = \sum_{i=0}^{n} z_i B_{i,m}(t) \end{cases}$$

The m^{th}-degree **B-splines** $B_{i,m}(t), i = 0, \cdots, n$ re defined for t in the parameter range $0, n-m+1$. The knot set $t_0, \cdots, t_{n-m+1}$ is chosen to be the set $\underbrace{0, \cdots, 0}, 1, 2, \cdots, n-m, \underbrace{n-m+1, \cdots, n-m-1}$

This use of repeated knots ensures that the endpoints of the spline coincide with the endpoints of the guiding polyline.

Bezier-B-Spline Approximation

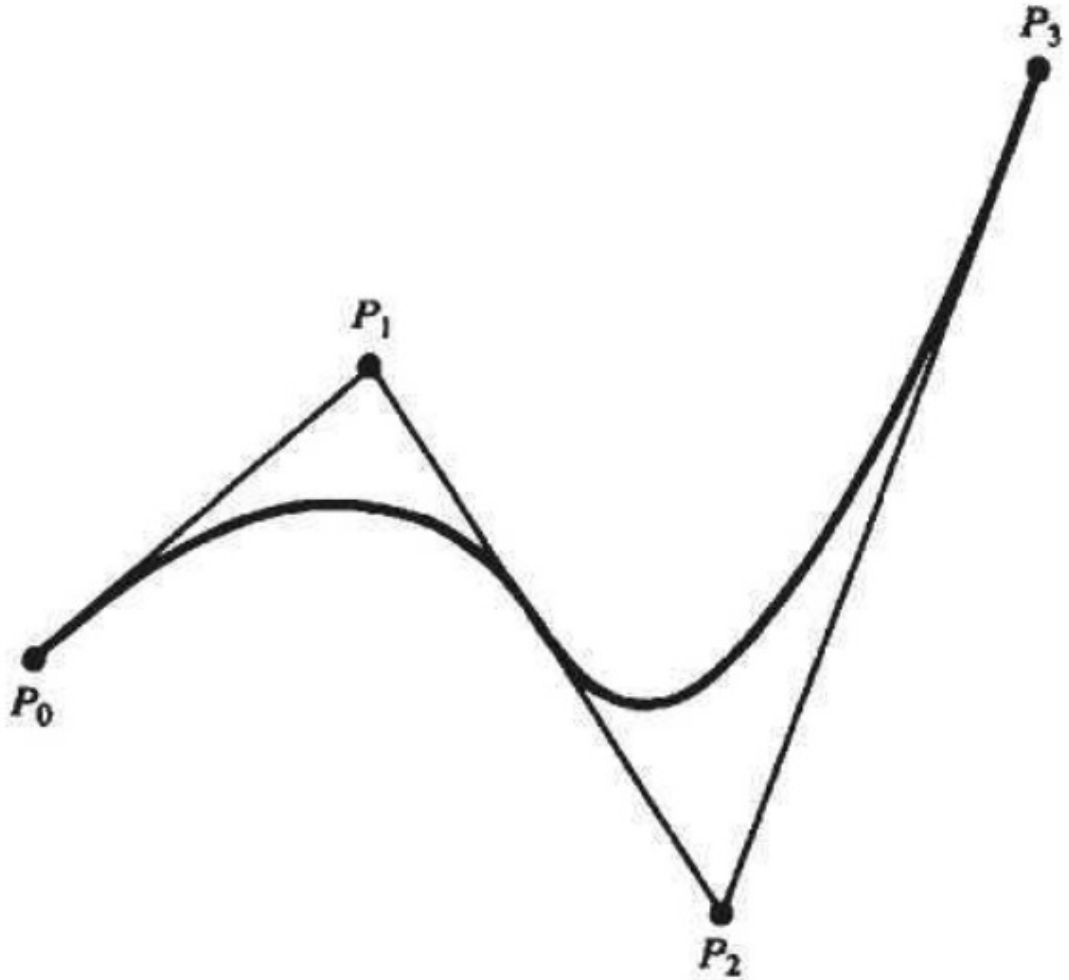


Figure: Bezier-B-Spline Approximation

Bezier-B-Spline Approximation

The following Python code:

```
from geomdl import BSpline
#pip install geomdl

curve = BSpline.Curve()
# Set degree
curve.degree = 3
# Set control points
curve.ctrlpts = [[10, 5, 10], [10, 20, -30],
[40, 10, 25], [-10, 5, 0]]
# Set knot vector
curve.knotvector = [0, 0, 0, 0, 1, 1, 1, 1]
# Set the number of curve points
curve.delta = 0.05
# Get curve points
curve_points = curve.evalpts
```

Bezier-B-Spline Approximation

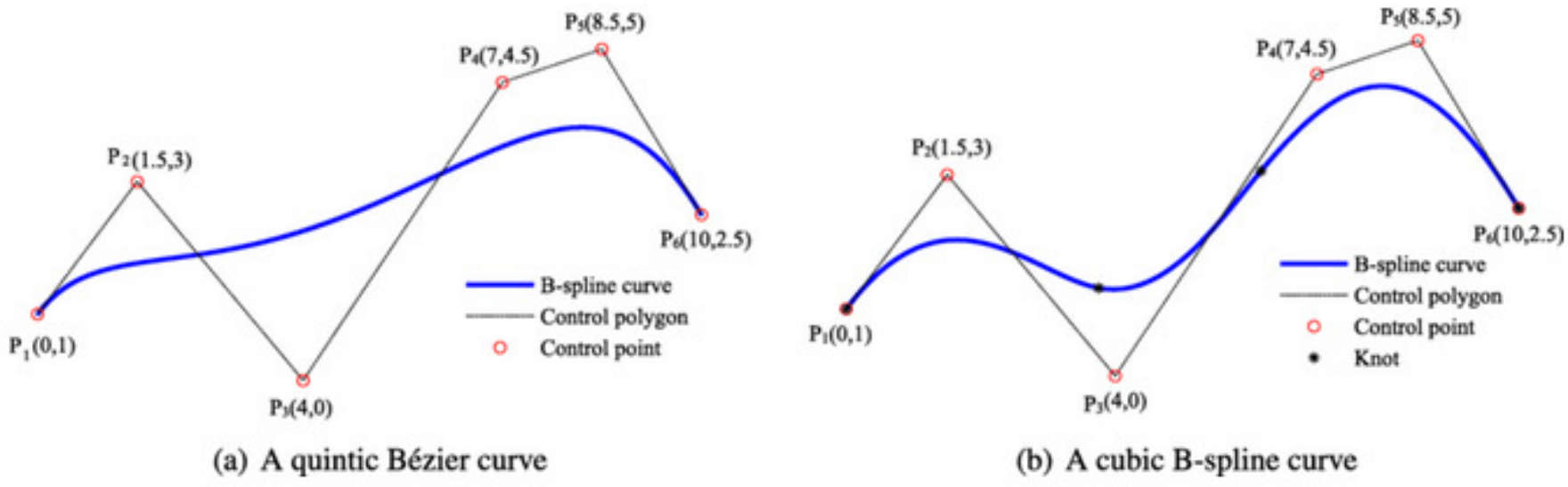


Figure: Bezier-B-Spline Approximation

Closed Curves

To construct a closed **B-spline** curve which approximates a given closed guiding polygon, we need only choose the knots $t_0, t_1, \cdots, t_{n+m+1}$ to be cyclic, i.e. $0, 1, \cdots, n, 0, 1, \cdots,$. So

$$t_{m+1} = t_0 = 0, \;\; t_{m+2} = t_1 = 1, \;\; t_{m+1+i} = t_i$$

Closed curve

A closed curve is a curve with no endpoints.

A closed curve flows continuously with no breaks or gaps. It forms a shape with a region or regions that have area. A closed curved can be made of curves and line segments. When drawing a closed curve, the starting point and ending point are the same point. A closed curve is the opposite of an open curve, which has two or more endpoints. The following are a few examples of open curves.

Closed Curves

Types of closed curves

There are two types of closed curves.

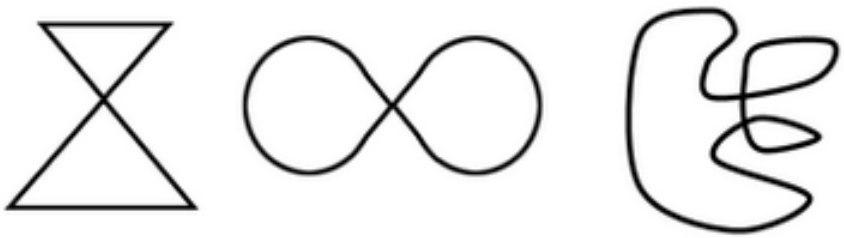
Simple closed curves

A simple closed curve is a closed curve that does not intersect anywhere except at its beginning point and end point.

Each figure above is a simple curve. None of these simple curves cross over themselves.

Non-simple closed curves

A non-simple closed curve is a closed curve that intersects itself at more than just its beginning point and endpoint. A non-simple closed curve creates two or more distinct regions.



Each of the three non-simple curves above overlaps at least one time.

Closed Curves

The following Python code:

```
import numpy as np
from scipy.interpolate import splprep, splev
import matplotlib.pyplot as plt

# define pts from the question

tck, u = splprep(pts.T, u=None, s=0.0, per=1)
u_new = np.linspace(u.min(), u.max(), 1000)
x_new, y_new = splev(u_new, tck, der=0)

plt.plot(pts[:,0], pts[:,1], 'ro')
plt.plot(x_new, y_new, 'b--')
plt.show()
```

Closed Curves

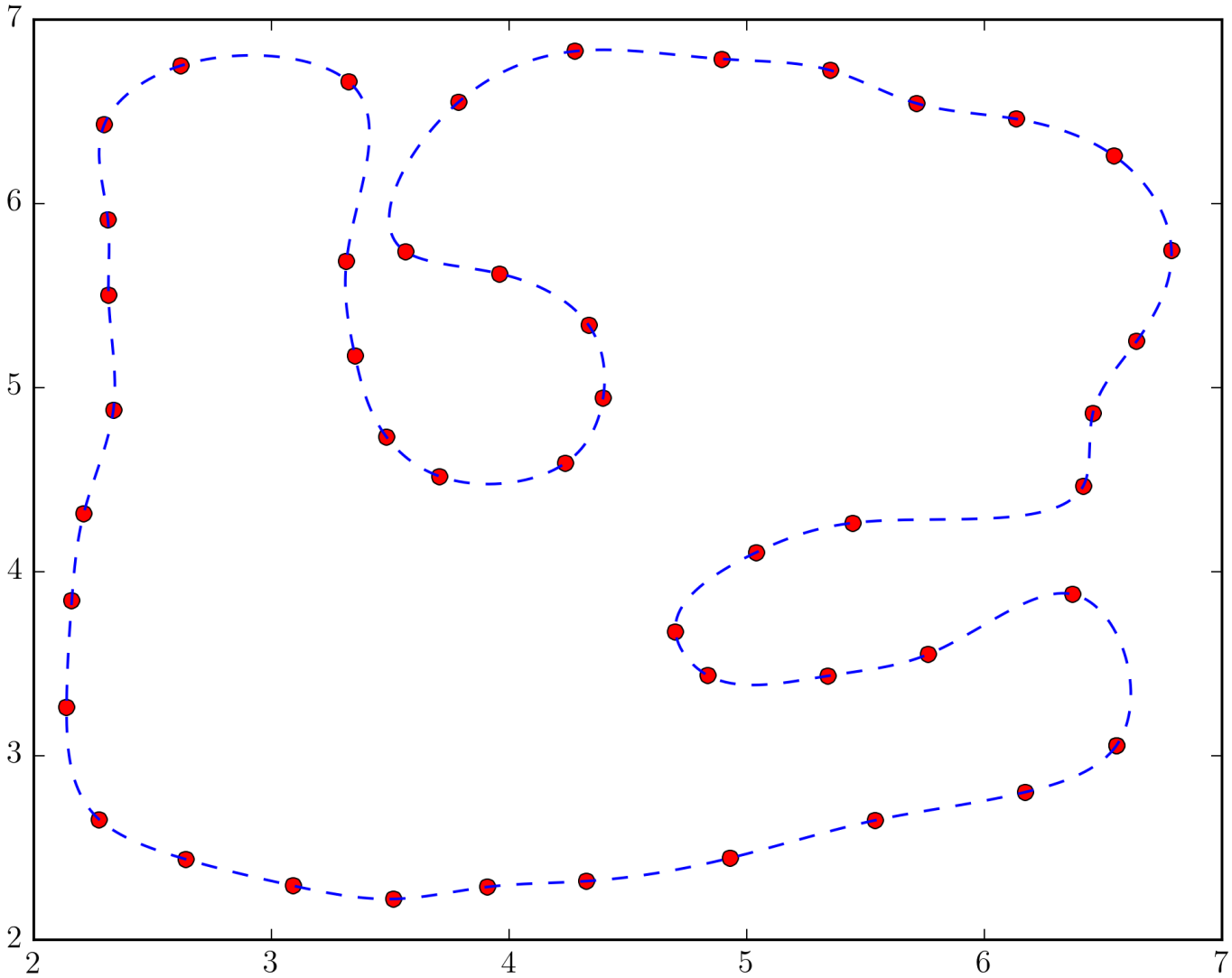


Figure: Closed Curves

Properties of Bezier-B-Spline Approximation

There are five basic properties:

1. The **Bezier-B-spline** approximation has the same properties as the **Bezier-Bernstein** approximation; in fact, they are the same piecewise polynomial if $m = n$;
2. If the guiding polyline has $m + 1$ consecutive control points which are collinear, the resulting span of the **Bezier-B-spline** will be linear;
3. The **Bezier-B-spline** approximation provides for the local control of curve shape;
4. **Bezier-B-splines** produce a closer fit to the guiding polygon than does the **Bezier-Bernstein** approximation;
5. The **Bezier-B-spline** approximation allows the use of control points P_i counted with multiplicities of 2 or more i.e. $P_i = P_{i+1} = \cdots, = P_{i+k}$ for $k \geq 1$;

The following Python code:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import comb as n_over_k
Mtk = lambda n, t, k: t**k * (1-t)**(n-k) * n_over_k(n,k)
BezierCoeff = lambda ts: [[Mtk(3,t,k) for k in range(4)] for t in ts]

fcn = np.log
tPlot = np.linspace(0. ,1. , 81)
xPlot = np.linspace(0.1,2.5, 81)
tData = tPlot[0:81:10]
xData = xPlot[0:81:10]
data = np.column_stack((xData, fcn(xData)))
Pseudoinverse = np.linalg.pinv(BezierCoeff(tData))
control_points = Pseudoinverse.dot(data)
Bezier = np.array(BezierCoeff(tPlot)).dot(control_points)
residuum = fcn(Bezier[:,0]) - Bezier[:,1]

fig, ax = plt.subplots()
ax.plot(xPlot, fcn(xPlot),'r-')
ax.plot(xData, data[:,1],'ro', label='input')
ax.plot(Bezier[:,0], Bezier[:,1],'k-', label='fit')
ax.plot(xPlot, 10.*residuum, 'b-', label='10*residuum')
ax.plot(control_points[:,0],control_points[:,1], 'ko:', fillstyle='none')
ax.legend()
fig.show()
```

Properties of Bezier-B-Spline Approximation

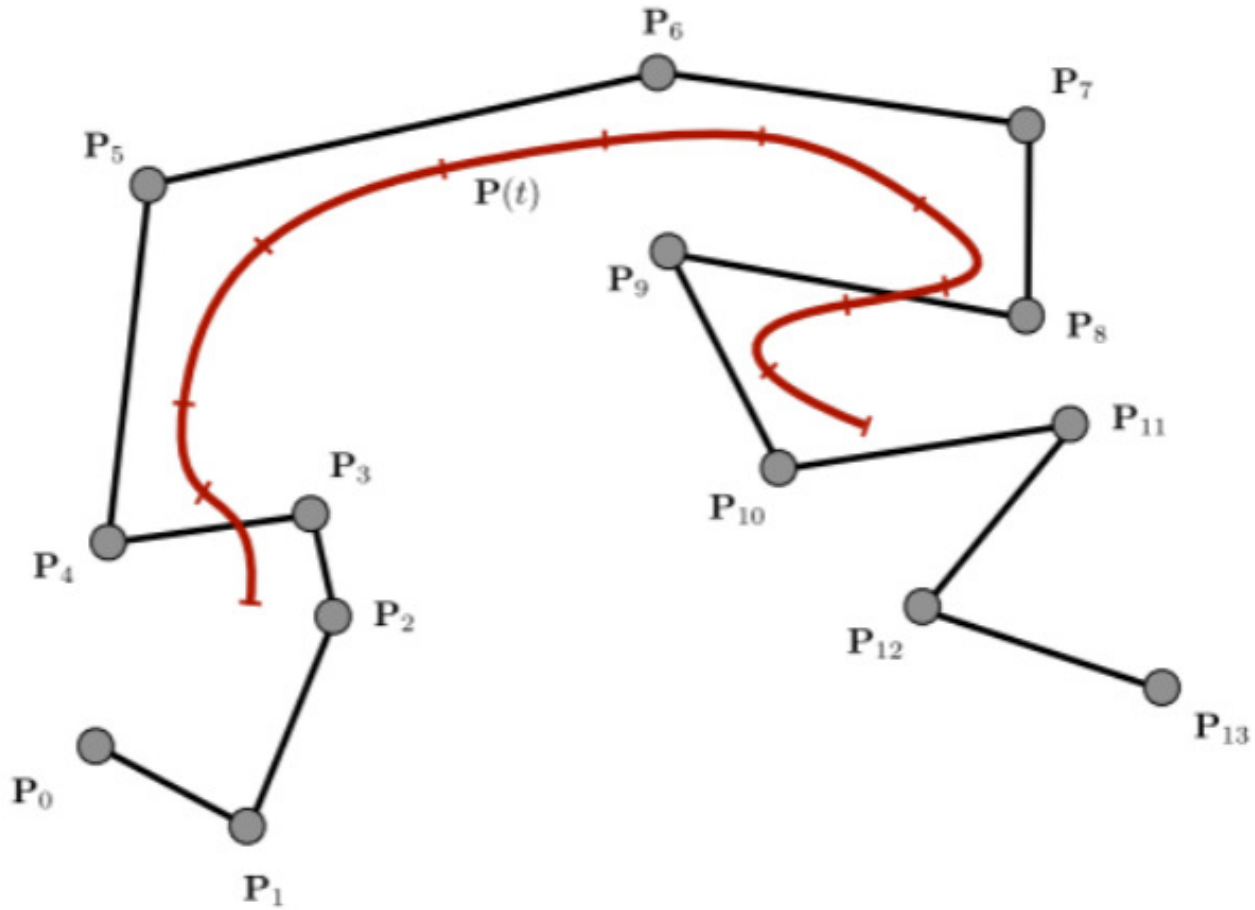


Figure: Bezier-B-Spline Approximation

Properties of Bezier-B-Spline Approximation

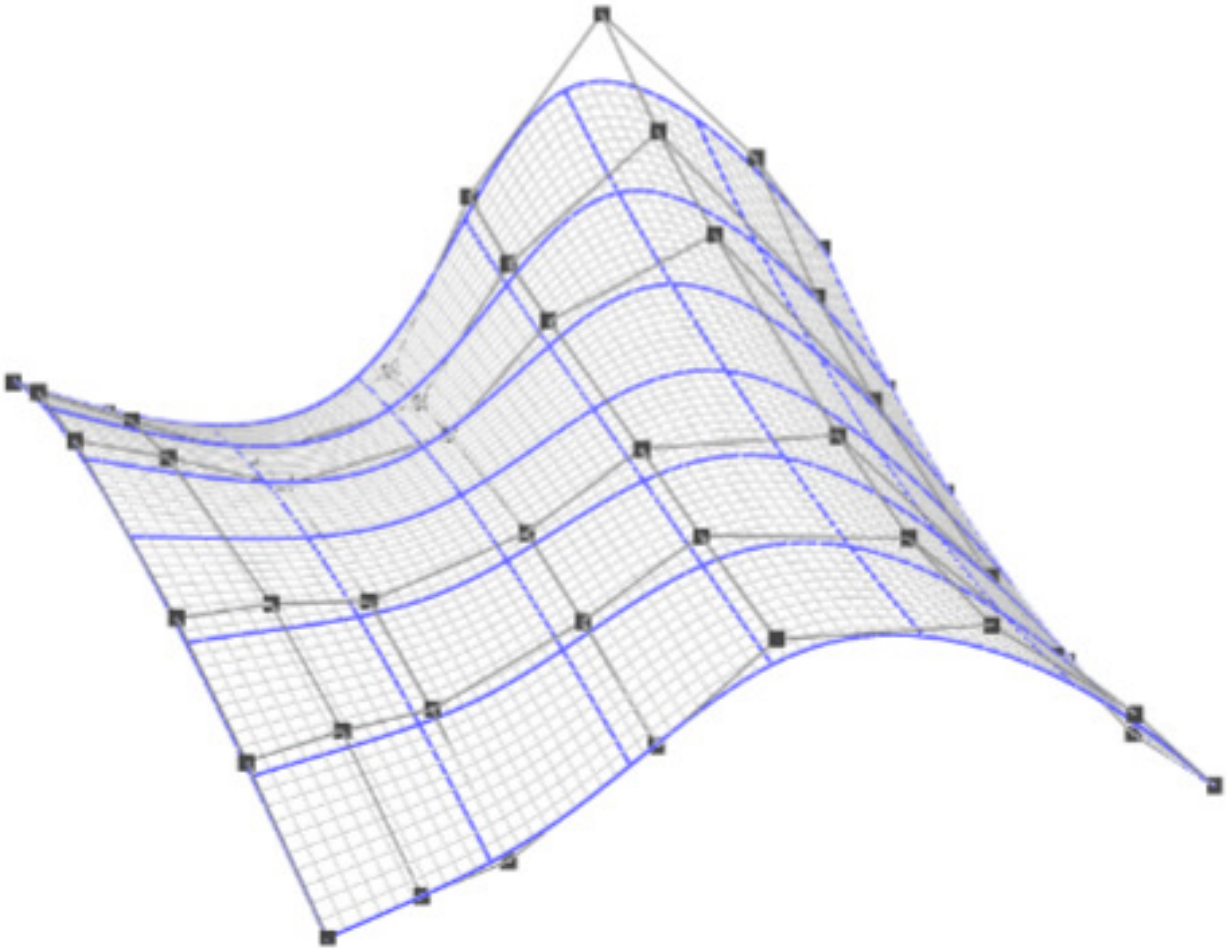


Figure: Bezier-B-Spline Approximation

Curved-Surface Design

The modeling and approximation of curved surfaces is difficult and two methods are used to represented a surface: **guiding nets** and **interpolating surface patches**.

Guiding Nets: This technique is a direct generalization of the **Bezier-Bernstein** and **Bezier-B-spline approximation** methods for curves. A **guiding net** is a polygonal net with vertices $P_{ij}(x_{ij}, y_{ij}, z_{ij})$, $i = 0, 1, \cdots, m$, and $j = 0, 1, \cdots, n$.

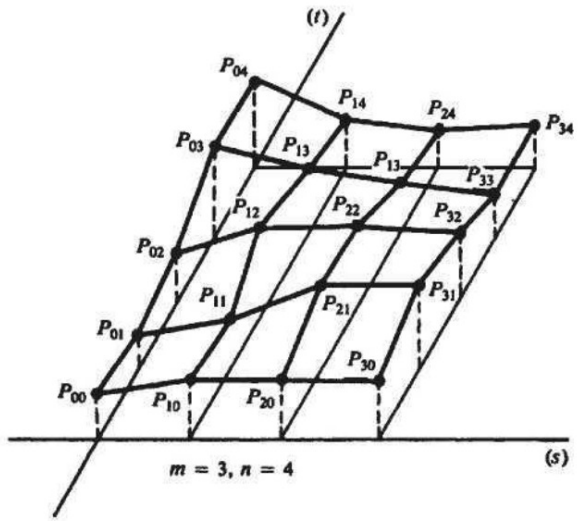


Figure: Guiding net

Curved-Surface

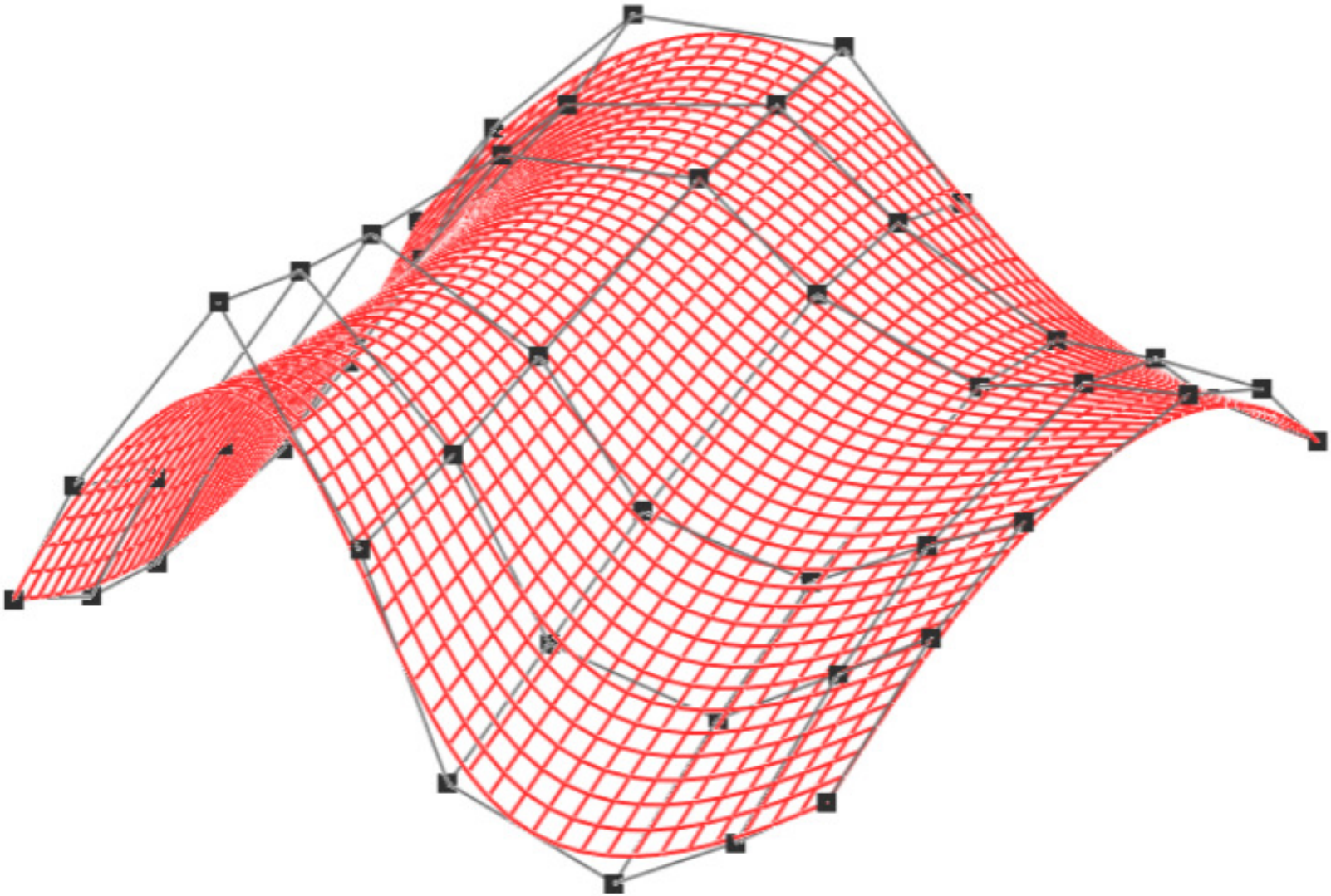


Figure: Curved-Surface

Bezier-Bernstein surface: This is the surface with parametric equations:

$$Q(s,t) = \begin{cases} x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} x_{ij} BE_{i,m} BE_{j,n}(t) \\ x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} y_{ij} BE_{i,m} BE_{j,n}(t) \\ x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} z_{ij} BE_{i,m} BE_{j,n}(t) \end{cases}$$

Here $0 < s, t < 1$ and BE are the Bernstein polynomials. This approximation has properties analogous to the one-dimensional case with respect to the corner points P_{00}, P_{m0}, P_{0n}, and P_{mn}.

Bezier-Bernstein surface

The following Python code:

```
import bezier #pip install bezier
nodes = np.asfortranarray([
      [0.0   , 0.0   ], [0.5   , 0.0   ],
      [1.0   , 0.25  ], [0.125 , 0.5   ],
      [0.375 , 0.375 ], [0.25  , 1.0   ],
       ])
surface = bezier.Surface(nodes, degree=2)
surface
```

Bezier-Bernstein surface

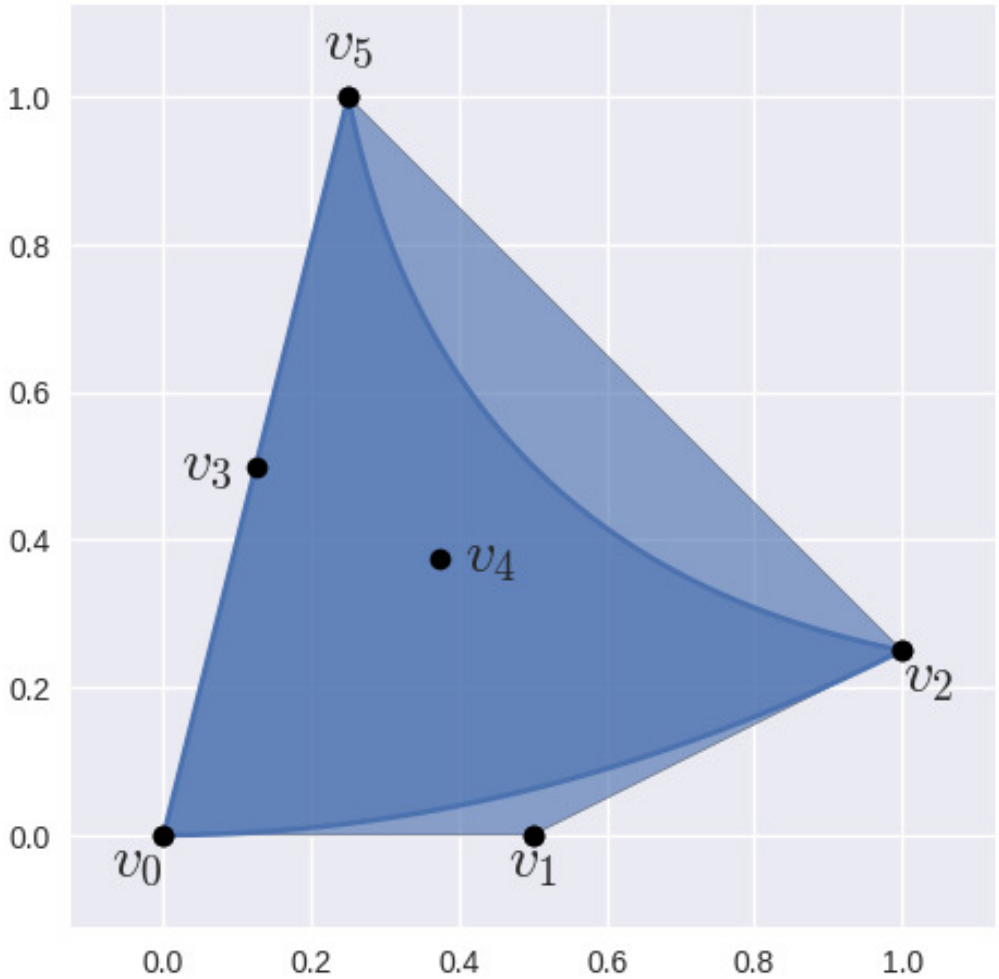


Figure: Bezier-Bernstein surface

Bezier-B-spline approximation

Bezier-B-spline approximation: In parametric form this is expressed as:

$$Q(s,t) = \begin{cases} x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} x_{ij} B_{i,\alpha}(s) B_{j,\beta}(t) \\ x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} y_{ij} B_{i,\alpha}(s) B_{j,\beta}(t) \\ x(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} z_{ij} B_{i,\alpha}(s) B_{j,\beta}(t) \end{cases}$$

where $0 \geq s \geq m - \alpha + 1$ and $0 \geq t \geq n - \beta + 1$. The knot sets for s and t used to define the B- splines $B_{i,\alpha}(s)$ and $B_{j,\beta}(t)$ are determined as in the one-dimensional case. Quadratic approximation occurs when $\alpha = \beta = 2$. Cubic approximation occurs when $\alpha = \beta = 3$.

Bezier-B-spline approximation

The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # pip install mpl-tools

def bernstein_poly(i, n, t):
    return np.math.comb(n, i) * (t ** i) * ((1 - t) ** (n - i))

def bezier_surface(control_points, num_points=100):
    n, m, _ = control_points.shape
    surface_points = np.zeros((num_points, num_points, 3))
    for i in range(num_points):
        for j in range(num_points):
            u = i / (num_points - 1)
            v = j / (num_points - 1)
            point = np.zeros(3)

            for k in range(n):
                for l in range(m):
                    b_u = bernstein_poly(k, n - 1, u)
                    b_v = bernstein_poly(l, m - 1, v)
                    point += control_points[k, l] * b_u * b_v

            surface_points[i, j] = point

    return surface_points
```

Bezier-B-spline approximation

The following Python code:

```
control_points = np.array([
    [[0, 0, 0], [1, 0, 0], [2, 0, 0]],
    [[0, 1, 0], [1, 1, 1], [2, 1, 0]],
    [[0, 2, 0], [1, 2, 0], [2, 2, 0]]
])

surface_points = bezier_surface(control_points)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(surface_points[:, :, 0], surface_points[:, :, 1],
surface_points[:, :, 2], rstride=1, cstride=1, color='cyan', alpha=0.7)

control_points_reshaped = control_points.reshape(-1, 3)
ax.scatter(control_points_reshaped[:, 0],
control_points_reshaped[:, 1], control_points_reshaped[:, 2], color='red', s=100)
plt.title("Bezier␣B-spline␣Surface")
plt.show()
```

Bezier-B-spline approximation

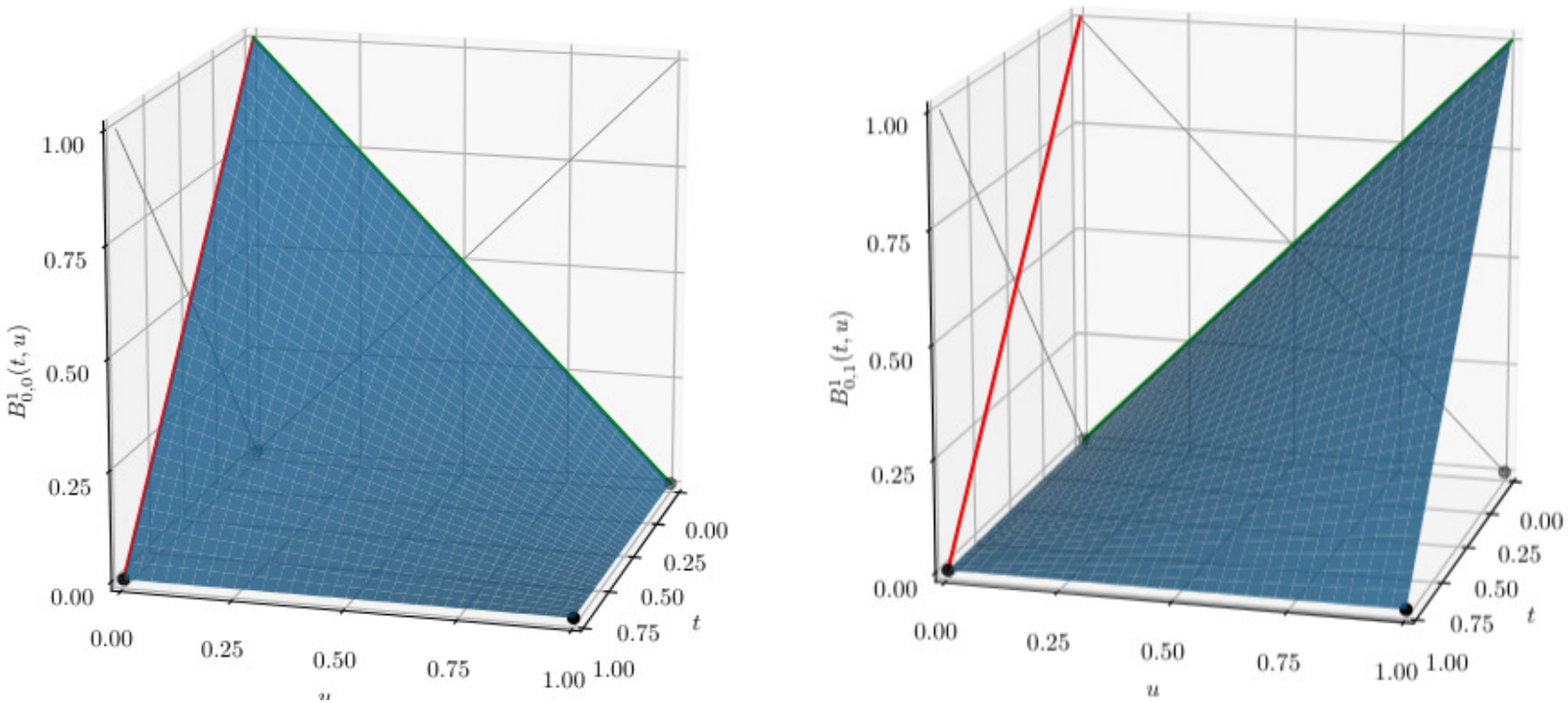


Figure: Bezier-B-spline approximation

Interpolating Surface Patches

Instead of using a given set of points $P_{ij}(x_{ij}, y_{ij}, z_{ij})$ to construct a given surface, the process of interpolating surface patches is based upon prescribing boundary curves for a surface patch and **"filling in"** the interior of the patch by interpolating between the boundary curves.

- **Coons surfaces:** For this technique, a patch is determined by specifying four bounding curves, denoted in parametric vector form as $P(s, 0)$, $P(s, 1)$, $P(0, t)$, and $P(1, t), 0 < s, t < 1$.

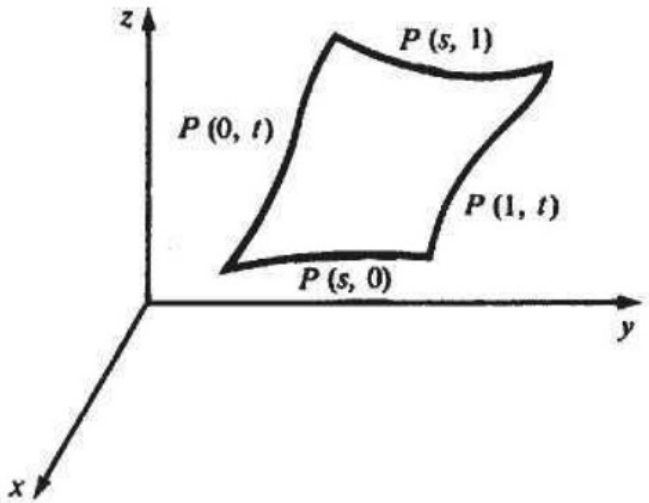


Figure: Interpolating Surface Patches

Coons Surface Patches

The Coons surface patch interpolating the boundary curves can be written in vector form by using linear interpolation:

$$Q(s,t) = P(s,0)(1-t) + P(s,1)(t) + P(0,t)(1-s) + P(1,s)(s) - P(0,0)(1-s)(1-t) - P(0,1)(1-s)(t) - P(1,0)(s)(1-t) - P(1,1)(s)(t)$$

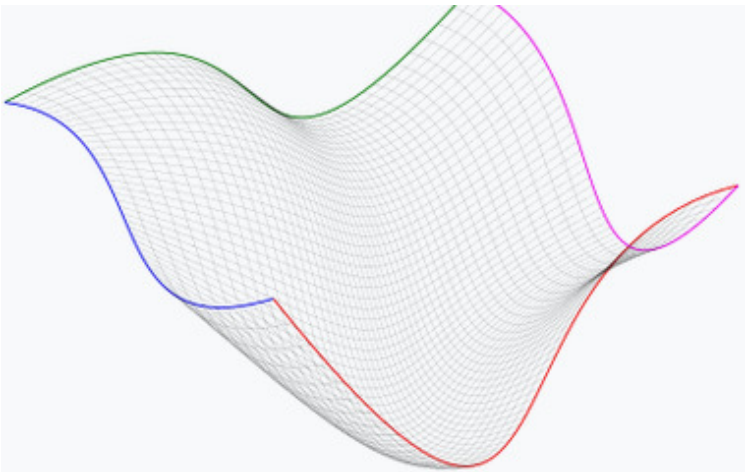


Figure: Coons Surface Patches

Coons Surface Patches in Python

The following Python code:

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

# Define the boundary curves
def P1(u):
    return np.array([u, 0, 0])

def P2(u):
    return np.array([u, 1, 0])

def P3(v):
    return np.array([0, v, 0])

def P4(v):
    return np.array([1, v, 0])

# Compute the Coons surface
def coons_surface(u, v):
    P1_u = P1(u)
    P2_u = P2(u)
    P3_v = P3(v)
    P4_v = P4(v)

    # Linear interpolation
    return (1 - u) * P3_v + u * P4_v + (1 - v) * P1_u + v * P2_u
```

Coons Surface Patches in Python

The following Python code:

```
# Generate parameters
u_vals = np.linspace(0, 1, 30)
v_vals = np.linspace(0, 1, 30)
U, V = np.meshgrid(u_vals, v_vals)

# Create an empty array for the surface points
surface_points = np.array([coons_surface(u, v) for u, v in zip(U.flatten(), V.flatten())])
X = surface_points[:, 0].reshape(U.shape)
Y = surface_points[:, 1].reshape(U.shape)
Z = surface_points[:, 2].reshape(U.shape)

# Plotting the Coons surface
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, alpha=0.7, rstride=1, cstride=1, color='cyan')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Coons Surface Patch')
plt.show()
```

Coons Surface Patches

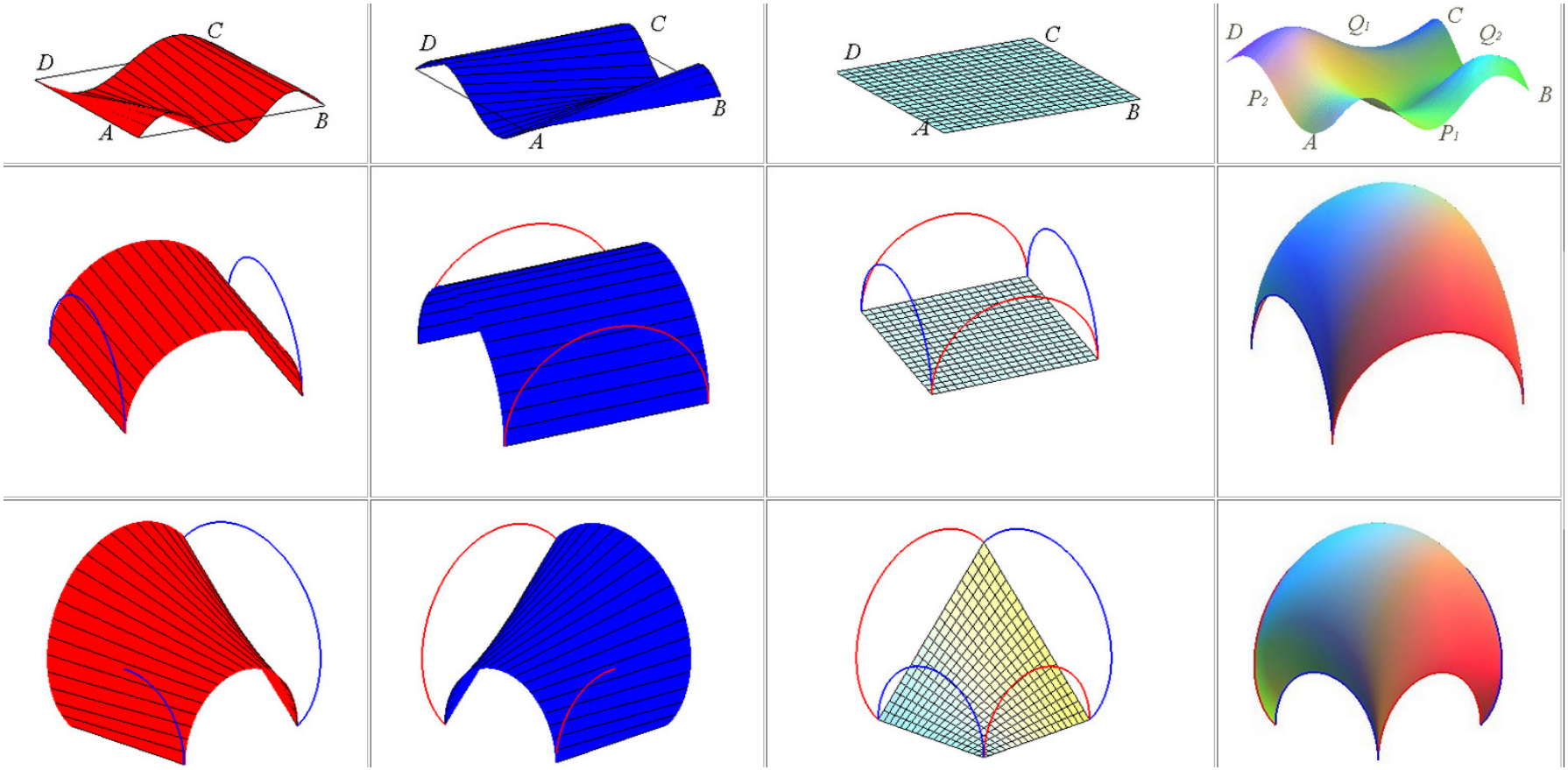


Figure: Coons Surface Patches

Lofted surfaces

Lofted surfaces:

- Lofting is used where the surface to be constructed stretches in a given direction;
- Given two or more space curves, called cross-section curves, lofting is the process of blending the cross sections together using longitudinal blending curves;
- The simplest example is linear blending between cross-section curves $P_1(s)$ and $P_2(s)$;
- The lofted surface is $Q(s,t) = (1-t)P_1(s) + tP_2(s)$;
- An example is the hull of a ship;

Lofted surfaces

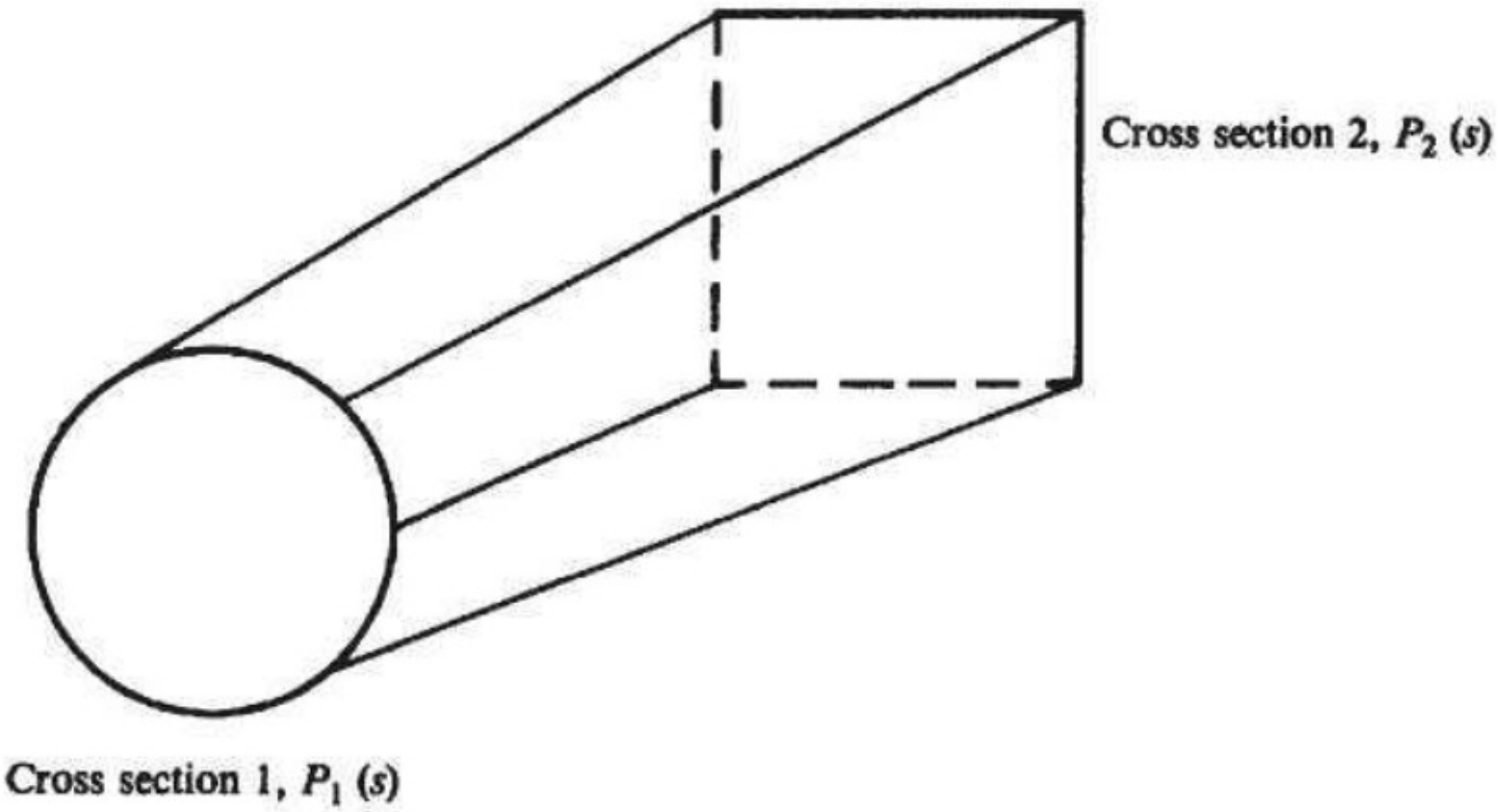


Figure: Lofted Surface

Lofted surfaces

The following Python code:

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

# Number of points
n_points = 100
theta = np.linspace(0, 2 * np.pi, n_points)
x1 = np.cos(theta)
y1 = np.sin(theta)
z1 = np.zeros(n_points)
x2 = np.cos(theta)
y2 = np.sin(theta)
z2 = np.ones(n_points)
# Prepare a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for t in np.linspace(0, 1, 30):
    x = (1 - t) * x1 + t * x2
    y = (1 - t) * y1 + t * y2
    z = (1 - t) * z1 + t * z2
    ax.plot(x, y, z, color='b')

ax.set_xlabel('X'),ax.set_ylabel('Y'),ax.set_zlabel('Z')
ax.set_xlim([-1.5, 1.5]),ax.set_ylim([-1.5, 1.5]),ax.set_zlim([-0.5, 1.5])
# Show the plot
plt.title('Lofted Surface between Two Circles'), plt.show()
```

Lofted surfaces

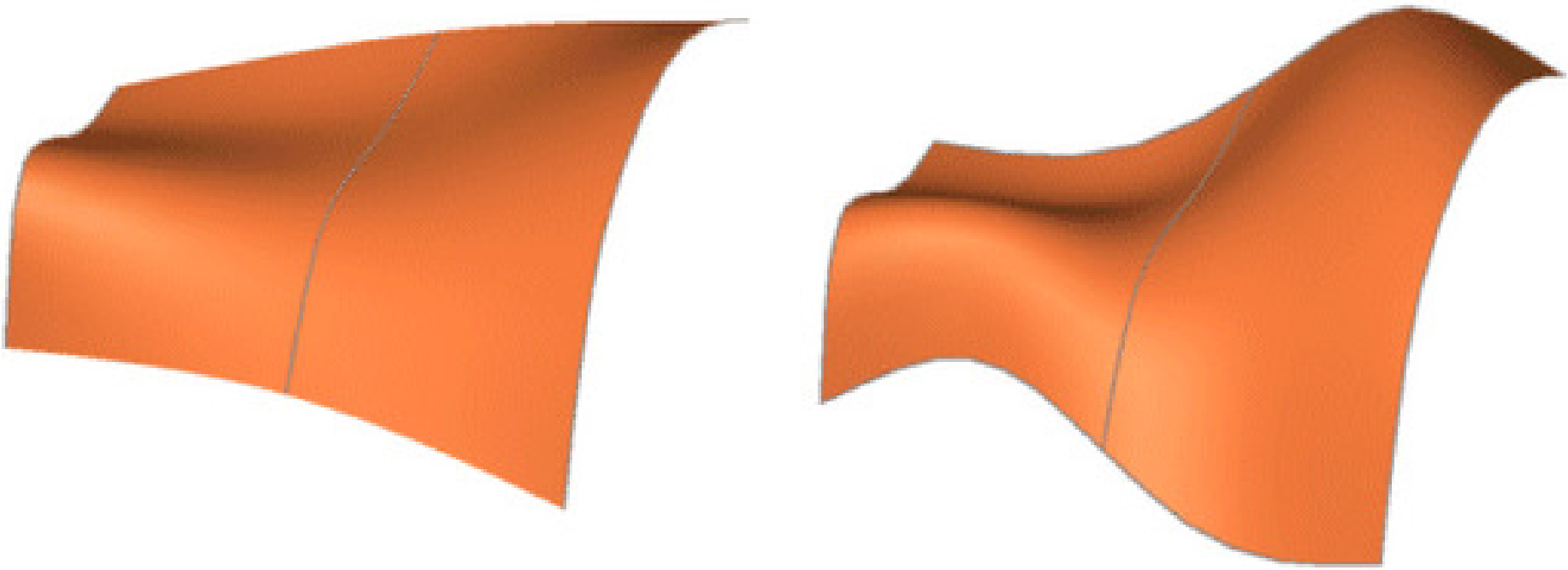


Figure: Lofted Surface

Transforming Curves and Surfaces

All the curve and surface models that we have constructed have the general form:

$$x = \Sigma x_i \Phi_i, y = \Sigma y_i \Phi_i, z = \Sigma z_i \Phi_i$$

If **M** is 2×2 transformation matrix as

$$\mathrm{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The matrix M is applied to the functions x, y and z:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a(\Sigma x_i \Phi_i) + b(\Sigma y_i \Phi_i) \\ c(\Sigma x_i \Phi_i) + d(\Sigma y_i \Phi_i) \end{pmatrix} = \begin{pmatrix} \Sigma(a x_i + b y_i)\Phi_i \\ \Sigma(c x_i + d y_i)\Phi_i \end{pmatrix}$$

The transformed functions are then

$$\tilde{x} = \Sigma(a x_i + b y_i)\Phi_i, \tilde{y} = \Sigma(c x_i + d y_i)\Phi_i$$

To transform these curves and surfaces, it is necessary only to transform the coefficients (x_i, y_i).

Quadric Surfaces

Spheres, **cylinders**, and **cones** are part of the family of surfaces called quadric surfaces. A **quadric surface** is defined by an equation which is of the second degree (x, y, orz). The canonical quadric surfaces are as follows:

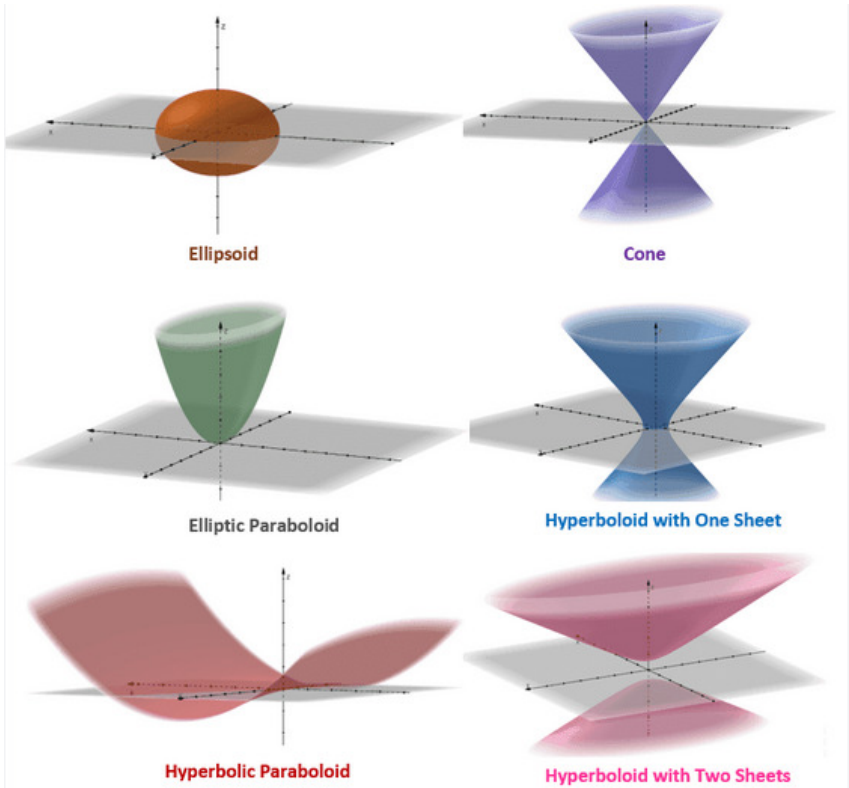


Figure: Different quadric surfaces

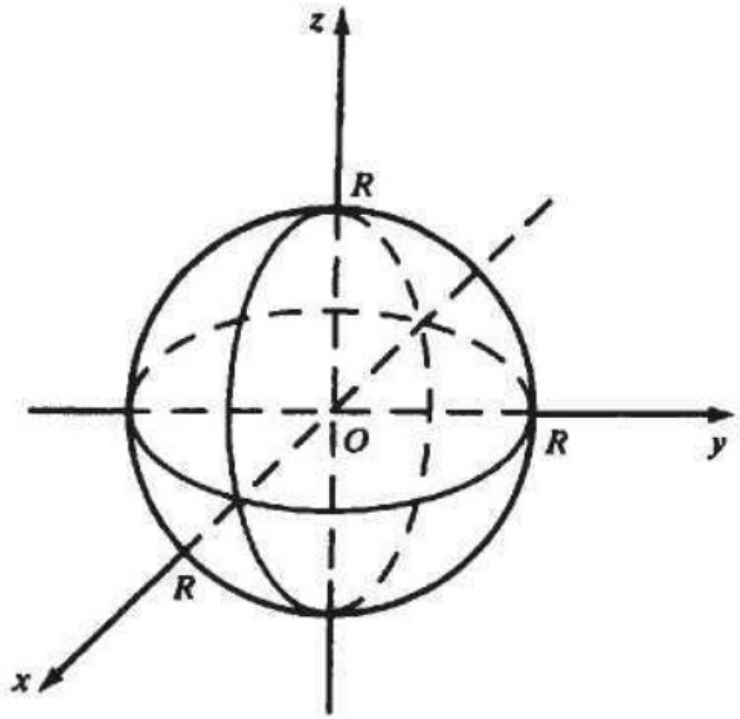
Quadric Surfaces

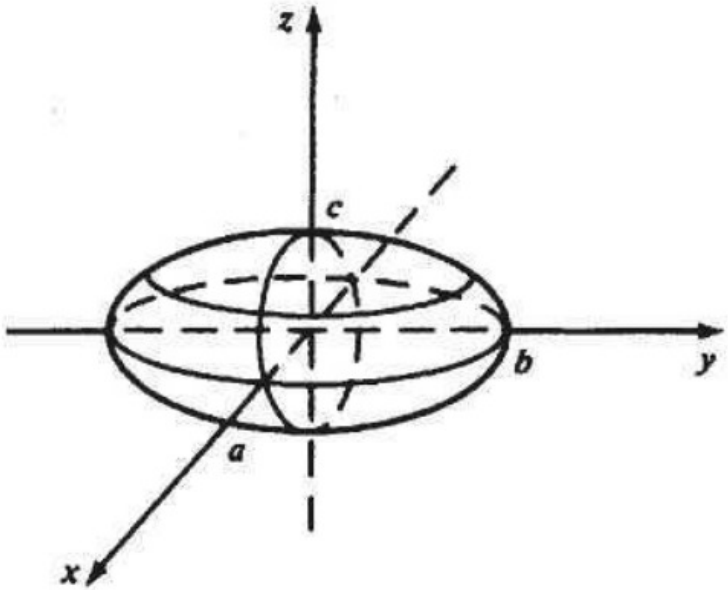
Sphere:

$$x^2 + y^2 + z^2 = R^2$$

Ellipsoid:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$





Draw Sphere and Ellipsoid in Python

The following Python code:

```
#----------------------Sphere----------------------#
from vpython import * #pip install vpython
# the first sphere
sphere(pos = vector(-1, 2, 0),size = vector(1, 1, 1),
        color = vector(15, 1, 1))
# the second sphere
sphere(pos = vector(1, -2, 0),size = vector(3, 3, 3),
        color = vector(5, 10, 1))
#--------------------Ellipsoid--------------------#
from vpython import * ellipsoid(color = vector(0.4,
 0.2, 0.6),opacity = 0.5,
 shininess = 1,emissive = False)
```

One-sheeted Hyperboloid and Two-sheeted Hyperboloid

One-sheeted Hyperboloid:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1$$

Two-sheeted Hyperboloid:

$$\frac{z^2}{c^2} - \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

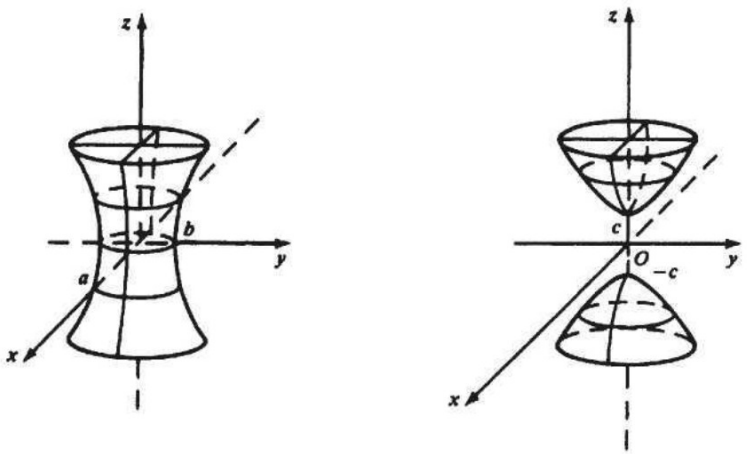


Figure: One-sheeted Hyperboloid and Two-sheeted Hyperboloid

One-sheeted Hyperboloid

The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
theta = np.linspace(0, 2*np.pi, 100)
z = np.linspace(-2, 2, 100)
Theta, Z = np.meshgrid(theta, z)
X = np.cosh(Z) * np.cos(Theta)
Y = np.cosh(Z) * np.sin(Theta)
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Hyperboloid of One Sheet')
plt.show()
```

Two-sheeted Hyperboloid

The following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
theta = np.linspace(0, 2*np.pi, 100)
z = np.linspace(-2, 2, 100)
Theta, Z = np.meshgrid(theta, z)
X = np.sinh(Z) * np.cos(Theta)
Y = np.sinh(Z) * np.sin(Theta)
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Hyperboloid of Two Sheets')
plt.show()
```

Quadric Surfaces

- **Elliptic Cylinder:**

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

When $a = b = R$, the cylinder is a circular cylinder of radius R

- **Elliptic Paraboloid:**

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = cz$$

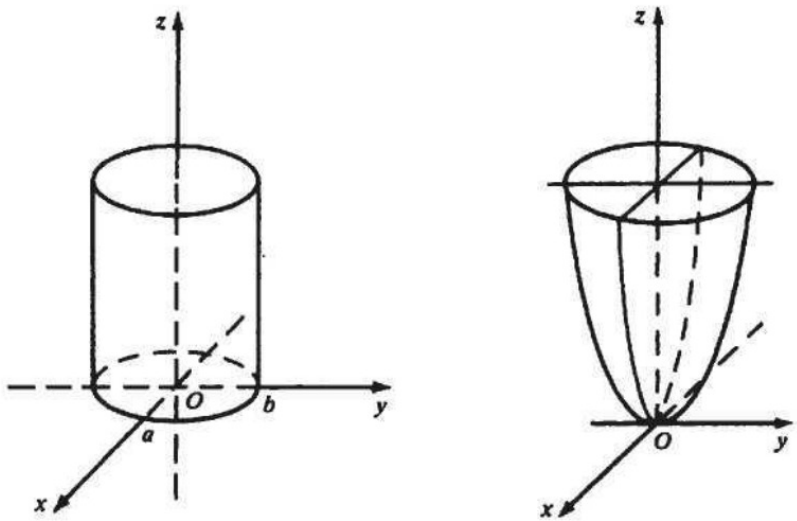


Figure: Elliptic Cylinder and Elliptic Paraboloid

Elliptic Cylinder

The following Python code:

```
import math #pip install vtk
# noinspection PyUnresolvedReferences
import vtkmodules.vtkInteractionStyle
# noinspection PyUnresolvedReferences
import vtkmodules.vtkRenderingOpenGL2
from vtkmodules.vtkCommonColor import vtkNamedColors
from vtkmodules.vtkCommonCore import (vtkMath, vtkPoints)
from vtkmodules.vtkCommonDataModel import (vtkCellArray,
    vtkPolyData, vtkPolyLine)
from vtkmodules.vtkFiltersModeling import vtkLinearExtrusionFilter
from vtkmodules.vtkInteractionStyle import vtkInteractorStyleTrackballCamera
from vtkmodules.vtkRenderingCore import ( vtkActor, vtkCamera,  vtkPolyDataMapper,
    vtkProperty, vtkRenderWindow, vtkRenderWindowInteractor,  vtkRenderer)

def main():
    colors = vtkNamedColors()

    angle = 0
    r1 = 50
    r2 = 30
    centerX = 10.0
    centerY = 5.0

    points = vtkPoints()
    idx = 0
    while angle <= 2.0 * vtkMath.Pi() + (vtkMath.Pi() / 60.0):
        points.InsertNextPoint(r1 * math.cos(angle) + centerX,
                               r2 * math.sin(angle) + centerY,0.0)
        angle = angle + (vtkMath.Pi() / 60.0)
        idx += 1
```

Elliptic Cylinder

The following Python code:

```
line = vtkPolyLine()
line.GetPointIds().SetNumberOfIds(idx)
for i in range(0, idx):
    line.GetPointIds().SetId(i, i)

lines = vtkCellArray()
lines.InsertNextCell(line)

polyData = vtkPolyData()
polyData.SetPoints(points)
polyData.SetLines(lines)

extrude = vtkLinearExtrusionFilter()
extrude.SetInputData(polyData)
extrude.SetExtrusionTypeToNormalExtrusion()
extrude.SetVector(0, 0, 100.0)
extrude.Update()

lineMapper = vtkPolyDataMapper()
lineMapper.SetInputData(polyData)

lineActor = vtkActor()
lineActor.SetMapper(lineMapper)
lineActor.GetProperty().SetColor(colors.GetColor3d("Peacock"))
```

Elliptic Cylinder

The following Python code:

```
mapper = vtkPolyDataMapper()
mapper.SetInputConnection(extrude.GetOutputPort())

back = vtkProperty()
back.SetColor(colors.GetColor3d("Tomato"))

actor = vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(colors.GetColor3d("Banana"))
actor.SetBackfaceProperty(back)

ren = vtkRenderer()
ren.SetBackground(colors.GetColor3d("SlateGray"))
ren.AddActor(actor)
ren.AddActor(lineActor)

renWin = vtkRenderWindow()
renWin.SetWindowName("EllipticalCylinder")
renWin.AddRenderer(ren)
renWin.SetSize(600, 600)

iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

style = vtkInteractorStyleTrackballCamera()
iren.SetInteractorStyle(style)
```

Elliptic Cylinder

The following Python code:

```
    camera = vtkCamera()
    camera.SetPosition(0, 1, 0)
    camera.SetFocalPoint(0, 0, 0)
    camera.SetViewUp(0, 0, 1)
    camera.Azimuth(30)
    camera.Elevation(30)

    ren.SetActiveCamera(camera)
    ren.ResetCamera()
    ren.ResetCameraClippingRange()
    renWin.Render()
    iren.Start()

if __name__ == '__main__':
    main()
```

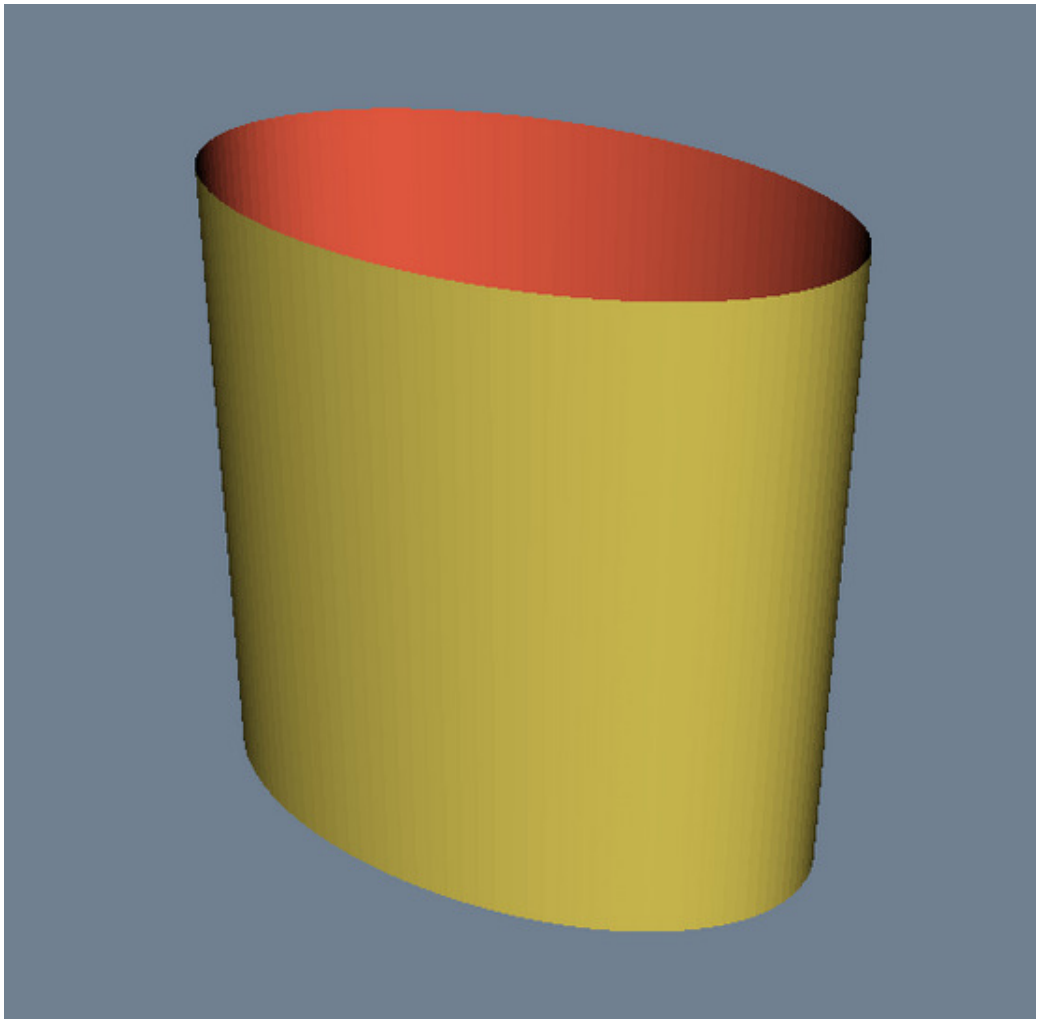


Figure: Elliptic Cylinder

Elliptic Paraboloid

The following Python code:

```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
# Create the surface
radius = 5
hole_radius = 4

# Generate the grid in cylindrical coordinates
r  = np.linspace(0, radius, 100)
theta = np.linspace(0, 2 * np.pi, 100)
R, THETA = np.meshgrid(r, theta)

X, Y = R * np.cos(THETA), R * np.sin(THETA)
a=0.6;b=0.6;c=0.6
Z1 = (X/a)**2+(Y/b)**2 # Elliptic paraboloid

# Do not plot the inner region
x = np.where(X**2+Y**2<=hole_radius**2,np.NAN,X)
y = np.where(X**2+Y**2<=hole_radius**2,np.NAN,Y)

# Plot the surface
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x, y, Z1, cmap=cm.coolwarm, linewidth=0,
antialiased=True, cstride=2, rstride=2)

ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```

Elliptic Paraboloid

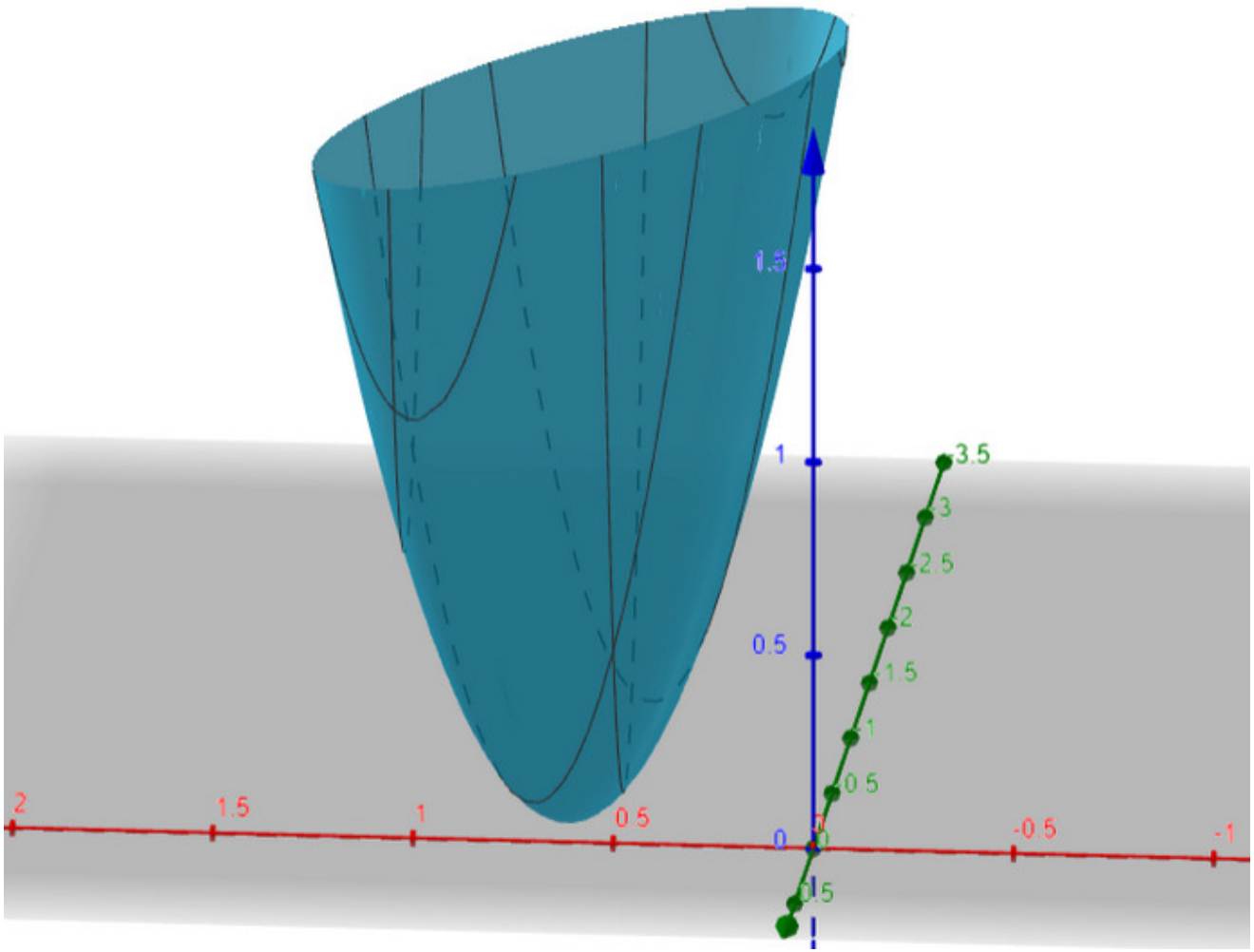


Figure: Elliptic Paraboloid

Quadric Surfaces

Hyperbolic Parboloid:

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = cz$$

Elliptic Cone:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = z^2$$

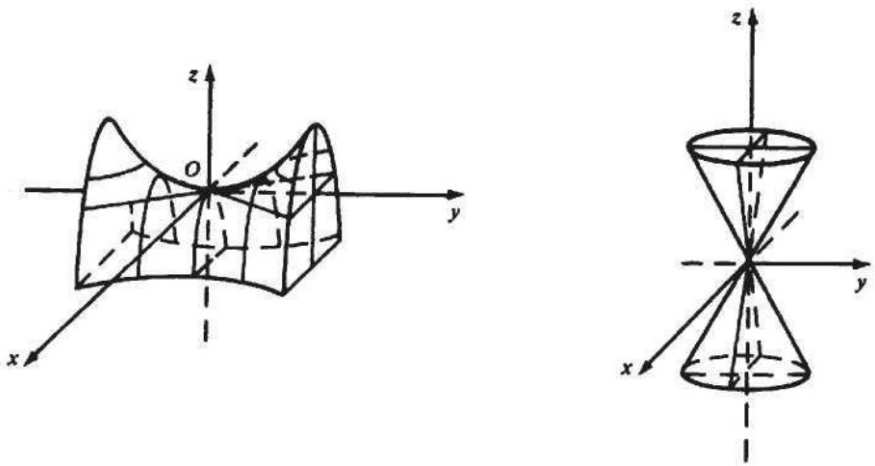


Figure: Hyperbolic Parboloid and Elliptic Cone

When $a = b = R$, we have a right circular cone. If $z > 0$, we have the upper cone.

Hyperbolic paraboloid

The following Python code:

```
import numpy as np
import pptk # pip install pptk
def f(x, y):
        return x ** 2 - y ** 2

t = np.linspace(-1.0, 1.0, 100)
x, y = np.meshgrid(t, t)
z = f(x, y)

P = np.stack([x, y, z], axis=-1).reshape(-1, 3)
v = pptk.viewer(P)
v.attributes(P[:, 2])  # color points by their z-coordinates
v.set(point_size=0.005)
# calculate gradients
dPdx = np.stack([np.gradient(x, axis=1), np.gradient(y, axis=1), np.gradient(z, axis=1)], ax
dPdy = np.stack([np.gradient(x, axis=0), np.gradient(y, axis=0), np.gradient(z, axis=0)], ax
# calculate gradient magnitudes
mag = np.sqrt(np.sum(dPdx ** 2 + dPdy ** 2, axis=-1)).flatten()

# calculate normal vectors
N = np.cross(dPdx, dPdy, axis=-1)
N /= np.sqrt(np.sum(N ** 2, axis=-1))[:, :, None]
N = N.reshape(-1, 3)
# set per-point attributes
v.attributes(P[:, 2], mag, 0.5 * (N + 1))
```

Hyperbolic paraboloid

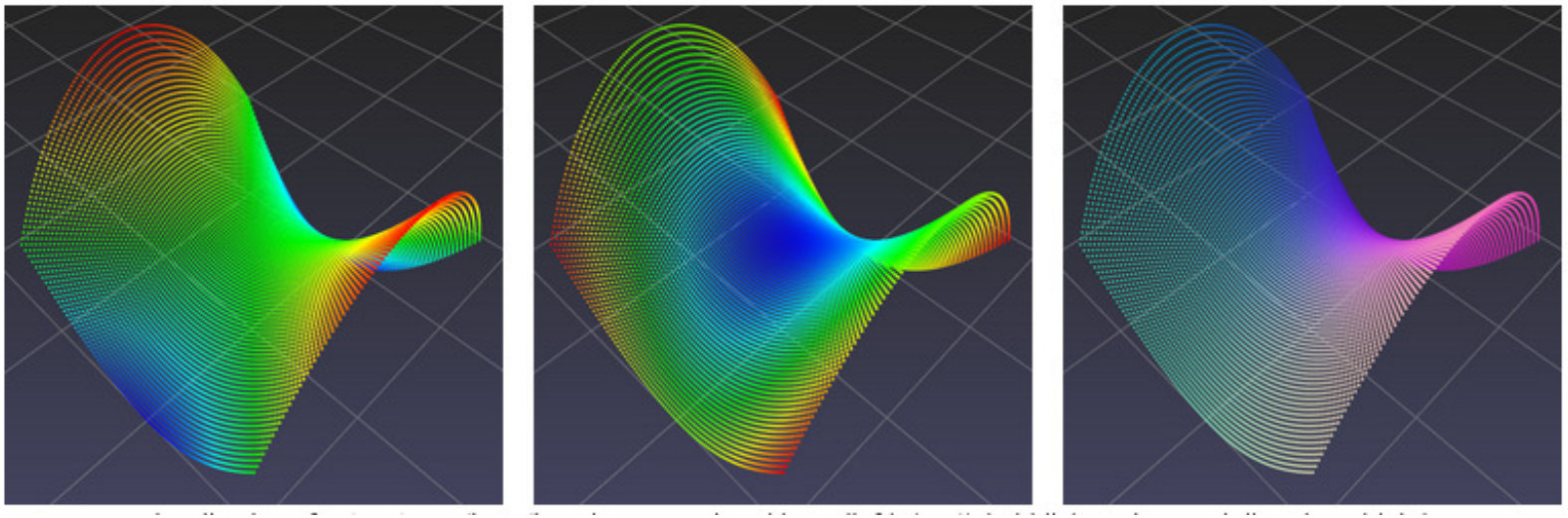


Figure: Hyperbolic paraboloid

Elliptic Cone

The following Python code:

```
import vtk
import time
cone = vtk.vtkConeSource()
cone.SetHeight( 3.0 )
cone.SetRadius( 1.0 )
cone.SetResolution( 10 )
coneMapper = vtk.vtkPolyDataMapper()
coneMapper.SetInputConnection( cone.GetOutputPort() )

coneActor = vtk.vtkActor()
coneActor.SetMapper( coneMapper )
ren1= vtk.vtkRenderer()
ren1.AddActor( coneActor )
ren1.SetBackground( 0.1, 0.2, 0.4 )
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer( ren1 )
renWin.SetSize( 300, 300 )
for i in range(0,360):
    time.sleep(0.03)

    renWin.Render()
    ren1.GetActiveCamera().Azimuth( 1 )
```

Elliptic Cone

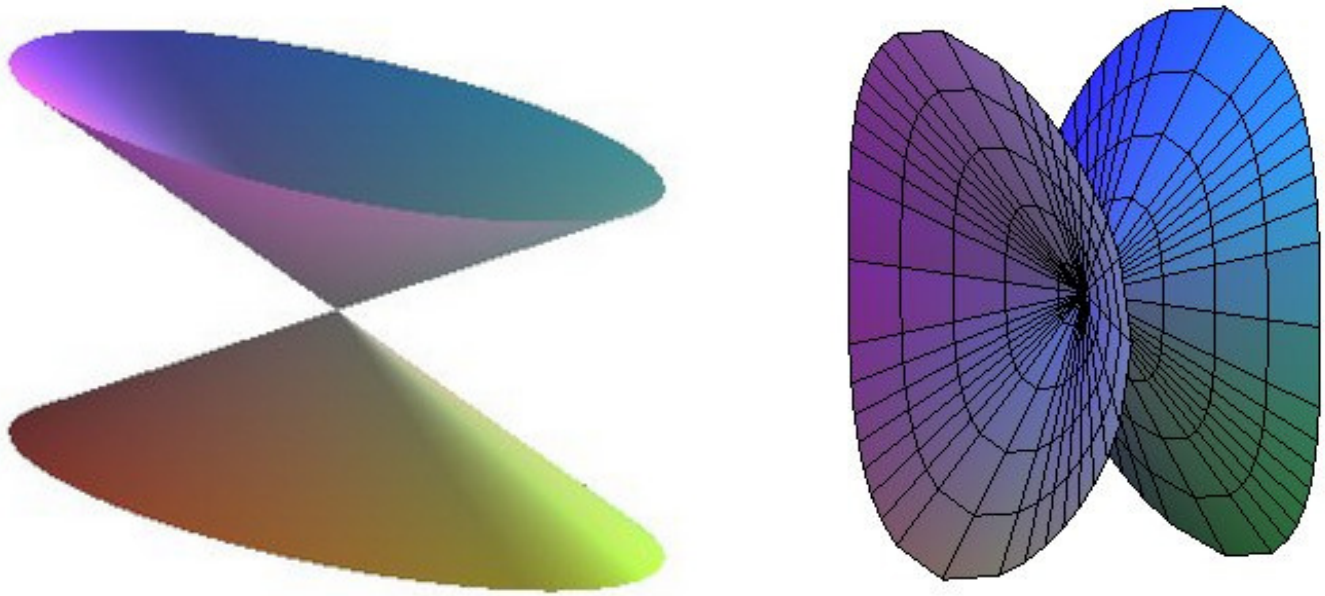


Figure: Elliptic Cone

Quadric Surfaces

surface	equation	ρ_3	ρ_4	sgn (Δ)	k
coincident planes	$x^2 = 0$	1	1		
ellipsoid (imaginary)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = -1$	3	4	+	1
ellipsoid (real)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$	3	4	−	1
elliptic cone (imaginary)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 0$	3	3		1
elliptic cone (real)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 0$	3	3		0
elliptic cylinder (imaginary)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} = -1$	2	3		1
elliptic cylinder (real)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$	2	3		1
elliptic paraboloid	$z = \frac{x^2}{a^2} + \frac{y^2}{b^2}$	2	4	−	1
hyperbolic cylinder	$\frac{x^2}{a^2} - \frac{y^2}{b^2} = -1$	2	3		0
hyperbolic paraboloid	$z = \frac{y^2}{b^2} - \frac{x^2}{a^2}$	2	4	+	0
hyperboloid of one sheet	$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1$	3	4	+	0
hyperboloid of two sheets	$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = -1$	3	4	−	0
intersecting planes (imaginary)	$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 0$	2	2		1
intersecting planes (real)	$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 0$	2	2		0
parabolic cylinder	$x^2 + 2\,r\,z = 0$	1	3		
parallel planes (imaginary)	$x^2 = -a^2$	1	2		
parallel planes (real)	$x^2 = a^2$	1	2		

Quadric Surfaces

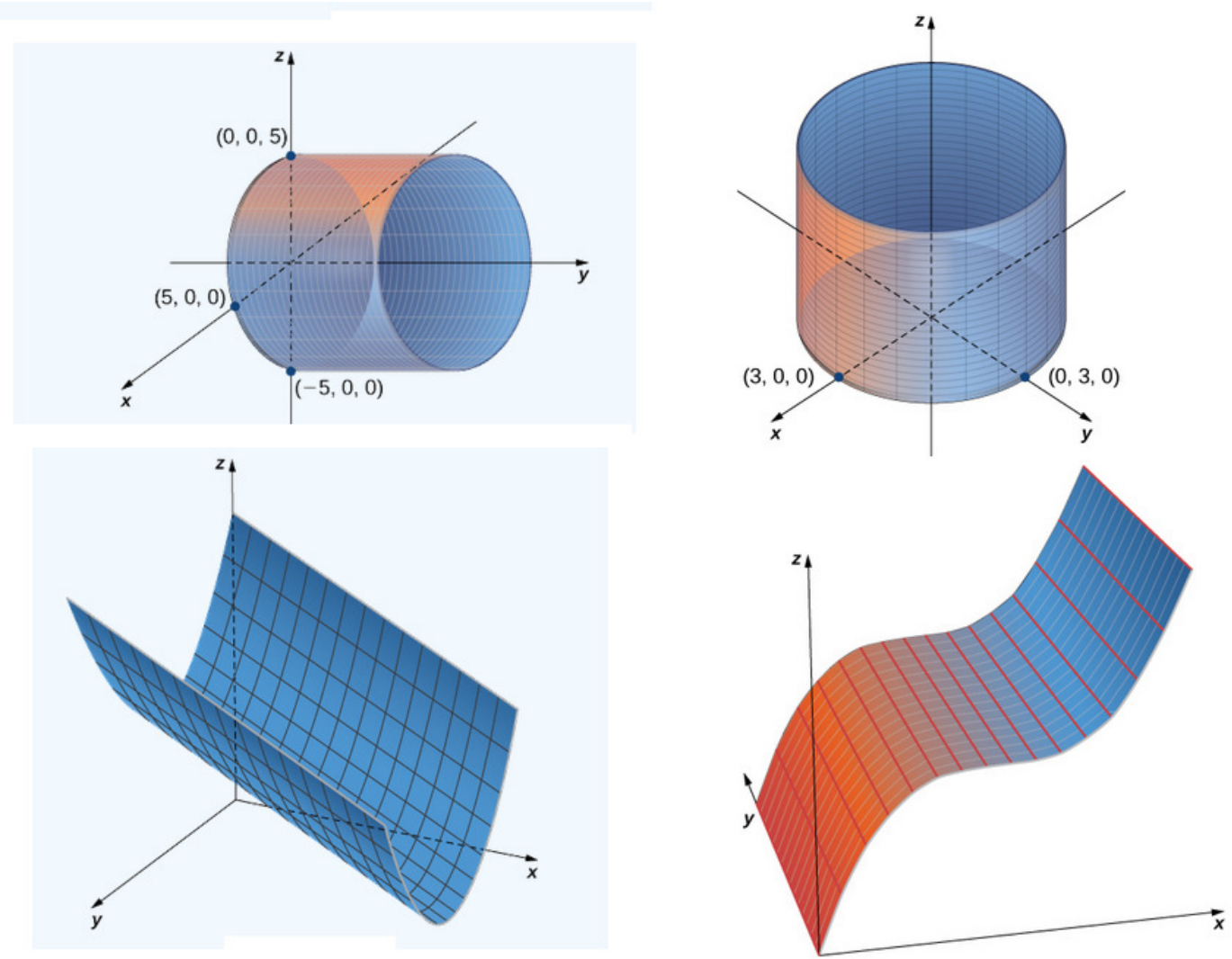


Figure: Different Quadric Surfaces

THANK YOU