Knots

November 18, 2023

[ ]:
```
%matplotlib widget
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import sympy as sp
import math
from svgpathtools import svg2paths, CubicBezier, QuadraticBezier, Line

def ot_in_cartesian(x_val, y_val, z_val):
    # Define symbolic variables
    x, y, z = sp.symbols('x y z')

    # Express r and theta in terms of x, y, z
    r = sp.sqrt(x**2 + y**2)
    theta = sp.atan2(y, x)

    # Calculate the derivatives of theta with respect to x and y
    dtheta_dx = sp.diff(theta, x)
    dtheta_dy = sp.diff(theta, y)

    # Components of the differential form in cylindrical coordinates
    dx_component = r * sp.sin(r) * dtheta_dx
    dy_component = r * sp.sin(r) * dtheta_dy
    dz_component = sp.cos(r)

    # Substitute specific values and evaluate each component
    dx_component_substituted = dx_component.subs({x: x_val, y: y_val, z:␣
↪z_val}).evalf()
    dy_component_substituted = dy_component.subs({x: x_val, y: y_val, z:␣
↪z_val}).evalf()
    dz_component_substituted = dz_component.subs({x: x_val, y: y_val, z:␣
↪z_val}).evalf()

    return [dx_component_substituted, dy_component_substituted,␣
↪dz_component_substituted]
```

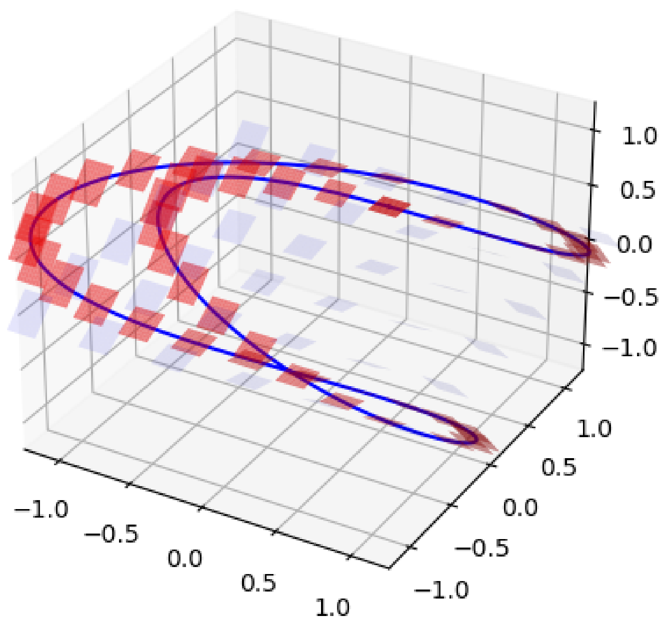
```
def get_basis(matrix, x_val, y_val, z_val):
    # Define symbolic variables x, y, z
    x, y, z = sp.symbols('x y z')

    # Convert the matrix elements to SymPy expressions
    matrix_sympy = [sp.sympify(element) for element in matrix]

    # For a matrix [a, b, c], the kernel is solved by ax + by + cz = 0
    a, b, c = matrix_sympy

    # Constructing the basis vectors
    basis1 = np.array([1, 0, -a/c], dtype=object) if c != 0 else np.array([1,␣
↪0, 0], dtype=object)
    basis2 = np.array([0, 1, -b/c], dtype=object) if c != 0 else np.array([0,␣
↪1, 0], dtype=object)

    # Function to substitute values into a symbolic expression or return the␣
↪value if it's not symbolic
    def substitute_if_symbolic(expr, substitutions):
        return expr.subs(substitutions) if isinstance(expr, sp.Expr) else expr

    # Substituting the values of x, y, z
    substitutions = {x: x_val, y: y_val, z: z_val}
    basis1_evaluated = np.array([substitute_if_symbolic(el, substitutions) for␣
↪el in basis1], dtype=float)
    basis2_evaluated = np.array([substitute_if_symbolic(el, substitutions) for␣
↪el in basis2], dtype=float)

    return basis1_evaluated, basis2_evaluated


def plot_plane(ax, x, y, z, form, size=0.1, height_limit=0.3, surfcolor='blue',␣
↪alpha=0.5):
    v1, v2 = get_basis(form, x, y, z)

    # Create a grid on the plane
    u, v = np.meshgrid(np.linspace(-size, size, 10), np.linspace(-size, size,␣
↪10))
    plane_x = x + u * v1[0] + v * v2[0]
    plane_y = y + u * v1[1] + v * v2[1]
    plane_z = z + u * v1[2] + v * v2[2]

    # Clamping the z-values to be within the height_limit
    plane_z = np.clip(plane_z, z - height_limit/2, z + height_limit/2)

    # Plot the plane
```

```
    ax.plot_surface(plane_x, plane_y, plane_z, color=surfcolor, alpha=alpha)
```

[ ]:
```
def knot1(ax, form, n):
    # Define the parametric equations for the knot
      t = np.linspace(0, 2*np.pi, 200)
      x = 3 * np.sin(t) * np.cos(t)
      y = np.cos(t)
      z = np.sin(t)**3

      ax.plot(x, y, z, color='b')

      # Take n equidistant points from t
      t = np.linspace(0, 2 * np.pi, n)
      x = 3 * np.sin(t) * np.cos(t)
      y = np.cos(t)
      z = np.sin(t)**3

      # Call plot_plane for these points
      for xi, yi, zi in zip(x, y, z):
          plot_plane(ax, xi, yi, zi, form, surfcolor='red')


def knot2(ax, form, n):
    # Define the parametric equations for the knot
      t = np.linspace(0, 2*np.pi, 200)
      x = np.cos(t)
      y = np.sin(2*t)
      z = 2/3 * np.sin(t)*np.cos(2*t) -4/3 * np.sin(2*t)*np.cos(t)

      ax.plot(x, y, z, color='b')

      # Take n equidistant points from t
      t = np.linspace(0, 2 * np.pi, n)
      x = np.cos(t)
      y = np.sin(2*t)
      z = 2/3 * np.sin(t)*np.cos(2*t) -4/3 * np.sin(2*t)*np.cos(t)

      # Call plot_plane for these points
      for xi, yi, zi in zip(x, y, z):
          plot_plane(ax, xi, yi, zi, form, surfcolor='red')
```

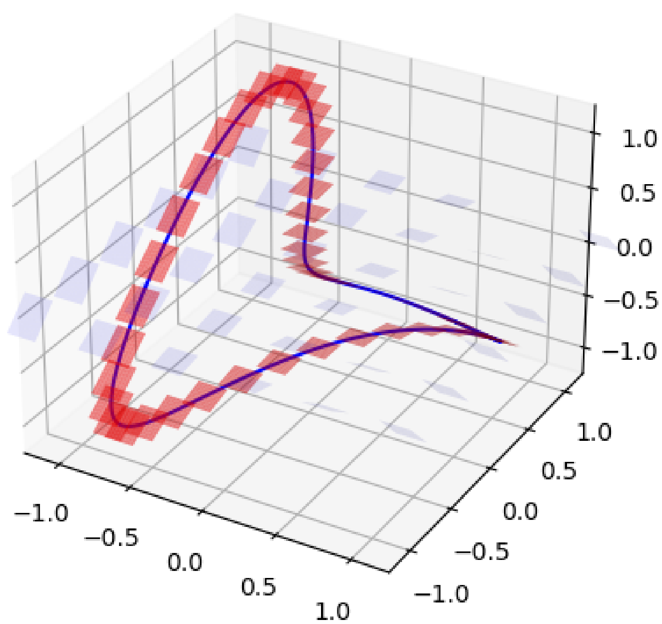
[ ]:
```
# Create a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

form = [0, 'x', 1]
```

```
# Generate a grid in the xy plane
grid_size = 1.2
step = 0.5
for x in np.arange(-grid_size, grid_size + step, step):
    for y in np.arange(-grid_size, grid_size + step, step):
        plot_plane(ax, x, y, 0, form, alpha=0.1)

knot1(ax, form, 50)

# Set plot limits
ax.set_xlim([-grid_size, grid_size])
ax.set_ylim([-grid_size, grid_size])
ax.set_zlim([-grid_size, grid_size])

# Show the plot
plt.show()
```



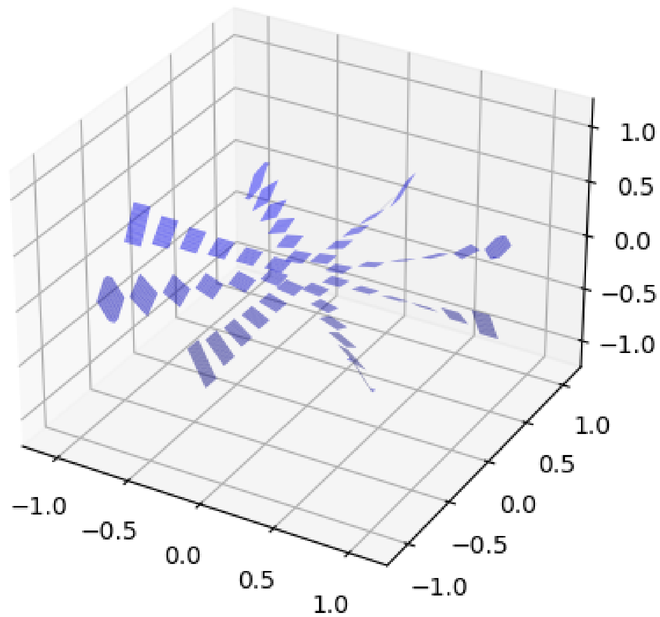
[ ]:
```
# Create a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

```
# Generate a grid in the xy plane
grid_size = 1.2
step = 0.5
for x in np.arange(-grid_size, grid_size + step, step):
    for y in np.arange(-grid_size, grid_size + step, step):
        plot_plane(ax, x, y, 0, form, alpha=0.1)

knot2(ax, form, 50)

# Set plot limits
ax.set_xlim([-grid_size, grid_size])
ax.set_ylim([-grid_size, grid_size])
ax.set_zlim([-grid_size, grid_size])

# Show the plot
plt.show()
```



[ ]:
```
# Create a 3D plot
fig = plt.figure()
```

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


# Define the polar grid parameters
max_radius = 1.2
radius_step = 0.2
angle_step = math.pi / 4

# Generate the polar grid and convert to Cartesian coordinates
for r in np.arange(0.2, max_radius + radius_step, radius_step):
    for theta in np.arange(0, 2 * math.pi, angle_step):
        x = r * math.cos(theta)
        y = r * math.sin(theta)

        # Call the function with Cartesian coordinates
        ot_values = ot_in_cartesian(x, y, 0)

        # Plotting - Assuming 'ax' is predefined and 'plot_plane' is a function␣
↪you have defined
        plot_plane(ax, x, y, 0, ot_values, alpha=0.5, size=0.06)


# Set plot limits
ax.set_xlim([-grid_size, grid_size])
ax.set_ylim([-grid_size, grid_size])
ax.set_zlim([-grid_size, grid_size])

# Show the plot
plt.show()
```

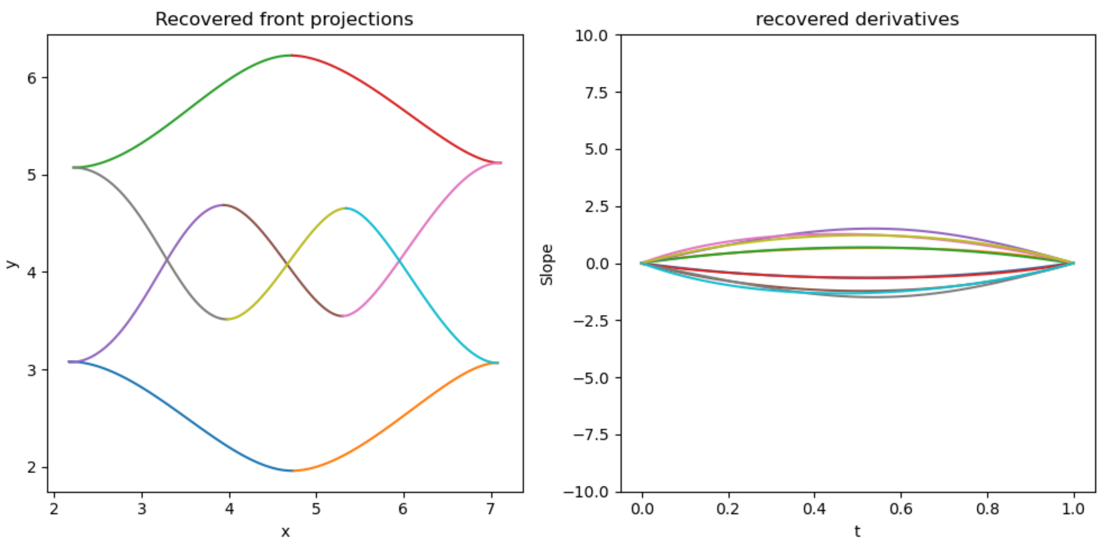


[ ]:
```
# Function to compute Bezier curve points
def bezier_curve(t, start, *controls, end):
    n = len(controls) + 1
    return sum(math.comb(n, k) * (1-t)**(n-k) * t**k * point for k, point in␣
 ↪enumerate([start] + list(controls) + [end]))

def bezier_derivative(t, start, *controls, end):
    if len(controls) != 2:
        raise ValueError("Two control points are required for a cubic Bézier␣
 ↪curve")

    P0, P1, P2, P3 = start, controls[0], controls[1], end

    # Derivative of cubic Bézier curve
    dP0 = -3 * (1 - t)**2 * P0
    dP1 = 3 * (1 - t)**2 * P1
    dP2 = -6 * t * (1 - t) * P1 + 6 * (1 - t) * t * P2
    dP3 = -3 * t**2 * P2 + 3 * t**2 * P3

    derivative = dP0 + dP1 + dP2 + dP3
```

```
    # Return the real and imaginary parts as the gradient components
    return  derivative


# Function to calculate the slope
def calculate_slope(derivative):
    return np.inf if np.isclose(derivative, 0) else derivative.imag /␣
 ↪derivative.real

# Function to process paths and return curve and derivative points
def process_paths(svg_file):
    paths, attributes = svg2paths(svg_file)
    curve_data = []
    for path in paths:
        for segment in path:

            t_values = np.linspace(0, 1, 100)
            if isinstance(segment, CubicBezier):
                #print("Cubic with ",segment)
                controls = segment.control1, segment.control2
            elif isinstance(segment, Line):
                #print("Line with", segment)
                controls = ()
            else:
                print("Error. Script can only handle linear and cubic Beziers")
            curve_points = np.array([bezier_curve(t, segment.start, *controls,␣
 ↪end=segment.end) for t in t_values])
            if controls:  # If there are control points, calculate derivative
                derivative_points = np.array([bezier_derivative(t, segment.
 ↪start, *controls, end=segment.end) for t in t_values])
            else:  # For lines, derivative is constant
                derivative = segment.end - segment.start
                derivative_points = np.array([derivative] * len(t_values))
            curve_data.append((curve_points, derivative_points))
    return curve_data
```

[ ]:
```
# Load and process the SVG file
svg_file = '/Users/slaus/bitmap1.svg'  # Replace with your SVG file path
curve_data = process_paths(svg_file)

# Plotting
fig = plt.figure(figsize=(10, 5))

# First subplot for the curve
plt.subplot(1, 2, 1)
for curve_points, _ in curve_data:
```

```
    plt.plot(curve_points.real, curve_points.imag)

plt.xlabel('x')
plt.ylabel('y')
plt.title('Recovered front projections')

# Second subplot for the slope
plt.subplot(1, 2, 2)
for _, derivative_points in curve_data:
    t_values = np.linspace(0, 1, 100)
    slope_values = [calculate_slope(derivative) for derivative in␣
 ↪derivative_points]
    plt.plot(t_values, slope_values)

plt.xlabel('t')
plt.ylabel('Slope')
plt.title('recovered derivatives')
plt.ylim(-10, 10)

plt.tight_layout()
plt.show()
```




```
[ ]: # 3D Plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for curve_points, derivative_points in curve_data:
    # Calculating -dz/dy
```

```
    y = curve_points.real
    z = curve_points.imag
    # Calculating -dz/dy
    x = [-calculate_slope(derivative) for derivative in derivative_points]  # x component is -dz/dy

    ax.plot(x, curve_points.real, curve_points.imag)
    x = [x[int(i * len(x) / 10)] for i in range(10)]
    y = [y[int(i * len(y) / 10)] for i in range(10)]
    z = [z[int(i * len(z) / 10)] for i in range(10)]
    for xi, yi, zi in zip(x, y, z):
        plot_plane(ax, xi, yi, zi, form, surfcolor='red')

ax.view_init(elev=15, azim=28)
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
ax.set_zlabel('Z Axis')
ax.set_title('Recovered Legendre knot')

plt.show()
```