Motion Control in Computer-Aided Modeling

Reality Check 5

Jenaveve White

Reality Check

Jenaveve White

Introduction

Computer-aided modeling and manufacturing play a crucial role in modern engineering and design, requiring precise spatial control along predefined motion paths. This project explores how Adaptive Quadrature can be applied to solve a fundamental problem within this field: the equipartition of an arbitrary path into subpaths of equal length. Maintaining constant speed along a path is essential for various applications, such as numerical machining, where equal increments ensure consistent material interaction, or motion planning in computer animation, where more complex velocity profiles may be necessary. In robotics and virtual reality, parametrized curves and surfaces are often needed to ensure smooth navigation. The project includes developing Python programs to compute arc lengths, determine parameters for path segmentation using numerical methods like the Bisection Method and Newton's Method, create visualizations for path equipartition, and demonstrate these concepts through Python animations. This practical approach will reinforce understanding of adaptive numerical techniques and their applications in engineering and computational modeling.

The parameteric curve that will be modeled is

$$P(t) = \begin{cases} x(t) = 0.5 + 0.3t + 3.9t^2 - 4.7t^3 \\ y(t) = 1.5 + 0.3t + 0.9t^2 - 2.7t^3 \end{cases}$$

Activity 1

Write a Python function that computes the arc length of the path P from t = 0 to t = s for a given 0 s 1.

Procedure

In this code, a procedure was developed to compute the arc length of a parametric path defined by polynomial functions ( ) and ( ). First, the functions dx_dt(t) and dy_dt(t) were implemented to calculate the derivatives of and with respect to , representing the rate of change along the path. The magnitude of the velocity vector at each point , computed using,

$$\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

was defined in the integrand() function. To find the total arc length from a start point of 0 to a given endpoint e, the quad() function from scipy.integrate was used to numerically integrate the integrand over the specified interval. The resulting arc length is displayed below. This represents the distance traveled along the path from =0 to =1

```
import numpy as np
import matplotlib as mlt
import plotly.express as px
from scipy.integrate import quad
import plotly.graph_objects as go
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def P(t):
    x = 0.5 + 0.3 * t + 3.9 * (t ** 2) - 4.7 * (t ** 3)
    y = 1.5 + 0.3 * t + 0.9 * (t ** 2) - 2.7 * (t ** 3)
    return x,y

def derP(t):
    x = 0.3 + 2 * 3.9 * t - 3 * 4.7 * (t ** 2)
    y = 0.3 + 2 * 0.9 * t - 3 * 2.7 * (t ** 2)
    return x,y
# Derivatives of x and y with respect to t
def dx_dt(t):
    return 0.3 + 2 * 3.9 * t - 3 * 4.7 * (t ** 2)

def dy_dt(t):
    return 0.3 + 2 * 0.9 * t - 3 * 2.7 * (t ** 2)

# Integrand for the arc length
def integrand(t):
    return np.sqrt(dx_dt(t)**2 + dy_dt(t)**2)
```

```
def arc_length(e):
    # Calculate the arc length from 0 to s
    length, _ = quad(integrand, 0, e)
    return length

# s is the start of t and e is the end of t
e = 1

length = arc_length(e)
print(f"The arc length from 0 to t={e} is: {length}")
```

```
The arc length from 0 to t=1 is: 2.495246747514834
```

Activity 2

Write a program that, for any input 0 s 1, finds the parameter t*(s) that is s of the way along the path. Use the Bisection Method to locate the point t*(s) to three correct decimal places. What function is being set to zero? What bracketing interval should be used to start the Bisection Method?

Procedure

The task involves dividing a parametric path ( ), defined by polynomial functions ( ) and ( ), into subpaths of equal length. The code begins by calculating the arc length of the path using numerical integration, with the function arc_length(e) integrating the magnitude of the velocity vector.

$$\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

To find the specific parameter ∗( ) such that the path length from =0 to ∗( ) equals times the total path length, the Bisection Method is applied through the bisect() function. The function iteratively locates the root of an objective function that represents the difference between the arc length up to ∗ and the target length. The function being set to zero is the arc length from 1 to the t* value - s * the total length. For the bracketing interval to start the Bisection Method [0,1] was used to ensure that f(b) * f(a) < 0.

```
#Bisection Method
def bisect(f, a, b, tol, s):
    fa = f(a,s)
    fb = f(b,s)
    n = 0
    if np.sign(fa*fb) >= 0:
        print('f(a)f(b) < 0 not satisfied!')
        quit()
    while (b-a)/2. > tol:
        n = n + 1
        c = (a+b)/2.
        fc = f(c,s)
        if fc == 0:
            return c
        if np.sign(fc*fa) < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    return [(a+b)/2., n]

def obj(tstar,s):
    return arc_length(tstar) - s * arc_length(1)

value, n = bisect(obj, 0, 1, .0005, 1/2)

#print(arc_length(.8003)/arc_length(1))
#
#print(value, arc_length(value)/ arc_length(1))

# Using bisect to find the root
#root = bisect(func1, 0, 1, 0.0005)
#print("Root found:", root)
```

Activty 3

Equipartition the path of Figure 5.6 into n subpaths of equal length, for n=4 and n=20. Plot analogues of Figure 5.6, showing the equipartitions.

Procedure

The equipar(n) function uses this bisection method to find $*$ values for different segments, dividing the path into equal subpaths. Finally, the plot_equipartitions(n, func) function visualizes the path and its partition points using Plotly, showing the uniform segmentation. This approach facilitates applications like CNC machining, where consistent tool movement is crucial, and animation, where equal path segments ensure natural motion.

```
def equipar(n):
    partition_points = [0]  # Start at t=0
    total_length = arc_length(1)
    segment_length = total_length / n
    for i in range(1, n):
        s = i / n
        t, _ = bisect(obj, 0, 1, 0.0005, s)
        partition_points.append(t)
    partition_points.append(1)  # End at t=1
    return np.array(partition_points)

def plot_equipartitions(n, func):
    t_vals = np.linspace(0, 1, 1000)
    x_vals, y_vals = func(t_vals)
    partition_points = equipar(n)
    partition_coords = [func(t) for t in partition_points]

    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_vals, y=y_vals, mode='lines'))
    for coord in partition_coords:
        fig.add_trace(go.Scatter(x=[coord[0]], y=[coord[1]], mode='markers'))
    fig.update_layout(title=f'Equipartition of Path into {n} Subpaths',showlegend=False, wid
    fig.update_xaxes(range=[-0.1, 1.8])
    fig.update_yaxes(range=[-0.1, 1.8])
    fig.show()

plot_equipartitions(4, P)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

```
plot_equipartitions(20, P)
#equipar(4)
```

```
Unable to display output for mime type(s): text/html
```

Activity 4

Replace the Bisection Method in Step 2 with Newton's Method, and repeat Steps 2 and 3. What is the derivative needed? What is a good choice for the initial guess? Is computation time decreased by this replacement?

Procedure

To solve the problem of finding the parameter t*(s) such that the arc length from t = 0 to t*(s) = s * the total path length, Newton's Method was used as an alternative to the Bisection Method. Newton's Method iteratively refines an initial guess x0 using the update formula x(n+1) equals x(n) minus f(x(n)) divided by f'(x(n)). For this problem, the function f(t*) equals arc_length(t)* - s * arc_length(1) was set to zero, where the derivative f'(t*) is the magnitude of the derivative vector, computed as:

$$\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

A good initial guess ensures convergence. A good choice for this initial guess is 0.5. Newton's Method reduces computation time compared to the Bisection Method because it converges more quickly when starting near the root, but it requires explicit computation of the derivative f'(t). The code provided uses Newton's Method to find partition points for an arbitrary path, which are then visualized using Python's plotly library to show path segmentation into equal-length subpaths.

```
def newton(f, fp, x0, k):
    xc = x0
    for i in range(1, k):
        xc = xc - f(xc)/fp(xc)
    return xc

def obj(tstar,s):
    return arc_length(tstar) - s * arc_length(1)

## What is the derivative needed to use Newton's Method?
```

```
def equipar_newton(n):
    partition_points = [0]
    total_length = arc_length(1)
    segment_length = total_length / n

    for i in range(1, n):
        s = i / n
        t_guess = 0.5
        # Use Newton's method to find the value of t for the given s
        t = newton(lambda t: obj(t, s), lambda t: np.sqrt(dx_dt(t)**2 + dy_dt(t)**2), t_guess
        partition_points.append(t)
    partition_points.append(1)
    return partition_points
#print(equipar_newton(4))
def plot_newequipartitions(n, func):
    t_vals = np.linspace(0, 1, 1000)
    x_vals, y_vals = func(t_vals)
    partition_points = equipar_newton(n)
    partition_coords = [func(t) for t in partition_points]

    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_vals, y=y_vals, mode='lines'))
    for coord in partition_coords:
        fig.add_trace(go.Scatter(x=[coord[0]], y=[coord[1]], mode='markers'))
    fig.update_layout(title=f'Equipartition of Path into {n} Subpaths',showlegend=False, widt
    fig.update_xaxes(range=[-0.1, 1.8])
    fig.update_yaxes(range=[-0.1, 1.8])
    fig.show()

plot_newequipartitions(4, P)
plot_newequipartitions(20, P)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Activity 5

Use Python animation commands to demonstrate traveling along the path, first at the original parameter 0 t 1 speed and then at the (constant) speed given by t*(s)for 0 s 1

Procedure

To animate traveling along a parametric path in Python at different speeds, we first prepare data arrays for path coordinates. This involves computing coordinates using functions like ( ) and ( ) and generating points based on uniform parameter values for original speed, as well as adaptive partitions ∗( )for constant-speed traversal. Using Matplotlib, the figure was created with subplots to display animations side by side, initialized with Line2D objects for movement representation. An update() function is defined to update plot data at each frame, and animation.FuncAnimation() is configured to animate the traversal by looping through frames of the data array. The animation showcases both traversal modes, enabling comparison and visualization of the path movement, helping to illustrate concepts in adaptive numerical methods and path segmentation.

```
#
#x4, y4 = P(equipar(4))
#x20, y20 = P(equipar(20))
#
#data4 = np.vstack([x4,y4])
#data20 = np.vstack([x20,y20])
#
##altdata4 = P(np.linspace(0,1,5))
#altdata4 = np.vstack(P(np.linspace(0,1,5)))
##print(altdata4)
#
#altdata20 = np.vstack(P(np.linspace(0,1,21)))
##print(altdata20)
#
#
## Create the figure, axis, and plot
#fig, ax = plt.subplots(ncols=2)
#
#left, = ax[0].plot([], [], 'r:o')
#right, = ax[1].plot([], [], 'r:o')
#
#for axis in ax:
#    axis.set_xlim(-0.1, 2)
#    axis.set_ylim(-0.1, 2)
#    axis.set_xlabel('x')
#    axis.set_title('test')
#
##update = lambda fnum: left.set_data(data20[..., :fnum]); right.set_data(altdata20[..., :fnu
#
#def update(fnum):
```

```
#    left.set_data(data20[..., :fnum])
#    right.set_data(altdata20[..., :fnum])
#    return left, right
#
## run a frame for every column
#r,c = np.shape(data20)
#
## Set up the animation
#ani = animation.FuncAnimation(fig, update, frames=c+1, interval=500)
#plt.show()
```

Activity 6

Experiment with equipartitioning a path of your choice. Choose a path defined by parametric equations (see Parametric Equation in Wikipedia for ideas), partition it into equal arc length segments, and animate as in Step 5

Procedure

To explore equipartitioning, a path is defined using parametric equations, specifically a circle parameterized by ( )=cos( ) and ( )=sin( ). The process begins by computing the path's arc length through numerical integration, implemented in the arc_length2() function. To partition the path into equal arc length segments, the equipar2() function applies the Bisection Method to find parameter values ∗( ) such that the arc length from =0 to ∗( ) matches the target segment length. The plot_equipartitions2() function visualizes the segmented path by plotting the path and marking the partition points. For the animation, a data array of coordinates along the path is created, and a plot is initialized with Matplotlib. An update() function is defined to animate the traversal frame-by-frame using animation.FuncAnimation(). This demonstrates the visual transition along the path with constant-speed traversal, highlighting the effectiveness of numerical methods in engineering visualization.

```
def newpath(t):
    x = np.cos(t)
    y = np.sin(t)
    return x,y

def dx_dt2(t):
    return -np.sin(t)

def dy_dt2(t):
```

```
    return np.cos(t)

# Integrand for the arc length
def integrand2(t):
    return np.sqrt(dx_dt2(t)**2 + dy_dt2(t)**2)

def arc_length2(e):
    # Calculate the arc length from 0 to s
    length, _ = quad(integrand2, 0, e)
    return length


def obj2(tstar,s):
    return arc_length2(tstar) - s * 6.283

def equipar2(n):
    partition_points = [0]  # Start at t=0
    total_length = arc_length2(2 * np.pi)
    #print(total_length)
    segment_length = total_length / n
    for i in range(1, n):
        s = i / n
        t, _ = bisect(obj2, 0, 2 * np.pi, 0.0005, s)
        partition_points.append(t)
    #partition_points.append(1)  # End at t=1
    return np.array(partition_points)

def plot_equipartitions2(n, func):
    t_vals = np.linspace(0, 2 * np.pi , 1000)
    x_vals, y_vals = func(t_vals)
    partition_points = equipar2(n)
    partition_coords = [func(t) for t in partition_points]

    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_vals, y=y_vals, mode='lines'))
    for coord in partition_coords:
        fig.add_trace(go.Scatter(x=[coord[0]], y=[coord[1]], mode='markers'))
    fig.update_xaxes(range=[-1.5, 1.5])
    fig.update_yaxes(range=[-1.5, 1.5])
    fig.update_layout(title=f'Equipartition of Path into {n} Subpaths',showlegend=False, wid
    fig.show()
```

```
plot_equipartitions2(20, newpath)

## ANIMATION PLOT
#
#x20, y20 = P(equipar2(20))
#
#data20 = np.vstack([x20,y20])
#
#
## Create the figure, axis, and plot
#fig, ax = plt.subplots(ncols=1)
#
#left, = ax[0].plot([], [], 'r:o')
#
#for axis in ax:
#    axis.set_xlim(-0.1, 2)
#    axis.set_ylim(-0.1, 2)
#    axis.set_xlabel('x')
#    axis.set_title('test')
#
##update = lambda fnum: left.set_data(data20[..., :fnum]); right.set_data(altdata20[..., :fnu
#
#def update(fnum):
#    left.set_data(data20[..., :fnum])
#    return left, right
#
## run a frame for every column
#r,c = np.shape(data20)
#
## Set up the animation
#ani = animation.FuncAnimation(fig, update, frames=c+1, interval=500)
#plt.show()
```

```
Unable to display output for mime type(s): text/html
```