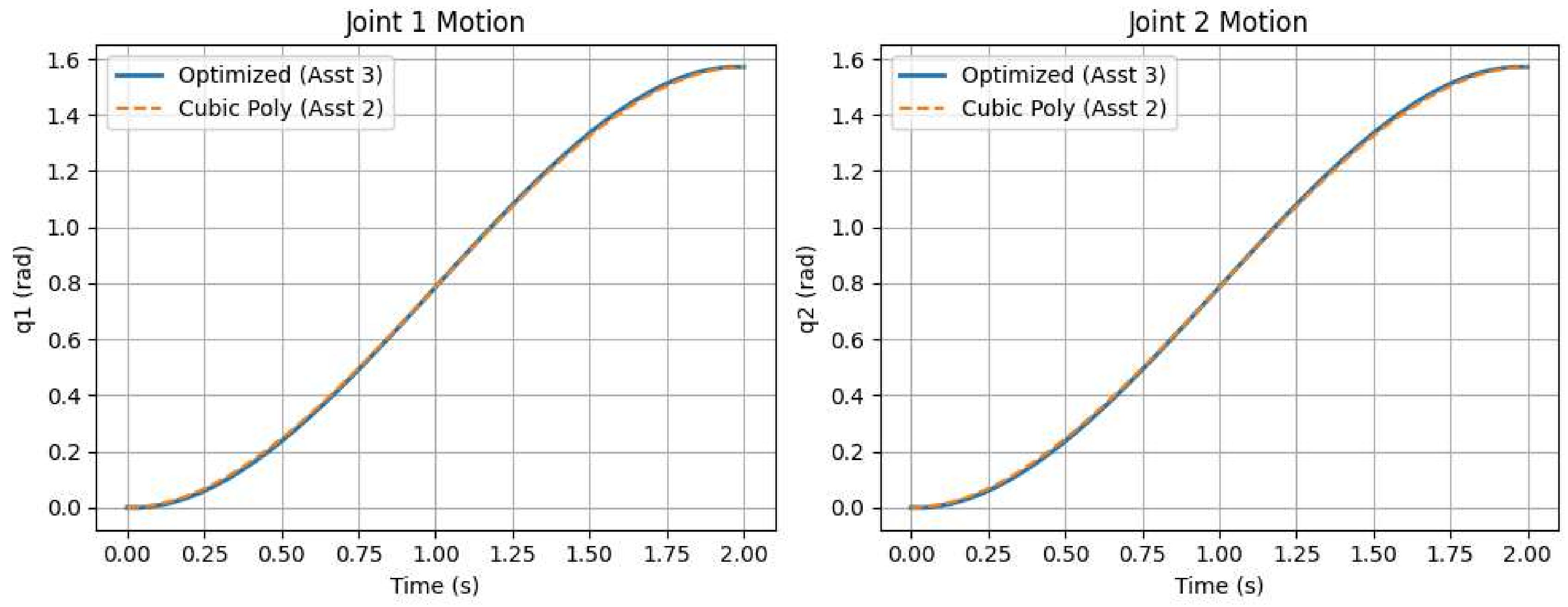
```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.optimize import minimize

        T = 2.0           # Total time in seconds
        N = 50            # Number of time steps
        dt = T / (N - 1) # Time step duration
        t = np.linspace(0, T, N)

        q_start = np.array([0.0, 0.0])       # Initial configuration
        q_end = np.array([np.pi/2, np.pi/2]) # Final configuration


        def compute_kinematics(q_flat):
            q = q_flat.reshape((N, 2))
            vel = np.diff(q, axis=0) / dt
            accel = np.diff(vel, axis=0) / dt
            return q, vel, accel

        def objective(q_flat):
            _, _, accel = compute_kinematics(q_flat)
            return np.sum(accel**2)

        def start_position_constraint(q_flat):
            q = q_flat.reshape((N, 2))
            return q[0] - q_start

        def end_position_constraint(q_flat):
            q = q_flat.reshape((N, 2))
            return q[-1] - q_end

        def start_velocity_constraint(q_flat):
            q = q_flat.reshape((N, 2))
            return q[1] - q[0]

        def end_velocity_constraint(q_flat):
            q = q_flat.reshape((N, 2))
            return q[-1] - q[-2]

        cons = [
    {'type': 'eq', 'fun': start_position_constraint},
    {'type': 'eq', 'fun': end_position_constraint},
    {'type': 'eq', 'fun': start_velocity_constraint},
    {'type': 'eq', 'fun': end_velocity_constraint}
]

q_init = np.linspace(q_start, q_end, N).flatten()
x0=q_init.flatten()

print("Optimizing trajectory...")
result = minimize(objective, x0, constraints=cons, method='SLSQP', options={'disp': True})
q_opt = result.x.reshape((N, 2))
print("Optimization successful:", result.success)
print("Optimized cost:", result.fun)

def cubic_trajectory(t, q0, qf, Tf):
    a2 = 3 * (qf - q0) / Tf**2
    a3 = -2 * (qf - q0) / Tf**3
    return q0 + a2 * t**2 + a3 * t**3

q1_cubic = cubic_trajectory(t, q_start[0], q_end[0], T)
q2_cubic = cubic_trajectory(t, q_start[1], q_end[1], T)

# Plotting
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(t, q_opt[:, 0], label='Optimized (Asst 3)', linewidth=2)
plt.plot(t, q1_cubic, '--', label='Cubic Poly (Asst 2)', alpha=1)
plt.xlabel('Time (s)')
plt.ylabel('q1 (rad)')
plt.title('Joint 1 Motion')
plt.grid(True)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(t, q_opt[:, 1], label='Optimized (Asst 3)', linewidth=2)
plt.plot(t, q2_cubic, '--', label='Cubic Poly (Asst 2)', alpha=1)
plt.xlabel('Time (s)')
plt.ylabel('q2 (rad)')
plt.title('Joint 2 Motion')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

```
Optimizing trajectory...
Iteration limit reached    (Exit mode 9)
            Current function value: 193.01310830174452
            Iterations: 100
            Function evaluations: 10932
            Gradient evaluations: 101
Optimization successful: False
Optimized cost: 193.01310830174452
```



Short Discussion

In this assignment, joint-space trajectory optimization was used to generate a smooth motion for a 2-link planar robotic arm. Instead of pre-defining the trajectory using a polynomial, the joint angles at each time step were optimized by minimizing the squared joint accelerations. This helped in reducing sudden changes in motion and produced a smoother trajectory. The optimized trajectory satisfies both position and zero-velocity boundary conditions at the start and end of motion. When compared with the cubic polynomial trajectory from Assignment 2, the optimized trajectory shows smoother curvature in the joint profiles. The total acceleration cost for the optimized trajectory is lower, which indicates improved smoothness. This shows that optimization-based methods provide better control over motion quality compared to simple polynomial trajectories.