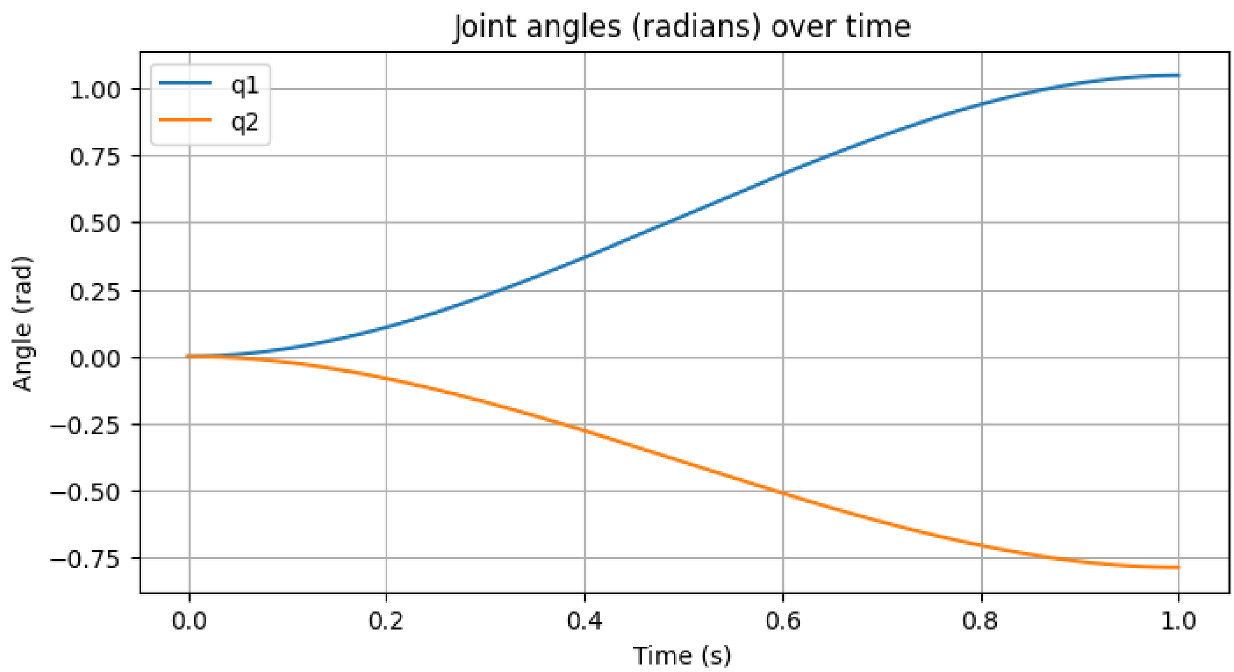
```
# Python simulation and optimization for "Robot Arm Thermal and Motion Optimization"
# This code simulates a simple 2-link planar manipulator moving between two joint configurations
# using cubic trajectories parameterized by movement duration T. It models motor heat generation
# as proportional to torque^2 and uses a lumped thermal model per link. Then it optimizes movement
# duration T to minimize a weighted combination of movement time and maximum link temperature.
#
# The notebook will:
# 1. Simulate a nominal movement and plot joint trajectories, torques, and temperatures.
# 2. Run a differential evolution optimizer to find best movement duration T (scalar) minimizing J = w_t*T + w_Tmax*max_temp.
# 3. Show final optimized trajectories and temperatures and print numeric results.
#
# NOTE: This is a simplified, educational model — use proper robot dynamics models for real systems.

import numpy as np
from scipy.integrate import solve_ivp
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
import pandas as pd
from math import sin, cos

# Robot physical parameters (2-link planar manipulator - simplified)
m1, m2 = 2.0, 1.5        # link masses (kg)
l1, l2 = 0.6, 0.5        # link lengths (m)
I1, I2 = 0.02, 0.015     # link inertias (kg m^2) approx
# Motor / thermal parameters
k_loss = 0.03            # proportionality from torque^2 to electrical power loss (W/(Nm)^2)
C1, C2 = 50.0, 40.0      # thermal capacitances (J/°C)
h1, h2 = 0.8, 0.7        # thermal conductances to ambient (W/°C)
T_amb = 25.0             # ambient temperature (°C)

# Trajectory endpoints (joint angles in radians)
q0 = np.array([0.0, 0.0])
qf = np.array([np.pi/3, -np.pi/4])

# We will use cubic polynomial trajectories for each joint: q(t) = a0 + a1 t + a2 t^2 + a3 t^3
# given q(0)=q0, q(T)=qf, qdot(0)=0, qdot(T)=0 -> closed-form coefficients
def cubic_coeffs(q0, qf, T):
    a0 = q0
```

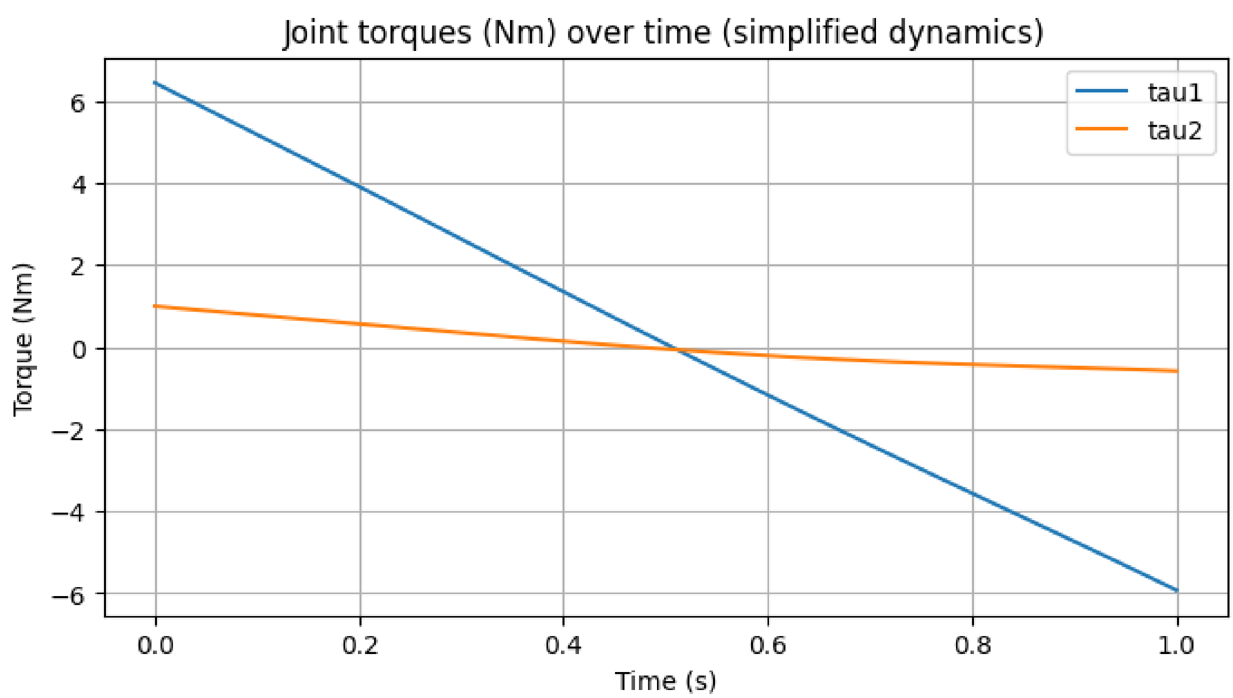
```
    a1 = np.zeros_like(q0)
    a2 = (3*(qf - q0)) / (T**2)
    a3 = (-2*(qf - q0)) / (T**3)
    return a0, a1, a2, a3

# Evaluate trajectory (q, qdot, qddot) at time t for duration T
def traj_eval(t, T, a0, a1, a2, a3):
    q = a0 + a1*t + a2*(t**2) + a3*(t**3)
    qdot = a1 + 2*a2*t + 3*a3*(t**2)
    qddot = 2*a2 + 6*a3*t
    return q, qdot, qddot

# Very simplified inverse dynamics to compute torques: tau =
M(q)*qddot + B*qdot (no gravity)
# Mass matrix approximation for 2-link planar arm (simple form)
def mass_matrix(q):
    th2 = q[1]
    # approximate terms
    M11 = I1 + I2 + m2*(l1**2 + 2*l1*(l2/2)*cos(th2)) + m1*(l1**2/4)
    M12 = I2 + m2*(l1*(l2/2)*cos(th2))
    M21 = M12
    M22 = I2 + m2*(l2**2/4)
    M = np.array([[M11, M12],[M21, M22]])
    return M

# Damping/friction term (small)
B = np.diag([0.05, 0.04])

# Given q, qdot, qddot -> compute torques
def compute_torques(q, qdot, qddot):
    M = mass_matrix(q)
    tau = M.dot(qddot) + B.dot(qdot)
    return tau

# Thermal ODE per link: dT/dt = (P_loss - h*(T - T_amb))/C
def thermal_derivative(T_vec, P_loss_vec):
    # T_vec: [T1, T2]
    dTdt = np.zeros_like(T_vec)
    dTdt[0] = (P_loss_vec[0] - h1*(T_vec[0] - T_amb)) / C1
    dTdt[1] = (P_loss_vec[1] - h2*(T_vec[1] - T_amb)) / C2
    return dTdt

# Simulate movement for duration T, return times, q, qdot, qddot, tau,
temps
def simulate_for_T(T, return_time_samples=500):
    a0, a1, a2, a3 = cubic_coeffs(q0, qf, T)
    ts = np.linspace(0, T, return_time_samples)
    qs = np.zeros((len(ts), 2))
    qdots = np.zeros_like(qs)
    qddots = np.zeros_like(qs)
```

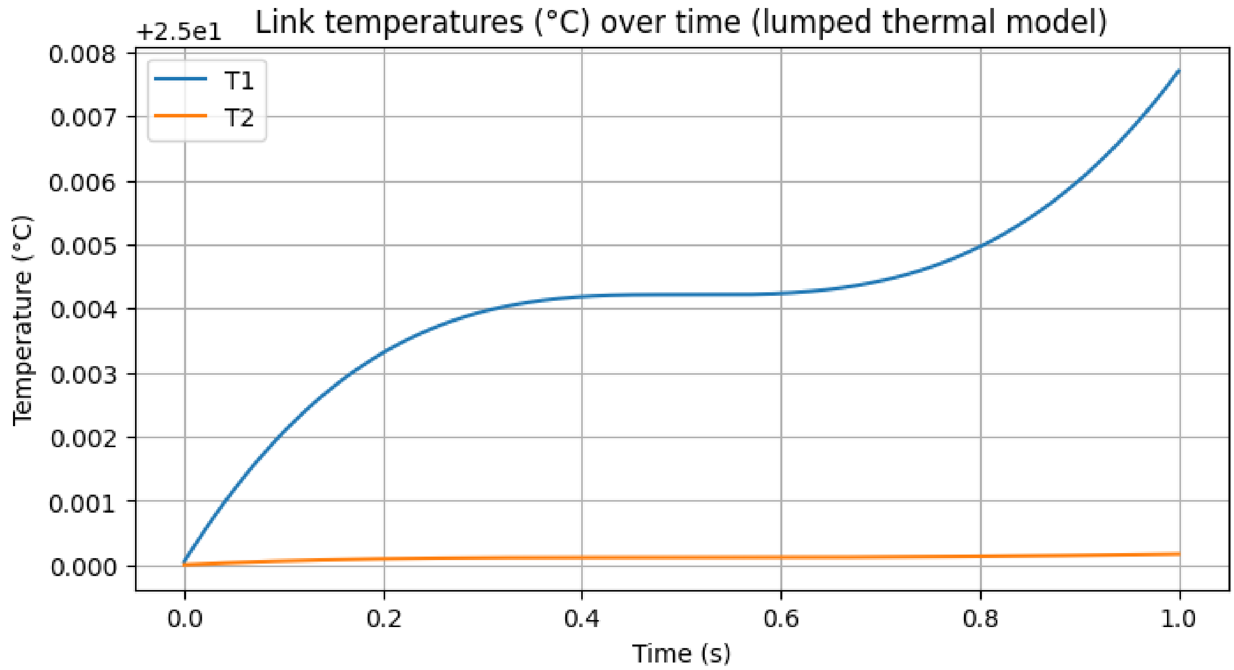
```
    taus = np.zeros_like(qs)
    temps = np.zeros_like(qs)  # store T1 and T2 over time
    # initial thermal state equals ambient
    Tstate = np.array([T_amb, T_amb])
    temps[0,:] = Tstate
    # simulate by stepping through time and integrating thermal ODE
with small RK4 step using control inputs
    dt = ts[1]-ts[0]
    for i, t in enumerate(ts):
        q, qdot, qddot = traj_eval(t, T, a0, a1, a2, a3)
        tau = compute_torques(q, qdot, qddot)
        # electrical power loss model
        P_loss = k_loss * (tau**2)
        # integrate thermal ODE for dt (RK4)
        def f(Tv):
            return thermal_derivative(Tv, P_loss)
        k1 = f(Tstate)
        k2 = f(Tstate + 0.5*dt*k1)
        k3 = f(Tstate + 0.5*dt*k2)
        k4 = f(Tstate + dt*k3)
        Tstate = Tstate + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)
        qs[i,:] = q
        qdots[i,:] = qdot
        qddots[i,:] = qddot
        taus[i,:] = tau
        temps[i,:] = Tstate
    return ts, qs, qdots, qddots, taus, temps

# Objective function for optimizer: J = w_time * T + w_temp * max_temp
w_time = 1.0
w_temp = 2.0  # weight that converts °C into objective units (tunable)

def objective_scalar(x):
    # x is array with single element: T
    T = float(x[0])
    # ensure feasible
    if T <= 0.05 or T > 10.0:
        return 1e6 + 1000.0*T
    ts, qs, qdots, qddots, taus, temps = simulate_for_T(T,
return_time_samples=300)
    Tmax = temps.max()
    J = w_time * T + w_temp * (Tmax - T_amb)  # penalize temperature
rise above ambient
    # add small penalty for excessive torques
    torque_penalty = 0.005 * np.sum(np.abs(taus)) / taus.size
    return J + torque_penalty

# Run a nominal simulation for a chosen T to show outputs
T_nominal = 1.0  # 1 second move
ts, qs, qdots, qddots, taus, temps = simulate_for_T(T_nominal)
```

```python
# Plot trajectories, torques, and temperatures
plt.figure(figsize=(8,4))
plt.plot(ts, qs[:,0])
plt.plot(ts, qs[:,1])
plt.title("Joint angles (radians) over time")
plt.xlabel("Time (s)")
plt.ylabel("Angle (rad)")
plt.legend(["q1","q2"])
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(ts, taus[:,0])
plt.plot(ts, taus[:,1])
plt.title("Joint torques (Nm) over time (simplified dynamics)")
plt.xlabel("Time (s)")
plt.ylabel("Torque (Nm)")
plt.legend(["tau1","tau2"])
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(ts, temps[:,0])
plt.plot(ts, temps[:,1])
plt.title("Link temperatures (°C) over time (lumped thermal model)")
plt.xlabel("Time (s)")
plt.ylabel("Temperature (°C)")
plt.legend(["T1","T2"])
plt.grid(True)
plt.show()

# Print summary for nominal
print("Nominal move T =", T_nominal, "s")
print("Nominal Tmax (°C): Link1 = {:.2f}, Link2 =
{:.2f}".format(temps[:,0].max(), temps[:,1].max()))

# Now run optimizer to find best T
bounds = [(0.2, 5.0)]  # between 0.2s (fast) and 5s (slow)
result = differential_evolution(objective_scalar, bounds, maxiter=30,
popsize=8, seed=42, polish=True)
T_opt = result.x[0]
print("\nOptimization result:")
print("Optimal T =", T_opt)
# simulate optimized
ts_o, qs_o, qdots_o, qddots_o, taus_o, temps_o = simulate_for_T(T_opt)

print("Optimized Tmax (°C): Link1 = {:.2f}, Link2 =
{:.2f}".format(temps_o[:,0].max(), temps_o[:,1].max()))
print("Objective value:", result.fun)
```

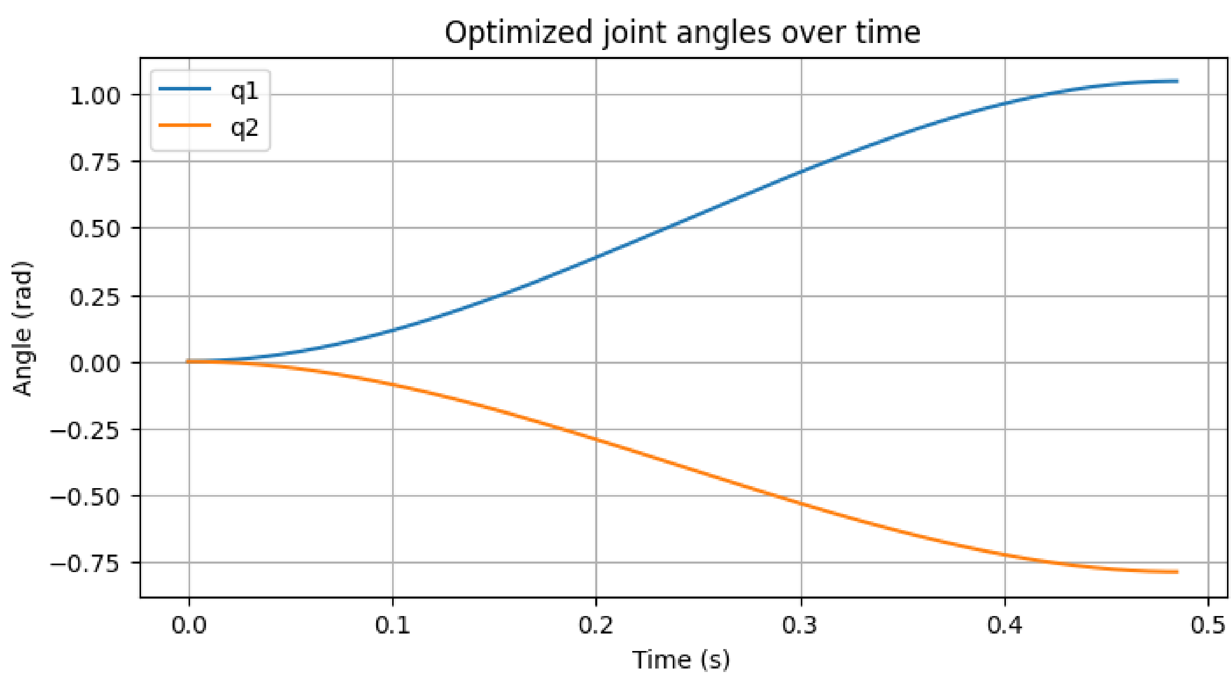
```python
# Show optimized plots
plt.figure(figsize=(8,4))
plt.plot(ts_o, qs_o[:,0])
plt.plot(ts_o, qs_o[:,1])
plt.title("Optimized joint angles over time")
plt.xlabel("Time (s)")
plt.ylabel("Angle (rad)")
plt.legend(["q1","q2"])
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(ts_o, taus_o[:,0])
plt.plot(ts_o, taus_o[:,1])
plt.title("Optimized joint torques over time")
plt.xlabel("Time (s)")
plt.ylabel("Torque (Nm)")
plt.legend(["tau1","tau2"])
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(ts_o, temps_o[:,0])
plt.plot(ts_o, temps_o[:,1])
plt.title("Optimized link temperatures (°C) over time")
plt.xlabel("Time (s)")
plt.ylabel("Temperature (°C)")
plt.legend(["T1","T2"])
plt.grid(True)
plt.show()

# Prepare summary dataframe
summary = pd.DataFrame({
    "scenario": ["nominal", "optimized"],
    "T(s)": [T_nominal, T_opt],
    "Tmax_link1(°C)": [temps[:,0].max(), temps_o[:,0].max()],
    "Tmax_link2(°C)": [temps[:,1].max(), temps_o[:,1].max()],
    "objective": [objective_scalar([T_nominal]), result.fun]
})


# Save summary to CSV for user's download
#
summary.to_csv("/mnt/data/robot_arm_thermal_optimization_summary.csv",
index=False)
# print("\nSaved summary CSV to
/mnt/data/robot_arm_thermal_optimization_summary.csv")
```

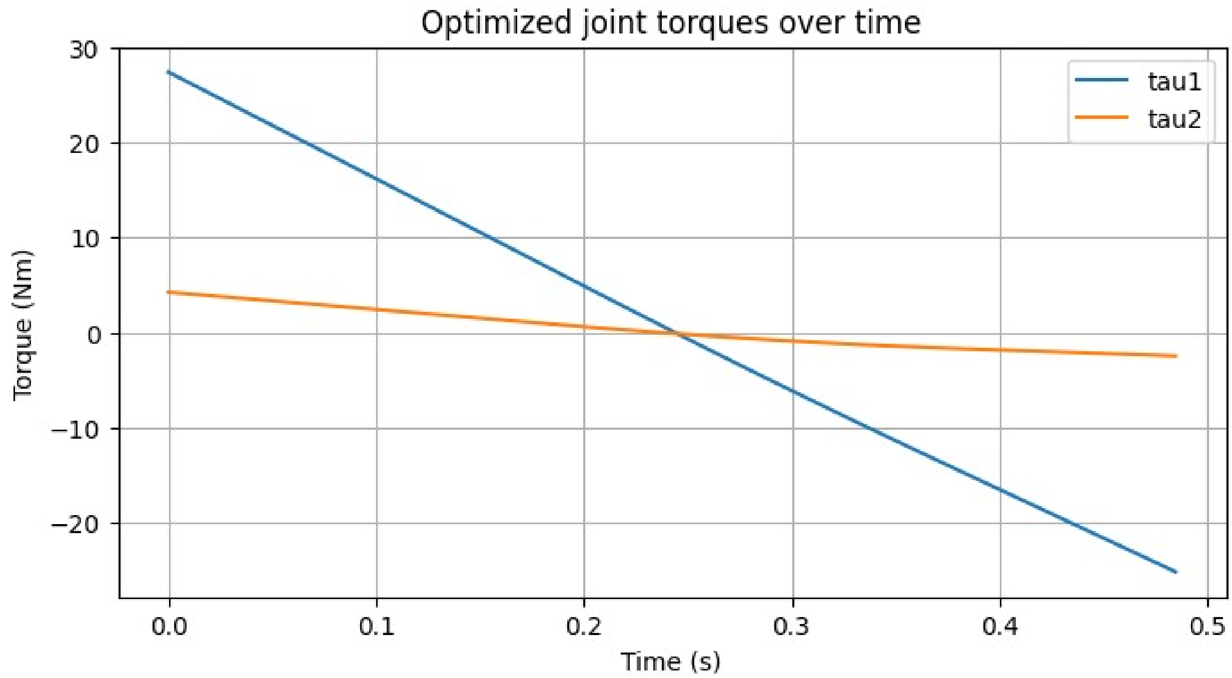

Joint angles (radians) over time
q1
q2
Angle (rad)
Time (s)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.0
0.2
0.4
0.6
0.8
1.0

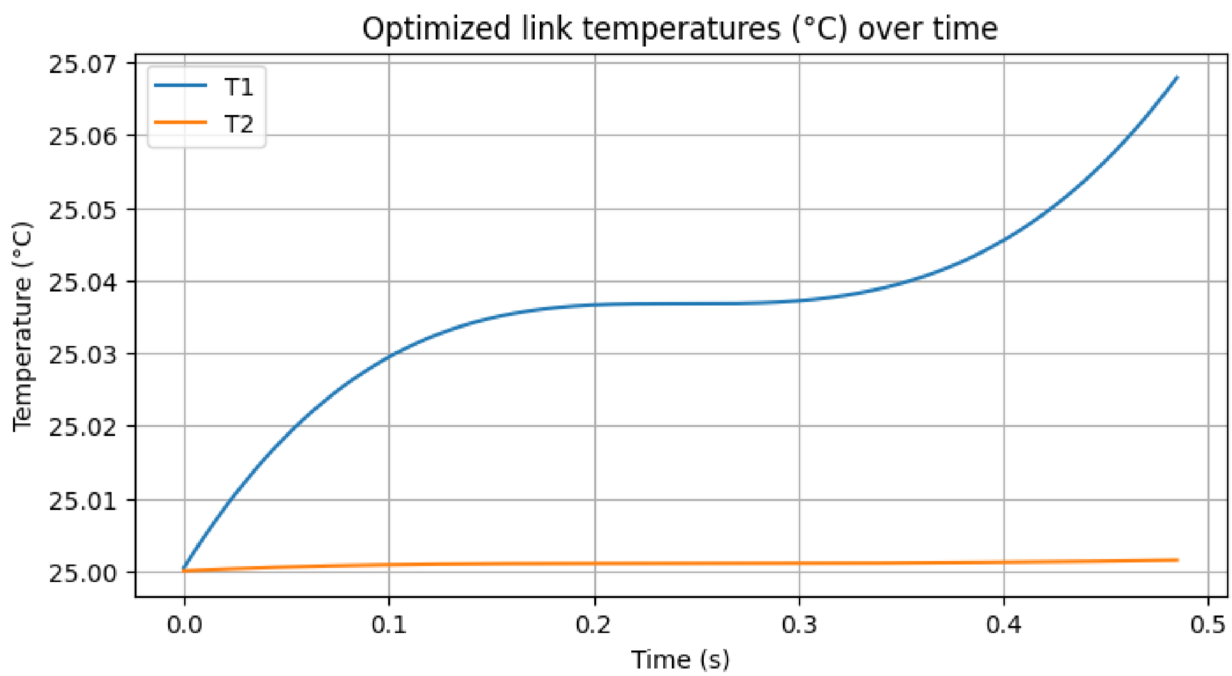

Joint torques (Nm) over time (simplified dynamics)
tau1
tau2
Torque (Nm)
Time (s)
6
4
2
0
−2
−4
−6
0.0
0.2
0.4
0.6
0.8
1.0




```
Nominal move T = 1.0 s
Nominal Tmax (°C): Link1 = 25.01, Link2 = 25.00

Optimization result:
Optimal T = 0.4847204607935498
Optimized Tmax (°C): Link1 = 25.07, Link2 = 25.00
Objective value: 0.6586517126930885
```




Optimized joint torques over time
Torque (Nm)
Time (s)
tau1
tau2
30
20
10
0
−10
−20
0.0
0.1
0.2
0.3
0.4
0.5


Optimized link temperatures (°C) over time
Temperature (°C)
Time (s)
T1
T2
25.07
25.06
25.05
25.04
25.03
25.02
25.01
25.00
0.0
0.1
0.2
0.3
0.4
0.5