```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import golden

# ----------------------------
# Problem definition (1D test)
# ----------------------------
np.random.seed(42)

Omega = (0.0, 1.0)
Gamma = 0.25
Omega1 = (0.0, Gamma)
Omega2 = (Gamma, 1.0)

def beta(x):
    x = np.array(x)
    return np.where(x < Gamma, 1.0, 10.0)

def u_exact(x):
    x = np.array(x)
    return np.where(x <= Gamma, np.sin(2*np.pi*x), np.cos(2*np.pi*x))

def du_exact(x):
    x = np.array(x)
    return np.where(x <= Gamma, 2*np.pi*np.cos(2*np.pi*x),
-2*np.pi*np.sin(2*np.pi*x))

# f(x) = -(beta u')' in subdomains where beta constant -> -beta *
u''.
def f_rhs(x):
    x = np.array(x)
    left = -1.0 * (1.0) * (-(2*np.pi)**2 * np.sin(2*np.pi*x))   # =
(2π)^2 sin
    right = -1.0 * (10.0) * (-(2*np.pi)**2 * np.cos(2*np.pi*x)) # =
10*(2π)^2 cos
```

```
    return np.where(x < Gamma, left, right)

# interface jumps h1 and h2
def h1_val():
    return u_exact(Gamma - 1e-12) - u_exact(Gamma + 1e-12)

def h2_val():
    # β_left * u'_left - β_right * u'_right
    du_left = 2*np.pi*np.cos(2*np.pi*Gamma)
    du_right = -2*np.pi*np.sin(2*np.pi*Gamma)
    return 1.0 * du_left - 10.0 * du_right

# boundary g
def g_val(x):
    return u_exact(x)

# ----------------------------
# Multi-TransNet parameters
# ----------------------------
# covering ball centers and radii for each subdomain
x_c1, R1 = 0.125, 0.135
x_c2, R2 = 0.625, 0.385

# neurons per subdomain
M1, M2 = 5, 5

# activation
def sigma(z):
    return np.tanh(z)

# hidden-layer randomized params
a1 = np.random.choice([-1.0, 1.0], size=M1)
a2 = np.random.choice([-1.0, 1.0], size=M2)
r1 = np.random.uniform(0.0, R1, size=M1)
r2 = np.random.uniform(0.0, R2, size=M2)
```

```
# ----------------------------
# Basis and operator builders
# ----------------------------
def build_basis(X, x_c, a, r, gamma, M):
    X = np.atleast_1d(X)
    N = len(X)
    B = np.ones((N, M+1))
    for m in range(M):
        z = gamma * (a[m] * (X - x_c) + r[m])
        B[:, m+1] = sigma(z)
    return B

def build_L_basis(X, x_c, a, r, gamma, M, beta_func):
    """
    Build matrix for L(phi) = - (beta phi')' approximated by -beta *
phi'' (inside subdomains).
    Use centered finite difference for phi'' with small h.
    """
    X = np.atleast_1d(X)
    N = len(X)
    B = np.zeros((N, M+1))
    h = 1e-5
    for m in range(M):
        # evaluate phi at x-h, x, x+h vectorized
        z_c = gamma * (a[m] * (X - x_c) + r[m])
        z_p = gamma * (a[m] * (X + h - x_c) + r[m])
        z_m = gamma * (a[m] * (X - h - x_c) + r[m])
        phi_c = sigma(z_c)
        phi_p = sigma(z_p)
        phi_m = sigma(z_m)
        d2phi = (phi_p - 2.0*phi_c + phi_m) / (h*h)
        B[:, m+1] = - beta_func(X) * d2phi
    # constant column stays zero (second derivative of const = 0)
    return B

def build_J_basis(X, x_c, a, r, gamma, M, beta_func):
```

```
    """
    Build matrix for flux J(phi) = beta * phi' (normal in 1D).
    Use centered finite difference for phi' with small h.
    """
    X = np.atleast_1d(X)
    N = len(X)
    B = np.zeros((N, M+1))
    h = 1e-6
    for m in range(M):
        z_p = gamma * (a[m] * (X + h - x_c) + r[m])
        z_m = gamma * (a[m] * (X - h - x_c) + r[m])
        phi_p = sigma(z_p)
        phi_m = sigma(z_m)
        dphi = (phi_p - phi_m) / (2.0*h)
        B[:, m+1] = beta_func(X) * dphi
    return B

# ----------------------------
# Sampling configuration (modified per your request)
# ----------------------------
# interior collocation points used in compute_loss (large sampling)
N_points = 10000
X1 = np.linspace(Omega1[0] + 0.01, Omega1[1] - 0.01, N_points//2)  # 500
X2 = np.linspace(Omega2[0] + 0.01, Omega2[1] - 0.01, N_points//2)  # 500

# boundary: 100 at x=0 and 100 at x=1 -> total 200 rows
n_bnd_each = 100
Xg_left = np.array([0.0] * n_bnd_each)
Xg_right = np.array([1.0] * n_bnd_each)

# interface: 100 points total, choose 50 left + 50 right near Gamma
n_if_total = 100
n_if_each_side = n_if_total // 2
eps = 1e-3
```

```
    # distribute points moving away from Gamma: small offsets
    left_offsets = np.linspace(eps/n_if_each_side, eps, n_if_each_side)
    right_offsets = np.linspace(eps/n_if_each_side, eps, n_if_each_side)
    Xr_left = Gamma - left_offsets
    Xr_right = Gamma + right_offsets
    Xr = np.concatenate([Xr_left, Xr_right])  # length 100

    # ----------------------------
    # Loss / assembly for given gammas
    # ----------------------------
    def compute_loss(g1):
        # predict g2 from g1 using the relation used in your code pattern
        g2 = g1 * (R1 / R2) * (M2 / M1)

        # build basis and operator matrices
        Phi1 = build_basis(X1, x_c1, a1, r1, g1, M1)
        Phi2 = build_basis(X2, x_c2, a2, r2, g2, M2)
        L1 = build_L_basis(X1, x_c1, a1, r1, g1, M1, beta)
        L2 = build_L_basis(X2, x_c2, a2, r2, g2, M2, beta)

        # boundary basis (many repeated rows)
        Phi_g1 = build_basis(Xg_left, x_c1, a1, r1, g1, M1)   # shape
    (100, M1+1)
        Phi_g2 = build_basis(Xg_right, x_c2, a2, r2, g2, M2)  # shape
    (100, M2+1)

        # interface bases and flux
        Phi_r1 = build_basis(Xr, x_c1, a1, r1, g1, M1)   # (100, M1+1)
        Phi_r2 = build_basis(Xr, x_c2, a2, r2, g2, M2)   # (100, M2+1)
        J_r1 = build_J_basis(Xr, x_c1, a1, r1, g1, M1, beta)
        J_r2 = build_J_basis(Xr, x_c2, a2, r2, g2, M2, beta)

        # count rows
        n1 = len(X1)
        n2 = len(X2)
        nb = len(Phi_g1) + len(Phi_g2)   # 200
```

```
    ni = len(Xr)                    # 100 (each produces 2 equations
=> 200 rows)
    total_rows = n1 + n2 + nb + 2*ni
    total_cols = M1 + M2 + 2  # alpha1..alpha_M1, alpha0_1,
alpha2..alpha_M2, alpha0_2

    F = np.zeros((total_rows, total_cols))
    T = np.zeros(total_rows)

    # PDE residuals Omega1
    F[0:n1, :M1+1] = L1
    T[0:n1] = f_rhs(X1)

    # PDE residuals Omega2
    F[n1:n1+n2, M1+1:] = L2
    T[n1:n1+n2] = f_rhs(X2)

    # boundary: left (x=0) -> fill rows
    idx_b_left_start = n1 + n2
    idx_b_left_end = idx_b_left_start + len(Phi_g1)
    F[idx_b_left_start:idx_b_left_end, :M1+1] = Phi_g1
    T[idx_b_left_start:idx_b_left_end] = g_val(0.0)

    # boundary: right (x=1)
    idx_b_right_start = idx_b_left_end
    idx_b_right_end = idx_b_right_start + len(Phi_g2)
    F[idx_b_right_start:idx_b_right_end, M1+1:] = Phi_g2
    T[idx_b_right_start:idx_b_right_end] = g_val(1.0)

    # interface rows start
    idx_if_start = idx_b_right_end

    # for each interface sampling point, add [u] equation: phi_left -
phi_right = h1
    for i in range(ni):
        row = idx_if_start + i
```

```
        # left phi (M1+1), right phi (M2+1)
        F[row, :M1+1] = Phi_r1[i]
        F[row, M1+1:] = -Phi_r2[i]
        T[row] = h1_val()

    # for each interface sampling point, add flux jump equation:
beta_left phi'_left - beta_right phi'_right = h2
    for i in range(ni):
        row = idx_if_start + ni + i
        F[row, :M1+1] = J_r1[i]
        F[row, M1+1:] = -J_r2[i]
        T[row] = h2_val()

    # solve least squares and return residual norm
    try:
        alpha, residuals, rank, s = np.linalg.lstsq(F, T, rcond=None)
        if len(residuals) > 0:
            return residuals[0]
        else:
            return np.linalg.norm(F @ alpha - T)
    except Exception as e:
        # fallback large penalty if solver fails
        return 1e12

# ----------------------------
# Golden-section search for gamma1
# ----------------------------
print("Starting golden-section search for gamma1 (this may take some
time)...")
gamma1_opt = golden(lambda g: compute_loss(g), brack=(0.1, 10.0),
tol=1e-3, full_output=False)
gamma2_opt = gamma1_opt * (R1 / R2) * (M2 / M1)
print(f"Found gamma1_opt = {gamma1_opt:.6e}, gamma2_opt =
{gamma2_opt:.6e}")

# ----------------------------
```

```
# Final solve using gamma_opt with denser final collocation (kept
moderate)
# ----------------------------
# final collocation inside each subdomain (smaller for speed but can
be increased)
N_final = 1000
X1_final = np.linspace(Omega1[0] + 0.001, Omega1[1] - 0.001,
N_final//2)
X2_final = np.linspace(Omega2[0] + 0.001, Omega2[1] - 0.001,
N_final//2)

# final boundary: still 100 each
Phi_g1_final = build_basis(Xg_left, x_c1, a1, r1, gamma1_opt, M1)
Phi_g2_final = build_basis(Xg_right, x_c2, a2, r2, gamma2_opt, M2)

# final interface points (same Xr used)
Phi_r1_final = build_basis(Xr, x_c1, a1, r1, gamma1_opt, M1)
Phi_r2_final = build_basis(Xr, x_c2, a2, r2, gamma2_opt, M2)
J_r1_final = build_J_basis(Xr, x_c1, a1, r1, gamma1_opt, M1, beta)
J_r2_final = build_J_basis(Xr, x_c2, a2, r2, gamma2_opt, M2, beta)

L1_final = build_L_basis(X1_final, x_c1, a1, r1, gamma1_opt, M1,
beta)
L2_final = build_L_basis(X2_final, x_c2, a2, r2, gamma2_opt, M2,
beta)

# assemble final F and T
n1f = len(X1_final)
n2f = len(X2_final)
nbf = len(Phi_g1_final) + len(Phi_g2_final)  # 200
nif = len(Xr)  # 100
rows_final = n1f + n2f + nbf + 2*nif
cols_final = M1 + M2 + 2

Ff = np.zeros((rows_final, cols_final))
Tf = np.zeros(rows_final)
```

```
# PDE
Ff[0:n1f, :M1+1] = L1_final
Tf[0:n1f] = f_rhs(X1_final)
Ff[n1f:n1f+n2f, M1+1:] = L2_final
Tf[n1f:n1f+n2f] = f_rhs(X2_final)

# boundary
b_left_s, b_right_s = n1f + n2f, n1f + n2f + len(Phi_g1_final)
Ff[b_left_s:b_right_s, :M1+1] = Phi_g1_final
Tf[b_left_s:b_right_s] = g_val(0.0)
Ff[b_right_s:b_right_s+len(Phi_g2_final), M1+1:] = Phi_g2_final
Tf[b_right_s:b_right_s+len(Phi_g2_final)] = g_val(1.0)

# interface rows
start_if = b_right_s + len(Phi_g2_final)
for i in range(nif):
    Ff[start_if + i, :M1+1] = Phi_r1_final[i]
    Ff[start_if + i, M1+1:] = -Phi_r2_final[i]
    Tf[start_if + i] = h1_val()
for i in range(nif):
    Ff[start_if + nif + i, :M1+1] = J_r1_final[i]
    Ff[start_if + nif + i, M1+1:] = -J_r2_final[i]
    Tf[start_if + nif + i] = h2_val()

# solve final least squares
alpha_f, residuals_f, rank_f, s_f = np.linalg.lstsq(Ff, Tf,
rcond=None)
print("Final LS residual (sum squares):", residuals_f if
len(residuals_f)>0 else np.linalg.norm(Ff @ alpha_f - Tf)**2)

# split coefficients
alpha1 = alpha_f[:M1+1]
alpha2 = alpha_f[M1+1:]

# ----------------------------
```

```
# Evaluate the multi-transnet solution on a dense test grid
# ----------------------------
x_test = np.linspace(0.001, 0.999, 1000)
u_pred = np.zeros_like(x_test)
for i, xx in enumerate(x_test):
    if xx <= Gamma:
        B = build_basis([xx], x_c1, a1, r1, gamma1_opt, M1)  # shape
(1, M1+1)
        u_pred[i] = B.dot(alpha1)
    else:
        B = build_basis([xx], x_c2, a2, r2, gamma2_opt, M2)
        u_pred[i] = B.dot(alpha2)

u_true = u_exact(x_test)

# ----------------------------
# Plot results
# ----------------------------
plt.figure(figsize=(12, 8))

plt.subplot(2,1,1)
plt.plot(x_test, u_true, 'b-', linewidth=2, label='Exact Solution')
plt.plot(x_test, u_pred, 'r--', linewidth=1.5, label='Multi-TransNet
Prediction')
plt.axvline(Gamma, color='k', linestyle='--', alpha=0.7,
label='Interface')
plt.xlabel('x'); plt.ylabel('u(x)')
plt.title('Solution Comparison')
plt.legend(); plt.grid(alpha=0.3)

plt.subplot(2,1,2)
err = np.abs(u_pred - u_true)
plt.semilogy(x_test, err, 'g-', linewidth=1.2)
plt.axvline(Gamma, color='k', linestyle='--', alpha=0.7)
plt.xlabel('x'); plt.ylabel('Absolute Error (log scale)')
plt.title('Absolute Error (log scale)')
```

```
    plt.grid(alpha=0.3)

    plt.tight_layout()
    plt.show()

    # ----------------------------
    # Compute relative errors
    # ----------------------------
    rel_l2 = np.linalg.norm(u_pred - u_true) / np.linalg.norm(u_true)
    rel_inf = np.max(np.abs(u_pred - u_true)) / np.max(np.abs(u_true))
    print(f"Relative L2 error: {rel_l2:.6e}")
    print(f"Relative L_inf error: {rel_inf:.6e}")
```