Initialization

Run the following code to import the modules you'll need. After your finish the assignment, **remember to run all cells** and save the note book to your local machine as a PDF for gradescope submission.

```
import os
import numpy as np
import matplotlib.pyplot as plt
```

Download data

In this section we will download the data and setup the paths.

```
# Download the data
if not os.path.exists('/content/bead_data.npz'):
    !wget https://www.cs.cmu.edu/~deva/data/bead_data.npz -O /content/bead_data.npz

if not os.path.exists('/content/hammer_handle_data.npz'):
    !wget https://www.cs.cmu.edu/~deva/data/hammer_handle_data.npz -O /content/hammer_handle_data.npz
```

```
--2025-09-26 19:17:18--  https://www.cs.cmu.edu/~deva/data/bead_data.npz
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 138394388 (132M)
Saving to: ‘/content/bead_data.npz’

/content/bead_data. 100%[===================>] 131.98M  1.58MB/s    in 77s

2025-09-26 19:18:36 (1.72 MB/s) - ‘/content/bead_data.npz’ saved [138394388/138394388]

--2025-09-26 19:18:37--  https://www.cs.cmu.edu/~deva/data/hammer_handle_data.npz
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 92698388 (88M)
Saving to: ‘/content/hammer_handle_data.npz’

/content/hammer_han 100%[===================>]  88.40M  1.95MB/s    in 46s

2025-09-26 19:19:23 (1.92 MB/s) - ‘/content/hammer_handle_data.npz’ saved [92698388/92698388]
```

We provide the following helper function, which computes the rotation matrix from the parameterization θ and its derivative with respect to rotation component of θ.

```
"""
Helper Functions
"""

def get_dR_dtheta_xyz(theta_x, theta_y, theta_z):
    """
    Compute parital derivatives of the rotation matrix R = Rz(theta_z) @ Ry(theta_y) @ Rx(theta_x) wrt theta_x, theta_y, theta_z.

    :param[float] theta_x              : x element of x-y-z Euler Rotation.
    :param[float] theta_y              : y element of x-y-z Euler Rotation.
    :param[float] theta_z              : z element of x-y-z Euler Rotation.
    :return
        [np.array (3, 3)] dR_dtheta_x   : The partial derivative of the rotation matrix with regards to theta_x.
        [np.array (3, 3)] dR_dtheta_y   : The partial derivative of the rotation matrix with regards to theta_y.
        [np.array (3, 3)] dR_dtheta_z   : The partial derivative of the rotation matrix with regards to theta_z.
    """
    cx, cy, cz = np.cos(theta_x), np.cos(theta_y), np.cos(theta_z)
    sx, sy, sz = np.sin(theta_x), np.sin(theta_y), np.sin(theta_z)

    # Rotation matrices
    Rx = np.array([
        [1, 0, 0],
        [0, cx, -sx],
        [0, sx, cx]
```

```
        ])
        Ry = np.array([
            [cy, 0, sy],
            [0, 1, 0],
            [-sy, 0, cy]
        ])
        Rz = np.array([
            [cz, -sz, 0],
            [sz,  cz, 0],
            [0,   0,  1]
        ])

        # Derivatives of Rx wrt x
        dRx_dx = np.array([
            [0, 0, 0],
            [0, -sx, -cx],
            [0, cx, -sx]
        ])
        # Derivatives of Ry wrt y
        dRy_dy = np.array([
            [-sy, 0, cy],
            [0, 0, 0],
            [-cy, 0, -sy]
        ])
        # Derivatives of Rz wrt z
        dRz_dz = np.array([
            [-sz, -cz, 0],
            [cz, -sz, 0],
            [0, 0, 0]
        ])

        # Chain rule for full R = Rz * Ry * Rx
        dR_dx = Rz @ Ry @ dRx_dx
        dR_dy = Rz @ dRy_dy @ Rx
        dR_dz = dRz_dz @ Ry @ Rx

        return dR_dx, dR_dy, dR_dz

    def get_R(theta_x, theta_y, theta_z):
        """
        Compute the rotation matrix R = Rz(theta_z) @ Ry(theta_y) @ Rx(theta_x).

        :param[float] theta_x        : x element of x-y-z Euler Rotation.
        :param[float] theta_y        : y element of x-y-z Euler Rotation.
        :param[float] theta_z        : z element of x-y-z Euler Rotation.
        :return[np.array (3, 3)] R   : The rotation matrix.
        """
        cx, cy, cz = np.cos(theta_x), np.cos(theta_y), np.cos(theta_z)
        sx, sy, sz = np.sin(theta_x), np.sin(theta_y), np.sin(theta_z)

        # Rotation matrices
        Rx = np.array([
            [1, 0, 0],
            [0, cx, -sx],
            [0, sx, cx]
        ])
        Ry = np.array([
            [cy, 0, sy],
            [0, 1, 0],
            [-sy, 0, cy]
        ])
        Rz = np.array([
            [cz, -sz, 0],
            [sz,  cz, 0],
            [0,   0,  1]
        ])
        R = Rz @ Ry @ Rx
        return R
```

Q5.2: Implement NormalFlow (15 points)

Make sure to comment your code and use proper names for your variables.

```
from scipy.interpolate import RectBivariateSpline
from numpy.linalg import lstsq

def rbs(img):
    H, W = img.shape
    return RectBivariateSpline(np.arange(H), np.arange(W), img, kx=1, ky=1)

def eval3(rbs_list, y, x):
    return np.stack([r.ev(y, x) for r in rbs_list], axis=1)

def gradients(img3):
    H, W, C = img3.shape
    Ix = np.zeros_like(img3, dtype=np.float64)
    Iy = np.zeros_like(img3, dtype=np.float64)
    for c in range(C):
        Iy[..., c], Ix[..., c] = np.gradient(img3[..., c])
    return Ix, Iy

def inside_bounds(x, y, W, H):
    return (x >= 0) & (x <= W - 1) & (y >= 0) & (y <= H - 1)

def NormalFlow(It, It1, Ht, Ht1, Ct, Ct1, max_iters=20, rot_threshold=0.0001, trans_threshold=0.1, init_theta=np.zeros(5)):
    """
    :param[np.array(H, W, 3)] It   : Normal Map at time t [float]
    :param[np.array(H, W, 3)] It1  : Normal Map at time t+1 [float]
    :param[np.array(H, W)] Ht      : Height Map at time t, unit: pixel [float]
    :param[np.array(H, W)] Ht1     : Height Map at time t+1, unit: pixel [float]
    :param[np.array(H, W)] Ct      : Contact Mask at time t [bool]
    :param[np.array(H, W)] Ct1     : Contact Mask at time t+1 [bool]
    :param[int] max_iters          : Max number of optimization iterations
    :param[float] rot_threshold    : If change in rotation parameters is less than thresh, terminate the optimization
    :param[float] trans_threshold  : If change in translation parameters is less than thresh, terminate the optimization
    :param[np.array(5)] init_theta : The initial parameter
    :return[np.array(4, 4)] t1_T_t : The Homogeneous transformation matrix of the object from frame t to frame t+1
    """
    # Initialize theta to zeros
    theta = init_theta.copy()

    # ===== your code here! =====
    # Hint: Iterate over max_iters and for each iteration, construct a linear system (Ax=b) that solves for a x=delta_theta update
    # Construct [A] by computing image gradients at (possibly fractional) pixel locations and for each channel
    # It might be easier to consider compute the gradient over each element of theta independently
    # For each iteration, we suggest to first compute the warped pixel, then compute the shared contact mask, finally construct the
    # We suggest using RectBivariateSpline from scipy.interpolate to interpolate pixel values at fractional pixel locations
    # We suggest using lstsq from numpy.linalg to solve the linear system
    # Once you solve for [delta_theta], add it to [theta] (and move on to next iteration)
    # Use the following termination condition:
    #
    # if (np.linalg.norm(delta_theta[:3]) < rot_threshold and np.linalg.norm(delta_theta[3:]) < trans_threshold):
    #     break
    #
    # HINT/WARNING:
    # RectBivariateSpline and Meshgrid use inconsistent defaults with respect to 'xy' versus 'ij' indexing:
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RectBivariateSpline.ev.html#scipy.interpolate.RectBiva
    # https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html

    It   = It.astype(np.float64)
    It1  = It1.astype(np.float64)
    Ht   = Ht.astype(np.float64)
    Ct   = Ct.astype(bool)
    Ct1  = Ct1.astype(bool)

    H, W, _ = It.shape
    yy, xx = np.meshgrid(np.arange(H), np.arange(W), indexing='ij')

    rbs_I1  = [rbs(It1[..., c]) for c in range(3)]
    Ix1, Iy1 = gradients(It1)
    rbs_Ix1 = [rbs(Ix1[..., c]) for c in range(3)]
    rbs_Iy1 = [rbs(Iy1[..., c]) for c in range(3)]
    rbs_Ct1 = rbs(Ct1.astype(np.float64))

    mask_ref = Ct
    y_ref = yy[mask_ref].ravel()
    x_ref = xx[mask_ref].ravel()
    N = x_ref.size

    if N == 0:
        return np.asarray(init_theta, dtype=np.float64).reshape(5,)
```

```
n_ref = It[mask_ref].reshape(N, 3)
h_ref = Ht[mask_ref].reshape(N)
x3_ref = np.stack([x_ref, y_ref, h_ref], axis=1)

def build_rotation_and_partials(th):
    thx, thy, thz = float(th[0]), float(th[1]), float(th[2])
    R = get_R(thx, thy, thz)
    dRx, dRy, dRz = get_dR_dtheta_xyz(thx, thy, thz)
    return R, dRx, dRy, dRz

for i in range(max_iters):
    R, dRx, dRy, dRz = build_rotation_and_partials(theta)

    RX = (x3_ref @ R.T)
    xw = RX[:, 0] + theta[3]
    yw = RX[:, 1] + theta[4]

    inb = inside_bounds(xw, yw, W, H)
    if not np.any(inb):
        break

    xw_in = xw[inb]
    yw_in = yw[inb]
    X3_in = x3_ref[inb, :]
    n_ref_in = n_ref[inb, :]

    Ct1_vals = rbs_Ct1.ev(yw_in, xw_in)
    shared = Ct1_vals > 0.5
    if not np.any(shared):
        break

    xw_c = xw_in[shared]
    yw_c = yw_in[shared]
    X3_c = X3_in[shared, :]
    n_ref_c = n_ref_in[shared, :]
    M = xw_c.size

    n_warp = eval3(rbs_I1,  yw_c, xw_c)
    Ix_s   = eval3(rbs_Ix1, yw_c, xw_c)
    Iy_s   = eval3(rbs_Iy1, yw_c, xw_c)

    Rn_ref = (R @ n_ref_c.T).T
    r = (n_warp - Rn_ref).reshape(-1)

    A = np.zeros((3*M, 5), dtype=np.float64)

    dRX_x = (X3_c @ dRx.T)
    dRX_y = (X3_c @ dRy.T)
    dRX_z = (X3_c @ dRz.T)

    dW_dthx = np.stack([dRX_x[:, 0], dRX_x[:, 1]], axis=1)
    dW_dthy = np.stack([dRX_y[:, 0], dRX_y[:, 1]], axis=1)
    dW_dthz = np.stack([dRX_z[:, 0], dRX_z[:, 1]], axis=1)
    dW_dtx  = np.tile(np.array([1.0, 0.0]), (M,1))
    dW_dty  = np.tile(np.array([0.0, 1.0]), (M,1))

    dR_n_thx = (dRx @ n_ref_c.T).T  # (M,3)
    dR_n_thy = (dRy @ n_ref_c.T).T
    dR_n_thz = (dRz @ n_ref_c.T).T

    row = 0
    for c in range(3):
        grad_c = np.stack([Ix_s[:, c], Iy_s[:, c]], axis=1)  # (M,2)
        A[row:row+M, 0] = np.sum(grad_c * dW_dthx, axis=1) - dR_n_thx[:, c]
        A[row:row+M, 1] = np.sum(grad_c * dW_dthy, axis=1) - dR_n_thy[:, c]
        A[row:row+M, 2] = np.sum(grad_c * dW_dthz, axis=1) - dR_n_thz[:, c]
        A[row:row+M, 3] = np.sum(grad_c * dW_dtx,  axis=1)
        A[row:row+M, 4] = np.sum(grad_c * dW_dty,  axis=1)
        row += M

    delta_theta, *_ = lstsq(A, -r, rcond=None)
    theta += delta_theta.reshape(5,)

    if (np.linalg.norm(delta_theta[:3]) < rot_threshold and
        np.linalg.norm(delta_theta[3:]) < trans_threshold):
        break
```

```
    theta = np.asarray(theta, dtype=np.float64).reshape(5,)

    # ===== End of code =====
    return theta
```

Debug Q5.2

Debugging NormalFlow can be challenging. We provide the ground-truth object pose (in θ) between frame 90 and frame 0 in the code below. You can use this to verify your implementation. Below are some tips for debugging:

- **Warping function**: Plot $I_{t+1}(\mathbf{W}(x;\theta))$ and $\mathbf{R}_\theta I_t(x)$, masked with the shared mask $\bar{C}$. If your implementation is correct, the two should match. A helper function below is provided to convert a normal map into an RGB image.
- **Gradient of the warped image**: Once you have a correct implementation of $I_{t+1}(\mathbf{W}(x;\theta))$, compute its numerical derivative $\big(I_{t+1}(\mathbf{W}(x;\theta+d\theta)) - I_{t+1}(\mathbf{W}(x;\theta))\big)/d\theta$ for each dimension of θ. Plot the results and check if they match your implementation of $\nabla I_{t+1}(\mathbf{W}(x;\theta))$.
- **Gradient of the rotated image**: Use the same procedure to debug your implementation of $\nabla \mathbf{R}_\theta I_t(x)$.
- **Convergence with provided initialization**: If you use the provided θ as `init_theta`, NormalFlow should converge very quickly to a value very close to the ground-truth θ.
- **Convergence plot**: Plot the iterations versus the norm of `delta_theta`. Your algorithm should show fast convergence.

```
"""
Helper Functions
"""

def normal2image(I):
    """
    Given a normal map, turn it into an image and return that

    :param[np.array (H, W, 3)] I        : The Normal Map
    :return[np.array (H, W, 3)] image   : The image
    """
    image = (np.clip((I + 1.0) / 2.0, 0.0, 1.0) * 255).astype(np.uint8)
    return image
```

```
# load the data
data_path = "/content/bead_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# We offer you the true object pose (in theta) of frame 90 relevant to frame 0
It = Is[0]; Ct = Cs[0]; Ht = Hs[0]
It1 = Is[90]; Ct1 = Cs[90]; Ht1 = Hs[90]
true_theta = [-0.00875, -0.00544, 0.15949, 43.419, -49.342]

# Plot the images
plt.imshow(normal2image(It))
plt.axis('off')
plt.show()
plt.imshow(normal2image(It1))
plt.axis('off')
plt.show()
```

```
Start coding or generate with AI.
```

Q5.3: Tracking with NormalFlow (10 points)

Implement the tracking method that, given a sequence of tactile-derived local geometries, returns the object pose of each frame relative to the first frame in the form of a 4×4 homogeneous transformation matrix. A helper function is provided to convert θ into a homogeneous matrix. For convenience, this helper assumes zero translation along the z-axis, since the object remains in contact with the sensor.

```
"""
Helper Functions
"""

def theta2T(theta):
    """
    Given the theta, assume zero z-translation, compute homogeneous transform matrix.

    :param[np.array (5)] theta        : The parameterization of the object transformation.
    :return[np.array (4, 4)] T        : The homogeneous transformation matrix.
    """
    T = np.eye(4, dtype=np.float32)
    T[:3, :3] = get_R(theta[0], theta[1], theta[2])
    T[0, 3] = theta[3]
    T[1, 3] = theta[4]
    return T
```

```
def TrackSequence(Is, Hs, Cs):
    """
    :param[np.array(N, H, W, 3)] Is   : Normal Map sequence [float]
```

```
    :param[np.array(N, H, W)] Hs      : Height Map sequence, unit: pixel [float]
    :param[np.array(N, H, W)] Cs      : Contact Mask sequence [bool]
    :param[int] max_iters             : Max number of optimization iterations
    :param[float] rot_threshold       : If change in rotation parameters is less than thresh, terminate the optimization
    :param[float] trans_threshold     : If change in translation parameters is less than thresh, terminate the optimization
    :return:[np.array(N, 4, 4)] Ts    : object pose with regards to the first frame in the form of homogeneous transformation
    """
    init_T = np.eye(4, dtype=np.float32)
    Ts = [init_T.copy()]

    max_iters = 20
    rot_threshold = 1e-4
    trans_threshold = 0.1

    # Iterate over the car sequence and track the car
    for i in range(Is.shape[0] - 1):

        # ===== your code here! =====
        # TODO: add your code track the 6DoF pose of the object
        It, It1 = Is[i], Is[i + 1]
        Ht, Ht1 = Hs[i], Hs[i + 1]
        Ct, Ct1 = Cs[i], Cs[i + 1]

        theta = NormalFlow(It, It1, Ht, Ht1, Ct, Ct1, max_iters=max_iters, rot_threshold=rot_threshold, trans_threshold=trans_thre
        t1_T_t = theta2T(theta)

        global_T = t1_T_t @ Ts[-1]
        Ts.append(global_T)

        # ===== End of code =====

    Ts = np.stack(Ts, 0)
    assert Ts.shape == (Is.shape[0], 4, 4), f"Your output sequence {Ts.shape} is not ({Is.shape[0]}x{4}x{4})"
    return Ts
```

Q5.3 (a) - Track Bead Sequence

Run the snippets below. If your implementations of NormalFlow and TrackSequence are correct, each normal image will show the correct estimated 6DoF object pose with an annotated coordinate frame. Feel free to experiment with the parameters in the code.

```
"""
Helper Functions
"""

def display_normal_with_coordinate_system(I, T, center_ref, title, vector_scale=50):
    """
    Show the normal map annotating with the computed coordinate system
    :param[np.array(H, W, 3)] I     : Normal Map [float]
    :param[np.array(4, 4)] T        : The homogeneous transformation matrix
    :param[np.array(2)] center_ref  : The x-y origin of the reference frame (first frame)
    :param[float] vector_scale      : The size of the coordinate axis arrow
    """
    fig, ax = plt.subplots(figsize=(3,3))
    ax.imshow(normal2image(I))

    # Get the center and the unit_vector
    unit_vectors_ref = np.eye(3)[:, :2]
    center_3d_ref = np.array([(center_ref[0] - I.shape[1] / 2 + 0.5), (center_ref[1] - I.shape[0] / 2 + 0.5), 0])
    unit_vectors_3d_ref = np.eye(3)
    remapped_center_3d_ref = (
        np.dot(T[:3, :3], center_3d_ref) + T[:3, 3]
    )
    remapped_cx_ref = remapped_center_3d_ref[0] + I.shape[1] / 2 - 0.5
    remapped_cy_ref = remapped_center_3d_ref[1] + I.shape[0] / 2 - 0.5
    remapped_center_ref = np.array([remapped_cx_ref, remapped_cy_ref]).astype(
        np.int32
    )
    remapped_unit_vectors_ref = (
        np.dot(T[:3, :3], unit_vectors_3d_ref.T).T
    )[:, :2]

    # Annotate the origin
    ax.scatter(remapped_center_ref[0], remapped_center_ref[1], c="k", alpha=1, s=10, zorder=3)
```

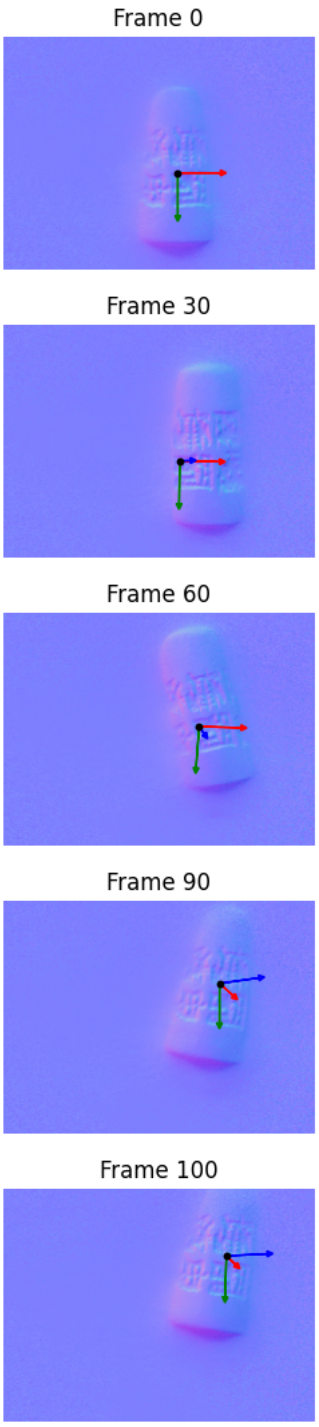
```
    # Annotate the three axes
    colors = ["r", "g", "b"]
    for i in range(3):
        dx, dy = remapped_unit_vectors_ref[i] * vector_scale
        ax.arrow(
            remapped_center_ref[0], remapped_center_ref[1], dx, dy,
            head_width=6, head_length=6,
            fc=colors[i], ec=colors[i],
            linewidth=1, length_includes_head=True
        )

    ax.set_axis_off()
    ax.set_title(title)
    plt.show()

def visualize_track(Is, Ts, vis_seq=[0]):
    # Visualize tracks on a seuqence of a selected sequence
    ys, xs = np.nonzero(Cs[0])
    cx_ref = xs.mean()
    cy_ref = ys.mean()
    center_ref = np.array([cx_ref, cy_ref]).astype(np.int32)
    for idx in vis_seq:
        display_normal_with_coordinate_system(Is[idx], Ts[idx], center_ref, "Frame %d"%idx)
```

```
# load the data
data_path = "/content/bead_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# Track the Bead
Ts = TrackSequence(Is, Hs, Cs)
# Visualize the tracking result
visualize_track(Is, Ts, [0, 30, 60, 90, 100])
```




```
Start coding or generate with AI.
```

Q5.3 (b) - Track Hammer Handle Sequence

Track the handle of the hammer, which is made by wood.

```
# load the data
data_path = "/content/hammer_handle_data.npz"
data = np.load(data_path, allow_pickle=True)
Cs = data["contact_masks"]
Is = data["normal_maps"]
Hs = data["height_maps"]

# Track the Hammer Handle
Ts = TrackSequence(Is, Hs, Cs)
```

```
# Visualize the tracking result
```