Question 1.1

Thursday, October 13, 2022 12:14 AM

What is ∂W(x;p) / ∂pT (should a 2x2 matrix)?

This 2x2 represents the Jacobian of the image, for the case of the translational warps in problem 1, are simply the identity matrix [1, 0; 0, 1].

What is A and b?

The A matrix refers to the "Steepest Descent" of the iteration defined by the gradient transpose times the Jacobian. The B matrix refers to the error image, or the template minus the warped image.

What conditions must AT A meet so that a unique solution to Δp can be found?

AtA must be invertible and well conditioned. For the first condition, in the Lk algorithm we are AtA is our hessian. In order to complete the tracking scheme we require the inverse of the hessian, so it would logically follow that the hessian must be invertible. Well conditioned refers to the ability of the frames to be tracked: the eigenvalues must be reasonably large such and one eigenvalue must be larger than the other. If both these conditions are met, we have located a good textured image that should be sufficiently tracked.

Question 1.2

Thursday, October 13, 2022 1:45 AM

```python
    # set up the threshold
    ################## TODO Implement Lucas Kanade ##################

    p = np.copy(p0)
    deltaP = np.zeros(2)

    # define the jacobian
    jac = np.array([[1, 0], [0, 1]])

    # make a spline of It and It1
    # it, it1 dims
    It_row, It_col = It.shape
    It1_row, It1_col = It1.shape
    # arrange vals then make a mesh to create the spline
    It_x = np.arange(0, It_row)
    It_y = np.arange(0, It_col)
    It_spline = RectBivariateSpline(It_x, It_y, It)
    It1_x = np.arange(0, It1_row)
    It1_y = np.arange(0, It1_col)
    It1_spline = RectBivariateSpline(It1_x, It1_y, It1)

    # obtain the template from It spline
    # first define the coordinates of the rectangle
    x1 = rect[0]
    y1 = rect[1]
    x2 = rect[2]
    y2 = rect[3]
    # define the meshgrid to obtain the template from
    rect_row = np.arange(y1, y2)
    rect_col = np.arange(x1, x2)
    rr_mesh, rc_mesh = np.meshgrid(rect_row, rect_col)
    T = It_spline.ev(rr_mesh, rc_mesh)

    # ? Debugging Log
    # // with open("lk_log.txt", "a") as log:
    # //     log.write(f"Start Values | Delta P {deltaP} | P: {str(p)}\n")

    error = 1
    i = 0
    while error > threshold and i < num_iters:
        # now find the warped image I(W(x;p))
        # the image is defined by its new x and y pos
        W_xp_row = np.arange(y1, y2) + p[1]
        W_xp_col = np.arange(x1, x2) + p[0]
        W_xp_r_mesh, W_xp_c_mesh = np.meshgrid(W_xp_row, W_xp_col)

        # evaluate the spline with the new meshes, and the derivatives
        I_Wxp = It1_spline.ev(W_xp_r_mesh, W_xp_c_mesh)
        Ix = It1_spline.ev(W_xp_r_mesh, W_xp_c_mesh, dx=1, dy=0)
        Iy = It1_spline.ev(W_xp_r_mesh, W_xp_c_mesh, dx=0, dy=1)

        # combine the derivatives into a tensor, the matrix A
        A = np.stack((Iy.flatten(), Ix.flatten()), axis=1)
        # mult with jacobian
        A = np.dot(A, jac)

        # find the error image B
        D = (T - I_Wxp).flatten()
        B = np.dot(A.T, D)

        # now solve the linear equations in steps
        Hessian = np.dot(A.T, A)
        invHessian = np.linalg.inv(Hessian)
        deltaP = np.dot(invHessian, B)

        p += deltaP
        i += 1

        # ? Debugging Log
        # //with open("lk_log.txt", "a") as log:
        # //     log.write(f"Iteration: {i} | Delta P {deltaP} | P: {str(p)}\n")

        error = np.linalg.norm(deltaP, ord=2)**2
        # //print(f"Iteration {i} || Error: {error} || P: {p}")

    # ? Debugging Log
    # // with open("lk_log.txt", "a") as log:
    # //     log.write(
    # //         f"Total Iterations: {i} | Final dp: {deltaP} | Final P: {str(p)}\n")

    # //print(f"Iteration {i} || Error: {error} || P: {p}")

    return p
```

Question 1.3

Thursday, October 13, 2022 1:46 AM

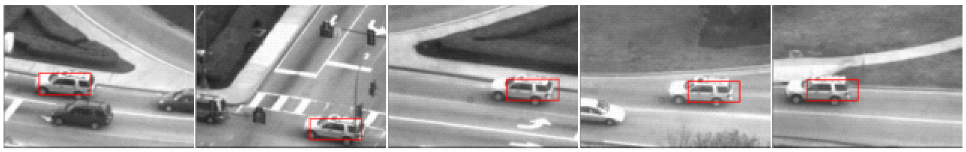

Fig. *Lucas Kanade Template Tracking on Car Sequence, without template correction.*

Fig. *Lucas Kanade Template Tracking on Girl Sequence, without template correction.*

Question 1.3

Thursday, October 13, 2022 1:50 AM

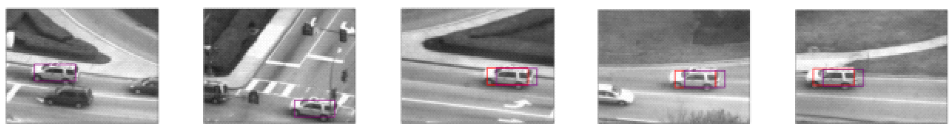

Fig. *Lucas Kanade Template Tracking on Car Sequence, with template correction.*

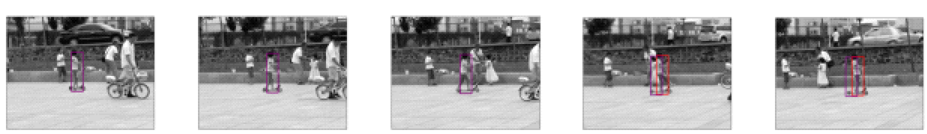

Fig. *Lucas Kanade Template Tracking on Girl Sequence, with template correction.*

Question 1.4 (Code)

Thursday, October 13, 2022 1:52 AM

```python
for i in tqdm(range(1, seq.shape[2])):

    ########## TODO Implement LK with drift correction ##########
    ########## TODO Implement LK with drift correction ##########
    pt_topleft = rect[:2]
    pt_bottomright = rect[2:4]
    It = seq[:, :, i-1]
    It1 = seq[:, :, i]

    # ? Debugging Log
    # // with open("lk_log.txt", "a") as log:
    # //     log.write(f"Iteration: {i} | Template Frame: {template_idx}\n")

    # first perform the tracking step: between the current template and current update image: like before
    p = LucasKanade(It, It1, rect, threshold, num_iters)

    # accumulate p to compare error between this and pstar, the dist from template 0
    p_acc += p

    # ? Debugging Log
    # // with open("lk_log.txt", "a") as log:
    # //    log.write(f"P_acc: {p_acc} | P: {p} | ")

    # then perform the update step with p0 being my p_n from the tracking step
    # p_star = LucasKanade(It0, It1, rect_0, threshold, num_iters, p_accumulated)
    p_star = LucasKanade(It0, It1, rect_0, threshold,
                         num_iters, p0=np.copy(p_acc))

    # this error represents how different the current template is from the original template (essentially)
    error_e = np.linalg.norm((p_star - p_acc))

    # ? Debugging Log
    # // with open("lk_log.txt", "a") as log:
    # //     log.write(f"P Star | {p_star} | Error: {error_e}")
    # //     log.write(
    # //         " \n\n++++++++++++++++++++++++ Iteration End +++++++++++++++++++++\n\n ")

    # if the image is not different from the original, then just keep updating as usual
    if error_e <= threshold_drift:
        rect = np.concatenate((pt_topleft + p, pt_bottomright + p))
    else:
        rect = np.concatenate(
            (Io_top_left + p_star, Io_bottom_right + p_star))
```

Question 2.1

Thursday, October 13, 2022 2:25 AM

```python
"""" Precomputation goals:

1. Create a spline that represents It, It1
    1.a Find the dims of the template, image
        - 1.a.1 It dims
        - 1.a.2 It1 dims
    1.b Find arrange axis from 0->w, 0->h for for both It, It1
        - 1.b.1 It axes
        - 1.b.2 It1 axes
    1.c Find the meshgrid that defines axes defined in 1.b
        - Will be used to evaluate the splines later on
    1.d Create the splines with the axes from 1.b and It, It1
    1.e Convert the meshgrid into a flattened array
    1.g Precompute the unwarpe template and gradients of It1"""

#! 1.a
x0, y0 = It.shape  # 1.a.1
x1, y1 = It1.shape  # 1.a.2

#! 1.b
rows0 = np.arange(0, x0)  # 1.b.1
cols0 = np.arange(0, y0)
# ---------------------
rows1 = np.arange(0, x1)  # 1.b.2
cols1 = np.arange(0, y1)

#! 1.c
rows_mesh0, cols_mesh0 = np.meshgrid(rows0, cols0)
rows_mesh1, cols_mesh1 = np.meshgrid(rows1, cols1)

#! 1.d
It_spline = RectBivariateSpline(rows0, cols0, It)
It1_spline = RectBivariateSpline(rows1, cols1, It1)

#! 1.e
xInd1 = rows_mesh1.flatten()
yInd1 = cols_mesh1.flatten()

#! 1.f
arr_1s = np.ones(shape=(1, len(xInd1)))
# origPixels = np.vstack((xInd1, yInd1, arr_1s))

#! 1.g
Ix = It1_spline.ev(rows_mesh1, cols_mesh1, dx=1, dy=0)
Iy = It1_spline.ev(rows_mesh1, cols_mesh1, dx=0, dy=1)

It1_frame = It1_spline.ev(rows_mesh1, cols_mesh1).T
template = It_spline.ev(rows_mesh0, cols_mesh0).T
# // disp_img(template, "template")
```

```python
error = 1
i = 0
p = np.zeros(6)
while error > threshold and i < num_iters:

    M[0, 0] = 1 + p[0]
    M[0, 1] = p[1]
    M[0, 2] = p[2]
    M[1, 0] = p[3]
    M[1, 1] = p[4] + 1
    M[1, 2] = p[5]

    # print("i: ", i, "\n", "M: ", M)

    """ Step 1: Affine warp It1, and the It1_grad  """

    # print("Step 1")

    warpedIt1 = affine_transform(It1_frame, M)
    # // disp_img(warpedIt1, "warped it1")
    warpedIx = affine_transform(Ix, M).T
    #disp_img(warpedIx, "warped ix")
    warpedIy = affine_transform(Iy, M).T
    #disp_img(warpedIy, "warped iy")

    # flatten the gradients and warped image
    warpedIt1_flat = warpedIt1.flatten()
    warpedIx_flat = warpedIx.flatten()
    warpedIy_flat = warpedIy.flatten()

    """Step 2: Fill in where 0s exist on the warped image """

    # print("Step 2")

    template_temp = np.copy(template)
    zero_ind = np.where(warpedIt1 == 0)
    template_temp[zero_ind] = 0
    # // disp_img(template, "template blacked out")
    template_temp_flat = template_temp.flatten()

    """Step 3: Find the error image """

    # print("Step 3")

    errorImg = (template_temp_flat - warpedIt1_flat)

    # print("Error Image\n", errorImg[20:30])

    """Step 4: Calculate the steepest descent directly  """

    # print("Step 4")

    # flatten the rows of the meshgrid, to signify the repeated x, y
    x_locs = xInd1.flatten()
    y_locs = yInd1.flatten()

    # find the elements of the steepest descent beforehand
    ele1 = warpedIx_flat * x_locs
    ele2 = warpedIy_flat * x_locs
    ele3 = warpedIx_flat * y_locs
    ele4 = warpedIy_flat * y_locs
    ele5 = warpedIx_flat
    ele6 = warpedIy_flat

    steepestDescent = np.array([ele2, ele4, ele6, ele1, ele3, ele5]).T

    #print("\n\nsteepest D\n", steepestDescent[0:10], "\n\n")

    """Step 5: use the steepest descent to find the hessian, hessian inverse"""

    # print("Step 5")
    hessian = np.dot(steepestDescent.T, steepestDescent)
    invHessian = np.linalg.inv(hessian)

    """Step 6: calculate delta p, update params"""

    # print("Step 6")
    deltaP = np.dot(invHessian, np.dot(steepestDescent.T, errorImg))

    p += deltaP

    #print("M\n", M)
    error = np.linalg.norm(deltaP)

    i += 1
#
print("Final Iterations: ", i)

return M
```

Question 2.2

Thursday, October 13, 2022 2:21 AM

```python
    ################### TODO Implement Substract Dominent Motion ###################

    # M = LucasKanadeAffine(
    #     image1, image2, threshold, num_iters)
    M = InverseCompositionAffine(
        image1, image2, threshold, num_iters)

    # //print("M: ", M)

    img2_w = affine_transform(np.copy(image2), M)

    errorImg = img2_w - np.copy(image1)

    mask[errorImg > tolerance] = 1
    mask[errorImg < tolerance] = 0

    mask = binary_erosion(mask, iterations=1)
    mask = binary_dilation(mask, iterations=4)

    return mask.astype(bool)
```

Question 2.3

Thursday, October 13, 2022 2:15 AM

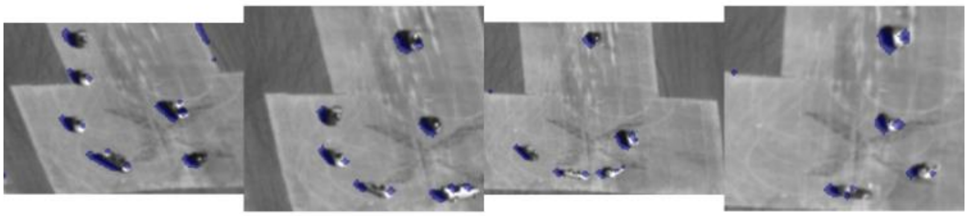

Fig. *Lucas Kanade Affine tracking on Aerial Test Sequence with motion detection.*

Fig. *Lucas Kanade Affine tracking on Ant Test Sequence with motion detection.*

Question 3.1

Thursday, October 13, 2022 1:59 AM

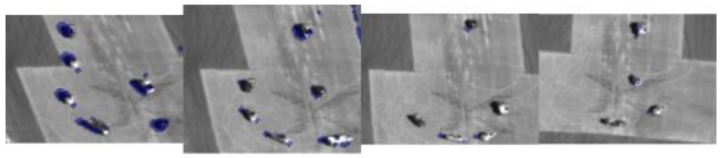

Fig. *Lucas Kanade Affine tracking on Aerial Test Sequence with Inverse Compositional warp.*

Fig. *Lucas Kanade Affine tracking on Aerial Test Sequence with Inverse Compositional warp.*

Table. *Runtime Comparison of Baseline Lucas Kanade and the Inverse Compositional Method on the Aerial Test Sequence.*

Num_Iters	Baseline Lucas Kanade	Inverse Compositional
1000	~ 17 minutes	~ 13 minutes
500	~ 9 minutes	~ 4.5 minutes

Discussion:

The inverse compositional method allows us to precompute many of the hindering mathematical terms up front: the hessian, steepest decent, and gradients are all handled before generating the correct affine transform. In the baseline Lucas Kanade tracking, since we were tracking the motion of the It1 frame to the It frame, we were forced to recalculate the aforementioned computationally expensive terms (hessian, steepest descents, gradients, etc.). By leveraging some extra math we are able to track the template to the It1 frame and then use that to find the correct warp from It1 to the template with an inverted incremental warp.

Question 3.1 (Code)

Thursday, October 13, 2022 1:54 AM

```python
# put your implementation here
M0 = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

#################### TODO Implement Inverse Composition Affine ####################

"""" Precomputation goals:

1. Create a spline that represents It, It1
    1.a Find the dims of the template, image
        - 1.a.1 It dims
        - 1.a.2 It1 dims
    1.b Find arrange axis from 0->w, 0->h for for both It, It1
        - 1.b.1 It axes
        - 1.b.2 It1 axes
    1.c Find the meshgrid that defines axes defined in 1.b
        - Will be used to evaluate the splines later on
    1.d Create the splines with the axes from 1.b and It, It1
    1.e Convert the meshgrid into a flattened array for x and y ind
    1.f Evaluate the spline
    1.g Precompute the unwarped template and gradients of It as well as It1"""

#! 1.a
x0, y0 = It.shape  # 1.a.1
x1, y1 = It1.shape  # 1.a.2

#! 1.b
rows0 = np.arange(0, x0)  # 1.b.1
cols0 = np.arange(0, y0)
# ----------------------
rows1 = np.arange(0, x1)  # 1.b.2
cols1 = np.arange(0, y1)

#! 1.c
rows_mesh0, cols_mesh0 = np.meshgrid(rows0, cols0)
rows_mesh1, cols_mesh1 = np.meshgrid(rows1, cols1)

#! 1.d
It_spline = RectBivariateSpline(rows0, cols0, It)
It1_spline = RectBivariateSpline(rows1, cols1, It1)

#! 1.e
xInd0 = rows_mesh0.flatten()
yInd0 = cols_mesh0.flatten()

#! 1.f
Ix_T = It_spline.ev(rows_mesh0, cols_mesh0, dx=1, dy=0)
Iy_T = It_spline.ev(rows_mesh0, cols_mesh0, dx=0, dy=1)

Ix_T_flat = Ix_T.flatten()
Iy_T_flat = Iy_T.flatten()

It1_frame = It1_spline.ev(rows_mesh1, cols_mesh1).T
template = It_spline.ev(rows_mesh0, cols_mesh0).T

#! 1.g
# flatten the rows of the meshgrid, to signify the repeated x, y
x_locs = xInd0.flatten()
y_locs = yInd0.flatten()

# find the elements of the steepest descent beforehand
ele1 = Ix_T_flat * x_locs

ele2 = Iy_T_flat * x_locs

ele3 = Ix_T_flat * y_locs

ele4 = Iy_T_flat * y_locs

ele5 = Ix_T_flat

ele6 = Iy_T_flat

steepestDescent = np.array([ele2, ele4, ele6, ele1, ele3, ele5]).T

#! 1.h
hessian = np.dot(steepestDescent.T, steepestDescent)
invHessian = np.linalg.inv(hessian)

# init loop params
i = 0
error = 1
deltaM = np.eye(3)
deltaP = np.zeros(6)
```

```python
#! 1.h
hessian = np.dot(steepestDescent.T, steepestDescent)
invHessian = np.linalg.inv(hessian)

# init loop params
i = 0
error = 1
deltaM = np.eye(3)
deltaP = np.zeros(6)

while error > threshold and i < num_iters:

    """ Warp It1 with the inverse of M0, M0 is the direction to go from template > It1
    """

    deltaM[0, 0] = 1 + deltaP[0]
    deltaM[0, 1] = deltaP[1]
    deltaM[0, 2] = deltaP[2]
    deltaM[1, 0] = deltaP[3]
    deltaM[1, 1] = deltaP[4] + 1
    deltaM[1, 2] = deltaP[5]

    M0 = np.dot(M0, np.linalg.inv(deltaM))

    warped_It1 = affine_transform(It1_frame, M0)

    # disp_img(template, "template")
    # disp_img(warped_It1, "It1 Frame")

    warped_It1_flat = warped_It1.flatten()

    """ Zero out out of bounds vals"""

    template_temp = np.copy(template)
    zero_ind = np.where(warped_It1 == 0)
    template_temp[zero_ind] = 0
    # // disp_img(template, "template blacked out")
    template_temp_flat = template_temp.flatten()

    """ calc the error image """

    errorImg = warped_It1_flat - template_temp_flat

    """ calculate dp  """

    deltaP = np.dot(invHessian, np.dot(steepestDescent.T, errorImg))

    # p += deltaP

    """ go directly off of delta p
    make mtemp 1 - dleta p 0
    m , inv mfinal  = matmul(M, inv(m_temp))"""

    error = np.linalg.norm(deltaP)

    i += 1

print("Final Iterations: ", i)

return M0  # invM0
```

Extra Credit

Thursday, October 13, 2022 6:53 PM

Fig. *Lucas Kanade Affine tracking on Aerial Test Sequence with Inverse Compositional warp.*

Changes:

Blurred with a Gaussian for smoother tracking and lowered the acceptable tolerance on the parameters to ensure a more selective fitting of the template to the It1 frame.