Homework 6

Student: Nguyen Van Thanh
ID: 20242084

I use python to solve this problem for my convinience.
Declaration: For some translation from matlab code to python, I do use AI tool for supports.

1.

- Downscale camera matrix K:

The downscaled camera matrix K is calculated with the fomular in page 13 of the slide:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \longrightarrow K_{\frac{1}{2}} = \begin{pmatrix} \frac{f_x}{2} & 0 & \frac{c_x+0.5}{2} - 0.5 \\ 0 & \frac{f_y}{2} & \frac{c_y+0.5}{2} - 0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

Therefore, we implement the following code in the downscale function:

```
# --- Downscale camera intrinsics ---
# formular from slide 13
Kd = K.copy()
Kd[0, 0] *= 0.5    # fx/2
Kd[1, 1] *= 0.5    # fy/2
Kd[0, 2] = 0.5 * (Kd[0, 2]+0.5) - 0.5    # 0.5*(cx+0.5)-0.5
Kd[1, 2] = 0.5 * (Kd[1, 2]+0.5) - 0.5    # 0.5*(cy+0.5)-0.5
```

- Downscale intensity image

We use equation (1) to downscale the intensity image. Equation (1) vectorized as below:

```
# --- Downscale intensity image
H,W = I.shape[0], I.shape[1] # resolution of the input image
# New resolution
H2 = H // 2
W2 = W // 2
# Initialize downscaled intensity image
Id = np.zeros((H2, W2), dtype=I.dtype)
# Vectorized from equation (1)
Id = 0.25 * (
    I[0:H:2, 0:W:2] +
    I[1:H:2, 0:W:2] +
    I[0:H:2, 1:W:2] +
    I[1:H:2, 1:W:2]
)
```

- Downscale Depth Image

We use equation (2) to downscale the depth image. The implementation is as follows:

```
# --- Downscale depth image
# initialize downscaled depth image
Dd = np.zeros((H2, W2), dtype=D.dtype)

# Extract 2x2 blocks, this is O(x,y) set
d00 = D[0:H:2, 0:W:2]
d10 = D[1:H:2, 0:W:2]
d01 = D[0:H:2, 1:W:2]
d11 = D[1:H:2, 1:W:2]

# Stack into (H2, W2, 4), then each pixel has 4 depth values
blocks = np.stack([d00, d10, d01, d11], axis=-1)

# Valid depth mask (non-zero) (H2, W2, 4) shape
valid_mask = blocks != 0 # check which pixel has valid depth, returns boolean (0/1) array

# Sum of valid depth values
valid_sum = np.sum(blocks * valid_mask, axis=-1)
```

```
    # Count of valid values (H2, W2) shape
    valid_count = np.sum(valid_mask, axis=-1)

    # Avoid division by zero → depth stays 0
    nonzero = valid_count > 0

    # Take average of valid depth values as in equation (2)
    Dd[nonzero] = valid_sum[nonzero] / valid_count[nonzero]

    # --- Recursive call ---
    return downscale(Id, Dd, Kd, level - 1)
```

We test the implemented function with a provided RGB and depth images. Figures below illustrate the downscaled images with different level.

Figure 1: Downscaled RGB images

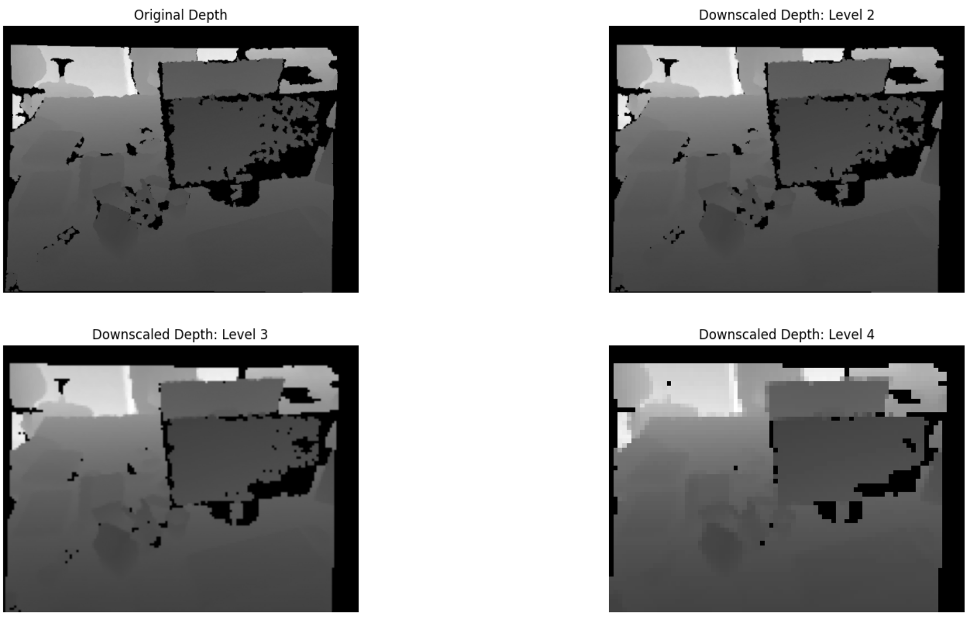


Figure 2: Downscale depth images

2.

In order to calculate the per-pixel residual $\mathbf{r}(\xi)$, we have two main computations. Fist, we warps a pixel from a ref.image to new image with $\omega(\mathbf{p}_i, d, \xi)$:

$$\omega(\mathbf{p}_i, d, \xi) = \pi(K(R_\xi)K^{-1}\begin{pmatrix} dp_{i,x} \\ dp_{i,y} \\ d \end{pmatrix} + \mathbf{t}_\xi))$$

Then projection on a new image is given by:

$$\pi(x, y, z) = \begin{pmatrix} x/z \\ y/z \end{pmatrix}$$

Then we have the following snip code

```
# TODO warp reference points to target image
d = DRef[y, x]
if d == 0:
    continue; # skip invalid depth
# backproject to 3D point (3,1)
temp = K@(R@K_inv @ np.array([x, y, 1])*d + t)
w = np.array([temp[0]/temp[2], temp[1]/temp[2]]) # [x/z, y/z]
# TODO project warped points onto image
xImg[y, x] = w[0]
yImg[y, x] = w[1]
```

Then the 2nd image is used to interpolate the intensity value given the calculated coordinates to archive new image and calculate the residual finally:

```
coords = np.array([yImg.ravel(), xImg.ravel()])
I_warped = map_coordinates(I, coords, order=1, mode='nearest')
I_warped = I_warped.reshape(H_ref, W_ref)
# --- 5. Photometric residual
err = IRef - I_warped
return err.reshape(-1, 1)
```

The residual contains intensity values and we can visualize as an image as below:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from downscale import downscale
# use cv2.imread to read grayscale images as float
# because map_coordinates requires float inputs
IRef = cv2.imread("rgb/1305031102.175304.png", cv2.IMREAD_GRAYSCALE).astype(float)
DRef = cv2.imread("depth/1305031102.160407.png", cv2.IMREAD_GRAYSCALE).astype(float)
I = cv2.imread("rgb/1305031102.275326.png", cv2.IMREAD_GRAYSCALE).astype(float) # new image
# camera intrinsics
K = np.array([[525.0, 0.0, 319.5],
              [0.0, 525.0, 239.5],
              [0.0, 0.0, 1.0]])
# camera pose 6D vector
xi = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]) # small rotation around z-axis
err = calcErr(IRef, DRef, I, xi, K)
#visualize error as image
err_image = err.reshape(IRef.shape)
plt.imshow(err_image, cmap='gray')
plt.title('Photometric Error Image')
plt.axis('off')
plt.show()
```

We have the following figure

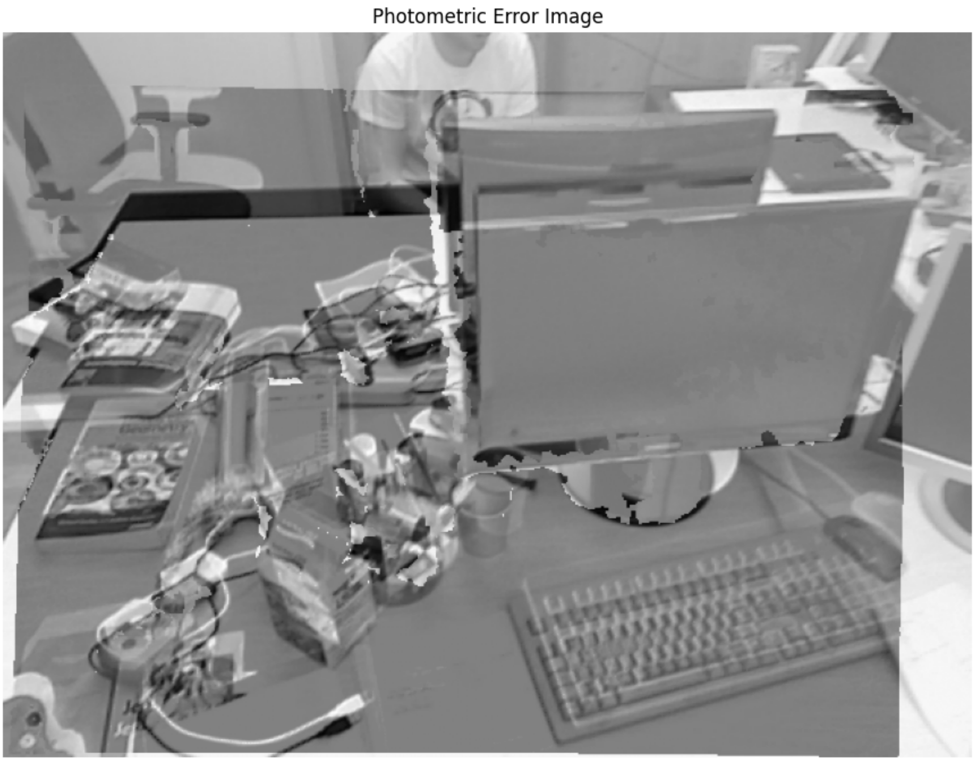


Figure 3: A calculated residual image

3.

The function to calculate the numeric derivative of the residual is implemented as belows:

```
def deriveErrNumeric(IRef, DRef, I, xi, K):
# calculate numeric derivative (SLOW!!!)

# compute residuals for xi
residual_xi = calcErr(IRef,DRef,I,xi,K)

H_I, W_I = I.shape[0], IRef.shape[1] # resolution of the reference image, heght and width
num_of_pixels = H_I * W_I
# initialize Jacobian
Jac = np.zeros((num_of_pixels, 6))

# compute Jacobian numerically
eps = 1e-6
for j in range(1, 7):
    epsVec = np.zeros((6, 1))
    epsVec[j-1] = eps
    # multiply epsilon from left onto the current estimate.
    xiPerm =  se3Log(se3Exp(epsVec) @ se3Exp(xi))
    # TODO compute respective column of the Jacobian
    # (hint: difference between residuals of xi and xiPerm)
    residual_perm = calcErr(IRef, DRef, I, xiPerm, K)
    Jac[:, j-1] = (residual_perm - residual_xi).flatten() / eps
return Jac, residual_xi
```

We use the calcErr() function implemented previously. Here we explain the main calculation in the function. The following line

```
xiPerm =  se3Log(se3Exp(epsVec) @ se3Exp(xi))
```

corresponds to:

$$\xi_{permutation} = \epsilon \mathbf{e}_1 o \xi$$

and

```
residual_perm = calcErr(IRef, DRef, I, xiPerm, K)
```

calculate $r(\epsilon \mathbf{e}_1 o \xi)$. Finally, the corresponding column of the Jacobian matrix is calculated as:

```python
Jac[:, j-1] = (residual_perm - residual_xi).flatten() / eps
```

which corresponds to:

$$J = \frac{r(\epsilon \mathbf{e}_1 \text{o} \xi) - r(\xi)}{\epsilon}$$

4.

The implementation of the Gauss-Newton step is referred to slide number 10, particularly on the calculation of the updates δ_ξ:

$$\delta_\xi = -(J_r^T J_r)^{-1} J_r^T r_0$$

and then update the solution:

$$\xi^{k+1} = \delta_\xi \text{o} \xi^k$$

Those calculations are captured in the snip code below:

```python
# ----------------------------
# Gauss{Newton step
# ----------------------------
H = Jac.T @ Jac
b = Jac.T @ residual
delta_xi = -np.linalg.solve(H, b) # solve for delta
# ----------------------------
# delta_xiate pose
# xi = se3Log(se3Exp(delta_xi) * se3Exp(xi))
# ----------------------------
lastXi = xi.copy()
# update solution
xi = se3Log(se3Exp(delta_xi.flatten()) @ se3Exp(xi.flatten())).reshape(6, 1)
```

The capture below shows the printed result when we run the doAlignment program:

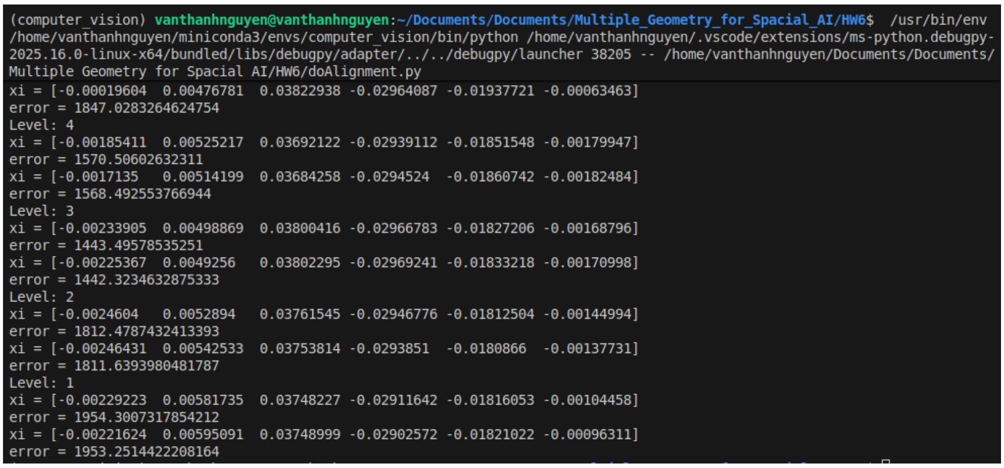


Figure 4: Estimation result of the campera pose

We see that the final result for the level-1 image is:

$$\xi = \begin{pmatrix} -0.00221624 & 0.00595091 & 0.03748999 & -0.02902572 & -0.01821022 & -0.00096311 \end{pmatrix}$$

5.

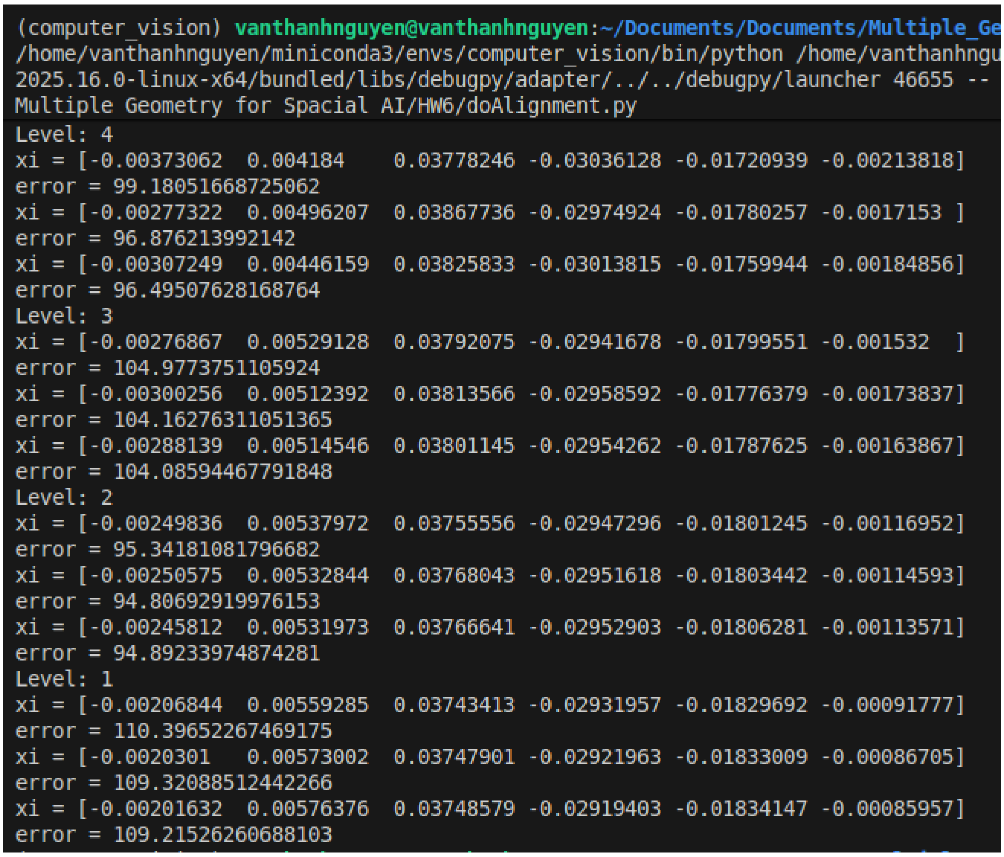
We briefly explain the calculation procedure for the analytical Jacobian.

- Calculate image gradient

Given a pixel at (x, y), the image gradient at that point is calculated as follows:

$$\frac{\partial I}{\partial x}(x, y) = \frac{I(x+1, y) - I(x-1, y)}{2}$$
$$\frac{\partial I}{\partial y}(x, y) = \frac{I(x, y+1) - I(x, y-1)}{2}$$

Therefore, we have following snip code:



```
(computer_vision) vanthanhnguyen@vanthanhnguyen:~/Documents/Documents/Multiple_Ge
/home/vanthanhnguyen/miniconda3/envs/computer_vision/bin/python /home/vanthanhngu
2025.16.0-linux-x64/bundled/libs/debugpy/adapter/../../debugpy/launcher 46655 --
Multiple Geometry for Spacial AI/HW6/doAlignment.py
Level: 4
xi = [-0.00373062  0.004184    0.03778246 -0.03036128 -0.01720939 -0.00213818]
error = 99.18051668725062
xi = [-0.00277322  0.00496207  0.03867736 -0.02974924 -0.01780257 -0.0017153 ]
error = 96.876213992142
xi = [-0.00307249  0.00446159  0.03825833 -0.03013815 -0.01759944 -0.00184856]
error = 96.49507628168764
Level: 3
xi = [-0.00276867  0.00529128  0.03792075 -0.02941678 -0.01799551 -0.001532   ]
error = 104.9773751105924
xi = [-0.00300256  0.00512392  0.03813566 -0.02958592 -0.01776379 -0.00173837]
error = 104.16276311051365
xi = [-0.00288139  0.00514546  0.03801145 -0.02954262 -0.01787625 -0.00163867]
error = 104.08594467791848
Level: 2
xi = [-0.00249836  0.00537972  0.03755556 -0.02947296 -0.01801245 -0.00116952]
error = 95.34181081796682
xi = [-0.00250575  0.00532844  0.03768043 -0.02951618 -0.01803442 -0.00114593]
error = 94.80692919976153
xi = [-0.00245812  0.00531973  0.03766641 -0.02952903 -0.01806281 -0.00113571]
error = 94.89233974874281
Level: 1
xi = [-0.00206844  0.00559285  0.03743413 -0.02931957 -0.01829692 -0.00091777]
error = 110.39652267469175
xi = [-0.0020301   0.00573002  0.03747901 -0.02921963 -0.01833009 -0.00086705]
error = 109.32088512442266
xi = [-0.00201632  0.00576376  0.03748579 -0.02919403 -0.01834147 -0.00085957]
error = 109.21526260688103
```

Figure 5: Camera estimation result using analytic Jacobian

```
# ========================
# Image gradients (central differences)
# ========================
dxI = np.zeros_like(I)
dyI = np.zeros_like(I)
dxI[:, 1:-1] = 0.5 * (I[:, 2:] - I[:, :-2]) # exclude 1st and last column
dyI[1:-1, :] = 0.5 * (I[2:, :] - I[:-2, :]) # exclude 1st and last row
```

- Calculate Jacobian

The Jacobian is calculated as in slide 11:

$$\frac{\partial r_i\left(\boldsymbol{\epsilon} \circ \boldsymbol{\xi}^{(k)}\right)}{\partial \boldsymbol{\epsilon}}\Bigg|_{\boldsymbol{\epsilon}=0} = -\frac{1}{z'}\left(\nabla I_x f_x \quad \nabla I_y f_y\right)\begin{pmatrix} 1 & 0 & -\frac{x'}{z'} & -\frac{x'y'}{z'} & \left(z' + \frac{x'^2}{z'}\right) & -y' \\ 0 & 1 & -\frac{y'}{z'} & -\left(z' + \frac{y'^2}{z'}\right) & \frac{x'y'}{z'} & x' \end{pmatrix}$$

Therefore, we have following snip code:

```
# ========================
# Analytic Jacobian (Kerl Eq. 4.14)
# ========================
Jac = np.zeros((N, 6))

inv_z = 1.0 / zp
inv_z2 = inv_z ** 2

Jac[:, 0] = dxInterp * inv_z
Jac[:, 1] = dyInterp * inv_z
Jac[:, 2] = -(dxInterp * xp + dyInterp * yp) * inv_z2

Jac[:, 3] = -(dxInterp * xp * yp) * inv_z2 - dyInterp * (1 + (yp * inv_z) ** 2)
Jac[:, 4] = dxInterp * (1 + (xp * inv_z) ** 2) + dyInterp * xp * yp * inv_z2
Jac[:, 5] = -(dxInterp * yp - dyInterp * xp) * inv_z
```

The final result as we run the doAlingment program is shown as below: As we see, the final estimation is:

$$\xi = \begin{pmatrix} -0.00206844 & 0.00559285 & 0.03743413 & -0.02931957 & -0.01829692 & -0.00091777 \end{pmatrix}$$

Appendix

Source code:
https://github.com/ThanhNV-Robotics/HW6_Multiple_Geomtry_for_SpatialAI