```
import numpy as np
from numpy.linalg import inv
import scipy.io
from scipy.linalg import block_diag, cholesky, solve_triangular, lu_factor,
lu_solve
from scipy.spatial.transform import Rotation
from tqdm import tqdm
import matplotlib.pyplot as plt
import pylgmath.se3.operations as lg
import pylgmath.so3.operations as l
import sys

# Load MATLAB file
mat_data = scipy.io.loadmat('dataset3.mat')

# Access data in the dictionary
# The keys in the dictionary correspond to variable names in the MATLAB file
theta_vk_i = mat_data['theta_vk_i']
r_i_vk_i = mat_data['r_i_vk_i']
t = mat_data['t']
w_vk_vk_i = mat_data['w_vk_vk_i']
w_var = mat_data['w_var']
v_vk_vk_i = mat_data['v_vk_vk_i']
v_var = mat_data['v_var']
rho_i_pj_i = mat_data['rho_i_pj_i']
y_k_j = mat_data['y_k_j']
y_var = mat_data['y_var']
C_c_v = mat_data['C_c_v']
rho_v_c_v = mat_data['rho_v_c_v']
fu = mat_data['fu'].item()
fv = mat_data['fv'].item()
cu = mat_data['cu'].item()
cv = mat_data['cv'].item()
b = mat_data['b'].item()

# Question 4
M_k = np.sum((y_k_j[0,:,:] != -1), axis=1)
t_k = t[0,:]
gt3_mask = M_k>=3
lt3_mask = M_k<3
M_k_3 = M_k[gt3_mask]
M_k_lt3 = M_k[lt3_mask]
t_k_3 = t_k[gt3_mask]
t_k_lt3 = t_k[lt3_mask]
plt.figure()
plt.scatter(t_k_lt3, M_k_lt3, color='red')
plt.scatter(t_k_3, M_k_3, color='green')
plt.title("Number of Visible Landmarks Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Number of Visible Landmarks")
```

```
plt.show()

# Question 5a
# Batch
def batch(k1,k2,T_check_k1):

    if np.all(np.eye(4) == T_check_k1):
        rot_init = theta_vk_i[:,k1]
        x_init = r_i_vk_i[:,k1]
        C_init = Rotation.from_rotvec(rot_init)

        T_check_k1_ = np.eye(4)
        T_check_k1_[:3, :3] = C_init.as_matrix()
        T_check_k1_[:3, 3] = x_init
    else:
        T_check_k1_ = T_check_k1.copy()

    D = np.array([[1,0,0],[0,1,0],[0,0,1],[0,0,0]])
    t_ = np.array([[0],[0],[0],[1]])
    xi_var = np.vstack((v_var,w_var))
    Qk = np.diag(xi_var.reshape(6,))
    Rjk_inv = inv(np.diag(y_var.reshape(4,)))
    Qks = np.repeat(Qk[np.newaxis, :], (k2-k1), axis = 0)

    T_op_init = np.eye(4)
    x_op = np.repeat(T_op_init[np.newaxis, :], (k2-k1+1), axis = 0)
    xi_k = np.vstack([v_vk_vk_i[:,k1+1:k2+1],w_vk_vk_i[:,k1+1:k2+1]])
    t_k1_k2 = t[0,k1:(k2+1)]
    delta_tk = t_k1_k2[1:] - t_k1_k2[:-1]

    Qks = np.transpose(np.transpose(Qks, axes=(1,2,0))*delta_tk, axes=(2,0,1))
    Qks_inv = inv(Qks)

    xi_k = xi_k.T[:,:,np.newaxis]
    delta_xi_k = np.transpose(np.transpose(xi_k, axes=(1,2,0))*delta_tk,
axes=(2,0,1))
    Xi_k = lg.vec2tran(delta_xi_k)

    for iteration in tqdm(range(10)):
        e_v0_xop = lg.tran2vec(T_check_k1_@inv(x_op[0,:,:]))
        E_0 = lg.vec2jacinv(-e_v0_xop)
        x_op_k = x_op[1:,:,:]
        x_op_k_1 = x_op[:-1,:,:]
        e_v_xop = lg.tran2vec(Xi_k@x_op_k_1@inv(x_op_k))
        E_k = lg.vec2jacinv(-e_v_xop)
        F_k_1 = -E_k@lg.tranAd(x_op_k@inv(x_op_k_1))
        y_mask = (y_k_j[0,k1:(k2+1),:] != -1)
        G = []
        R_inv = []
        e_y_xop = []
```

```
        for k in range(k2-k1+1):
            T_op = x_op[k,:,:]
            G_k = []
            e_y_k_xop = []
            for j in range(rho_i_pj_i.shape[1]):
                if y_mask[k,j] == True:
                    rho_i = rho_i_pj_i[:,j].reshape(3,1)
                    term_1 = -(D.T)@T_op@D@(D.T)@circledot(T_op@t_)
                    term_2 = (D.T)@circledot(T_op@D@rho_i)
                    term_3 = -(D.T)@circledot(T_op@D@(D.T)@T_op@t_)
                    P_ck = C_c_v@(term_1+term_2+term_3)

                    def f(T_):
                        p_T = C_c_v@(((D.T)@T_@D@(rho_i-((D.T)@T_@t_))) -
rho_v_c_v)
                        g_p_T = g(p_T)
                        return g_p_T

                    # Used numerical approximation of Jacobian since it seems to
work better
                    def numerical_jacobian(func, T_num, epsilon):

                        jac = np.zeros((4, 6))
                        epsilon_zero = np.zeros((6,1))

                        for col in range(6):
                            T_perturbed = T_num.copy()
                            epsilon_preturbed = epsilon_zero.copy()
                            epsilon_i = epsilon[col,0]
                            epsilon_preturbed[col,0] = epsilon_i
                            T_perturbed = lg.vec2tran(epsilon_preturbed)@T_num

                            f_perturbed_val = func(T_perturbed)
                            f_val = func(T_num)
                            jac[:, col] = ((f_perturbed_val - f_val) /
epsilon_i).reshape(4,)
                        return jac

                    if iteration == 0:
                        epsilon_it_0 = 1e-6*np.ones((6,1))
                        G_jk_numerical = numerical_jacobian(f, T_op, epsilon_it_0)
                    else:
                        G_jk_numerical = numerical_jacobian(f, T_op, epsilon_star)

                    p_Top = C_c_v@(((D.T)@T_op@D@(rho_i-((D.T)@T_op@t_))) -
rho_v_c_v)
                    g_p = g(p_Top)
                    S_p = S(p_Top)
                    G_jk = S_p@P_ck
```

```
                G_k.append(G_jk_numerical)
                e_y_jk_xop = y_k_j[:,k,j].reshape(4,1) - g_p
                e_y_k_xop.append(e_y_jk_xop)
                R_inv.append(Rjk_inv)

        if len(G_k) != 0:
            G_k = np.vstack(G_k)
        else:
            G_k = np.empty((0,6))

        G.append(G_k)

        if len(e_y_k_xop) != 0:
            e_y_k_xop = np.vstack(e_y_k_xop)
            e_y_xop.append(e_y_k_xop)

    if len(e_y_xop) != 0:
        e_y_xop = np.vstack(e_y_xop)

    E_k_list = [E_0] + E_k.tolist()
    E_matrix = block_diag(*E_k_list)
    F_matrix = block_diag(*(F_k_1.tolist()))
    F_matrix = np.block([[np.zeros((6,F_matrix.shape[0])), np.zeros((6,6))],
[F_matrix, np.zeros((F_matrix.shape[0],6))]])
    G_matrix = block_diag(*G)
    H_top = E_matrix + F_matrix
    H = np.vstack((H_top,G_matrix))

    e_v_xop_list = [e_v0_xop] + e_v_xop.tolist()
    e_v_xop = np.vstack(e_v_xop_list)
    if len(e_y_xop) != 0:
        e_xop = np.vstack((e_v_xop,e_y_xop))
    else:
        e_xop = e_v_xop

    P_check_k1_inv = inv((t[0,k1] - t[0,(k1-1)])*Qk)
    Qk_inv_list = [P_check_k1_inv] + Qks_inv.tolist()
    W_inv_list = Qk_inv_list + R_inv
    W_inv = block_diag(*W_inv_list)

    A = H.T@W_inv@H
    b_vec = H.T@W_inv@e_xop

    L = cholesky(A,lower=True)
    d_vec = solve_triangular(L,b_vec,lower=True)
    delta_x_star = solve_triangular(L.T,d_vec,lower=False)
    epsilon_star = delta_x_star.reshape((-1,6,1))
    x_op = lg.vec2tran(epsilon_star)@x_op

lu, piv = lu_factor(A)
```

```
    A_inv = lu_solve((lu, piv), np.eye(A.shape[0]))
    cov = np.diag(A_inv)

    return x_op, cov

def circledot(v):
    D = np.array([[1,0,0],[0,1,0],[0,0,1],[0,0,0]])
    neg_rho_hat = -l.hat((D.T)@v)
    I = np.ones_like(neg_rho_hat)
    Z = np.zeros((1,6))
    inter = np.concatenate((I,neg_rho_hat), axis=1)
    result = np.concatenate((inter,Z), axis=0)
    return result

def S(p):
    return np.array([[fu/p[2,0], 0, -fu*p[0,0]/(p[2,0]**2)],[0, fv/p[2,0],
-fv*p[1,0]/(p[2,0]**2)],[fu/p[2,0],0,(-fu*p[0,0] + fu*b)/(p[2,0]**2)],[0,
fv/p[2,0], -fv*p[1,0]/(p[2,0]**2)]])

def g(p):
    return np.array([[fu*p[0,0]/p[2,0] + cu],[fv*p[1,0]/p[2,0] +
cv],[fu*(p[0,0]-b)/p[2,0] + cu],[fv*p[1,0]/p[2,0] + cv]])

def plot_graphs(k1_val,k2_val,traj_pred,cov,method,kappa):
    if method == "Sliding_Window":
        tag = f"{method}_kappa_{kappa}"
    else:
        tag = f"{method}"

    x_diff = traj_pred[:,0,3] - r_i_vk_i[0,k1_val:(k2_val+1)]
    y_diff = traj_pred[:,1,3] - r_i_vk_i[1,k1_val:(k2_val+1)]
    z_diff = traj_pred[:,2,3] - r_i_vk_i[2,k1_val:(k2_val+1)]

    rot_vec = theta_vk_i[:,k1_val:(k2_val+1)]
    C_true = Rotation.from_rotvec(rot_vec.T)

    identity = np.eye(3)
    identities = np.repeat(identity[np.newaxis, :], (k2_val-k1_val+1), axis = 0)
    theta_diff_crosses = identities -
traj_pred[:,:3,:3]@np.transpose((C_true.as_matrix()),(0,2,1))
    theta_x_diff = theta_diff_crosses[:,1,2]
    theta_y_diff = theta_diff_crosses[:,2,0]
    theta_z_diff = theta_diff_crosses[:,0,1]

    stds = np.sqrt(cov)
    epsilon_stds = stds.reshape(-1,6,1)
    T_stds = lg.vec2tran(epsilon_stds)
    x_stds = T_stds[:,0,3]
    y_stds = T_stds[:,1,3]
    z_stds = T_stds[:,2,3]
```

```
C_stds = T_stds[:,:3,:3]
r_vec_stds = Rotation.from_matrix(C_stds).as_rotvec()
theta_x_stds = r_vec_stds[:,0]
theta_y_stds = r_vec_stds[:,1]
theta_z_stds = r_vec_stds[:,2]

t_k1_k2 = t[0,k1_val:(k2_val+1)].T

plt.figure()
plt.plot(t_k1_k2,x_diff,label = "delta r_x")
plt.plot(t_k1_k2,3*x_stds,label = "+3 sigma r_x")
plt.plot(t_k1_k2,-3*x_stds,label = "-3 sigma r_x")
plt.title("Difference in X Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_x_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,y_diff,label = "delta r_y")
plt.plot(t_k1_k2,3*y_stds,label = "+3 sigma r_y")
plt.plot(t_k1_k2,-3*y_stds,label = "-3 sigma r_y")
plt.title("Difference in Y Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_y_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,z_diff,label = "delta r_z")
plt.plot(t_k1_k2,3*z_stds,label = "+3 sigma r_z")
plt.plot(t_k1_k2,-3*z_stds,label = "-3 sigma r_z")
plt.title("Difference in Z Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_z_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,theta_x_diff,label = "delta theta_x")
plt.plot(t_k1_k2,3*theta_x_stds,label = "+3 sigma theta_x")
plt.plot(t_k1_k2,-3*theta_x_stds,label = "-3 sigma theta_x")
plt.title("Difference in Theta X Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Angle (rad)")
file_name = f"k1_{k1_val}_k2_{k2_val}_theta_x_{tag}"
plt.savefig(file_name)

plt.figure()
```

```
    plt.plot(t_k1_k2,theta_y_diff,label = "delta theta_y")
    plt.plot(t_k1_k2,3*theta_y_stds,label = "+3 sigma theta_y")
    plt.plot(t_k1_k2,-3*theta_y_stds,label = "-3 sigma theta_y")
    plt.title("Difference in Theta Y Over Time")
    plt.xlabel("Time (s)")
    plt.ylabel("Angle (rad)")
    file_name = f"k1_{k1_val}_k2_{k2_val}_theta_y_{tag}"
    plt.savefig(file_name)

    plt.figure()
    plt.plot(t_k1_k2,theta_z_diff,label = "delta theta_z")
    plt.plot(t_k1_k2,3*theta_z_stds,label = "+3 sigma theta_z")
    plt.plot(t_k1_k2,-3*theta_z_stds,label = "-3 sigma theta_z")
    plt.title("Difference in Theta Z Over Time")
    plt.xlabel("Time (s)")
    plt.ylabel("Angle (rad)")
    file_name = f"k1_{k1_val}_k2_{k2_val}_theta_z_{tag}"
    plt.savefig(file_name)

    return

#Question 5a Batch
k1_val = 1215
k2_val = 1714
traj_pred, cov = batch(k1_val,k2_val,np.eye(4))
plot_graphs(k1_val,k2_val,traj_pred,cov,"Batch",None)

#Question 5b Sliding Window

def sliding_window(k1_val, k2_val, kappa):
    I = np.eye(4)
    traj_final = np.repeat(I[np.newaxis, :], (k2_val-k1_val+1), axis = 0)
    cov_final = np.ones((6*(k2_val-k1_val+1),))
    for i in range(k2_val-k1_val+1):
        k1_slide = k1_val + i
        k2_slide = k1_slide + kappa
        if k1_slide == k1_val:
            traj_pred, cov = batch(k1_slide,k2_slide,np.eye(4))
            traj_i = traj_pred[i,:,:]
            traj_final[i,:,:] = traj_i
            cov_final[i:i+6] = cov[:6]
        else:
            batch(k1_slide,k2_slide,traj_i)
    plot_graphs(k1_val,k2_val,traj_final,cov_final,"Sliding_Window",kappa)
    return

k1_val = 1215
k2_val = 1714
kappa = 50
```

```
sliding_window(k1_val, k2_val, kappa)

#Question 5c Sliding Window

k1_val = 1215
k2_val = 1714
kappa = 10

sliding_window(k1_val, k2_val, kappa)
```