```python
################################################################################
# IMPORTS
################################################################################
import numpy as np
##############################################################################
# Class definitions
##############################################################################
class kalman_filter():
    def __init__(self, init_state, init_est_err, Q, R, F, G, H, y_k, num_step):
        self.x_0_hat = init_state
        self.P_0_hat = init_est_err
        self.Q = Q
        self.R = R
        self.F = F
        self.G = G
        self.H = H
        self.y_k = y_k
        self.num_step = num_step
        self.P_f = None
        self.P_b = None
        self.x_f = None
        self.x_b = None
        self.x_fb = None
        self.K_f_val = None
        self.K_b_val = None
        self.K_fb_val = None
        self.I = None
        self.R = np.reshape(self.R, (self.y_k.shape[0], 1))

    ############################################################################
    # This function estimates states using forward Kalman filter
    ############################################################################
    def run_kalman(self, P_0_hat=None, x_0_hat=None):
        if (None is P_0_hat):
            P_0_hat = self.P_0_hat
        if (None is x_0_hat):
            x_0_hat = self.x_0_hat
        self.P_f = np.zeros((P_0_hat.shape[0], P_0_hat.shape[1], self.num_step + 1))
        self.P_f[:, :, 0] = P_0_hat
        self.x_f = np.zeros((x_0_hat.shape[0], self.num_step + 1))
        self.x_f[:, 0] = x_0_hat[:, 0]
        self.K_f_val = np.zeros((x_0_hat.shape[0], self.num_step + 1))
        self.I = np.identity(x_0_hat.shape[0])
        for step in range(self.num_step):
            P_f_minus = self.F @ self.P_f[:, :, step] @ self.F.T + self.Q
            K_f = P_f_minus @ self.H.T @ np.linalg.inv(self.H @ P_f_minus @ self.H.T + self.R)
            x_f_minus = self.F @ self.x_f[:, step]
            self.x_f[:, step + 1] = x_f_minus + K_f @ (self.y_k[:, step + 1] - self.H @ x_f_minus)
            self.P_f[:, :, step + 1] = (self.I - K_f @ self.H) @ P_f_minus @ (self.I - K_f @ self.H).T\
                                      + K_f @ self.R @ K_f.T
            # Store the kalman gain value
            self.K_f_val[:, step + 1] = K_f[:, 0]

    def get_predicted_state(self, state_id):
        return self.x_f[state_id, :]
    def get_kalman_gains(self):
        return self.K_f_val
    def get_est_error_cov(self):
        return self.P_f
    def calculate_theoretical_cov(self, x_0, P_0, num_step=None):
        if (num_step is not None):
            self.num_step = num_step
        self.run_kalman(P_0_hat=P_0, x_0_hat=x_0)
        return self.P_f


    ################################################################################
    # This function estimates states using backward Kalman filter
    # Used for smoothers
    ################################################################################
    def run_backward_kalman(self, num_meas, num_stop):
        self.I_bk = np.zeros((self.P_0_hat.shape[0], self.P_0_hat.shape[1], num_meas-num_stop + 1))
        self.I_bk[:, :, -1] += 1e-5 * np.ones(self.P_0_hat.shape)
        self.P_b = np.zeros((self.P_0_hat.shape[0], self.P_0_hat.shape[1], num_meas-num_stop + 1))
        self.P_b[:, :, -1] *= 1e+5
        self.x_b = np.zeros((self.x_0_hat.shape[0], num_meas - num_stop + 1))
        I_bk_plus = np.zeros(self.P_0_hat.shape)
        s_k_minus = np.zeros(self.x_0_hat.shape)
        self.s_k = np.zeros((self.x_0_hat.shape[0], num_meas-num_stop + 1))
        self.K_b_val = np.zeros((self.x_0_hat.shape[0], num_meas-num_stop + 1))
        R_inv = np.linalg.inv(self.R)
        Q_inv = np.linalg.inv(self.Q + 1e-5 * np.ones(self.Q.shape))
        for step in range(num_meas, num_stop, -1):
            I_bk_plus = self.I_bk[:, :, step] + self.H.T @ R_inv @ self.H
            s_k_plus = self.s_k[:, step] + self.H.T @ R_inv @ self.y_k[:, step]
            self.I_bk[:, :, step-1] = self.F.T @ np.linalg.inv(np.linalg.inv(I_bk_plus) + self.Q) @ self.F
            self.s_k[:, step-1] = self.I_bk[:, :, step-1] @ np.linalg.inv(self.F) @ np.linalg.inv(I_bk_plus) @ s_k_plus
            self.x_b[:, step-1] = np.linalg.inv(self.I_bk[:, :, step-1]) @ self.s_k[:, step-1]
            # Store P_b minus values
            self.P_b[:, :, step-1] = np.linalg.inv(self.I_bk[:, :, step-1])
            # Store backward Kalman gain
            self.K_b_val[:, step-1] = (np.linalg.inv(I_bk_plus) @ self.H.T @ R_inv)[:, 0]
        # Last step
        self.I_bk[:, :, num_stop] = Q_inv - Q_inv @ np.linalg.inv(self.F) @\
                    np.linalg.inv(I_bk_plus + np.linalg.inv(self.F).T @ Q_inv @ np.linalg.inv(self.F))\
                    @ np.linalg.inv(self.F).T @ Q_inv
        self.s_k[:, num_stop] = self.I_bk[:, :, num_stop] @ np.linalg.inv(self.F) @ np.linalg.inv(I_bk_plus) @ s_k_plus
        self.x_b[:, num_stop] = np.linalg.inv(self.I_bk[:, :, num_stop]) @ self.s_k[:, num_stop]
        # Store P_b minus values
        self.P_b[:, :, num_stop] = np.linalg.inv(self.I_bk[:, :, num_stop])


    ################################################################################
    # This function run smoother using forward-backward smoothing algo.
    # This calls forward and backward Kalman filter function
    ################################################################################
    def run_fb_smoother(self, num_meas, est_idx):
        if (est_idx < 0):
            num_stop = 0
        else:
            num_stop = est_idx
        self.num_step = num_meas
        self.run_kalman()
        self.run_backward_kalman(num_meas, num_stop)
        self.x_fb = np.zeros(self.x_f.shape)
        self.x_fb[:, 0] = self.x_0_hat[:, 0]
        self.P_fb = np.zeros((self.P_0_hat.shape[0], self.P_0_hat.shape[1], self.num_step + 1))
        self.P_fb[:, :, 0] = self.P_0_hat
        self.K_fb_val = np.zeros(self.K_f_val.shape)
        for idx in range(num_meas):
            K_smoothed_f = self.P_b[:, :, idx] @ np.linalg.inv(self.P_f[:, :, idx] + self.P_b[:, :, idx])
            self.x_fb[:, idx] = K_smoothed_f @ self.x_f[:, idx] + (self.I - K_smoothed_f) @ self.x_b[:, idx]
            self.P_fb[:, :, idx] = np.linalg.inv(np.linalg.inv(self.P_f[:, :, idx]) + np.linalg.inv(self.P_b[:, :, idx]))
            self.K_fb_val[:, idx+1] = K_smoothed_f[0, :]

    def get_predicted_state_fb_sm(self, state_id):
        return self.x_fb[state_id, :]

    def get_kalman_gains_fb_sm(self):
        return self.K_fb_val

    def get_est_error_cov_fb_sm(self):
        return self.P_fb

```