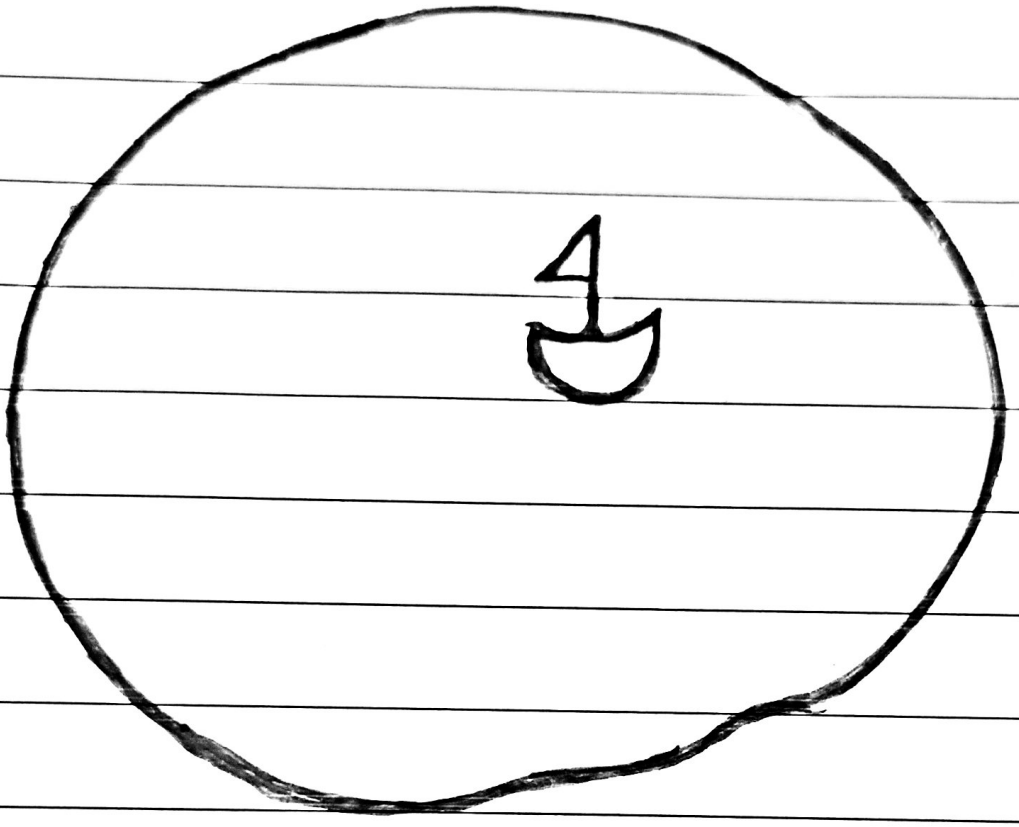
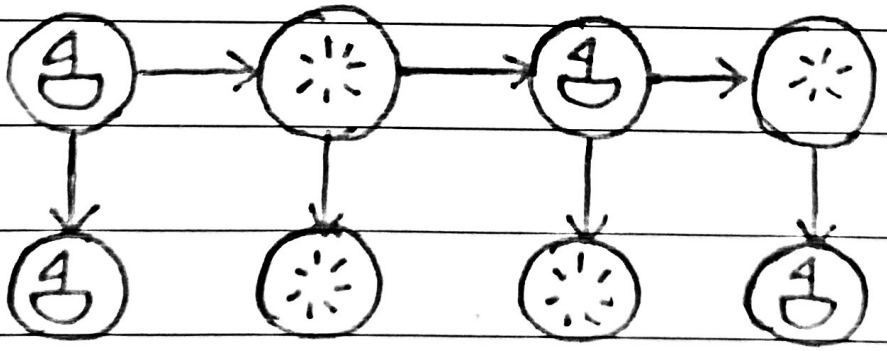
Bayes

1a)





$P_1(x_{t+1}|x_t) = P_S$ | $P_2(x_{t+1}|x_t) = P_E$

$P_1(x_t) = r_k$ | $P_2(x_t) = (1 - r_k)$

$\Downarrow$

$r_{k+1|k} = P_S \cdot r_k + P_E(1 - r_k)$

1b) $c_{k+1} = p(x_{k+1} \mid z_{1:k+1}) = \eta\, p(z_{k+1} \mid x_{k+1}) p(x_{k+1} \mid z_{1:k})$

$= \eta \cdot c_{k+1|k}\, p(z_{k+1} \mid x_{k+1}) = \eta\, c_{k+1|k} (P_D + (1-P_D) P_{FA})$

where η is the normalizing term.

2. $x_k = [p_k^T \; u_k^T]^T$

CV model: $x_{k+1} = \begin{bmatrix} p_{k+1} \\ u_{k+1} \end{bmatrix} = F x_k + v_k$

$v_k \sim \mathcal{N}(0, Q)$, $z_k = [I_2 \; 0_2] x_k + w_k = p_k + w_k$

$w_k \sim \mathcal{N}(0, R) = \mathcal{N}(0, \sigma_r^2 I_2)$

$$\hat{x}_1 = \begin{bmatrix} \hat{p}_1 \\ \hat{u}_1 \end{bmatrix} = \begin{bmatrix} k_{p1} & k_{p0} \\ k_{u1} & k_{u0} \end{bmatrix} \begin{bmatrix} z_1 \\ z_0 \end{bmatrix}$$

a) $$F = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} T^3/3 & 0 & T^2/2 & 0 \\ 0 & T^3/3 & 0 & T^2/2 \\ T^2/2 & 0 & T & 0 \\ 0 & T^2/2 & 0 & T \end{bmatrix} \sigma_a^2$$

$$z_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_k^T \\ u_k^T \end{bmatrix}$$

CV model: $\dot{x} = Ax + Gn$, $A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, $G = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$

$\bar{A} = e^{AT}$

$n \sim \mathcal{N}(0, D\delta(t-\pi))$, $D = \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_a^2 \end{bmatrix}$

$z_1 = H x_1 + w_1$, $z_0 = H x_0 + w_0$

$x_1 = \bar{A} x_0 + \bar{G} n \Rightarrow x_0 = \bar{A}^{-1}(x_1 - \bar{G} n)$, $x_1 = \begin{bmatrix} p_1 \\ z_1 \end{bmatrix}$

$z_0 = H \bar{A}^{-1}(x_1 - \bar{G} n) + w_0$

2.

b) $\hat{x}_1 = \begin{bmatrix} \hat{p}_1 \\ \hat{u}_1 \end{bmatrix} = \begin{bmatrix} K_{p1} & K_{p0} \\ K_{u1} & K_{u0} \end{bmatrix} \begin{bmatrix} z_1 \\ z_0 \end{bmatrix}$

$E[\hat{x}_1] = K \cdot E\begin{bmatrix} z_1 \\ z_0 \end{bmatrix}$

$= K \begin{bmatrix} E[z_1] \\ E[z_0] \end{bmatrix} = \begin{bmatrix} K_{p1}E[z_1] + K_{p0}E[z_0] \\ K_{u1}E[z_1] + K_{p0}E[z_0] \end{bmatrix}$

and must be $E[\hat{x}_1] = \begin{bmatrix} p_1 \\ u_1 \end{bmatrix}$

$z_1 = p_1 + w_1$, $E[z_1] = p_1$

$E[z_0] = H\bar{A}^{-1}x_1$

$K_{p1}p_1 + K_{p0}H\bar{A}^{-1}x_1 = K_{p1}p_1 + K_{p0}\begin{bmatrix} 1 & 0 & -T & 0 \\ 0 & 1 & 0 & -T \end{bmatrix}\begin{bmatrix} p_1 \\ u_1 \end{bmatrix}$

$= K_{p1}p_1 + K_{p0}\begin{bmatrix} 1 & 0 & -T & 0 \\ 0 & 1 & 0 & -T \end{bmatrix}\begin{bmatrix} x_1 \\ y_1 \\ u_{x1} \\ u_{y1} \end{bmatrix}$

$= K_{p1}p_1 + K_{p0}\begin{bmatrix} x_1 - Tu_{x1} \\ y_1 - Tu_{y1} \end{bmatrix}$

I which must be equal to p_1:

$K_{p1}p_1 + K_{p0}\begin{bmatrix} x_1 - Tu_{x1} \\ y_1 - Tu_{y1} \end{bmatrix} = p_1$

$\Rightarrow \underline{K_{p0} = 0_2}$, $\underline{K_{p1} = I_2}$

The same for u_1:

$K_{u1}p_1 + K_{u0}\begin{bmatrix} x_1 - Tu_{x1} \\ y_1 - Tu_{y1} \end{bmatrix} = \begin{bmatrix} u_{x1} \\ u_{y1} \end{bmatrix}$

$\Rightarrow \underline{K_{u1} = I_2/2}$, $\underline{K_{u0} = -I_2/T}$

2c) $\text{cov}[\hat{x}_1] = E[(\hat{x}_1 - x_1)^T(\hat{x}_1 - x_1)]$

$$= E\left[\left[w_1 \quad \frac{1}{T}(w_1 - w_0 + H\bar{A}^{-1}v_0)\right]\left[w_1 \quad \frac{1}{T}(w_1 - w_0 + H\bar{A}^{-1}v_0)\right]^T\right]$$

$$= E\left[\begin{matrix} w_1 w_1^T & \frac{w_1 w_1^T}{T} \\ \frac{w_1 w_1^T}{T} & \frac{1}{T^2}(w_1 w_1^T + w_0 w_0^T + H\bar{A}^{-1} v_0 v_0^T \bar{A}^{-1^T} H^T) \end{matrix}\right] + \ldots^0$$

$$= \begin{bmatrix} R & R/2 \\ R/2 & \frac{1}{T^2}(R + H\bar{A}^{-1} Q \bar{A}^{-1^T} H^T) \end{bmatrix}$$

d) $\hat{x}_1$ must be Gaussian as it is a linear combination of Gaussian variables. It has mean $\hat{x}$ and covariance $\text{cov}\,\hat{x}$.

e) This is an optimal estimator. Probably good in practice unless we have a system which requires a very precise model.

```python
def f(self,
        x: np.ndarray,
        Ts: float,
      ) -> np.ndarray:
    """
    Calculate the zero noise Ts time units transition from x.

    x[:2] is position, x[2:4] is velocity
    """
    # TODO

    return (np.identity(4) + Ts*np.array([[0, 0, 1, 0],
   [0, 0, 0, 1],
   [0, 0, 0, 0],
   [0, 0, 0, 0]]))@x

def F(self,
        x: np.ndarray,
        Ts: float,
      ) -> np.ndarray:
    """ Calculate the transition function jacobian for Ts time units at x."""
    # TODO
    # Isn't this just f not multiplied by x?
    return (np.identity(4) + Ts*np.array([[0, 0, 1, 0],
   [0, 0, 0, 1],
   [0, 0, 0, 0],
   [0, 0, 0, 0]]))
```

```python
def Q(self,
        x: np.ndarray,
        Ts: float,
      ) -> np.ndarray:
    """
    Calculate the Ts time units transition Covariance.
    """

    # TODO
    # Hint: sigma can be found as self.sigma, see variable declarations
    # Note the @dataclass decorates this class to create an init function that takes
    # sigma as a parameter, among other things.
    return np.array([[Ts**3/3, 0, Ts**2/2, 0],
                     [0, Ts**3/3, 0, Ts**2/2],
                     [Ts**2/2, 0, Ts, 0],
                     [0, Ts**2/2, 0, Ts]])*(self.sigma)**2
```

```python
def h(self,
        x: np.ndarray,
        *,
        sensor_state: Dict[str, Any] = None,
      ) -> np.ndarray:
    """Calculate the noise free measurement location at x in sensor_state."""
    # TODO
    # x[0:2] is position
    # you do not need to care about sensor_state
    return np.array([[1, 0, 0, 0], [0, 1, 0, 0]])@x

def H(self,
        x: np.ndarray,
        *,
        sensor_state: Dict[str, Any] = None,
      ) -> np.ndarray:
    """Calculate the measurement Jacobian matrix at x in sensor_state."""
    # TODO
    # x[0:2] is position
    # you do not need to care about sensor_state
    # if you need the size of the state dimension it is in self.state_dim
    return np.array([[1, 0, 0, 0], [0, 1, 0, 0]])

def R(self,
        x: np.ndarray,
        *,
        sensor_state: Dict[str, Any] = None,
        z: np.ndarray = None,
      ) -> np.ndarray:
    """Calculate the measurement covariance matrix at x in sensor_state having potential
    # TODO
    # you do not need to care about sensor_state
    # sigma is available as self.sigma, and @dataclass makes it available in the init cl
    return self.sigma*np.identity(2)
```

```python
def predict(self,
            ekfstate: GaussParams,
            # The sampling time in units specified by dynamic_model
            Ts: float,
            ) -> GaussParams:
    """Predict the EKF state Ts seconds ahead."""

    x, P = ekfstate  # tuple unpacking

    F = self.dynamic_model.F(x, Ts)
    Q = self.dynamic_model.Q(x, Ts)

    x_pred = self.dynamic_model.f(x, Ts)  # TODO
    print(F.shape)
    print(P.shape)
    print(Q.shape)
    P_pred = F@P@np.transpose(F)+Q  # TODO

    state_pred = GaussParams(x_pred, P_pred)

    return state_pred
```

```python
def innovation_mean(
        self,
        z: np.ndarray,
        ekfstate: GaussParams,
        *,
        sensor_state: Dict[str, Any] = None,
) -> np.ndarray:
    """Calculate the innovation mean for ekfstate at z in sensor_state."""

    x = ekfstate.mean
    print("x = ")
    print(x)
    zbar = self.sensor_model.h(x)  # TODO predicted measurement can I omit the rest
    print("z = ")
    print(zbar)
    v = x - zbar  # TODO the innovation
    return v

def innovation_cov(self,
                   z: np.ndarray,
                   ekfstate: GaussParams,
                   *,
                   sensor_state: Dict[str, Any] = None,
                   ) -> np.ndarray:
    """Calculate the innovation covariance for ekfstate at z in sensorstate."""

    x, P = ekfstate

    H = self.sensor_model.H(x, sensor_state=sensor_state)
    R = self.sensor_model.R(x, sensor_state=sensor_state, z=z)

    S = H@P@np.transpose(H) + R  # TODO the innovation covariance

    return S
```

```python
def innovation(self,
               z: np.ndarray,
               ekfstate: GaussParams,
               *,
               sensor_state: Dict[str, Any] = None,
               ) -> GaussParams:
    """Calculate the innovation for ekfstate at z in sensor_state."""

    # TODO: reuse the above functions for the innovation and its covariance
    v = self.innovation_mean(z, ekfstate, sensor_state=sensor_state) #TODO
    S = self.innovation_cov(z,ekfstate, sensor_state=sensor_state) #TODO

    innovationstate = GaussParams(v, S)

    return innovationstate

def update(self,
           z: np.ndarray,
           ekfstate: GaussParams,
           sensor_state: Dict[str, Any] = None
           ) -> GaussParams:
    """Update ekfstate with z in sensor_state"""

    x, P = ekfstate

    v, S = self.innovation(z, ekfstate, sensor_state=sensor_state)

    H = self.sensor_model.H(x, sensor_state=sensor_state)

    W = P*np.divide(np.transpose(H),S)  # TODO: the kalman gain, Hint: la.solve, la.in

    x_upd = x + W*v  # TODO: the mean update
    P_upd = (np.identity(4) - W*H)*P  # TODO: the covariance update

    ekfstate_upd = GaussParams(x_upd, P_upd)

    return ekfstate_upd
```

```python
def step(self,
         z: np.ndarray,
         ekfstate: GaussParams,
         # sampling time
         Ts: float,
         *,
         sensor_state: Dict[str, Any] = None,
         ) -> GaussParams:
    """Predict ekfstate Ts units ahead and then update this prediction with z in
    # TODO: reuse the above functions
    ekfstate_pred = self.predict(ekfstate, Ts, ) # TODO Is this correct?
    ekfstate_upd = self.update(z, ekfstate, sensor_state=sensor_state)  # TODO
    return ekfstate_upd

def NIS(self,
        z: np.ndarray,
        ekfstate: GaussParams,
        *,
        sensor_state: Dict[str, Any] = None,
        ) -> float:
    """Calculate the normalized innovation squared for ekfstate at z in sensor_s

    v, S = self.innovation(z, ekfstate, sensor_state=sensor_state)

    NIS = np.transpose(v)*np.divide(v, S)  # TODO

    return NIS
```

```python
def NEES(cls,
         ekfstate: GaussParams,
         # The true state to comapare against
         x_true: np.ndarray,
         ) -> float:
    """Calculate the normalized etimation error squared from ekfstate to x_true."""

    x, P = ekfstate

    x_diff = x - x_true  # Optional step
    NEES = np.transpose(x_diff)@np.inv(P)@x_diff  # TODO
    return NEES
```

```python
def loglikelihood(self,
                  z: np.ndarray,
                  ekfstate: GaussParams,
                  sensor_state: Dict[str, Any] = None
                  ) -> float:
    """Calculate the log likelihood of ekfstate at z in sensor_state"""
    # we need this function in IMM, PDA and IMM-PDA exercises
    # not necessary for tuning in EKF exercise
    v, S = self.innovation(z, ekfstate, sensor_state=sensor_state)

    # TODO: log likelihood, Hint: log(N(v, S))) -> NIS, la.slogdet.
    NIS = self.NIS(z, ekfstate, sensor_state=sensor_state)
    ll = -1/2*NIS + self._MLOG2PIby2 + np.log(np.det(S))*(1/2)
    return ll
```

```python
def estimate_sequence(
        self,
        # A sequence of measurements
        Z: Sequence[np.ndarray],
        # the initial KF state to use for either prediction or update (see start_with_prediction)
        init_ekfstate: GaussParams,
        # Time difference between Z's. If start_with_prediction: also diff before the first Z
        Ts: Union[float, Sequence[float]],*,
        # An optional sequence of the sensor states for when Z was recorded
        sensor_state: Optional[Iterable[Optional[Dict[str, Any]]]] = None,
        # sets if Ts should be used for predicting before the first measurement in Z
        start_with_prediction: bool = False,
) -> Tuple[GaussParamList, GaussParamList]:
    """Create estimates for the whole time series of measurements."""
    # sequence length
    K = len(Z)

    # Create and amend the sampling array
    Ts_start_idx = int(not start_with_prediction)
    Ts_arr = np.empty(K)
    Ts_arr[Ts_start_idx:] = Ts
    # Insert a zero time prediction for no prediction equivalence
    if not start_with_prediction:
        Ts_arr[0] = 0

    # Make sure the sensor_state_list actually is a sequence
    sensor_state_seq = sensor_state or [None] * K

    # initialize and allocate
    ekfupd = init_ekfstate
    n = init_ekfstate.mean.shape[0]
    ekfpred_list = GaussParamList.allocate(K, n)
    ekfupd_list = GaussParamList.allocate(K, n)
    # TODO loop over the data and get both the predicted and updated states in the lists
    # the predicted is good to have for evaluation purposes
    # A potential pythonic way of looping through  the data
    for k, (zk, Tsk, ssk) in enumerate(zip(Z, Ts_arr, sensor_state_seq)):
      ekf_pred = self.predict(ekfupd, Tsk)
      ekfpred_list[k] = ekf_pred
      ekfupd = self.update(zk, ekf_pred, ssk)
      ekfupd_list[k] = ekfupd_list[k]
    return ekfpred_list, ekfupd_list
```