1) a) $r_{k+1|k}$

$= P(S_{k+1}) \cdot P(R_k) + P(E_{k+1}) \cdot P(\neg R_k)$

$= P(S_{k+1}) \cdot r_k + P(E_{k+1}) \cdot (1 - r_k)$

$= P_S \cdot r_k + P_E \cdot (1 - r_k)$

Bayes' prediction dep: 2

b) $p(r_{k+1} | D) = \frac{P(r_{k+1} \cap M_{k+1})}{P(M_{k+1})}$

$= \frac{r_{k+1|k} \cdot P_D}{(r_{k+1}) \cdot P_D + (1 - r_{k+1|k}) P_{FA}}$

$= \frac{r_{k+1|k} \cdot P_D}{r_{k+1|k} \cdot P_D + (1 - r_{k+1|k}) \cdot P_{FA}}$

(k)

3) $z_1 = [I_2 \;\; 0]\, x_1 + w_k$

$x(k+1) = F x_k + v_k$

$z_k = [I_2 \;\; 0]\, x_k + w_k$

$$x_{k+1} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_k .$$

$x_k = F^{-1} x_{k+1}$

$z_1 = f(p_1, u_1, v_0, w_0, w_1)$

$$\begin{cases} z_1 = p_1 + w_1 \\ z_0 = p_0 + w_0 \\ \quad = F^{-1}(p_1 + v_0) + w_0 \\ \quad = F^{-1}\begin{bmatrix} p_1 \\ u_1 \end{bmatrix} + v_0 + w_0 \end{cases}$$

b) $E[\hat{x}_k] = x_1$

$$\int_{-N}^{\infty} \hat{x}_k \cdot p(\hat{x}_k)\, d\hat{x}_k = x_1$$

$$E[z_0] = x_0$$

$$x_0 = F^{-1}(x_1 - v_0)$$

$$z_0 = E[z_2\ 0]\, x_0 + w_0$$

$$z_1 = p_1 + w_0$$

2a) $x_0 = F^{-1}(x_1 - v_0)$

$z_0 = [I_2 \;\; 0_2]\, x_0 + w_0$

$z_0 = [I_2 \;\; 0_2]\, F^{-1}(x_1 - v_0) + w_0$

$\leftarrow [z_0, z_1]$

$z_1 = \rho_1 + w_1$

$z_0 = (\rho_1 - T u_1) - [I_2 - T I_2]\, v_0 + w_0$

b) $E[\hat{\rho}_1] = K_{\rho 1} E[z_1] + K_{\rho 0} E[z_0]$

$E[z_1] = E[\rho_1] = \rho_1$

$E[z_0] = [I_2 \;\; 0_2]\, F^{-1} x_1$

Finding F^{-1}

$x_{k+1} = F x_k$

$$x_{k+1} = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_k$$

$x_{k+1\_1} = x_{k\_1} + T x_{k\_3}$

$x_{k+1\_2} = x_{k\_2} + T x_{k\_4}$

$x_{k+1\_3} = x_{k\_3}$

$x_{k+1\_4} = x_{k\_4}$

$$F^{-1} = \begin{bmatrix} 1 & 0 & -T & 0 \\ 0 & 1 & 0 & -T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$E[z_0] = [I_2 \quad O_2] \begin{bmatrix} 1 & 0 & -T & 0 \\ 0 & 1 & 0 & -T \end{bmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$= [I_2 \quad -T \cdot I_2] \begin{bmatrix} \rho_1 \\ u_1 \end{bmatrix}$$

$$= \rho_1 - T u_1$$

$$\rho_1 = K_{\rho 1} \rho_1 + K_{\rho 0} \cdot (\rho_1 - T u_1)$$

$$u_1 = K_{u1} \cdot \rho_1 + K_{u0} \cdot (\rho_1 - T u_1)$$

$$0 = \begin{bmatrix} (K_{\rho 1} + K_{\rho 0} - I_2) & T \cdot K_{\rho 0} \\ K_{u1} + K_{u0} & -T K_{u0} - I_2 \end{bmatrix}$$

$$\Rightarrow \cdot K_{u0} = I_2 \cdot \frac{1}{T}$$

$$\cdot K_{p0} = 0_2$$

$$K_{u1} + K_{u0} = 0$$

$$K_{u1} = -K_{u0}$$

$$\cdot K_{u1} = I_2 \cdot \frac{1}{T}$$

$$\cdot K_{p1} = I_2$$

c) $$Var\begin{bmatrix} \hat{\rho}_1 \\ \hat{u}_1 \end{bmatrix} = K_x^T \begin{bmatrix} Var[z_1] & \\ & Var[z_0] \end{bmatrix} K_x$$

$$= K_x^2 \begin{bmatrix} R & \\ & [I_2 - T \cdot I_2]^2 \, Q \end{bmatrix}$$

d) $x_k = [I_2 \quad 0_2]^{-1}(z_k - w_k)$

$x_k = [I_2 \quad 0_2]^{-1}(K_x \hat{x}_k - w_k)$

$=$

c) $Var[\hat{x}_1] = K_x \, cov(z) \, K_x^T$

$$cov\, z = \begin{bmatrix} cov(z_0, z_0) & cov(z_0, z_1) \\ cov(z_1, z_0) & cov(z_1, z_1) \end{bmatrix}$$

$$cov(z_0, z_0) = [I_2 \quad -T I_2] \, Q \, [I_2 - T I_2]^T + R$$

$$cov(z_0, z_1) = E\left[(z_1 - E[z_1]) \cdot (z_0 - E[z_0])^T\right]$$

$$= E\left[(w_1) \cdot (-[I_2 \quad -T I_2] \, v_0 + w_0)^T\right]$$

$$= 0$$

$$Var\left[\hat{x}_1\right] = K_x \begin{bmatrix} [I_2 \quad -T I_2] \, Q \, [I_2 \quad -T I_2]^T + R & 0 \\ 0 & R \end{bmatrix} K_x^T$$

d) only x_1

$$\hat{x}_1 = \begin{bmatrix} I_2 & 0 \\ \frac{1}{T} I_2 & -\frac{1}{T} I_2 \end{bmatrix} \begin{bmatrix} z_1 \\ z_0 \end{bmatrix}$$

$$= \begin{bmatrix} I_2 z_1 \\ \frac{1}{T} I_2 \cdot z_1 - \frac{1}{T} I_2 z_0 \end{bmatrix}$$

$$= \begin{bmatrix} I_2 \cdot (p_1 + w_1) \\ \frac{1}{T} I_2 (p_1 + w_1) - \frac{1}{T} I_2 (x_0 + w_0) \end{bmatrix}$$

$$= \begin{bmatrix} I_2 (p_1 + w_1) \\ \frac{1}{T} I_2 (p_1 + w_1) - \frac{1}{T} \left(F^{-1} \begin{bmatrix} p_1 - v_1 \\ u_1 - v_1 \end{bmatrix} + w_0 \right) \end{bmatrix}$$

$$= \begin{bmatrix} I_2 (p_1 + w_1) \\ \frac{1}{T} I_2 (p_1 + w_1) - \frac{1}{T} I_2 \left(p_1 - T u_1 - [I_2 - T I_2] v_0 + w_0 \right) \end{bmatrix}$$

$$\begin{cases} I_2(\rho_1 + w_1) \\ \frac{1}{T}(\rho_1) + \frac{w_1}{T} \end{cases} - \frac{1}{T}(\rho_1) I_2 - u_1 I_2 \quad \frac{I_2 w_0}{T} - [I_2 \; I_2 T] v_0$$

$$= \begin{bmatrix} \rho_1 \\ u_1 \end{bmatrix} + \begin{bmatrix} w_1 \\ \frac{w_1}{T} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{w_0}{T} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -I_2 & I_2 T \end{bmatrix} v_0$$

x_1 B C D

$$= \begin{bmatrix} \rho_1 \\ u_1 \end{bmatrix} + \begin{bmatrix} I_2 \\ I_2 T \end{bmatrix} w_1 + \begin{bmatrix} 0 \\ -\frac{I_2}{T} \end{bmatrix} w_0 + \begin{bmatrix} 0 & 0 \\ -I_2 & I_2 T \end{bmatrix} v_0$$

- Linear transformat° of gaussian variable $\Rightarrow$ Gaussian

$$\begin{cases} \hat{x}_1 = x_1 + B w_1 + C w_0 + D v_0 \\ x_1 = \hat{x}_1 + B w_1 - C w_0 - D v_0 \end{cases}$$

$$E[x_1] = E[\hat{x}_1] + 0 + 0 + 0$$

$$= \hat{x}_1$$

$$\text{Cov}[x_1] = 0 + B \text{Cov} w_1 B^T + (C \text{Cov}(w_0) C^T) + D \text{Cov} v_0 D^T$$

(Because not correlated!)

$$= 0 + BRB^T + CRC^T + DQD^T$$

x_1 is Gaussian

$$x_1 \sim N(\hat{x}_1, BRB^T + (PC^T + DQD^T))$$

e) Optimality:

Given the information we have, this method gives us the best estimate of x_0 and P_0, x_1 and P_1 however it is much worse having an optimistic (small) P than the opposite. Therefore, in practice one should increase P.

5a) Tuned! $\sigma_a = 5$

$\sigma_z = 5$

Obtained nice plot that followed the curve, minimal noise

b) NIS compares $\frac{1}{\epsilon_{k|k-1}}$ to measurement z_k, probably good as does not rely on model. (CT ≠ CV)

I obtain an interval for NIS for my values σ_a and σ_z

c) Not as good to use ANEES here as CT ≠ CV and ANEES puts.

Still, I want two conf. intervals to cross. $\sigma_a \approx 3,1$

$\sigma_z \approx 2,6$

d) When run on CT python-generated data, the estimates seem worse, but difficult to conclude

$$AX = B$$

$$XA = B$$

$$(XA)^T = B^T$$

$$A^T X^T = B^T$$

```
n: int = 4

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
    """
    F_km1 = np.eye(4, dtype = float)
    F_km1[0, 2] = Ts
    F_km1[1, 3] = Ts
    x_k1= self.F(x, Ts)@x
    return x_k1


def F(self,
        x: np.ndarray,
        Ts: float,
      ) -> np.ndarray:
    """ Calculate the transition function jacobian for Ts time units at x."""
    F_km1 = np.eye(4, dtype = float)
    F_km1[0, 2] = Ts
    F_km1[1, 3] = Ts
    return F_km1

def Q(self,
        x: np.ndarray,
        Ts: float,
      ) -> np.ndarray:
    """
    Calculate the Ts time units transition Covariance.
    """
    Q_mat = np.eye(4)
    Q_mat[0,0] = Ts**3/3
    Q_mat[0,2] = Ts**2/2
    Q_mat[1,1] = Ts**3/3
    Q_mat[1,3] = Ts**2/2

    Q_mat[2,0]= Ts**2/2
    Q_mat[2,2] = Ts
    Q_mat[3,1] = Ts**2/2
    Q_mat[3,3] = Ts
    # 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 Q_mat*self.sigma**2
```

```python
class MeasurementModel(Protocol):
    m: int

    def h(self, x: np.ndarray, *,
          sensor_state: Dict[str, Any] = None) -> np.ndarray: ...

    def H(self, x: np.ndarray, *,
          sensor_state: Dict[str, Any] = None) -> np.ndarray: ...

    def R(self, x: np.ndarray, *,
          sensor_state: Dict[str, Any] = None, z: np.ndarray = None) -> np.ndarray: ...
```

```python
# %% Models
```

```python
@dataclass
class CartesianPosition:
    sigma: float
    m: int = 2
    state_dim: int = 4

    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
        return x[0:2]


    def H(self,
            x: np.ndarray,
            *,
            sensor_state: Dict[str, Any] = None,
          ) -> np.ndarray:
        """Calculate the measurement Jacobian matrix at x in sensor_state."""
        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 potentially received measurement z."""
        return np.eye(self.m)*self.sigma**2
```

```python
class EKF:
    # A Protocol so duck typing can be used
    dynamic_model: dynmods.DynamicModel
    # A Protocol so duck typing can be used
    sensor_model: measmods.MeasurementModel

    #_MLOG2PIby2: float = field(init=False, repr=False)

    def __post_init__(self) -> None:
        self._MLOG2PIby2: Final[float] = self.sensor_model.m * \
            np.log(2 * np.pi) / 2

    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)
        P_pred = F@P@F.T+Q# TODO

        state_pred = GaussParams(x_pred, P_pred)

        return state_pred

    def innovation_mean(
```

```python
    return state_pred

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

    zbar = self.sensor_model.h(x)  # TODO predicted measurement

    v = z-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@H.T+R  # TODO the innovation covariance

    return S

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)
    S = self.innovation_cov(z, ekfstate)

    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 = np.linalg.solve(S.T, H@P).T # TODO:  kalman gain, Hint: la.solve, la.inv
```

```
        ) -> 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 = np.linalg.solve(S.T, H@P).T # TODO:  kalman gain, Hint: la.solve, la.inv

    x_upd = x + W@v  # TODO: the mean update
    P_upd = (np.eye(*P.shape)-W@H)@P

    ekfstate_upd = GaussParams(x_upd, P_upd)

    return ekfstate_upd

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 sensor_state."""

    # TODO: resue the above functions
    ekfstate_pred = self.predict(ekfstate, Ts)
    ekfstate_upd = self.update(z, ekfstate_pred)
    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_state"""

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

    NIS = v.T@np.linalg.solve(S, v)
    #S@x = V
    #x = Sinv@V
    #NIS =

    return NIS

@classmethod
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 = x_diff.T@np.linalg.solve(P, x_diff) # TODO
    return NEES
```

```
                ) -> 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.
        ll = None

        return ll

    @classmethod
    def estimate(cls, ekfstate: GaussParams):
        """Get the estimate from the state with its covariance. (Compatibility method)"""
        # dummy function for compatibility with IMM class
        return ekfstate

    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_pred
            init_ekfstate: GaussParams,
            # Time difference between Z's. If start_with_prediction: also diff before the firs
            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)

        # perform the actual predict and update cycle
        # 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)):
            ekfpred = self.predict(ekfupd, Tsk)
            ekfupd = self.update(zk, ekfpred)

            ekfpred_list[k] = ekfpred
            ekfupd_list[k] = ekfupd

        return ekfpred_list, ekfupd_list

    def performance_state(
```