python code for task 2

Bjørn Kåre Sæbø

Task 5

Code is included farther down in the file. See 1 for the ground truth trajectory

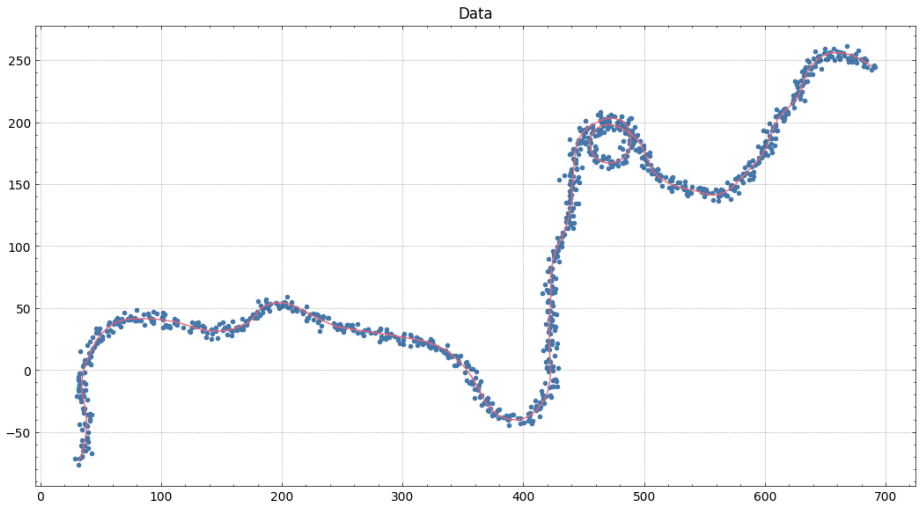

Figure 1: Measurements and GT trajectory which we try to estimate

5A

Went with $\sigma_a = 1$ and $\sigma_z = 3$. This seemed to work well enough, 2

5B

Plots included in 3. based on this it seems like increasing σ_a a bit could be a good idea, seems good somewhere around 3. Higher uncertanty in our model would probably be a good thing when using a CV model on a CT trajectory, as we can assume our model and the actual trajectory would differ quite a bit.

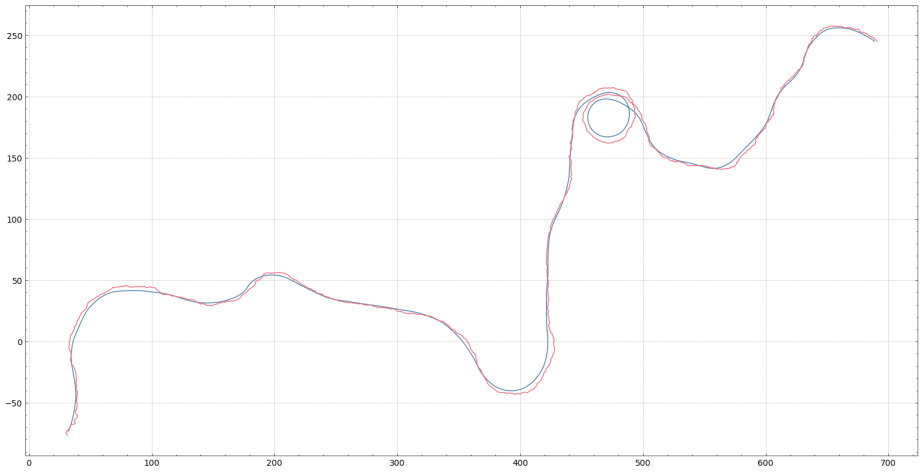

Figure 2: Estimated and GT values for tuned EKF

5C

Here in 4 we see much different result, with optimal values for σ_a somewhere around 1. Would probably not be great for tuning a CV model for CT trajectory, as we could end up trusting our model too much.

5D

Ran out of studasstime before getting to this part, happy enough that my code worked

Task 3 Code

dynamicmodels.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Dynamic models to be used with eg. EKF.

"""
# %%
from typing import Optional, Sequence
from typing_extensions import Final, Protocol
from dataclasses import dataclass, field

import numpy as np
```

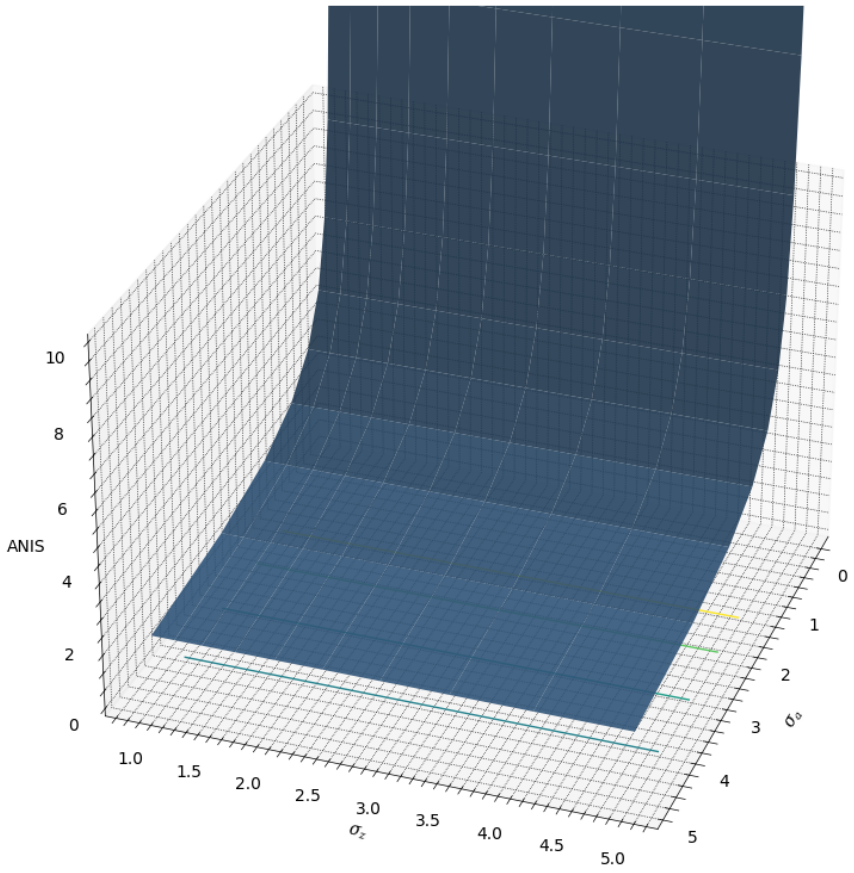

Figure 3: Contour plots of NIS

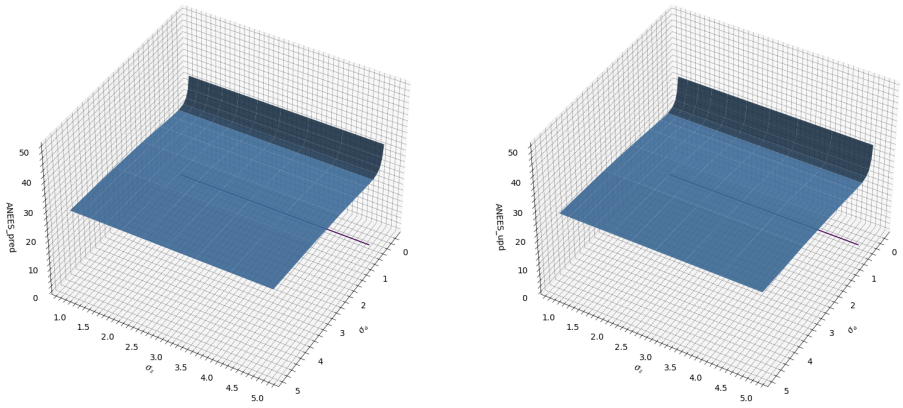

Figure 4: Contour plots of NEES

```python
# %% the dynamic models interface declaration


class DynamicModel(Protocol):
    n: int
    def f(self, x: np.ndarray, Ts: float) -> np.ndarray: ...
    def F(self, x: np.ndarray, Ts: float) -> np.ndarray: ...
    def Q(self, x: np.ndarray, Ts: float) -> np.ndarray: ...

# %%


@dataclass
class WhitenoiseAccelleration:
    """
    A white noise accelereation model also known as CV, states are position and speed.

    The model includes the discrete prediction equation f, its Jacobian F, and
    the process noise covariance Q as methods.
    """
    # noise standard deviation
    sigma: float
    # number of dimensions
    dim: int = 2
    # number of states
    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
        """
        x_p = np.array(x)
        x_p[:2] = x[:2] + Ts * x[2:4]
        x_p[2:4] = x[2:4]
        return x_p

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

    def Q(self,
            x: np.ndarray,
            Ts: float,
          ) -> np.ndarray:
        """
        Calculate the Ts time units transition Covariance.
        """
        Q_1_1 = np.eye(2)*1/3*Ts**3
        Q_1_2 = np.eye(2)*1/2*Ts**2
        Q_2_1 = np.eye(2)*1/2*Ts**2
        Q_2_2 = np.eye(2)*Ts

        Q = np.block([[Q_1_1, Q_1_2], [Q_2_1, Q_2_2]])

        # 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
```

measurementmodels.py

```
    # %% Imports
from typing import Any, Dict, Sequence, Optional
from dataclasses import dataclass, field
from typing_extensions import Protocol
```

```python
import numpy as np

# %% Measurement models interface declaration


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: ...

# %% Models


@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."""
        measurement = np.block([np.eye(2), np.zeros([2,2])]) @ x

        # x[0:2] is position
        # you do not need to care about sensor_state
        return measurement

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

```
        # 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 H

    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 potentia
        R = self.sigma**2*np.eye(2)
        # you do not need to care about sensor_state
        # sigma is available as self.sigma, and @dataclass makes it available in the init c
        return R
```

Task 4 code

EKF.py

```
"""
Notation:
----------
x is generally used for either the state or the mean of a gaussian. It should be clear from
P is used about the state covariance
z is a single measurement
Z (capital) are mulitple measurements so that z = Z[k] at a given time step
v is the innovation z - h(x)
S is the innovation covariance
"""
# %% Imports
# types
from typing import Union, Any, Dict, Optional, List, Sequence, Tuple, Iterable
from typing_extensions import Final

# packages
from dataclasses import dataclass, field
import numpy as np
import numpy.linalg as npla
import scipy.linalg as la
import scipy

# local
import dynamicmodels as dynmods
```

```python
import measurmentmodels as measmods
from gaussparams import GaussParams, GaussParamList

# %% The EKF


@dataclass
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

        state_pred = GaussParams(x_pred, P_pred)

        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)

        v = z - zbar

        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

        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."""

        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)

        # TODO: the kalman gain, Hint: la.solve, la.inv. FIkse mer her?
        W = P@H.T@la.inv(S)

        x_upd = x + W@v
        P_upd = (np.eye(self.dynamic_model.n) - 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_

        ekfstate_pred = self.predict(ekfstate, Ts)
        ekfstate_upd = self.update(z, ekfstate_pred, sensor_state)
        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 @ la.inv(S) @ v

        return NIS
```

```python
@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
    NEES = x_diff.T @ la.inv(P) @ x_diff
    return NEES

def gate(self,
         z: np.ndarray,
         ekfstate: GaussParams,
         *,
         sensor_state: Dict[str, Any],
         gate_size_square: float,
         ) -> bool:
    """ Check if z is inside sqrt(gate_sized_squared)-sigma ellipse of ekfstate in sens

    # a function to be used in PDA and IMM-PDA
    gated = None  # TODO in PDA exercise
    return gated

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)
    ll = -0.5 * (NIS + npla.slogdet(S))

    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_pr
        init_ekfstate: GaussParams,
        # Time difference between Z's. If start_with_prediction: also diff before the f
        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 list
    # 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)
```

```
            ekfpred_list[k] = ekfpred

            ekfupd = self.update(zk, ekfpred, ssk)
            ekfupd_list[k] = ekfupd

        return ekfpred_list, ekfupd_list

    def performance_stats(
            self,
            *,
            z: Optional[np.ndarray] = None,
            ekfstate_pred: Optional[GaussParams] = None,
            ekfstate_upd: Optional[GaussParams] = None,
            sensor_state: Optional[Dict[str, Any]] = None,
            x_true: Optional[np.ndarray] = None,
            # None: no norm, -1: all idx, seq: a single norm for given idxs, seqseq: a norms
            norm_idxs: Optional[Iterable[Sequence[int]]] = None,
            # The sequence of norms to calculate for idxs, see np.linalg.norm ord argument.
            norms: Union[Iterable[int], int] = 2,
    ) -> Dict[str, Union[float, List[float]]]:
        """Calculate performance statistics available from the given parameters."""
        stats: Dict[str, Union[float, List[float]]] = {}

        # NIS, needs measurements
        if z is not None and ekfstate_pred is not None:
            stats['NIS'] = self.NIS(
                z, ekfstate_pred, sensor_state=sensor_state)

        # NEES and RMSE, needs ground truth
        if x_true is not None:
            # prediction
            if ekfstate_pred is not None:
                stats['NEESpred'] = self.NEES(ekfstate_pred, x_true)

                # distances
                err_pred = ekfstate_pred.mean - x_true
                if norm_idxs is None:
                    stats['dist_pred'] = np.linalg.norm(err_pred, ord=norms)
                elif isinstance(norm_idxs, Iterable) and isinstance(norms, Iterable):
                    stats['dists_pred'] = [
                        np.linalg.norm(err_pred[idx], ord=ord)
                        for idx, ord in zip(norm_idxs, norms)]

            # update
            if ekfstate_upd is not None:
                stats['NEESupd'] = self.NEES(ekfstate_upd, x_true)
```

```
            # distances
            err_upd = ekfstate_upd.mean - x_true
            if norm_idxs is None:
                stats['dist_upd'] = np.linalg.norm(err_upd, ord=norms)
            elif isinstance(norm_idxs, Iterable) and isinstance(norms, Iterable):
                stats['dists_upd'] = [
                    np.linalg.norm(err_upd[idx], ord=ord)
                    for idx, ord in zip(norm_idxs, norms)]
    return stats

def performance_stats_sequence(
        self,
        # Sequence length
        K: int,
        *,
        # The measurements
        Z: Optional[Iterable[np.ndarray]] = None,
        ekfpred_list: Optional[Iterable[GaussParams]] = None,
        ekfupd_list: Optional[Iterable[GaussParams]] = None,
        # An optional sequence of all the sensor states when Z was recorded
        sensor_state: Optional[Iterable[Optional[Dict[str, Any]]]] = None,
        # Optional ground truth for error checking
        X_true: Optional[Iterable[Optional[np.ndarray]]] = None,
        # Indexes to be used to calculate errornorms, multiple to separate the state spa
        # None: all idx, Iterable (eg. list): each element is an index sequence into the
        norm_idxs: Optional[Iterable[Sequence[int]]] = None,
        # The sequence of norms to calculate for idxs (see numpy.linalg.norm ord argumen
        norms: Union[Iterable[int], int] = 2,
) -> np.ndarray:
    """Get performance metrics on a pre-estimated sequence"""

    None_list = [None] * K

    for_iter = []
    for_iter.append(Z if Z is not None else None_list)
    for_iter.append(ekfpred_list or None_list)
    for_iter.append(ekfupd_list or None_list)
    for_iter.append(sensor_state or None_list)
    for_iter.append(X_true if X_true is not None else None_list)

    stats = []
    for zk, ekfpredk, ekfupdk, ssk, xtk in zip(*for_iter):
        stats.append(
            self.performance_stats(
                z=zk, ekfstate_pred=ekfpredk, ekfstate_upd=ekfupdk, sensor_state=ssk, x_
```

```
                    norm_idxs=norm_idxs, norms=norms
                )
            )

        # make structured array
        dtype = [(key, *((type(val[0]), len(val)) if isinstance(val, Iterable)
                         else (type(val),))) for key, val in stats[0].items()]
        stats_arr = np.array([tuple(d.values()) for d in stats], dtype=dtype)

        return stats_arr


# %% End
```

runekf.py

```
# %% Imports
from gaussparams import GaussParams
import measurmentmodels
import dynamicmodels
import ekf
import scipy
import scipy.stats
import scipy.io
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# to see your plot config
print(f'matplotlib backend: {matplotlib.get_backend()}')
print(f'matplotlib config file: {matplotlib.matplotlib_fname()}')
print(f'matplotlib config dir: {matplotlib.get_configdir()}')
plt.close('all')

# set styles
try:
    # installed with "pip install SciencePLots" (https://github.com/garrettj403/SciencePlots
    # gives quite nice plots
    plt_styles = ['science', 'grid', 'bright', 'no-latex']
    plt.style.use(plt_styles)
    print(f'pyplot using style set {plt_styles}')
except Exception as e:
    print(e)
    print('setting grid and only grid and legend manually')
    plt.rcParams.update({
        # set grid
```

```
        'axes.grid': True,
        'grid.linestyle': ':',
        'grid.color': 'k',
        'grid.alpha': 0.5,
        'grid.linewidth': 0.5,
        # Legend
        'legend.frameon': True,
        'legend.framealpha': 1.0,
        'legend.fancybox': True,
        'legend.numpoints': 1,
    })


# %% get and plot the data
data_path = 'data_for_ekf.mat'

# TODO: choose this for the last task
usePregen = True  # choose between own generated data and pre generated

if usePregen:
    loadData: dict = scipy.io.loadmat(data_path)
    K: int = int(loadData['K'])  # The number of time steps
    Ts: float = float(loadData['Ts'])  # The sampling time
    Xgt: np.ndarray = loadData['Xgt'].T  # ground truth
    Z: np.ndarray = loadData['Z'].T  # the measurements
else:
    from sample_CT_trajectory import sample_CT_trajectory
    np.random.seed(10)  # random seed can be set for repeatability

    # initial state distribution
    x0 = np.array([0, 0, 1, 1, 0])
    P0 = np.diag([50, 50, 10, 10, np.pi/4]) ** 2

    # model parameters to sample from # TODO for toying around
    sigma_a_true = 0.25
    sigma_omega_true = np.pi/15
    sigma_z_true = 3

    # sampling interval a length
    K = 1000
    Ts = 0.1

    # get data
    Xgt, Z = sample_CT_trajectory(
        K, Ts, x0, P0, sigma_a_true, sigma_omega_true, sigma_z_true)
```

```python
# show ground truth and measurements
fig, ax = plt.subplots(num=1, clear=True)
ax.scatter(*Z.T, color='C0', marker='.')
ax.plot(*Xgt.T[:2], color='C1')
ax.set_title('Data')

# show turn rate
fig2, ax2 = plt.subplots(num=2, clear=True)
ax2.plot(Xgt.T[4])
ax2.set_xlabel('time step')
ax2.set_ylabel('turn rate')


# %% a: tune by hand and comment

# set parameters
sigma_a = 1
sigma_z = 3

# create the model and estimator object
dynmod = dynamicmodels.WhitenoiseAccelleration(sigma_a)
measmod = measurmentmodels.CartesianPosition(sigma_z)
ekf_filter = ekf.EKF(dynmod, measmod)
print(ekf_filter)  # make use of the @dataclass automatic repr

# initialize mean and covariance
x_bar_init = np.array([0, 0, 1, 1]).T # ???
P_bar_init = np.square(np.diag([75, 75, 10, 10]))
init_ekfstate = ekf.GaussParams(x_bar_init, P_bar_init)

# estimate
ekfpred_list, ekfupd_list = ekf_filter.estimate_sequence(Z, init_ekfstate, Ts)

# get statistics:
stats = ekf_filter.performance_stats_sequence(
    K, Z=Z, ekfpred_list=ekfpred_list, ekfupd_list=ekfupd_list, X_true=Xgt[:, :4],
    norm_idxs=[[0, 1], [2, 3]], norms=[2, 2]
)

print(f'keys in stats is {stats.dtype.names}')

# %% Calculate average performance metrics
# stats['dists_pred'] contains 2 norm of position and speed for each time index
# same for 'dists_upd'
RMSE_pred = np.sqrt(np.mean(np.square(stats['dists_pred']), axis=1))
RMSE_upd = np.sqrt(np.mean(np.square(stats['dists_upd']), axis=1))
```

```python
fig3, ax3 = plt.subplots(num=3, clear=True)

ax3.plot(*Xgt.T[:2])
ax3.plot(*ekfupd_list.mean.T[:2])
RMSEs_str = ", ".join(f"{v:.2f}" for v in (*RMSE_pred, *RMSE_upd))
ax3.set_title("")
    #rf'$\sigma_a = {sigma_a}$, $\sigma_z= {sigma_z}$,' + f'\nRMSE tunin(p_p, p_v, u_p, u_v

# %% Task 5 b and c

# % parameters for the parameter grid
# TODO: pick reasonable values for grid search
# n_vals = 20  # is Ok, try lower to begin with for more speed (20*20*1000 = 400 000 KF step
n_vals = 5 # 20
sigma_a_low = 0.1
sigma_a_high = 5
sigma_z_low = 1
sigma_z_high = 5

# % set the grid on logscale(not mandatory)
sigma_a_list = np.logspace(
    np.log10(sigma_a_low), np.log10(sigma_a_high), n_vals, base=10
)
sigma_z_list = np.logspace(
    np.log10(sigma_z_low), np.log10(sigma_z_high), n_vals, base=10
)

dtype = stats.dtype  # assumes the last cell has been run without faults
stats_array = np.empty((n_vals, n_vals, K), dtype=dtype)
# %% run through the grid and estimate
# ? Should be more or less a copy of the above
for i, sigma_a in enumerate(sigma_a_list):
    dynmod = dynamicmodels.WhitenoiseAccelleration(sigma_a)
    for j, sigma_z in enumerate(sigma_z_list):
        measmod = measurmentmodels.CartesianPosition(sigma_z)
        ekf_filter = ekf.EKF(dynmod, measmod)

        ekfpred_list, ekfupd_list = ekf_filter.estimate_sequence(Z, init_ekfstate, Ts)
        stats_array[i, j] = ekf_filter.performance_stats_sequence(
            K, Z=Z, ekfpred_list=ekfpred_list, ekfupd_list=ekfupd_list, X_true=Xgt[:, :4],
            norm_idxs=[[0, 1], [2, 3]], norms=[2, 2]
        )

# %% calculate averages
```

```
# TODO, remember to use axis argument, see eg. stats_array['dists_pred'].shape
RMSE_pred = np.sqrt(np.mean(np.square(stats_array['dists_pred']), axis=2))
RMSE_upd = np.sqrt(np.mean(np.square(stats_array['dists_upd']), axis=2))
ANEES_pred =  np.mean(stats_array['NEESpred'], axis=2) # mean of NEES over time
ANEES_upd = np.mean(stats_array['NEESupd'], axis=2)
ANIS = np.mean(stats_array['NIS'], axis=2) # mean of NIS over time


# %% find confidence regions for NIS and plot
confprob = 0.9  # ??? number to use for confidence interval
CINIS = np.array(scipy.stats.chi2.interval(confprob, 2*K)) / K  # ??? confidence intervall f
print(CINIS)

# plot
fig4 = plt.figure(4, clear=True)
ax4 = plt.gca(projection='3d')
ax4.plot_surface(*np.meshgrid(sigma_a_list, sigma_z_list),
                 ANIS, alpha=0.9)
ax4.contour(*np.meshgrid(sigma_a_list, sigma_z_list),
            ANIS, [1, 1.5, *CINIS, 2.5, 3], offset=0)  # , extend3d=True, colors='yellow')
ax4.set_xlabel(r'$\sigma_a$')
ax4.set_ylabel(r'$\sigma_z$')
ax4.set_zlabel('ANIS')
ax4.set_zlim(0, 10)
ax4.view_init(30, 20)

# %% find confidence regions for NEES and plot
confprob = 0.9
CINEES = np.array(scipy.stats.chi2.interval(confprob, 4*K)) / K  # TODO, not NIS now, but v
print(CINEES)

# plot
fig5 = plt.figure(5, clear=True)
ax5s = [fig5.add_subplot(1, 2, 1, projection='3d'),
        fig5.add_subplot(1, 2, 2, projection='3d')]
ax5s[0].plot_surface(*np.meshgrid(sigma_a_list, sigma_z_list),
                     ANEES_pred, alpha=0.9)
ax5s[0].contour(*np.meshgrid(sigma_a_list, sigma_z_list),
                ANEES_pred, [3, 3.5, *CINEES, 4.5, 5], offset=0)
ax5s[0].set_xlabel(r'$\sigma_a$')
ax5s[0].set_ylabel(r'$\sigma_z$')
ax5s[0].set_zlabel('ANEES_pred')
ax5s[0].set_zlim(0, 50)
ax5s[0].view_init(40, 30)

ax5s[1].plot_surface(*np.meshgrid(sigma_a_list, sigma_z_list),
```

```
                    ANEES_upd, alpha=0.9)
ax5s[1].contour(*np.meshgrid(sigma_a_list, sigma_z_list),
                ANEES_upd, [3, 3.5, *CINEES, 4.5, 5], offset=0)
ax5s[1].set_xlabel(r'$\sigma_a$')
ax5s[1].set_ylabel(r'$\sigma_z$')
ax5s[1].set_zlabel('ANEES_upd')
ax5s[1].set_zlim(0, 50)
ax5s[1].view_init(40, 30)

# %% see the intersection of NIS and NEESes
fig6, ax6 = plt.subplots(num=6, clear=True)
cont_upd = ax6.contour(*np.meshgrid(sigma_a_list, sigma_z_list),
                       ANEES_upd, CINEES, colors=['C0', 'C1'], labels='ANEESupd')
cont_pred = ax6.contour(*np.meshgrid(sigma_a_list, sigma_z_list),
                        ANEES_pred, CINEES, colors=['C2', 'C3'], labels='ANEESpred')
cont_nis = ax6.contour(*np.meshgrid(sigma_a_list, sigma_z_list),
                       ANIS, CINIS, colors=['C4', 'C5'], labels='NIS')

for cs, l in zip([cont_upd, cont_pred, cont_nis], ['NEESupd', 'NEESpred', 'NIS']):
    for c, hl in zip(cs.collections, ['low', 'high']):
        c.set_label(l + '_' + hl)
ax6.legend()
ax6.set_xlabel(r'$\sigma_a$')
ax6.set_ylabel(r'$\sigma_z$')

# %% show all the plots
plt.show()
```