complex_navigation_sensor_fusion2

December 21, 2025

[1]:
```
from pathlib import Path
from scipy.io import loadmat
import sys
import os


dataset_path = Path('data') / 'data.mat'
if not dataset_path.exists():
    alt = Path.cwd().parent / 'data' / 'data.mat'
    if alt.exists():
        dataset_path = alt
    else:
        raise FileNotFoundError(f"data.mat not found under {Path.cwd()} or its␣
 ↪parent")

notebook_path = os.getcwd()
print (f"Current notebook path: {notebook_path}")
project_root = os.path.dirname(notebook_path)
if project_root not in sys.path:
    sys.path.insert(0, project_root)
print (f"Added {project_root} to sys.path")

mat_data = loadmat(dataset_path)
print(mat_data.keys())
```

```
Current notebook path: /home/luky/skola/KalmanNet-for-state-
estimation/navigation NCLT dataset
Added /home/luky/skola/KalmanNet-for-state-estimation to sys.path
dict_keys(['__header__', '__version__', '__globals__', 'hB', 'souradniceGNSS',
'souradniceX', 'souradniceY', 'souradniceZ'])
```

[2]:
```
import torch
import matplotlib.pyplot as plt
from utils import trainer
from utils import utils
from Systems import DynamicSystem
import Filters
import torch.nn.functional as F
```

```
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
from scipy.io import loadmat
from scipy.interpolate import RegularGridInterpolator
import random

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
DEVICE = device  # For backward compatibility
print(f"device: {device}")
```

```
device: cuda
```

[3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
import matplotlib.pyplot as plt
import os
import Systems


# Parametry sekvencí
TRAIN_SEQ_LEN = 100       # Délka sekvence pro trénink (např. 100 kroků = 100␣
 ↪sekund při 1Hz)
VAL_SEQ_LEN = 400
TEST_SEQ_LEN = 1000      # Délka sekvence pro testování (delší sekvence pro␣
 ↪stabilnější vyhodnocení)
STRIDE = 20          # Posun okna (překryv) pro data augmentation
BATCH_SIZE = 256
DATA_PATH = 'data/processed'
print(f"Běží na zařízení: {device}")
```

```
Běží na zařízení: cuda
```

[4]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader
import os

def prepare_sequences(dataset_list, seq_len, stride, mode='train'):
    """
```

```
    Zpracuje list trajektorií na sekvence pro trénink dle článku.

    Nový formát dle [Song et al., 2024]:
    - Vstup u (4D): [v_left, v_right, theta_imu, omega_imu]
    - Cíl x (6D):   [px, py, vx, vy, theta, omega]
    """
    X_seq_list = [] # Ground Truth (Cíl)
    Y_seq_list = [] # GPS Měření (Vstup do korekce)
    U_seq_list = [] # Control Input (IMU/Odo)

    print(f"Zpracovávám {len(dataset_list)} trajektorií pro {mode}...")

    for traj in dataset_list:
        # 1. Extrahuje data
        # GT z preprocessingu je [px, py, theta]
        gt = traj['ground_truth'].float()

        # GPS: [x, y] (obsahuje NaN!)
        gps = traj['filtered_gps'].float()

        # IMU: [ax, ay, theta, omega]
        imu = traj['imu'].float()
        theta_imu = imu[:, 2] # Orientace z IMU
        omega_imu = imu[:, 3] # Úhlová rychlost z IMU

        # ODO: [v_left, v_right]
        odo = traj['filtered_wheel'].float()

        # Fix NaN v odometrii (nahradíme nulou)
        v_left = torch.nan_to_num(odo[:, 0], nan=0.0)
        v_right = torch.nan_to_num(odo[:, 1], nan=0.0)

        # 2. Sestavení vstupu u = [v_l, v_r, theta_imu, omega_imu] (4D)
        # Toto odpovídá "State Model" definovanému v článku (sekce II.C.2)
        u = torch.stack((v_left, v_right, theta_imu, omega_imu), dim=1)

        # 3. Sestavení cíle x (6D) pro state vector [px, py, vx, vy, theta,
↪omega]
        # Vyplníme to, co máme z Ground Truth (px, py, theta).
        # Rychlosti (vx, vy, omega) v GT implicitně nemáme (nebo je složité je
↪derivovat přesně),
        # ale pro trénink Loss funkce budeme stejně porovnávat primárně pozici.
        T = gt.shape[0]
        x_target = torch.zeros(T, 6)
        x_target[:, 0] = gt[:, 0] # px
        x_target[:, 1] = gt[:, 1] # py
        x_target[:, 4] = gt[:, 2] # theta
```

```
        # Ostatní (vx, vy, omega) zůstávají 0, protože v Loss funkci budeme␣
 ↪maskovat nebo brát jen pozici.

        # 4. Sliding Window (Rozsekání na sekvence)
        num_samples = gt.shape[0]
        current_stride = stride if mode == 'train' else seq_len # U testu bez␣
 ↪překryvu

        for i in range(0, num_samples - seq_len + 1, current_stride):
            # Cíl: 6D stav
            x_seq = x_target[i : i+seq_len, :]

            # Měření: GPS [px, py]
            y_seq = gps[i : i+seq_len, :]

            # Vstup: 4D control input
            u_seq = u[i : i+seq_len, :]

            X_seq_list.append(x_seq)
            Y_seq_list.append(y_seq)
            U_seq_list.append(u_seq)

    # Stack do tenzorů
    X_out = torch.stack(X_seq_list)
    Y_out = torch.stack(Y_seq_list)
    U_out = torch.stack(U_seq_list)

    return X_out, Y_out, U_out

# === NAČTENÍ DAT ===
# Ujistíme se, že cesty a konstanty jsou definované (pokud nejsou, doplňte je␣
 ↪nahoře)
# if 'DATA_PATH' not in locals(): DATA_PATH = 'data/processed'
# if 'TRAIN_SEQ_LEN' not in locals(): TRAIN_SEQ_LEN = 100
# if 'VAL_SEQ_LEN' not in locals(): VAL_SEQ_LEN = 200
# if 'TEST_SEQ_LEN' not in locals(): TEST_SEQ_LEN = 500
# if 'STRIDE' not in locals(): STRIDE = 20
# if 'BATCH_SIZE' not in locals(): BATCH_SIZE = 256

train_data_raw = torch.load(os.path.join(DATA_PATH, 'train.pt'))
val_data_raw = torch.load(os.path.join(DATA_PATH, 'val.pt'))
test_data_raw = torch.load(os.path.join(DATA_PATH, 'test.pt'))

# === PŘÍPRAVA SEKVENCÍ ===
print("--- Generuji trénovací data (Paper compatible) ---")
train_X, train_Y, train_U = prepare_sequences(train_data_raw, TRAIN_SEQ_LEN,␣
 ↪STRIDE, 'train')
```

```
print("\n--- Generuji validační data ---")
val_X, val_Y, val_U = prepare_sequences(val_data_raw, VAL_SEQ_LEN, VAL_SEQ_LEN,␣
 ↪'val')

print("\n--- Generuji testovací data ---")
test_X, test_Y, test_U = prepare_sequences(test_data_raw, TEST_SEQ_LEN,␣
 ↪TEST_SEQ_LEN, 'test')

# Vytvoření DataLoaderů
train_loader = DataLoader(TensorDataset(train_X, train_Y, train_U),␣
 ↪batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(TensorDataset(val_X, val_Y, val_U),␣
 ↪batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(TensorDataset(test_X, test_Y, test_U),␣
 ↪batch_size=BATCH_SIZE, shuffle=False)

print(f"\n Data připravena.")
print(f"Train batches: {len(train_loader)}")
print(f"Shapes -> X: {train_X.shape} (6D State), U: {train_U.shape} (4D Input),␣
 ↪Y: {train_Y.shape} (2D Meas)")
```

```
--- Generuji trénovací data (Paper compatible) ---
Zpracovávám 22 trajektorií pro train…

--- Generuji validační data ---
Zpracovávám 2 trajektorií pro val…

--- Generuji testovací data ---
Zpracovávám 3 trajektorií pro test…

 Data připravena.
Train batches: 22
Shapes -> X: torch.Size([5446, 100, 6]) (6D State), U: torch.Size([5446, 100,
4]) (4D Input), Y: torch.Size([5446, 100, 2]) (2D Meas)
```

[5]:
```
# === INICIALIZACE DYNAMICKÉHO MODELU (System Instance - Paper Version) ===

# 1. Parametry systému podle článku [Song et al., 2024]
# State (6D): [px, py, vx, vy, theta, omega]
# Referenční rovnice (5) v článku.
state_dim = 6
# Meas (2D):  [gps_x, gps_y]
# Referenční rovnice (6) v článku.
obs_dim = 2
# Časový krok (z preprocessingu)
dt = 1.0
```

```
# 2. Definice Matice Q (Procesní šum / Model Uncertainty)
# Nyní máme 6 stavů. Musíme definovat nejistotu pro každý z nich.
# Hodnoty jsou nastaveny heuristicky (lze ladit):
# - Pozice (idx 0,1): 0.1
# - Rychlost (idx 2,3): 0.1
# - Úhel/Omega (idx 4,5): 0.01 (IMU je v NCLT docela přesné, ale driftuje)
q_diag = torch.tensor([0.1, 0.1, 0.1, 0.1, 0.01, 0.01])
Q = torch.diag(q_diag)

# 3. Definice Matice R (Šum měření / Sensor Noise)
# GPS měří jen pozici (px, py).
# Nastavujeme 1.0 m^2. To odpovídá standardní odchylce 1m.
# Pokud je GPS v datasetu horší, KalmanNet se naučí "nedůvěřovat" vstupu y
# a spoléhat více na predikci z u (odometrie).
r_diag = torch.tensor([1.0, 1.0])
R = torch.diag(r_diag)

# 4. Počáteční podmínky (Prior)
# Ex0: Nulový vektor 6x1
Ex0 = torch.zeros(state_dim, 1)

# P0: Počáteční kovariance
# Autoři používají P k inicializaci EKF[cite: 700].
# Nastavíme rozumnou počáteční nejistotu.
P0 = torch.eye(state_dim) * 0.5

# 5. Vytvoření instance DynamicSystemNCLT
# Důležité: f=None zajistí, že se použije interní `_f_paper_dynamics` (rovnice␣
 ↪5),
# která očekává 4D vstup (v_l, v_r, theta, omega).
sys_model = Systems.DynamicSystemNCLT(
    state_dim=state_dim,
    obs_dim=obs_dim,
    Q=Q,
    R=R,
    Ex0=Ex0,
    P0=P0,
    dt=dt,
    f=None,  # None -> Použije se model z článku: px += vc*cos(theta_imu)...
    h=None,  # None -> Použije se GPS model: y = [px, py]
    device=DEVICE
)

print(f" System Model NCLT inicializován (Paper Version).")
print(f" - State Dim: {sys_model.state_dim} [px, py, vx, vy, theta, omega]")
print(f" - Meas Dim:  {sys_model.obs_dim} [gps_x, gps_y]")
```

```
print(f" - Input Dim: 4 [v_l, v_r, theta_imu, omega_imu]") # Implicitně v modelu
print(f" - Q Diag: {q_diag.tolist()}")
```

```
 System Model NCLT inicializován (Paper Version).
 - State Dim: 6 [px, py, vx, vy, theta, omega]
 - Meas Dim:  2 [gps_x, gps_y]
 - Input Dim: 4 [v_l, v_r, theta_imu, omega_imu]
 - Q Diag: [0.10000000149011612, 0.10000000149011612, 0.10000000149011612,
0.10000000149011612, 0.009999999776482582, 0.009999999776482582]
```

[7]:
```
import torch
import torch.optim as optim
import os
from state_NN_models import StateKalmanNet
from utils import trainer

# === 1. KONFIGURACE A INICIALIZACE MODELU ===

# Hyperparametry sítě
# State dim je 3. Multiplier 40 znamená hidden state velikosti 120.
# To je pro navigaci s nelinearitami (sin/cos) rozumná kapacita.
print("Inicializuji StateKalmanNet...")
state_knet = StateKalmanNet(
    system_model=sys_model,
    device=DEVICE,
    hidden_size_multiplier=6,      # Větší kapacita pro složitější dynamiku
    output_layer_multiplier=4,
    num_gru_layers=1,               # 1 vrstva GRU obvykle stačí a je␣
 ↪stabilnější
    gru_hidden_dim_multiplier=6
).to(DEVICE)
print(state_knet)

# Počet trénovatelných parametrů
params_count = sum(p.numel() for p in state_knet.parameters() if p.
 ↪requires_grad)
print(f"Model má {params_count} trénovatelných parametrů.")

# === 2. NASTAVENÍ TRÉNINKU (TBPTT) ===

# Parametry pro Sliding Window trénink (TBPTT)
# NCLT sekvence jsou dlouhé (100 kroků), gradienty by mohly explodovat.
# Dělíme je na okna délky 20 a gradienty ořezáváme.
TBPTT_WINDOW = 8  # Délka okna (w)
TBPTT_STEP = 2    # Krok pro detach (k) - obvykle polovina w

EPOCHS = 100
```

```python
LEARNING_RATE = 1e-3
WEIGHT_DECAY = 1e-4 # Jemná regularizace
CLIP_GRAD = 1.0    # Důležité: Ořezání gradientů pro stabilitu RNN

# === 3. SPUŠTĚNÍ TRÉNINKU ===
print("\n Spouštím tréninkovou smyčku...")

trained_knet = trainer.train_state_KalmanNet_sliding_windowNCLT(
    model=state_knet,
    train_loader=train_loader,
    val_loader=val_loader,
    device=DEVICE,
    epochs=EPOCHS,
    lr=LEARNING_RATE,
    weight_decay_=WEIGHT_DECAY,
    clip_grad=CLIP_GRAD,
    early_stopping_patience=20, # Zastaví, pokud se 20 epoch nezlepší loss
    tbptt_k=TBPTT_STEP,
    tbptt_w=TBPTT_WINDOW
)

# === 4. ULOŽENÍ MODELU ===
save_path = 'best_kalmannet_nclt_sensor_fusion.pth'
torch.save(trained_knet.state_dict(), save_path)
print(f"\n Trénink dokončen. Nejlepší model uložen do: {save_path}")
```

```
Inicializuji StateKalmanNet…
DEBUG: Layer 'output_final_linear.0' initialized near zero (Start K=0).
StateKalmanNet(
  (dnn): DNN_KalmanNet(
    (input_layer): Sequential(
      (0): Linear(in_features=16, out_features=384, bias=True)
      (1): ReLU()
    )
    (gru): GRU(384, 240)
    (output_hidden_layer): Sequential(
      (0): Linear(in_features=240, out_features=48, bias=True)
      (1): ReLU()
    )
    (output_final_linear): Sequential(
      (0): Linear(in_features=48, out_features=12, bias=True)
    )
  )
)
Model má 469404 trénovatelných parametrů.

 Spouštím tréninkovou smyčku…
```

```
INFO: Starting training with TBPTT(k=2, w=8)
INFO: Returns covariance: False
Epoch [1/100] | Train Loss: 9.5125 | Val Loss: 22.5723
 -> New best model saved! (Val Loss: 22.5723)
Epoch [2/100] | Train Loss: 6.0817 | Val Loss: 15.2679
 -> New best model saved! (Val Loss: 15.2679)
Epoch [3/100] | Train Loss: 5.9775 | Val Loss: 18.2815
Epoch [4/100] | Train Loss: 5.3788 | Val Loss: 13.6975
 -> New best model saved! (Val Loss: 13.6975)
Epoch [5/100] | Train Loss: 5.3908 | Val Loss: 17.4238
Epoch [6/100] | Train Loss: 5.3412 | Val Loss: 13.0771
 -> New best model saved! (Val Loss: 13.0771)
Epoch [7/100] | Train Loss: 4.8077 | Val Loss: 10.7931
 -> New best model saved! (Val Loss: 10.7931)
Epoch [8/100] | Train Loss: 4.8601 | Val Loss: 13.5151
Epoch [9/100] | Train Loss: 4.7902 | Val Loss: 19.6605
Epoch [10/100] | Train Loss: 4.9070 | Val Loss: 12.2328
Epoch [11/100] | Train Loss: 4.8015 | Val Loss: 15.2048
Epoch [12/100] | Train Loss: 4.5990 | Val Loss: 14.5768
Epoch [13/100] | Train Loss: 4.7410 | Val Loss: 19.0976
Epoch [14/100] | Train Loss: 4.7292 | Val Loss: 17.6687
Epoch [15/100] | Train Loss: 4.6063 | Val Loss: 16.5987
Epoch [16/100] | Train Loss: 4.5577 | Val Loss: 15.1895
Epoch [17/100] | Train Loss: 4.4941 | Val Loss: 10.1394
 -> New best model saved! (Val Loss: 10.1394)
Epoch [18/100] | Train Loss: 4.4889 | Val Loss: 65.2614
Epoch [19/100] | Train Loss: 4.5291 | Val Loss: 314.7125
Epoch [20/100] | Train Loss: 4.5236 | Val Loss: 14.6616
Epoch [21/100] | Train Loss: 4.4670 | Val Loss: 17.1359
Epoch [22/100] | Train Loss: 4.4224 | Val Loss: 16.6346
Epoch [23/100] | Train Loss: 4.4790 | Val Loss: 27.4227
Epoch [24/100] | Train Loss: 4.3904 | Val Loss: 18.8214
Epoch [25/100] | Train Loss: 4.3553 | Val Loss: 20.6535
Epoch [26/100] | Train Loss: 4.4038 | Val Loss: 18.6869
Epoch [27/100] | Train Loss: 4.3146 | Val Loss: 49.1382
Epoch [28/100] | Train Loss: 4.2550 | Val Loss: 20.0896
Epoch [29/100] | Train Loss: 4.4484 | Val Loss: 11.4378
Epoch [30/100] | Train Loss: 4.3568 | Val Loss: 264.3260
Epoch [31/100] | Train Loss: 4.1888 | Val Loss: 14.3947
Epoch [32/100] | Train Loss: 4.1927 | Val Loss: 12.6092
Epoch [33/100] | Train Loss: 4.2740 | Val Loss: 33.7934
Epoch [34/100] | Train Loss: 4.2219 | Val Loss: 13.2243
Epoch [35/100] | Train Loss: 4.3017 | Val Loss: 27.9949
Epoch [36/100] | Train Loss: 4.1944 | Val Loss: 12.1453
Epoch [37/100] | Train Loss: 4.2959 | Val Loss: 20.7706

Early stopping triggered after 37 epochs.
Training completed.
```

```
Loading best model with validation loss: 10.139379

 Trénink dokončen. Nejlepší model uložen do:
best_kalmannet_nclt_sensor_fusion.pth
```

[8]:
```python
import torch
import torch.nn.functional as F
import numpy as np
import Filters  # Tvůj modul pro klasické filtry
from utils import utils

# ==============================================================================
# 0. KONFIGURACE A PŘÍPRAVA MODELŮ
# ==============================================================================
# 1. KalmanNet (tvůj natrénovaný model)
try:
    trained_model_classic = state_knet
    trained_model_classic.eval()
    print("INFO: KalmanNet (state_knet) připraven k testování.")
except NameError:
    raise NameError("Chyba: Proměnná 'state_knet' neexistuje. Spusťte nejprve␣
 ↪trénink.")

# 2. Klasické filtry (EKF, UKF)
# Poznámka: Aby EKF/UKF na NCLT fungovaly dobře, musí jejich implementace␣
 ↪podporovat
# vstup 'u' (rychlost/omega) v predikčním kroku f(x, u).
# Pokud tvá třída Filters.py nepodporuje 'u', budou tyto filtry fungovat jen␣
 ↪jako 'GPS smoother'.
print("Inicializuji EKF a UKF...")
ekf_filter = Filters.ExtendedKalmanFilter(sys_model)
ukf_filter = Filters.UnscentedKalmanFilter(sys_model)

# ==============================================================================
# 1. VYHODNOCOVACÍ SMYČKA
# ==============================================================================
mse_knet = []
mse_ekf, anees_ekf = [], []
mse_ukf, anees_ukf = [], []

traj_idx = 0
total_trajectories = len(test_loader.dataset)

print(f"\nVyhodnocuji {total_trajectories} sekvencí z testovací sady...")
print("POZNÁMKA: Startujeme z Ground Truth pozice.")

with torch.no_grad():
```

```
    # ZMĚNA: Unpacking 3 hodnot (x, y, u)
    for x_true_batch, y_meas_batch, u_input_batch in test_loader:

        batch_size = x_true_batch.shape[0]

        for i in range(batch_size):
            traj_idx += 1

            # Příprava dat pro jednu trajektorii
            y_seq = y_meas_batch[i].to(DEVICE)    # [Seq_Len, 2]
            u_seq = u_input_batch[i].to(DEVICE)   # [Seq_Len, 2]
            x_true = x_true_batch[i].to(DEVICE)   # [Seq_Len, 3]

            seq_len = y_seq.shape[0]

            # --- PŘÍPRAVA POČÁTEČNÍHO STAVU (GROUND TRUTH) ---
            # KalmanNet: [1, State_Dim]
            knet_init_state = x_true[0, :].unsqueeze(0)

            # Filtry: [State_Dim, 1]
            filter_init_state = x_true[0, :].unsqueeze(1)

            # --- A. KalmanNet (Neural Network) ---
            trained_model_classic.reset(batch_size=1,␣
↪initial_state=knet_init_state)

            knet_preds = [knet_init_state] # Startujeme z GT

            for t in range(1, seq_len):
                # ZMĚNA: Posíláme y_t i u_t (řízení)
                y_t = y_seq[t, :].unsqueeze(0)
                u_t = u_seq[t, :].unsqueeze(0)

                # KNet Step
                x_hat_t = trained_model_classic.step(y_t, u_t)
                knet_preds.append(x_hat_t)

            full_x_hat_knet = torch.cat(knet_preds, dim=0)

            # --- B. Extended Kalman Filter (EKF) ---
            # Pokusíme se použít standardní process_sequence.
            # Pokud Filters.py neumí 'u', tento odhad bude horší než KNet.
            # try:
            #     # Předpokládáme, že process_sequence umí buď 'u' jako␣
↪argument, nebo ho ignoruje.
            #     # Pokud ho ignoruje, EKF pojede jen na GPS modelu (Constant␣
↪Position).
```

```
            #     # ekf_res = ekf_filter.process_sequence(y_seq,␣
↪Ex0=filter_init_state, P0=sys_model.P0)
            #     # full_x_hat_ekf = ekf_res['x_filtered']
            #     # full_P_hat_ekf = ekf_res['P_filtered']
            # except Exception as e:
            #     # Fallback, pokud EKF selže (např. kvůli dimenzím)
            #     full_x_hat_ekf = torch.zeros_like(x_true)
            #     full_P_hat_ekf = torch.eye(3).unsqueeze(0).repeat(seq_len, 1,␣
↪1).to(DEVICE)

            # --- C. Unscented Kalman Filter (UKF) ---
            try:
                ukf_res = ukf_filter.process_sequence(y_seq,u_seq,␣
↪Ex0=filter_init_state, P0=sys_model.P0)
                full_x_hat_ukf = ukf_res['x_filtered']
                full_P_hat_ukf = ukf_res['P_filtered']
            except Exception as e:
                full_x_hat_ukf = torch.zeros_like(x_true)
                full_P_hat_ukf = torch.eye(3).unsqueeze(0).repeat(seq_len, 1,␣
↪1).to(DEVICE)

            # --- VÝPOČET METRIK ---
            # MSE pro pozici (první 2 stavy: x, y)
            # Ignorujeme theta, protože MSE na úhlech je ošemetné (periodicta)
            mse_val_knet = F.mse_loss(full_x_hat_knet[1:, :2], x_true[1:, :2]).
↪item()
            # mse_val_ekf = F.mse_loss(full_x_hat_ekf[1:, :2], x_true[1:, :2]).
↪item()
            mse_val_ukf = F.mse_loss(full_x_hat_ukf[1:, :2], x_true[1:, :2]).
↪item()

            mse_knet.append(mse_val_knet)
            # mse_ekf.append(mse_val_ekf)
            mse_ukf.append(mse_val_ukf)

            # ANEES (Pokud filtry vrátily P)
            # Funkce ANEES by měla zvládnout celou trajektorii najednou
            # (Pozor na dimenze: [1, T, Dim])
            def safe_anees(x_t, x_h, P_h):
                if torch.all(x_h == 0): return 0.0 # Skip failed filters
                return utils.calculate_anees_vectorized(
                    x_t.unsqueeze(0).cpu(),
                    x_h.unsqueeze(0).cpu(),
                    P_h.unsqueeze(0).cpu()
                )
```

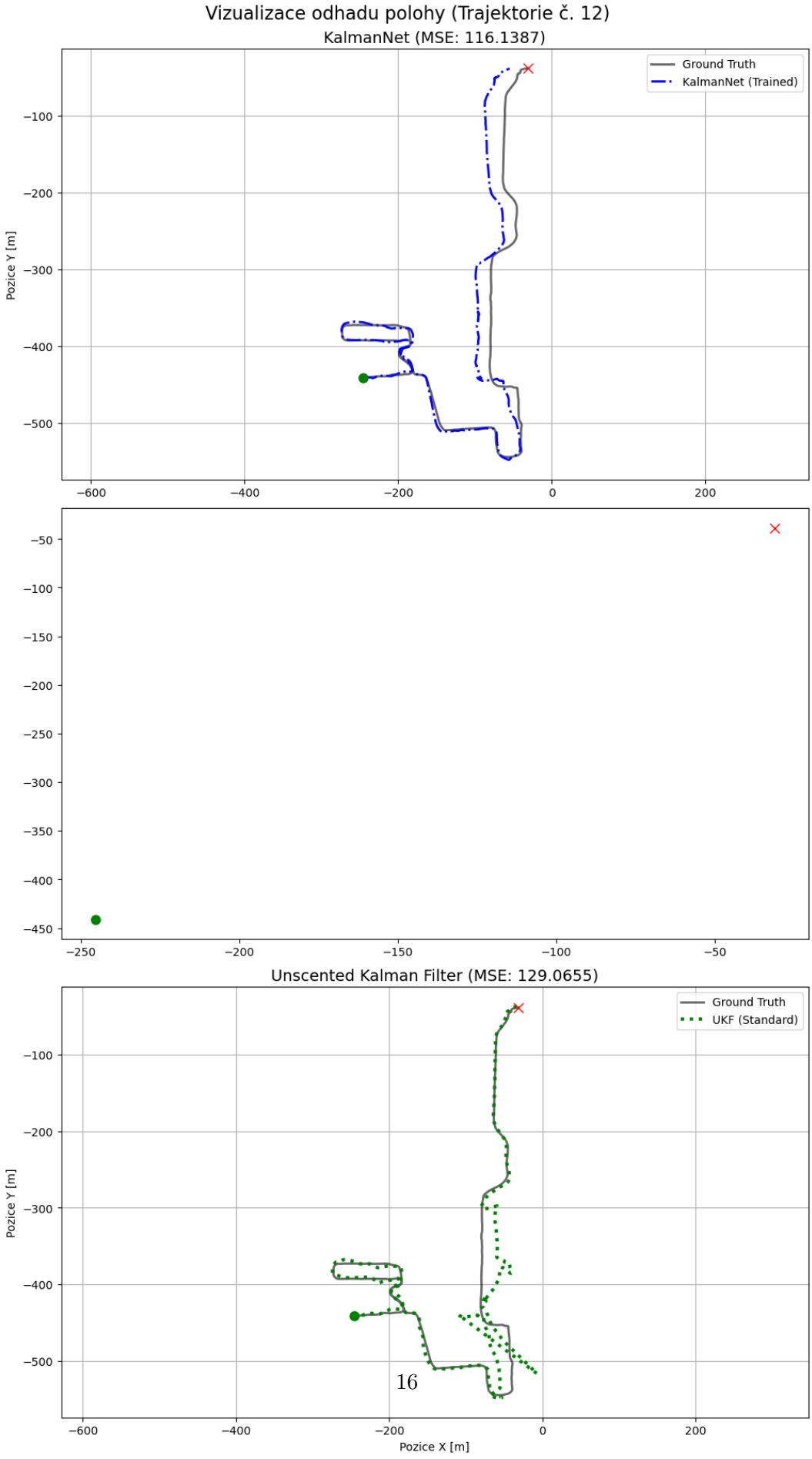
```
            # anees_ekf.append(safe_anees(x_true, full_x_hat_ekf,␣
 ↪full_P_hat_ekf))
            anees_ukf.append(safe_anees(x_true, full_x_hat_ukf, full_P_hat_ukf))

            if traj_idx % 50 == 0:
                print(f"Zpracováno {traj_idx}/{total_trajectories} trajektorií..
 ↪.")

# ================================================================================
# 2. VÝPIS VÝSLEDKŮ
# ================================================================================
def avg(lst): return np.mean([l for l in lst if l is not None])

print("\n" + "="*80)
print(f"FINÁLNÍ VÝSLEDKY NA NCLT DATASETU (Test Set)")
print(f"Metrika: MSE Pozice [m^2] (Nižší je lepší)")
print("="*80)

# KalmanNet
print(f"{'KalmanNet (Trained)':<30} | MSE: {avg(mse_knet):.4f} | ANEES: N/A")

# EKF
# print(f"{'EKF (Standard)':<30} | MSE: {avg(mse_ekf):.4f} | ANEES:␣
 ↪{avg(anees_ekf):.4f}")

# UKF
print(f"{'UKF (Standard)':<30} | MSE: {avg(mse_ukf):.4f} | ANEES:␣
 ↪{avg(anees_ukf):.4f}")

# print("="*80)
# if avg(mse_knet) < avg(mse_ekf):
#     print(" KalmanNet překonal EKF!")
# else:
#     print(" KalmanNet zatím nepřekonal EKF. Zkuste více epoch nebo ladit Q/R.
 ↪")
```

```
INFO: KalmanNet (state_knet) připraven k testování.
Inicializuji EKF a UKF…

Vyhodnocuji 12 sekvencí z testovací sady…
POZNÁMKA: Startujeme z Ground Truth pozice.

================================================================================
FINÁLNÍ VÝSLEDKY NA NCLT DATASETU (Test Set)
Metrika: MSE Pozice [m^2] (Nižší je lepší)
================================================================================
KalmanNet (Trained)            | MSE: 91.9216 | ANEES: N/A
```

```
UKF (Standard)                | MSE: 294.2805 | ANEES: 802.4825
```

[9]:
```python
import matplotlib.pyplot as plt

# === VIZUALIZACE POSLEDNÍ TESTOVACÍ TRAJEKTORIE ===

# 1. Převod na NumPy (pro matplotlib)
# Používáme data z poslední iterace smyčky (poslední trajektorie v test setu)
gt_np = x_true.cpu().numpy()
knet_np = full_x_hat_knet.cpu().numpy()
# ekf_np = full_x_hat_ekf.cpu().numpy()
ukf_np = full_x_hat_ukf.cpu().numpy()

# Názvy indexů pro lepší čitelnost
PX, PY = 0, 1

# Vytvoření subplotů (3 řádky, 1 sloupec)
fig, axs = plt.subplots(3, 1, figsize=(10, 18), constrained_layout=True)
fig.suptitle(f"Vizualizace odhadu polohy (Trajektorie č. {traj_idx})",␣
 ↪fontsize=16)

# --- 1. Graf: KalmanNet ---
axs[0].plot(gt_np[:, PX], gt_np[:, PY], 'k-', linewidth=2, label='Ground␣
 ↪Truth', alpha=0.6)
axs[0].plot(knet_np[:, PX], knet_np[:, PY], 'b-.', linewidth=2,␣
 ↪label='KalmanNet (Trained)')
axs[0].set_title(f"KalmanNet (MSE: {F.mse_loss(full_x_hat_knet[:, :2], x_true[:
 ↪, :2]):.4f})", fontsize=14)
axs[0].set_ylabel("Pozice Y [m]")
axs[0].legend()
axs[0].grid(True)
axs[0].axis('equal') # Aby mapa nebyla deformovaná

# --- 2. Graf: EKF ---
# axs[1].plot(gt_np[:, PX], gt_np[:, PY], 'k-', linewidth=2, label='Ground␣
 ↪Truth', alpha=0.6)
# axs[1].plot(ekf_np[:, PX], ekf_np[:, PY], 'r--', linewidth=2, label='EKF␣
 ↪(Standard)')
# axs[1].set_title(f"Extended Kalman Filter (MSE: {F.mse_loss(full_x_hat_ekf[:,␣
 ↪:2], x_true[:, :2]):.4f})", fontsize=14)
# axs[1].set_ylabel("Pozice Y [m]")
# axs[1].legend()
# axs[1].grid(True)
# axs[1].axis('equal')

# --- 3. Graf: UKF ---
```

```
axs[2].plot(gt_np[:, PX], gt_np[:, PY], 'k-', linewidth=2, label='Ground␣
 ↪Truth', alpha=0.6)
axs[2].plot(ukf_np[:, PX], ukf_np[:, PY], 'g:', linewidth=3, label='UKF␣
 ↪(Standard)')
axs[2].set_title(f"Unscented Kalman Filter (MSE: {F.mse_loss(full_x_hat_ukf[:, :
 ↪2], x_true[:, :2]):.4f})", fontsize=14)
axs[2].set_xlabel("Pozice X [m]")
axs[2].set_ylabel("Pozice Y [m]")
axs[2].legend()
axs[2].grid(True)
axs[2].axis('equal')

# Přidání start/cíl markerů do všech grafů
for ax in axs:
    ax.plot(gt_np[0, PX], gt_np[0, PY], 'go', markersize=8, label='Start')
    ax.plot(gt_np[-1, PX], gt_np[-1, PY], 'rx', markersize=8, label='Cíl')

plt.show()
```


Vizualizace odhadu polohy (Trajektorie č. 12)
KalmanNet (MSE: 116.1387)
Ground Truth
KalmanNet (Trained)
Pozice Y [m]
−100
−200
−300
−400
−500
−600
−400
−200
0
200
−50
−100
−150
−200
−250
−300
−350
−400
−450
−250
−200
−150
−100
−50
Unscented Kalman Filter (MSE: 129.0655)
Ground Truth
UKF (Standard)
Pozice Y [m]
Pozice X [m]

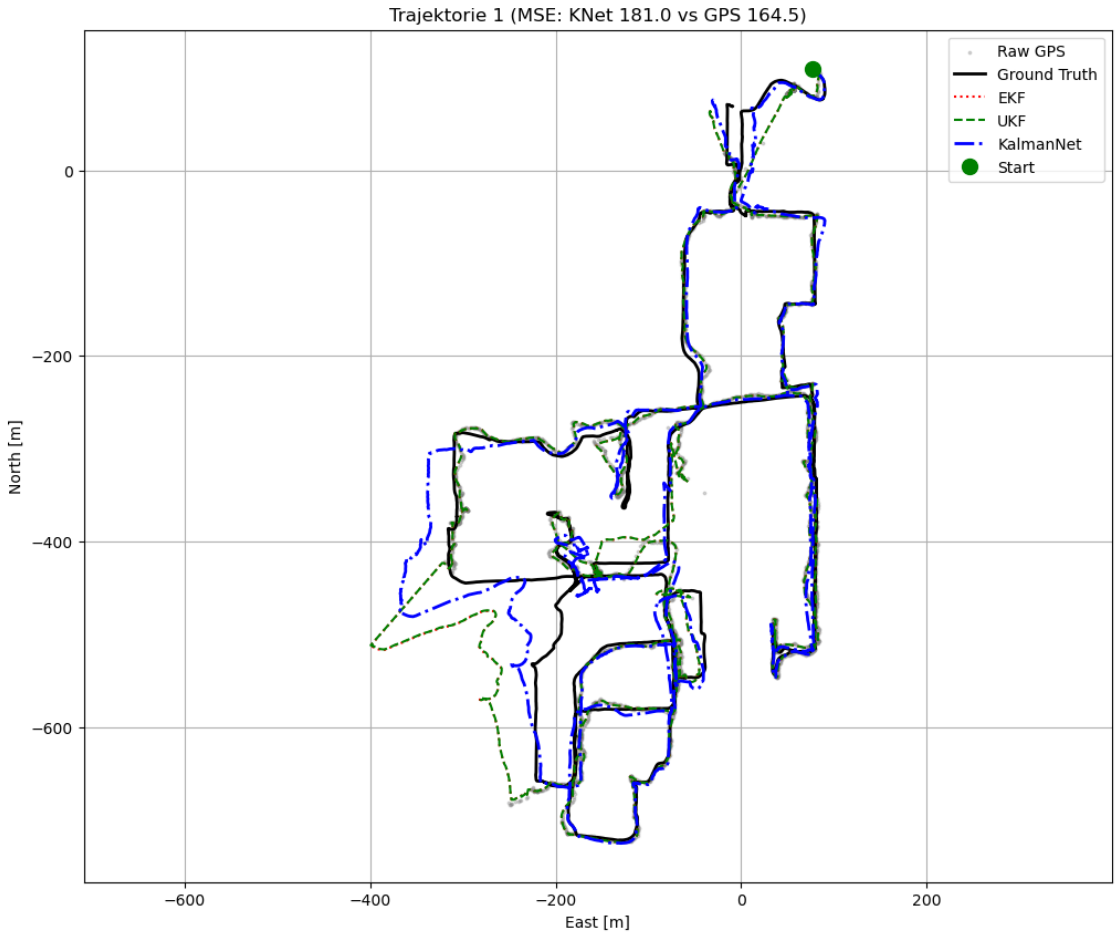
[10]:
```
import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
import Filters
from utils import utils

# ================================================================================
# 0. KONFIGURACE
# ================================================================================
try:
    trained_model_classic = state_knet
    trained_model_classic.eval()
    print("INFO: KalmanNet připraven.")
except NameError:
    raise NameError("Chyba: 'state_knet' neexistuje.")

print("Inicializuji filtry...")
ukf_filter = Filters.UnscentedKalmanFilter(sys_model) # Tvá robustní verze
ekf_filter = Filters.ExtendedKalmanFilter(sys_model)  # Moje opravená verze výše

# ================================================================================
# 1. EVALUACE NA TEST DATASETU (Full Trajectories)
# ================================================================================
mse_results = {'KNet': [], 'UKF': [], 'EKF': [], 'GPS': []}

print(f"\nSpouštím test na {len(test_data_raw)} trajektoriích...")

for i, traj in enumerate(test_data_raw):
    print(f"\n--- Trajektorie {i+1} / {len(test_data_raw)} ---")

    # 1. Příprava dat (Full Batch)
    gt_raw = traj['ground_truth'].float()
    gps_raw = traj['filtered_gps'].float()
    imu_raw = traj['imu'].float()
    odo_raw = traj['filtered_wheel'].float()

    T_len = gt_raw.shape[0]

    # Vstup U [T, 4]
    v_l = torch.nan_to_num(odo_raw[:, 0], nan=0.0)
    v_r = torch.nan_to_num(odo_raw[:, 1], nan=0.0)
    u_full = torch.stack((v_l, v_r, imu_raw[:, 2], imu_raw[:, 3]), dim=1).
 ↪to(DEVICE)
```

```
    # Měření Y [T, 2]
    y_full = gps_raw.to(DEVICE)

    # Ground Truth X [T, 6] (použijeme jen pozici pro MSE)
    x_true = torch.zeros(T_len, 6).to(DEVICE)
    x_true[:, :3] = gt_raw[:, :3] # Copy px, py, theta

    # Počáteční stavy
    x0_row = x_true[0, :].unsqueeze(0) # [1, 6]
    x0_col = x_true[0, :].unsqueeze(1) # [6, 1]

    # --- BĚH FILTRŮ ---

    # A. KalmanNet
    trained_model_classic.reset(batch_size=1, initial_state=x0_row)
    knet_path = [x0_row]
    with torch.no_grad():
        for t in range(1, T_len):
            x_est = trained_model_classic.step(y_full[t].unsqueeze(0),␣
↪u_full[t].unsqueeze(0))
            knet_path.append(x_est)
    x_knet = torch.cat(knet_path, dim=0)

    # B. UKF
    try:
        res_ukf = ukf_filter.process_sequence(y_full, u_seq=u_full, Ex0=x0_col,␣
↪P0=sys_model.P0)
        x_ukf = res_ukf['x_filtered']
    except Exception as e:
        print(f"UKF Error: {e}")
        x_ukf = torch.zeros_like(x_true)

    # C. EKF
    try:
        res_ekf = ekf_filter.process_sequence(y_full, u_seq=u_full, Ex0=x0_col,␣
↪P0=sys_model.P0)
        x_ekf = res_ekf['x_filtered']
    except Exception as e:
        print(f"EKF Error: {e}")
        x_ekf = torch.zeros_like(x_true)

    # --- VÝPOČET MSE (jen pozice x, y) ---
    def calc_mse(pred, target):
        return F.mse_loss(pred[:, :2], target[:, :2]).item()

    mse_k = calc_mse(x_knet, x_true)
    mse_u = calc_mse(x_ukf, x_true)
```

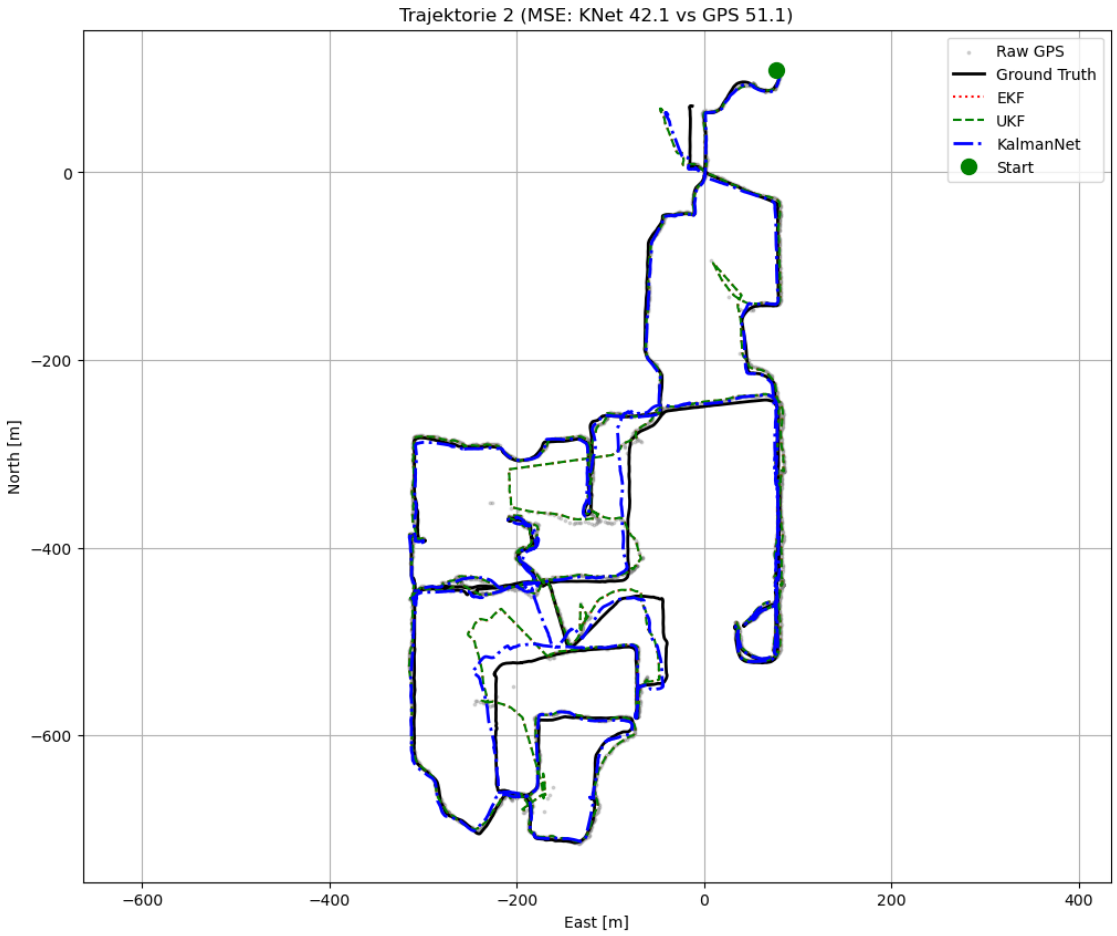
```
    mse_e = calc_mse(x_ekf, x_true)

    # D. GPS Baseline (ignorujeme NaN)
    mask = ~torch.isnan(y_full[:, 0])
    if mask.sum() > 0:
        mse_g = F.mse_loss(y_full[mask], x_true[mask, :2]).item()
    else:
        mse_g = float('nan')

    # Uložení
    mse_results['KNet'].append(mse_k)
    mse_results['UKF'].append(mse_u)
    mse_results['EKF'].append(mse_e)
    mse_results['GPS'].append(mse_g)

    print(f"MSE -> KNet: {mse_k:.2f} | UKF: {mse_u:.2f} | EKF: {mse_e:.2f} |␣
↪Raw GPS: {mse_g:.2f}")

    # --- VIZUALIZACE ---
    # Převod na CPU numpy
    gt_np = x_true.cpu().numpy()
    knet_np = x_knet.cpu().numpy()
    ukf_np = x_ukf.cpu().numpy()
    ekf_np = x_ekf.cpu().numpy()
    gps_np = y_full.cpu().numpy()

    plt.figure(figsize=(12, 10))
    plt.title(f"Trajektorie {i+1} (MSE: KNet {mse_k:.1f} vs GPS {mse_g:.1f})")

    # Raw GPS (šedé tečky)
    plt.scatter(gps_np[:, 0], gps_np[:, 1], c='gray', s=3, alpha=0.3,␣
↪label='Raw GPS')

    # Ground Truth
    plt.plot(gt_np[:, 0], gt_np[:, 1], 'k-', linewidth=2, label='Ground Truth')

    # Filtry
    plt.plot(ekf_np[:, 0], ekf_np[:, 1], 'r:', linewidth=1.5, label='EKF')
    plt.plot(ukf_np[:, 0], ukf_np[:, 1], 'g--', linewidth=1.5, label='UKF')
    plt.plot(knet_np[:, 0], knet_np[:, 1], 'b-.', linewidth=2,␣
↪label='KalmanNet')

    # Start
    plt.plot(gt_np[0, 0], gt_np[0, 1], 'go', markersize=10, label='Start')

    plt.legend()
    plt.grid(True)
```

```
    plt.axis('equal')
    plt.xlabel('East [m]')
    plt.ylabel('North [m]')
    plt.show()

# ================================================================================
# 2. FINÁLNÍ TABULKA
# ================================================================================
def get_avg(lst): return np.nanmean(lst)

print("\n" + "="*80)
print(f"FINÁLNÍ VÝSLEDKY NA TESTOVACÍM DATASETU ({len(test_data_raw)} jízdy)")
print("="*80)
print(f"{'Metoda':<20} | {'MSE [m^2]':<15} | {'RMSE [m]':<15} | {'Zlepšení vs␣
 ↪GPS':<15}")
print("-" * 75)

avg_gps = get_avg(mse_results['GPS'])
methods = ['EKF', 'UKF', 'KNet']

# GPS Řádek
print(f"{'Raw GPS':<20} | {avg_gps:<15.2f} | {np.sqrt(avg_gps):<15.2f} | {'-':
 ↪<15}")

# Ostatní
for m in methods:
    avg_mse = get_avg(mse_results[m])
    avg_rmse = np.sqrt(avg_mse)
    imp = ((avg_gps - avg_mse) / avg_gps) * 100
    print(f"{m:<20} | {avg_mse:<15.2f} | {avg_rmse:<15.2f} | {imp:+.1f}%")

print("="*80)
```

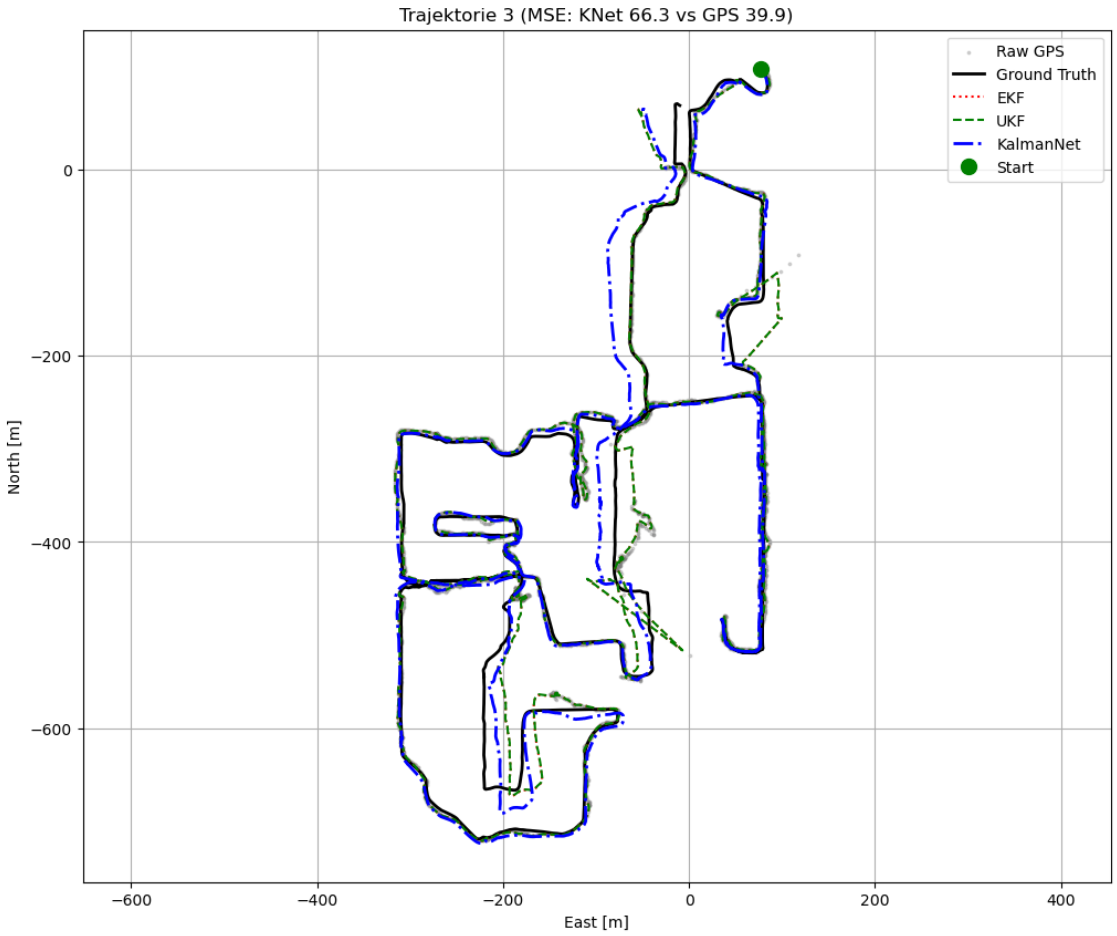
```
INFO: KalmanNet připraven.
Inicializuji filtry…

Spouštím test na 3 trajektoriích…

--- Trajektorie 1 / 3 ---
MSE -> KNet: 181.00 | UKF: 440.28 | EKF: 443.11 | Raw GPS: 164.46
```




```
--- Trajektorie 2 / 3 ---
MSE -> KNet: 42.14 | UKF: 221.81 | EKF: 221.68 | Raw GPS: 51.13
```




```
--- Trajektorie 3 / 3 ---
MSE -> KNet: 66.33 | UKF: 128.71 | EKF: 128.44 | Raw GPS: 39.94
```




```
================================================================================
FINÁLNÍ VÝSLEDKY NA TESTOVACÍM DATASETU (3 jízdy)
================================================================================
Metoda               | MSE [m^2]       | RMSE [m]        | Zlepšení vs GPS
--------------------------------------------------------------------------------
Raw GPS              | 85.18           | 9.23            | -
EKF                  | 264.41          | 16.26           | -210.4%
UKF                  | 263.60          | 16.24           | -209.5%
KNet                 | 96.49           | 9.82            | -13.3%
================================================================================
```