Linear_velocity_integration

December 20, 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")
print(f"device: {device}")
```

```
device: cuda
```

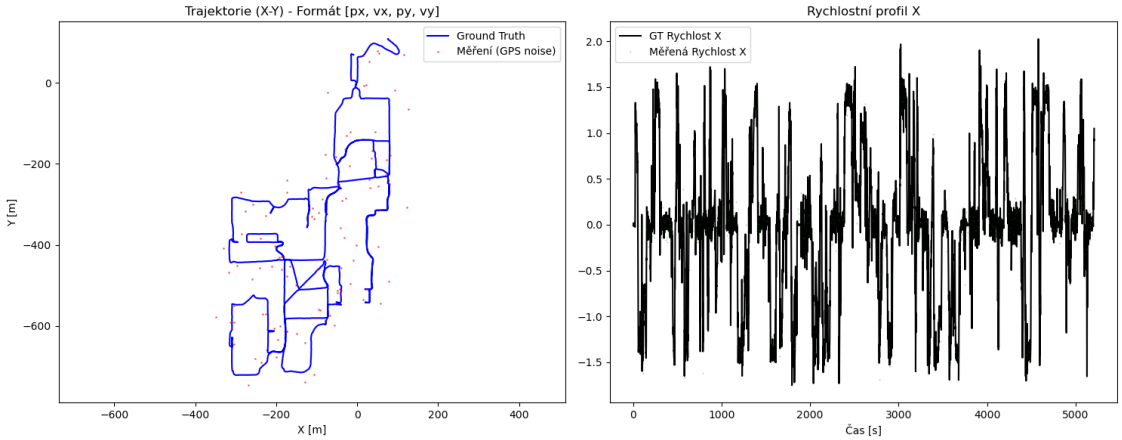
[3]:
```
import pandas as pd
import numpy as np
import torch
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# === 1. KONFIGURACE ===
file_path = 'ground_truth/groundtruth_2012-01-22.csv'
DT = 1.0  # Časový krok

# Parametry šumu pro generování trénovacích dat (Simulace měření)
# Abychom replikovali autory, musíme vygenerovat měření, která odpovídají
# jejich matici R: diag([700, 0.01, 700, 0.01]).
# Tedy: Velký šum na pozici (GPS), malý na rychlosti (Odometrie).
std_pos = np.sqrt(700.0)  # cca 26.4 metrů
std_vel = np.sqrt(0.01)   # 0.1 m/s

# === 2. NAČTENÍ A PŘÍPRAVA SUROVÝCH DAT ===
print(f"Načítám soubor: {file_path}...")
try:
    df = pd.read_csv(file_path, header=None, names=['time', 'x', 'y', 'z', 'r',␣
 ↪'p', 'h'])
    raw_time = df['time'].values / 1e6
    raw_time = raw_time - raw_time[0]
    raw_pos = df[['x', 'y']].values

    # --- Resampling (Interpolace) ---
    t_resampled = np.arange(0, raw_time[-1], DT)
    interp_func = interp1d(raw_time, raw_pos, kind='linear', axis=0,␣
 ↪fill_value="extrapolate")
```

```
    pos_resampled = interp_func(t_resampled) # [N, 2] -> (x, y)

    # --- Výpočet rychlosti (Derivace) ---
    vel_resampled = np.zeros_like(pos_resampled)
    vel_resampled[1:] = (pos_resampled[1:] - pos_resampled[:-1]) / DT
    vel_resampled[0] = vel_resampled[1]

    # === 3. SKLÁDÁNÍ DO FORMÁTU AUTORŮ [px, vx, py, vy] ===
    # Toto je ta klíčová změna!
    # Sloupec 0: Pozice X
    # Sloupec 1: Rychlost X
    # Sloupec 2: Pozice Y
    # Sloupec 3: Rychlost Y

    GT_DATA = np.stack((
        pos_resampled[:, 0], # px
        vel_resampled[:, 0], # vx
        pos_resampled[:, 1], # py
        vel_resampled[:, 1]  # vy
    ), axis=1)

    print(f"\n Data seřazena do formátu [px, vx, py, vy]. Shape: {GT_DATA.
↪shape}")

    # === 4. KONVERZE DO PYTORCH A GENEROVÁNÍ MĚŘENÍ ===
    X_target = torch.from_numpy(GT_DATA).float()
    N_samples = X_target.shape[0]

    # Generování šumu měření podle specifikace autorů
    # noise_std vektor: [std_px, std_vx, std_py, std_vy]
    noise_std_vec = torch.tensor([std_pos, std_vel, std_pos, std_vel])

    # Y = X + Noise
    # Rozšiřujeme noise_std_vec na velikost dat
    noise = torch.randn(N_samples, 4) * noise_std_vec
    Y_measured = X_target + noise

    print(f"Měření vygenerována (Simulace GPS+Odo). Shape: {Y_measured.shape}")
    print(f"  -> Šum polohy (std): {std_pos:.2f} m")
    print(f"  -> Šum rychlosti (std): {std_vel:.2f} m/s")

    # === 5. ROZDĚLENÍ TRAIN / VAL / TEST ===
    n_train = int(0.6 * N_samples)
    n_val = int(0.2 * N_samples)
    n_test = N_samples - n_train - n_val

    train_input = Y_measured[:n_train]
```

```
    train_target = X_target[:n_train]

    val_input = Y_measured[n_train:n_train+n_val]
    val_target = X_target[n_train:n_train+n_val]

    test_input = Y_measured[n_train+n_val:]
    test_target = X_target[n_train+n_val:]

    print(f"Dataset rozdělen: Train={len(train_input)}, Val={len(val_input)},␣
↪Test={len(test_input)}")

    # === 6. VIZUALIZACE KONTROLY ===
    fig, ax = plt.subplots(1, 2, figsize=(15, 6))

    # Pro vizualizaci musíme brát správné sloupce z nového GT_DATA
    # X=0, VX=1, Y=2, VY=3

    # Trajektorie (X vs Y) -> Sloupec 0 vs Sloupec 2
    ax[0].plot(GT_DATA[:, 0], GT_DATA[:, 2], 'b-', label='Ground Truth')
    # Vykreslíme i kousek měření (jen pro představu šumu), bereme jen každý 50.␣
↪bod ať to není čmouha
    ax[0].plot(Y_measured[::50, 0], Y_measured[::50, 2], 'r.', markersize=2,␣
↪alpha=0.5, label='Měření (GPS noise)')
    ax[0].set_title('Trajektorie (X-Y) - Formát [px, vx, py, vy]')
    ax[0].set_xlabel('X [m]')
    ax[0].set_ylabel('Y [m]')
    ax[0].axis('equal')
    ax[0].legend()

    # Rychlost X (VX) -> Sloupec 1
    ax[1].plot(t_resampled, GT_DATA[:, 1], 'k-', label='GT Rychlost X')
    ax[1].plot(t_resampled[::10], Y_measured[::10, 1], 'g.', markersize=1,␣
↪alpha=0.3, label='Měřená Rychlost X')
    ax[1].set_title('Rychlostní profil X')
    ax[1].set_xlabel('Čas [s]')
    ax[1].legend()

    plt.tight_layout()
    plt.show()

except FileNotFoundError:
    print(f" CHYBA: Soubor '{file_path}' nenalezen.")
except Exception as e:
    print(f" CHYBA: {e}")
```

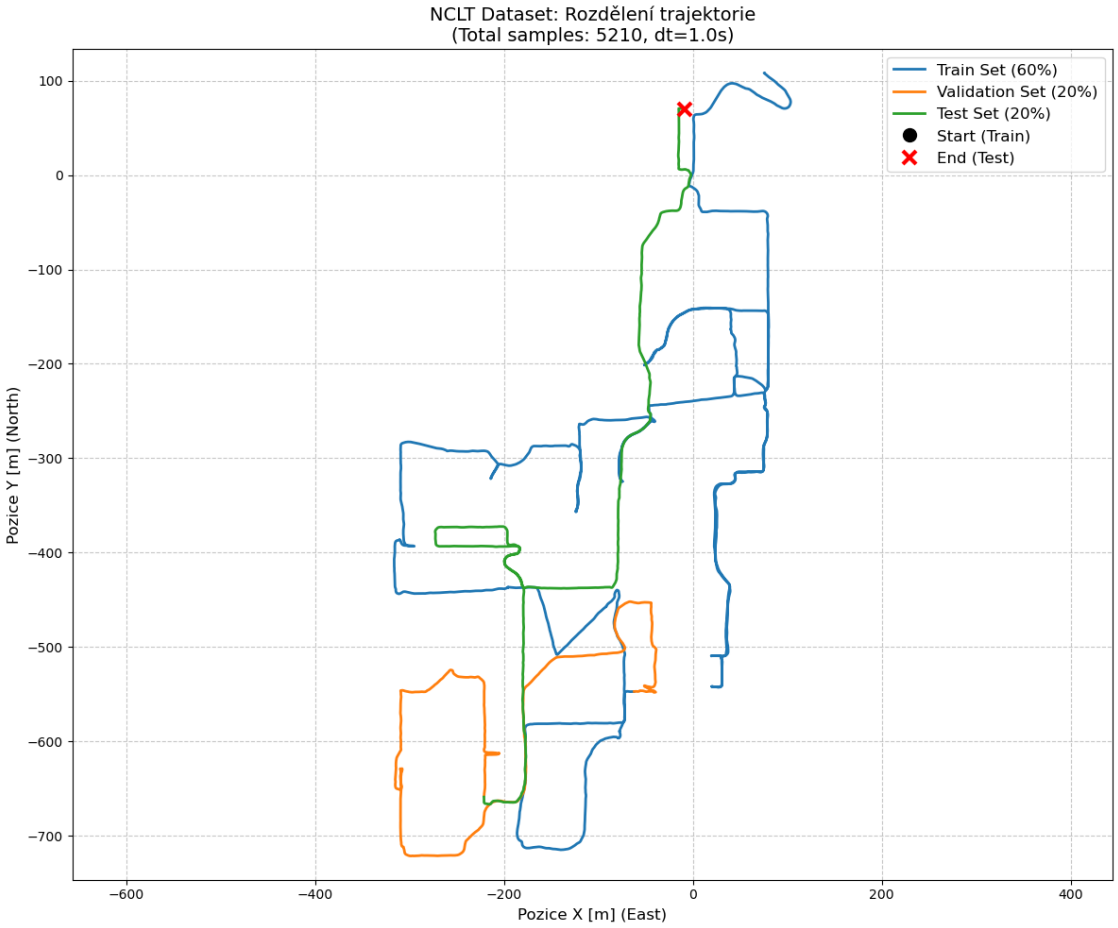
```
Načítám soubor: ground_truth/groundtruth_2012-01-22.csv…
/tmp/ipykernel_104069/1253375658.py:21: DtypeWarning: Columns (1,2,3,4,5,6) have
```

```
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv(file_path, header=None, names=['time', 'x', 'y', 'z', 'r',
'p', 'h'])


 Data seřazena do formátu [px, vx, py, vy]. Shape: (5210, 4)
Měření vygenerována (Simulace GPS+Odo). Shape: torch.Size([5210, 4])
  -> Šum polohy (std): 26.46 m
  -> Šum rychlosti (std): 0.10 m/s
Dataset rozdělen: Train=3126, Val=1042, Test=1042
```



[4]:
```python
import matplotlib.pyplot as plt

# === VIZUALIZACE ROZDĚLENÍ DATASETU ===

plt.figure(figsize=(12, 10))

# 1. Vykreslení Trénovací části
# Musíme převést tenzory zpět na numpy (a pro jistotu na CPU, kdyby byly na GPU)
plt.plot(train_target[:, 0].cpu().numpy(),
         train_target[:, 2].cpu().numpy(),
         label='Train Set (60%)', color='#1f77b4', linewidth=2)

# 2. Vykreslení Validační části
plt.plot(val_target[:, 0].cpu().numpy(),
         val_target[:, 2].cpu().numpy(),
         label='Validation Set (20%)', color='#ff7f0e', linewidth=2)

# 3. Vykreslení Testovací části
plt.plot(test_target[:, 0].cpu().numpy(),
         test_target[:, 2].cpu().numpy(),
         label='Test Set (20%)', color='#2ca02c', linewidth=2)
```

```
# Zvýraznění začátku a konce
plt.plot(train_target[0, 0].cpu(), train_target[0, 2].cpu(), 'ko',␣
 ↪markersize=10, label='Start (Train)')
plt.plot(test_target[-1, 0].cpu(), test_target[-1, 2].cpu(), 'rx',␣
 ↪markersize=10, markeredgewidth=3, label='End (Test)')

# Formátování grafu
plt.title(f'NCLT Dataset: Rozdělení trajektorie\n(Total samples: {N_samples},␣
 ↪dt={DT}s)', fontsize=14)
plt.xlabel('Pozice X [m] (East)', fontsize=12)
plt.ylabel('Pozice Y [m] (North)', fontsize=12)
plt.legend(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.axis('equal')  # Důležité: Aby mapa nebyla deformovaná

plt.tight_layout()
plt.show()
```



1 Model definition

[5]:
```python
import torch
import math
import Systems # Předpokládám, že toto je tvůj modul s třídou DynamicSystem

# Nastavení zařízení
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

##########################
### Design Parameters ###
### (Dle autorů)      ###
##########################

# 1. Rozměry
# Stav: [px, vx, py, vy] (Všimni si pořadí!)
m = 4
n = 4
delta_t = 1.0 # Sampling NCLT datasetu

# 2. Dynamika (F)
# x_{t+1} = x_t + v_t * dt
# v_{t+1} = v_t
F_dim = torch.tensor([[1.0, delta_t],
                      [0.0, 1.0]])

# Vytvoření 4x4 matice pro X a Y osy
# Výsledek je blokově diagonální, což znamená pořadí stavů: [px, vx, py, vy]
F_design = torch.block_diag(F_dim, F_dim).float()

# 3. Měření (H)
# Měříme vše: [px, vx, py, vy]
H_design = torch.eye(n).float()

# 4. Šum procesu (Q)
# Autoři definují Q diagonálně závislé na čase
lambda_q_mod = 1.0
Q_dim = torch.diagflat(torch.tensor([delta_t, delta_t]))
Q_design = (lambda_q_mod**2) * torch.block_diag(Q_dim, Q_dim).float()

# 5. Šum měření (R) - TOTO JE KLÍČOVÉ PRO REPLIKACI
# 700 = Obrovská chyba polohy (GPS) -> Filtr ji bude potlačovat
# 0.01 = Malá chyba rychlosti (Odometrie) -> Filtr jí bude věřit
# Pořadí odpovídá stavům: [Pos_X, Vel_X, Pos_Y, Vel_Y]
R_design = torch.tensor([[700.0, 0.0,   0.0,   0.0],
                         [0.0,   0.01,  0.0,   0.0],
                         [0.0,   0.0,   700.0, 0.0],
```

```
                          [0.0,   0.0,   0.0,   0.01]]).float()

# 6. Počáteční podmínky
# Autoři začínají na nule
m1x_0 = torch.zeros(m, 1).float()       # Mean
m2x_0 = torch.eye(m).float() * 1e-5

print("\nInicializuji systém dle parametrů KalmanNet (Author's replication)...")
print(f"Dimenze stavu: {m}, Dimenze měření: {n}")
print(f"R (diagonal): {torch.diagonal(R_design)}")

# === INICIALIZACE SYSTÉMŮ ===

# Sys True (Generativní model - pokud bychom generovali syntetická data)
# Pro NCLT data se toto tolik nepoužije (data načítáme), ale pro konzistenci␣
 ↪nastavíme stejně.
sys_true = Systems.DynamicSystem(
    state_dim=m, obs_dim=n,
    Ex0=m1x_0, P0=m2x_0,
    Q=Q_design, R=R_design,
    F=F_design, H=H_design,
    device=device
)

# Sys Model (To, co ví KalmanNet/Filtr)
sys_model = Systems.DynamicSystem(
    state_dim=m, obs_dim=n,
    Ex0=m1x_0, P0=m2x_0,
    Q=Q_design, R=R_design,
    F=F_design, H=H_design,
    device=device
)

print("... Systémy inicializovány.")
print("POZOR: Tento model očekává pořadí stavů [px, vx, py, vy].")
```

```
Inicializuji systém dle parametrů KalmanNet (Author's replication)…
Dimenze stavu: 4, Dimenze měření: 4
R (diagonal): tensor([7.0000e+02, 1.0000e-02, 7.0000e+02, 1.0000e-02])
… Systémy inicializovány.
POZOR: Tento model očekává pořadí stavů [px, vx, py, vy].
```

[6]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader

# === 1. KONFIGURACE ===
```

```
TRAIN_SEQ_LEN = 50    # Délka sekvence pro RNN
VAL_SEQ_LEN = 200
TEST_SEQ_LEN = n_test
STRIDE = 10     # Posun okna (pro trénink s překryvem)
BATCH_SIZE = 32 # Velikost dávky

def create_sequences(X, Y, seq_len, stride=1):
    """
    Rozseká dlouhé tenzory [Total_Len, Dim] na sekvence [N_seq, Seq_Len, Dim].
    """
    xs = []
    ys = []
    num_samples = X.shape[0]

    for i in range(0, num_samples - seq_len + 1, stride):
        x_seq = X[i : i+seq_len, :]
        y_seq = Y[i : i+seq_len, :]
        xs.append(x_seq)
        ys.append(y_seq)

    if len(xs) == 0:
        return torch.empty(0, seq_len, X.shape[1]), torch.empty(0, seq_len, Y.
 ↪shape[1])

    return torch.stack(xs), torch.stack(ys)

def clean_sequences(X_seq, Y_seq, name="Dataset"):
    """
    Filtruje sekvence, které obsahují jakékoliv NaN nebo Inf hodnoty.
    """
    if X_seq.numel() == 0:
        print(f" {name}: Prázdný vstup!")
        return X_seq, Y_seq

    # Zkontrolujeme NaN/Inf pro každou sekvenci zvlášť
    # X_seq shape: [N, Seq_Len, Dim] -> reshape na [N, -1] pro kontrolu celého␣
 ↪řádku
    is_nan_x = torch.isnan(X_seq).reshape(X_seq.shape[0], -1).any(dim=1)
    is_inf_x = torch.isinf(X_seq).reshape(X_seq.shape[0], -1).any(dim=1)

    is_nan_y = torch.isnan(Y_seq).reshape(Y_seq.shape[0], -1).any(dim=1)
    is_inf_y = torch.isinf(Y_seq).reshape(Y_seq.shape[0], -1).any(dim=1)

    # Maska vadných dat (pokud je chyba v X nebo v Y)
    invalid_mask = is_nan_x | is_inf_x | is_nan_y | is_inf_y

    # Vybereme jen ta dobrá
```

```
    valid_mask = ~invalid_mask

    X_clean = X_seq[valid_mask]
    Y_clean = Y_seq[valid_mask]

    n_dropped = invalid_mask.sum().item()
    if n_dropped > 0:
        print(f"  {name}: Odstraněno {n_dropped} vadných sekvencí (NaN/Inf). Zbývá: {len(X_clean)}")
    else:
        print(f"  {name}: Data jsou čistá. ({len(X_clean)} sekvencí)")

    return X_clean, Y_clean

print("--- ZPRACOVÁNÍ DAT ---")

# === 2. TVORBA SEKVENCÍ A PŘETYPOVÁNÍ ===
# Používáme .float() hned zde, abychom vyřešili Double vs Float error
print("Generuji sekvence...")
train_X_raw, train_Y_raw = create_sequences(train_target.float(), train_input.float(), TRAIN_SEQ_LEN, STRIDE)
val_X_raw, val_Y_raw = create_sequences(val_target.float(), val_input.float(), VAL_SEQ_LEN, VAL_SEQ_LEN)
test_X_raw, test_Y_raw = create_sequences(test_target.float(), test_input.float(), TEST_SEQ_LEN, TEST_SEQ_LEN)

# === 3. ČIŠTĚNÍ DAT (NAN/INF FILTER) ===
print("\nFiltruji NaN hodnoty...")
train_X_seq, train_Y_seq = clean_sequences(train_X_raw, train_Y_raw, "Train")
val_X_seq, val_Y_seq = clean_sequences(val_X_raw, val_Y_raw, "Val")
test_X_seq, test_Y_seq = clean_sequences(test_X_raw, test_Y_raw, "Test")

# === 4. VYTVOŘENÍ DATALOADERŮ ===
print("\nVytvářím DataLoadery...")
# Train: Shuffle=True
train_dataset = TensorDataset(train_X_seq, train_Y_seq)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)

# Val/Test: Shuffle=False
val_dataset = TensorDataset(val_X_seq, val_Y_seq)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

test_dataset = TensorDataset(test_X_seq, test_Y_seq)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)

print(f"\n HOTOVO. Připraveno k tréninku.")
print(f"Batch shape: {next(iter(train_loader))[0].shape}")
```

```
--- ZPRACOVÁNÍ DAT ---
Generuji sekvence…

Filtruji NaN hodnoty…
  Train: Odstraněno 1 vadných sekvencí (NaN/Inf). Zbývá: 307
  Val: Data jsou čistá. (5 sekvencí)
  Test: Data jsou čistá. (1 sekvencí)

Vytvářím DataLoadery…

  HOTOVO. Připraveno k tréninku.
Batch shape: torch.Size([32, 50, 4])
```

[7]:
```python
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
import os
import random
import csv
from datetime import datetime
import pandas as pd
from copy import deepcopy
from state_NN_models import StateKalmanNet

# Nastavení seedu pro reprodukovatelnost tohoto běhu
torch.manual_seed(42)
np.random.seed(42)
random.seed(42)
state_knet = StateKalmanNet(sys_model, device=device,␣
 ↪hidden_size_multiplier=10,output_layer_multiplier=4,num_gru_layers=1,gru_hidden_dim_multipl
 ↪to(device)
trainer.train_state_KalmanNet(
    model=state_knet,
    train_loader=train_loader,
    val_loader=val_loader,
    device=device,
    epochs=100,
    lr=1e-3,
    early_stopping_patience=30
)
```

```
DEBUG: Layer 'output_final_linear.0' initialized near zero (Start K=0).
New best model saved! Epoch [1/100], Train Loss: 4671273433779738.000000, Val
Loss: 75.407059
New best model saved! Epoch [2/100], Train Loss: 3886.009692, Val Loss:
59.809681
New best model saved! Epoch [4/100], Train Loss: 1051.350888, Val Loss:
```

```
47.343712
Epoch [5/100] | Train Loss: 245.4994 (Pos: 0.00, Vel: 0.00) | Val Loss: 37.6954
| Val MSE: 37.70
Epoch [4/100], Train Loss: 1051.350888, Val Loss: 47.343712
New best model saved! Epoch [5/100], Train Loss: 245.499406, Val Loss: 37.695389
New best model saved! Epoch [6/100], Train Loss: 144.206699, Val Loss: 25.464903
New best model saved! Epoch [7/100], Train Loss: 80.683084, Val Loss: 13.837410
New best model saved! Epoch [8/100], Train Loss: 44.341483, Val Loss: 5.066244
New best model saved! Epoch [9/100], Train Loss: 23.390773, Val Loss: 2.847550
Epoch [10/100] | Train Loss: 15.2109 (Pos: 0.00, Vel: 0.00) | Val Loss: 1.4030 |
Val MSE: 1.40
Epoch [9/100], Train Loss: 23.390773, Val Loss: 2.847550
New best model saved! Epoch [10/100], Train Loss: 15.210896, Val Loss: 1.402951
New best model saved! Epoch [11/100], Train Loss: 15.072844, Val Loss: 1.043615
Epoch [15/100] | Train Loss: 12.0884 (Pos: 0.00, Vel: 0.00) | Val Loss: 14.9769
| Val MSE: 14.98
Epoch [11/100], Train Loss: 15.072844, Val Loss: 1.043615
New best model saved! Epoch [16/100], Train Loss: 13.767892, Val Loss: 0.750487
Epoch [20/100] | Train Loss: 10.4835 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.9846 |
Val MSE: 0.98
Epoch [16/100], Train Loss: 13.767892, Val Loss: 0.750487
Epoch [25/100] | Train Loss: 8.8151 (Pos: 0.00, Vel: 0.00) | Val Loss: 2.8375 |
Val MSE: 2.84
Epoch [16/100], Train Loss: 13.767892, Val Loss: 0.750487
New best model saved! Epoch [27/100], Train Loss: 7.489140, Val Loss: 0.709064
New best model saved! Epoch [28/100], Train Loss: 5.963594, Val Loss: 0.690415
Epoch [30/100] | Train Loss: 5.8913 (Pos: 0.00, Vel: 0.00) | Val Loss: 1.1085 |
Val MSE: 1.11
Epoch [28/100], Train Loss: 5.963594, Val Loss: 0.690415
New best model saved! Epoch [32/100], Train Loss: 10.311841, Val Loss: 0.551545
Epoch [35/100] | Train Loss: 8.9137 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.5926 |
Val MSE: 0.59
Epoch [32/100], Train Loss: 10.311841, Val Loss: 0.551545
New best model saved! Epoch [36/100], Train Loss: 7.469261, Val Loss: 0.483676
Epoch [40/100] | Train Loss: 7.4267 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.5942 |
Val MSE: 0.59
Epoch [36/100], Train Loss: 7.469261, Val Loss: 0.483676
New best model saved! Epoch [41/100], Train Loss: 6.802444, Val Loss: 0.481015
New best model saved! Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
Epoch [45/100] | Train Loss: 6.9778 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.5975 |
Val MSE: 0.60
Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
Epoch [50/100] | Train Loss: 5.6229 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.9002 |
Val MSE: 0.90
Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
Epoch [55/100] | Train Loss: 5.7036 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.6821 |
Val MSE: 0.68
Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
```

```
Epoch [60/100] | Train Loss: 7.2501 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.8462 |
Val MSE: 0.85
Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
Epoch [65/100] | Train Loss: 10.6048 (Pos: 0.00, Vel: 0.00) | Val Loss: 1.1335 |
Val MSE: 1.13
Epoch [42/100], Train Loss: 7.727523, Val Loss: 0.453355
New best model saved! Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [70/100] | Train Loss: 7.9443 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.5488 |
Val MSE: 0.55
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [75/100] | Train Loss: 4.9642 (Pos: 0.00, Vel: 0.00) | Val Loss: 1.7260 |
Val MSE: 1.73
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [80/100] | Train Loss: 6.7875 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.6783 |
Val MSE: 0.68
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [85/100] | Train Loss: 6.4266 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.7955 |
Val MSE: 0.80
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [90/100] | Train Loss: 6.0664 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.8888 |
Val MSE: 0.89
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
Epoch [95/100] | Train Loss: 7.2530 (Pos: 0.00, Vel: 0.00) | Val Loss: 1.0151 |
Val MSE: 1.02
Epoch [68/100], Train Loss: 8.328999, Val Loss: 0.327155
New best model saved! Epoch [96/100], Train Loss: 6.569129, Val Loss: 0.268119
Epoch [100/100] | Train Loss: 8.6671 (Pos: 0.00, Vel: 0.00) | Val Loss: 0.4103 |
Val MSE: 0.41
Epoch [96/100], Train Loss: 6.569129, Val Loss: 0.268119
Training completed.
Loading best model with validation loss: 0.268119
```

[7]:
```
StateKalmanNet(
  (dnn): DNN_KalmanNet(
    (input_layer): Sequential(
      (0): Linear(in_features=16, out_features=640, bias=True)
      (1): ReLU()
    )
    (gru): GRU(640, 320)
    (output_hidden_layer): Sequential(
      (0): Linear(in_features=320, out_features=64, bias=True)
      (1): ReLU()
    )
    (output_final_linear): Sequential(
      (0): Linear(in_features=64, out_features=16, bias=True)
    )
  )
```

```
)
```

[8]:
```
import torch
import torch.nn.functional as F
import numpy as np
import Filters  # Předpokládám, že máš tento modul
from utils import utils # Předpokládám, že máš utils pro výpočet ANEES

# ================================================================================
# 0. KONFIGURACE A PŘÍPRAVA MODELŮ
# ================================================================================
# Přiřadíme 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.")

# Inicializace klasických filtrů
# Používáme 'sys_model', který obsahuje parametry (Q, R atd.)
print("Inicializuji EKF a UKF...")
ekf_filter = Filters.ExtendedKalmanFilter(sys_model)
ukf_filter = Filters.UnscentedKalmanFilter(sys_model)

# ================================================================================
# 1. VYHODNOCOVACÍ SMYČKA
# ================================================================================
# Seznamy pro ukládání výsledků
mse_knet = []
mse_ekf, anees_ekf = [], []
mse_ukf, anees_ukf = [], []

# Počítadla
traj_idx = 0
total_trajectories = len(test_loader.dataset)

print(f"\nVyhodnocuji {total_trajectories} sekvencí z testovací sady...")
print("POZNÁMKA: Všechny filtry (KNet, EKF, UKF) jsou inicializovány na Ground␣
 ↪Truth.")

with torch.no_grad():
    for x_true_batch, y_meas_batch in test_loader:
        # Iterujeme přes každou trajektorii v batchi zvlášť

        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, Obs_Dim]
            x_true = x_true_batch[i].to(device)   # [Seq_Len, State_Dim]

            seq_len = y_seq.shape[0]

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

            # 2. Pro EKF/UKF: Očekává [State_Dim, 1] (Column vector)
            # Musíme vzít x_true[0] a udělat z něj sloupec
            filter_init_state = x_true[0, :].unsqueeze(1)

            # --- A. KalmanNet (Neural Network) ---
            # Reset stavu sítě s přesnou inicializací
            trained_model_classic.reset(batch_size=1,␣
↪initial_state=knet_init_state)

            knet_preds = []
            knet_preds.append(knet_init_state) # Přidáme počáteční stav

            for t in range(1, seq_len):
                # KNet krok
                y_t = y_seq[t, :].unsqueeze(0) # [1, Obs_Dim]
                x_hat_t = trained_model_classic.step(y_t)
                knet_preds.append(x_hat_t)

            full_x_hat_knet = torch.cat(knet_preds, dim=0) # [Seq_Len,␣
↪State_Dim]

            # --- B. Extended Kalman Filter (EKF) ---
            # Předáváme filter_init_state jako Ex0 (Initial Mean)
            # P0 necháváme obecnou ze systému (počáteční nejistota), nebo␣
↪bychom ji mohli zmenšit,
            # pokud věříme, že GT známe naprosto přesně. Zde necháváme P0␣
↪modelu.
            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']

            # --- C. Unscented Kalman Filter (UKF) ---
```

```
            ukf_res = ukf_filter.process_sequence(y_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']

            # --- VÝPOČET METRIK PRO TUTO TRAJEKTORII ---
            # 1. MSE (Mean Squared Error)
            # Počítáme od indexu 1, nebo 0? Pokud inicializujeme přesně, index␣
↪0 má error 0.
            # Standardně se počítá celá trajektorie, nebo se přechodový jev␣
↪(začátek) ignoruje.
            # Zde počítáme od t=1 dál, abychom hodnotili schopnost sledování,␣
↪ne inicializaci.
            mse_val_knet = F.mse_loss(full_x_hat_knet[1:], x_true[1:]).item()
            mse_val_ekf = F.mse_loss(full_x_hat_ekf[1:], x_true[1:]).item()
            mse_val_ukf = F.mse_loss(full_x_hat_ukf[1:], x_true[1:]).item()

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

            # 2. ANEES
            def calc_anees_traj(x_true_seq, x_hat_seq, P_hat_seq):
                return utils.calculate_anees_vectorized(
                    x_true_seq.unsqueeze(0).cpu(),
                    x_hat_seq.unsqueeze(0).cpu(),
                    P_hat_seq.unsqueeze(0).cpu()
                )

            anees_val_ekf = calc_anees_traj(x_true, full_x_hat_ekf,␣
↪full_P_hat_ekf)
            anees_val_ukf = calc_anees_traj(x_true, full_x_hat_ukf,␣
↪full_P_hat_ukf)

            anees_ekf.append(anees_val_ekf)
            anees_ukf.append(anees_val_ukf)

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

# ================================================================================
# 2. FINÁLNÍ VÝPIS VÝSLEDKŮ
# ================================================================================
def avg(lst): return np.mean(lst) if len(lst) > 0 else 0.0
```

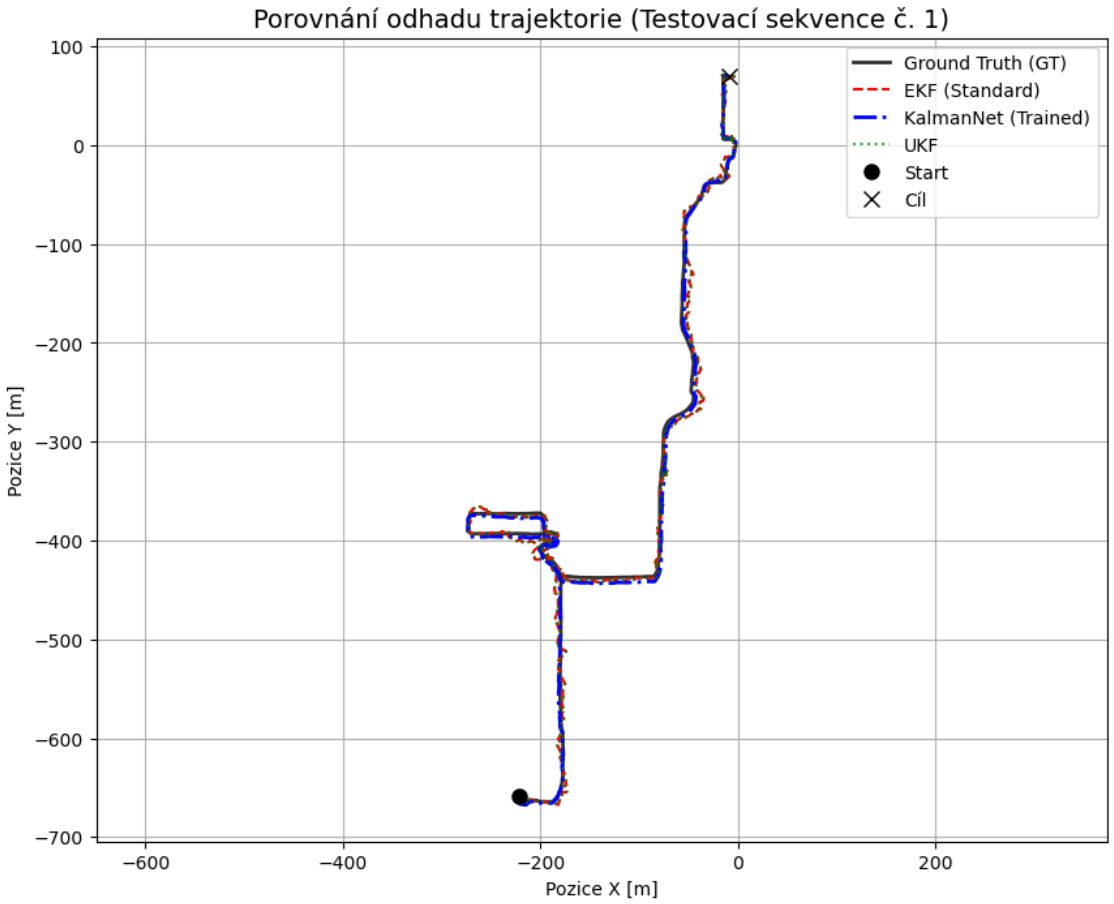
```
state_dim = sys_model.state_dim

print("\n" + "="*80)
print(f"FINÁLNÍ VÝSLEDKY NA NCLT DATASETU (Start z Ground Truth)")
print("="*80)
print(f"{'Model':<30} | {'Průměrné MSE':<20} | {'Průměrný ANEES':<20}")
print("-" * 80)
print(f"Dimenze stavu pro ANEES: {state_dim} (Ideální ANEES   {state_dim})")
print("-" * 80)

# KalmanNet
print(f"{'KalmanNet (NN)':<30} | {avg(mse_knet):<20.4f} | {'N/A (No Cov)':<20}")

# EKF
print(f"{'EKF (Standard)':<30} | {avg(mse_ekf):<20.4f} | {avg(anees_ekf):<20.
 ↪4f}")

# UKF
print(f"{'UKF (Standard)':<30} | {avg(mse_ukf):<20.4f} | {avg(anees_ukf):<20.
 ↪4f}")

print("="*80)
```

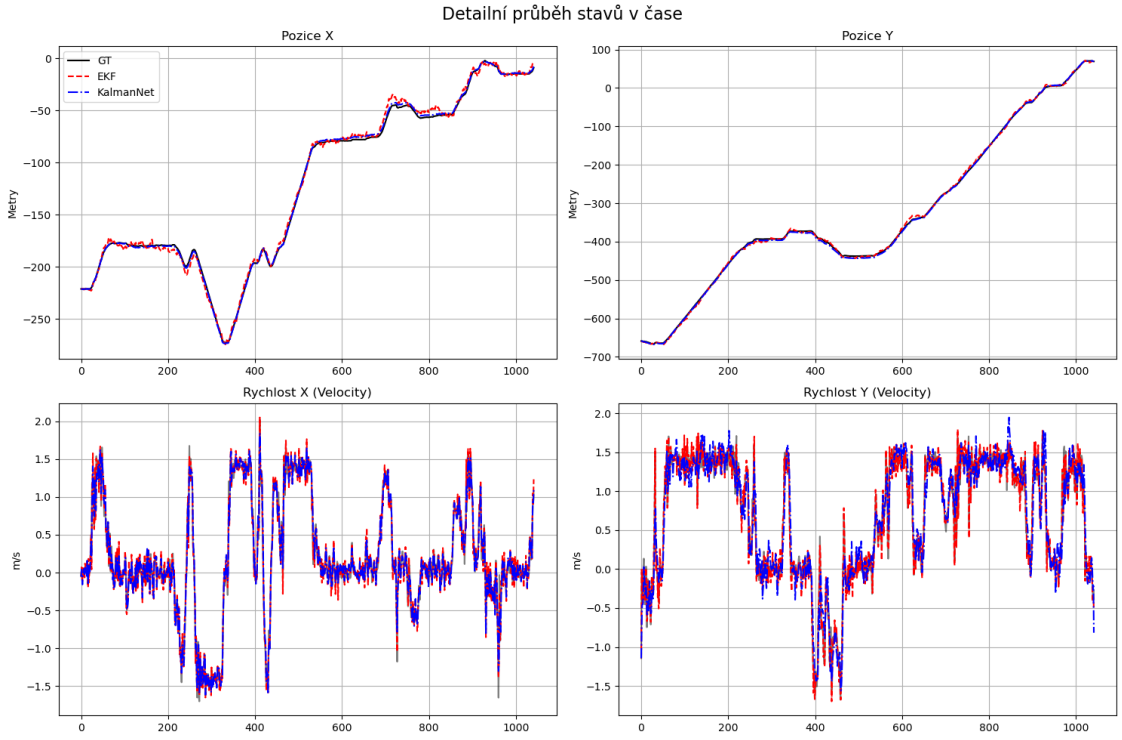
```
INFO: KalmanNet (state_knet) připraven k testování.
Inicializuji EKF a UKF…

Vyhodnocuji 1 sekvencí z testovací sady…
POZNÁMKA: Všechny filtry (KNet, EKF, UKF) jsou inicializovány na Ground Truth.

================================================================================
FINÁLNÍ VÝSLEDKY NA NCLT DATASETU (Start z Ground Truth)
================================================================================
Model                          | Průměrné MSE         | Průměrný ANEES
--------------------------------------------------------------------------------
Dimenze stavu pro ANEES: 4 (Ideální ANEES   4)
--------------------------------------------------------------------------------
KalmanNet (NN)                 | 2.2804               | N/A (No Cov)
EKF (Standard)                 | 7.0031               | 3.0127
UKF (Standard)                 | 7.0030               | 3.0126
================================================================================
```

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

# === VIZUALIZACE POSLEDNÍ TESTOVACÍ TRAJEKTORIE ===

# 1. Převod tenzorů na NumPy pole (pro vykreslování)
# Bereme data z poslední iterace předchozí smyčky
```

```
gt_np = x_true.cpu().numpy()
knet_np = full_x_hat_knet.cpu().numpy()
ekf_np = full_x_hat_ekf.cpu().numpy()
# (Pokud chceš i UKF, odkomentuj:)
ukf_np = full_x_hat_ukf.cpu().numpy()

# Časová osa (kroky)
time_steps = np.arange(gt_np.shape[0])

# Připomínka indexů (Author's format):
# 0: Pos X, 1: Vel X, 2: Pos Y, 3: Vel Y
idx_px, idx_vx, idx_py, idx_vy = 0, 1, 2, 3

# === GRAF 1: 2D TRAJEKTORIE (MAPA) ===
plt.figure(figsize=(10, 8))
plt.title(f"Porovnání odhadu trajektorie (Testovací sekvence č. {traj_idx})",␣
 ↪fontsize=14)

# Ground Truth
plt.plot(gt_np[:, idx_px], gt_np[:, idx_py], 'k-', linewidth=2, label='Ground␣
 ↪Truth (GT)', alpha=0.8)

# EKF
plt.plot(ekf_np[:, idx_px], ekf_np[:, idx_py], 'r--', linewidth=1.5, label='EKF␣
 ↪(Standard)')

# KalmanNet
plt.plot(knet_np[:, idx_px], knet_np[:, idx_py], 'b-.', linewidth=2,␣
 ↪label='KalmanNet (Trained)')

# (Volitelně UKF)
plt.plot(ukf_np[:, idx_px], ukf_np[:, idx_py], 'g:', linewidth=1.5,␣
 ↪label='UKF', alpha=0.7)

# Start a Cíl
plt.plot(gt_np[0, idx_px], gt_np[0, idx_py], 'ko', markersize=8, label='Start')
plt.plot(gt_np[-1, idx_px], gt_np[-1, idx_py], 'kx', markersize=8, label='Cíl')

plt.xlabel("Pozice X [m]")
plt.ylabel("Pozice Y [m]")
plt.legend()
plt.grid(True)
plt.axis('equal') # Aby mapa nebyla deformovaná
plt.show()

# === GRAF 2: DETAILNÍ PRŮBĚHY STAVŮ ===
fig, axs = plt.subplots(2, 2, figsize=(15, 10))
```

```
fig.suptitle("Detailní průběh stavů v čase", fontsize=16)

# --- Pozice X ---
axs[0, 0].plot(time_steps, gt_np[:, idx_px], 'k-', label='GT')
axs[0, 0].plot(time_steps, ekf_np[:, idx_px], 'r--', label='EKF')
axs[0, 0].plot(time_steps, knet_np[:, idx_px], 'b-.', label='KalmanNet')
axs[0, 0].set_title("Pozice X")
axs[0, 0].set_ylabel("Metry")
axs[0, 0].grid(True)
axs[0, 0].legend()

# --- Pozice Y ---
axs[0, 1].plot(time_steps, gt_np[:, idx_py], 'k-', label='GT')
axs[0, 1].plot(time_steps, ekf_np[:, idx_py], 'r--', label='EKF')
axs[0, 1].plot(time_steps, knet_np[:, idx_py], 'b-.', label='KalmanNet')
axs[0, 1].set_title("Pozice Y")
axs[0, 1].set_ylabel("Metry")
axs[0, 1].grid(True)

# --- Rychlost X ---
axs[1, 0].plot(time_steps, gt_np[:, idx_vx], 'k-', label='GT', alpha=0.5)
axs[1, 0].plot(time_steps, ekf_np[:, idx_vx], 'r--', label='EKF')
axs[1, 0].plot(time_steps, knet_np[:, idx_vx], 'b-.', label='KalmanNet')
axs[1, 0].set_title("Rychlost X (Velocity)")
axs[1, 0].set_ylabel("m/s")
axs[1, 0].grid(True)

# --- Rychlost Y ---
axs[1, 1].plot(time_steps, gt_np[:, idx_vy], 'k-', label='GT', alpha=0.5)
axs[1, 1].plot(time_steps, ekf_np[:, idx_vy], 'r--', label='EKF')
axs[1, 1].plot(time_steps, knet_np[:, idx_vy], 'b-.', label='KalmanNet')
axs[1, 1].set_title("Rychlost Y (Velocity)")
axs[1, 1].set_ylabel("m/s")
axs[1, 1].grid(True)

plt.tight_layout()
plt.show()
```


Porovnání odhadu trajektorie (Testovací sekvence č. 1)
Ground Truth (GT)
EKF (Standard)
KalmanNet (Trained)
UKF
Start
Cíl
Pozice Y [m]
Pozice X [m]
100
0
−100
−200
−300
−400
−500
−600
−700
−600
−400
−200
0
200


Detailní průběh stavů v čase
Pozice X
Pozice Y
GT
EKF
KalmanNet
Metry
Rychlost X (Velocity)
Rychlost Y (Velocity)
m/s