Nama: Andi Cleopatra Maryam Jamila

Nim: 1103213071

Analisis Week 12

1. Analisis Python

```
import numpy as np
import matplotlib.pyplot as plt
```

Analisis: Library yang digunakan numpy untuk komputasi numerik dan matplot untuk grafik.

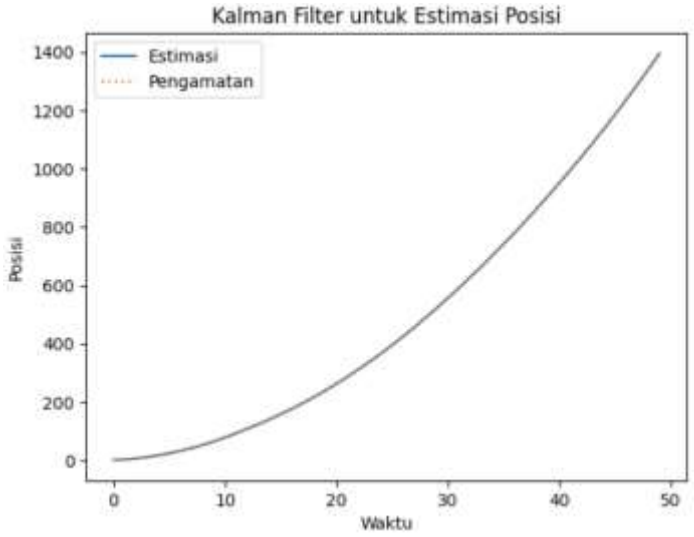
```
# 1. Implementasi Filter Kalman untuk Estimasi Posisi Robot
class KalmanFilter:
    def __init__(self, A, B, H, Q, R, P, x):
        self.A = A  # Matriks Transisi
        self.B = B  # Matriks Kontrol Input
        self.H = H  # Matriks Pengamatan
        self.Q = Q  # Noise Proses
        self.R = R  # Noise Pengamatan
        self.P = P  # Kovarian Error
        self.x = x  # State

    def predict(self, u):
        self.x = self.A @ self.x + self.B @ u
        self.P = self.A @ self.P @ self.A.T + self.Q

    def update(self, z):
        K = self.P @ self.H.T @ np.linalg.inv(self.H @ self.P @ self.H.T + self.R)
        self.x = self.x + K @ (z - self.H @ self.x)
        self.P = (np.eye(len(self.P)) - K @ self.H) @ self.P
```

```
# Parameter Kalman Filter
A = np.array([[1, 1], [0, 1]])
B = np.array([[0.5], [1]])
H = np.array([[1, 0]])
Q = np.array([[0.1, 0], [0, 0.1]])
R = np.array([[1]])
P = np.eye(2)
x = np.array([0, 0])

kf = KalmanFilter(A, B, H, Q, R, P, x)
```

```
# Simulasi
steps = 50
u = np.array([1])  # Input kecepatan
positions = []
measurements = []
for _ in range(steps):
    # Prediksi dan update filter
    kf.predict(u)
    z = np.array([kf.x[0] + np.random.normal(0, 1)])  # Simulasi pengamatan dengan noise
    kf.update(z)

    positions.append(kf.x[0])
    measurements.append(z[0])
```

```
# Visualisasi
plt.plot(range(steps), positions, label="Estimasi")
plt.plot(range(steps), measurements, label="Pengamatan", linestyle='dotted')
plt.legend()
plt.xlabel("Waktu")
plt.ylabel("Posisi")
plt.title("Kalman Filter untuk Estimasi Posisi")
plt.show()
```

Output:



Analisis:

Implementasi **Filter Kalman** digunakan untuk melakukan **estimasi posisi robot** berdasarkan data pengamatan yang terkontaminasi noise. Kode ini dimulai dengan mendefinisikan kelas **KalmanFilter** yang mengimplementasikan dua langkah utama: **prediksi** dan **update**. Model dinamis robot dijelaskan menggunakan matriks transisi (**A**), matriks kontrol input (**B**), dan matriks pengamatan (**H**). Selain itu, terdapat juga matriks noise proses (**Q**) dan noise pengamatan (**R**) untuk mengatur tingkat ketidakpastian dalam model dan pengukuran. Selama simulasi, robot bergerak dengan kecepatan tetap, namun pengamatan posisi dilengkapi dengan noise untuk mensimulasikan kondisi dunia nyata. Filter Kalman memperbarui estimasi posisi robot berdasarkan perbedaan antara pengamatan dan prediksi. Hasil dari simulasi ini divisualisasikan, memperlihatkan perbandingan antara posisi yang diperkirakan oleh Kalman Filter dan pengamatan yang terkontaminasi noise. Pada grafik, posisi yang diperkirakan oleh filter terlihat lebih stabil dibandingkan dengan pengamatan yang bervariasi, menunjukkan kemampuan Filter Kalman dalam mengurangi noise dan memberikan estimasi posisi yang lebih akurat.

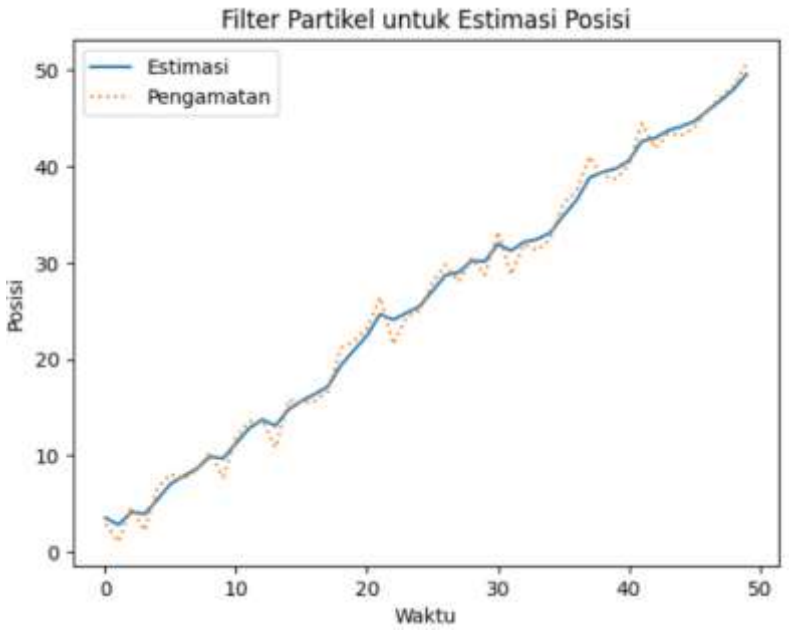
```python
# 2. Implementasi Filter Partikel untuk Estimasi Posisi Robot
class ParticleFilter:
    def __init__(self, num_particles, x_range, measurement_noise):
        self.num_particles = num_particles
        self.particles = np.random.uniform(x_range[0], x_range[1], num_particles)
        self.weights = np.ones(num_particles) / num_particles
        self.measurement_noise = measurement_noise

    def predict(self, control):
        self.particles += control + np.random.normal(0, 1, self.num_particles)

    def update(self, measurement):
        self.weights = np.exp(-((self.particles - measurement)**2) / (2 * self.measurement_noise**2))
        self.weights /= np.sum(self.weights)

    def resample(self):
        indices = np.random.choice(range(self.num_particles), size=self.num_particles, p=self.weights)
        self.particles = self.particles[indices]
        self.weights.fill(1.0 / self.num_particles)

    def estimate(self):
        return np.mean(self.particles)
```

```python
# Parameter Filter Partikel
num_particles = 1000
x_range = [0, 100]
measurement_noise = 2
pf = ParticleFilter(num_particles, x_range, measurement_noise)
```

```python
# Simulasi
positions_pf = []
measurements_pf = []
true_position = 0
for _ in range(steps):
    true_position += 1
    pf.predict(1)  # Kecepatan konstan
    z = true_position + np.random.normal(0, measurement_noise)
    pf.update(z)
    pf.resample()

    positions_pf.append(pf.estimate())
    measurements_pf.append(z)
```

```python
# Visualisasi
plt.plot(range(steps), positions_pf, label="Estimasi")
plt.plot(range(steps), measurements_pf, label="Pengamatan", linestyle='dotted')
plt.legend()
plt.xlabel("Waktu")
plt.ylabel("Posisi")
plt.title("Filter Partikel untuk Estimasi Posisi")
plt.show()
```

Output:



Analisis:

Filter Partikel digunakan untuk estimasi posisi robot yang bergerak dengan kecepatan konstan, dengan pengamatan yang terkontaminasi noise. Filter Partikel bekerja dengan menyebarkan sejumlah partikel di ruang kemungkinan posisi robot, yang masing-masing partikel memiliki bobot yang mencerminkan seberapa cocok prediksi partikel tersebut dengan pengamatan yang diterima. Setiap langkah, partikel diprediksi berdasarkan kontrol (kecepatan) dan diperbarui dengan pengamatan yang terdistorsi noise. Bobot partikel dihitung berdasarkan kesesuaian dengan pengamatan, kemudian dilakukan proses **resampling** untuk memperbarui distribusi partikel agar lebih fokus pada area yang lebih mungkin. Estimasi posisi robot diperoleh dari rata-rata posisi partikel. Hasil simulasi menunjukkan perbandingan antara estimasi posisi yang dihasilkan oleh Filter Partikel dan pengamatan yang bervariasi akibat noise. Grafik menunjukkan bahwa Filter Partikel berhasil memberikan estimasi yang lebih stabil dan akurat meskipun pengamatan memiliki noise, berkat mekanisme pembaruan dan resampling yang diterapkan pada partikel.

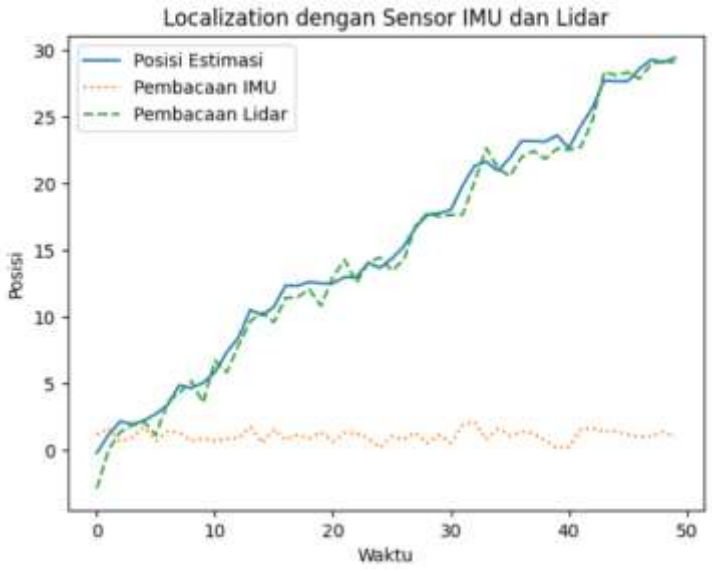
```
# 3. Implementasi Localization dengan Sensor IMU dan Lidar
class Localization:
    def __init__(self, initial_position, imu_noise, lidar_noise):
        self.position = initial_position
        self.imu_noise = imu_noise
        self.lidar_noise = lidar_noise

    def update_with_imu(self, imu_reading):
        self.position += imu_reading + np.random.normal(0, self.imu_noise)

    def update_with_lidar(self, lidar_reading):
        self.position = (self.position + lidar_reading + np.random.normal(0, self.lidar_noise)) / 2
```

```
# Parameter
imu_noise = 0.5
lidar_noise = 1.0
localization = Localization(0, imu_noise, lidar_noise)
```

```python
# Simulasi
positions_loc = []
imu_readings = []
lidar_readings = []
for _ in range(steps):
    imu = 1 + np.random.normal(0, imu_noise)
    lidar = localization.position + np.random.normal(0, lidar_noise)
    localization.update_with_imu(imu)
    localization.update_with_lidar(lidar)

    imu_readings.append(imu)
    lidar_readings.append(lidar)
    positions_loc.append(localization.position)
```

```python
# Visualisasi
plt.plot(range(steps), positions_loc, label="Posisi Estimasi")
plt.plot(range(steps), imu_readings, label="Pembacaan IMU", linestyle='dotted')
plt.plot(range(steps), lidar_readings, label="Pembacaan Lidar", linestyle='dashed')
plt.legend()
plt.xlabel("Waktu")
plt.ylabel("Posisi")
plt.title("Localization dengan Sensor IMU dan Lidar")
plt.show()
```

Output:



Analisis:

Lokalisasi robot dilakukan menggunakan dua sensor, yaitu **IMU (Inertial Measurement Unit)** dan **Lidar**, dengan masing-masing sensor memiliki noise yang berbeda. Sensor IMU memberikan pembacaan yang meliputi informasi kecepatan atau akselerasi yang terdistorsi oleh noise, sementara sensor Lidar memberikan pembacaan posisi yang lebih langsung namun juga terpengaruh oleh noise. Dalam setiap iterasi, posisi robot diperbarui menggunakan pembacaan dari kedua sensor tersebut: pertama dengan IMU, yang mengubah posisi berdasarkan kecepatan terukur, dan kemudian dengan Lidar, yang menghitung posisi baru dengan mengambil rata-rata antara posisi IMU dan pembacaan Lidar. Dengan cara ini, kedua sensor saling melengkapi untuk memberikan estimasi posisi yang lebih akurat. Hasil simulasi menunjukkan bahwa meskipun terdapat noise pada setiap pembacaan sensor, estimasi posisi robot dapat dipertahankan lebih stabil dengan menggabungkan informasi dari IMU dan Lidar. Grafik yang dihasilkan menggambarkan bagaimana posisi estimasi robot mengikuti dinamika

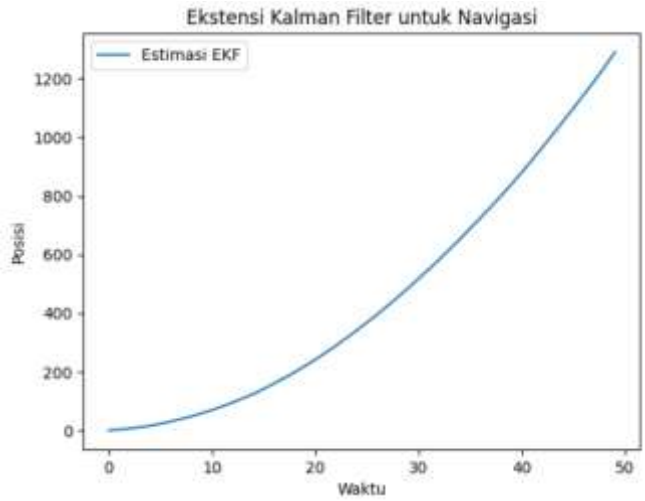
pembacaan IMU dan Lidar sepanjang waktu, di mana posisi estimasi cenderung lebih halus berkat penggabungan informasi dari kedua sensor tersebut.

```
# 4. Implementasi Simulasi Ekstensi Kalman Filter untuk Navigasi
class ExtendedKalmanFilter(KalmanFilter):
    def __init__(self, A, B, H, Q, R, P, x, nonlinear_h):
        super().__init__(A, B, H, Q, R, P, x)
        self.nonlinear_h = nonlinear_h

    def update(self, z):
        H_jacobian = self.H  # Jacobian matrix of H (simplified here)
        K = self.P @ H_jacobian.T @ np.linalg.inv(H_jacobian @ self.P @ H_jacobian.T + self.R)
        self.x = self.x + K @ (z - self.nonlinear_h(self.x))
        self.P = (np.eye(len(self.P)) - K @ H_jacobian) @ self.P
```

```
# Nonlinear measurement function
nonlinear_h = lambda x: np.array([np.sqrt(x[0]**2 + x[1]**2)])
ekf = ExtendedKalmanFilter(A, B, H, Q, R, P, x, nonlinear_h)

# Simulasi untuk EKF
positions_ekf = []
for _ in range(steps):
    ekf.predict(u)
    z = nonlinear_h(ekf.x) + np.random.normal(0, 1)
    ekf.update(z)

    positions_ekf.append(ekf.x[0])
```

```
# Visualisasi
plt.plot(range(steps), positions_ekf, label="Estimasi EKF")
plt.legend()
plt.xlabel("Waktu")
plt.ylabel("Posisi")
plt.title("Ekstensi Kalman Filter untuk Navigasi")
plt.show()
```

Output:



Analisis:

Pada implementasi **Extended Kalman Filter (EKF)** untuk navigasi ini, model filter diperluas untuk menangani sistem non-linear. Sebagai perbedaan dengan Kalman Filter konvensional, yang mengasumsikan model linier, EKF mengatasi masalah non-linearitas dengan menggunakan pendekatan **linearization** melalui Jacobian. Fungsi pengukuran non-linear digunakan untuk memperkirakan jarak antara dua titik dalam sistem dua dimensi, yang dihitung dengan rumus Euclidean (akar dari jumlah kuadrat posisi X dan Y). Pada setiap

iterasi, EKF memprediksi posisi berdasarkan model gerak, dan kemudian memperbarui estimasi posisi dengan membandingkan pengukuran yang terdistorsi oleh noise dengan prediksi sebelumnya. Estimasi posisi yang dihasilkan menunjukkan bahwa meskipun terdapat ketidakpastian dalam pengukuran, EKF mampu memperbaiki dan memprediksi posisi robot dengan akurat, meski menghadapi sistem non-linear. Visualisasi hasil menunjukkan bagaimana estimasi posisi robot mengikuti pola yang lebih halus dan stabil seiring berjalannya waktu, meskipun terdapat fluktuasi akibat noise pada pengukuran. EKF, dengan menggabungkan prediksi dan update berbasis non-linear, berhasil meningkatkan ketepatan navigasi robot dibandingkan dengan filter linier seperti Kalman Filter.

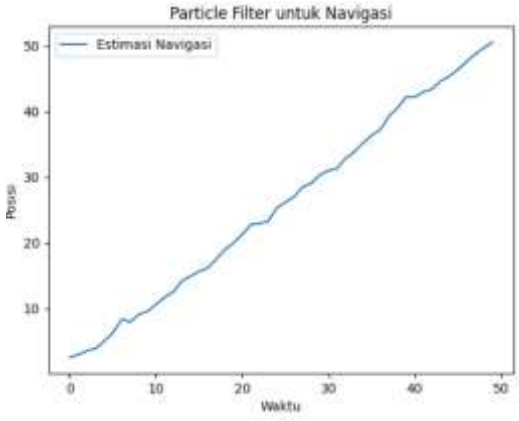
```
# 5. Implementasi Particle Filter untuk Navigasi
class ParticleFilterNavigation(ParticleFilter):
    def __init__(self, num_particles, x_range, measurement_noise, control_noise):
        super().__init__(num_particles, x_range, measurement_noise)
        self.control_noise = control_noise

    def predict(self, control):
        self.particles += control + np.random.normal(0, self.control_noise, self.num_particles)
```

```
# Parameter Navigasi
control_noise = 0.5
pf_navigation = ParticleFilterNavigation(num_particles, x_range, measurement_noise, control_noise)
```

```
# Simulasi Navigasi
positions_nav = []
true_position_nav = 0
for _ in range(steps):
    true_position_nav += 1
    pf_navigation.predict(1)
    z_nav = true_position_nav + np.random.normal(0, measurement_noise)
    pf_navigation.update(z_nav)
    pf_navigation.resample()

    positions_nav.append(pf_navigation.estimate())
```

```
# Visualisasi
plt.plot(range(steps), positions_nav, label="Estimasi Navigasi")
plt.legend()
plt.xlabel("Waktu")
plt.ylabel("Posisi")
plt.title("Particle Filter untuk Navigasi")
plt.show()
```

Output:



Analisis:
Pada implementasi **Particle Filter untuk Navigasi** ini, filter partikel digunakan untuk mengestimasi posisi robot dalam lingkungan yang penuh dengan ketidakpastian, baik dalam pengukuran maupun dalam kontrol. Berbeda dengan metode berbasis Gaussian seperti Kalman Filter, Particle Filter menggunakan sekumpulan partikel untuk mewakili kemungkinan posisi robot. Setiap partikel dipengaruhi oleh kontrol dan noise, kemudian dipertimbangkan dalam pembaruan berbasis pengukuran yang diperoleh dari sensor. Pada simulasi ini, filter partikel memperkirakan posisi robot yang bergerak dengan kecepatan konstan sambil mengatasi noise dalam pengukuran dan kontrol. Hasil estimasi menunjukkan bahwa meskipun terdapat fluktuasi akibat noise, filter partikel mampu menghasilkan estimasi posisi yang akurat dan lebih stabil dari waktu ke waktu, berkat proses prediksi, pembaruan, dan resampling. Visualisasi menunjukkan bagaimana filter partikel secara dinamis menyesuaikan distribusi partikel untuk memperbaiki estimasi posisi seiring dengan adanya pembaruan dari pengukuran sensor yang terkontaminasi noise, memperlihatkan keunggulan dalam menangani masalah ketidakpastian dalam navigasi.

2. Analisis Webots
   Analisis:
   Simulasi di Webots dengan implementasi Kalman Filter untuk lokalitas robot berfokus pada peningkatan akurasi estimasi posisi robot dalam lingkungan yang dinamis. Kalman Filter digunakan untuk memadukan data dari berbagai sensor, seperti odometri (dari encoder roda), LIDAR, dan gyroscope, dengan model sistem yang menggambarkan dinamika robot. Proses ini memungkinkan pengurangan noise pada pengukuran dan koreksi kesalahan akumulatif dari odometri. Dalam simulasi, robot dijalankan dalam lingkungan yang mencakup hambatan dan landmark yang terdeteksi oleh sensor. Estimasi posisi diperbarui secara iteratif menggunakan prediksi berbasis model gerak robot dan pembaruan berdasarkan pembacaan sensor. Analisis menunjukkan bahwa penggunaan Kalman Filter secara signifikan meningkatkan keakuratan lokalitas dibandingkan hanya mengandalkan satu sumber data, menjadikannya alat penting untuk navigasi otonom yang andal di berbagai aplikasi robotika.