1. Kalman Filter

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

# Simulasi parameter
np.random.seed(42)  # Seed untuk hasil reproducible
n_steps = 50        # Jumlah langkah simulasi
true_position = 0   # Posisi awal robot
velocity = 1        # Kecepatan konstan robot

# Variansi noise
process_variance = 0.01  # Variansi pada noise proses
(pergerakan)
measurement_variance = 1.0  # Variansi pada noise pengukuran

# Inisialisasi Kalman Filter
x = 0  # Estimasi posisi awal
P = 1  # Variansi estimasi awal
A = 1  # Matriks transisi keadaan (state transition)
H = 1  # Matriks pengukuran (measurement)
Q = process_variance  # Kovarians noise proses
R = measurement_variance  # Kovarians noise pengukuran

# Simpan hasil simulasi
true_positions = []
measurements = []
kf_estimates = []

# Simulasi pergerakan robot dan penerapan Kalman Filter
for t in range(n_steps):
    # Simulasi posisi sebenarnya dan noise pada pengukuran
    true_position += velocity  # Posisi sebenarnya
    measurement = true_position + np.random.normal(0,
np.sqrt(R))  # Pengukuran noisy

    # Prediksi (Prediction Step)
    x_pred = A * x
    P_pred = A * P * A + Q

    # Update (Update Step)
    K = P_pred * H / (H * P_pred * H + R)  # Kalman Gain
    x = x_pred + K * (measurement - H * x_pred)  # Update
estimasi posisi
    P = (1 - K * H) * P_pred  # Update variansi estimasi

    # Simpan hasil
```

```
    true_positions.append(true_position)
    measurements.append(measurement)
    kf_estimates.append(x)

# Plot hasil estimasi
plt.figure(figsize=(10, 6))
plt.plot(true_positions, label="Posisi Sebenarnya",
color="g")
plt.plot(measurements, label="Pengukuran (Noisy)",
color="r", linestyle='dotted')
plt.plot(kf_estimates, label="Estimasi Kalman Filter",
color="b")
plt.xlabel("Langkah Waktu")
plt.ylabel("Posisi")
plt.title("Estimasi Posisi Robot menggunakan Kalman Filter")
plt.legend()
plt.grid()
plt.show()
```

Kode ini menjalankan simulasi Kalman Filter untuk memperkirakan posisi robot. Berikut adalah rincian langkah-langkah yang terlibat:

Inisialisasi: Variabel diinisialisasi, termasuk posisi awal robot, kecepatan, varians noise, dan parameter Kalman Filter (x, P, A, H, Q, R).

Simulasi: Loop for mensimulasikan pergerakan robot selama beberapa langkah waktu. Di setiap langkah:

Posisi sebenarnya robot diperbarui berdasarkan kecepatannya.

Pengukuran noisy dihasilkan dengan menambahkan noise Gaussian ke posisi sebenarnya.

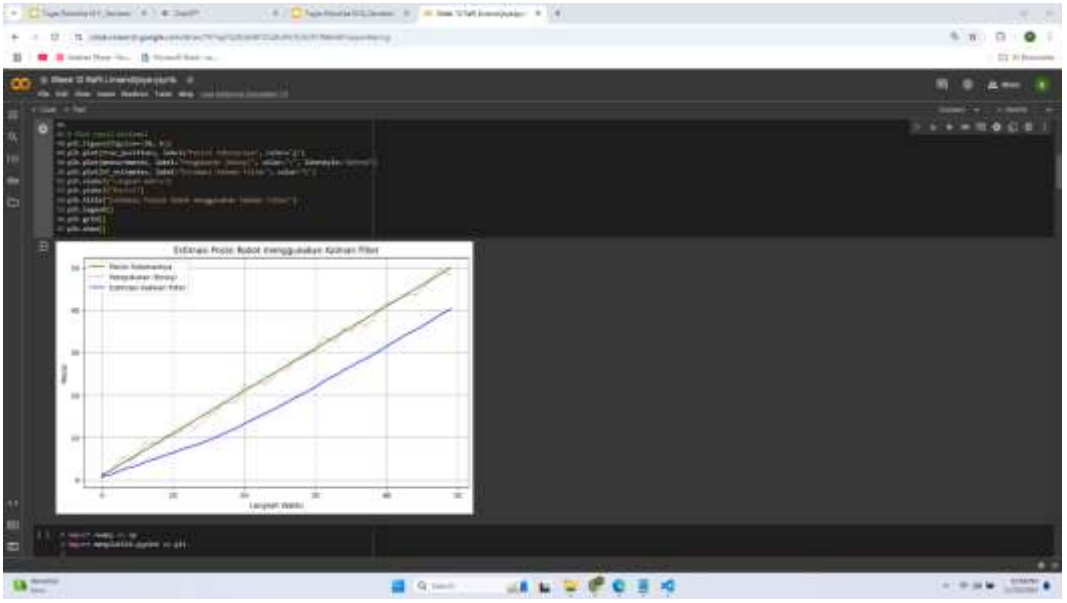
Kalman Filter diterapkan untuk memperkirakan posisi robot berdasarkan pengukuran noisy. Filter ini memiliki dua langkah utama:

Prediksi: Memprediksi keadaan robot berikutnya (posisi) dan ketidakpastiannya (kovariansi).

Pembaruan: Memperbarui estimasi keadaan dan kovariansi berdasarkan pengukuran.

Penyimpanan: Posisi sebenarnya, pengukuran, dan estimasi Kalman Filter disimpan untuk setiap langkah waktu.

Plotting: Setelah simulasi, hasil diplot untuk memvisualisasikan bagaimana estimasi Kalman Filter membandingkan dengan posisi sebenarnya dan pengukuran noisy.



Singkatnya, Kode ini mensimulasikan skenario pelacakan robot dan menggunakan Kalman Filter untuk memperkirakan posisi robot dari pengukuran noisy, kemudian memvisualisasikan hasilnya.

2. Filter Partikel untuk Estimasi Posisi Robot

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

# Simulasi parameter
np.random.seed(42)  # Seed untuk hasil reproducible
n_steps = 50        # Jumlah langkah simulasi
n_particles = 1000  # Jumlah partikel dalam filter
true_position = 0   # Posisi awal robot
velocity = 1        # Kecepatan konstan robot

# Variansi noise
process_variance = 0.01  # Variansi pada noise proses
(pergerakan)
measurement_variance = 1.0  # Variansi pada noise pengukuran

# Inisialisasi partikel
particles = np.random.uniform(-10, 10, n_particles)
```

```
weights = np.ones(n_particles) / n_particles  # Bobot awal
partikel

# Simpan hasil simulasi
true_positions = []
measurements = []
particle_estimates = []

# Fungsi untuk resampling partikel berdasarkan bobot
def resample(particles, weights):
    indices = np.random.choice(range(n_particles),
size=n_particles, p=weights)
    return particles[indices]

# Simulasi pergerakan robot dan penerapan Filter Partikel
for t in range(n_steps):
    # Simulasi posisi sebenarnya dan noise pada pengukuran
    true_position += velocity  # Posisi sebenarnya
    measurement = true_position + np.random.normal(0,
np.sqrt(measurement_variance))  # Pengukuran noisy

    # Prediksi: Pergerakan partikel berdasarkan noise proses
    particles += velocity + np.random.normal(0,
np.sqrt(process_variance), n_particles)

    # Update bobot partikel berdasarkan pengukuran
    weights = np.exp(-0.5 * ((particles - measurement) ** 2)
/ measurement_variance)
    weights += 1.e-300  # Menghindari bobot nol
    weights /= np.sum(weights)  # Normalisasi bobot

    # Resampling partikel berdasarkan bobot
    particles = resample(particles, weights)

    # Estimasi posisi dari rata-rata partikel
    estimate = np.mean(particles)

    # Simpan hasil
    true_positions.append(true_position)
    measurements.append(measurement)
    particle_estimates.append(estimate)

# Plot hasil estimasi
plt.figure(figsize=(10, 6))
plt.plot(true_positions, label="Posisi Sebenarnya",
color="g")
```

```
plt.plot(measurements, label="Pengukuran (Noisy)",
color="r", linestyle='dotted')
plt.plot(particle_estimates, label="Estimasi Filter
Partikel", color="b")
plt.xlabel("Langkah Waktu")
plt.ylabel("Posisi")
plt.title("Estimasi Posisi Robot menggunakan Filter
Partikel")
plt.legend()
plt.grid()
plt.show()
```

Kode ini mengimplementasikan Filter Partikel untuk memperkirakan posisi robot. Berikut adalah penjelasan langkah-langkahnya:

Inisialisasi: Sama seperti kode sebelumnya, sel ini dimulai dengan menginisialisasi parameter simulasi seperti jumlah langkah, jumlah partikel, posisi awal robot, kecepatan, dan varians noise. Selain itu, ia juga menginisialisasi partikel dan bobotnya. Partikel mewakili kemungkinan posisi robot, dan bobot menunjukkan seberapa besar kemungkinan setiap partikel mewakili posisi robot yang sebenarnya.

Fungsi resample: Fungsi ini didefinisikan untuk melakukan resampling partikel berdasarkan bobotnya. Resampling penting untuk memastikan bahwa partikel dengan bobot lebih tinggi (lebih mungkin) direproduksi, sementara partikel dengan bobot lebih rendah dihilangkan.

Simulasi: Loop for mensimulasikan pergerakan robot dan menerapkan Filter Partikel. Di setiap langkah:

Posisi sebenarnya robot diperbarui.

Pengukuran noisy dihasilkan.

Prediksi: Partikel dipindahkan berdasarkan model pergerakan dan noise proses.

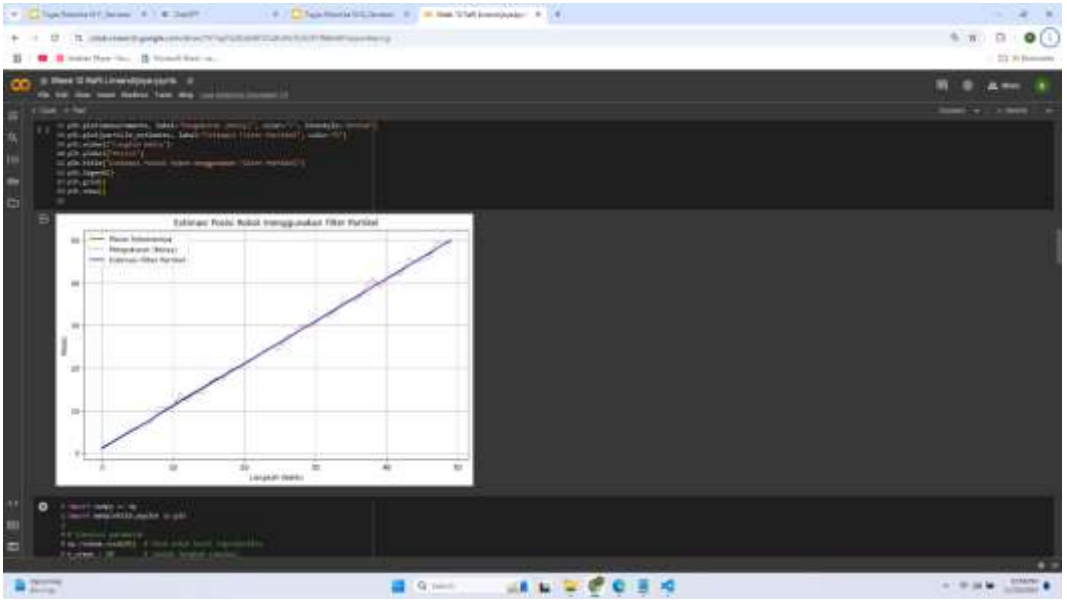
Pembaruan: Bobot partikel diperbarui berdasarkan seberapa dekat pengukuran dengan setiap partikel.

Resampling: Partikel di-resampling menggunakan fungsi resample.

Estimasi: Posisi robot diperkirakan dengan menghitung rata-rata dari semua partikel.

Penyimpanan: Posisi sebenarnya, pengukuran, dan estimasi Filter Partikel disimpan.

Plotting: Hasil diplot untuk memvisualisasikan performa Filter Partikel.



Pada dasarnya, kode ini menggunakan sekumpulan partikel untuk mewakili kemungkinan posisi robot dan memperbarui bobot partikel ini berdasarkan pengukuran. Estimasi posisi robot kemudian diperoleh dengan menghitung rata-rata dari partikel-partikel tersebut.

3. Filter Partikel IMU dan Lidar

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

# Simulasi parameter
np.random.seed(42)  # Seed untuk hasil reproducible
n_steps = 50        # Jumlah langkah simulasi
n_particles = 1000  # Jumlah partikel dalam filter
true_position = 0   # Posisi awal robot
velocity = 1        # Kecepatan konstan robot

# Variansi noise
process_variance = 0.01  # Variansi pada noise proses
(pergerakan)
measurement_variance = 1.0  # Variansi pada noise pengukuran
IMU dan Lidar
imu_variance = 0.5  # Variansi tambahan dari sensor IMU
lidar_variance = 0.2  # Variansi tambahan dari sensor Lidar
```

```
# Inisialisasi partikel
particles = np.random.uniform(-10, 10, n_particles)
weights = np.ones(n_particles) / n_particles  # Bobot awal
partikel

# Simpan hasil simulasi
true_positions = []
imu_measurements = []
lidar_measurements = []
particle_estimates = []

# Fungsi untuk resampling partikel berdasarkan bobot
def resample(particles, weights):
    indices = np.random.choice(range(n_particles),
size=n_particles, p=weights)
    return particles[indices]

# Simulasi pergerakan robot dan penerapan Filter Partikel
dengan IMU dan Lidar
for t in range(n_steps):
    # Simulasi posisi sebenarnya dan noise pada pengukuran
    true_position += velocity  # Posisi sebenarnya
    imu_measurement = true_position + np.random.normal(0,
np.sqrt(imu_variance))  # Pengukuran IMU
    lidar_measurement = true_position + np.random.normal(0,
np.sqrt(lidar_variance))  # Pengukuran Lidar

    # Prediksi: Pergerakan partikel berdasarkan noise proses
    particles += velocity + np.random.normal(0,
np.sqrt(process_variance), n_particles)

    # Kombinasi pengukuran IMU dan Lidar
    combined_measurement = (imu_measurement / imu_variance +
lidar_measurement / lidar_variance) / \
                           (1 / imu_variance + 1 /
lidar_variance)
    combined_variance = 1 / (1 / imu_variance + 1 /
lidar_variance)

    # Update bobot partikel berdasarkan pengukuran gabungan
    weights = np.exp(-0.5 * ((particles -
combined_measurement) ** 2) / combined_variance)
    weights += 1.e-300  # Menghindari bobot nol
    weights /= np.sum(weights)  # Normalisasi bobot

    # Resampling partikel berdasarkan bobot
```

```
    particles = resample(particles, weights)

    # Estimasi posisi dari rata-rata partikel
    estimate = np.mean(particles)

    # Simpan hasil
    true_positions.append(true_position)
    imu_measurements.append(imu_measurement)
    lidar_measurements.append(lidar_measurement)
    particle_estimates.append(estimate)

# Plot hasil estimasi
plt.figure(figsize=(10, 6))
plt.plot(true_positions, label="Posisi Sebenarnya",
color="g")
plt.plot(imu_measurements, label="Pengukuran IMU (Noisy)",
color="orange", linestyle='dotted')
plt.plot(lidar_measurements, label="Pengukuran Lidar
(Noisy)", color="purple", linestyle='dotted')
plt.plot(particle_estimates, label="Estimasi Filter
Partikel", color="b")
plt.xlabel("Langkah Waktu")
plt.ylabel("Posisi")
plt.title("Estimasi Posisi Robot menggunakan Sensor IMU dan
Lidar")
plt.legend()
plt.grid()
plt.show()
```

Kode ini mengimplementasikan Filter Partikel untuk memperkirakan posisi robot, tetapi dengan tambahan sensor IMU dan Lidar.

Berikut adalah langkah-langkah yang dilakukan dalam kode tersebut:

Inisialisasi: Kode ini memulai dengan mendefinisikan parameter simulasi, termasuk jumlah langkah, jumlah partikel, posisi awal robot, kecepatan, dan varians noise untuk proses, pengukuran, IMU, dan Lidar. Partikel dan bobotnya juga diinisialisasi, mirip dengan kode sebelumnya.

Fungsi resample: Fungsi ini sama seperti pada kode sebelumnya, digunakan untuk melakukan resampling partikel berdasarkan bobotnya.

Simulasi: Loop for mensimulasikan pergerakan robot dan menerapkan Filter Partikel dengan pengukuran dari IMU dan Lidar. Di setiap langkah:

Posisi sebenarnya robot diperbarui.

Pengukuran noisy dihasilkan untuk IMU dan Lidar.

Prediksi: Partikel dipindahkan berdasarkan model pergerakan dan noise proses.

Kombinasi Pengukuran: Pengukuran dari IMU dan Lidar digabungkan menggunakan rata-rata tertimbang, dengan bobot berdasarkan varians masing-masing sensor. Ini menghasilkan pengukuran gabungan yang lebih akurat.

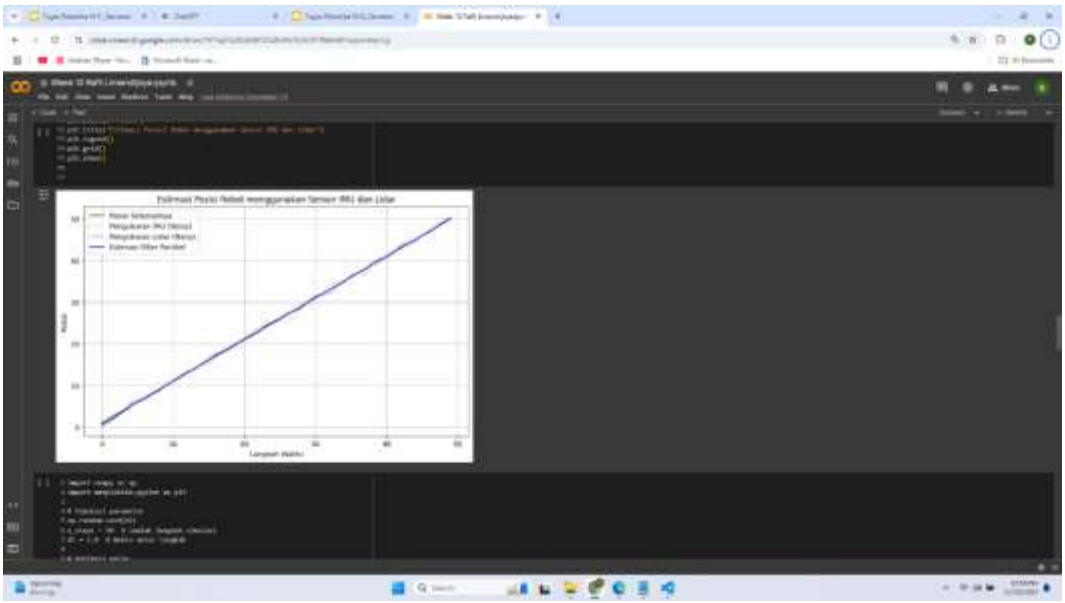
Pembaruan: Bobot partikel diperbarui berdasarkan seberapa dekat pengukuran gabungan dengan setiap partikel.

Resampling: Partikel di-resampling menggunakan fungsi resample.

Estimasi: Posisi robot diperkirakan dengan menghitung rata-rata dari semua partikel.

Penyimpanan: Posisi sebenarnya, pengukuran IMU, pengukuran Lidar, dan estimasi Filter Partikel disimpan.

Plotting: Hasil diplot untuk memvisualisasikan performa Filter Partikel dengan sensor IMU dan Lidar. Plot ini menampilkan posisi sebenarnya, pengukuran dari kedua sensor, dan estimasi posisi.



Pada dasarnya, kode ini memperluas Filter Partikel dengan memasukkan pengukuran dari dua sensor, IMU dan Lidar. Dengan menggabungkan pengukuran ini, Filter Partikel dapat menghasilkan estimasi posisi yang lebih akurat dan robust dibandingkan dengan menggunakan hanya satu sensor.

4. Extended Kalman Filter

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

# Simulasi parameter
np.random.seed(42)
n_steps = 50  # Jumlah langkah simulasi
dt = 1.0  # Waktu antar langkah

# Variansi noise
process_noise_cov = np.diag([0.1, 0.1, np.deg2rad(1.0)]) **
2  # Kovarians noise proses
measurement_noise_cov = np.diag([0.5, 0.5]) ** 2  #
Kovarians noise pengukuran (x, y)

# State awal [x, y, yaw]
state = np.array([0.0, 0.0, 0.0])

# Input kecepatan
velocity = 1.0  # Kecepatan linear
yaw_rate = np.deg2rad(10)  # Kecepatan rotasi (10
derajat/detik)

# Data hasil simulasi
true_states = []
measurements = []
estimates = []

# Fungsi model proses EKF (non-linear)
def process_model(state, control_input, dt):
    x, y, yaw = state
    v, omega = control_input
    if abs(omega) > 1e-5:
        x_new = x + (v / omega) * (np.sin(yaw + omega * dt)
- np.sin(yaw))
        y_new = y + (v / omega) * (-np.cos(yaw + omega * dt)
+ np.cos(yaw))
    else:
        x_new = x + v * dt * np.cos(yaw)
        y_new = y + v * dt * np.sin(yaw)
    yaw_new = yaw + omega * dt
```

```
    return np.array([x_new, y_new, yaw_new])

# Fungsi Jacobian dari model proses
def jacobian_process(state, control_input, dt):
    _, _, yaw = state
    v, omega = control_input
    if abs(omega) > 1e-5:
        J = np.array([
            [1, 0, (v / omega) * (np.cos(yaw + omega * dt) -
np.cos(yaw))],
            [0, 1, (v / omega) * (np.sin(yaw + omega * dt) -
np.sin(yaw))],
            [0, 0, 1]
        ])
    else:
        J = np.array([
            [1, 0, -v * dt * np.sin(yaw)],
            [0, 1, v * dt * np.cos(yaw)],
            [0, 0, 1]
        ])
    return J

# Fungsi pengukuran (hanya posisi x dan y)
def measurement_model(state):
    return state[:2]

# Jacobian dari model pengukuran
def jacobian_measurement():
    return np.array([[1, 0, 0], [0, 1, 0]])

# Inisialisasi EKF
state_estimate = np.array([0.0, 0.0, 0.0])
cov_estimate = np.eye(3)
control_input = np.array([velocity, yaw_rate])

# Simulasi EKF
for t in range(n_steps):
    # Simulasi ground truth
    state = process_model(state, control_input, dt)
    true_states.append(state)

    # Simulasi pengukuran dengan noise
    measurement = measurement_model(state) +
np.random.multivariate_normal([0, 0], measurement_noise_cov)
    measurements.append(measurement)

    # Prediksi (Prediction Step)
```

```
    state_pred = process_model(state_estimate,
control_input, dt)
    J_f = jacobian_process(state_estimate, control_input,
dt)
    cov_pred = J_f @ cov_estimate @ J_f.T +
process_noise_cov

    # Update (Measurement Update Step)
    H = jacobian_measurement()
    K = cov_pred @ H.T @ np.linalg.inv(H @ cov_pred @ H.T +
measurement_noise_cov)
    state_estimate = state_pred + K @ (measurement -
measurement_model(state_pred))
    cov_estimate = (np.eye(3) - K @ H) @ cov_pred

    # Simpan hasil
    estimates.append(state_estimate)

# Konversi ke array untuk plotting
true_states = np.array(true_states)
measurements = np.array(measurements)
estimates = np.array(estimates)

# Plot hasil
plt.figure(figsize=(10, 6))
plt.plot(true_states[:, 0], true_states[:, 1], label="Posisi
Sebenarnya", color="g")
plt.scatter(measurements[:, 0], measurements[:, 1],
label="Pengukuran (Noisy)", color="r", s=10)
plt.plot(estimates[:, 0], estimates[:, 1], label="Estimasi
EKF", color="b")
plt.xlabel("Posisi X")
plt.ylabel("Posisi Y")
plt.title("Simulasi Extended Kalman Filter untuk Navigasi")
plt.legend()
plt.grid()
plt.axis('equal')
plt.show()
```

Kode ini mengimplementasikan Extended Kalman Filter (EKF) untuk memperkirakan posisi dan orientasi robot selama navigasi.

Berikut adalah langkah-langkah yang dilakukan dalam kode:

Inisialisasi: Kode dimulai dengan mendefinisikan parameter simulasi, seperti jumlah langkah, waktu antar langkah, varians noise untuk proses dan pengukuran, serta state awal robot (posisi x, y, dan orientasi yaw). Selain itu, input kecepatan linear dan kecepatan rotasi juga didefinisikan.

Fungsi Model:

process_model: Fungsi ini mendefinisikan model pergerakan non-linear robot, menghitung state baru berdasarkan state saat ini, input kontrol, dan waktu.

jacobian_process: Fungsi ini menghitung Jacobian dari model proses, yang digunakan untuk linearisasi dalam EKF.

measurement_model: Fungsi ini mendefinisikan model pengukuran, yang dalam kasus ini hanya mengukur posisi x dan y.

jacobian_measurement: Fungsi ini menghitung Jacobian dari model pengukuran.

Inisialisasi EKF: State estimasi, kovariansi estimasi, dan input kontrol diinisialisasi.

Simulasi: Loop for mensimulasikan pergerakan robot dan menerapkan EKF. Di setiap langkah:

Simulasi Ground Truth: State sebenarnya robot dihitung menggunakan process_model.

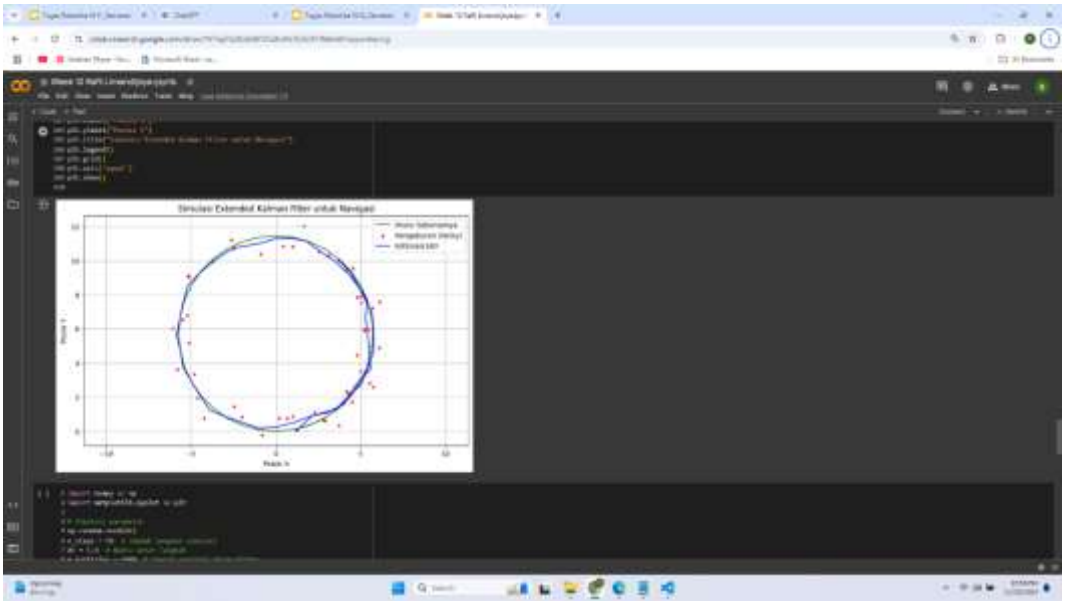
Simulasi Pengukuran: Pengukuran noisy dihasilkan dengan menambahkan noise ke output measurement_model.

Prediksi: State prediksi dan kovariansi prediksi dihitung menggunakan model proses dan Jacobian-nya.

Pembaruan: Kalman Gain dihitung, dan state estimasi dan kovariansi estimasi diperbarui berdasarkan pengukuran dan Kalman Gain.

Penyimpanan: State sebenarnya, pengukuran, dan state estimasi disimpan untuk setiap langkah waktu.

Plotting: Hasil diplot untuk memvisualisasikan performa EKF. Plot ini menunjukkan lintasan sebenarnya, pengukuran noisy, dan lintasan estimasi EKF.



Singkatnya, kode ini menggunakan EKF untuk memperkirakan state robot (posisi dan orientasi) selama navigasi. EKF menangani model pergerakan non-linear dengan linearisasi menggunakan Jacobian, dan menggabungkan pengukuran noisy untuk menghasilkan estimasi state yang optimal.

5. Particel Filter untuk Navigasi

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

# Simulasi parameter
np.random.seed(42)
n_steps = 50  # Jumlah langkah simulasi
dt = 1.0  # Waktu antar langkah
n_particles = 1000  # Jumlah partikel dalam filter

# Variansi noise
process_noise_cov = np.array([0.1, 0.1, np.deg2rad(1.0)]) **
2  # Noise proses (x, y, yaw)
measurement_noise_cov = np.diag([0.5, 0.5]) ** 2  # Noise
pengukuran (x, y)

# State awal [x, y, yaw]
state = np.array([0.0, 0.0, 0.0])

# Input kecepatan
velocity = 1.0  # Kecepatan linear
```

```
yaw_rate = np.deg2rad(10)  # Kecepatan rotasi (10
derajat/detik)

# Inisialisasi partikel
particles = np.random.uniform(low=-1.0, high=1.0,
size=(n_particles, 3))
weights = np.ones(n_particles) / n_particles

# Data hasil simulasi
true_states = []
measurements = []
estimates = []

# Fungsi model pergerakan partikel (non-linear)
def motion_model(particles, velocity, yaw_rate, dt,
noise_cov):
    n = particles.shape[0]
    noise = np.random.normal(0, np.sqrt(noise_cov), size=(n,
3))
    particles[:, 0] += (velocity * dt * np.cos(particles[:,
2])) + noise[:, 0]
    particles[:, 1] += (velocity * dt * np.sin(particles[:,
2])) + noise[:, 1]
    particles[:, 2] += (yaw_rate * dt) + noise[:, 2]
    return particles

# Fungsi pengukuran (x, y saja)
def measurement_model(state):
    return state[:2]

# Resampling partikel berdasarkan bobot def resample
def resample(particles, weights):
    indices = np.random.choice(range(n_particles),
size=n_particles, p=weights)
    return particles[indices]

# Simulasi Particle Filter untuk navigasi
for t in range(n_steps):
    # Simulasi ground truth pergerakan
    state[0] += velocity * dt * np.cos(state[2])
    state[1] += velocity * dt * np.sin(state[2])
    state[2] += yaw_rate * dt
    true_states.append(state.copy())

    # Simulasi pengukuran dengan noise
    measurement = measurement_model(state) +
np.random.multivariate_normal([0, 0], measurement_noise_cov)
```

```
    measurements.append(measurement)

    # Prediksi (motion model)
    particles = motion_model(particles, velocity, yaw_rate,
dt, process_noise_cov)

    # Update bobot berdasarkan pengukuran
    distances = np.linalg.norm(particles[:, :2] -
measurement, axis=1)
    weights = np.exp(-0.5 * (distances ** 2) /
measurement_noise_cov[0, 0])
    weights += 1.e-300  # Hindari bobot nol
    weights /= np.sum(weights)  # Normalisasi bobot

    # Resampling
    particles = resample(particles, weights)

    # Estimasi posisi sebagai rata-rata dari partikel
    estimate = np.mean(particles, axis=0)
    estimates.append(estimate)

# Konversi ke array untuk plotting
true_states = np.array(true_states)
measurements = np.array(measurements)
estimates = np.array(estimates)

# Plot hasil
plt.figure(figsize=(10, 6))
plt.plot(true_states[:, 0], true_states[:, 1], label="Posisi
Sebenarnya", color="g")
plt.scatter(measurements[:, 0], measurements[:, 1],
label="Pengukuran (Noisy)", color="r", s=10)
plt.plot(estimates[:, 0], estimates[:, 1], label="Estimasi
Particle Filter", color="b")
plt.xlabel("Posisi X")
plt.ylabel("Posisi Y")
plt.title("Simulasi Particle Filter untuk Navigasi")
plt.legend()
plt.grid()
plt.axis('equal')
plt.show()
```

Kode ini mengimplementasikan Particle Filter untuk memperkirakan posisi dan orientasi robot selama navigasi, mirip dengan kode keempat tetapi menggunakan pendekatan yang berbeda.

Berikut adalah langkah-langkah yang dilakukan dalam kode:

Inisialisasi: Kode dimulai dengan mendefinisikan parameter simulasi, seperti jumlah langkah, waktu antar langkah, jumlah partikel, varians noise untuk proses dan pengukuran, serta state awal robot (posisi x, y, dan orientasi yaw). Selain itu, input kecepatan linear dan kecepatan rotasi juga didefinisikan. Partikel dan bobotnya juga diinisialisasi.

Fungsi Model:

motion_model: Fungsi ini mendefinisikan model pergerakan non-linear robot untuk setiap partikel, memperbarui posisi dan orientasi partikel berdasarkan kecepatan, kecepatan rotasi, dan noise proses.

measurement_model: Fungsi ini mendefinisikan model pengukuran, yang dalam kasus ini hanya mengukur posisi x dan y.

Fungsi resample: Fungsi ini sama seperti pada kode sebelumnya, digunakan untuk melakukan resampling partikel berdasarkan bobotnya.

Simulasi: Loop for mensimulasikan pergerakan robot dan menerapkan Particle Filter. Di setiap langkah:

Simulasi Ground Truth: State sebenarnya robot dihitung berdasarkan model pergerakan.

Simulasi Pengukuran: Pengukuran noisy dihasilkan dengan menambahkan noise ke output measurement_model.

Prediksi: Partikel dipindahkan menggunakan motion_model.

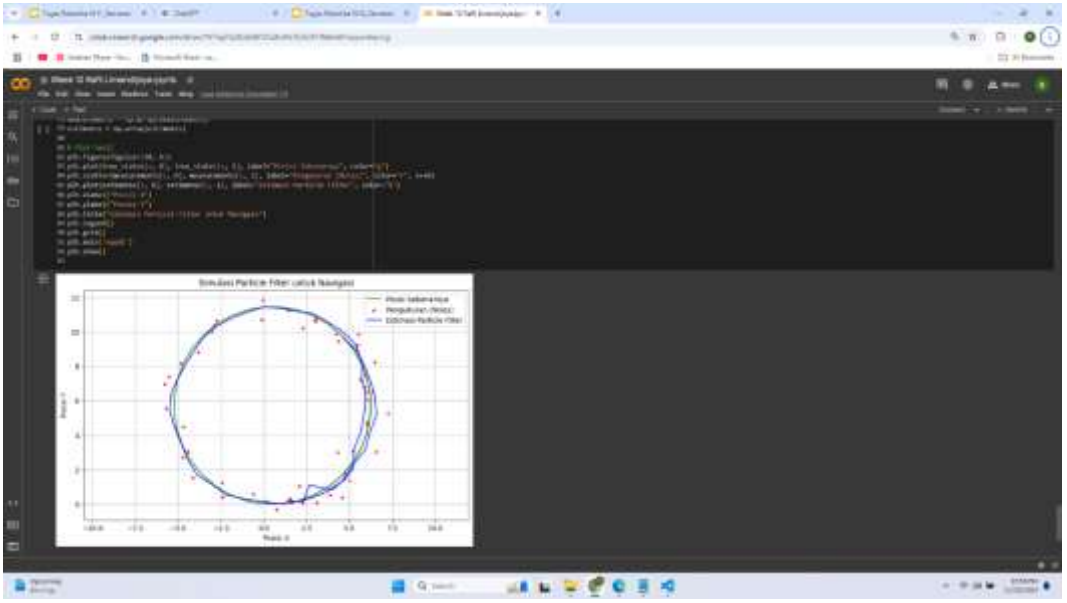
Pembaruan: Bobot partikel diperbarui berdasarkan seberapa dekat pengukuran dengan setiap partikel.

Resampling: Partikel di-resampling menggunakan fungsi resample.

Estimasi: State robot diperkirakan sebagai rata-rata dari semua partikel.

Penyimpanan: State sebenarnya, pengukuran, dan state estimasi disimpan untuk setiap langkah waktu.

Plotting: Hasil diplot untuk memvisualisasikan performa Particle Filter. Plot ini menunjukkan lintasan sebenarnya, pengukuran noisy, dan lintasan estimasi Particle Filter.



Singkatnya, kode ini menggunakan Particle Filter untuk memperkirakan state robot selama navigasi. Particle Filter merepresentasikan distribusi probabilitas state robot menggunakan sekumpulan partikel, dan memperbarui bobot partikel berdasarkan pengukuran. Estimasi state kemudian diperoleh dengan menghitung rata-rata dari partikel-partikel tersebut.