## CSCI Overview

This section presents the design decisions for the Computer Software Configuration Item (CSCI) in the context of multi-target tracking. The design decisions cover inputs, outputs, behavioural responses, safety, security, and privacy requirements, with detailed subparagraphs where necessary. These decisions ensure that the software meets its functional requirements effectively and efficiently.

### 1. Design Decisions Regarding Inputs and Outputs

\*\*Inputs:\*\*

- \*\*Radar Plots:\*\* The CSCI will accept radar plots as input. Each plot includes range, azimuth, elevation, and timestamp data.

- \*\*Configuration Data:\*\* The CSCI will receive configuration data from the Human-Machine Interface (HMI), specifying parameters for the data association process.

\*\*Outputs:\*\*

- \*\*Clustered Plots:\*\* The CSCI will output clusters of radar plots organized with respect to existing tracks.

- \*\*Hypotheses:\*\* The CSCI will generate and output possible hypotheses for target associations within each cluster.

- \*\*Probabilities:\*\* The CSCI will compute and output joint and marginal probabilities for each hypothesis.

- \*\*Best Hypothesis:\*\* The CSCI will output the best hypothesis for target plot association.

- \*\*Visualization Plots:\*\* The CSCI will generate visual plots for various data association techniques for analysis.

### 2. Design Decisions on CSCI Behaviour

\*\*Clustering of Plots:\*\*

- The CSCI will group radar plots into clusters based on spatial and temporal proximity.

- It will handle clusters with a time difference threshold of 0.050 seconds to ensure real-time processing.

\*\*Hypothesis Generation and Evaluation:\*\*

- For each cluster, the CSCI will generate possible hypotheses for target plot associations.

- It will compute joint probabilities for each hypothesis based on the Kalman filter's state and measurement models.

- Marginal probabilities will be calculated for each hypothesis to determine the most likely associations.

\*\*Handling Inputs and Conditions:\*\*

- The CSCI will process radar plots continuously, ensuring timely updates to target associations.

- It will handle unallowed inputs by discarding invalid radar plots and logging errors for further analysis.

\*\*Response Time and Performance:\*\*

- The CSCI is designed to perform clustering and hypothesis evaluation in real-time, with minimal latency.

- Performance metrics such as processing time and accuracy of target associations will be monitored.

### 3. Safety, Security, and Privacy Requirements

\*\*Safety:\*\*

- The CSCI will ensure that the radar plot data is processed accurately to prevent false target associations that could compromise safety.

- It will include error-checking mechanisms to handle anomalous data inputs safely.

\*\*Security:\*\*

- The CSCI will secure the radar plot data and configuration inputs through encryption and access control mechanisms.

- It will implement authentication procedures to verify the integrity of data inputs and outputs.

\*\*Privacy:\*\*

- The CSCI will ensure that sensitive data, such as radar plot information, is handled in compliance with privacy regulations.

- It will anonymize or redact data where necessary to protect privacy.

### Design Conventions

\*\*System States and Modes:\*\*

- The CSCI operates in a real-time mode, continuously receiving and processing radar plots.

- It transitions between initialization, clustering, hypothesis generation, and output phases seamlessly.

\*\*Selected Equations/Algorithms:\*\*

- The Kalman filter algorithm is used for state prediction and update.

- Chi-square clustering is employed to evaluate the likelihood of hypotheses.

\*\*Design Dependencies:\*\*

- The CSCI's behaviour depends on the accuracy and timeliness of radar plot inputs.

- Configuration data from the HMI influences clustering thresholds and hypothesis evaluation parameters.

In summary, the CSCI is designed to process radar plots for multi-target tracking using advanced data association techniques. It ensures real-time performance, accuracy, and compliance with safety, security, and privacy requirements.

ertainly! Here is the detailed Software Design Description (SDD) for your code, formatted according to the provided template.

---

## 6.2 Package Wise Description (GOM Wise)

The package or GOM details are provided here. This section lists all the classes or modules and provides structured design diagrams.

### 6.2.1 Class `CVFilter`

This class implements a Constant Velocity Kalman Filter used for state estimation.

#### 6.2.1.1 Responsibility (Role)

The `CVFilter` class is responsible for performing state estimation using a Constant Velocity Kalman Filter. This includes initializing the state, performing prediction and update steps, and handling measurement gating.

#### 6.2.1.2 Attributes (In case of Classes)

- \*\*Private:\*\*

- `Sf` (np.array): Filter state vector

- `Pf` (np.array): Filter state covariance matrix

- `Sp` (np.array): Predicted state vector

- `Pp` (np.array): Predicted state covariance matrix

- `plant\_noise` (float): Plant noise covariance

- `H` (np.array): Measurement matrix

- `R` (np.array): Measurement noise covariance

- `Meas\_Time` (float): Measured time

- `prev\_Time` (float): Previous time

- `Q` (np.array): Process noise covariance matrix

- `Phi` (np.array): State transition matrix

- `Z` (np.array): Current measurement vector

- `Z1` (np.array): First measurement vector

- `Z2` (np.array): Second measurement vector

- `first\_rep\_flag` (bool): Flag for the first measurement

- `second\_rep\_flag` (bool): Flag for the second measurement

- `gate\_threshold` (float): Threshold for gating (Chi-square distribution)

#### 6.2.1.2.1 Computational Resources & Real-Time Constraints for the Module or Operations

- The `CVFilter` class operates within a real-time environment with constraints on processing time to ensure timely state updates. The time complexity of prediction and update steps is primarily linear with respect to the state vector dimension.

#### 6.2.1.2.2 Pseudo Code

```pseudo

Initialize Filter State(x, y, z, vx, vy, vz, time):

if not first\_rep\_flag:

Set Z1 with (x, y, z)

Set initial state vector Sf with (x, y, z)

Set Meas\_Time and prev\_Time to time

Set first\_rep\_flag to True

else if first\_rep\_flag and not second\_rep\_flag:

Set Z2 with (x, y, z)

Calculate velocity components (vx, vy, vz)

Update Meas\_Time and prev\_Time

Set second\_rep\_flag to True

else:

Update Z with (x, y, z)

Update Meas\_Time and prev\_Time

Predict Step(current\_time):

Calculate time difference dt

Update state transition matrix Phi

Update process noise covariance matrix Q

Predict state vector Sp and covariance Pp

Update Meas\_Time

Update Step(Z):

Calculate innovation Inn

Calculate innovation covariance S

Calculate Kalman gain K

Update state vector Sf and covariance Pf

Gating(Z):

Calculate innovation Inn

Calculate innovation covariance S

Calculate Mahalanobis distance d2

Return True if d2 < gate\_threshold, else False

```

#### 6.2.1.3 Operations (Modules)

- \*\*Public:\*\*

- `initialize\_filter\_state(self, x, y, z, vx, vy, vz, time)`

- `predict\_step(self, current\_time)`

- `update\_step(self, Z)`

- `gating(self, Z)`

### 6.3 Special Critical Functions

This section describes special critical functions such as ISRs, time-out functions, task functions, etc.

#### 6.3.1 Function Wise

This section lists out all the operations with their attributes.

- \*\*Function: `form\_measurement\_groups`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Forms groups of measurements based on the time difference.

- \*\*Function: `read\_measurements\_from\_csv`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Reads measurements from a CSV file and converts them to Cartesian coordinates.

- \*\*Function: `chi\_square\_clustering`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Performs clustering of measurements based on the Chi-square test.

- \*\*Function: `form\_clusters`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Forms clusters from measurements using the Chi-square clustering method.

- \*\*Function: `generate\_hypotheses`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Generates hypotheses from the clusters.

- \*\*Function: `compute\_hypothesis\_likelihood`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Computes the likelihood of a hypothesis.

- \*\*Function: `jpda`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Performs Joint Probabilistic Data Association (JPDA) for data association.

- \*\*Function: `sph2cart`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Converts spherical coordinates to Cartesian coordinates.

- \*\*Function: `cart2sph`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Converts Cartesian coordinates to spherical coordinates.

- \*\*Function: `cart2sph2`\*\*

- \*\*Attributes:\*\* Public

- \*\*Responsibility:\*\* Converts a list of Cartesian coordinates to spherical coordinates.

### 6.3.1.1 Hardware Resources and Constraints

#### 6.3.1.1.1 Computational Resources & Real-Time Constraints for the Module or Operations

\*\*Hardware Resources:\*\*

1. \*\*Processor:\*\*

- The system requires a processor capable of handling real-time computation with sufficient clock speed to ensure timely processing of measurements and state updates. A multi-core processor can be beneficial for parallel processing of measurements and track updates.

2. \*\*Memory:\*\*

- Adequate RAM is needed to store the state vectors, covariance matrices, and measurement data. Depending on the number of targets and the complexity of the state vector, at least 4GB of RAM is recommended.

3. \*\*Storage:\*\*

- Sufficient storage is necessary to store measurement data, logs, and configuration files. An SSD is preferred for faster read/write operations.

4. \*\*Network:\*\*

- A stable network connection may be required if the system needs to communicate with other systems or sensors in real-time. Low latency and high bandwidth are crucial for timely data transmission.

5. \*\*Sensors:\*\*

- High-precision sensors are needed to provide accurate measurements. The sensors should have a high sampling rate to ensure that measurements are taken at appropriate intervals for real-time processing.

\*\*Computational Resources & Real-Time Constraints:\*\*

1. \*\*State Estimation:\*\*

- The state estimation process, including prediction and update steps, should be performed within a fixed time window to ensure real-time performance. For instance, if measurements are received every 50 milliseconds, the state estimation should be completed within this period.

2. \*\*Measurement Grouping:\*\*

- Grouping measurements based on time intervals should be efficient. The process should be completed within a few milliseconds to allow sufficient time for subsequent processing steps.

3. \*\*Chi-Square Clustering:\*\*

- The Chi-square clustering operation should be optimized for speed. It should handle multiple measurements simultaneously and complete within the real-time constraints.

4. \*\*Hypothesis Generation and Likelihood Computation:\*\*

- Generating hypotheses and computing their likelihoods should be done efficiently. These operations are computationally intensive and should be optimized to run within the real-time requirements.

5. \*\*Kalman Filter Operations:\*\*

- The prediction and update steps of the Kalman filter should be optimized for speed. Matrix operations involved in these steps should leverage efficient numerical libraries and possibly hardware acceleration if available.

6. \*\*Data Association (JPDA):\*\*

- The Joint Probabilistic Data Association (JPDA) algorithm should be optimized to handle multiple targets and measurements within the real-time constraints. Parallel processing or GPU acceleration can be used to speed up this process.

7. \*\*Conversion Functions:\*\*

- Conversion functions such as `sph2cart` and `cart2sph` should be lightweight and optimized for real-time execution.

\*\*Real-Time Constraints:\*\*

1. \*\*Latency:\*\*

- The system should have minimal latency from measurement acquisition to state update. A latency of less than 50 milliseconds is desired to ensure timely state estimation.

2. \*\*Throughput:\*\*

- The system should be able to handle a high throughput of measurements, especially in scenarios with multiple targets. The ability to process several measurements per second is crucial.

3. \*\*Jitter:\*\*

- The system should exhibit low jitter in processing times to maintain consistent performance. Variations in processing times should be minimized to ensure reliable real-time performance.

4. \*\*Deterministic Execution:\*\*

- The system should ensure deterministic execution of all operations. This means that the time taken for each operation should be predictable and within the specified real-time constraints.

By adhering to these computational resources and real-time constraints, the system can ensure reliable and timely performance for state estimation and multi-target tracking.

Certainly! Below is the pseudo code structure for your Software Design Description (SDD) document following the provided template:

### Pseudo Code

#### a. Function Name:

`main`

#### b. File Name:

`tracking\_system.py`

#### c. Function Definition:

```python

def main():

```

#### d. Description of Input Parameters:

None

#### e. Description of Return Parameters:

None

#### f. Brief Description of Function:

The `main` function orchestrates the reading of measurements from a CSV file, performs Kalman filtering, applies the Joint Probabilistic Data Association (JPDA) algorithm for clustering and hypothesis generation, and finally plots the filtered results.

#### g. Global Objects (Used as Input/Referenced):

None

#### h. Global Objects/Input Parameters (Modified):

None

#### i. Local Objects:

- `file\_path`: Path to the CSV file containing measurements.

- `measurements`: List of measurements read from the CSV file.

- `kalman\_filter`: Instance of `CVFilter` class.

- `measurement\_groups`: Groups of measurements formed based on time difference.

- `df\_predicted`: DataFrame containing the predicted values read from the CSV file.

- `filtered\_values\_csv`: Array of filtered values from the CSV file.

- `measured\_values\_csv`: Array of measured values from the CSV file.

- `A`: Tuple containing spherical coordinates derived from filtered values.

- `r`, `az`, `el`, `t`: Lists to store filtered range, azimuth, elevation, and time values respectively.

- `group\_idx`, `group`: Variables for iterating through measurement groups.

- `x`, `y`, `z`: Cartesian coordinates converted from spherical coordinates.

- `clusters`: List of clusters formed for each measurement group.

- `hypotheses`: List of hypotheses generated from clusters.

- `best\_hypothesis`: The best hypothesis selected by JPDA.

#### j. Algorithm:

1. \*\*Initialize Variables\*\*:

- Set `file\_path` to the CSV file path.

- Read measurements from the CSV file into `measurements`.

- Instantiate `kalman\_filter`.

- Form measurement groups based on time difference.

- Read predicted and measured values from the CSV file into `filtered\_values\_csv` and `measured\_values\_csv`.

- Convert filtered values to spherical coordinates and scale the range values.

2. \*\*Process Measurement Groups\*\*:

- For each measurement group:

- For each measurement in the group:

- Convert spherical coordinates to Cartesian coordinates.

- Initialize filter state if it's the first or second measurement.

- For subsequent measurements:

- Predict the next state using the Kalman filter.

- Form clusters from the group measurements.

- Generate hypotheses from the clusters.

- Apply JPDA to select the best hypothesis.

- Update the filter state with the best hypothesis.

- Append the filtered range, azimuth, elevation, and time values to their respective lists.

3. \*\*Plot Results\*\*:

- Plot range vs. time.

- Plot azimuth vs. time.

- Plot elevation vs. time.

#### k. Functions Used Internally (Standard APIs):

- `np.zeros`, `np.eye`, `np.array`, `np.dot`, `np.linalg.inv`, `np.exp`, `np.argmax`: Numpy functions.

- `csv.reader`, `pd.read\_csv`: CSV and Pandas functions for reading files.

- `math.atan2`, `math.sqrt`, `math.cos`, `math.sin`, `math.pi`: Math functions.

- `plt.figure`, `plt.subplot`, `plt.scatter`, `plt.xlabel`, `plt.ylabel`, `plt.title`, `plt.grid`, `plt.legend`, `plt.tight\_layout`, `mplcursors.cursor`, `plt.show`: Matplotlib functions for plotting.

#### l. Functions Used Internally (Custom Functions):

- `read\_measurements\_from\_csv`

- `form\_measurement\_groups`

- `cart2sph2`

- `sph2cart`

- `cart2sph`

- `form\_clusters`

- `generate\_hypotheses`

- `compute\_hypothesis\_likelihood`

- `jpda`

- `CVFilter` class methods (`initialize\_filter\_state`, `predict\_step`, `update\_step`, `gating`)

#### m. Called By:

This function is called when the script `tracking\_system.py` is executed directly.

---

This template should help you in documenting your function in the SDD. You can apply a similar structure to other functions and methods in your code. Let me know if you need further details or assistance!

Certainly! Below is the structured pseudo code for the functions in your provided code, formatted according to the specified template for your Software Design Document (SDD).

### 1. Function: `initialize\_filter\_state`

a. \*\*Function Name:\*\* initialize\_filter\_state

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def initialize\_filter\_state(self, x, y, z, vx, vy, vz, time):`

d. \*\*Description of Input Parameters:\*\*

- `x, y, z` (float): Cartesian coordinates of the initial position.

- `vx, vy, vz` (float): Initial velocity components (unused in current implementation).

- `time` (float): Measurement time.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Initializes the filter state with the first and second measurements.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\*

- `self.Sf`

- `self.Meas\_Time`

- `self.prev\_Time`

- `self.Z1`

- `self.Z2`

- `self.first\_rep\_flag`

- `self.second\_rep\_flag`

i. \*\*Local objects:\*\* None

j. \*\*Algorithm:\*\*

1. If the first report flag is not set:

- Set `self.Z1` to the input coordinates `[x, y, z]`.

- Set the filter state `self.Sf` to `[x, y, z]`.

- Set the measurement and previous time to `time`.

- Set the first report flag.

2. Else if the second report flag is not set:

- Set `self.Z2` to the input coordinates `[x, y, z]`.

- Update the previous time to the current measurement time.

- Calculate the velocity components based on the time difference.

- Set the second report flag.

3. Else:

- Set the measurement `self.Z` to `[x, y, z]`.

- Update the previous and measurement time.

k. \*\*Functions used internally (standard APIs):\*\* None

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `main`, `process\_measurement\_group`

### 2. Function: `predict\_step`

a. \*\*Function Name:\*\* predict\_step

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def predict\_step(self, current\_time):`

d. \*\*Description of Input Parameters:\*\*

- `current\_time` (float): The current measurement time.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Predicts the next state of the filter based on the elapsed time.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\*

- `self.Phi`

- `self.Q`

- `self.Sp`

- `self.Pp`

- `self.Meas\_Time`

i. \*\*Local objects:\*\*

- `dt` (float): Time difference.

- `T\_2`, `T\_3` (float): Time squared and cubed terms.

j. \*\*Algorithm:\*\*

1. Calculate the time difference `dt`.

2. Update the state transition matrix `self.Phi`.

3. Update the process noise covariance matrix `self.Q`.

4. Predict the next state vector `self.Sp`.

5. Predict the next state covariance matrix `self.Pp`.

6. Update the measurement time to `current\_time`.

k. \*\*Functions used internally (standard APIs):\*\*

- `np.dot`

- `np.eye`

- `np.linalg.inv`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `main`, `process\_measurement\_group`

### 3. Function: `update\_step`

a. \*\*Function Name:\*\* update\_step

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def update\_step(self, Z):`

d. \*\*Description of Input Parameters:\*\*

- `Z` (np.array): Measurement vector.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Updates the filter state with the new measurement.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\*

- `self.Sf`

- `self.Pf`

i. \*\*Local objects:\*\*

- `Inn` (np.array): Innovation vector.

- `S` (np.array): Innovation covariance.

- `K` (np.array): Kalman gain.

j. \*\*Algorithm:\*\*

1. Calculate the innovation vector `Inn`.

2. Compute the innovation covariance `S`.

3. Compute the Kalman gain `K`.

4. Update the state vector `self.Sf`.

5. Update the state covariance matrix `self.Pf`.

k. \*\*Functions used internally (standard APIs):\*\*

- `np.dot`

- `np.eye`

- `np.linalg.inv`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `main`, `process\_measurement\_group`

### 4. Function: `gating`

a. \*\*Function Name:\*\* gating

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def gating(self, Z):`

d. \*\*Description of Input Parameters:\*\*

- `Z` (np.array): Measurement vector.

e. \*\*Description of Return Parameters:\*\*

- `bool`: True if the measurement is within the gate, False otherwise.

f. \*\*Brief Description of function:\*\* Checks if the measurement lies within the gate threshold.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `Inn` (np.array): Innovation vector.

- `S` (np.array): Innovation covariance.

- `d2` (float): Mahalanobis distance.

j. \*\*Algorithm:\*\*

1. Calculate the innovation vector `Inn`.

2. Compute the innovation covariance `S`.

3. Compute the Mahalanobis distance `d2`.

4. Return True if `d2` is less than the gate threshold, else False.

k. \*\*Functions used internally (standard APIs):\*\*

- `np.dot`

- `np.linalg.inv`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `form\_clusters`

### 5. Function: `form\_measurement\_groups`

a. \*\*Function Name:\*\* form\_measurement\_groups

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def form\_measurement\_groups(measurements, max\_time\_diff=0.050):`

d. \*\*Description of Input Parameters:\*\*

- `measurements` (list): List of measurements.

- `max\_time\_diff` (float): Maximum allowed time difference for grouping.

e. \*\*Description of Return Parameters:\*\*

- `list`: List of measurement groups.

f. \*\*Brief Description of function:\*\* Groups measurements based on time difference.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `measurement\_groups` (list): List to store groups.

- `current\_group` (list): List to store current group.

- `base\_time` (float): Time of the first measurement in the group.

j. \*\*Algorithm:\*\*

1. Initialize `base\_time` with the time of the first measurement.

2. Iterate through each measurement:

- If the time difference is within `max\_time\_diff`, add to the current group.

- Otherwise, save the current group and start a new group.

3. Append the last group if not empty.

k. \*\*Functions used internally (standard APIs):\*\* None

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `main`

### 6. Function: `read\_measurements\_from\_csv`

a. \*\*Function Name:\*\* read\_measurements\_from\_csv

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def read\_measurements\_from\_csv(file\_path):`

d. \*\*Description of Input Parameters:\*\*

- `file\_path` (str): Path to the CSV file.

e. \*\*Description of Return Parameters:\*\*

- `list`: List of measurements.

f. \*\*Brief Description of function:\*\* Reads measurements from a CSV file and converts them to Cartesian coordinates.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `measurements` (list): List to store measurements.

- `file` (file object): CSV file object.

- `reader` (csv.reader): CSV reader object.

- `row` (list): List of row values from CSV.

j. \*\*Algorithm:\*\*

1. Open the CSV file.

2. Read each row and convert spherical coordinates to Cartesian.

3. Append the converted coordinates to the measurements list.

k. \*\*Functions used internally (standard APIs):\*\*

- `csv.reader`

k. \*\*Functions used internally (Custom functions):\*\*

- `sph2cart`

- `cart2s

ph`

m. \*\*Called By:\*\* `main`

### 7. Function: `plot\_results`

a. \*\*Function Name:\*\* plot\_results

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def plot\_results(measurements, predicted\_states):`

d. \*\*Description of Input Parameters:\*\*

- `measurements` (list): List of measurement vectors.

- `predicted\_states` (list): List of predicted state vectors.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Plots the measurements and predicted states in 3D.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `fig` (plt.figure): Matplotlib figure object.

- `ax` (plt.axes): Matplotlib 3D axes object.

- `meas\_x`, `meas\_y`, `meas\_z` (list): Lists of measurement coordinates.

- `pred\_x`, `pred\_y`, `pred\_z` (list): Lists of predicted state coordinates.

j. \*\*Algorithm:\*\*

1. Create a 3D plot.

2. Extract x, y, z coordinates from measurements and predicted states.

3. Plot the measurements and predicted states on the 3D plot.

4. Set labels and show the plot.

k. \*\*Functions used internally (standard APIs):\*\*

- `matplotlib.pyplot.figure`

- `matplotlib.pyplot.axes`

- `matplotlib.pyplot.show`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\* `main`

### 8. Function: `sph2cart`

a. \*\*Function Name:\*\* sph2cart

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def sph2cart(r, az, el):`

d. \*\*Description of Input Parameters:\*\*

- `r` (float): Radius.

- `az` (float): Azimuth angle.

- `el` (float): Elevation angle.

e. \*\*Description of Return Parameters:\*\*

- `tuple`: Cartesian coordinates (x, y, z).

f. \*\*Brief Description of function:\*\* Converts spherical coordinates to Cartesian coordinates.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `x`, `y`, `z` (float): Cartesian coordinates.

j. \*\*Algorithm:\*\*

1. Compute `x`, `y`, and `z` from `r`, `az`, and `el`.

2. Return the computed `x`, `y`, and `z`.

k. \*\*Functions used internally (standard APIs):\*\*

- `np.cos`

- `np.sin`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\*

- `read\_measurements\_from\_csv`

- `main`

### 9. Function: `cart2sph`

a. \*\*Function Name:\*\* cart2sph

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def cart2sph(x, y, z):`

d. \*\*Description of Input Parameters:\*\*

- `x`, `y`, `z` (float): Cartesian coordinates.

e. \*\*Description of Return Parameters:\*\*

- `tuple`: Spherical coordinates (r, az, el).

f. \*\*Brief Description of function:\*\* Converts Cartesian coordinates to spherical coordinates.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `r`, `az`, `el` (float): Spherical coordinates.

j. \*\*Algorithm:\*\*

1. Compute `r`, `az`, and `el` from `x`, `y`, and `z`.

2. Return the computed `r`, `az`, and `el`.

k. \*\*Functions used internally (standard APIs):\*\*

- `np.sqrt`

- `np.arctan2`

l. \*\*Functions used internally (Custom functions):\*\* None

m. \*\*Called By:\*\*

- `main`

### 10. Function: `process\_measurement\_group`

a. \*\*Function Name:\*\* process\_measurement\_group

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def process\_measurement\_group(self, measurement\_group):`

d. \*\*Description of Input Parameters:\*\*

- `measurement\_group` (list): List of measurements in the group.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Processes a group of measurements for tracking.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\* None

j. \*\*Algorithm:\*\*

1. Initialize the filter state for the first measurement.

2. For each subsequent measurement in the group:

- Predict the next state.

- Check if the measurement is within the gate.

- If within the gate, update the filter state.

k. \*\*Functions used internally (standard APIs):\*\* None

l. \*\*Functions used internally (Custom functions):\*\*

- `initialize\_filter\_state`

- `predict\_step`

- `update\_step`

- `gating`

m. \*\*Called By:\*\*

- `main`

### 11. Function: `main`

a. \*\*Function Name:\*\* main

b. \*\*File Name:\*\* tracking.py

c. \*\*Function Definition:\*\* `def main(file\_path):`

d. \*\*Description of Input Parameters:\*\*

- `file\_path` (str): Path to the CSV file containing measurements.

e. \*\*Description of Return Parameters:\*\* None

f. \*\*Brief Description of function:\*\* Main function to perform tracking using Kalman filter and JPDA.

g. \*\*Global Objects (Used as input/referenced):\*\* None

h. \*\*Global Objects/Input Parameters (Modified):\*\* None

i. \*\*Local objects:\*\*

- `measurements` (list): List of measurements.

- `measurement\_groups` (list): List of measurement groups.

- `tracker` (Tracker): Tracker object.

j. \*\*Algorithm:\*\*

1. Read measurements from the CSV file.

2. Form measurement groups.

3. Initialize the tracker.

4. For each group of measurements:

- Process the measurement group.

5. Plot the results.

k. \*\*Functions used internally (standard APIs):\*\* None

l. \*\*Functions used internally (Custom functions):\*\*

- `read\_measurements\_from\_csv`

- `form\_measurement\_groups`

- `Tracker`

- `process\_measurement\_group`

- `plot\_results`

m. \*\*Called By:\*\* None

---

Feel free to adapt or modify these descriptions based on the specific details and requirements of your project.

Certainly! Here's a sample table that lists the tasks (functions) in your code, along with their priority and reasons:

| Task Name | Priority | Reasons |

|-------------------------------|----------|-------------------------------------------------------------------------------------------|

| `initialize\_filter\_state` | 101 | Initializes the filter state; active only during the first and second measurements. |

| `predict\_step` | 100 | Predicts the next state of the filter based on elapsed time; critical for state estimation. |

| `update\_step` | 100 | Updates the filter state with new measurements; essential for accurate tracking. |

| `gating` | 102 | Checks if a measurement is within the gate threshold; used to validate measurements. |

| `form\_measurement\_groups` | 103 | Groups measurements based on time difference; organizes data for processing. |

| `read\_measurements\_from\_csv` | 104 | Reads measurements from a CSV file and converts to Cartesian coordinates; initial data input. |

| `plot\_results` | 105 | Plots measurements and predicted states in 3D; used for visualization and analysis. |

| `sph2cart` | 106 | Converts spherical coordinates to Cartesian coordinates; helper function for data conversion. |

| `cart2sph` | 106 | Converts Cartesian coordinates to spherical coordinates; helper function for data conversion. |

| `process\_measurement\_group` | 101 | Processes a group of measurements for tracking; handles multiple measurements effectively. |

| `main` | 100 | Main function to perform tracking using Kalman filter and JPDA; orchestrates all tasks. |

\*\*Explanation of Priority Levels:\*\*

- \*\*100\*\*: Critical tasks that are essential for the core functionality of the tracking system.

- \*\*101\*\*: Important tasks that are necessary for initialization and initial processing.

- \*\*102\*\*: Tasks that validate measurements and ensure data integrity.

- \*\*103\*\*: Tasks that organize data for subsequent processing.

- \*\*104\*\*: Initial data input tasks.

- \*\*105\*\*: Tasks that provide visualization and analysis.

- \*\*106\*\*: Helper functions that support data conversion and other utility operations.

Feel free to adjust the priorities and reasons based on the specific needs and criticality of your tasks in the context of your project.

Here's a table formatted as per your specifications, listing the tasks, their priority, and the reasons for their priority, based on the data association techniques for multi-target tracking and the additional tasks you provided.

| \*\*Task Name\*\* | \*\*Priority\*\* | \*\*Reasons\*\* |

|---------------|--------------|-------------|

| `group\_plots\_into\_clusters` | 101 | Initial grouping of radar plots into clusters is fundamental for further processing. |

| `receive\_radar\_plots` | 100 | Essential task that initiates the data association process by receiving radar plots. |

| `perform\_clustering\_of\_plots\_wrt\_tracks` | 102 | Critical for organizing the radar plots in relation to existing tracks, enabling better tracking accuracy. |

| `generate\_possible\_hypothesis\_for\_each\_cluster` | 103 | Necessary for hypothesizing potential track associations within each cluster. |

| `compute\_joint\_probabilities\_for\_each\_hypothesis` | 104 | Required for evaluating the likelihood of each hypothesis, crucial for accurate tracking. |

| `compute\_marginal\_probabilities\_for\_each\_hypothesis` | 105 | Important for determining the individual likelihoods of hypotheses, aiding in the selection of the best hypothesis. |

| `find\_best\_hypothesis\_for\_target\_plot\_association` | 106 | Essential for finalizing the track-to-plot associations, ensuring accurate target tracking. |

| `perform\_data\_association\_using\_JPDA\_GNN\_PDA` | 107 | Utilizes advanced data association techniques to enhance tracking accuracy and reliability. |

| `receive\_configuration\_from\_HMI` | 108 | Required for receiving user-defined configurations, ensuring system operates as intended. |

| `generate\_plots\_for\_various\_data\_association\_techniques` | 109 | Important for visualizing the effectiveness of different data association techniques, aiding in analysis and debugging. |

This table outlines each task, prioritizes them based on their importance in the multi-target tracking process, and provides reasons for their assigned priority.