- Mechanical

1 -

- **Functions**: This refers to the specific tasks and operations that the designed part or product must perform. The design should address these functions accurately and efficiently.

- **Safety:** Safety is a critical aspect of design, ensuring that the product doesn't pose risks or hazards to users, operators, or the environment during its operation and lifecycle.

- **Reliability:** A reliable design ensures that the part consistently performs its intended functions under various conditions over its expected lifespan, minimizing the chances of failures or malfunctions.

- **Manufacturability:** Design for manufacturability involves creating a design that can be produced using available manufacturing processes, materials, and techniques. This criterion aims to minimize production challenges, costs, and time.

- **Weight and Size:** The weight and size of a design can impact its performance, transportation costs, and ease of use. Balancing strength and functionality with weight and size considerations is essential.

- **Wear:** Consider the potential wear and tear that the part may experience during its operation. Design features that reduce friction, abrasion, and deterioration can improve the part's longevity.

- **Maintenance:** Design for ease of maintenance and repair. Parts that are designed with accessibility in mind, allowing for straightforward disassembly and replacement, can minimize downtime and maintenance costs.

- **Liability:** Liability refers to the legal responsibility a designer or manufacturer has if their product causes harm or damage. Design should account for potential risks, adhere to safety standards, and include appropriate warnings or instructions.
-

2 –

- Compression, Tension, Shear, Bending, Torsion, and Fatigue.

3 –

- Balance rotating parts.
- Use damping materials or devices.
- Isolate components with mounts or springs.
- Adjust natural frequencies to avoid resonance.
- Modify stiffness to change natural frequencies.
- Install active or passive damping devices.
- Align components properly.
- Minimize friction between moving parts.
- Use vibration analysis to identify issues.
- Strengthen structures to withstand vibrations.

4-

Harmonic Drive (Strain Wave Gear):

Advantages: Compact, precise, high torque, smooth motion, lightweight.

Planetary Gear System:

Advantages: Compact, high torque, efficient, versatile.

Cycloidal Drive:

Advantages: Compact, high torque, precise, strong.

Rack and Pinion:

Advantages: Linear motion, efficient, strong.

Worm Gear System:

Advantages: High gear reduction, self-locking, strong.

Spur Gear System:

Advantages: Simple, efficient, versatile.

- ## Electrical

1 –

Preferred Battery Type: LiPo (Lithium Polymer) batteries.

Why: LiPo batteries are lightweight, provide high energy, and offer strong power bursts, which suit the performance needs of remote vehicles.

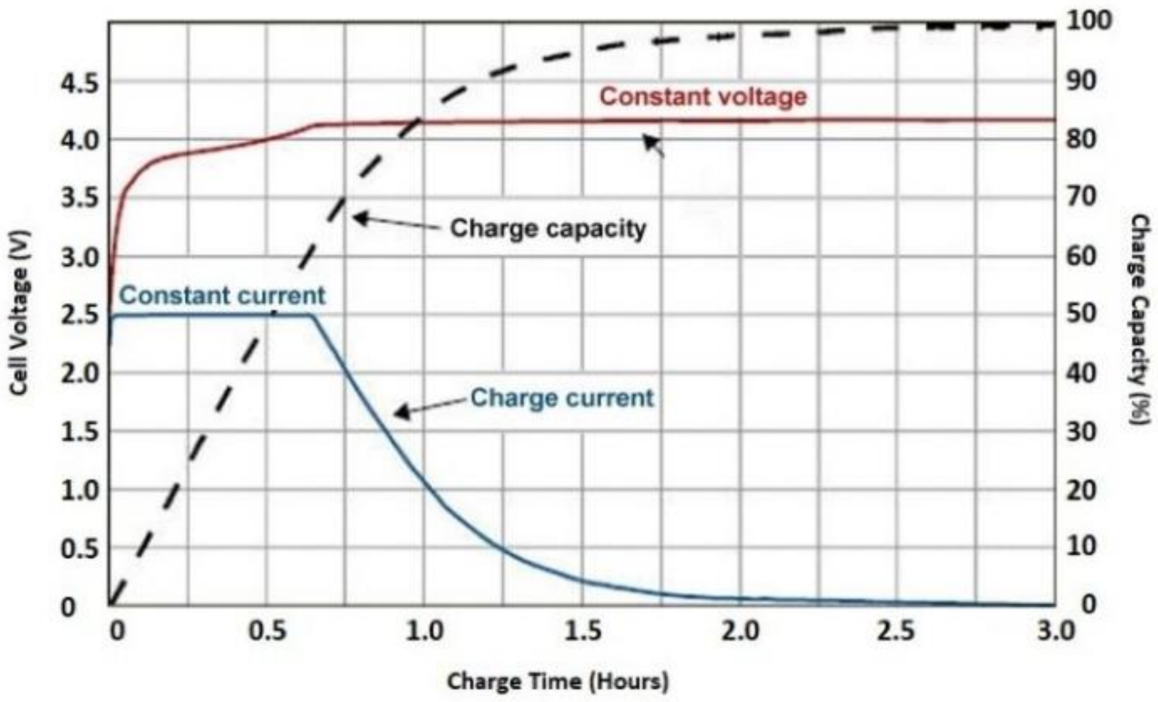
Choosing Battery Specifications:

- Voltage (Number of Cells): Match the battery's voltage to your vehicle's needs.
- Capacity (mAh): Choose capacity for desired runtime.
- Discharge Rate (C Rating): Match or exceed component current draw.
- Connector Type: Ensure compatibility.
- Size and Weight: Fit within vehicle, balance capacity and weight.
- Balance Charging: Use a charger that balances cells.
- Safety: Store and charge properly for safety.

2 –

The function of a BMS (Battery Management System) is to monitor, control, and protect a battery pack in various applications. Its primary goal is to ensure the safe and optimal operation of the battery by managing its charging, discharging, and overall health.

3 –



This charge curve of a Lithium-ion cell plots various parameters such as voltage, charging time, charging current and charged capacity. When the cells are assembled as a battery pack for an application, they must be charged using a constant current and constant voltage (CC-CV) method. Hence, a CC-CV charger is highly recommended for Lithium-ion batteries.

• The CC-CV method starts with constant charging while the battery pack's voltage rises.

• When the battery reaches its full charge cut-off voltage, constant voltage mode takes over, and there is a drop in the charging current.

• The charging current keeps coming down until it reaches below 0.05C.

• The battery reaches full charge voltage some time after the CV mode starts (as soon as one of the cells reaches its full charge voltage). At this stage, estimating SoC (state of charge) based on the battery voltage would mean that the battery is fully charged.

- The battery reaching its full charge voltage at this stage does not mean that it is 100% charged. Trickle charge mode kicks in immediately after this stage, where a reducing charging current charges the remaining battery capacity while balancing the cells at the same time.
- When every cell has been balanced and has reached its full charge voltage, at this point, the battery pack is really 100% charged. One way to know this is when the charging current has reached close to 0.05C.

4 –

The general electrical architecture of an Unmanned Aerial Vehicle (UAV), commonly known as a drone, consists of various components that work together to enable its operation and control. Here's an overview of the key components and their purposes:

Flight Controller: The brain that controls how the drone flies.

- Power Distribution Board (PDB): Sends power to all parts of the drone.
- Battery: Provides energy for flying.
- Motors and ESCs: Make the drone move by spinning the propellers.
- Flight Sensors: Help the drone stay stable and know where it's going.
- Radio Control System: Lets a pilot control the drone with a remote.
- Telemetry System: Sends data between the drone and the ground.
- Communication System: Lets the drone talk to the pilot and other drones.
- Onboard Computer: Thinks and makes decisions for advanced tasks.
- Camera and Sensors (optional): Collect information or take pictures.
- Safety Systems: Keep the drone and people safe in emergencies.

5 –

Minimizing electromagnetic interference (EMI) in printed circuit boards (PCBs) is crucial to ensure the proper functioning of electronic devices and to comply with electromagnetic compatibility (EMC) regulations. Here are some simplified strategies to reduce EMI in PCBs:

Keep Components Apart: Separate sensitive parts and group similar ones together.

- Solid Ground Plane: Use a good ground layer as a reference for signals.
- Match Traces: Make signal paths equal in length to avoid confusion.
- Use Capacitors: Add capacitors near power pins to clean up noise.
- Shielding and Covers: Use shields or cases to keep signals contained.
- Pick Components Wisely: Choose parts that naturally reduce noise.
- Add Filters: Put small parts to filter noise in signal and power lines.
- Be Careful with Clocks: Handle clocks well to avoid noise problems.
- Get Expert Advice: Ask experienced engineers for help if needed.
- Follow Rules: Stick to standards and guidelines for good design.

Perception

1 –

Neural Network:

A neural network is a computational model inspired by the structure and functioning of the human brain. It consists of interconnected nodes, or "neurons," organized in layers. Each neuron processes information and passes it to the next layer, allowing the network to learn patterns and make predictions from data.

Types:

- Feedforward (Simple pattern recognition)
- Convolutional (Recognizing images)
- Recurrent (Understanding sequences)
- Generative (Creating new things)

Applications:

- Images: Identifying objects in pictures.
- Language: Translating languages, understanding text.
- Speech: Converting spoken words to text.
- Cars: Helping cars drive on their own.
- Health: Diagnosing diseases from medical images.
- Finance: Predicting stock prices and finding fraud.
- Games: Making game characters smart.
- Art: Creating music, art, and more.

- Industry: Making factories work better.
- Science: Solving complex problems and analyzing data.
- Neural networks help computers do many smart things by learning from examples, just like we learn from experience.

2 –

```
import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers


# Load the Fashion MNIST dataset

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()


# Preprocess the data

x_train = x_train / 255.0

x_test = x_test / 255.0


# Build the neural network architecture

model = keras.Sequential([

    layers.Flatten(input_shape=(28, 28)),  # Flatten the 28x28 images

    layers.Dense(128, activation='relu'),   # Hidden layer with 128 units

    layers.Dropout(0.2),                    # Dropout for regularization

    layers.Dense(10, activation='softmax')  # Output layer with 10 classes

])


# Compile the model

model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])
```

```
# Train the model
model.fit(x_train, y_train, epochs=10, validation_split=0.2)

# Evaluate the model on test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

3 –

```
import tensorflow as tf
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Load a small subset of the ImageNet dataset
# For the complete ImageNet dataset, consider using TensorFlow Datasets or other sources.
# Here, we assume you have a directory structure with train and validation folders.
train_data_dir = 'path_to_train_data'
validation_data_dir = 'path_to_validation_data'

# Preprocess images and apply data augmentation
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

```
validation_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Load pretrained ResNet50 model
base_model = ResNet50(include_top=False, weights='imagenet')

# Add custom classification head
model = tf.keras.Sequential([
  base_model,
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(1024, activation='relu'),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(1000, activation='softmax')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.001),
       loss='categorical_crossentropy',
       metrics=['accuracy'])

# Load and preprocess data using the data generators
train_generator = train_datagen.flow_from_directory(
  train_data_dir,
  target_size=(224, 224),
  batch_size=32,
  class_mode='categorical')

validation_generator = validation_datagen.flow_from_directory(
  validation_data_dir,
  target_size=(224, 224),
  batch_size=32,
  class_mode='categorical')


# Train the model
model.fit(
  train_generator,
  steps_per_epoch=train_generator.samples // train_generator.batch_size,
  epochs=10,
  validation_data=validation_generator,
  validation_steps=validation_generator.samples // validation_generator.batch_size)
```

4 –

LiDAR (measuring distances with lasers), RGB cameras (taking color pictures), and depth cameras (measuring depth in images).

Working Principle:

1. Combine data from these sensors to find and understand objects in the world:
2. Collect Data: Sensors gather information about the environment.
3. Combine Data: Mix data from different sensors to understand what's around.
4. Analyze Features: Look at colors, shapes, and more to figure out what objects are there.
5. Find Objects: Use smart methods to detect and locate objects.
6. Know Where: Understand where objects are in 3D space.

Improvements:

1. Better Mixing: Improve how sensors' data are combined.
2. Smarter Algorithms: Develop better computer programs to understand the data.
3. Understand Context: Teach computers to understand the scene better.
4. Use More Data: Make models smarter by using more examples.
5. Faster Processing: Make everything work faster for real-time use.
6. Better Sensors: Use newer, more accurate sensors.

5 –

Stereo-Camera:

- Has two cameras for depth perception.
- Takes two pictures to understand distance.
- Good for 3D tasks like mapping.

Mono-Camera:

- Has one camera, no depth perception.
- Used for regular pictures.

Transforming 2D to 3D (Stereo Camera):

- Use two images and find the difference between their positions (disparity).
- Use the disparity to calculate depth.
- Convert the pixel coordinates to world coordinates using equations.

State estimaton

1 –

Odometry is the process of estimating a vehicle's position and movement based on sensor data. Localization goes a step further by determining the vehicle's position within a known map. In self-driving cars, both are essential for safe and accurate navigation. Different approaches to odometry and localization include:

1. GPS: Uses satellites for position, not super precise.
2. IMU: Measures movement, but can get confused.
3. Wheel Odometry: Uses wheel rotations, but can slip.
4. Visual Odometry: Looks at pictures for motion, needs good textures.
5. LiDAR Odometry: Uses laser data, works well but can be noisy.
6. Sensor Fusion: Mixes data from many sensors for better accuracy.
7. SLAM: Makes maps and finds position at the same time.
8. Map-Based: Compares sensor data to maps.
9. Radio-Based: Uses Wi-Fi or signals to know location.
10. Machine Learning: Uses smart algorithms to guess where it is.

Self-driving cars often use a mix of these to know where they are and where they're going.

2 –

SLAM is a technique used in robotics and autonomous systems to create a map of an unknown environment while simultaneously determining the system's location within that environment. In simpler terms, it's a method for a robot or vehicle to build a map as it moves around and figure out where it is on the map at the same time.

Types:

- Full SLAM: Map and location at the same time.
- Online SLAM: Makes maps in real-time.
- Offline SLAM: Makes maps after collecting data.
- EKF SLAM: Good for simple places.
- Graph-Based SLAM: Uses graphs to find maps and location.
- Particle Filter SLAM: Deals with uncertainty.
- Visual SLAM: Uses camera pictures for maps.
- LiDAR SLAM: Uses laser data for maps.

SLAM is crucial in applications like self-driving cars, drones, and robotics, where the ability to navigate and understand the environment is essential. It enables these systems to autonomously explore, map, and operate in unknown or dynamic surroundings.

3 –

ICP (Iterative Closest Point) Algorithm:

The ICP algorithm is a widely used technique for aligning two sets of points in a 3D space. It's often employed in robotics and computer vision for registering (aligning) point clouds obtained from sensors like LiDAR for localization or mapping purposes.

Algorithm Steps:

- Initialization:

Initialize transformation (rotation and translation) parameters.

Choose a maximum number of iterations and a threshold for convergence.

Iterative Process:

For each iteration:

Find the closest points in the target point cloud to the source point cloud.

Calculate the transformation that aligns the source points to the closest target points.

- Transformation Update:

Update the transformation parameters based on the calculated alignment.

- Convergence Check:

Check if the transformation change is below the convergence threshold or if the maximum iterations are reached.

- Termination:

If converged or maximum iterations reached, end the algorithm.

- Pseudocode :

```
function ICP(source_points, target_points, max_iterations, convergence_threshold):
    initialize transformation parameters
    for iteration in range(max_iterations):
        closest_points = find_closest_points(source_points, target_points)
        transformation = calculate_transformation(source_points, closest_points)
        update_transformation_parameters(transformation)
        if transformation_change_below_threshold(convergence_threshold):
            break
    return transformed_source_points

function find_closest_points(source_points, target_points):
    # For each source point, find the closest point in the target cloud
    closest_points = []
    for each source_point in source_points:
        closest_point = find_closest_point(source_point, target_points)
        closest_points.append(closest_point)
    return closest_points

function calculate_transformation(source_points, closest_points):
    # Calculate the transformation that aligns source points to closest target points
    transformation = some_math_here(source_points, closest_points)
    return transformation

function update_transformation_parameters(transformation):
    # Update rotation and translation based on the calculated transformation
    update_rotation(transformation)
    update_translation(transformation)

function transformation_change_below_threshold(convergence_threshold):
    # Check if the change in transformation is below the threshold
    return change_below_threshold

# Call ICP with the appropriate data and parameters
transformed_points = ICP(source_point_cloud, target_point_cloud, max_iterations,
convergence_threshold)
```

The ICP algorithm repeatedly calculates transformations to align source and target point clouds until the desired alignment is achieved or the algorithm converges. It's a basic explanation, and real implementations involve considerations for point correspondence, outlier rejection, and optimization techniques for efficient convergence.

4 –

Types of Kalman Filters:

- Kalman Filter (KF): Basic version for simple situations.
- Extended Kalman Filter (EKF): Handles twists and turns.
- Unscented Kalman Filter (UKF): Good for tricky twists and turns.
- Iterated Kalman Filter (IKF): Repeats to be more accurate.
- Square Root Kalman Filter (SRKF): Keeps math stable.
- Dual Kalman Filter (DKF): Uses two filters for better info.

Uses and Applications:

Helps computers be smart in many areas:

- Navigation: Finds where you are (GPS, self-driving cars).
- Aerospace: Guides rockets and planes.
- Control Systems: Keeps things steady and precise.
- Signal Cleanup: Removes noise from signals.
- Computer Vision: Tracks moving things in videos.
- Robotics: Helps robots know where they are.
- Finance: Predicts money trends.
- Healthcare: Monitors bodies and devices.
- Kalman filters make things accurate even when there's confusion and messiness.

5 –

```
#include <iostream>

#include <Eigen/Dense>


using namespace Eigen;


// Kalman filter variables

VectorXd x;  // State vector [position; velocity]

MatrixXd P;  // Covariance matrix

MatrixXd F;  // State transition matrix

MatrixXd H;  // Measurement matrix

MatrixXd R;  // Measurement noise covariance
MatrixXd Q;  // Process noise covariance

void predict() {
    // Predict step
    x = F * x;      // State prediction
    P = F * P * F.transpose() + Q;  // Covariance prediction
}

void update(double measurement) {
    // Update step
    VectorXd y = VectorXd(1);
    y << measurement - H * x;  // Residual
    MatrixXd S = H * P * H.transpose() + R;  // Residual covariance
    MatrixXd K = P * H.transpose() * S.inverse();  // Kalman gain

    // Update state and covariance
    x = x + K * y;
    P = (MatrixXd::Identity(2, 2) - K * H) * P;
}

int main() {
    // Initialize Kalman filter variables
    x = VectorXd(2);
    x << 0, 0;  // Initial position and velocity
    P = MatrixXd(2, 2);
    P << 1, 0, 0, 1;  // Initial covariance
    F = MatrixXd(2, 2);
    F << 1, 1, 0, 1;  // State transition matrix
    H = MatrixXd(1, 2);
    H << 1, 0;  // Measurement matrix
    R = MatrixXd(1, 1);
    R << 1;  // Measurement noise covariance
    Q = MatrixXd(2, 2);
    Q << 0.01, 0, 0, 0.01;  // Process noise covariance

    // Simulated measurements
    double measurement1 = 1.2;
    double measurement2 = 1.7;

    // Predict and update steps
    predict();
    update(measurement1);

    std::cout << "Estimated position after measurement 1: " << x(0) << std::endl;

    predict();
    update(measurement2);

    std::cout << "Estimated position after measurement 2: " << x(0) << std::endl;

    return 0;
}
```

Path Planning

1 –

1. Dijkstra's Algorithm: Finds shortest path, good for small spaces.
2. A Algorithm*: Quick and smart, works for bigger areas.
3. RRT: Builds a tree of paths, useful for complex places.
4. RRT*: Improved RRT with better paths.
5. PRM: Plans paths ahead, good for complex environments.
6. Wavefront Algorithm: Fills grid to find paths.
7. Potential Fields: Moves like forces, handles obstacles.
8. Hybrid A*: Mixes grids with smooth paths for vehicles.
9. Lattice-based: Grid paths, fits certain robots.
10. Visibility Graphs: Connects seen points for paths.
11. Voronoi Diagrams: Splits space for safe paths.
12. Sampling-based: Uses randomness to explore.

2 –

```
import numpy as np

import heapq


# Define grid dimensions

grid_rows, grid_cols = 5, 5


# Define the occupancy grid

occupancy_grid = np.zeros((grid_rows, grid_cols), dtype=int)

occupancy_grid[2:3, 0:3] = 1  # Set obstacles


# Define start and goal positions

start = (1, 1)

goal = (4, 1)


# Define movement directions: up, down, left, right

dx = [-1, 1, 0, 0]

dy = [0, 0, -1, 1]
```

```
def is_valid(x, y):
    return 0 <= x < grid_rows and 0 <= y < grid_cols and occupancy_grid[x][y] == 0


def heuristic(node):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])


def a_star():
    open_list = [(0, start)]  # Priority queue with (f, node)
    came_from = {}
    g_score = {pos: float('inf') for pos in np.ndindex(grid_rows, grid_cols)}
    g_score[start] = 0

    while open_list:
        f, current = heapq.heappop(open_list)
        if current == goal:
            return reconstruct_path(came_from, current)

        for i in range(4):
            new_x, new_y = current[0] + dx[i], current[1] + dy[i]
            new_pos = (new_x, new_y)

            if is_valid(new_x, new_y):
                tentative_g_score = g_score[current] + 1
                if tentative_g_score < g_score[new_pos]:
                    came_from[new_pos] = current
                    g_score[new_pos] = tentative_g_score
                    f_score = g_score[new_pos] + heuristic(new_pos)
                    heapq.heappush(open_list, (f_score, new_pos))
    return None  # No path found


def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    return path[::-1]


# Find and print the path
path = a_star()
if path:
    for row in range(grid_rows):
        for col in range(grid_cols):
            if (row, col) == start:
                print("S", end=" ")
            elif (row, col) == goal:
                print("G", end=" ")
            elif (row, col) in path:
                print("*", end=" ")
            else:
                print(occupancy_grid[row][col], end=" ")
        print()
else:
    print("No path found!")
```

Navigation Control

1 –

Kinematic Modeling of Vehicles:

- Focuses on motion without considering forces and torques.
- Describes the vehicle's motion using simple equations and constraints.
- Assumes ideal conditions and ignores factors like tire grip, friction, and external forces.
- Suitable for high-level path planning and basic control systems.

Dynamic Modeling of Vehicles:

- Considers forces and torques affecting the vehicle's motion.
- Takes into account factors like tire dynamics, friction, vehicle weight, and external forces.
- More accurate but also more complex compared to kinematic modeling.
- Used for detailed analysis, advanced control, and simulations.

In essence, kinematic modeling simplifies to describe vehicle motion, while dynamic modeling considers the physics and forces involved for more accurate predictions.

2 -

Longitudinal Vehicle Controllers:

1. Cruise Control: Keeps a constant speed.
2. Adaptive Cruise Control: Adjusts speed based on traffic.
3. Speed Limiter: Caps maximum speed.
4. Traction Control: Stops wheels from spinning.

Lateral Vehicle Controllers:

1. Steering Control: Guides direction.
2. Lane Keeping Assist: Stays in the lane.

3. Lane Centering: Stays in the lane's center.
4. Automated Parking: Parks the car.
5. Electronic Stability Control: Stops skidding.
6. Path Following Controllers: Follows a set path.
7. Obstacle Avoidance: Steers away from obstacles.

3 –

```
#include <iostream>
#include <cmath>

class KinematicBicycleModel {
private:
  double x;      // X position
  double y;      // Y position
  double theta;  // Orientation (yaw angle)
  double L;      // Wheelbase (distance between front and rear axles)

public:
  KinematicBicycleModel(double initial_x, double initial_y, double initial_theta, double wheelbase) {
    x = initial_x;
    y = initial_y;
    theta = initial_theta;
    L = wheelbase;
  }

  // Forward kinematics update based on inputs [v, delta] and time step dt
  void update(double v, double delta, double dt) {
    double delta_x = v * cos(theta) * dt;
    double delta_y = v * sin(theta) * dt;
    double delta_theta = (v / L) * tan(delta) * dt;
```

```
        x += delta_x;
        y += delta_y;
        theta += delta_theta;
    }

    // Get current pose [x, y, theta]
    void getPose(double& out_x, double& out_y, double& out_theta) {
        out_x = x;
        out_y = y;
        out_theta = theta;
    }
};

int main() {
    // Initial pose values
    double initial_x = 0.0;
    double initial_y = 0.0;
    double initial_theta = 0.0;  // Initial yaw angle in radians

    // Create a kinematic bicycle model with a specific wheelbase
    double wheelbase = 2.5;  // Example wheelbase value
    KinematicBicycleModel bicycle(initial_x, initial_y, initial_theta, wheelbase);

    // Simulation parameters
    double dt = 0.1;  // Time step in seconds
    int num_steps = 50;  // Number of simulation steps

    // Simulate forward kinematics with inputs [v, delta]
    for (int step = 0; step < num_steps; ++step) {
        double v = 10.0;  // Example velocity in m/s
        double delta = 0.1;  // Example steering angle in radians
        bicycle.update(v, delta, dt);


        double x, y, theta;
        bicycle.getPose(x, y, theta);
        std::cout << "Step " << step << ": X=" << x << " Y=" << y << " Theta=" << theta << std::endl;
    }


    return 0;
}
```

4 –

Importance of Modeling in Control:

- Understanding: Models show how things work, helping design better control.
- Testing: Models let us try control ideas without real risks.
- Planning: Models help choose paths and actions for the best results.
- Safety: Models find issues before they happen in real systems.

Using a Model for Control:

- Feedback: Adjusts actions based on system response.
- Feedforward: Acts before problems appear.
- Predictive: Looks ahead to choose smart actions.

Controlling Without a Model:

- PID Control: Adjusts things based on errors.
- Machine Learning: Learns from data to control.
- Trial and Error: Adjusts things until they work.

But control without a model might not be as accurate, stable, or smart. Models make control safer and more effective.

5 –

MPC (Model Predictive Control):

- What: Smart control method using future predictions.
- How: Predicts how system behaves, finds best actions.
- Steps: Model -> Predict -> Optimize -> Apply -> Update.

Common Issues:

1. Hard Math: Needs lots of calculations, can be slow.
2. Wrong Model: Bad model means bad control.
3. Time Prediction: Choosing right time frame is tricky.
4. Delays: Slow measurements can mess up predictions.
5. Rules and Limits: Constraints can complicate things.
6. Guessing Uncertainty: Guessing future problems is hard.
7. Real-time Trouble: Can be tough to use instantly.

MPC is strong but needs careful setup to work well.

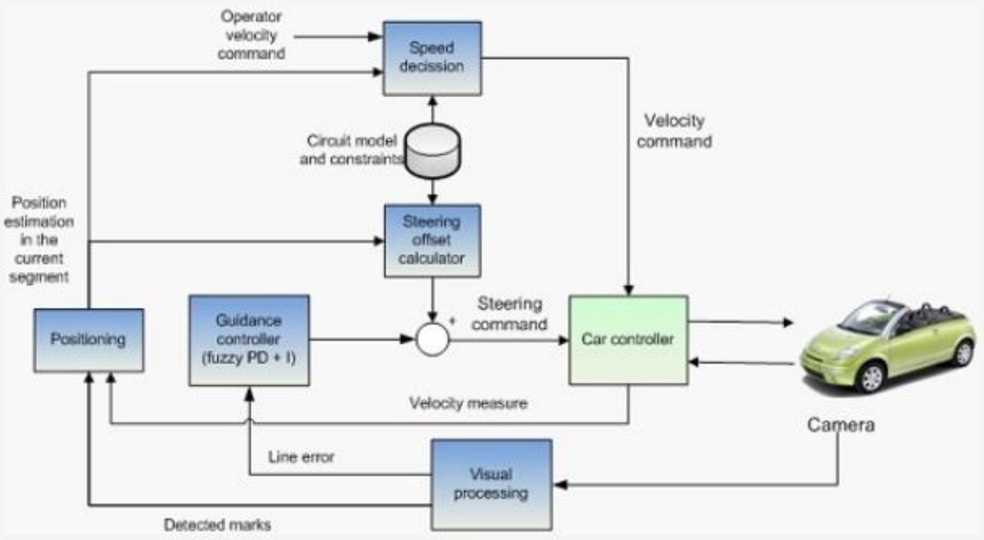
Low Level Control :

1 –

Controller: A Proportional-Integral-Derivative (PID) Controller is commonly used for steering control.

Final Control Element: The final control element could be an electric steering motor that physically turns the steering wheel.

Process Variable: The process variable is the actual position of the steering wheel, measured by sensors.


Operator velocity command
Speed decission
Velocity command
Circuit model and constraints
Position estimation in the current segment
Steering offset calculator
Steering command
Positioning
Guidance controller (fuzzy PD + I)
Car controller
Velocity measure
Camera
Line error
Visual processing
Detected marks

Graphical Abstract

2 -

Hardware and Actuators for Low-Level Control in Self-Driving Cars:

Electric Steering:

- Actuator: Electric motor
- Function: Turns the steering wheel.

Throttle Control:

- Actuator: Throttle actuator
- Function: Controls engine power.

Brake Control:

- Actuator: Brake actuators
- Function: Controls braking force.

Transmission Control:

- Actuator: Gear shift solenoids
- Function: Manages gear changes.

Suspension Control:

- Actuator: Adaptive suspension parts
- Function: Adjusts ride comfort and handling.

Tire Pressure Monitoring:

- Actuator: None
- Function: Checks tire pressure.

Stability Control:

- Actuator: Brakes, engine torque
- Function: Prevents skidding.

Steer-by-Wire (advanced):

- Actuator: Electric actuators
- Function: Controls steering directly.

3 –

Context Switching on STM32 with Interrupt:

When an interrupt occurs in an STM32 microcontroller, context switching is the process of temporarily suspending the execution of the current task (main program) to handle the interrupt. Here are the steps involved in the context switching process:

- Interrupt Happens: Something important occurs, like a button press.
- Interrupt Handling: The system stops what it's doing and checks the interrupt.
- Save Current State: It notes where it was in the task and saves key information.
- Go to Interrupt Code: Jumps to the special code for that interrupt.
- Deal with Event: Handles the important thing that happened.
- Cleanup and Resume: If needed, it fixes things up, like saving changed values. Then it goes back to what it was doing.
- Continue Main Task: Resumes where it left off in the main work.

This lets the STM32 handle urgent stuff without losing track of its regular job.

4 –

```
#include <stdio.h>

// Function to count how many times it has been called
int countFunctionCalls() {
    static int callCount = 0;  // Static variable to store the count
    callCount++;  // Increment the count with each call
    return callCount;  // Return the updated count
}

int main() {
    for (int i = 0; i < 5; i++) {
        int count = countFunctionCalls();
        printf("Function has been called %d times.\n", count);
    }

    return 0;
}
```

5 –

```
#include <stdio.h>

// Function to swap two pointers
void swapPointers(int **ptr1, int **ptr2) {
    int *temp = *ptr1;  // Store the value of the first pointer
    *ptr1 = *ptr2;      // Assign the value of the second pointer to the first
    *ptr2 = temp;       // Assign the stored value to the second pointer
}
```

```
int main() {
    int num1 = 5;
    int num2 = 10;

    int *ptr1 = &num1;
    int *ptr2 = &num2;

    printf("Before swapping:\n");
    printf("num1: %d, num2: %d\n", *ptr1, *ptr2);

    swapPointers(&ptr1, &ptr2);

    printf("After swapping:\n");
    printf("num1: %d, num2: %d\n", *ptr1, *ptr2);

    return 0;
}
```

6 –

```
#include <stdio.h>
#include <stdlib.h>

// Function to allocate memory for a 3x3 matrix and initialize it
double** createMatrix() {
    double **matrix = (double **)malloc(3 * sizeof(double *));
    for (int i = 0; i < 3; i++) {
        matrix[i] = (double *)malloc(3 * sizeof(double));
        for (int j = 0; j < 3; j++) {
            matrix[i][j] = i * 3 + j + 1;  // Example initialization
        }
    }
    return matrix;
}

// Function to print the matrix
void printMatrix(double **matrix) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%.2lf\t", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    double **matrix = createMatrix();

    printf("Matrix:\n");
    printMatrix(matrix);

    // Free the allocated memory
    for (int i = 0; i < 3; i++) {
        free(matrix[i]);
    }
    free(matrix);

    return 0;
}
```

7 –

CAN (Controller Area Network) bus arbitration is a fundamental process in the communication protocol used by electronic devices to share data efficiently and determine which device gets to transmit its message when multiple devices want to communicate simultaneously on the same bus.

1. Priority: Messages have different importance levels.
2. ID Comparison: Devices send their IDs bit by bit.
3. Conflict Check: If bits don't match, a device stops sending.
4. Dominant vs. Recessive: Logical 0 wins over 1.
5. Bit by Bit: The device with higher ID sends more 0s.
6. Winner Transmit: Highest priority ID sends its message.
7. Benefits: Efficient, fair, works without a boss.

CAN bus arbitration is essential for maintaining orderly and efficient communication in systems with multiple devices, ensuring that the most important messages are transmitted without conflicts.

8 –

UART as Multi-Master Multi-Slave:

UART is a basic communication method, but it lacks built-in support for multi-master multi-slave setups like more advanced protocols. However, here's how you might manage it:

1 - Multi-Master:

- Choose Masters: Devices take turns being the boss. They talk when it's their turn.
- Timing: Masters agree on when to talk. No overlap to avoid confusion.
- Bus Control: A system is needed to ensure one master talks at a time.

2 - Multi-Slave:

- Addresses: Masters address messages to specific slaves. Slaves listen for their names.
- Responses: Slaves wait their turn to reply. No chaos.

Challenges:

- Collision Detection: Detecting clashes (when multiple devices talk together) is tricky.
- Timing: Everything must happen at the right time, so data isn't mixed up.

- Sorting Disputes: Deciding which master talks first when two want to talk can be complex.

Benefits:

- Simple Communication: It's basic and might work for simple setups.
- Common Hardware: UART is found in many devices, so it's easy to find.

9 –

Sending Data from SPI Slave to Master:

1. Setup: Both SPI master and slave must be configured properly.
2. Slave Ready: Slave prepares data to send.
3. Master Initiation: Master sends a clock signal to initiate communication.
4. Bit Exchange: During each clock pulse, master sends a bit and reads a bit from the slave.
5. Slave Response: Slave prepares its response bit and sends it back to the master.
6. Repeat: Clock pulses continue, and data bits are exchanged.
7. Completion: After sending all bits, master completes the communication.
8. Read Data: Master collects the response bits from the slave to form the complete response.
9. Process Data: Master processes the received data as needed.

Benefits:

- SPI is fast and efficient for data transfer.
- It works well for point-to-point communication.