2D DIFFERENTIAL DRIVE

A differential drive robot is a type of mobile robot with two independently driven wheels mounted on the same axis, typically with a passive caster wheel for stability. It cannot move sideways and only controls linear and angular velocity in a 2D plane. Motion is constrained to a 2D plane (x, y), with rotation about the vertical axis (yaw = θ).

FORWARD KINEMATICS (WHEEL VELOCITIES → ROBOT MOTION):
Forward kinematics is the process of calculating the robot's velocity or pose based on the wheel velocities

Step 1: **Convert wheel angular velocity → linear velocity**
Each wheel rotates with angular velocity as given. These are the linear speeds at the edge of each wheel.

$$v_r = R.w_r$$
$$v_l = R.w_l$$

w_r, w_l = Angular velocities of the right and left wheels (rad/s)

Step 2: Linear velocity of robots center:

$$v = \frac{v_r + v_l}{2}$$

This is the average of both wheel speeds, the robots forward velocity

Angular velocity of the robot:

$$w = \frac{v_r - v_l}{L}$$

This describes how fast the robot is rotating about its vertical z axis

Step 3: Robots velocity in its own frame
Robots velocity in its own coordinate frame (body frame)

$$\begin{bmatrix}\overline{x_r}\end{bmatrix} = [v]$$
$$\begin{bmatrix}\overline{y_r}\end{bmatrix} = [0] \qquad \text{(single matrix)}$$
$$[\bar{\theta}] = [w]$$

Here $\overline{y_r}$ = 0 because differential drive robots can't move sideways due to their non holonomic constraint.

Step 4: Transform velocity to global frame.
Robot will have to face any direction θ

$$\overline{x} = v.cos\theta = \frac{R}{2}(w_r + w_l)\, cos\, \theta$$
$$\overline{y} = v.sin\theta = \frac{R}{2}(w_r + w_l)\, sin\, \theta$$
$$\overline{\theta} = w = \frac{R}{L}(w_r - w_l)$$

Special cases:
Case 1: $w_r = w_l$, w=0 no rotation, robot moves straight
Case 2: $w_r = -\ w_l$, v=0, no forward motion, robot rotates in place.
Case 3: $w_r > w_l$, robot arcs right
Case 4: $w_r < w_l$, robot arcs left

NOTE: If time is discretized into steps of Δt, new pose, usually in ROS to update robots pose over time we use this forward kinematics

$$x_{t+1} = x_t + \overline{x}.\Delta t$$
$$y_{t+1} = y_t + \overline{y}.\Delta t$$
$$\theta_{t+1} = \theta_t + \overline{\theta}.\Delta t$$

INVERSE KINEMATICS (POSE VELOCITY→WHEEL SPEEDS)

This is essential in motion planning, trajectory following, and controllers like DWA (Dynamic Window Approach), where higher-level planners compute velocities for the robot, and to convert them into wheel commands.

$$w_r + w_l = \frac{2v}{R}$$
$$w_r - w_l = \frac{Lw}{R}$$

$$2w_r = \frac{2v}{R} + \frac{Lw}{R}$$
$$w_r = \frac{1}{2}\left(\frac{2v+Lw}{R}\right)$$
$$w_r = \frac{2v+Lw}{2R}$$
$$2w_l = \frac{2v}{R} - \frac{Lw}{R}$$
$$w_l = \frac{2v-Lw}{2R}$$

Special cases:

Case 1: move straight v>0 , w=0 , $w_r = w_l = \frac{v}{R}$

Case 2; rotate in place v=0, w≠ 0, $w_r = \frac{Lw}{2R}$, $w_l = -\frac{Lw}{2R}$

Case 3: curve motion v>0, w>0 arc left
v>0, w<0 arc right

<u>WORKING & IMPLEMENTATION:</u>

To autonomously navigate a differential drive robot from point A to point B in a simulated world (Gazebo) using the DWA local planner, after integrating:

- Forward kinematics
- Inverse kinematics
- Sensor feedback
- Map and localization(SLAM or AMCL)

Forward kinematics is used to estimate robots pose using wheel encoders and translate wheel speeds into robot velocity and motion over time.

Inverse kinematics converts desired robot velocity to left and right wheel speeds.

While simulating a robot in a gazebo we use an xacro model to define physical structures like wheels, base link and sensors. First we apply inverse kinematics then simulate wheel motion by deciding wheel speed and then forward kinematics.

For sensor feedback typically we use liDAR to detect obstacles and update the costmap of DWA, wheel encoders for odometry and to track robot displacement and from these sensors we are feeding to costmap(obstacle aware planning), AMCL or SLAM(localization), and move_base(for planning and navigation)

For mapping and localization we can either go with SLAM or AMCL. SLAM(simultaneous localization and mapping) is used when no map exists we can build one using SLAM. AMCL(adaptive monte carlo localization) to localize robot on map which is usually YAML + PGM files. AMCL uses a particular filter to guess the robots location by taking sensor data , comparing it to the map and refining the estimate of the robots position over time.

Now navigation stack is done using global + local planning, we use the local planner DWA at each timestep:

1. Samples multiple v,ω values
2. Uses Forward Kinematics to simulate trajectories
3. Evaluates cost: distance to path, obstacles, goal heading
4. Chooses best trajectory
5. Publishes best v,ω as /cmd_vel

Now we apply motion via inverse kinematics, by using gazebo plugin, the inverse kinematics is directly applied to move the wheels, if gazebo plugin (gazebo_ros_diff_drive) is not used we have to write a custom controller node that does it and sends the w to the motor drivers.

Now as the wheels rotate the robot moves, we use the forward kinematics to calculate how far it moved and a feedback is generated which closes the loop for localization and trajectory correction.

Each cycle:

- Global path → DWA selects trajectory → sends cmd_vel → wheels turn → position updates → repeat until goal
- [Start: User Sets 2D Nav Goal in RViz]→ Global Planner → DWA Local Planner → Simulate Trajectories (forward kinematics) → Best (v, ω) → Publish /cmd_vel→ [Controller or Plugin: Inverse Kinematics → Wheels] → Robot Moves → Forward Kinematics → /odom→ Localization (AMCL/SLAM updated)→Path adjusted by planner based on:robot pose, obstacles, goal →Robot reaches goal and stops.

DERIVATION AND CALCULATION

The position of robot (x,y,z), goal position (x_1,y_1) and the gains can be taken from the simulation. From odometry */odom* or AMCL */amcl_pose* or Gazebo *debug /gazebo/model_states* we can extract the x, y, θ. From Rviz 2D Nav Goal we will get the goal position, if using navigation stack, planner sets the internal goals which we are doing here. Gains can be controlled by us on the behaviour and turning.

Step 1: distance to goal

From robot pose (x,y) to goal (x_g,y_g):

$$\rho = \sqrt{(x_g - x)^2 + (y_g - y)^2}$$

This is the euclidean distance to calculate how far is the robot from the goal.

Step 2: heading error

We know the robots orientation is θ

Desired heading:

$$\theta_{goal} = arctan2(y_g - y, x_g - x)$$

Heading error :

$$\alpha = \theta_{goal} - \theta$$

This is how much the robot must turn to face the goal.

Step 3: angular velocity for correction

We use proportional control to rotate robot towards the goal

$$w = k_\alpha . \alpha$$

Where k_α is positive gain and ensures smooth turning

Step 4: linear velocity

We will choose v based on how well aligned the robot is to the goal

$$v = k_{\rho}.\rho$$

But we must scale it down when angular error is large it can be done using.

$$v = k_{\rho}.\rho\, cos\, \alpha$$

This ensures the robot moves forward only when its facing close to the goal

Note: velocity limits

We must clip the velocity at

$$v \in \left[0,\ v_{max}\right]$$

$$w \in \left[-\ w_{max},\ w_{max}\right]$$

ODOMETRY FROM WHEEL ENCODERS

Odometry is the method of estimating a robot's position and orientation over time using data from its wheel encoders. Wheel encoders are cheap as it uses existing wheel motors + encoders. Each motor has an encoder that tells how much the wheel has rotated. With lidar we still use odometry to provide initial guess between frames in SLAM, predict next position before matching map during localization (AMCL), continues working when liDAR/ GPS is lost and also used in control loops to stabilize motion by providing velocity feedback. Each wheel has an incremental rotary encoder that reports: Number of ticks (or counts) since last reading.

Convert encoder ticks to distance:

$$\Delta_s = \frac{2\pi R}{N}.\Delta_{ticks}$$

Where Δ_{ticks} = change in encoder ticks since last step

Linear displacement of robot (center):

$$\Delta s\ =\ \frac{\Delta s_R + \Delta s_L}{2}$$

Change in orientation :

$$\Delta\theta\ =\ \frac{\Delta s_R - \Delta s_L}{L}$$

Pose update equation:

Current pose is (x,y,θ)

New pose is:

$$x' = x + \Delta s.\ cos(\theta + \frac{\Delta\theta}{2})$$

$$y' = y + \Delta s.\ sin(\theta + \frac{\Delta\theta}{2})$$

$$\theta' = \theta + \Delta\theta$$

Odometry helps the robot estimate where it is after moving without it robot is like a blindfolded person walking with no memory of where they came from. So we can conclude that forward and inverse kinematics helps to compute or control robots motion before and while moving whereas odometry helps in tracking and estimating where the robot ends up after moving.

DIFFERENTIAL DRIVE DYNAMICS

Dynamics is used to find the forces and torque making the robot move.

Step 1: Newtons law for translation

Lets consider total force F causing linear acceleration,

$$F = M.\bar{v}$$

Where v is the linear velocity of robot center and $\bar{v}$ is the linear acceleration.

The force is created by wheel torques:

$$F = \frac{R}{2}(\frac{\tau_l}{R} + \frac{\tau_r}{R}) = \frac{\tau_l + \tau_r}{2R}$$

$$\bar{v} = \frac{\tau_l + \tau_r}{2MR}$$

Where M is the mass of robot

Step 2: newtons law for rotation

Total torque causes angular acceleration:

$$\tau = I_z . \overline{w}$$

The net torque from the wheels is:

$$\tau = \frac{L}{2R} (\tau_r - \tau_l)$$

$$\overline{w} = \frac{L}{2RI_z} (\tau_r - \tau_l)$$

Step 3: combining into full state dynamics

The full robot is defined by the state vector: x=

$$\begin{vmatrix} x \\ y \\ \theta \\ v \\ w \end{vmatrix}$$

Now we use the following differential equations,

$$\overline{x} = v cos\theta$$

$$\overline{y} = v sin\theta$$

$$\overline{\theta} = w$$

$$\overline{v} = \frac{\tau_l + \tau_r}{2MR}$$

$$\overline{w} = \frac{L}{2RI_z} (\tau_r - \tau_l)$$

Dynamics are required when we want the robot to move realistically, safely, and accurately especially in real-world or physics-based simulation (like Gazebo). If we are using a real robot than we need to convert desired velocities into torques using inverse dynamics.Gazebo uses a

physics engine so it needs mass, inertia, friction and torque to simulate motion. We also have to convert desired velocities into torques using inverse dynamics. (for eg what torque do I apply to the wheels, considering mass, friction, and inertia). So dynamics comes into picture after kinematics before odometry.

INVERSE DIFFERENTIAL DRIVE DYNAMICS

This is the opposite of forward dynamics, where we calculate acceleration from torque. It's critical for motor control, force-limited motion, simulation, and realistic physics in Gazebo. When we are given linear and angular motion , we will find the torque that must be applied to achieve this motion using inverse dynamics.

Step 1: robots dynamic equations

Translational motion(newtons second law):

$$F_{total} = M.\overline{v}$$

$$F_{total} = \frac{\tau_l + \tau_r}{R}$$

$$\frac{\tau_l + \tau_r}{R} = M.\overline{v}$$

Rotational motion (yaw dynamics)

$$\tau_{yaw} = I_z.\overline{w}$$

$$\tau_{yaw} = \frac{L}{2R}(\tau_r - \tau_l)$$

$$I_z.\overline{w} = \frac{L}{2R}(\tau_r - \tau_l)$$

Step 2: solving for τ_l and τ_r

$$2\tau_r = MR.\overline{v} + \frac{2RI_Z}{L}.\overline{w}$$

$$\tau_r = \frac{1}{2}MR.\overline{v} + \frac{RI_Z}{L}.\overline{w}$$

$$2\tau_r = MR.\overline{v} - \frac{2RI_z}{L}.\overline{w}$$

$$\tau_r = \frac{1}{2}MR.\overline{v} - \frac{RI_z}{L}.\overline{w}$$

This gives the torque required by each wheel motor to achieve the desired robot acceleration.

DWA CORE LOOP

1. Current state:
    - Robot's current pose (x,y,θ) from odometry
    - Current velocity (v_{curr},w_{curr})
2. Generate velocity samples:
    - Sample linear velocity v and angular velocity ω from a feasible set (dynamic window)
    - Respect limits:
        - Max/min linear & angular velocities
        - Max acceleration/deceleration
3. For each (v, ω) pair:
    - Simulate trajectory over a short time horizon (e.g., 1s or 2s)
    - Use forward kinematics to predict future poses
4. Score each trajectory based on cost functions:
    - Heading: How well the trajectory aligns with the goal direction
    - Clearance: Distance from obstacles

- Velocity: Preference for higher speeds

5. Choose the best scoring trajectory
    - Send corresponding (v, ω) as cmd_vel to the robot

TRAJECTORY SIMULATION IN DWA

Trajectory simulation is the process of predicting the future path of a robot over a short time window (e.g., 1–2 seconds), given a candidate control input typically linear and angular velocities (v,ω)

We stimulate robots motion over a short horizon using discrete integration

$$x_{t+\Delta t} = x_t + v.\ cos\ (\theta_t).\Delta t$$

$$y_{t+\Delta t} = y_t + v.\ sin(\theta_t).\Delta t$$

$$\theta_{t+\Delta t} = \theta_t + w.\Delta t$$

Where $\Delta t = simulation\ time\ step$

Repeat for N steps to get full trajectory

(x_t, y_t, θ_t) = robot pose at time t

Obstacle avoidance: for each point (x,y) in the trajectory check if it collides with any obstacle from the costmap if yes than discard this command if no then evaluate cost to goal, velocity and clearance and each simulated trajectory is evaluated using the multi objective cost function formula of DWA. the command with lowest cost will be executed.

After trajectory simulation DWA in ROS can visualize the generated trajectories, and we can see the curved lines each is a trajectory simulated from a candidate.

TYPICAL NAVIGATION STACK

The "Typical Control Stack" refers to the standard architecture used in ROS-based robots to convert high-level navigation goals into actual movement of the robot all in a modular, reusable,

and scalable way. It's the entire pipeline that takes your goal (like "go to (1, 1)") and makes the robot actually move there smoothly, safely, and intelligently while avoiding obstacles, correcting errors, and adapting to its environment.

Typical navigation stack creates a layered architecture by creating high level planning(navigation, goals), Mid-level control (velocities to wheel speeds), and Low-level hardware execution (motor drivers). This separation makes the system modular and easy to debug.

Typical navigation stack enables autonomous motion. With this you just configure your robot's kinematic + dynamic limits, plug in localization and sensors the stack will handle the rest.

We provide limits to DWA planner and controller the navigation stack uses them to plan, simulate, and move robot in Gazebo or real-world.

CONTROL THEORY IN DIFFERENTIAL DRIVE ROBOTS

Control theory provides the mathematical framework and tools for regulating the behavior of dynamic systems. In the context of a 2D differential drive robot, control theory is essential for ensuring the robot moves in a stable, accurate, and responsive manner toward its goal, while adapting to environmental changes and disturbances.

Control theory is essential for Trajectory Tracking: Follow a desired path (straight line, curve, or waypoint path), Pose Stabilization: Reach and stay at a specific position and orientation, Error Correction: Compensate for deviations due to slip, drift, or actuator delay and Robustness: Handle modeling inaccuracies or dynamic changes (e.g., terrai**n).**

OPEN LOOP & CLOSED LOOP CONTROL SYSTEMS

1. Open-loop Control: Robot receives a fixed velocity command (v, ω) without feedback. There is no requirement for correction for disturbances or sensor noise. Uses precomputed trajectories (e.g., splin-es, Bezier curves).No feedback = prone to drift due to unmodeled disturbances.

2. Closed-loop (Feedback) Control: continuously adjusts the robots motion based on current state (from odometry and localization) and errors are computed and used to modify wheel speeds to keep the robot on track.

FEEDBACK CONTROL

1. Pose Stabilization (Point-to-Point Control)

When navigating from current pose to a specific target pose. Typically used in small indoor robots.

Controller:

$$\rho = \sqrt{(x_t - x)^2 + (y_t - y)^2}$$

$$\alpha = -\theta + arctan2(y_t - y, x_t - x)$$

$$\beta = -\theta - \alpha + \theta_t$$

Control laws:

$$v = k_\rho \rho$$

$$w = k_\alpha \alpha + k_\beta \beta$$

Gain $k_\rho > 0$, $k_\beta < 0$, $k_\alpha > k_\rho$

Ensures asymptotic stability for goal pose. Control laws are send to motor controller → actuate → observe pose → loop continues.

2. Trajectory Tracking

Used when the robot must follow a continuous path, not just reach a goal. Used in outdoor navigation.

Defining the reference path to be $(x_r(t), y_r(t), \theta_r(t))$ *with velocities* $v_r(t), w_r(t)$

We use error variables in robots local frame:

$$e_x = cos\theta(x_r - x) + sin\theta(y_r - y)$$

$$e_y = -sin\theta(x_r - x) + cos\theta(y_r - y)$$

$$e_\theta = \theta_r - \theta$$

Feedback control:

$$v = v_r cos(e_\theta) + k_1 e_x$$

$$w = w_r + k_2 v_r + k_3 sin(e_\theta)$$

This non linear control law enables stable tracking of arbitrary paths

PROPORTIONAL CONTROL (P Controller)

This is the simplest and most common form used in differential drive robots.

Angular Control:

Usually before linear movement in pose control, or during obstacle avoidance. Used to compute ω for heading correction. to rotate robot towards the goal and it ensures smooth rotation without overshoot if properly tuned.

$$w = k_{\theta}.(\theta_{goal} - \theta_{current})$$

k_{θ} is the angular gain

θ *is the heading angle*

Linear Control:

After angular alignment, to move straight toward the goal. To move the robot forward

$$v = k_{v}.\ distance\ to\ goal$$

To ensure motion only when facing the goal

$$v = k_{v}.distance\ to\ goal.\ cos(\theta_{error})$$

This suppresses forward speed when orientation is off

NAVIGATION CONTROL STABILITY IN 2D DIFFERENTIAL DRIVE ROBOTS

Ensure the robot converges to a target pose or trajectory smoothly, without oscillations, overshoot, or divergence, under feedback control laws. This requires mathematical stability analysis often using Lyapunov theory, error dynamics, and proper controller design.

Stability: A control system is stable if the robot's pose error (difference between current and desired pose) tends to zero over time.

Types of stability:

- Lyapunov stability: Small disturbances → remain small.

- Asymptotic stability: Error → zero as t→∞
- Exponential stability: Error decays exponentially fast.

In navigation we aim for asymptotic or exponential stability

As we know the common control tasks are pose stabilization and trajectory tracking, now we will dive into the stability.

Stability of Pose Stabilization Controller

Defining lyapunov function

$$V = \frac{1}{2}\rho^2 + \frac{1}{2}\alpha^2 + \frac{1}{2}\beta^2$$

Derivative along robot trajectory

$$\overline{V} = \rho.\overline{\rho} + \alpha.\overline{\alpha} + \beta.\overline{\beta}$$

The system is asymptotically stable at

$$\rho = 0,\ \alpha = 0,\ \beta = 0$$

Stability of Trajectory Tracking Controller

Using non-linear control theory, define Lyapunov function:

$$V = \frac{1}{2}e_x^{\ 2} + \frac{1}{2}e_y^{\ 2} + \frac{1}{2}e_\theta^{\ 2}$$

We have to compute time derivative $\overline{V}$ and substituting in control laws, we will get $\overline{V}$ ≤0. All error terms converge to zero and the system becomes asymptotically stable and robot tracks trajectory, if gains are tuned well exponential convergence is possible.

Important factors that affect stability:

- Gaining tune: Poor gains → oscillation, overshoot, instability
- Control loop rate: Too slow → delays in feedback → instability
- Odometry drift: Leads to wrong error computation → poor tracking
- Nonholonomic constraints: Robot cannot move sideways → control must respect kinematic limits

Navigation control stability ensures a 2D differential drive robot safely and smoothly converges to its goal.Using Lyapunov-based analysis of control laws (pose or tracking), we guarantee stability under specific gain conditions. These analyses ensure robots don’t diverge, oscillate, or overshoot, forming the core of intelligent autonomous navigation.
