Planning :—

Perception → Planning → Control

Perception:
- Environment
- Robot State Info.

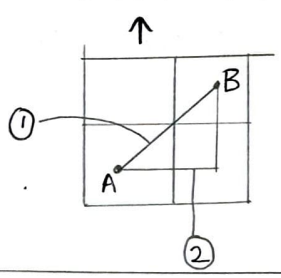
Planning:
- Motion Planning
- Collison Avoidance (Reaction Avoidance)

Environment → Occupancy Features
- Mapping — Modelling the environment
- Localization — Process of finding/estimating Robot's position.

Maps:
- Feature Based Maps → Localization
- Occupancy Maps (Navigate) → Localization → (LIDAR)
  - Occupancy Grid Map
  - Cost Grid Map
  - SDF
  - ESDF

① Euclidian Distance :— Represents shortest distance between two points.

② Manhattan Distance :— Sum of absotute difference between two points across all dimensions.



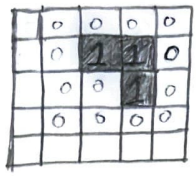
* Costs according to Euclidian or Manhattan distance (Grid) are assigned and then various algorithms are applied for finding path. (A*, Djikstra etc)

Cost-Based Grid Path Planner (G→S)

```
int count = 0;
while (count != 0)
    { count = 0;
      for (i=1; i < GRID_SIZE -1; i++)
          {for (j=1; j < GRID_SIZE -1; j++)
              {Count += count + cell_cost(i,j);
              }
          }
    }
```

Grid Based Navigation.

Occupancy Grid Map — The basic idea of the occupancy grid is to represent a Map of the environment as an evenly spaced field of binary random variables each representing the the presence of an obstacle at that location in the environment. (presence of obstacle — (1), absence — (0))

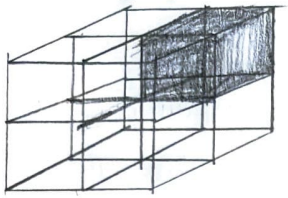


Limitation of Occupancy Mapping :—

① Every Grid cell is one of two state :- filled or empty. No info. about partially filled cells, if needed.

② Dealing with semi-transparent obstacles become tough, as laser range find may hit and return about half the time.

In 3-D Octomap can be used instead of Grid.



Possible Sol" :- ▷ More Continuous measuring technique.

▷ Using Random Variable.

Cost Grid Map — A Cost grid map represents the cost of traversing each cell. The values of the grid can be Real Numbers, and they typically represent the cost of traversing across that cell, distance to nearest obstacle, slope of terrain, or some factor that affects robot's ability to move through the environment.

Methodology :— From the starting position move along the direction of the most (−) ve gradient and reach the next along the direction. Repeat the above step till goal is reached.

▷ Construction of Cost - Grid Map :—

→ Each cell in Cost-grid denotes the distance from that location to goal.

→ Initialize all cells to very high costs, except goal cell. Obstacle cells to (∞) or something higher than the cell costs.

→ A list "open" stores the cells whose values have been lowered from their initialized values. The list is sorted with lowest values at the top, that one popped. (Initially list contains only goal node)

→ The top most node in the list is popped and expanded to its neighbours, whose values are in turn lowered and inserted to the sorted lists at their respective places.

The process continues till all cells have been expanded once

(*) From Program POV
(*) Goal to Start

→ How cell Costs are lowered :—

The lower cost is computed by looking at neighboring cells and using an estimate of travel cost from the adjacent cells to current cell. For laterally adjacent cells the estimate is 1 and for diagonally adjacent cells its $\sqrt{1^2+1^2} = \sqrt{2} = 1.414$

This estimate is added to the previously computed cost at the adjacent cells.

The minimum of such costs becomes the cost at current cell.

Cell_Cost() returns 0 if there is no change in cost, 1 otherwise

→ Robot then looks at the region in Cost-Grid Map corresponding to where it is in real world. Then it choosed direction along the shortest path to goal. The Gradient is downhill in Cost-Grid. X & Y are computed separately and then atan2() func is used to convert into direction betⁿ 0 and 2π. If obstacle is there it takes direction towards lowest cost neighbour instead of Grad, i.e. ∇f

▷ Similarly we can start from 'Start' and proceed towards Goal.

6.8 a_1	5.8 a_2	4.8 a_3	3.8 a_4	3.4 a_5	3 a_6	3.4 a_7
6.4 b_1	5.4 b_2	4.4 b_3	3.4 b_4	2.4 b_5	2 b_6	2.4 b_7
6.8 c_1	5.8 c_2	c_3	c_4	c_5	1 c_6	1.4 c_7
7.2 d_1	6.8 d_2	7.2 G d_3	d_4	d_5	S d_6	d_7
6.8 e_1	5.8 e_2	e_3	e_4	e_5	1 e_6	1.4 e_7
6.4 f_1	5.4 f_2	4.4 f_3	3.4 f_4	2.4 f_5	2 f_6	2.4 f_7

(*) Start to Goal

▷ Initialize all cells to very high cost
▷ Obstacle cells to ∞ or something higher than the cell costs.
▷ Make Cost of S → 0
(d_6) → 0
▷ Find Cost (Nbhrs(S)) →

$c_6 = e_6 = d_7 = 1$
$c_7 = e_7 = 1.4$

→ ▷ Choose the Nbhr with least cost from S (d_6) (now) → (c_6)

Expand from c_6 or find cost of (Nbhrs (c_6)) from S

→ $b_6 = 2$ $b_7 = 2.4$

→ ▷ Choose Node amongst all expanded nodes, with the minimum cost from S (now) → (e_6)

Expand (e_6) → f_6, f_7, f_5

Cost (Nbhrs(e_6)) → $f_6 = 2$ $f_7 = 2.4$, $f_5 = 2.4$

→ ▷ Choose Node amongst all expanded nodes, with minimum cost from S (now) → (d_7) → No Node to expand.

→ Continue to expand till goal is reached or all nodes in the cells are expanded. (Either of two way is fine)

Signed Distance Field (SDF)

(*) Not in Curriculum.

$f \rightarrow [0, 1, -1]$ (*) For Discrete Field

$f_{inside}(x,y) \rightarrow -1$

$f_{on}(x,y) \rightarrow 0$

$f_{out}(x,y) \rightarrow 1$

Euclidian Signed Distance Field (ESDF)

(*) Gradient value is used (*) For Continuous field

(*) Distance from closest surface is used here.

Other points :— (*) Grid's resolution refers to how large an area in real world is represented by one cell in the grid.

(*) For bigger robots we inflate the occupancy by that unit in the map.

(*) For much detail, a grid can be made with higher resolution, but the major disadvantage would be memory consumption as more computations need to be done at a time,

(*) Low resolution may result in less optimal path & jerky robot motion,

(*) Occupancy Grids are easy to update if the robot discovers a discrepancy / obstacle.

Example :— If robot discovers an obstacle in the North (+Y direction) the occupancy grid is updated like this,

occupancy [robot_x] [robot_y+1] = FULL

If the previously considered occupied space turns out to be free, then. (in the east)

occupancy [robot_x+1] [robot_y] = EMPTY.

▷ This will require a replanning step.

(*) Resolution should depend on Robot's sensor's accuracy, how fast will it move and how much memory is available

(*) Note :- If cost estimates are multiplied by resolution of grid, the values at each cell would reflect the true distance of goal.

(*) Robot using a Grid-based planner follows this cycle :-

Optimisation :-

Need :- If grid has N cell each side and we plan over a complicated map, it might cycle through the grid N^2 time and make N^4 cell evaluation. This may be very heavy fast computationally & resource wise.

(*) Why start with Goal / Start = 0

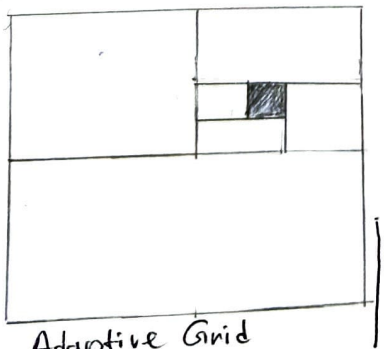
→ As all cells are set to high cost initially, There is no way to lower the cost if the adjacent cells are also at high cost.

A cell should not be evaluated until after one of its neighbours has been lowered, (otherwise we are wasting time in that cycle)

Initially the only cells with a low cost neighbours are the cells next to goal, so they are evaluated first.

→ Other Ways are :-

① For Mapping Dividing only the part where Obstruction is present.

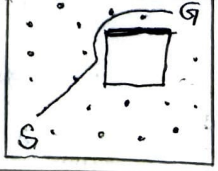


Resolution of Grid is more where obstacle is present and low in open spaces.

② Sampling is another technique used in grid-based navigations system to improve efficiency.

Instead of dividing entire space into a grid, sampling focuses on selecting a subset of points or nodes statistically placed in environment.

This reduces no. of nodes to be considered for computation.



Discrete Planning ..

▷ Each action, u, when applied from Current State, x, produces a new state, x', as specified by a state transition function, f.

$$x' = f(x, u)$$

Let $U(x)$ denote the action space for each state x, which represents the set of all actions that could be applied from x.

for distinct x, $x' \in X$, $U(x)$ and $U(x')$ are not necessarily disjoint
The same action may be applicable in multiple states.

$$U = \bigcup_{x \in X} U(x)$$

As a part of planning problem, a set $X_G \subset X$ of goal states is defined.

(*) The task of planning algorithm is to find a finite sequence of actions that when applied, transforms the initial state x_1 to some state x_G

Discrete Feasible Planning

1) A non-empty state-space X, which is a finite or countably infinite set of states.
2) For each states $x \in X$, a finite action space $U(x)$.
3) A state transition function f that produces a state $f(x,u) \in X$ for every $x \in X$ and $u \in U(x)$. The state transition equation is derived from f as $x' = f(x,u)$.
4) An initial state $x_1 \in X$,
5) A goal set $X_G \subset X$

▷ Planning problem is usually specified without explicitly representing the entire state transition graph. Instead it is revealed incrementally in the planning process.

▷ For a problem in which X is infinite, the input length must be finite.

▷ From a planning perspective, it is assumed that the planning algorithm has a complete specification of the machine transitions & is able to read its current state at any time.

General Forward Search → (Starts from "Start")

Three kind of States:—

- ▷ **Unvisited :—** States that have not been visited yet.
- ▷ **Dead :—** States that have been visited and for which every possible next state has also been visited.
- ▷ **Alive :—** States that have been encountered but possibly have unvisited next states.
  The set alive is stored in priority queue, Q, for which a priority function must be specified.

```
Q. Insert(x_I) and mark x_I as visited
While Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    for all u ∈ U(x)
        x' ← f(x,u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return failure
```

Q → follows FIFO (assume)

→ While loop terminates when Q is empty, which occurs only when entire graph has been explored.

→ reports SUCCESS when x is in X_G

→ otherwise, algo tries every possible action, $u \in U(x)$

→ for next state $x' = f(x,u)$ it must determine whether x' is being encountered for the first time.

→ If unvisited, then it is inserted into Q. else no need to consider because it either must be dead or already in Q.

→ Failure occurs when entire graph has been explored and X_G (goal state) is not found.

* In each While iteration, the highest ranked element, x, of Q is removed.

* This algo only finds if the solution (or goal) exists and not the path. This problem can be solved by inserting a segment which associates x' with its parent x at ◬. If this is done one can easily trace the pointers from final to initial state to recover the path.

* In some cases the concept of cost can also be used for every cell instead of pointers to set the cell priority and also to get the path.

Breadth First. ▷ Search Frontier Grows Uniformly

▷ In this search algorithm, All plans that have K steps are exhausted before plans with K+1 steps are investigated.

▷ It guarantees that first solution/path found will use the least no. of steps.

▷ Run time → $O(|V|+|E|)$ $|V| \rightarrow$ no. of vertices $|E| \rightarrow$ no. of edges.

Depth First.

▷ By making Q a stack (LIFO), aggressive exploration of state transition graph occurs.

▷ Preference is given towards investigating longer plans very early.

▷ Running time is $O(|V|+|E|)$

Dijkstra's Algorithm.

▷ Concept of "Cost" is used here. — Every Edge, $e \in E$ in the graph representation of a discrete planning problem has an associated non-negative cost $l(e)$, which is cost to apply the action.

The Cost $l(e)$ can be written using state-space representation as $l(x,u)$, indicating action $l(x,u)$ cost is from state x to apply.

The total cost of a plan is just the sum of the edge costs over the path from the initial state to goal state.

▷ Priority Queue, Q, will be sorted according to function $C: X \rightarrow [0, \infty]$ called Cost-to-Come.

$C^*(x) \rightarrow$ Optimal Cost-to-Come for each state x. (from initial state x_1)

▷ The optimal Cost is obtained by summing edge costs, $l(e)$ over all possible paths from x_1 to x and using the path that produces least Cumulative Cost.

If cost is not Optimal then it is written as $C(x)$.

▷ Cost-to-Come is computed incrementally during execution of search algo.

Initially $C^*(x_1) = 0$, Each time x' is generated a cost is computed as

$C(x') = C^*(x) + l(e)$, in which e is the edge from x to x'

$\approx C(x') = C^*(x) + l(e,u)$

If x' is already present in Q, then it is possible that new path discovered, is more efficient,

If so, then cost-to come value $C(x')$ must be lowered for x', and Q must be recorded accordingly.

Once x is removed from Q (popping out), the state becomes dead, & it is known that x cannot be reached with a lower cost.

▷ Running time is $O(|V| \lg |V| + |E|)$ in which $|E|$ and $|V|$ are edges & vertices respectively in the graph representation of Discrete planning path.

(A*) A - Star Algorithm :-

▷ Extension of Djkistra Algorithm, that tries to reduce total no. of states to explored by incorporating a heuristic estimate of cost to get to the goal from given state.

▷ It uses heuristics to stop certain unnecessary nodes from being searched. This is done by weighting the cost values of each node distance by their Euclidian distance from Desired end point. (→ distance (x to goal))

Therefore, only paths that are headed generally in the correct direction will be evaluated.

▷ A* algo works exactly like Djkistra, the only diff. is the function used to sort Q, In A* the sum $C^*(x') + G^*(x')$ is used implying that the priority queue is sorted by estimates of the optimal cost from x_1 to X_G

Best first :- The priority queue is sorted according to an estimate of the optimal cost-to-go. The sol"s obtained in this way are not necessarily optimal.

Iterative deepening :- Usually preferable if the search tree has a large branching factor. The basic idea is to use depth-first search and find all states that are distance i or less from x_I. If goal is not found the prev. work is discarded and depth first is applied to find all states of distance i+1 or less from x_I. The no. of states reached for i+1 is expected to far exceed the number reached for i.

Its worst case performance is better than Breadth first search. Space requirements are also reduced.

General Backward Search :-

Backward versions of any of the forward Search algorithms can be made. In this case the algorithms start exploring from and find their path to "start"

Some planning problem may have the problem of large Branching factor when started from x_I, in those case Backward search may be more beneficial.

Q. Insert (x_G) and mark x_G as visited.
While Q not empty do
    $x' \leftarrow$ Q.GetFirst()
    if $x = x'_I$
        return SUCCESS
    for all $u^{-1} \in U^{-1}(x)$
        $x \leftarrow f^{-1}(x', u^{-1})$
        if x not visited
            Mark x as visited
            Q.Insert (x)
        else
            Resolve Duplicate x
return FAILURE.

Bidirectional Search

Bidirectional Search can dramatically reduce the amount of required exploration that are required in one-way based searches.

There are Djkistra and A* variants of Bidirectional Search which lead to optimal solutions.

For best-first and other variants it may be challenging to ensure that the two trees meet quickly, they may come very close and then fail. Connecting the trees become even more complicated and expensive.

Code next →

Q_I.Insert(x_I) and mark x_I as visited
Q_G.Insert(x_G) and mark x_G as visited
While Q_I not empty and Q_G not empty do

```
    if Q_I not empty
        x ← Q_I.GetFirst()
        if x already visited from x_G
            return SUCCESS
        for all u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q_I.Insert(x')
            else
                Resolve duplicate x'
    if Q_G not empty
        x' ← Q_G.GetFirst()
        if x' already visited from x_I
            return SUCCESS
        for all u^-1 ∈ U^-1(x')
            x ← f^-1(x', u^-1)
            if x is not visited
                Mark x as visited
                Q_G.Insert(x)
            else
                Resolve duplicate x
return FAILURE.
```

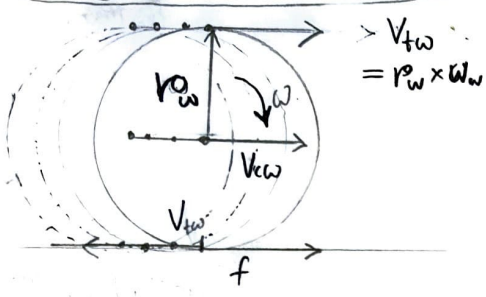
FORWARD KINEMATICS.

DEALS WITH:— (If the motors of a robot rotate with commanded angular velocities where would the robot reach in given time interval)

Kinematics:- Effect of Robot's Geometry on its motion. (Assumes we control encoder readings)

Dynamics:- The effect of all forces (internal & external) on a robot's motion. (Assume we control motor current)

▷ Rolling Without Slipping:-



Consider a wheel that rotates freely about the centre in air.

When this wheel is placed on the ground aided by the force of friction, f, begins to move forward.

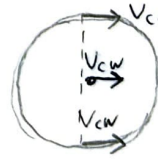
In other words, in absence of such a force the wheel would rotate but its velocities at the top and bottom are of opposite direction and of equal magnitude, that it would not move.

V_{tw} at top $\longrightarrow$
V_{tw} at bottom $\longleftarrow$
} thus cancelling each other, preventing any translation.

Although at point of contact, wheels tendency is to move towards $\longleftarrow$ f opposes this tendency by acting in $\longrightarrow$, this causes the wheel to move in $\longrightarrow$ direction.

Let $\overrightarrow{V_{cw}}$ be translational velocity (Rate of change of linear position) of wheel centre (same for all parts of wheel)

✱ Note:- V_{cw} is the effect of frictional force, f.

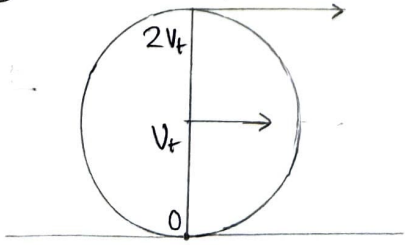


Total Velocity at top $= \overrightarrow{V_{cw}} + \overrightarrow{V_{tw}}$

Velocity at point of contact $= \overrightarrow{V_{cw}} - \overrightarrow{V_{tw}}$

Rolling without slipping or pure rolling entails $\therefore V_{cw} - V_{tw} = 0$

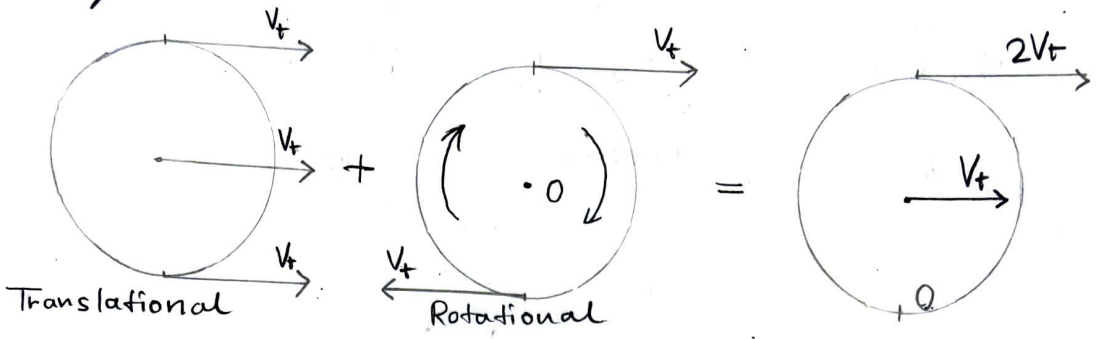
$\Rightarrow \underline{V_{cw} = V_{tw}} = r_w \omega_w = r\omega$ (represents wheel)



At point of contact with the ground the wheel is instantaneously at rest.

Note:- The translational velocity of the wheel is V_{cw} is the same at all points on the wheel. However, the velocity due to rotation of the wheel centre, V_t is different at different locations of wheel.

Other way to Understand is:—



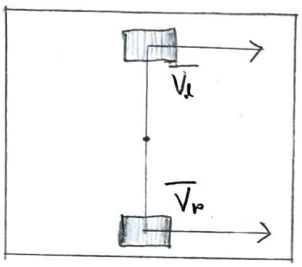
$$V_{axle} = \frac{\Delta x}{\Delta t} = \frac{2\pi r}{T} = r\left(\frac{2\pi}{T}\right) = r\omega = V_t$$

for one cycle

Differential Drive :—

Differential Drive kinematics :— Kinematics involved with Left and Right wheels being independently controlled.

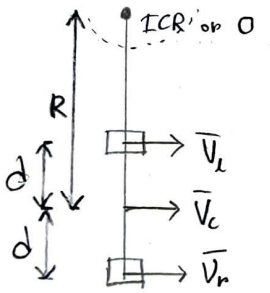
(*) Difference in wheels speeds determines the turning angle of Robot.



Let linear velocity of Left wheel Centre (translatational velocity) be $\overline{V_l}$
and of Right wheel Centre be $\overline{V_r}$

The entire robot rotates about an Instantaneous centre of rotation, ICR that lies on the line joining the left and right wheel (the wheel base). Also known as ICC ⟶ Curvature

▷ To minimize wheel slippage, this point must lie at the intersection of the wheel's axis. Each wheel must be travelling at the same angular velocity around ICC.



Let the angular velocity at which the robot rotates around O be ω

Then $\overline{V_c} = \overline{\omega} \times \overline{r}$ or $|V_c| = R\omega$ — ①

$\overline{V_l} = (R-d)\omega$ — ②

$\overline{V_r} = (R+d)\omega$ — ③

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = f\left(V_l, V_r\right)$$

$V_p = f(V\omega)$ (Platform Velocity)

$$R = \frac{(V_r + V_l)}{(V_r - V_l)}\, d$$

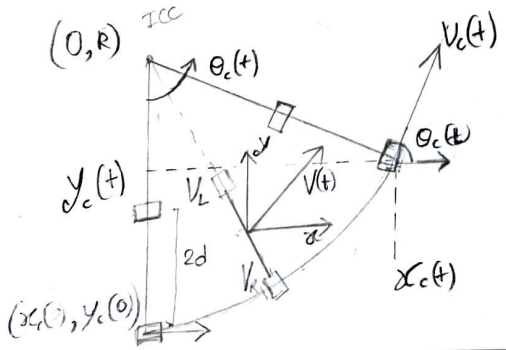
$\Rightarrow V_c = \frac{V_l + V_r}{2} = \omega R$ — ④

③ − ② $\Rightarrow \overline{V_r} - \overline{V_l} = (R+d)\omega - (R-d)\omega$

$\Rightarrow V_r - V_l = 2d\omega \Rightarrow \omega = \frac{V_r - V_l}{2d}$ — ⑤

$V_x = V(t)\cos\theta(t)$

$V_y = V(t)\sin\theta(t)$



$$x_c(t) = \int_0^t V_c(t)\cos(\theta_c(t))\,dt$$

$$= \int_0^t V_c \cos(\theta_c)\,dt \quad — ⑥$$

$$y_c(t) = \int_0^t V_c \sin(\theta_c)\,dt \quad — ⑦$$

$$\theta_c(t) = \int_0^t \omega\,dt \quad — ⑧$$

$\theta_c(t) = \omega t$ ——⑨

Then, $x_c(t) = \int_0^t V_c \cos(\omega t)\, dt = \frac{V_c \sin(\omega t)}{\omega}\Big|_0^t = \frac{V_c}{\omega}\sin(\omega t)$ → ⑨

$y_c(t) = \int_0^t V_c \sin(\omega t)\, dt = -\frac{V_c \cos(\omega t)}{\omega}\Big|_0^t = -\frac{V_c}{\omega}[\cos(\omega t) - 1]$ → ⑩

From ⑨ and ⑩ we get,

$$x_c^2 + \left(y_c(t) - \frac{V_c}{\omega}\right)^2 = \frac{V_c^2}{\omega^2}\left(\cos^2\omega t + \sin^2\omega t\right) = R^2$$

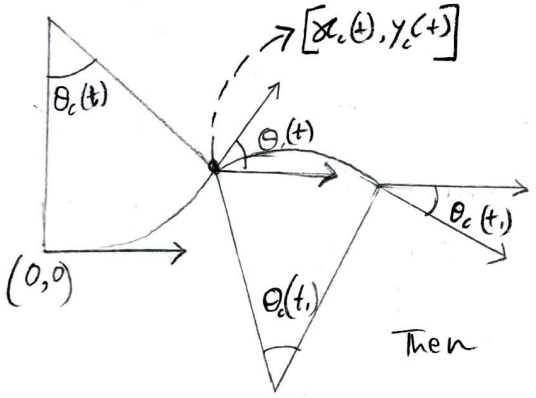
$$\Rightarrow \boxed{x_c^2(t) + (y_c(t) - R)^2 = R^2}$$

✱ Given the wheel's velocities or positions, how to find what is robot's velocity/position?

1) Specify System Measurements.
2) Determine the point (the radius) around which robot is turning.
3) Determine the speed at which the robot is turning to obtain the robot velocity.
4) Integrate them to find position.

Aggregation of Multiple Such Segments :-

Here we use Co-ordinate Transform.



Let $t=0$ translate the spatial and temporal origin to $(x_c(t), y_c(t), t)$

$t' = t_1 - t$

Then $x_c(t') = \frac{V_c'}{\omega'}\sin(\omega t')$ — ⑪

$y_c(t') = \frac{V_c'}{\omega'}[1 - \cos(\omega t')]$ — ⑫

where, V_c', ω' is the linear and angular velocity from $t \to t'$

▷ Current position of robot wrt to original starting position.

$$\begin{bmatrix} x_c(t_1) \\ y_c(t_1) \end{bmatrix} = \begin{bmatrix} \cos\theta_c & -\sin\theta_c \\ \sin\theta_c & \cos\theta_c \end{bmatrix} \begin{bmatrix} x_c(t') \\ y_c(t') \end{bmatrix} + \begin{bmatrix} x_c(t) \\ y_c(t) \end{bmatrix} \quad -⑬$$

$$\theta_c(t_1) = \theta_c(t) + \theta_c(t') \quad ——⑭$$

▷ Kinematics in the presence of acceleration.

$$x_c(t) = x_c(0) + \int_0^t (V_c + at)\cos\left(\theta_c(0) + \omega t + \frac{\alpha t^2}{2}\right) dt \quad \text{—— (15)}$$

(V_c: linear Acceleration; $\frac{\alpha t^2}{2}$: Angular acceleration.)

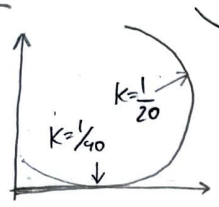
$$y_c(t) = y_c(0) + \int_0^t (V_c + at)\sin\left(\theta_c(0) + \omega t + \frac{\alpha t^2}{2}\right) dt \quad \text{—— (16)}$$

Angular Displacement :— $\theta = \omega t + \frac{1}{2}\alpha t^2$

The above integrals need to be numerically computed by the method of Fresnel Integrals.

The resulting curve obtained from numerical integration is called

CLOTHOID (Also known as Euler Spiral or Cornu Spiral)



→ It is a curve whose curvature "K" changes linearly with its curve length

$$K = \frac{d\theta}{ds} = \frac{d\theta}{dt} \cdot \frac{dt}{ds}$$

(ds: arc length)

Fresnel Integrals :—

$$x(t) = \int_0^t \cos\left(\frac{\pi}{2}u^2\right) du = C(t)$$

$$y(t) = \int_0^t \sin\left(\frac{\pi}{2}u^2\right) du = S(t)$$

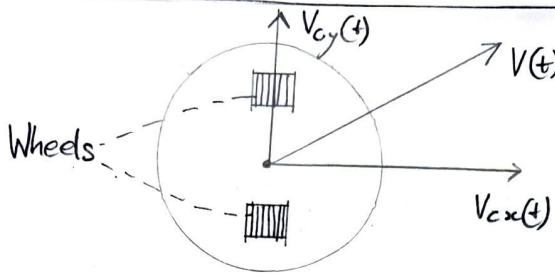
Not Important

Holonomic Robot :— A holonomic robot is a type of mobile robot that can move in any direction and rotate freely without having to change its orientation.

Holonomic robots achieve this level of maneuverability through use of special mechanisms such as Omnidirectional wheels, holonomic drives, or other systems that allow for motion in multiple directions without constraint.

Holonomicity implies that the robot's motion is not restricted to predefined paths or motions but can be freely controlled in any direction.

Holonomic Kinematics:—

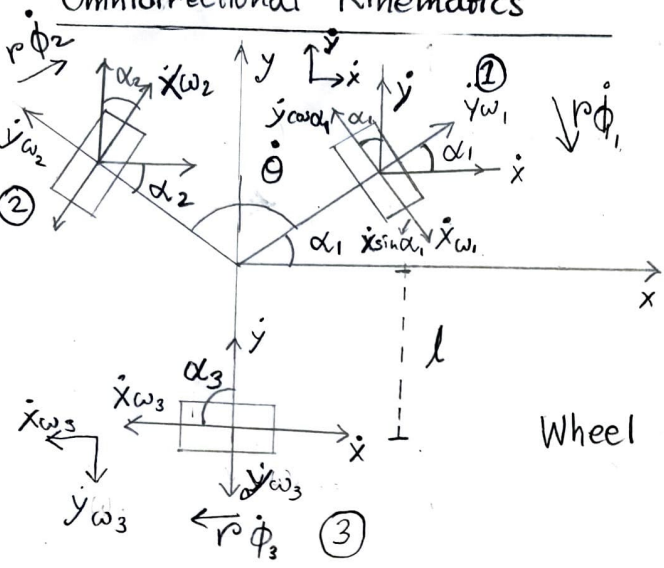


Independent Control of V_x, V_y

$$x(t) = x(0) + \int_0^t V_x \, dt \quad \text{—— (17)}$$

$$y(t) = y(0) + \int_0^t V_y \, dt \quad \text{—— (18)}$$

Omnidirectional Kinematics



$\dot{x} = [\dot{x}\ \dot{y}\ \dot{\theta}]$ is the body center velocity

$\dot{x}_{\omega i}, \dot{y}_{\omega i}$ is the wheel velocity of the i^{th} wheel.

$\dot{\phi}$ is the angular velocity of the i^{th} wheel.

Wheel 1 :→ $\dot{x}\sin\alpha_1 - \dot{y}\cos\alpha_1 + l\dot{\theta} = r\dot{\phi}_1$ —— ①

$\dot{x}\cos\alpha_1 + \dot{y}\sin\alpha_1 = 0$ —— ②

(No lateral Side Constraint)

Wheel 2 :→ $\dot{x}\sin\alpha_2 + \dot{y}\cos\alpha_2 + l\dot{\theta} = r\dot{\phi}_2$ —— ③

$\dot{x}\cos\alpha_2 - \dot{y}\sin\alpha_2 = 0$ —— ④

(No lateral Side Constraint)

Wheel 3 :→ $\dot{x}\cos(90+\alpha_3) + \dot{y}\cos\alpha_3 + l\dot{\theta} = r\dot{\phi}_3$ —— ⑤

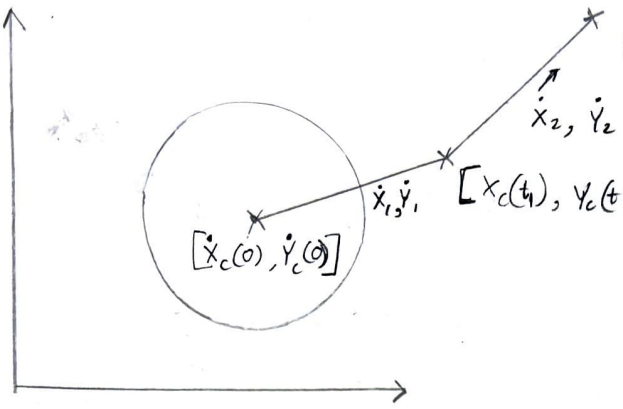
$\dot{x}\sin(90+\alpha_3) + \dot{y}\sin\alpha_3 = 0$ —— ⑥

Taking together 1, 3, 5 we get:—

$$\begin{bmatrix} \sin\alpha_1 & -\cos\alpha_1 & l \\ \sin\alpha_2 & \cos\alpha_2 & l \\ \cos(90+\alpha_3) & \cos\alpha_3 & l \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} r\dot{\phi}_1 \\ r\dot{\phi}_2 \\ r\dot{\phi}_3 \end{bmatrix} \quad \text{—— ⑦}$$

or $J\dot{X}_p = \dot{X}_\omega$ —— ⑧

$\dot{X}_p = [\dot{x}\ \dot{y}\ \dot{\theta}]^T \quad \dot{X}_\omega = [r\dot{\phi}_1\ \ r\dot{\phi}_2\ \ r\dot{\phi}_3]^T$



$$\dot{X}_1, \dot{Y}_1 \xrightarrow{a_x, a_y} \dot{X}_2, \dot{Y}_2$$

$$\dot{X}_1(t) = \dot{X}_1(t_0) + a_x(t - t_0)$$ until you reach $\dot{X}_2$

$$\dot{Y}_1(t) = \dot{Y}_1(t_0) + a_y(t - t_0)$$ until you reach $\dot{Y}_2$

$$X_1(t) = \int_{t_0}^{t} \dot{X}_1(t)\,dt = X_1(t_0) + \dot{X}_1(t - t_0) + \frac{a_x}{2}(t - t_0)^2$$

$$Y_1(t) = \int_{t_0}^{t} \dot{Y}_1(t)\,dt = Y_1(t_0) + Y_1(t - t_0) + \frac{a_y}{2}(t - t_0)^2$$

Non-Holonomic Robot

A non-holonomic robot is a robot that has limitations in its movement capabilities.

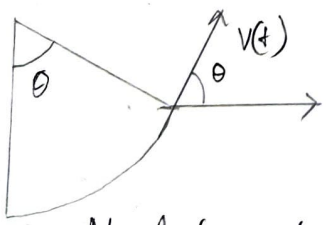
The robots have fewer controllable directions of movement compared to their total degrees of freedom. The constraints come from factors like wheel configuration, drive mechanisms or physical limitations. Due to the constraints, planning and controlling the movement of these robots can be more challenging.

Non-holonomic Kinematic :-

The Differential Drive robot is non-Holonomic. [Refer to Prev. Derivations]

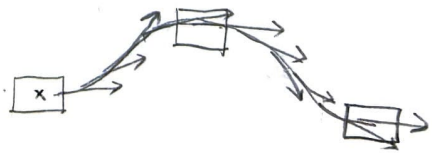
$$\left.\begin{aligned} \dot{x}_c(t) &= V\cos(\omega t) \\ \dot{y}_c(t) &= V\sin(\omega t) \end{aligned}\right\} \quad \boxed{\dot{y}_c(t) = \dot{x}_c(t)\tan\theta}$$

V_y and V_x are not Decoupled. They are coupled through the robots instantaneous direction/heading θ.



The robots tangential velocity $v(t)$ is always along the heading direction $\theta(t)$.

▶ No Independent Control of V_x and V_y.



Control degrees of freedom (v, ω).

Configuration / Cartesian degrees of freedom (x, y, θ)

The degree of Controls is less than the degree of configuration, accessible by robot.

Omnidirectional Wheels :—

Also known as Mecanum wheels or Ilon wheels, are type of wheels that allows a mobile robot to move in any direction, including diagonally, without the need to turn.

This is achieved by having a series of small rollers mounted around the circumference of the wheel, which are set at an angle to the direction of travel.

Advantages :—

① Increase maneuverability → Move any direction.

② Improved precision → Move very precisely without much need to change direction.

③ Less skidding/sliding while commiting movement on ground, comparative to traditional wheels.

Disadvantages :—

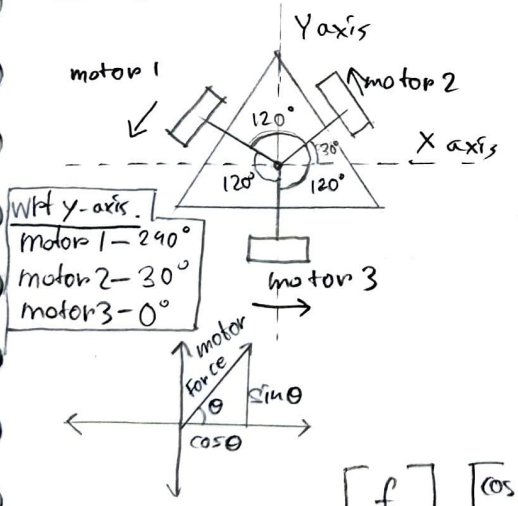
① More complex to design, manufacture and operate.

② Increased cost.

③ Omnidirectional wheels may have less traction than traditional wheels, which can create problem in slippery surface.

Usage :— (Consider a 3-wheeled robot)

F → Force



Wrt y-axis.
motor 1 – 240°
motor 2 – 30°
motor 3 – 0°

$$\sin\theta = \frac{F_{y\text{-axis}}}{F_{motor}}$$

$$\Rightarrow F_{motor} \times \sin\theta = F_{y\,axis}$$

$$\cos\theta = \frac{F_{x\text{-axis}}}{F_{motor}}$$

$$\Rightarrow F_{motor} \times \cos\theta = F_{x\,axis}$$

$$f_x = \cos(240^\circ) F_{motor1} + \cos(120^\circ) F_{motor2} + \cos(0^\circ) F_{motor3}$$

$$f_y = \sin(240^\circ) F_{motor1} + \sin(120^\circ) F_{motor2} + \sin(0^\circ) F_{motor3}$$

$$f_w = 1 \times F_{motor1} + 1 \times F_{motor2} + 1 \times F_{motor3}$$

$$\begin{bmatrix} f_x \\ f_y \\ f_w \end{bmatrix} = \begin{bmatrix} \cos 240^\circ & \cos 120^\circ & \cos 0^\circ \\ \sin 240^\circ & \sin 120^\circ & \sin 0^\circ \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} F_{motor1} \\ F_{motor2} \\ F_{motor3} \end{bmatrix}$$

F_{cord} = A × F_{motor}

→ Find A^{-1} (Let $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$)

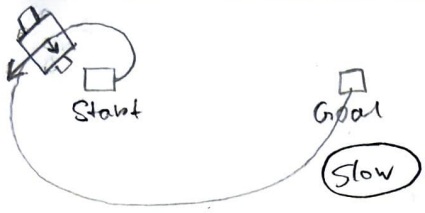
$$\Rightarrow [A^{-1}][F_{cord}] = [F_{motor}]$$

$$\therefore \begin{bmatrix} af_x & bf_y & cf_w \\ df_x & ef_y & ff_w \\ gf_x & hf_y & if_w \end{bmatrix} = \begin{bmatrix} F_{motor1} \\ F_{motor2} \\ F_{motor3} \end{bmatrix}$$

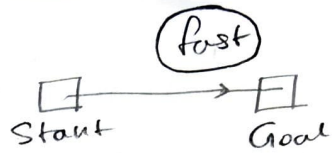
$$A^{-1} = \frac{1}{\det A} \text{Adj } A$$

$$\text{Adj } A = \begin{bmatrix} +M_{11} & -M_{21} & +M_{31} \\ -M_{12} & +M_{22} & -M_{32} \\ +M_{13} & -M_{23} & +M_{33} \end{bmatrix}$$

Positional Control of Differential Drive Robot :—



▶ Gain of Orientation Controller is small



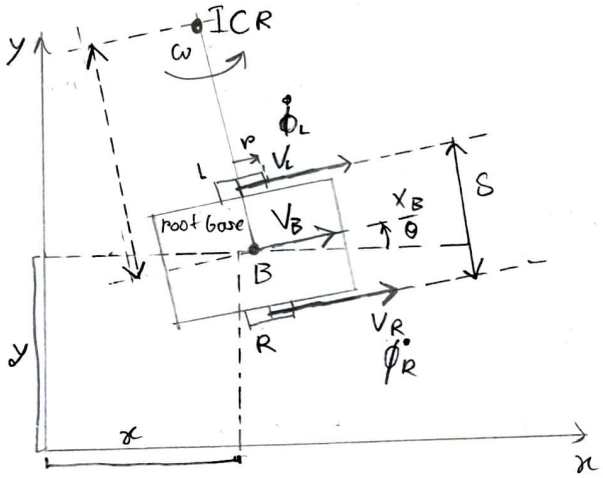
▶ Gain of orientation controller is increased

Orientation Controller — Controls Robots orientation θ

Velocity Intensity Controller — Controls intensity of robots velocity vector of Robot's centre of mass

Velocity Vector is tangent to the trajectory, so by controlling velocity vector we can control the direction of motion,

The controller calculate the angular velocity of the left and right wheel from the velocity vector, to achieve the desired motion.



ω → Instantaneous angular velocity of Robot body,

V_L, V_R → Velocity of Centre of wheels

$\dot{\phi}_L$, $\dot{\phi}_R$ → Angular velocity of wheels,

V_B → velocity of point B

We control Robot's motion by controlling $\dot{\phi}_L$ & $\dot{\phi}_R$

$\dot{x}$ → projection of velocity V_B on x-axis

$\dot{y}$ → projection of velocity V_B on y-axis

$$\dot{x} = \frac{r\dot{\phi}_L}{2}\cos(\theta) + \frac{r\dot{\phi}_R}{2}\cos(\theta)$$

$$\dot{y} = \frac{r\dot{\phi}_L}{2}\sin(\theta) + \frac{r\dot{\phi}_R}{2}\sin(\theta)$$

$$\dot{\theta} = -\frac{r}{S}\dot{\phi}_L + \frac{r}{S}\dot{\phi}_R$$

} Integrate them as a function of time to get the position and orientation of Robot

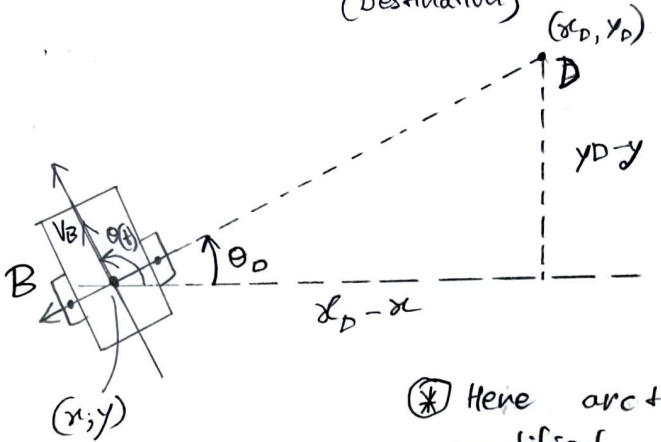
For implementation of Control Algorithm, we need another equation relating V_B & θ with $\dot{\phi}_L$ and $\dot{\phi}_R$

$$\begin{bmatrix} V_B \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} r/2 & r/2 \\ -r/S & r/S \end{bmatrix} \begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix}$$

Control Problem Formulation :-

Let x_D & y_D be the destination coordinates.

We need to determine a sequence of wheel angular velocity control variables $\dot{\phi}_L$ & $\dot{\phi}_R$ such that current robot position $(x(t), y(t))$ is equal to the desired robot position (x_D, y_D) (Destination)



Start ⟶ Destination.

$$\tan\theta_D = \frac{y_D - y}{x_D - x}$$

$$\theta_D = \arctan 2\left(\frac{y_D - y}{x_D - x}\right)$$

(*) Here arctan 2() or atan 2() function is the modified arctan() or 2-argument arc tangent function, that takes into account the signs of arguments when computing angles.

Orientation controller,

$$\dot{\theta} = K_\theta\left(\theta_D - \theta(t)\right) \longrightarrow \text{changes } \theta.$$

Here we use, Proportional Controller ↑

$\theta(t)$ ↓ Current value of orientation of robot.

K_θ ↓ Proportional control gain for orientation control.

∴ When $\theta_D = \theta(t)$, then according to equation $\dot{\theta} = 0$, we have achieved the desired robot orientation.

Velocity Controller,

$$V_B = K_V\sqrt{(x_D - x(t))^2 + (y_D - y(t))^2}$$

→ Changes intensity of Velocity

This controller penalizes the Euclidian distance between point B and D

Longer the distance longer is the velocity

As we reach closer to D, the velocity gradually decrease.

Finally we need to control the angular velocities of the wheels $(\dot{\phi}_L$ & $\dot{\phi}_R)$ by transforming $\dot{\theta}$ & V_B to $\dot{\phi}_L$ & $\dot{\phi}_R$

PTO

We can do that by inverting the system of equations.

$$\begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix} = \begin{bmatrix} r/2 & r/2 \\ -r/s & r/s \end{bmatrix}^{-1} \begin{bmatrix} V_B \\ \dot{\theta} \end{bmatrix}$$

↳ Invert the parameter matrix to obtain $\dot{\phi}_L$ & $\dot{\phi}_R$

Summary :— Process

① Observe $x(t)$, $y(t)$ & $\phi(t)$

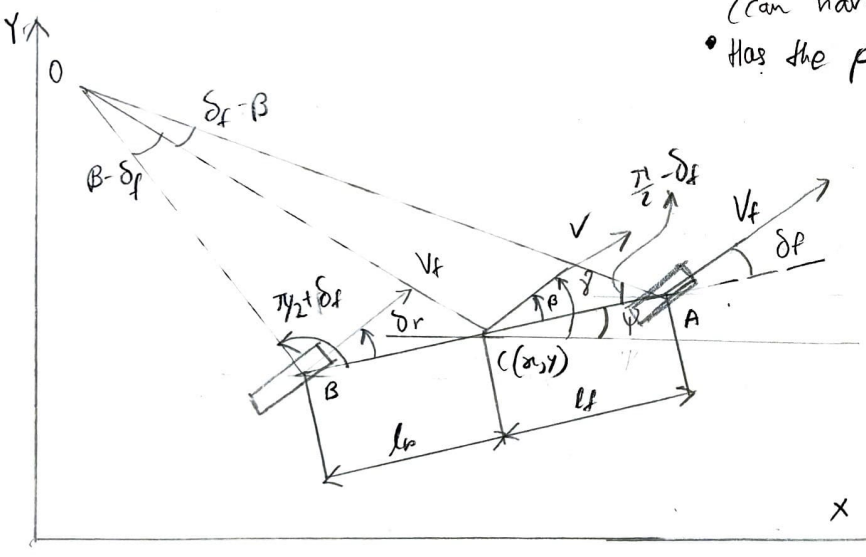
② Calculate $\dot{\theta}$ & V_B

③ Compute $\begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix}$

④ Apply $\dot{\phi}_L$ & $\dot{\phi}_R$ to the left and right motor controllers that will generate the torques and move the wheels of the differential drive robot.

FR SteerBot at iRL.

- Has both front and rear wheel steer.
- Gives rise to smaller turning radius, (can navigate in tighter spaces).
- Has the property of parallel steer.



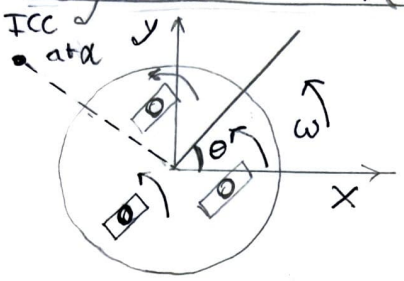
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} V\cos(\psi+\beta) \\ V\sin(\psi+\beta) \\ \frac{V\cos\beta(\tan\delta_f - \tan\delta_r)}{l_f + l_r} \end{bmatrix}$$

$$\frac{l_f + l_r}{\cos\beta(\tan\delta_f - \tan\delta_r)}$$

The radius of the car is given by $\frac{l_f + l_r}{\cos\beta(\tan\delta_f - \tan\delta_r)}$ centered at 0

▷ The radius is the distance from 0 to C. The parallel steer condition occurs when $\delta_f = \delta_r$ resulting in an infinite turn radius. This implies that the car translates without changing orientation.

Synchro Drive. (Holonomic)



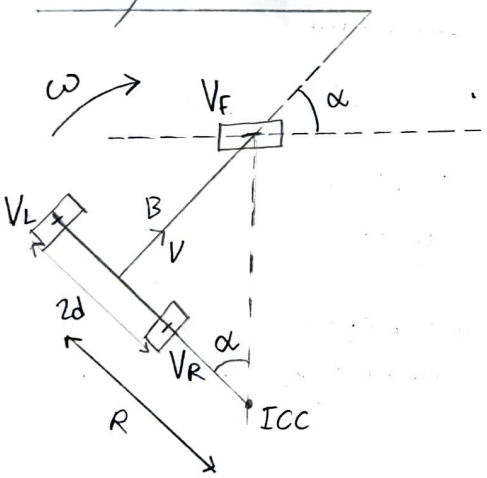
$$V_{robot} = V_{wheels}$$
$$\omega_{robot} = \omega_{wheels}$$
} Velocity

$$\theta(t) = \int \omega(t)\, dt$$
$$x(t) = \int V_{wheels}(t)\cos(\theta(t))\, dt$$
$$y(t) = \int V_{wheels}(t)\sin(\theta(t))\, dt$$
} Position

Synchro drive is a specific type of Omnidirectional Drive System used in robots and mobile platforms.

- Wheels rotate in tandem and remain parallel all
- All wheels are driven at the same speed.
- One motor drives all wheels at same speed and another one controls the orientation of all wheels simultaneously, achieving turning and diagonal movements.

Tricycle Drive.



$$V_F = \frac{\omega B}{\sin\alpha}$$
$$R = \frac{B}{\tan\alpha}$$

$$V = \omega R = V_F \cos(\alpha)$$
$$V_L = \omega(R+d) = V + d V_F \sin(\alpha)/B$$
$$V_R = \omega(R-d) = V - d V_F \sin(\alpha)/B$$

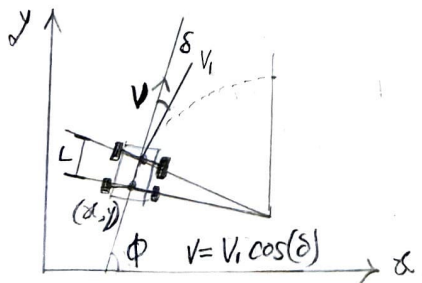
* It is a tricycle configuration in which front wheel is steered and driven.

Kinematics of Car.

$$\dot{x} = v\cos\phi$$
$$\dot{y} = v\sin\phi$$
$$\dot{\phi} = \frac{v}{L}\tan\delta$$

Integrating these equations one will find the car moves in a circle of radius $L\cot(\delta)$

When $\delta = 0$, it represents infinite turning radius or car being moving in a straight line.



Note:- Since the linear velocity appears in the time evolution of all the state variables, this car model does not have the possibility to make arbitrary rotations while standing still in Real plane. It is only able to follow paths that are least cont. differentiable

Solution :—

: Minimum Reversal Methods :

1) Let Initial state/configuration of robot be X_0. Add this to tree T.
2) Generate discrete successors for current state X (a leaf node of T) by varying the steering angle δ in the kinematic equation of car.
Each delta gives a successor node, call it X_n.
3) Among the generated successors X_n (all the leaf nodes of T so far) choose the node with minimum number of reversals and insert it into a heap (if it is collision free). The step is repeated till a collision free successor node is found.
4) Now X = top most node in the heap.
5) If X is the goal state or near the goal state STOP else repeat 2 to 4 till goal is reached or heap is empty.

RRT — Rapidly Exploring Random Tree.

RRT is a powerful algorithm used in motion planning, particularly for robots and other systems navigating complex environments.
It works by building a tree-like structure in the space where the system can move. This tree gradually expands by exploring new areas randomly while respecting any obstacles or constraints present in the environment.

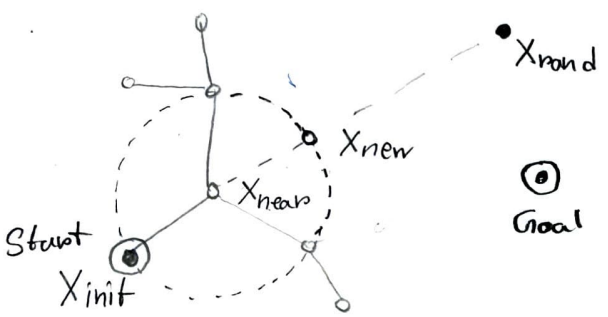
Basic Idea :-

1) Start with a single :— This represents the initial configuration of the system.
2) Sample a random point :— This point could be anywhere in the allowed space.
3) Find the nearest point in the existing tree :— This is the 'parent' of the new point.
4) Connect the new point to the parent :— This creates a new edge in the tree, but only if the connection is feasible (doesn't hit obstacles and follow constraints)
5) Repeat step 2-4 :— This process continues iteratively, expanding the tree and exploring new areas.

△ Suitable for Rapidly Exploring Space.
△ Inefficient for one-time queries.

RRT — Explore

```
1  T = {Src}
2  for k=1 to K //  (K number of random nodes in Workspace W)
                    or max number of iterations.
3      X_K = Random - State()    // one such random node.
4      X_n = Nearest - Neighbour(T, X) // finds the node among the
                                          leaf nodes that is closest to X_K
5      X_s = SuccessorState(X_n, X, U) // generate a successor to X_n,
                                          X_s, that takes X_s closer to
                                          X_K on some metric.
6      if (collision - free(X_s))
7      then  T = T ∪ {X_s}
```



Disadvantages of RRT:—

1) RRT bypasses need of local planners. Hence it is always difficult to reach final state exactly or even nearly exactly through a discrete search.

2) The RRT uses a metric by which it decides the node closest to random state. However many a time deciding the metric is equivalent to solving the motion planning problem as such. In other words there is no suitable or appropriate metric often to decide closeness between two states.

Bidirectional RRT:—

It works by simultaneously growing two trees: one from the starting configuration and another from goal configuration. At each step, both trees are extended towards each other until they meet, forming a path between start and goal.

▶ Disadvantage:— ① Complexity:— More complex than unidirectional approach. Coordinating the growth of trees from both start and goal configurations requires careful synchronization.

② Difficulty in maintaining Connectivity:— Ensuring connectivity between the two trees, throughout the exploration process can be challenging. It may require additional algo or heuristics to maintain continuity

③ Potential for Dead lock:— The two trees from start & goal configuration may be unable to connect due to obstacles or sampling limitations.

④ Computationally expensive with limited applicability.

▶ Advantages:— ① Efficient for one-time queries.

② Multiple Rendezvous points.

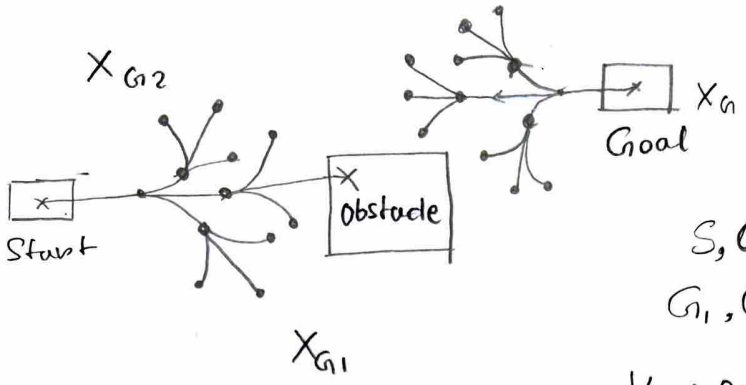
③ Faster Search ④ Reduces risk of getting stuck.

RRT - Create (Time T, State src, State dest)

```
1    T1 = {Src}; T2 = {dest}
2    while (true)
3        X = Random - State();
4        if (Extend (T1, X, Xn)! = TRAPPED)
5        then if (Extend (T2, Xn, Xn') == Reached)
6            then break;
7        swap (T1, T2);
```

Extend (Tree T, State Xn, State & Xs)

```
1    Xn = Nearest - Neighbour (T, Xn).
2    Xs = Successor State (Xn, X, U)
3    if (collision - free (Xs))
4    then T = T ∪ {Xs}
5    else return TRAPPED;
6    if (Xs is close - enough to Xn)
7    then return REACHED;
8    else return ADVANCED;
```



$S, G \rightarrow$ Start and Goal Configuration.

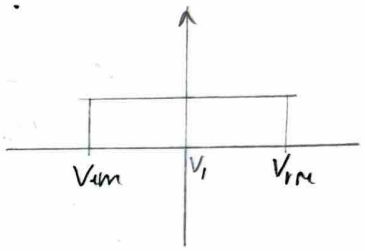
$G_1, G_2 \ldots \rightarrow$ Random Goals

$V \rightarrow$ set of feasible controls, we can discretize and represent it as

$$U = [[V_{11}, V_{12}], \ldots, [V_{1n}, V_{1m}]]$$

where each V_i is s.t.

$$V_m \leq V_i \leq V_M$$

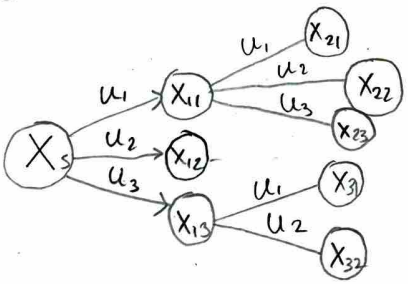


One can also think if this set as a distribution such as

$$S \rightarrow \begin{bmatrix} x_s \\ y_s \\ \theta_s \end{bmatrix} \quad G \rightarrow \begin{bmatrix} x_G \\ y_G \end{bmatrix}$$

$X_{11} = f(X_s, u, dt) \longrightarrow$ differential drive kinematics.

RRT $\longrightarrow$



MOTION PLANNING & CONFIGURATION SPACE.

▷ Motion planning problem in presence of moving objects is inherently harder than stationary case.

▷ The problem of finding a path for a point robot in a plane with bounded velocity in presence of moving objects that are convex polygons moving with linear velocities without rotation the problem is classified as NP Hard.

▷ For stationary objects and robots with arbitrary degrees of freedom the notion of probabilistic completeness is used for ascertaining complexity.

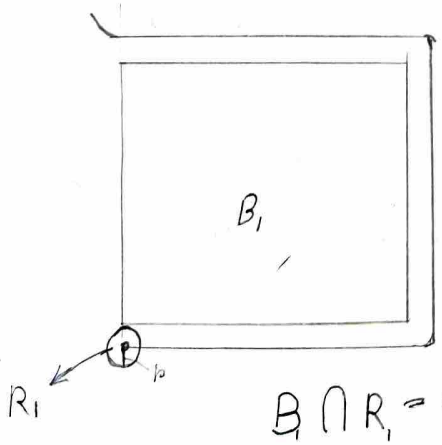
▷ For stationary objects and a robot with two degrees of freedom the motion planning problem can be solved in polynomial time generally.

Configuration Space.

▷ Main idea:— Grow the obstacles by Robot's size. and Reduce the robot to a point size.

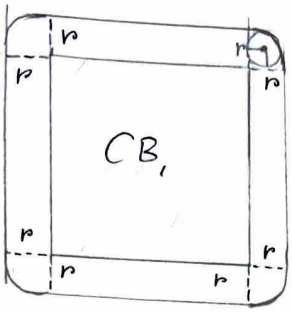
▷ This method of 2-D planning assume a set of 2-D convex polygon obstacles and a 2-D convex polygon robot.

▷ The point will always be a safe distance away from each obstacle due to the growing step of each obstacle.
Once we shrink the robot to a point, we can find a safe path for the robot using a graph search technique.



▷ The planning is done for a point on the robot but it makes sure if a path is collision free for this point then it is collision free for the whole robot.

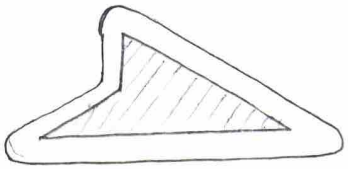
▷ The no. of degrees of freedom of robot is the dimension of the Configuration Space.



→ Configuration Obstacle.

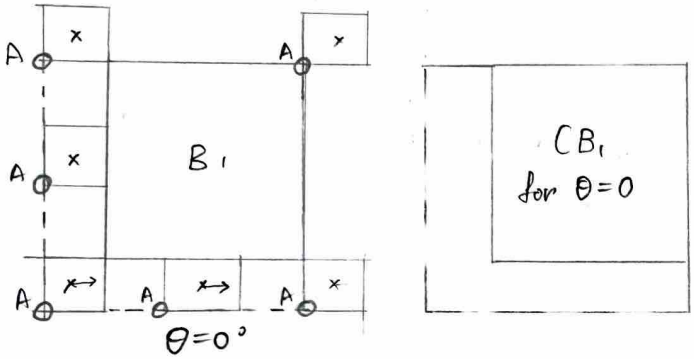
CB_1 :— The set of all points for which $B_1 \cap R_1 = \phi$

— Configuration Rotation space of the obstacle B_1



▷ **How to Obtain CB :—** Traces the locus of all points for which the robot R_1 just grazes the obstacle B, The locus is always traced with respect to a fixed point in R_1. In this case it is traced with respect to the centre of R_1.

▷ **How to obtain CB_1 for non Circular robot :—**

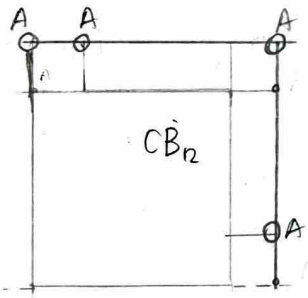


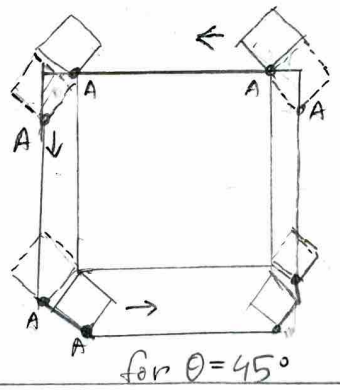
B_1 is the original obstacle

Dashed line traces the locus of point A for which R_1 graced B_1

CB_1 is the grown obstacle.

(*) When we change θ (orientation of the robot) the CB_1 will also change accordingly for each θ.

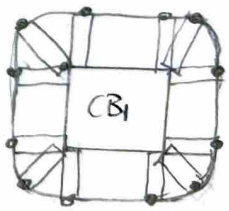




▷ **How is CB actually coded ?**

— There are libraries that compute CB for a 2-D planar robot that can be considered a circle or a polygon.

▷ **Configuration Space when R_1 is able to rotate.**



▷ **Dimension of C-space :-**

▷ **What is the dimension in which CB resides ?**

— The set of all points of A for which R grazes B_1

— For different orientation of R_1 we get different CB $(\theta(R_1))$ where $\theta(R_1)$ represents orientation of R_1 wrt a global reference frame.

— (*) The Dimension of CB is the same as the number of degrees of freedom of R.

Problems with Configuration Space Approach:—

→ Configuration spaces can become prohibitively large as the dimensionality of system increases. Also depends on DOF of robot.

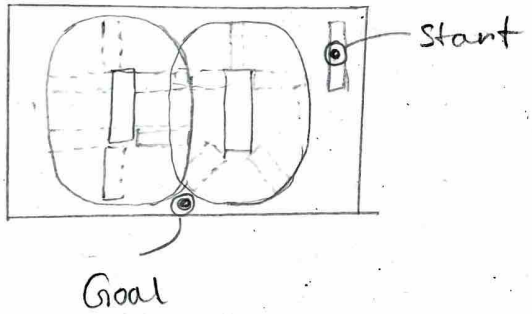
→ Config. Spaces are Continuous, which can make precise planning and computation challenging.
Algos must discretize the space or use approximation techniques to make planning feasible.

→ Presence of non-holonomic constraints may make the system more complex and more constrained, thus complicating motion planning.

→ If Dynamic Environment is present, the Config-Space will need to be updated dynamically in response to changing environment, which will be computationally expensive.

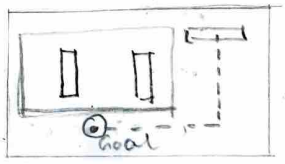
→ If multiple robots are present in the system, the C-space will become even complex.

→ Sometimes. Navigating to goal through cluttered space may become very complex with conventional C-space approach.

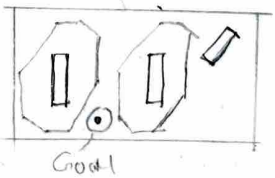


▷ Changing orientation only at certain positions of space may give a solution.
Ex — Consider Moving piano in a cluttered room.

▷ Changing Robot orientation smartly may help in such cases.



▷ Some orientations may also provide more complex results.



→ Kinematic planning in Configuration space does not consider Robot's dynamics. The path planned may turn out infeasible dynamically.

→ No optimality guaranteed as path planning with Cspace approach does not consider path length, smoothness or energy used.

Visibility Graph.

The visibility graph is an undirected graph $G = (V, E)$ where the V is the set of vertices of the grown obstacles plus the start and goal points, and E is a set of edges consisting of all polygon obstacle boundary edges, or an edge between any 2 vertices in the V that lies entirely in free space except for its endpoints.

▷ It is a computational tool used in robot motion planning to find optimal collison free paths by reducing the continuous configuration space to a discrete graph representation.

▷ Visibility graph encodes free space around obstacle by connecting visible nodes through edges.

▷ Once constructed, the shortest path on the visibility graph between Start and Goal corresponds to optimal collison-free path in the configuration space.

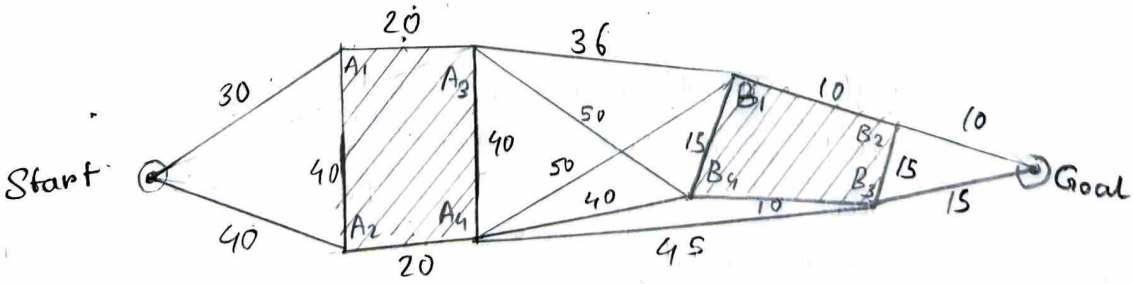
▷ How to draw Visibility Graph?

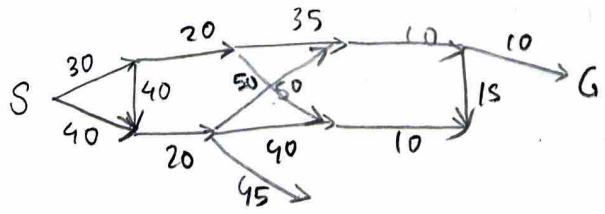
→ Connect all vertices of the CB's that are visible to one another.
→ Connect the Start and Goal vertices to the vertices of CB that are visible from Start and Goal.

Resulting Graph is visibility Graph.

→ Let the weight of the edges be the Euclidian distance between the two visible nodes (visible to one another).

→ The shortest path lies on the Graph and can be obtained by Dijkistra's algorithm.





∴ $S \rightarrow A_1 \rightarrow A_3 \rightarrow B_1 \rightarrow B_2 \rightarrow G$

* Similarly we can do this for 3-D also.

Algorithm for Visibility Graph:—

$G = (V, E)$

↑ undirected Graph.

Brute force Method:—

Assume all N vertices of G are connected.
This forms $N(N-1)/2$ edges.
Check each edge to see if it intersects (excepting its end points) any others edge in the graph.
If yes reject the graph.
The remaining edges are edges of visibility graph.

It is Brute force, slow but simple to compute.

$O(N^3)$

- Shortest path in distance can be found by searching Graph G using a shortest path search (like Djkistra) or other heuristic search method.

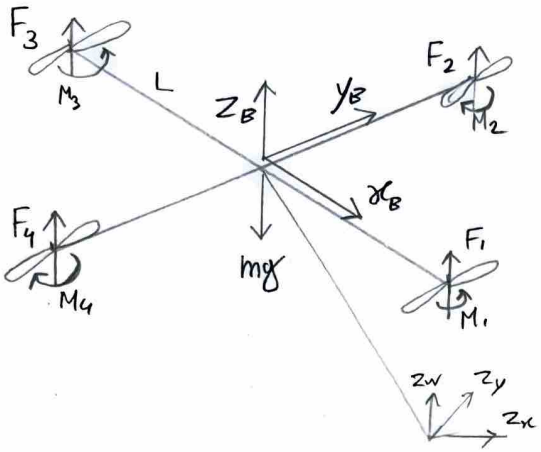
⊙ If initial and final locations of the robot are vertices on the G, then shortest path is contained on the visibility graph.
In other words if a computation between S and T involves the visibility graph then it needs to be the shortest post as long as S and T are connected to the vertices of G by straight line segments.

S — Start G — Goal

Which means ⟶ If Start and Goal lie on the vertices of the obstacles in the environment, then the shortest path between S and T on the visibility Graph is Guaranted to be the globally shortest Collison free path in the original Configuration System.

Motion Planning for Quadrotors (Drone — Dynamics & Kinematics)

Sensors → Navigation → Motion Planning → Control → Actuators



Cross Entropy Method.

▷ Two Constraints of motion planning:—
→ Kinematics/Dynamics → Obstacle Constraints

▷ Problem with Gradient based Optimization Method:—

→ Gradient based optimization approaches like stochastic gradient descent are not suitable as they need a starting guess and the obstacle avoidance constraints may be non-smooth.
Without a good starting guess the algorithm might get stuck in a local minima.

→ Discontinuities in the config-space from obstacles can cause issues with gradient calculation & convergence.

→ Computing will become costly with higher dimensions.

→ Vehicles with non-holonomic constraints have restricted motion so gradients do not apply.

→ Discontinuous configurations changes like flipping can't be represented through incremental gradients.

(*) An alternative approach is to discretize the vehicle state space and generate a path of transitioning between adjacent cells.

▷ Problem with Graph Based Method:—

→ If state space is very large than the approach becomes computationally expensive.

→ Can't add kinematic constraints in these kind of approaches.

→ Poor real time performance and high node expansion during search.

→ Incorporating uncertainty and sensor noise into graphs is challenging.

→ Constraints like joint limits are difficult to encode within graph structure.

→ Graph search methods can get trapped in local minima due to greedy heuristics.

▷ Problem with Sampling based motion Planning :—

→ No optimality. Does not Guarantee shortest or smoothest path.

→ Important areas of space can be missed due to sparse sampling. Hard to sufficiently sample narrow Corridors.

→ Constraints like dynamics & joint limits are hard to incorporate, Post processing of paths is needed.

→ Full queries are required, Hard to partially update prior data.

→ Sampling becomes more complex (grows exponentially) as we increase the dimension making scaling difficult.

→ In areas densely populated with obstacles, sampling progress can stagnate as valid samples become rare.

→ Storing large graphs with samples and connections has considerable memory overhead and can be computationally expensive for processing.

Solution :— (Join Best of both worlds)

▷ CROSS ENTROPY BASED MOTION PLANNING :—

→ CEM is a general Monte Carlo approach for optimization and importance sampling.

STEPS :— 1) Initialize a Gaussian Distribution over the robots config. space to sample paths.

2) Sample — Draw N random path configurations from the distribution as initial candidates.

3) Evaluate — Check Collisions and compute cost metrics like path length, smoothness, cost etc, for each sampled path.

4) Select — Select top P% lowest cost path samples and designate as "elites". P is usually 10% - 20%.

5) Update — Update distribution parameters to maximize likelihood of elites, Recalculate mean and covariances.

6) **Smooth** — Apply smoothing to the updated mean path to remove unnecessary oscillations.

7) **Repeat** — Resample new candidate paths from updated distribution. Go back to step 3.

8) **Terminate** — End iterations when cost converges or max iterations reached.

9) **Extract** — Extract the final mean path as optimized collision-free output.

10) **Shortcut** — Perform shortcutting on the path to remove any redundant waypoints.

11) **Return** — Return smoothed and shortcut path.

Algorithm:—

① Sample Controls from a Gaussian Distribution:—

Let say we are planning 100 ($N=100$) trajectories for 10 timesteps into the future ($H=10$) and our action space is of 2 dimension consisting of velocity and angular velocity.

→ `Controls = torch.randn(H, N, a_dim)`

- H → Horizon
- N → no. of samples
- a_dim → Dimension of space,

The sample controls are from a normal distribution with 0 mean & 1 SD.

`Controls = a_mu + a_std * controls.`

- a_mu → vector of dimension [10,1,2] and desired mean
- a_std → vector of dimension [10,1,2] and desired Standard Deviation.

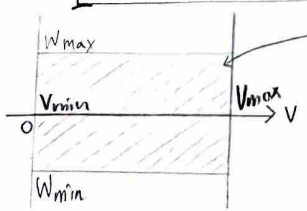
② Clip the controls within the control bounds of the agent:—

The sampled controls may have values that are beyond the max. and min. velocities and angular velocities of ego-vehicle.

→ `u_min = torch.tensor([v_min, w_min])`

→ `u_max = torch.tensor([v_max, w_max])`

→ `clipped_controls = torch.min(torch.max(controls, u_min), u_max)`



clip controls within this region.

③ Rollout the trajectories

Using a simplified kinematics model of vehicle we now pass the sampled controls through our kinematics model to get our trajectories.

→ theta_t = theta_{t-1} + w * dt

→ x_t = x_{t-1} + v * cos(theta_t) * dt

→ y_t = y_{t-1} + v * sin(theta_t) * dt

We will have a tensor named traj of dimension [H, N, 3] since each trajectory has x, y & theta.

④ Score the Trajectories.

Score our N trajectories using some cost functions.

- Goal reaching Cost
- Smoothness Cost
- Obstacle Avoidance cost.

For each trajectory the total cost is :—

Goal_reaching cost + Smoothness Cost + Obstacle avoidance Cost

⑤ Select Elite Trajectories.

Select the top "K" trajectories with smallest cost.

These are the elite trajectories. K <<< N (K is 10% of N)

We can also put various weights to various cost parameters by multiplying them with a no. and then adding them.

⑥ Set the mean and Covariance of the Gaussian Distribution as the mean and covariance of Elite trajectories.

Find mean — torch.mean } for elite trajectories
SD — torch.std

Set the mean & std of the sampling distribution as mean and std of elite trajectories.

⑦ Go to step 1

Repeat the above steps until K-L divergence between the previous distributions and the current distribution is less than a small constant or if the covariances have nearly shrunk to 0.

▷ **Goal Reaching Cost** :— Trajectory whose endpoint is closest to goal position gets smallest cost.

```
goal_state = [goal_x, goal_y, goal_theta]
for i in range(N):
    goal_cost[i] = sqrt(
        (traj[9, i, x] - goal_x) ** 2
        (traj[9, i, y] - goal_y) ** 2
        (traj[9, i, theta] - goal_theta) * 2
        )
```

▷ **Smoothness Cost** :— Trajectory where change in angular velocity and velocity is minimum is considered smoothest.

```
for i in range(N):
    smooth_v[i] = torch.norm(controls[:, i, 0])
    smooth_angv_v[i] = torch.norm(controls[:, i, 1])
```

▷ **Obstacle Avoidance Cost** :— If a trajectory lies inside obstacle, assign a very high cost to trajectory.

```
obs = [obs_x, obs_y]
agent_radius = 1
obstacle_radius = 1

for i in range(N):
    for j in range(H):
        d = sqrt((traj[j, i, 0] - obs[0]) ** 2 + (traj[j, i, 1]
                                                  - obs[1]) ** 2)
        if (d < agent_radius + obstacle_radius):
            obs_cost[i] += 5000
```

Initialization of Parameters:

→ We compute the optimal trajectory that reaches the goal ignoring the obstacles and kinematics/dynamics constraints.

→ The initial parameters for the Gaussian distribution is centered around this optimal trajectory.

→ We can use RRT to find optimal trajectory to goal.

→ Use the controls of this trajectory to obtain the mean and the covariance.

→ We can also generate trajectories using Bernstein polynomials. The second derivative of the Bernstein trajectories gives us the velocities which was can use as the initial guess.

Once we get a trajectory using CEM method, we can use that as a guess value for next call to the CEM function for faster consumption.

⊛ One issue with this approach is that if the environment has very narrow passages, the feasible region in the trajectory space will shrink. As a result a large no. of the samples will be rejected and as a result the resultant trajectory may not be close to the optimal trajectory.