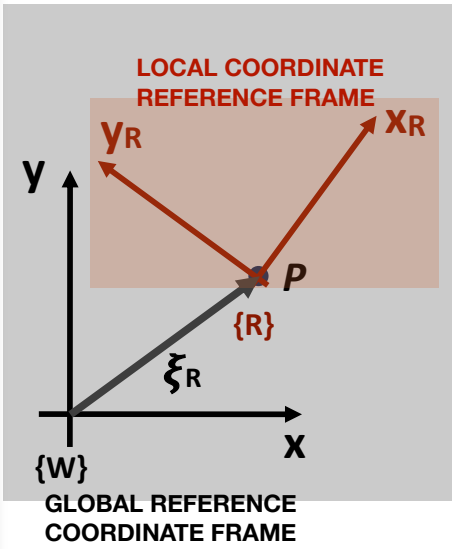
Robotics - Lecture 2
23/02/2024

What is the pose of a Robot

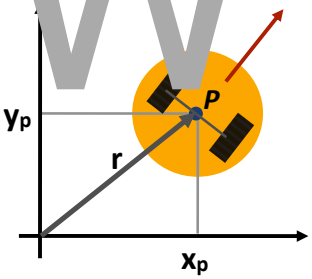
Before going straight to the slide content, I think is important to define the following concepts:

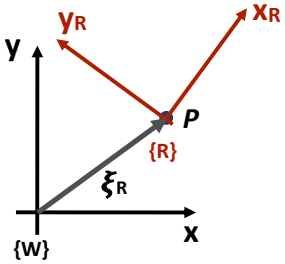
- **Coordinate Frame/Reference Frame:** *a set of axes with an origin that defines the spatial orientation and position of objects in n-dimensional space.* Typically consists of three perpendicular axes: X, Y, and Z, which define a three-dimensional space. The point where these axes intersect is known as the origin of the coordinate frame. **NW.** Every reference frame is conceptually "contained" within the world frame Cartesian coordinate system. The world frame **{W}** is the **global reference frame** (i.e. global set of axes) inside/with respect to which all other **local** reference frames are defined. {W} provides the overarching **context for all spatial measurements and orientations**. When we define a local reference frame, we are essentially **creating a new set of axes** that have a specific **orientation** and **position relative to the world frame**. Note that each point within this local frame can always be translated back into the coordinates of the world frame.
- **Point:** a point is a **specific location within** a particular **reference frame**. A point has no dimensions, direction, or area. A point does not define orientation or scale; it only indicates position —> It is essentially a set of values that indicate its position along the axes of the reference frame, with no dimensions, area, or volume.



How do we define a Robot's pose?

1. Firstly, we need to define a **fixed world reference coordinate frame**, denoted as **{W}** which serves as a point of reference for all positions and orientations of the "world" (in this case the world is the "Cartesian plot").
2. We define a **local coordinate frame**, {R}, which is centered at the robot's reference point P. Usually, the robot reference point P is placed in the middle between the two wheels. This point P is an arbitrary but specific location on the robot that we use to track its position. The orientation of frame {R} is aligned with the robot's natural orientation, meaning it moves and rotates as the robot does. A point in space, in this case, the chosen reference point P, is described by a vector r. This vector **r** represents the **displacement** of point P from the origin of the reference coordinate frame {W}.





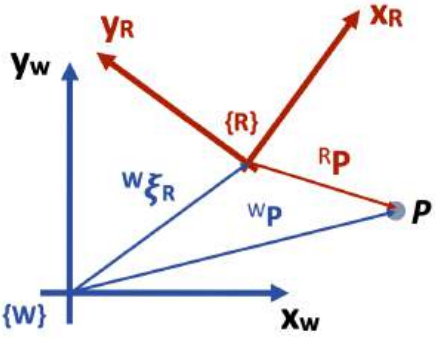
The robot's pose is then defined by both the position and orientation of the local coordinate frame {R} with respect to the world frame {W}. To clarify:

- **{R} represents the orientation of the robot:** the axes of {R} are aligned according to the robot's current orientation, and they rotate and move with the robot.
- **{R} w.r.t. {W} represents the position of the robot:** {R} within {W} coordinates represents the position of the robot. More specifically, the **position** of the robot is not represented by the frame {R} itself but by the position of the origin of {R} within the world frame {W}. Robot position is given by the **vector from the origin of {W} to the origin of {R}**. This vector represents the displacement of the robot's reference point P within the world frame.

The robot **pose** is both the **orientation** and **position**. **ξ**, called **relative pose**, in general, refers to the position and orientation of one coordinate frame relative to another. Thus, ξ_R, is the (relative) pose of the robot w.r.t {W}, the reference coordinate frame.

Use and properties of relative poses

The relative pose ${}^{W}\xi_{R}$ describes the frame {R} with respect to the frame {W}. ${}^{W}\xi_{R}$ could be seen as **describing some *motion***: <u>first applying a displacement and then a rotation to {W}</u>. This because encapsulates both the position and the rotation/orientation of an object with respect to the global reference frame. (Look at the image), note that ${}^{W}P$ vector encodes the position of point P w.r.t. {W} (global reference frame). It can be seen as the displacement of a point w.r.t. the {W}. We do not write ${}^{W}\xi_{P}$ because ξ encodes only reference frames. Now note that we can express ${}^{W}P$ as a combination of ${}^{W}\xi_{R}$ and ${}^{R}P$:



$${}^{W}P = {}^{W}\xi_{R} \cdot {}^{R}P$$

the right-hand side expresses the *motion*

{W} → {R} → *P*

NW. The operator · transforms the vector, resulting in a new vector that describes the same point but with respect to a different coordinate frame.

Thus, we saw that a point or a reference frame can be described by compositions of vectors. We said that one frame can be described in terms of another by a relative pose ξ . This procedure can be applied sequentially using ⊕ (the pose composition operator). **NW.** ⊕can be used only with ξs (i.e. compositions of relative poses), not with composition of points or relative pose with a point (in that case we still use ·).

NW. perfo… se op…ations (⊕and…) w…n **vec…rs**, …e never use …er… wit… po…ts or s…ecific referen… e frames (we …ill never see so…e…ning lik… **P** = ${}^{W}\xi_{R} \cdot {}^{R}P$ or **{R** : **WP.** …

Poses forms an additive group

The set of poses equipped with the combination operator ⊕ form an **additive group**. More specifically:

- In the case of a 2d pose this is the **SE(2) Special Euclidean group**
- In the case of a 3d pose this is the **SE(3) Special Euclidean group**

An additive group is a set G together with an operation $+$ that combines any two elements a and b in G to form another element in G, denoted $a + b$. The operation $+$ is called the group addition, and the group G must satisfy the following four axioms:

- **Closure**: For all a, b in G, the result of the operation $a + b$ is also in G.
- **Associativity**: For all a, b, c in G, the equation $(a + b) + c = a + (b + c)$ holds.
- **Identity Element**: There exists an element 0 in G, called the identity element, such that for all a in G, the equation $a + 0 = a$ holds.
- **Inverse Element**: For each a in G, there exists an element $-a$ in G, called the inverse of a, such that $a + (-a) = 0$. In this case the identity element is the *null relative pose, which refers to "not moving"*

The operation $+$ is often called "addition," and the group is called "additive," but the operation does not necessarily correspond to the usual addition of numbers. It is simply an abstract operation that satisfies these axioms. In some groups, especially when the operation is not commutative (i.e., $a + b$ does not necessarily equal $b + a$), the group might not be referred to as additive, and the operation might be denoted differently (for example, with a dot or asterisk). A commutative grouop can be either commutative (Abelian) or non-commutative (non-Abelian).

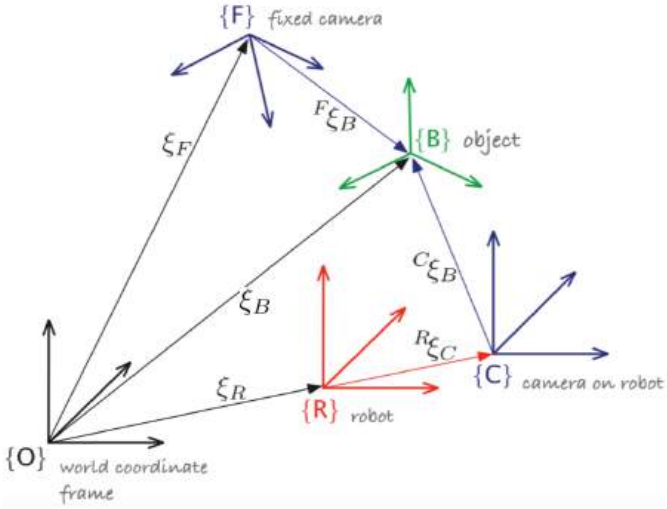
Poses, being an additive group, must respect the following properties:

Closure:	$\xi_1 \oplus \xi_2 = \xi_3$	Composition of two rel. poses results in a new rel. pose ${}^{A}\xi_{B} \oplus {}^{B}\xi_{C} = {}^{A}\xi_{C}$ ${}^{B}\xi_{A} \oplus {}^{A}\xi_{C} = {}^{B}\xi_{C}$
Associativity:	$({}^{A}\xi_{B} \oplus {}^{B}\xi_{C}) \oplus {}^{C}\xi_{D} = {}^{A}\xi_{B} \oplus ({}^{B}\xi_{C} \oplus {}^{C}\xi_{D})$	
Identity element:	$\xi \oplus 0 = 0 \oplus \xi = \xi$	0 is the *null relative pose*
Inverse:	$\ominus {}^{A}\xi_{B} = {}^{B}\xi_{A}$	$(\ominus\xi) \oplus \xi = \xi \oplus (\ominus\xi) = 0$ $\xi \oplus (\ominus 0) = \xi$
Composition is NOT commutative: (because of the angle part of poses)		$\xi_1 \oplus \xi_2 \neq \xi_2 \oplus \xi_1$

Not commutative because we allow for rotation

${}^A\xi_B \oplus {}^B\xi_C = {}^A\xi_C$	Pose composition (binary)
$\ominus {}^A\xi_B = {}^B\xi_A$	Pose inversion (unary)
${}^A\xi_B \cdot {}^BP = {}^AP$ To the right we always must have a point	Change of reference frame for a point

3D Example

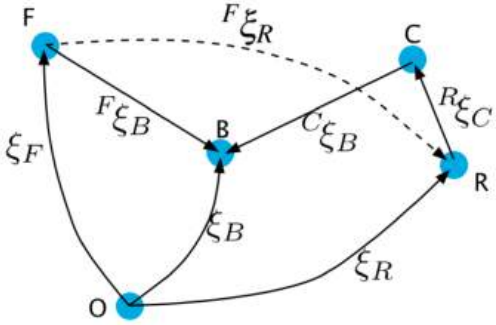


A fixed camera, at a known pose {F} wrt the world, observes an object and its orientation; {B} is the pose of an object. A robot {R}, at a known pose {R} wrt the world, is equipped with a camera (mounted at a known pose {C} wrt the robot). The robot camera observes the same object and its orientation. Reply the following:

1. I see the object in the robot's camera. I know where the robot is in the world. Where is the object in the world? (i.e. Given ${}^C\xi_B$, ${}^R\xi_C$, ξ_R find ξ_B)
2. I know the object pose in the fixed camera frame. I know where the robot is in the world. Where is the object with respect to the robot? (i.e. Given ${}^F\xi_B$, ξ_F , ξ_R find ${}^R\xi_B$).
3. I[illegible] ${}^F\xi_R$.

Solu[illegible]

1. $\xi_[illegible] = \xi_R \oplus \xi_C^R \oplus \xi_B^C$
2. $\xi_B^R = \xi_R^0 \oplus \xi_F^0 \oplus \xi_B^F$
3. ${}^F\xi_R = \ominus\xi_F \oplus \xi_R$ or ${}^F\xi_R = {}^F\xi_O \oplus \xi_R$. The inverse means to run on the vector in the opposite direction.

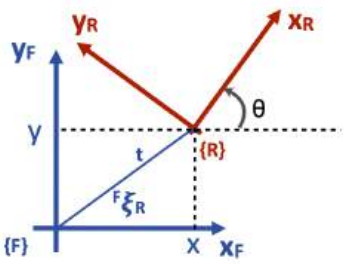


We can also use a graph to represent the poses of different entities and the relative poses between them. In the graph, each node corresponds to a pose of an entity, such as the fixed camera (F), robot (R), object (B), and camera on the robot (C). The nodes are connected by arcs, each representing a relative pose, indicating the spatial relationship between the entities.A **spatial relation** in this graph forms a loop, which means you can start at one node and follow the arcs that represent relative poses to eventually return to the starting node. This loop forms an equation that must be consistent and satisfy the closure property of the transformation group. For instance, if you follow the loop F → B → C → R and back to F, the combined transformations should equal the identity transformation, $\longrightarrow {}^F\xi_B \oplus {}^B\xi_C \oplus {}^C\xi_R \ominus {}^F\xi_R = 0$ indicating that you've returned to your starting pose in the world frame.

$\xi_F \oplus {}^F\xi_B = \xi_R \oplus {}^R\xi_C \oplus {}^C\xi_B$

$\xi_F \oplus {}^F\xi_R = \xi_R$

Representation of poses in 2D (2D Rotation Matrices)



We have seen that 3D poses can be represented by a graph, but how can we represent 2D poses in an efficient way? (we will see that we do this by using matrices). Let's first define the following:
-the displacement of a reference plane (or point) can be defined by a vector **t = (x,y)**
-the orientation by an angle **θ**.

Which means that: ${}^F\xi_R \sim (x, y, \theta)$

In short, In 2D, for a wheeled robot, a concrete representation of a pose is in a Cartesian coordinate system through the (x,y) coordinates and the θ angle for the orientation. Unfortunately this representation is not really convenient for compounding since (x1, y1, θ1) ⊕ (x2, y2, θ2) would require some complex trigonometric functions. Instead of using trigonometric functions, we can proceed as follows ———> https://www.youtube.com/watch?v=7j5yW5QDC2U .

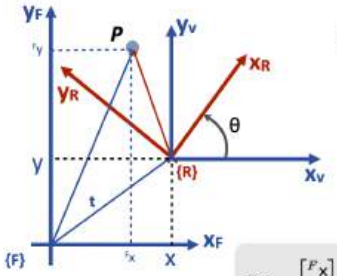
Rotation matrix

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$ Has nice properties ...

- Instead of using one scalar, the angle θ, we are using a 2x2 matrix to represent orientation
- The rotation matrix is **orthonormal**: each of its columns is a unit vector and the columns are orthogonal, that is, represent an orthonormal *basis* (the columns are the unit vectors that define {R} with respect to {V}) → 4 parameters, 3 functional relations/constraints → one independent value (the angle!)
- The determinant is +1: **R** belongs to the special orthogonal group of dimension 2, ***SO(2)***, acting as an isometry ➔ the **length of a vector is unchanged after a rotation**
- **The inverse is the same as the transpose: $R^{-1} = R^T$**
- Inverting the matrix is the same as swapping superscript and subscript, which leads to the identity $R(-\theta) = R^T(\theta) = R^{-1}(\theta)$

$$\begin{bmatrix} {}^R x \\ {}^R y \end{bmatrix} = ({}^V R_R)^{-1} \begin{bmatrix} {}^V x \\ {}^V y \end{bmatrix} = ({}^V R_R)^T \begin{bmatrix} {}^V x \\ {}^V y \end{bmatrix} = {}^R R_V \begin{bmatrix} {}^V x \\ {}^V y \end{bmatrix}$$

Add the translation



{F} and {V} have parallel coordinate axes:

$${}^F\mathbf{P} = \begin{bmatrix} {}^F x \\ {}^F y \end{bmatrix} = \begin{bmatrix} {}^V x \\ {}^V y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

${}^V\mathbf{P}$ was obtained from the rotation:

$$\begin{bmatrix} {}^V x \\ {}^V y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} {}^R x \\ {}^R y \end{bmatrix}$$

$${}^F\mathbf{P} = \begin{bmatrix} {}^F x \\ {}^F y \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} {}^R x \\ {}^R y \end{bmatrix}}_{Rotation} + \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{Translation}$$

In a more compact form: $\begin{bmatrix} {}^F x \\ {}^F y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \end{bmatrix} \begin{bmatrix} {}^R x \\ {}^R y \\ 1 \end{bmatrix}$

Homogeneous transformation Matrix

$$\begin{bmatrix} {}^F x \\ {}^F y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} {}^R x \\ {}^R y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

2D vector!
(allows us to represent a translation matrix product)

$$\begin{bmatrix} {}^F x \\ {}^F y \\ 1 \end{bmatrix} = \begin{bmatrix} {}^F\mathbf{R}_R & \mathbf{t} \\ \mathbf{0}_{1\times 2} & 1 \end{bmatrix} \begin{bmatrix} {}^R x \\ {}^R y \\ 1 \end{bmatrix}$$

Coordinate vectors for **P** are now in 2D homogenous form:
homogenous coordinate transformation

$${}^F\tilde{\mathbf{P}} = \begin{bmatrix} {}^F\mathbf{R}_R & \mathbf{t} \\ \mathbf{0}_{1\times 2} & 1 \end{bmatrix} {}^R\tilde{\mathbf{P}}$$

$${}^F\tilde{\mathbf{P}} = {}^F\mathbf{T}_R\, {}^R\tilde{\mathbf{P}}$$

$${}^F\mathbf{T}_R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}$$

The roto-translation matrix **T** is a *homogenous transformation* and belongs to SE(2)

The pose of a robot / rigid body is fully described by a *homogeneous transformation matrix*

${}^F\mathbf{T}_R$ represents the robot pose w.r.t. the word frame {F}. What you need to understand from these slides, is that the pose of a robot can be expressed as a rotation and a translation w.r.t. the word frame. And this roto-translation (i.e. the robot's pose) can be expressed in matrix form (and also in homogenous coordinates, which we are not delved in this course). **NW This is different from a 3D rotation matrix. This here is a 2D rotation + translation in homogenous coordinates, the 3D rotation matrix expresses a 3D rotation and encodes no translation.**

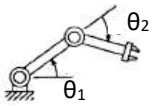
Thus, by being able to represent poses trough matrices, we get that the properties and operators of the additive group can be expressed in matrix form:

⊕ becomes a matrix product: $^A\xi_R \oplus {}^R\xi_B \equiv {}^A\mathbf{T}_R\, {}^R\mathbf{T}_B$

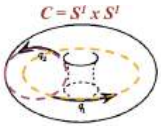
· becomes a matrix-vector product: $^A\mathbf{P} = {}^A\xi_B \cdot {}^B\mathbf{P} \equiv {}^A\mathbf{T}_B \cdot {}^B\mathbf{P}$

The identity element is the identity matrix

⊖ ***The inverse rel pose is the matrix inverse***

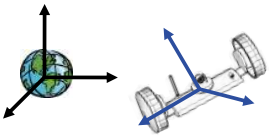


Robot arm with 2 rotational joints: the joint parameter set is given by the two θs.The configuration space is the product of two circles SxS (or, more technically, two intervals of angles), representing all possible combinations of the two joint angles.



$q = (\theta_1, \theta_2)$

$C = S \times S$



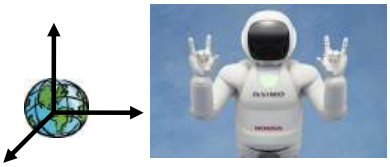
$q = (x, y, z, \alpha, \beta, \gamma)$

$C = \mathbb{R}^3 \times SO(3)$

group of 3d rotations

Rigid body in 3D space: we have (as already explained) a 6-D vector of joint parameters. The configuration space combines the 3D space ($\mathbb{R}^3$ -> RxRxR, i.e. x,y,z) for position and the special orthogonal group SO(3) representing all possible orientations of the rigid body in 3D space. SO(3) is a mathematical group that describes rotations in three dimensions.

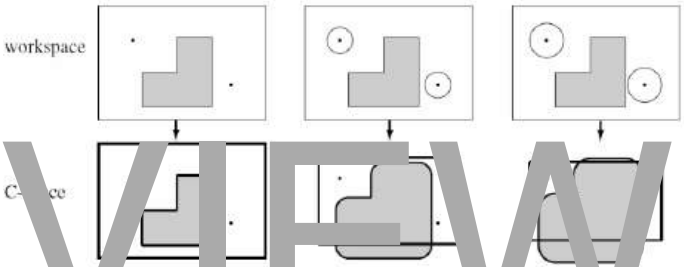
Non-Rigid Body in 3D space: in this case this humanoid robots is composed of **34 joints**, which can freely rotate or translate in a 3D environment. Thus, in this configuration space, ($\mathbb{R}^3$) represents the position of the robot, SO(3) its orientation in space, and $\mathbb{R}^{34}$ represents the 34 degrees of freedom from the robot's joints.



$C = \mathbb{R}^3 \times SO(3) \times \mathbb{R}^{34}$

Configuration Space with obstacles

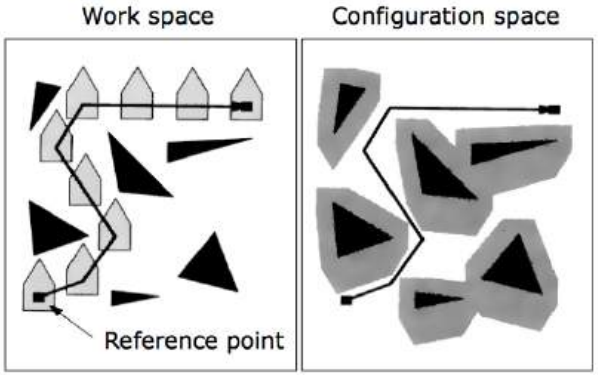
As you can see by the figure, the C-space must also takes into account the obstacles, i.e. spatial elements with such the robot cannot overlap. In the first exa[illegible] we hav[illegible] + th[illegible] borders of t[illegible] space. The [illegible] robo[illegible] size (the wh[illegible] e circle) the [illegible] bec[illegible] ne the obsta[illegible] e.

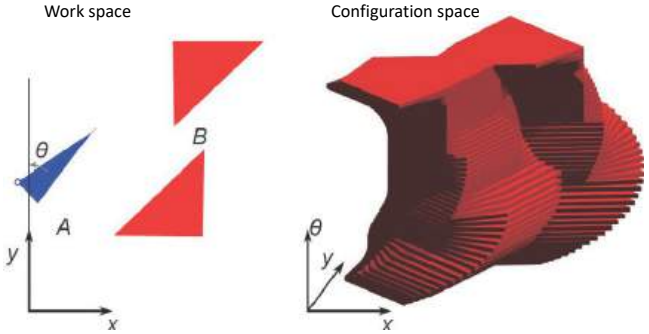


One *reference point* on the robot is selected, then the C-space is obtained wrt the reference point by sliding the robot along the edge of the obstacle regions "blowing them up" by the robot radius.
In image processing, the operation would be described as a *binary morphological dilation* with a robot-shaped structuring element.

Special case: The robot is a *polygonal* one and can only *translate*

Let's see now a special case, where the shape of the robot is a *polygon (but the robot cannot rotate). **NW.** with the use of the configuration space,* a mobile robot of any shape can be "reduced" to a **point. The robot configuration is represented by a point in the configuration space.** When the point is in the obstacle, it means that the robot is hitting the obstacles.



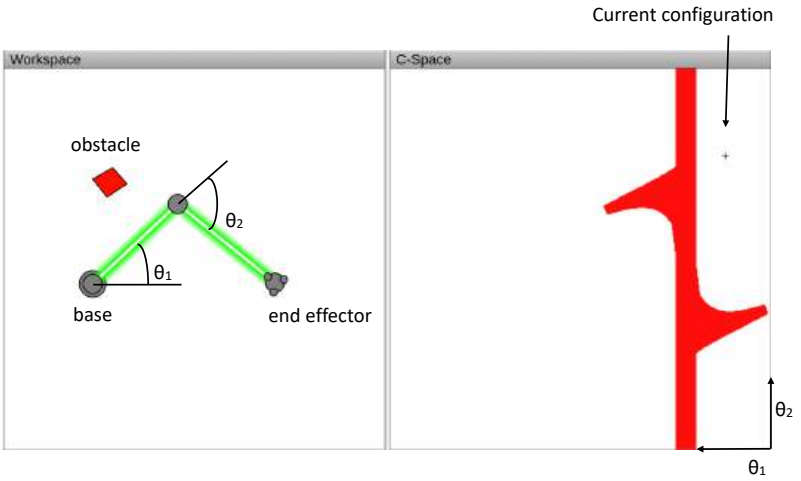


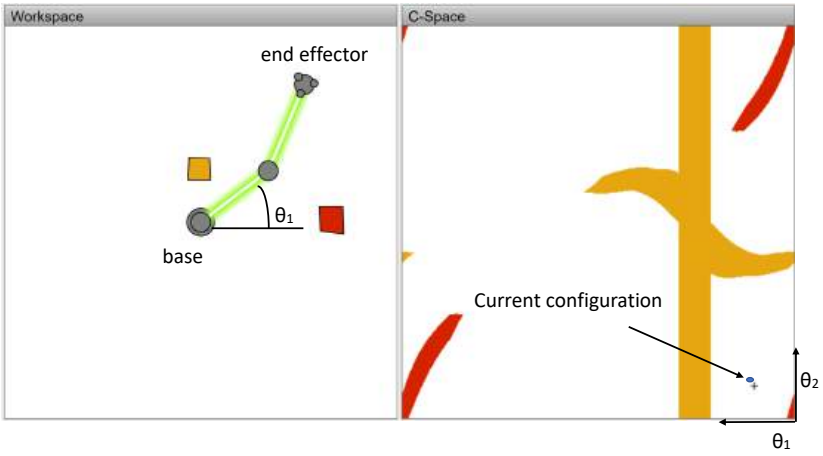
—> Configuration shape of a triangular shaped mobile robot which can translate and **rotate**. The C-Space adds a dimension which is θ, the possible angle of rotation of the robot.

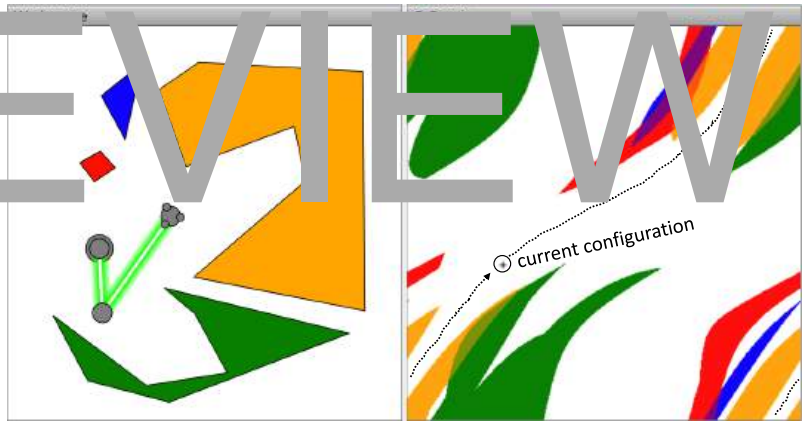
Play with C-spaces for robot arms

It is important to train and be able to somewhat predict what the C0-space is going to be given a specific arm robot with specific rotational joints and specific obstacles (is going to be in the exam).

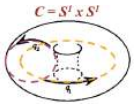
Go here to train: https://www.cs.unc.edu/~jeffi/c-space/robot.xhtml







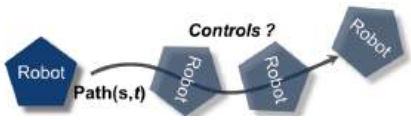
Can you find out what motion the dotted line corresponds to?
(remember the toroidal topology of this C-space!)



https://www.cs.unc.edu/~jeffi/c-space/robot.xhtml

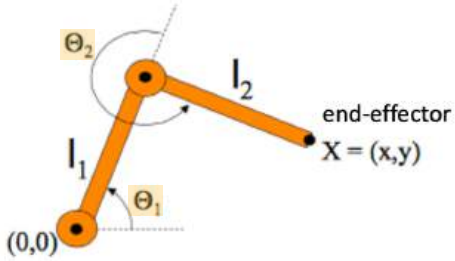
from one point to another within certain spatial constraints, navigating around obstacles as necessary.

- **Trajectory Following -> Path(set).** This method adds another layer to the concept of path following by incorporating time into the equation. This involves specifying not only the sequence of positions the robot should occupy but also the speed and acceleration at which it should move to transition between these positions. Trajectory Following requires the robot to reach specific locations at specific times, enabling more controlled and dynamic interactions with the environment and other moving objects.



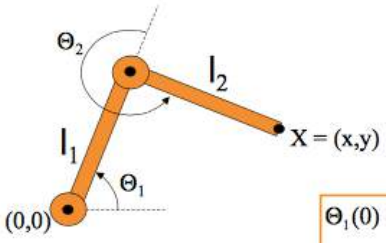
Example of Forward Kynematics

Given joint angles
determine the end-effector position



$$X = (l_1 \cos\Theta_1 + l_2 \cos(\Theta_1 + \Theta_2), l_1 \sin\Theta_1 + l_2 \sin(\Theta_1 + \Theta_2))$$

Given joint angles **and velocities**
determine the end-effector **trajectory in time**



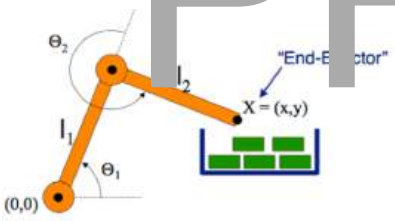
$$\Theta_1(0) = 60^\circ \qquad \Theta_2(0) = 250^\circ$$

$$\frac{d\Theta_1}{dt} = 1.2 \qquad \frac{d\Theta_2}{dt} = -0.1$$

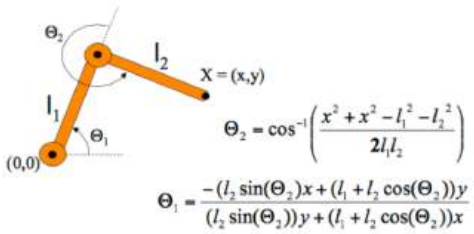
Here $\Theta_1(0)$ is the starting position of the first joint at time zero, while is $\Theta_2(0$ the starting position of the second joint at time ze. By deriving the two angles over time, you get the **angular velocity**, is the rate of change of the angle with respect to time, which tells us how fast the angle is changing, measured in degrees per second or radians per second. In this context, θ(t) is indeed a function of time, which, when given a specific time value, returns the corresponding angle of the joint at that particular moment. This function describes how the angle of the joint changes as time progresses. When you derive θ w.r.t. *t*, you are calculating the angular velocity of the joint. Angular velocity is the rate at which the angle changes over time, such that:

- if the derivative is **>0**, the joint is rotating in a direction that increases the angle θ (typically is counterclockwise).
- if the derivative is **<0** the joint is rotating in the opposite direction, decreasing the angle (typically clockwise)
- if the derivative **=0** , the joint is not rotating

Exa ple of ver e Kyner atic for Arms

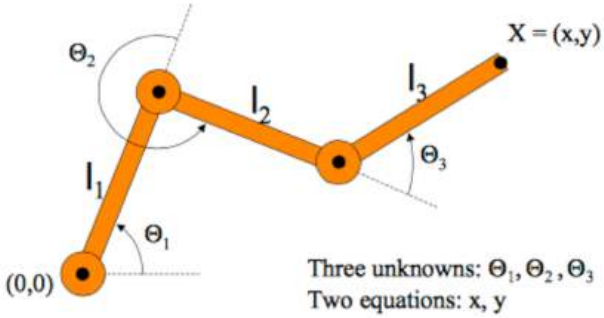


Given the desired pos X of the end-effector, determine the values for the joint variables



Solve the kinematic equations wrt the final pose

NW. In the general case, IK problems admit multiple solutions! A single/specific goal position can be reached with different setups of the parameters. The underlying mathematical equations that relate the positions (x, y) to the angles (θ1, θ2, θ3) do not have a unique inverse due to the trigonometric functions involved, which are naturally periodic and symmetrical. This means there can be different sets of angle values that satisfy the equations for the same point (x, y): due to the presence of more unknown variables (the joint angles than available equations (the end effector's x and y coordinates), the system becomes underdetermined. This underdetermination suggests that there can be multiple sets of joint angles that satisfy the same end effector position.



Forward Kynematics For a Differential Drive Mobile Robot —Lecture 5

In this section, we will focus on the kinematics of a Differential Drive mobile robot (i.e. a wheeled robot with two separately driven wheels mounted on a common axis and, typically, a castor wheel that provide balance and support but do not contribute to the robot's motion). This means that we are dealing with a robot that is not fixed in {W}, but can move around the environment, and its movement is controlled by two independently driven wheels.

Forward Kinematic Model (for a DDM Robot):
The forward kinematic model for a Differential Drive Mobile (DDM) robot translates the rotation speeds of the robot's wheels into the robot's linear and angular velocity in a global or reference frame. This model helps in understanding and predicting the position and orientation of the robot based on the movements of its wheels. The forward Kinematic Model aims to calculate the robot's **generalized velocity ξ_W**, which includes its linear velocity in the x and y directions, as well as its angular velocity around the z-axis (rotation rate). In short, **ξ_W** represents how quickly and in what direction the robot is moving and turning.

Given:
- the geometric parameters: number and type of wheels, wheel(s) radius, length of axes, …
- the initial conditions: pose and velocity
- and assigned the spinning speeds of each wheel: φ_1, φ_2

a forward kinematic model aims to predict the robot's generalized velocity (rate of pose change) in the global reference frame:

ξ_W's equation is described as:

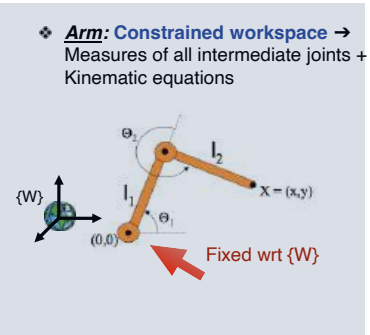
$$\dot{\xi}_W = [\dot{x} \quad \dot{y} \quad \dot{\theta}]^T = f(l, r, \theta, \varphi_1, \varphi_2)$$

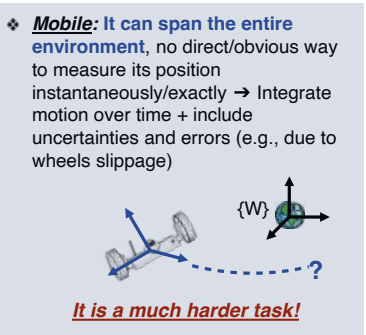
Where:
- **x:** is the rate of change of the robot's position along the x-axis, which means it's the robot's **linear velocity in the x direction**
- **y:** is the rate of change of the robot's position along the y-axis, equating to the robot's linear velocity in the y direction
- **θ**: is the rate of change of the robot's orientation (its angular velocity), representing how quickly the robot is rotating around the

The value of is computed relying on the function which encapsulates how the robot's motion parameters (the geometrical characteristics like wheel radius and separation, the robot's orientation, and the individual wheel speeds) determine its general movement within the environment:
- **l :** the radius of the wheels
- **r:** the distance between the wheels
- **θ**: the current orientation (heading) of the robot
- **ϕ1, ϕ2:** the rotation speeds of the wheels (note that ϕ1 can be different from ϕ2).

In short, the function f takes these as inputs and produces robot's linear velocity in x and y direction and robot's angular velocity.

Arm vs Mobile Forward Kinematic
FK for a Robotical Arm (i.e. fixed w.r.t. {W}) is generally straightforward since the arm's movement is restricted to a specific area, and the arm's structure provides a clear relationship between joint angles and the end effector's position.

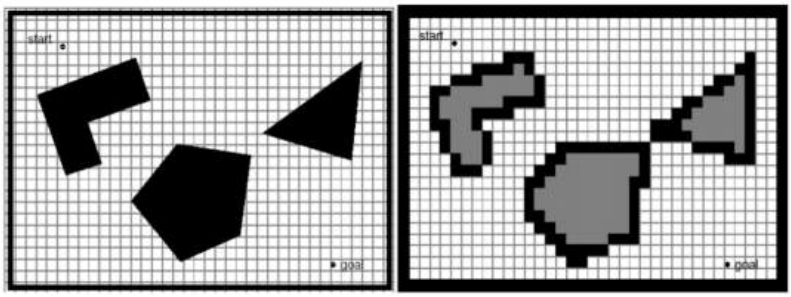




On the other hand, forward kinematics for mobile robots, particularly differential drive types, deals with movement across the potentially limitless area of the entire environment. Unlike robotic arms, mobile robots do not have a fixed workspace and can move freely. Determining their precise position at any given moment is more challenging. There is no direct or obvious way to measure their exact location instantly due to factors like wheel slippage, uneven terrain, or sensor inaccuracies. Instead, **their motion must be integrated over time**, accumulating all the small movements to estimate the robot's path and current position. This process requires accounting for uncertainties and errors that can accumulate, making the task considerably more complex.

Approximate cell Decomposition

- Regular cell tiling (squares, hexes)
- Any cell that overlaps an obstacle (even in part) is marked as obstacle
- Note: some narrow passages may disappear, depending on:
  - grid size
  - grid offset

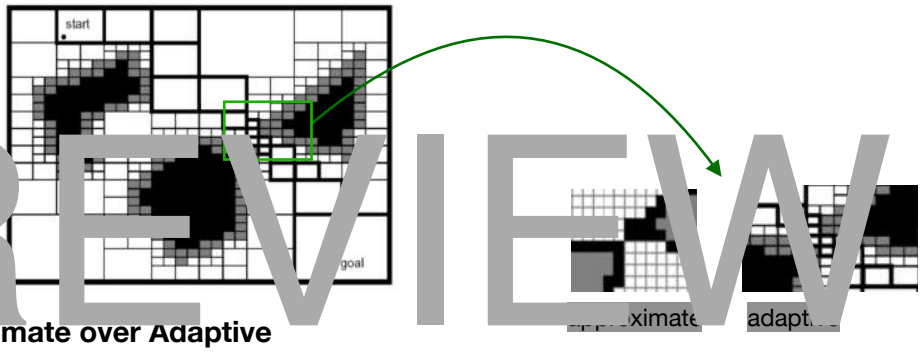


You decompose the word into a grid. Every point of your grid it can either bee fully free (white), completely occupied (grey) or partially occupied by an obstacle (black). This is an approximate information because we are loosing some information, and the loos elf information depends on the resolutions of the grid. We can also use hexagons instead of squares.

Adaptive cell decomposition

- Increases resolution close to obstacle borders (where more resolution is needed). E.g. quadtree
- Limits memory usage (even by a few orders of magnitude)
- Can handle huge areas while keeping very high resolution for obstacles
- Very effective for sparse maps

You first discrete with very large cells, the cells which are either not all free or all occupied are further split and you fo this iteratively until you get to some maximum resolution you don't want to exceed.



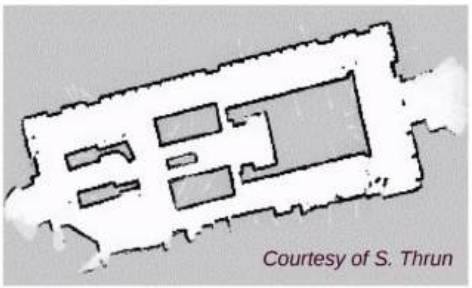
One advantage of Approximate over Adaptive

Even if the two map representations look similar, in reality with adaptive approach we have some advantages, e.g. look at this part (image). In approximate is considered as "not navigable", in the adaptive the robot can still use it as a passage.

Building a grid map

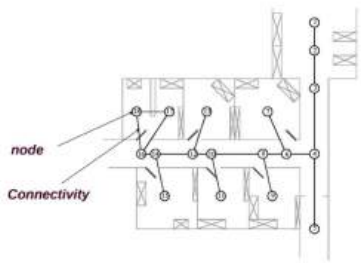
initially all the map is grey (we do not know whether there is an obstacle our not). Then, assume there is a range sensor. Given a certain position and you measure in one direction and detect an obstacle 5m away. You then can model the space between your position (which is known) and the obstacle 5m away as *white* (i.e. free). At 5m, instead, there is probably n obstacle so you can mark that as *black* (i.e. full obstacle).

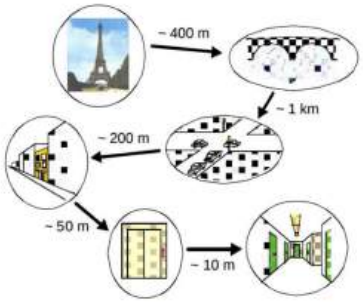
- Cells initialized gray
- A cell hit by a ranging measurement is made darker (becomes black when many rays hit that cell)
- A cell crossed by a ranging measurement is made brighter (becomes white when many rays cross that cell without being blocked)
- May change over time (e.g. dynamic obstacles)



Topological Decomposition

Very useful for high-level cognition —> similar to how our brains represent spaces. In an indoor environment you can represent all the rooms connectivity with a graph (even omitting spatial infos on the rooms). (2nd example) we can also represent rooms/POIs as nodes and edges as corridors.



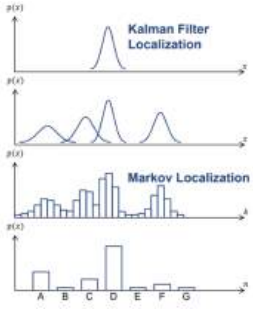


How for outdoor environments? You have different landmarks connected by the distance one between the other.

Belief representation
How to represent the belief about the robot pose? In many cases we do not want to represent the pose of the robot as one specific position in the environment, but we want to treat it as a multitude of options.

Representing the robot belief about its pose

- **Continuous** map with **single hypothesis**
- **Continuous** map with **multiple hypotheses**
- **Discretized** metric map (grid with probability distribution, **multiple hypoteses**)
- **Discretized** topological map (nodes with probability distribution , **multiple hypoteses**)



One approach to representing a robot's belief about its pose is to use a continuous map, specifically a metric map on x and y coordinates. In this context, you might have a single hypothesis about the robot's position. For example, in the frame of reference of a campus, the robot could be at coordinates (x, y, z) with a small degree of uncertainty, such as being unsure whether it is at 3.5 meters or 3 meters. This uncertainty can be represented using a covariance matrix or Gaussian distribution. Another approach is to display different hypothesis, each with its own covariance. Or, again, we can have a discretised metric map, where, for every single cell of the grid, we have a value of probability. In a topological map, instead, we can have a value of probability for every node of the graph.

Continuous vs Discrete Belief Representation

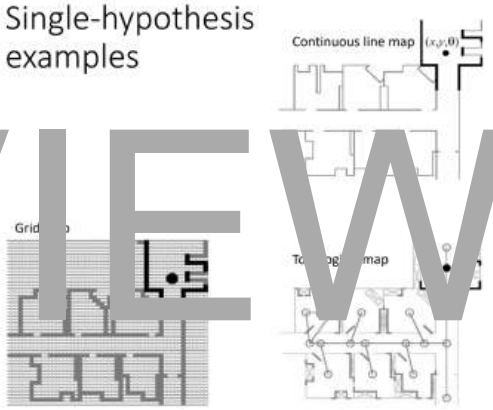
Continuous

- Precision bound by sensor data
- Typically single hypothesis (may get lost if hypothesis is v
- C mpact rep enta on and effi nt

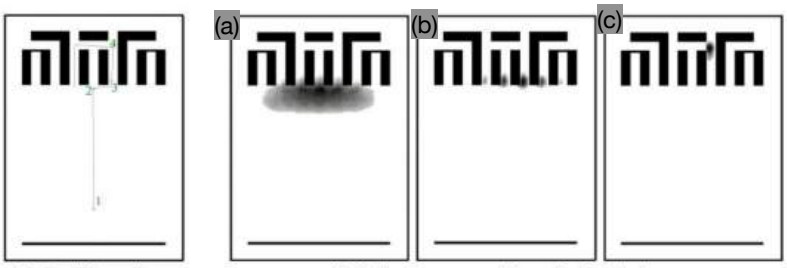
Discre e

- P cision bound by solution d iscret tion
- T ically multiple h thesis pos estim e (won't get l)
- Memory and processing requirements

Single-hypothesis examples



Multi Hypothesis on a grid map



Path of the robot *Belief states at positions 2, 3 and 4*

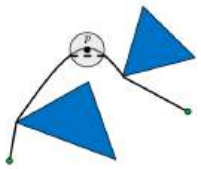
- Grid size around 20 sq cm
- Clouds represent possible robot locations
- Darker coloring means higher probability

We have a robot starting at position 1, moving at 2, then 3,4,5,6. Let's assume that the robot knows in advance he is at position 1. Then it moves forward and then gets at 2. When it gets at 2, it has some uncertainty (a). Then it moves to 3, and because it has integrated the information up to that point, it has excluded the possibility/risk of being in other positions(b), given that at the previous point it was in front of a corridor, i.e. it is supposing to be in 5 possible corridors. Then the robot moves towards 4 and understand that the corridor branches too the left(c), then the only compatible place where a corridor turn to left is the one shown in map, i.e. the beliefs get more precise.

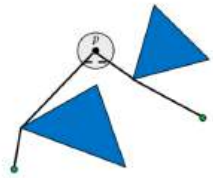
Proof of this theorem

- Assume that the path is not polygonal
  → then there is a shorter path
  → **the path must be polygonal**
- Assume that an inner vertex is in the interior of the free space
  → then there is a shorter path
  → **the inner vertices must be on the perimeter of obstacles**
- Assume that an inner vertex is on the edge of an obstacle
  → then there is a shorter path
  → **the inner vertices must be on the vertices of obstacles**

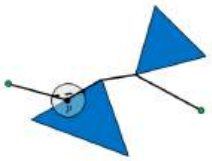
By assuming the path is not polygonal, we allow the polygon to have "curves". If I take a point on this curve, and I consider a small enough area around, then I will be able to find a "cord" that makes my path short, i.e. there is a shorter path —> **the shortest path is polygonal**.



By assuming an inner vertex does not correspond to the vertex of an obstacle: In this case, i can still pick a point on this inner vertex, and consider a small area around this point. I can "cut"/find a "cord" that would make my path shorter —> **Inner vertices must be on the perimeter of the obstacles.**



By assuming an inner vertex is on the edge of an obstacle, I can still pick a point and a small area and find a sort of cut —> the **inner vertices must be on the vertex of the obstacles, not on the edges of the obstacles.**

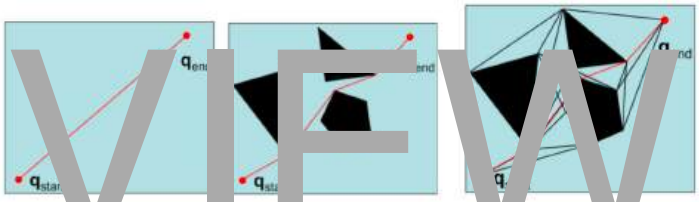


Visibility Graphs

Now I want to build a graph that contains only nodes ate the vertices of obstacles (+ the starting and target point)., How do I link the vertices? The challenge is to determine the edges, E which represent direct, unobstructed paths between these vertices. Linking all vertices to all other vertices would create many edges that are not valid due to obstacle blocking the paths. To correctly build the graph, you need an algorithm that only includes edges for paths that are in free space, not intersecting any obstacles.

Given polygonal obstacles, a road map can be defined as visibility graph (V, E)

- V = set of vertices of the polygons ∪ {start, end}
- E = set of unblocked line segments between the vertices in V



Visibility graph construction (algorithm)

An approach to build such edges is to iteratively, starting from the starting node, to sweep a line in all the direction and keep track of all the vertices that I intersect. (e.g. image) Consider the red point (starting point), if I trace lines starting from this point I find only 3 intersections with object vertices —> these are the only edges that I would build/consider (the black ones).In the next step, consider these edges and repeat the process for each connected vertex, continuously adding more vertices and edges until the graph is complete.

To clarify:

To construct the visibility graph, we use a method that involves iteratively sweeping lines from each vertex and recording intersections with other vertices. Starting from the starting node, you sweep a line in all directions and keep track of all vertices that intersect with this line.
For example, consider the red point (qstart) in the image. When tracing lines from this point, only a few intersections with obstacle vertices are found. These intersections represent the edges that will be included in the graph (indicated by the black lines). Here's how the process works:

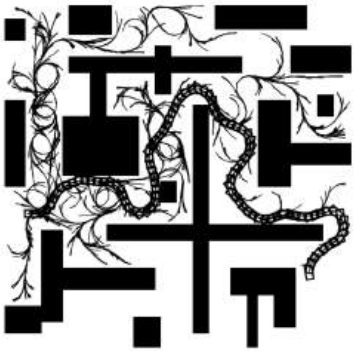
1. Begin at the starting point qstart.
2. Sweep lines from qstart in all directions, recording intersections with the vertices of the obstacles.
3. For each intersection found, draw an edge between qstart and the intersected vertex if the path does not intersect any obstacle.
4. Once all possible edges from qstart are found, repeat the process for each newly connected vertex, considering only those edges that do not intersect obstacles.
5. Continue this process until all possible edges are found and recorded.

This method ensures that only valid edges in free space are included in the visibility graph, enabling efficient path planning by connecting all visible vertices without intersecting obstacles.

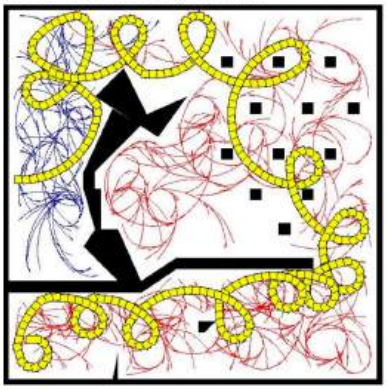
Planning with RRT (and a great trick)
the trick is, once every 10-1000 new nodes that are added, instead of sampling qrand as random, I sample it as the target —> i.e. I will replace qrand with the target and see if from the qnear i can reach the target. In any case, I am orienting my tree growth to=wards the target.

- Root the RRT at start configuration q_{start}
- At every nth iteration (e.g., n = 100), force $q_{rand} = q_{goal}$
  Think: why not at every iteration?
  Note: n acts as a greediness parameter for the algorithm
- When q_{goal} is reached, problem is solved
  The path from q_{start} to q_{goal} is immediately ready from the rooted tree structure: each node has a unique parent, such that retracing the tree from the q_{goal} node based on the parent relation, the q_{start} node (the root of the tree) is eventually reached → No need to plan the path on a graph

(Example) RRT for car-like robots
(in this case we have a car that can only turn left). This map is a 3D space (just plotted in 2D).
NW. once you have the path, you can add infinitely more vertices, the path will remain sub-optimal (i.e. won't find a best one), since, as soon as it founds a path, it uses it.



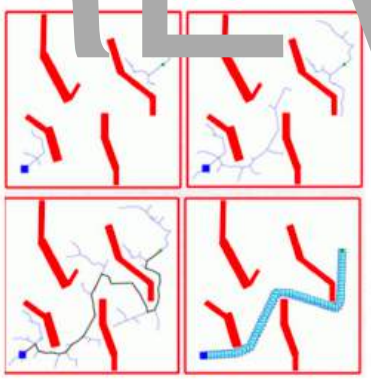
A car that can only turn left



RRT with dynamics
RRT can also be extended with **dynamics** (i.e. velocities). This CS is not 2D (x,y) and neither 3D (x,y,tetha- i.e.orientatiom), but is 5D (y,tetha velocity on x axis, velocity on y axis).

Bi-directional search with RRT: RRT-Connect

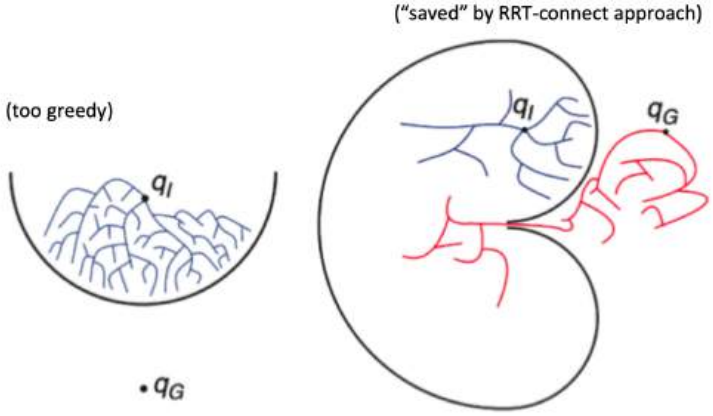
- Grow two RRTs, one from the start configuration and one from the goal configuration
- Periodically, try to extend each tree towards the newest vertex of the other tree



RRT-connect builds two trees, one rooted at start and one rooted at target. And every n iteration you will try to grow a vertex of the first tree with the vertex of the second.

(Check slides for step by step simulation/example of RRT-Connect, SLIDE 67-76)

Difficult cases for RRT
There might have situation where you experience local minima: if you bias your growth too much you do not explore too much and you do not find the path.
The double tree (RRT-connect) may help to del with this kind of situations



(i skipped RRT examples since he doe not saying anything interesting)

RRT paths are very suboptimal

Intuitively, the problem is that the tree grows and does not adapt as new nodes are added: old branches are never removed as the tree grows, even when they are very suboptimal

RRT*

A variant of RRT that rewires the tree as it is grown.

- RRT only tries to extend the nearest vertex towards the new sample
- RRT* tries to extend several nearby vertices towards the new sample
  - If the new vertex is reachable through more than one way, the lowest-cost connection is made
  - the other connecting vertices are then reconsidered: they might in fact be reachable through a cheaper path via the new vertex!
  - the resulting tree is continuously rewired as new nodes are added: grows in a much smoother way

A solution tooth's sub-optimality is RRT*. Basically RRT*, when it grows the tree, it can re-wire the rest of the tree to find a better way to get to the same target. This solves the main issue of RRT, i.e. its in-possibility to modify the tree once is grown.

(Check slides for step by step simulation/example of RRT*, SLIDE 82-89)

RRT vs RRT* as the tree gets denser

Look how by expanding again in the tree after having found the path, RRT still remains the same while RRT* improves.