Good Coverage: almost any point of the configuration space can be connected
Connectivity is difficult to capture when there are narrow passages
Visibility: All configurations in Free Space that can be seen by a free configuration p

- ***e-good*:** Every free configuration "sees" **at least** a *e* fraction of the free space, *e* is in (0, 1]
- ***b-lookout*** of a subspace S: Subset of points in S that can see a *b* fraction of F/S
- ***(e,a,b)*-expansive:** The free space F is *(e,a,b)*-expansive if
  - Free space F is *e-good*
  - For each subspace S of F, its b-lookout is at least *a* fraction of S.
  - *a* = Volumn(*b-lookout*) / Volumn(S)
- Bigger *(e,a,b)* means easier to construct a roadmap with good connectivity / coverage -> small means narrow
- A C-space with *larger lookout* set has *higher probability* to construct a linking sequence with the same number
- Probability of achieving good connectivity *increases exponentially* with the number of milestones
- **Limitation**: No theoretical guidance about the stopping time / stops either path found or max steps has taken

Simple car:
Car configuration: $R \times S^1$ (Translation and rotation in 2D space)

- Control input: Velocity: s; Steering angle: ø
- In small time interval $\triangle t$, the car must move approximately in the direction that the rear wheels are pointing.
- Input controls: U_S (speed) and $U_ø$ (steering angle)
-
- Simple car is considered as nonholonomic because there are differential constraints that are cannot be completely integrated.

- Input controls: u_s (speed) and u_ϕ (steering angle)
- Equations of motions: $\dot{x} = u_s \cos\theta$ $\dot{y} = u_s \sin\theta$ $\dot{\theta} = \frac{u_s}{L} \tan u_\phi$

$$\dot{x} = u_s \cos\theta$$
$$\dot{y} = u_s \sin\theta$$
$$\dot{\theta} = u_\omega.$$

- $\dot{x} = r u_\omega \cos\theta$
- $\dot{y} = r u_\omega \sin\theta$
- $\dot{\theta} = r u_\psi / L$

What are the bounds on the steering angle?
What are the bounds on the speed?

Tricycle
- $u_s \in [-1, 1]$ and $u_\phi \in [-\pi/2, \pi/2]$
- Can it rotate in place?

Standard simple car
- $u_s \in [-1, 1]$
- $u_\phi \in (-\phi_{\max}, \phi_{\max})$ for some $\phi_{\max} < \pi/2$

Reeds-Shepp car
- $u_s \in \{-1, 0, 1\}$ (i.e., "reverse", "park", "forward")
- u_ϕ same as in the standard simple car

Dubins car
- $u_s \in \{0, 1\}$ (i.e., "park", "forward")
- u_ϕ same as in the standard simple car

Kinematic (first-order) model

State $s = (x, y, \theta)$
- Position $(x, y) \in \mathbb{R}^2$
- Orientation $\theta \in S^1$

Control inputs $u = (u_\sigma, u_\omega)$
- Translational velocity $u_\sigma \in \mathbb{R}$
- Rotational velocity $u_\omega \in \mathbb{R}$

Motion equations $\dot{s} = f(s, u)$, where

$$\dot{s} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} u_\sigma r \cos\theta \\ u_\sigma r \sin\theta \\ u_\omega \end{bmatrix}$$

Dynamics (second-order) model

State $s = (x, y, \theta, \sigma, \omega)$
- Position $(x, y) \in \mathbb{R}^2$
- Orientation $\theta \in S^1$
- Translational velocity $\sigma \in \mathbb{R}$
- Rotational velocity $\omega \in \mathbb{R}$

Control inputs $u = (u_1, u_2)$
- Translational acceleration $u_1 \in \mathbb{R}$
- Rotational acceleration $u_2 \in \mathbb{R}$

Motion equations $\dot{s} = f(s, u)$, where

$$\dot{s} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\sigma} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \sigma r \cos\theta \\ \sigma r \sin\theta \\ \omega \\ u_1 \\ u_2 \end{bmatrix}$$

Differential Drive:
reducing degree by increasing dimension

- Input controls: u_l and u_r, the angular velocity of left wheel and right wheel
  - if $u_l = u_r$, moves forward with current direction
  - if $u_l = -u_r$, rotates clockwise because wheels are turning in opposite directions

Potential functions
-Potential function is a differentiable real-valued function, can be viewed as energy
-Potential function viewed as landscapes moving the robot from **high-value state** to **low-value state**.
-Robot moves **downhill** by following the negated gradient U / Robot terminates when it reaches a point where ∇U vanishes
-Terminating point called critical point, can be maximum, minimum, or saddle point
Hessian: / positive-definite: local minimum / negative-definite: local maximum
Only consider Hessians are nonsingular -> have isolated critical points -> potential function is never flat
For gradient descent methods, no need to compute Hessian because the robot generically terminates its motion at a local minimum.

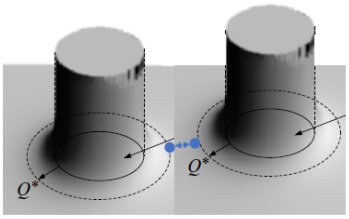
Since gradient descent decreases, so robot cannot arrive at local maximum / Saddle point is unstable, so unlikely to arrive.
Problem: the existence of local minima is not corresponding to the goal -> does not lead to complete path planner.
Tow solutions:
1. augments the potential field with a search-based planner
2. defines potential function with one local minimum, navigation function

*Both methods require full knowledge of the configuration space prior to the planning event.

Attractive / Repulsive Potential: The goal attracts the robot while the obstacles repel it.

The sum of these effects draws the robot to the goal while deflecting it from obstacles.

- Designing **Uatt:** should be monotonically increasing with distance from q(goal), using simplest: conic potential
- Problems with conic:
  - In numerical implementation shows chattering: *discontinuity in the attractive gradient at the origin*
  - (del)U(q) is discontinous at qgoal
  - Need continuously differentiable: magnitude of attractive gradient decrease as robot approaches qgoal.
  - One solution is: grows quadratically **(Why it is continuously differentiable)**



$$U_{\text{att}}(q) = \frac{1}{2}\zeta d^2(q, q_{\text{goal}})$$

  - The farther away q is from qgoal, the bigger the magnitude of the vector.
  - Reduces “overshoot” of the goal (resulting from step quantization) - good / bad: far away from goal will produce high velocity
  - **So we have combine conic with quadratic for Uatt**
- Repulsive potential: keeps the robot from an obstacle
  - The closer to an obstacle, the stronger the repulsive force
  - Define Urep in terms of distance to the closest obstacle: D(q)
  - Problem: numerical implementation may result in oscillations between points that are two-way equidistant from obstacles
  - Solution:
    - Instead of define potential function in term of the ***closest*** obstacle, redefine to ***individual*** obstacles / oscillations do not occur for convex obstacles
    - no more radical changes in the closet point / Non-convex obstacles can be decomposed into convex pieces

Gradient Descent Algoritm

- **Idea:** start at initial configuration -> small step in direction opposite to gradient -> now at new config-> repeat
- **Compute distance:** att distance easy to get because distance to point / rep distance hard to get because distance to obstacles
- Repulsive: can use range sensor distance / discrete distance
- **Local Minima Problem:** all gradient descent algorithms -> Getting stuck in local minima
  - Gradient descent is guaranteed to converge to a minimum in the filed
    - No guarantee this minimum is the global minimum / No guarantee gradient descent will find a path to q(goal)
  - **Solution: Wave-front planner** Use a grid / Set obstacles to 1, start cell to 2 / Grows a wave front from the goal
    - *Drawback: has to visit entire grid*

Navigation Potential Functions

- Navigation Potential Function is a function if it: **has unique minimum at q(goal)**

- Define attractive function

$$\gamma_\kappa(q) = d^{2\kappa}(q, q_{\text{goal}})$$

- $\gamma_\kappa(q) = 0$ for $q = q_{\text{goal}}$
- $\gamma_\kappa(q)$ increases continuously as q moves away from q_{goal}

- Define repulsive function

$$QO_i = \{\, q \mid \beta_i(q) \le 0 \,\}$$

$$\beta(q) = \prod_{i=0}^{n} \beta_i(q)$$

- $\beta_i(q)$ is negative in the interior of obstacle i
- $\beta_i(q)$ is zero on the boundary of obstacle i
- $\beta_i(q)$ is positive in the exterior of obstacle i

- $\beta(q) = 0$ for q on the obstacle boundary
- $\beta(q) > 0$ for all $q \in C_{\text{free}}$

- Navigation potential function

$$\varphi(q) = \frac{\gamma_\kappa(q)}{\lambda\,\beta(q) + \gamma_\kappa(q)}$$

- $\varphi(q_{\text{goal}}) = 0$
- $\varphi(q) \to 1$ as $q \to \partial\, QO_i$
- For large enough κ, φ has a unique minimum

- Problem with this φ
  - Not necessarily a Morse function
    - i.e., may have degenerate critical points

  - Sphere Space: Assume everything is **sphere**
  - Navigation potential function
    - With increasing k:
      - critical points gravitate towards the goal / local minima turn into saddles
      - it becomes flat near the goal and far away from goal / Sharp transition between the two
      - Difficulties in implementation due to numerical errors
  - **Star - Space**
    - Star-space are diffeomorphic to sphere-space / All convex sets are star-shaped, but converse is not true
    - Mapping between star & shpere - spaces
  - **Jacobians**

- A Jacobian defines the dynamic relationship between two different representations of a system

Discrete Search

- Discretization of C-space using grid, can be represented as a graph.
- Discrete cells are nodes, connectivity of neighbors captured by edges.
- A* Algorithm: BFS search produces the shortest path in terms of link lengths
  - A* algorithm use ***heuristic*** (estimates the cost to the goal node) to search a graph efficiently.
  - A* always terminates, ensuring completeness, does not search the entire space -> efficient
  - all path with overall cost lower than the goal must be explored to guarantee that shorter path
- Dijkstra's Algorithm: Special cases of A*
  - f(n) = h(n): becomes greedy because it only considers what is "believes" is the best path
  - f(n) = g(n): planner not using heuristic, grows a path that is shortest from the start until goal

Cell Decomposition:

- (1) Exact cell decomposition: trapezoidal decomposition
- (2) Approximate cell decomposition: quadtree, octree / Tree branches until each sub-region satisfies some property
- Use of cell decomposition: Image processing, Image compression, 3D rendering, Location queires

Optimal Motion: Find a feasible path minizes the cost function over the path

- *cost function:* minimize / maximize: path length / Smoothness / Energy Usage / Predictability / Legibility

Asymptotically Optimal Planners: sampling-based planners that optimize for a cost

- ***RRT*:*** RRT* will produce smoother line, no big difference with RRT when iteration is small.
  - Will consume lots of computation power and time since it will reconnect nodes in graph.
- **PRM*:**
  - The radius used to find neighbors varies by the number of nodes that have already been placed. radius will be large when there are few nodes and shrink as more nodes are added.
  - The benefit is that there are more straight roads when a path is generated.
- **Lifelong Planning A* (LPA*):** "Lifelong" because it *saves and reuses information* from past searches
  - **Designed for edge additions / deletions and changing weights**
  - Consistent nodes: save g(x) from previous query is still the shortest distance among neighbors
  - Inconsistent: edges were added, saved g(x) is no longer the shortest distance
- **RRT#:** RRT* with LPA* ensuring that the tree is always consistent.
  - Promising nodes: estimated cost of a path through that node is less than the current best path
  - After adding a new node to the tree, updates parent and cost-to-come for promising nodes
  - You only have to rewire a small fraction of all nodes in the graph
- **FMT*:** Sample all points upfront / Always terminates
  - Asymptotically optimal in terms of the number of samples, not running time
- **Informed RRT*:** tries to reduce the cost of rejection sampling
  - RRT* until you find an initial path / Bound the problem with the L2-Norm, prune nodes outside the bound
  - Sample from the PHS defining that L2-Norm /Repeat until convergence / Final path
- **BIT*:** Batch Informed Trees -- Creates an edge-implicit graph is batches
  - Searches the graph in order of potential solution quality until no more samples or the solution can't be improved
  - Adds a batch from the informed set: old samples are pruned or reused
- **Optimizing Planners:** refine and perturb an initial trajectory to find some "optimal" motion / No completeness guarantees
  - **CHOMP**: Covariant Hamiltonian Optimization for Motion Planning
    - start with some initial trajectory (usually straight line)
    - Represents a trajectory as a series of waypoints
    - Perform gradient descent on U
  - **TrajOpt**: Optimization -- Constraints: Dealt with separately from costs
    - Ensures path smoothness doesn't have priority over obstacle avoidance
    - Uses sequential quadratic programming (SQP): Can be treated as a black box
      - Given the optimization problem, QP generates a direction which makes progress on the original problem
- An alternate approach: **Anytime Solution Optimization:**
  - Shortcutting: Find a shorter, collision-free segment between two waypoints along a path / Randomized; fast
  - Path Hybridization: Combine protions of multiple solutions paths to yield a shorter, hybird path
- *Asymptotically Optimal Sampling Based Planners:* Sample and states connect / complete and relatively slow
- *Optimizing Planners:* Begin with an trajectory and perturb until optimal / No completeness guarantees and fast

Motion Planning Under Uncertainty

- *External uncertainty:* ***environment***
  - Maps have limited accuracy and detail / Objects may be moving; Doors open / close
  - Broadest category of uncertainty -> Difficult to model a priori: information gathering / may not valid
  - Methods are typically reactive: Behavior-based protocols / Vector field histogram / Dynamic window

- **Markov Decision Process(MDP):** consider the probability of future uncertain events during planning to avoid failure at runtime.
- Compute a **policy**: Specification of an action for *every* possible state
- We assume robot has the Markov property
    - Effects of future actions do not depend on past history
    - Next state is a function of current state & action
- An MDP is: Set of states S, Set of actions A, Transition probability function: T(s' | s, a), Real-valued reward function: R(s)
- A policy is a mapping: S -> A: For each state, what kind of action we should pick
    - Specification of an action for every possible state
- An optimal policy maximizes the total expected reward over all states
- Computing a **policy** may be **worthwhile** if:
    - Actions are not reversible / we care about worst-case performance .
    - We require *predictable* behavior / we care about *realtime* behavoir.
    - We expect many repetitive disruptions. since we only compute policy once and can reuse.
- Bellman's Equiation: $J^*(s) = R(s) + \max_{a \in A}[\gamma \sum_{s' \in S} P(s'|s,a)J^*(s')]$
- J*(s) is the maximum total expected reward for state s
- **Stochastic Motion Roadmap:**
    - analogous to a PRM: Sample n valid states / Discover probabilistic transitions between states
    - Input: Validity checker / Discrete action set / Uncertain motion model
    - Query: Start, goal states / Maximize probability of reaching the goal
    - After sampling, the transition probabilities between each pair of states must be discovered
    - For each state s and action a, Simulate system m times from s using a
    - Find nearest state after each simulation / Add the transitions to model
    - **Construction:** Once constructed, the SMR is a weighted, directed graph
    - **Reward Function:** maximize probability of reaching the goal
        - Use infinite horizon dynamic programming to find the optimal policy
    - **SMR: Takeaways:**
        - Motion planning under action uncertainty is feasible for real-world systems / Scalability is a concern
        - Method is sensitive to the number of Monte Carlo simulations during construction
        - Incorporation of additional kinds of uncertainty is an open question
- **Action and Sensing Uncertainty:**
    - robot need to observate the environment / System may not know exactly where it is
    - If the state is partially-observable, how can we be sure that we select the right action
        - MDP-like policy fails: Transitions must still maintain Markov assumption
    - *Belif Dynamic:* After taking an action ***a*** and receiving an observation ***o***, the current belief estimate changes
        - Denote ***b*** the current belief, and b(s) the probability that the system is at state s
        - belif update shorthand denotes: $b' = \tau(b, a, o)$
    - POMDP (Belief-space MDP): partially-observable Markov decision process
    - $$J^*(b) = R(b) + \max_{a \in A}\left[\gamma \sum_{o \in O} \Omega(o|b) J^*(\tau(b,a,o))\right]$$
        - **Core idea:** compute optimal policy over reachable and probable beliefs
- *Internal effects:* ***motion / actuation***
    - Motor commands are not exact / Systematic bias; Mechanical imperfections.
- *Perceptual limitations:* ***sensing***
    - Configuration of robot is not known exactly / Sensors have finite resolution / Processing sensor data is difficult.

Task Planning

- **Motivation:** We want a household robot to be able to perform tasks / Give some high-level goal to achieve
- We assume motion planning is trivial. In reality, the difficulty of motion planning means there is an interplay between task planning and motion planning that makes Task and Motion Planning more than just Task Planning + Motion Planning.
- Formal logic: We want a way to express the planning problem to a computer
- **Propositional Logic:** Boolean algebra or Boolean logic
- 1 implies 1 = 1, 1 implies 0 = 0, 0 imples 1 = 1, 0 implies 0 = 1
- **First-Order Logic:** Propositional Logic + Functions, Relations, and Quantifiers
    - Functions: $2^{var} \rightarrow var$ / Relations: $2^{var} \rightarrow \{0, 1\}$ / Quantifiers: $\forall (all)$, $\exists (exist)$
- **Planning Domain Definition Language:**
    - PDDL is a way to express a task planning problem and only allows finite domains, can't representing infinite domain
    - Each step could be true or false
- **The Task Planning Problem:**

- Find: Sequence of actions from start to goal
- Approaches: Heuristic Search / SAT-Plan / SAT-Plan: Encode planning problem as Boolean formula