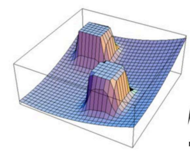
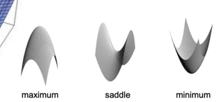
Topics:

- What is Motion Planning?
  - Taking a **start pose** and a **goal pose**, and building a **reference motion** (plan) within the C-space, that adheres to **constraints**
- Bugs
  - **Bug 0**
    - Go toward goal, turn left/right if hitting an obstacle
  - **Bug 1**
    - Go Towards the Goal, until we reach the Goal/Obstacle, if Goal - end, if obstacle, circumnavigate around the obstacle, keep memory of nearest point towards the goal, Go to the leave point, go towards goal again - Continue till goal
  - **Bug 2**
    - Define an “m-line” that points from the start to the goal
    - follow the m-line until you hit an obstacle, circumnavigate until you encounter the m-line again, Continue until you hit the goal.
- Potential Fields
  - Define obstacles as high potential and goal as lowest potential, and go downwards. The field is defined by a **Potential Function**.
  - Simplest version is attractive-repulsive fields, where obstacles repulse and goal attracts. F’n —————> $U = U_{att} + U_{rep}$
  - These have the same problems as all gradient descent functions. (Local minima and friends)





$$H = \begin{bmatrix} \frac{\partial^2 U}{\partial q_1^2} & \cdots & \frac{\partial^2 U}{\partial q_1 \partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 U}{\partial q_1 \partial q_n} & \cdots & \frac{\partial^2 U}{\partial q_n^2} \end{bmatrix}$$

- Roadmaps and Discrete Search
  - Generally: create a graph from known obstacle information
    - Approach 1: Rasterize using a grid or other shape
    - Approach 2: Create a roadmap based on vertices of obstacles
  - Path Planning in graphs
    - **DFS** - enter start node, put all its neighbors in a stack, enter top of stack, continue until at exit
    - **BFS** - same as DFS but with a queue instead
    - **A***
      - BFS but with a priority queue. Priority is defined by distance to get here g(x) and heuristic h(x)
      - **Consistent heuristic**: Optimistic (underestimates distance)
      - **Admissible heuristic**: h(A) ≤ c(A, B) + h(B)
  - **Homotopic paths**: paths that can continuously be deformed into one another.
- C-space, and C-space obstacles.
  - We need C-space as a measure to represent the joint-values (q).
    - **C- space is mapped by joint values**, But it does not represent the robot's position in space.
    - a Valid path in C-space, is valid in workspace as well.
    - C- space contains all possible configurations.
  - Dimension of a C-space = DoF of the Robot.
    - the number of parameters defining a Robot is not Always equal to DoF of Robot.
  - C-space obstacle can be found using **Minkwoski Difference** of (Obstacle) O and A (Robot).
    - computation efficiency : O(n+m) (n = num of vertex in obstacle, m = num of vertex in robot)
  - Convex obstacles can be made by **Gift-wrapping algorithm** - O(nh) complexity
    - n = number of point, h = number of points in the convex hull.

Obstacle region $C_{obs} \subseteq C$ is defined as

$$C_{obs} = \{ q \in C \mid A(q) \cap WO \neq \emptyset \},$$

The *free space* C_{free} is the set of free configurations

$$C_{\text{free}} = C \setminus C_{obs}$$

$$O \ominus A = O \oplus -A$$

$$P \ominus Q = \{ p - q \mid p \in P\ ,\ q \in Q \}$$

- **Topology**——————————————>>>

Cube $\sim S^2$

$SO(2) \sim S^1$

$SE(3) \sim \mathbb{R}^3 \times SO(3)$

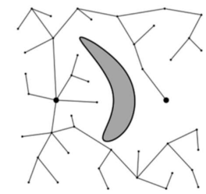
$\mathbb{R}$:	real number line
$\mathbb{R}^n$:	n-dimensional Cartesian space
S^1:	boundary of circle in 2D
S^2:	surface of sphere in 3D
$SO(2), SO(3)$:	set of 2D, 3D orientations (special orthogonal group)
$SE(2), SE(3)$:	set of rigid 2D, 3D translations and rotations (special Euclidean group)
$A \times B$:	Cartesian product, power notation $A^n = A \times A \times \cdots \times A$
$T = S^1 \times S^1$:	torus

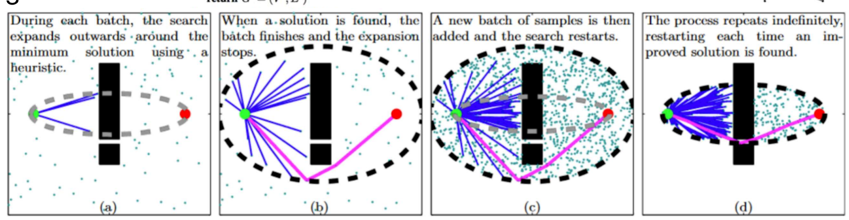
- Path Planning
  - Probabilistic Roadmap planners
    - Sample random points in the configuration space without creating the c-space obstacles
  - Tree Based Planners
    - **RTP** : Sample a random node, and then make a collision free path to it, from a random node.

    - **RRT**: Sample a random node, and then make a collision free path to it, from the nearest node. (on the right of the algorithm below)
  - Asymptotically optimal planners. NOTE: most of these require additive weights (clearance doesn't work for example)
    - **RRT*** extend (on the left) —>
    - **RRT#** ^ but with knowledge of which paths can reach goal better than existing sol.
    - **FMT*** : places nodes first and then extends quickly,
    - **IRRT***: RRT and then creates oval based on pathlen
    - **BIT***: Starts with a tiny oval and expands to get a path early, then samples again using that oval ——————————————————>

```
V' ← V; E' ← E;
x_nearest ← Nearest(G, x);
x_new ← Steer(x_nearest, x);
if ObstacleFree(x_nearest, x_new) then
    V' ← V' ∪ {x_new};
    x_min ← x_nearest;
    X_near ← Near(G, x_new, |V|);
    for all x_near ∈ X_near do
        if ObstacleFree(x_near, x_new) then
            c' ← Cost(x_near) + c(Line(x_near, x_new));
            if c' < Cost(x_new) then
                x_min ← x_near;
    E' ← E' ∪ {(x_min, x_new)};
    for all x_near ∈ X_near \ {x_min} do
        if ObstacleFree(x_new, x_near) and
        Cost(x_near) >
        Cost(x_new) + c(Line(x_new, x_near)) then
            x_parent ← Parent(x_near);
            E' ← E' \ {(x_parent, x_near)};
            E' ← E' ∪ {(x_new, x_near)};
return G' = (V', E')
```

```
V' ← V; E' ← E;
x_nearest ← Nearest(G, x);
x_new ← Steer(x_nearest, x);
if ObstacleFree(x_nearest, x_new) then
    V' ← V' ∪ {x_new};
    E' ← E' ∪ {(x_nearest, x_new)};
return G' = (V', E')
```





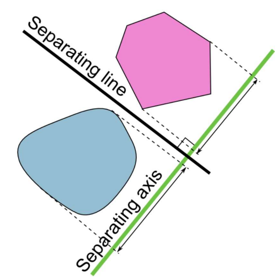
(a) During each batch, the search expands outwards around the minimum solution using a heuristic.

(b) When a solution is found, the batch finishes and the expansion stops.

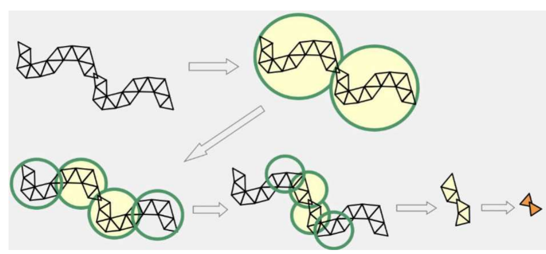
(c) A new batch of samples is then added and the search restarts.

(d) The process repeats indefinitely, restarting each time an improved solution is found.

    - A*
- Collision checking
  - Done to reduce amount of effort required to know things AREN'T colliding. Doesn't help much with checking for genuine collision though.
  - Bounding Volumes (**most are O(1) to check collisions**):
    - **Bounding Spheres**: fast but imprecise
    - **Axis Aligned Bounding Boxes**: may be more or less precise than Spheres
    - **Oriented Bounding Boxes**: slower to make. Hard to find a good orientation.
    - **Discrete Orientation Polytope**: based on k pre-determined axes.
    - **Convex Hull**: O(nlogn) to create, most precise.
    - All of these rely on the **separating axis theorem**: for two non-overlapping convex objects, there is an axis that you can project both objects on without intersection on that axis.



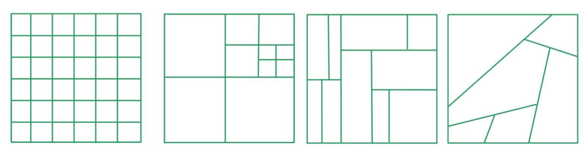
  - To make sure you're actually colliding, **hierarchical bounding volumes** are used. By going smaller and smaller, you can find collisions accurately even if you start with something coarse. ——————————————————>



  - Total Cost of checking:
    - N_{bv}: number of bounding volume overlap checks
    - C_{bv} : cost of a bounding volume overlap check
    - N_{ex} : number of exact intersection checks
    - C_{ex} : cost of an exact intersection check

$$N_{bv} \cdot C_{bv} + N_{ex} \cdot C_{ex}$$

  - Space Partitions
    - **Uniform Grid**: grid of uniform cell size
    - **Octree**: more details in specific places
      - can be used with point clouds to do easy perception based object modeling



Uniform grid | Quadtree / Octree | k-d tree | BSP tree

Good Luck!