MEAM 620

PATH PLANNING

Last Time: Planning in Graphs

Last time we focused on *graph search*.

We showed how to apply graph search to path planning in grid/voxel worlds.

Pac Man

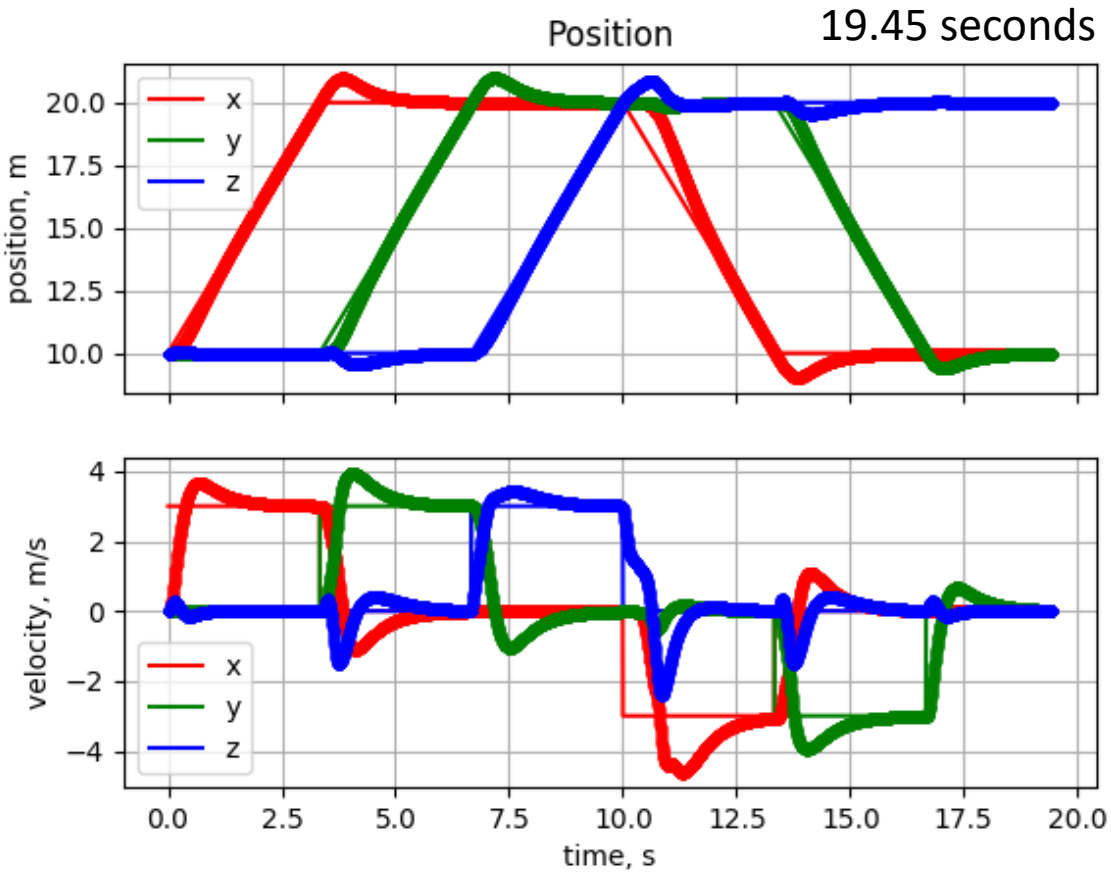
What we'll Cover Today

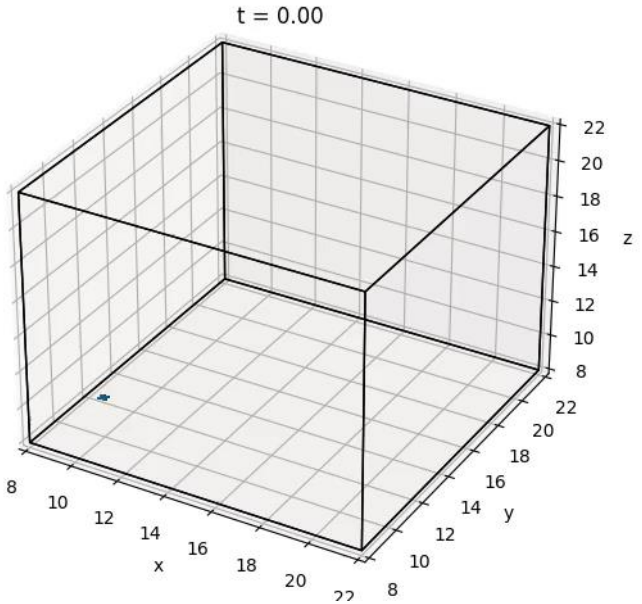
- The "configuration space."
- Three techniques which reduce the path planning problem to graph search:
  - Cell decompositions
  - Roadmaps
  - Sampling / Probabilistic Techniques

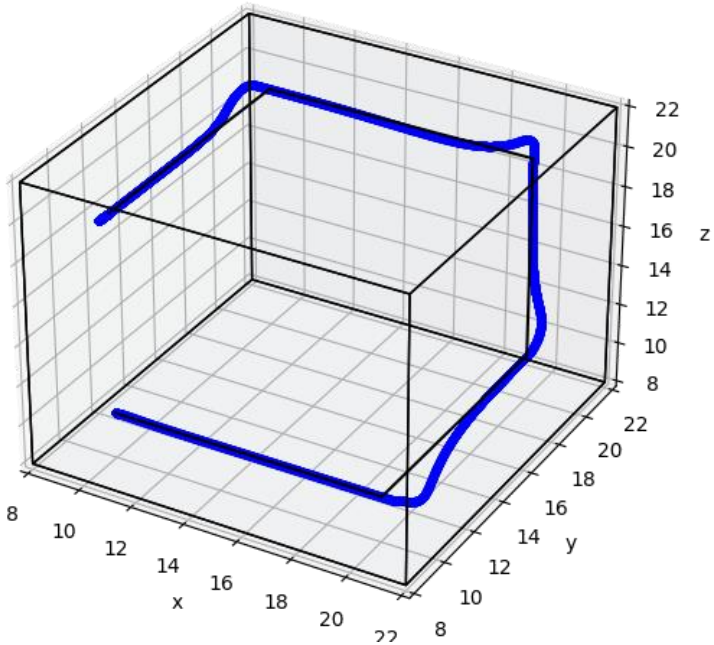
But first…

Project 1-1 Debrief

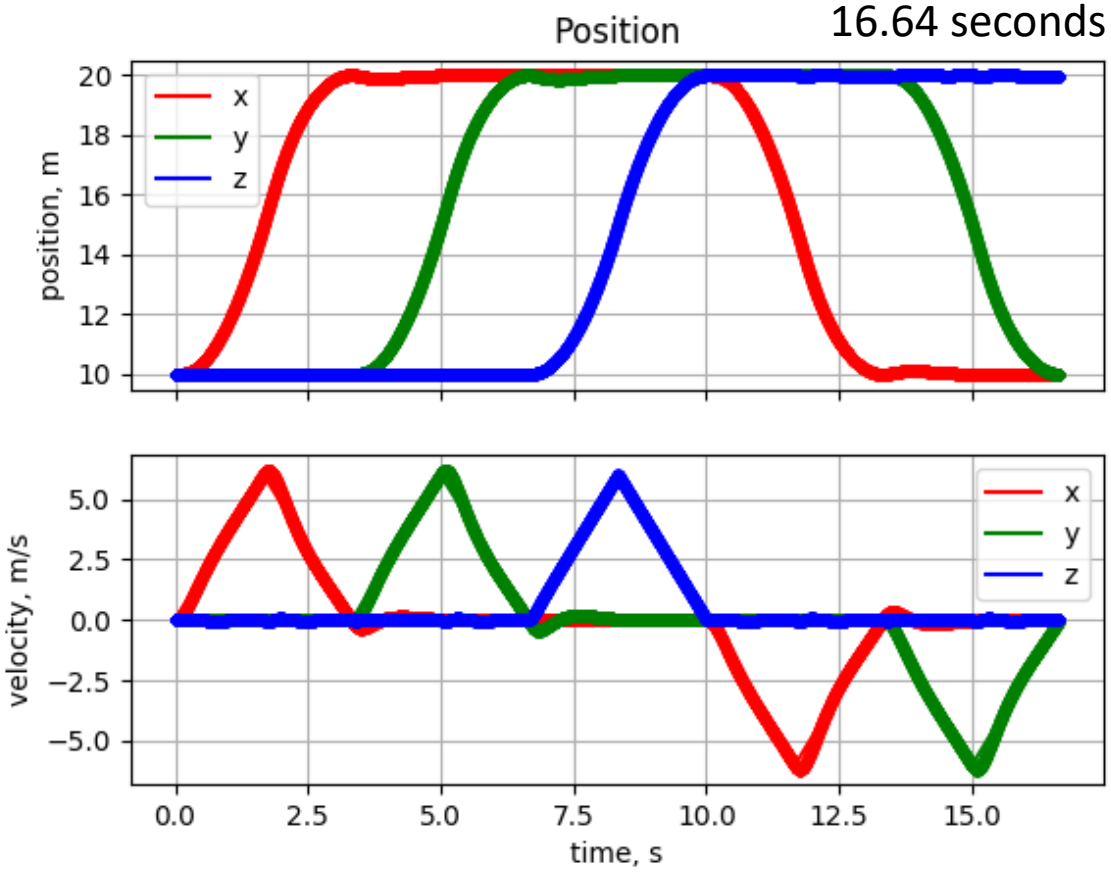
Straight Lines, Constant Speed

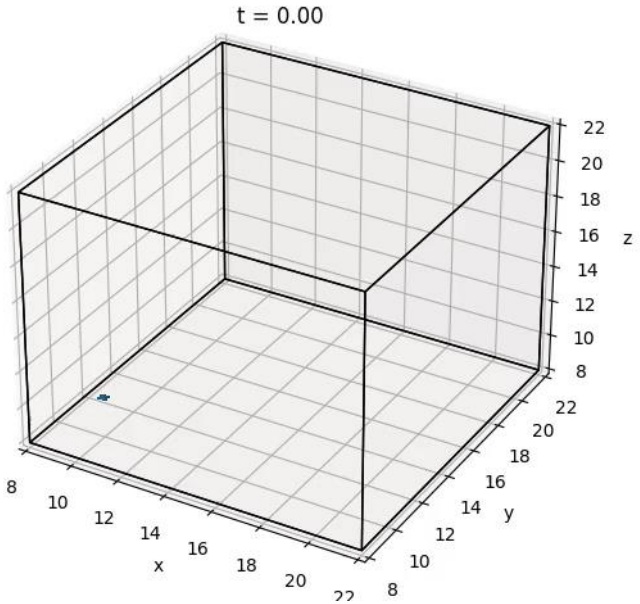


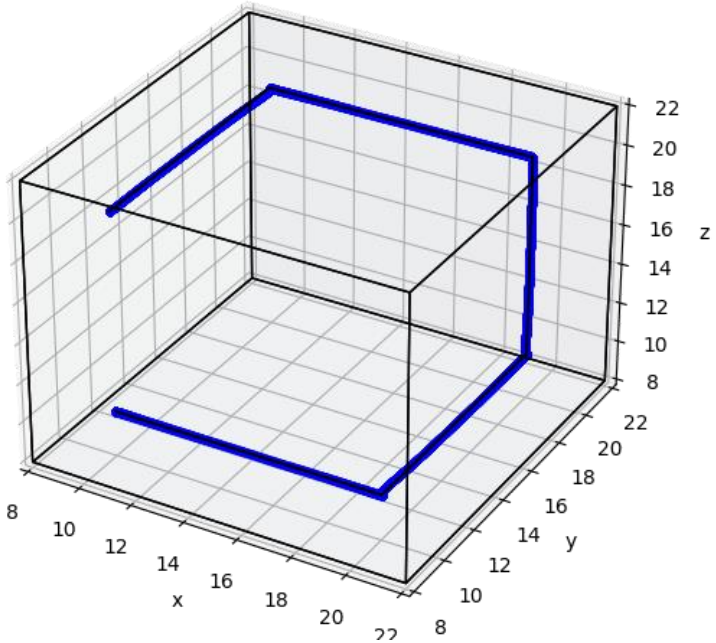




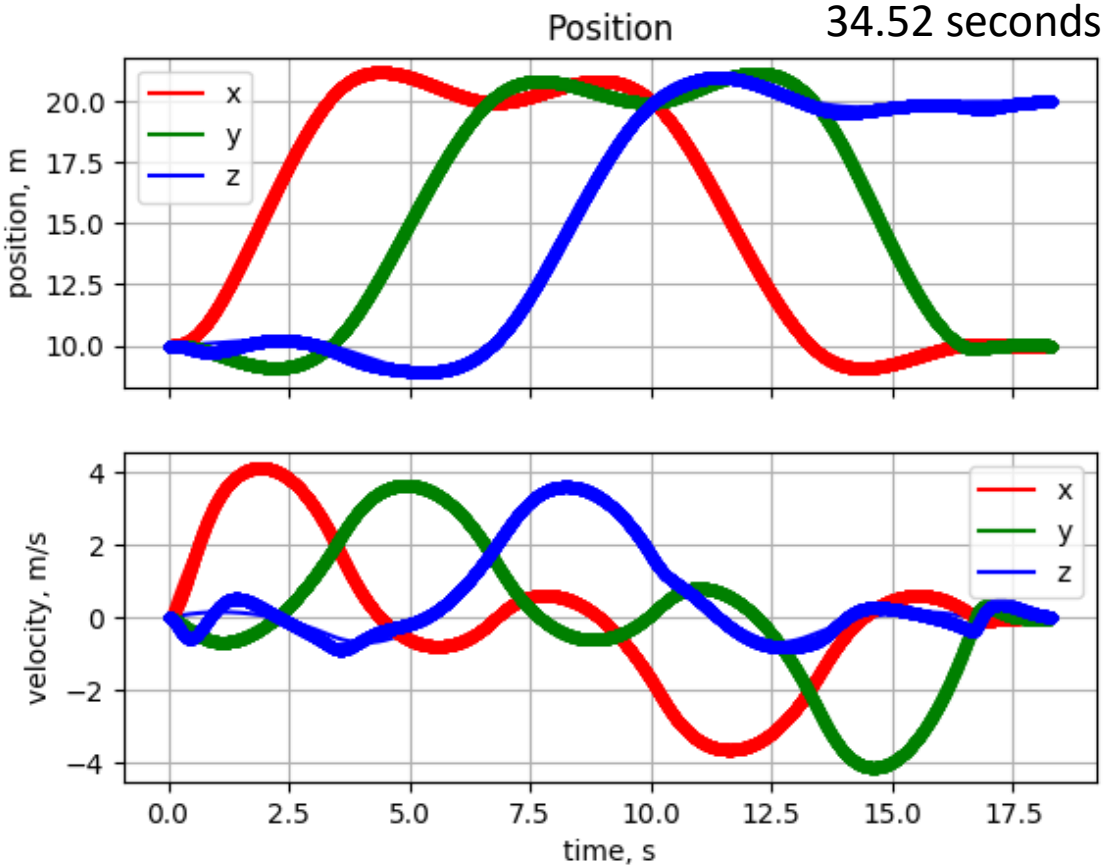
Straight Lines, Constant Acceleration

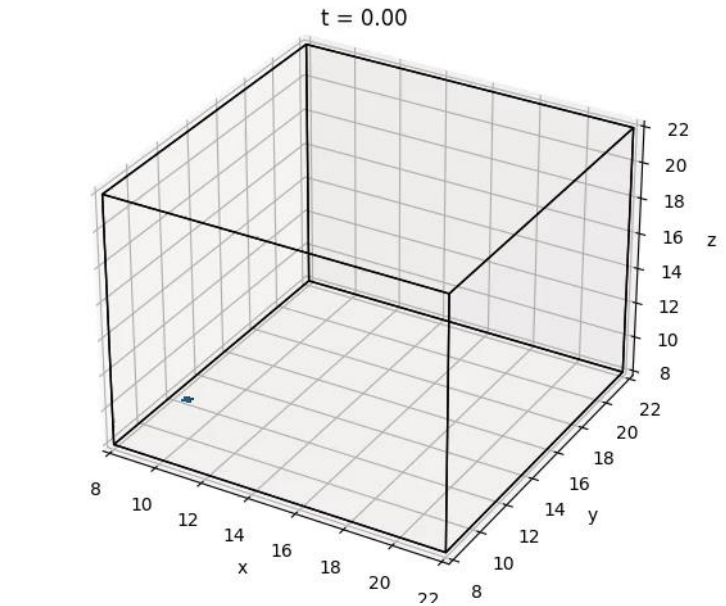


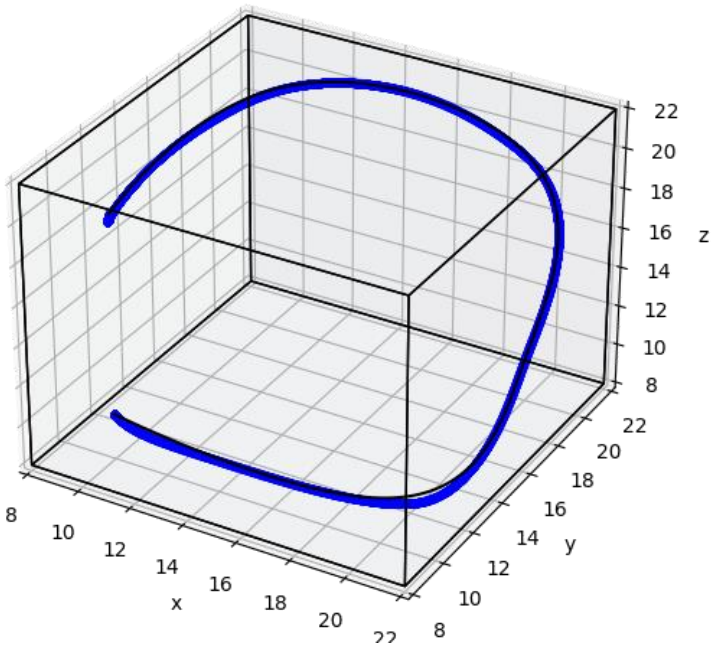




Splines


Position
34.52 seconds
position, m
velocity, m/s
time, s
x
y
z


t = 0.00
x
y
z


x
y
z

Collaboration and Integrity

Your submitted work must be your own work.

Good Collaboration

Easy examples:

- Pointing to a good resource.
- Talking about how to solve part of a problem at a white board.

Bad Collaboration

Easy examples:

- Someone emailing you their code.
- Someone giving you numbers to use.

How to you manage the grey area?

- Give credit to your peers if they helped you.
- Give credit to your sources.

Everyone makes mistakes.

If you forgot to acknowledge people who helped you,
or if you're worried you accepted the "wrong" kind of help,

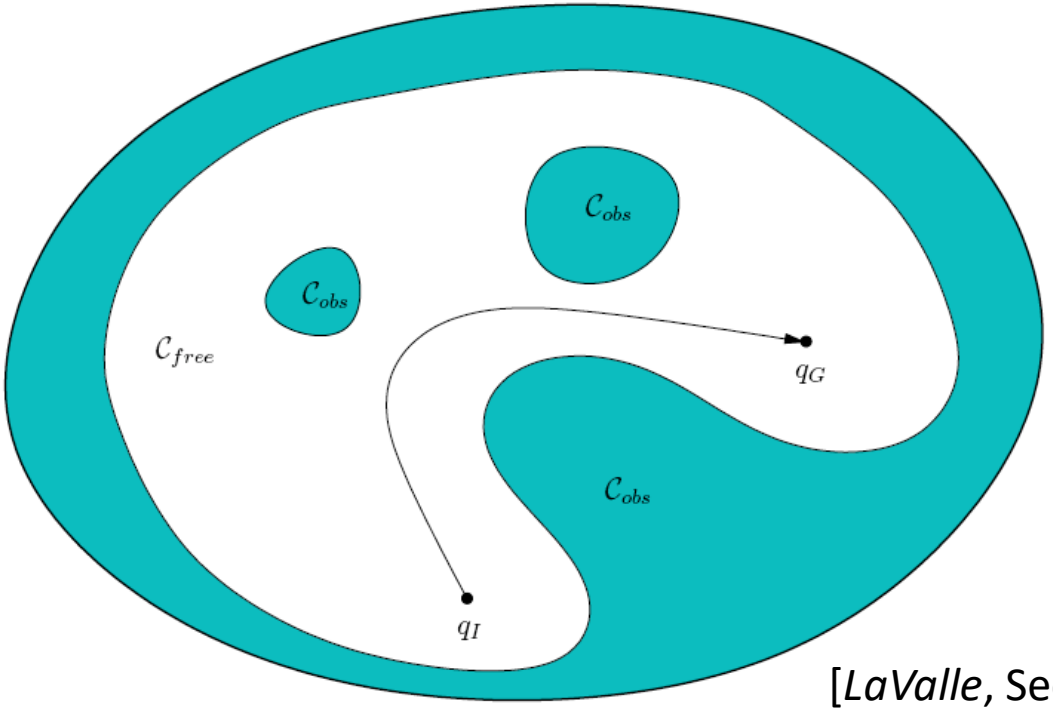
send a private note to 'instructors' on Piazza to credit your collaborators and/or sources.

In the future we will make a 'readme.txt' file mandatory.

Path Planning

The Path Planning Problem

Find a continuous sequence of **configurations** that takes the robot from an initial configuration q_I to a goal configuration q_G such that no configuration along the path intersects an obstacle.



[*LaValle*, Section 4.3.1]

Configuration Space

Configuration Space

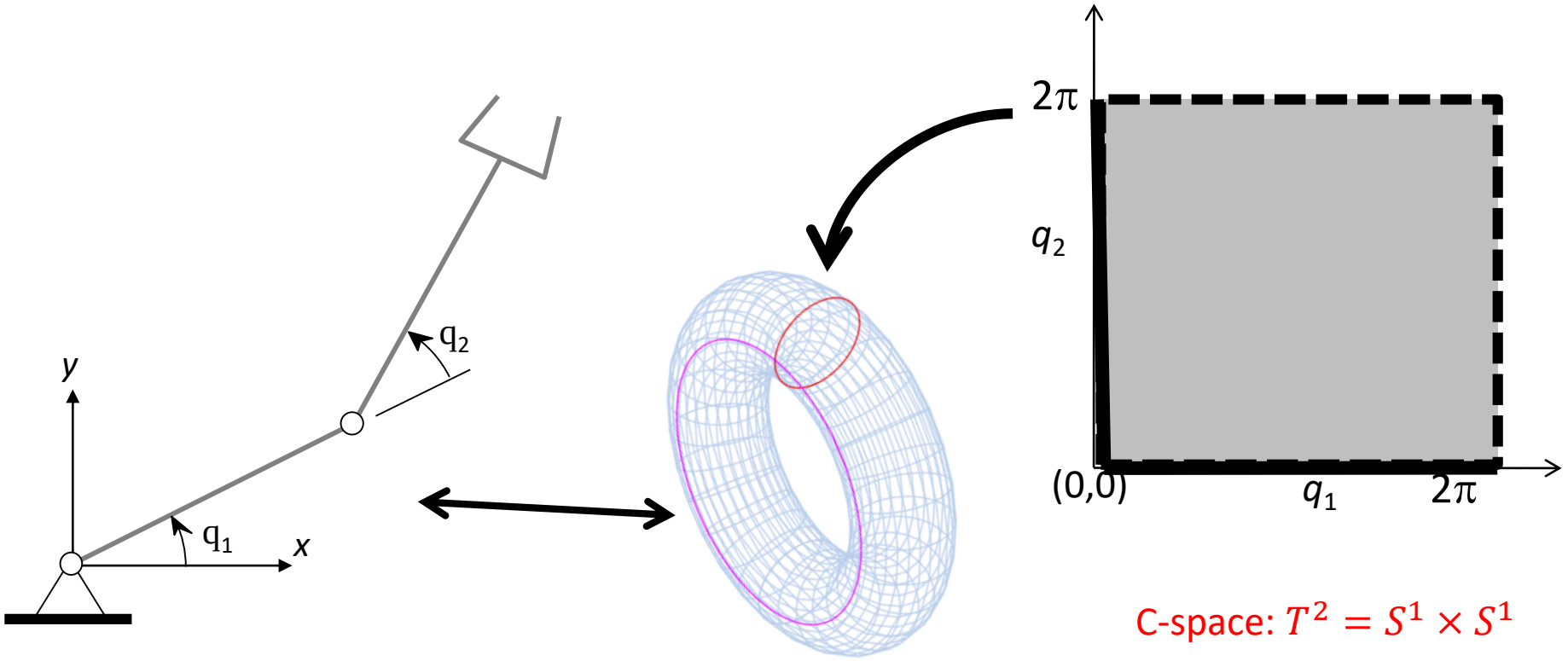
Motion planning is more easily solved in configuration space (C space)

- C is the set of all configurations (positions and orientations) for the system
- $C \neq R^n$ generally

Examples

- Point robot in n dimensional space R^n — R^n
- Rectangular robot in R^3 — $SE(3) = R^3 \times SO(3)$
- Fixed robot with 2 revolute joints — $T^2 = S^1 \times S^1$

Example

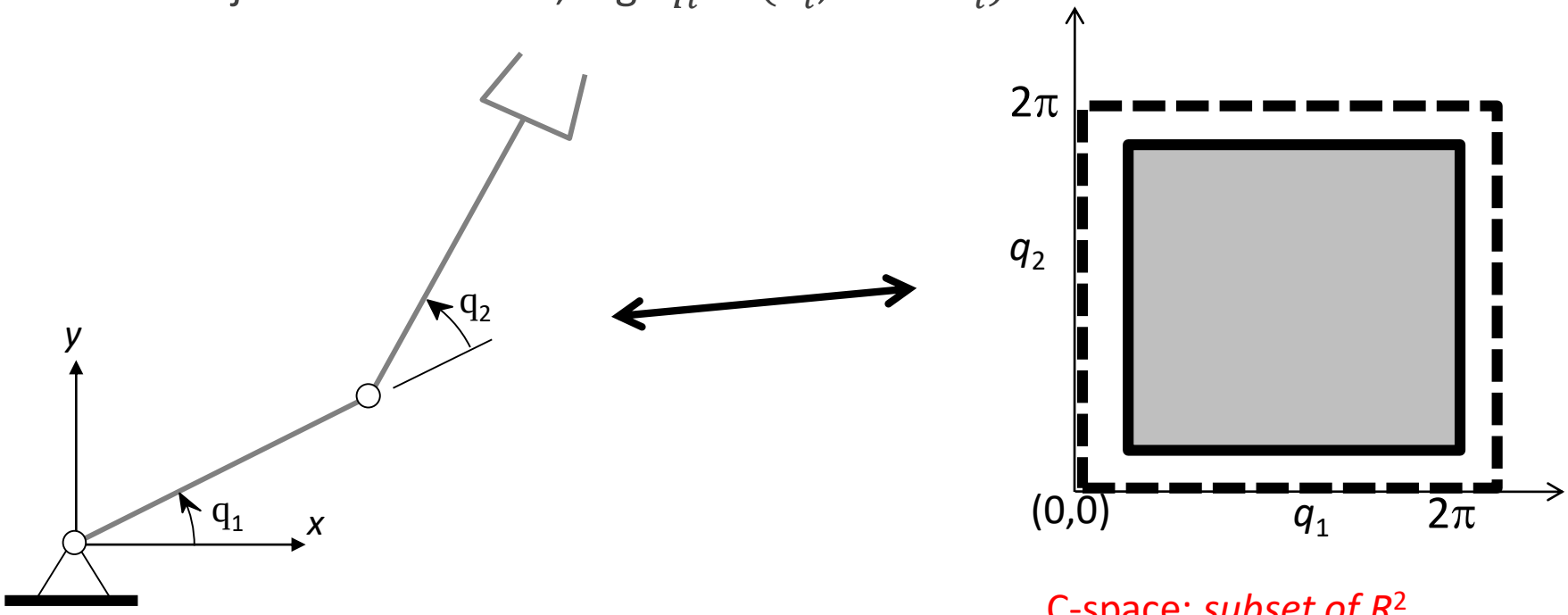


C-space: $T^2 = S^1 \times S^1$

The configuration space *locally* looks like R^2, and hence is a 2-D *manifold*

Example with Joint Limits

If the joints have limits, e.g. $q_i \in (\theta_i, 2\pi - \theta_i)$



C-space: *subset of R^2*

Modeling the C-Space

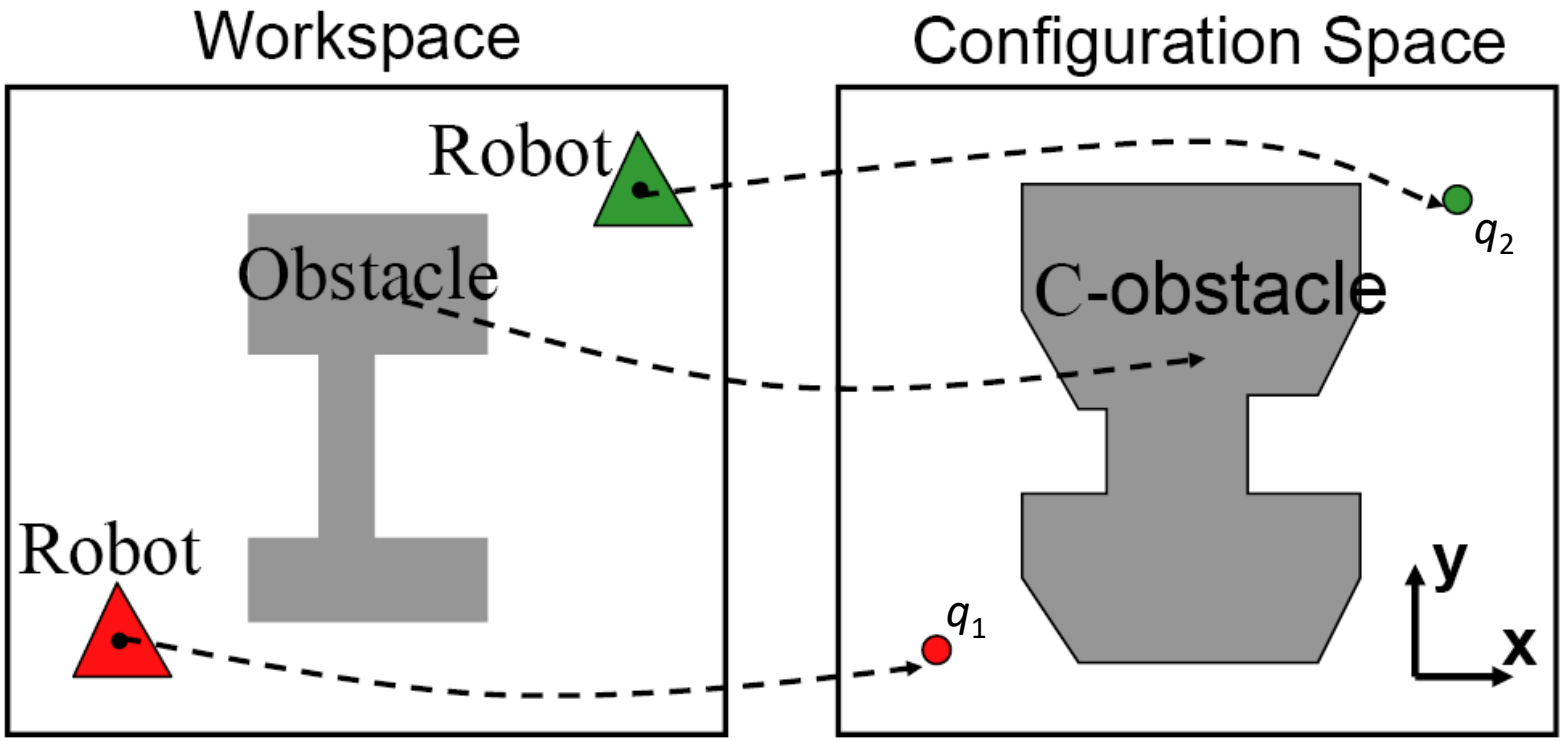


Figure by Dinesh Manocha, UNC

- **Note**: This robot can only translate in $W \subset \mathbb{R}^2$ hence $C \subset \mathbb{R}^2$ as well.
- **How do we construct C^{obs}?**

Modeling C^{obs} Geometric Intuition

Polygon robot that does not rotate.

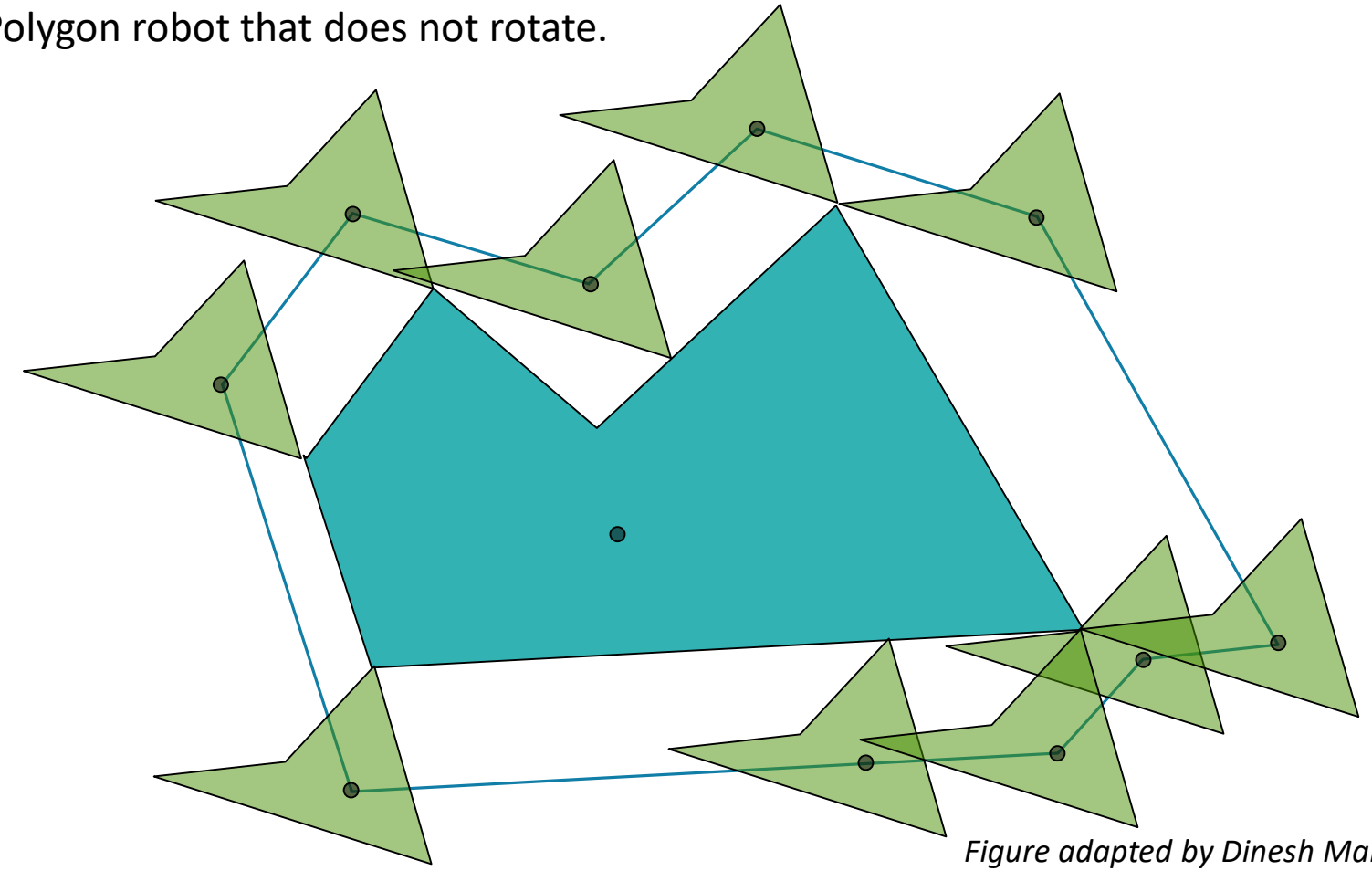


Figure adapted by Dinesh Manocha, UNC

Modeling C^{obs} Geometric Intuition

Polygon robot that does not rotate.

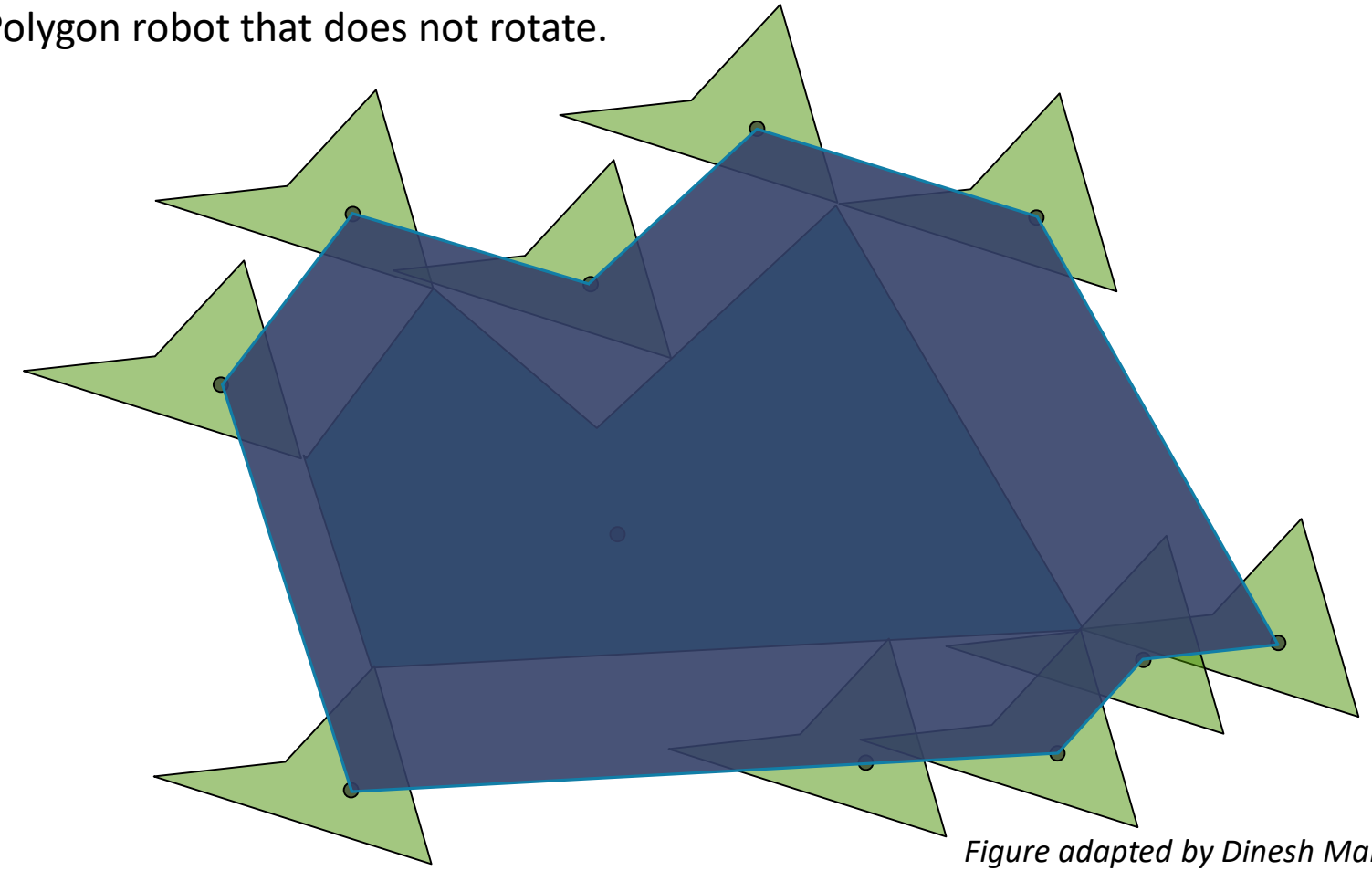


Figure adapted by Dinesh Manocha, UNC

Definition: Minkowski Sum

- The Minkowski sum of two sets, A and B is

$$A \oplus B = \{a + b | \ a \in A, b \in B\}$$

$$A \oplus B = \{a + B | \ a \in A\}$$

- Notice the origin, these are **vector** additions

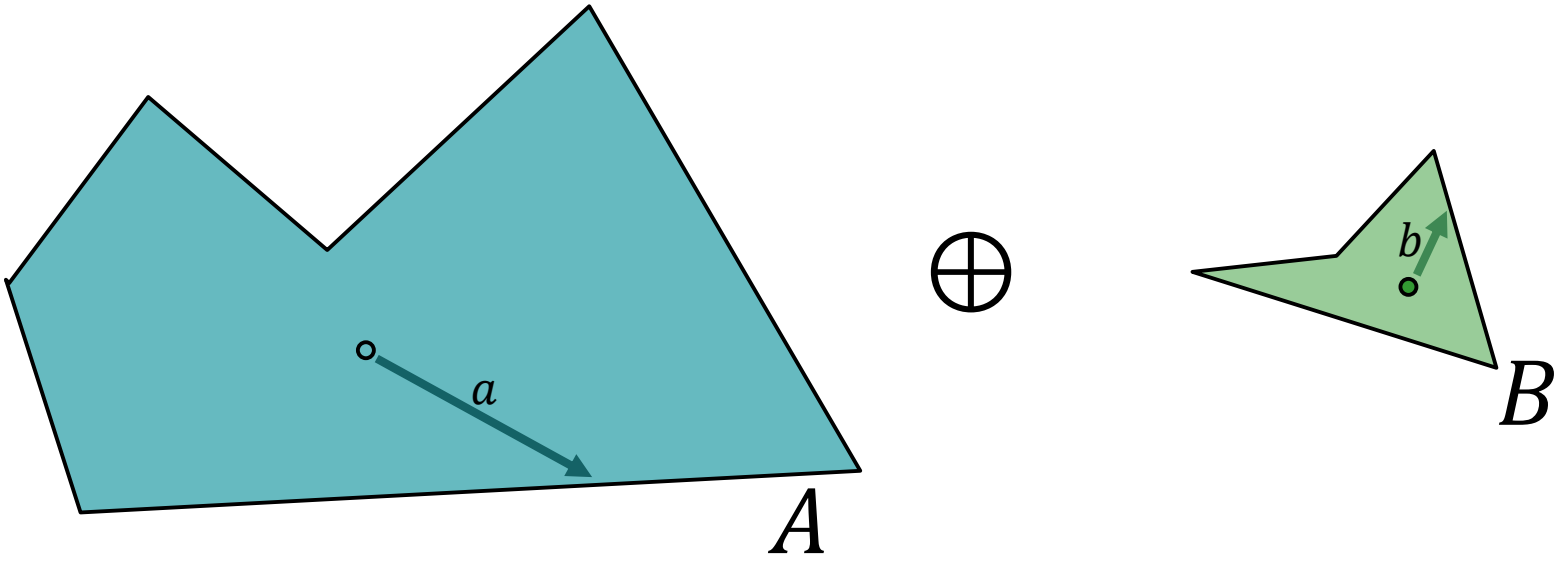


Figure by Dinesh Manocha, UNC

Minkowski Sum

- The Minkowski sum of two sets, A and B is

$$A \oplus B = \{a + b | \ a \in A, b \in B\}$$

$$A \oplus B = \{a + B | \ a \in A\}$$

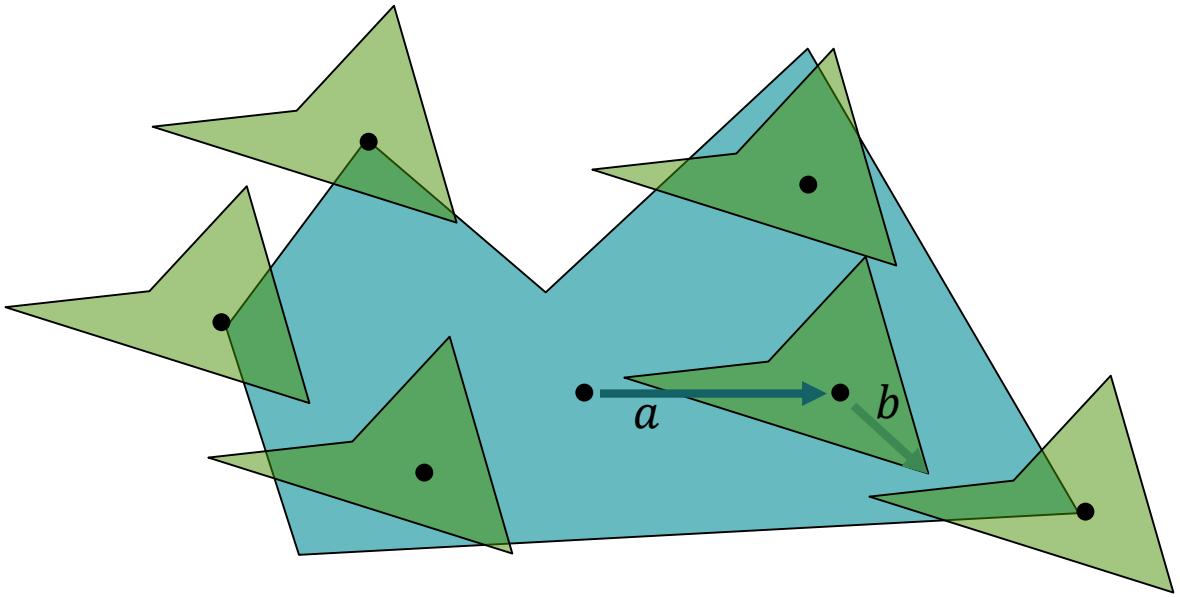


Figure by Dinesh Manocha, UNC

Minkowski Sum

- The Minkowski sum of two sets, A and B is

$$A \oplus B = \{a + b | \ a \in A, b \in B\}$$

$$A \oplus B = \{a + B | \ a \in A\}$$

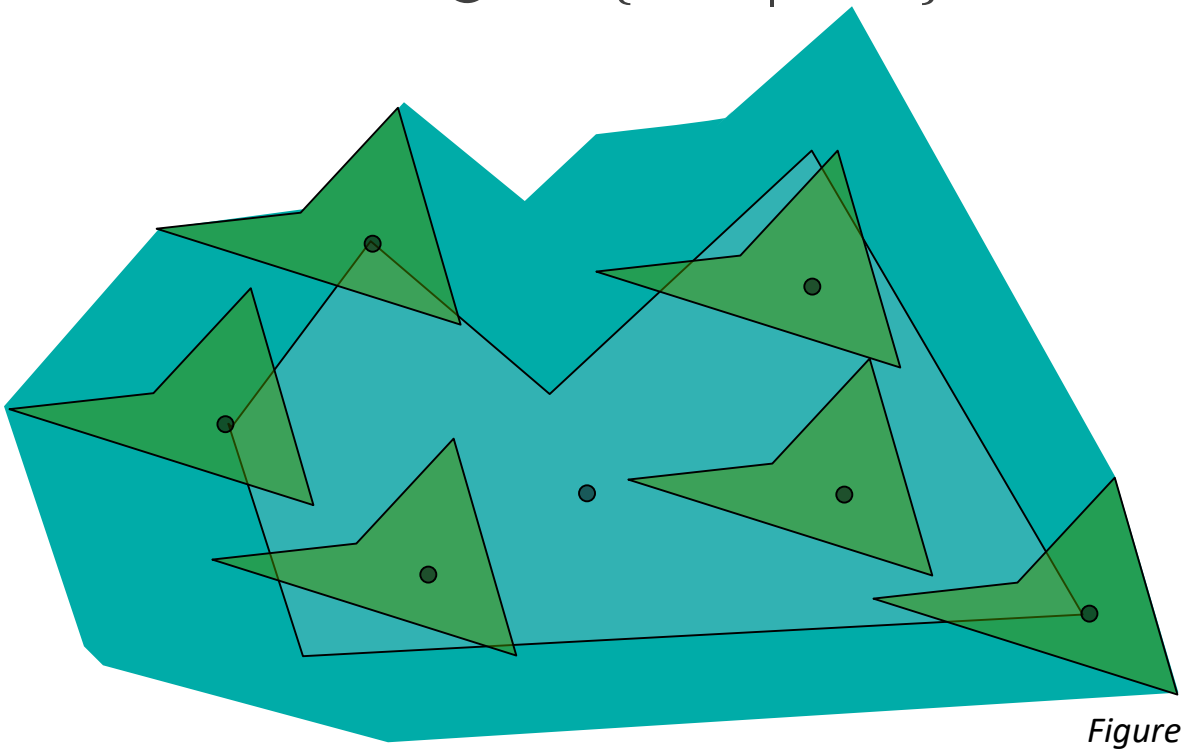


Figure by Dinesh Manocha, UNC

Minkowski Sum

- The Minkowski sum of two sets, A and B is

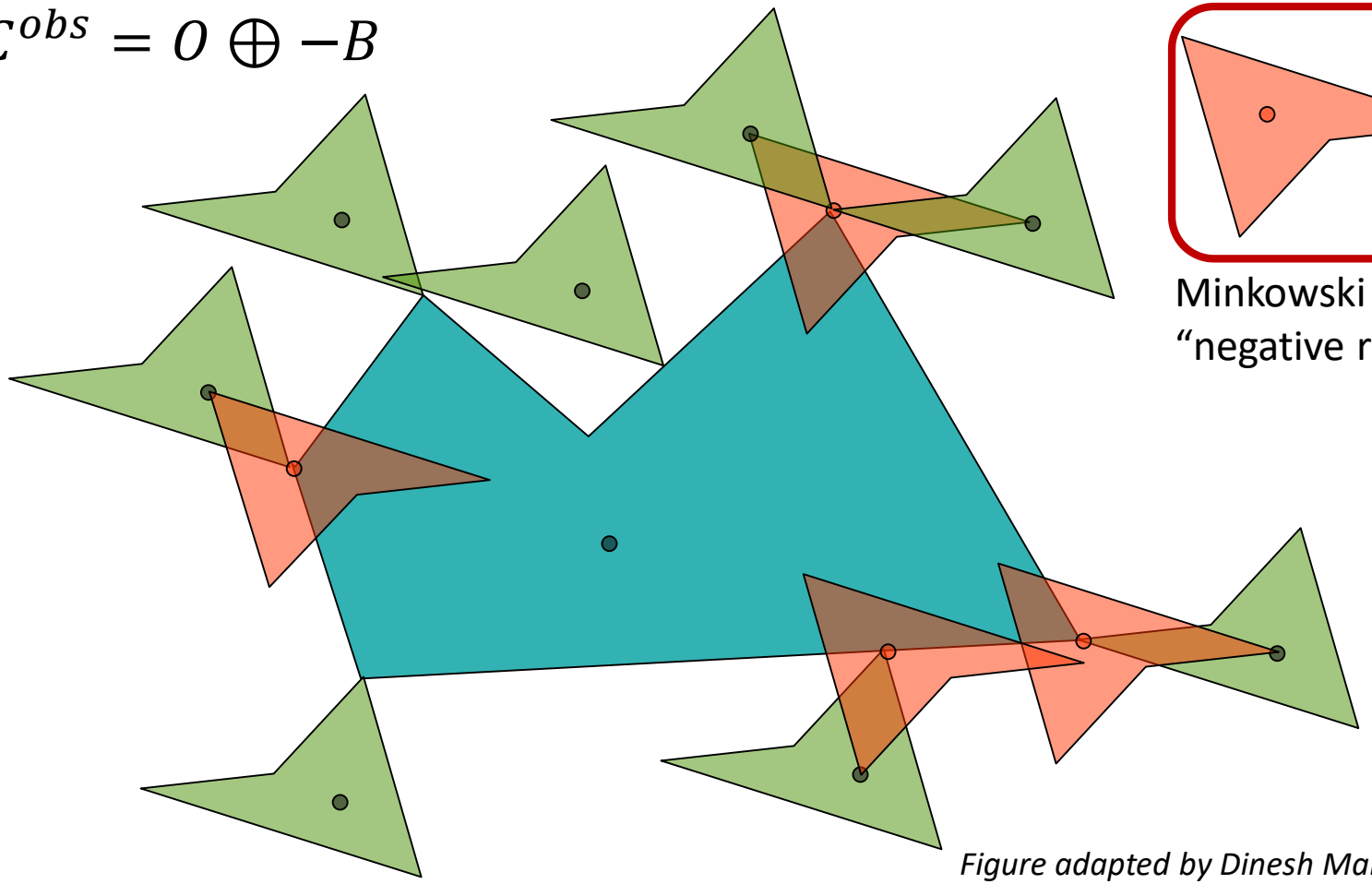
$$A \oplus B = \{a + b | \ a \in A, b \in B\}$$

$$A \oplus B = \{a + B | \ a \in A\}$$

Figure by Dinesh Manocha, UNC

Modeling C^{obs} Using Minkowski Sum

$C^{obs} = O \oplus -B$



Minkowski sum with "negative robot."

Figure adapted by Dinesh Manocha, UNC

Modeling C^{obs} Using Minkowski Sum

$C^{obs} = O \oplus -B$

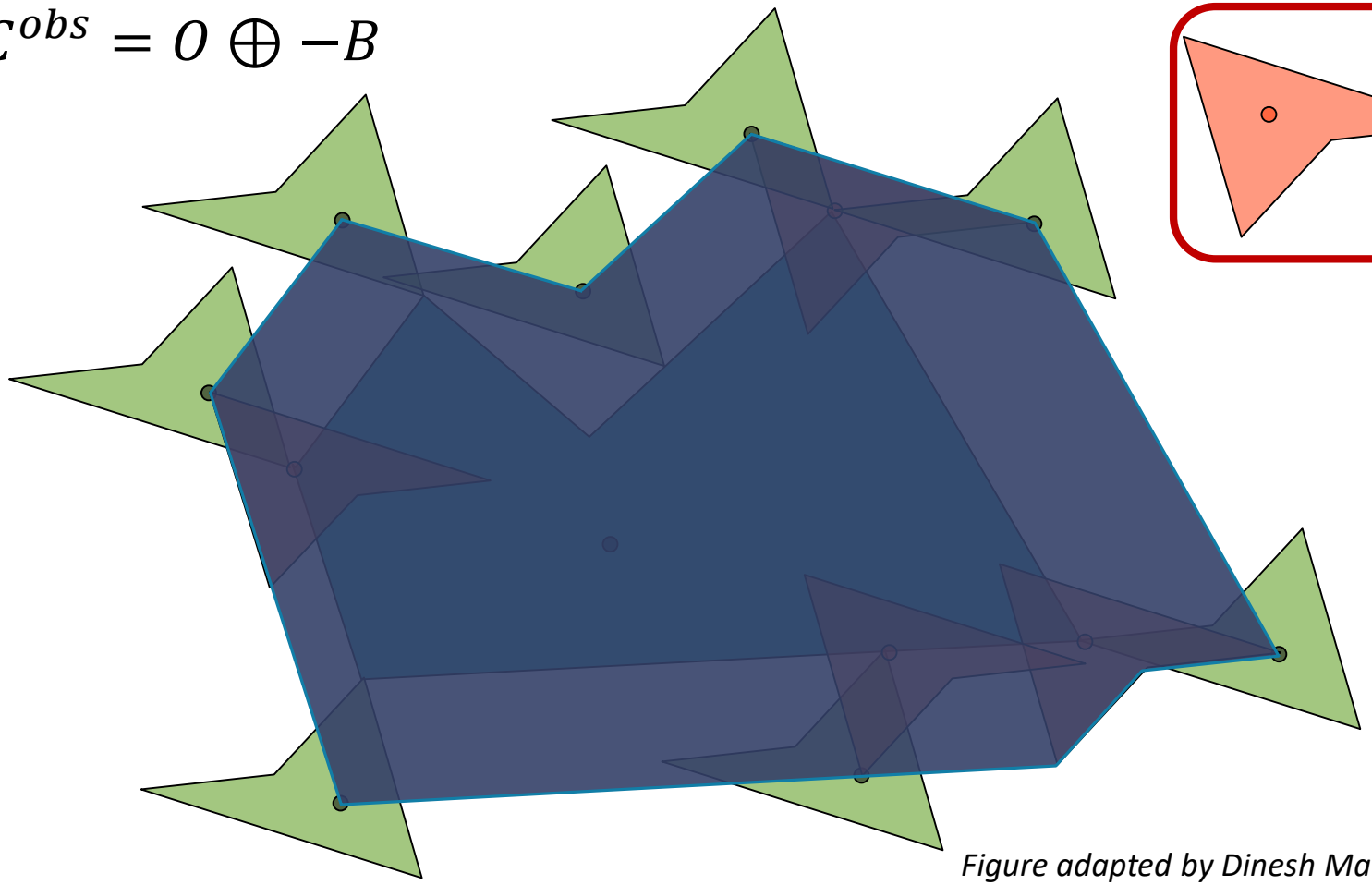
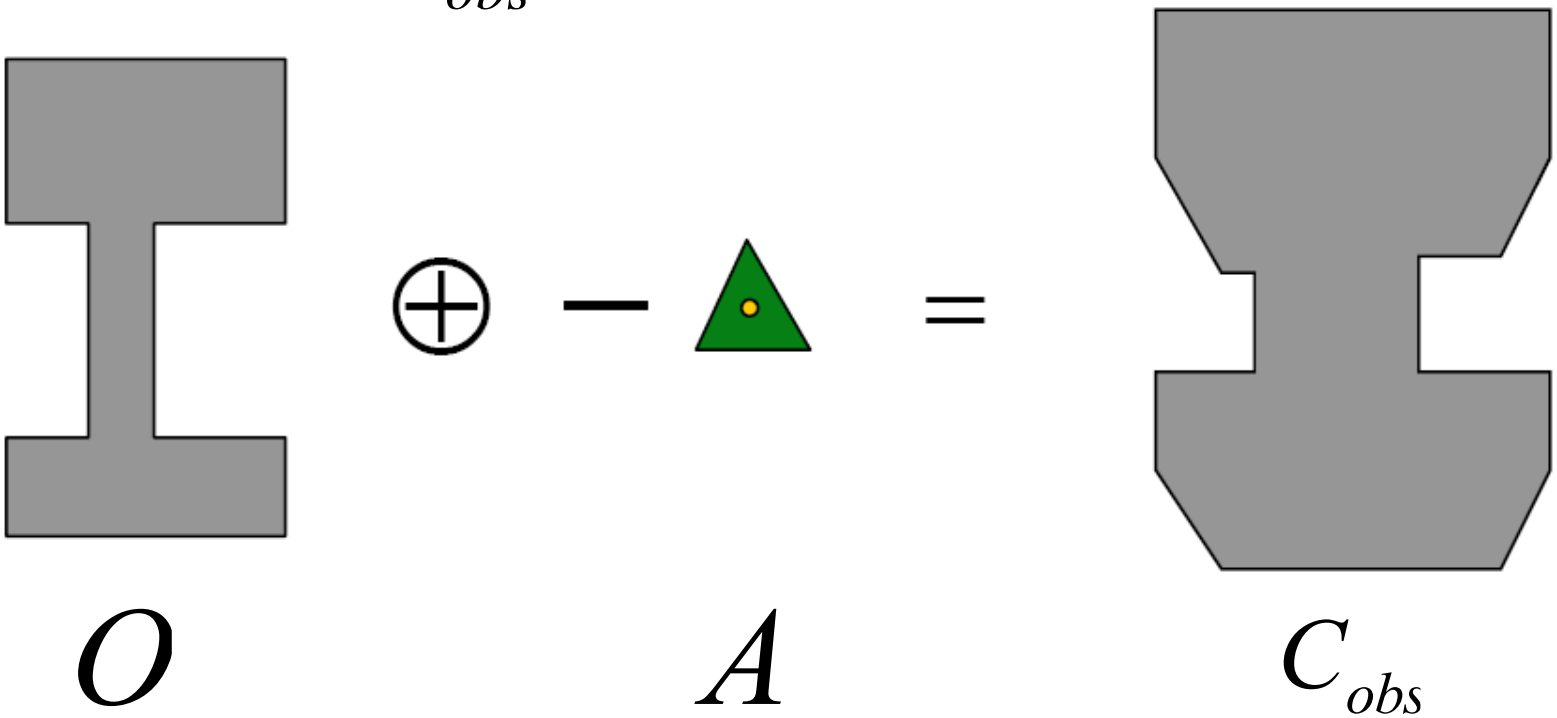


Figure adapted by Dinesh Manocha, UNC

C^{obs}

$$C_{obs} = O \oplus -A$$



O A C_{obs}

Complexity

O, a 2D convex polygon with m vertices

A, a 2D convex polygon with n vertices

Minkowski sum is a convex polygon of $m + n$ vertices

- Run time $\sim O(n + m)$

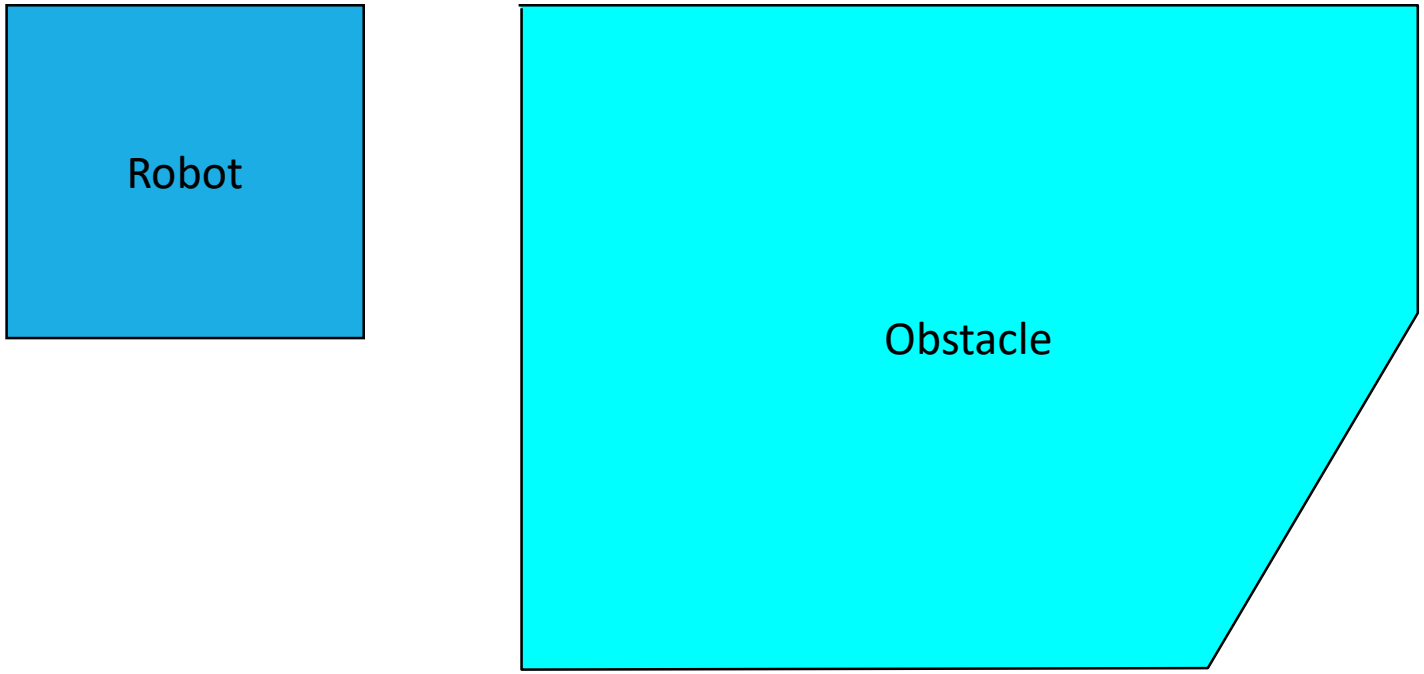
Nonconvex case is much harder

- Decompose into convex polygons
- Compute Minkowski sums
- Take unions
- Run time $\sim O(n^2 m^2)$

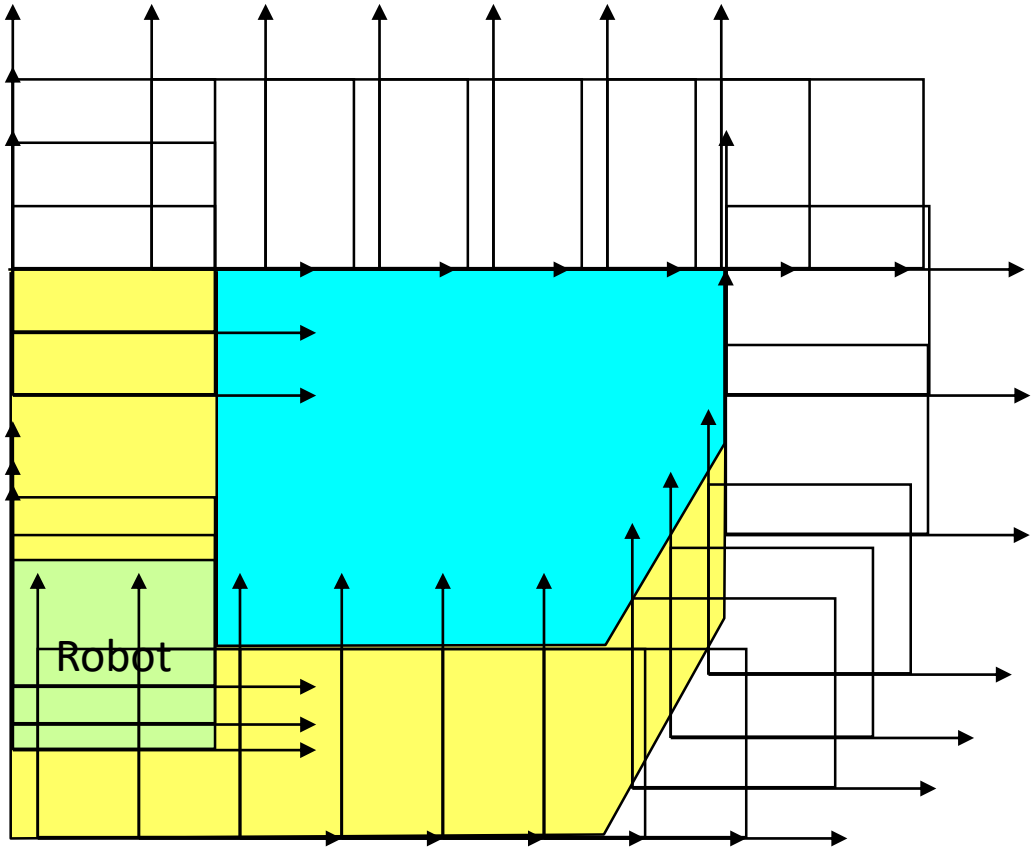
In 3D, run time $\sim O(n^3 m^3)$

Now Add Rotations

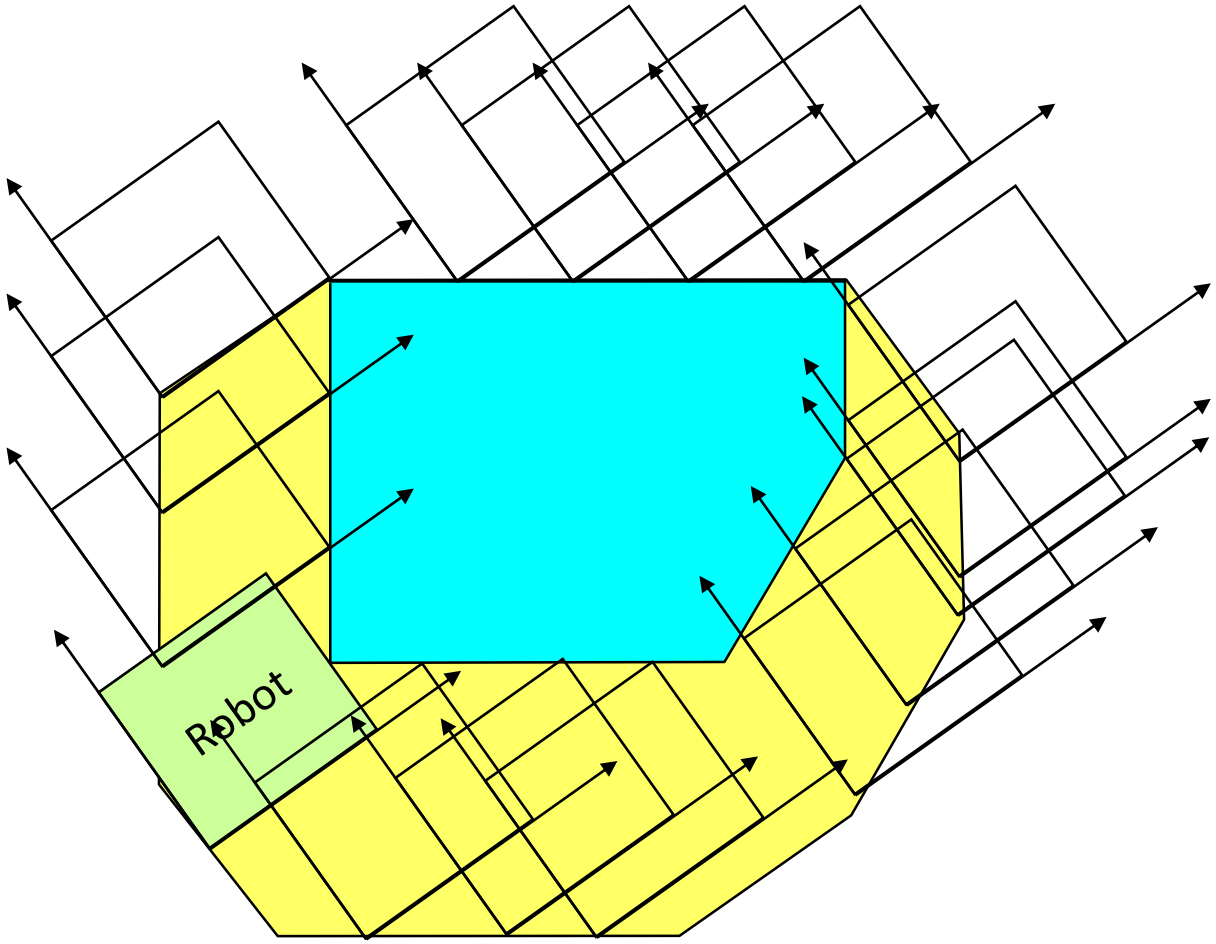
What if the rigid body can translate *and* rotate?



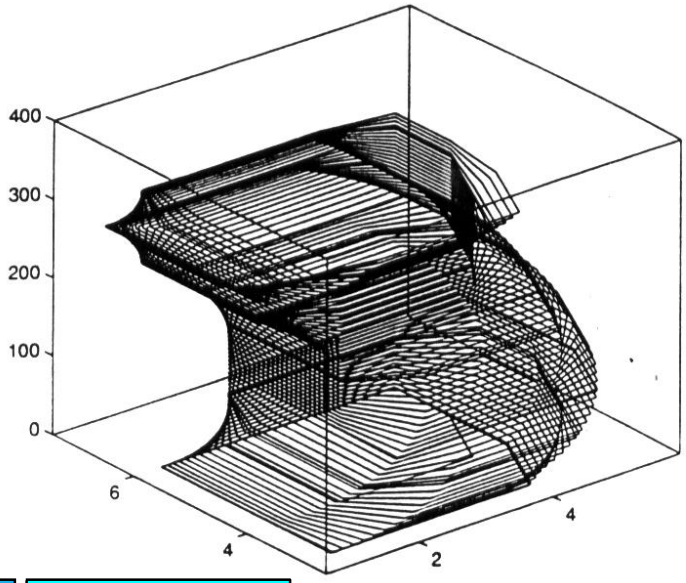
C^{obs} at $\theta = 0$

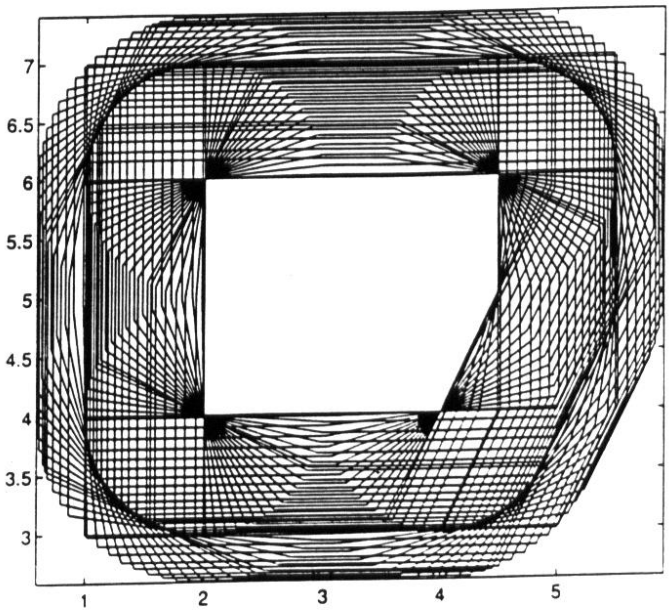


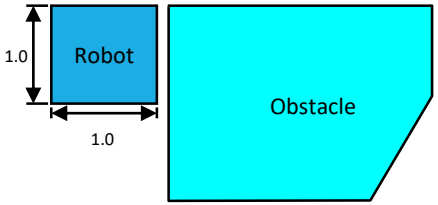
C^{obs} at $\theta \neq 0$



Slices of C^{obs}

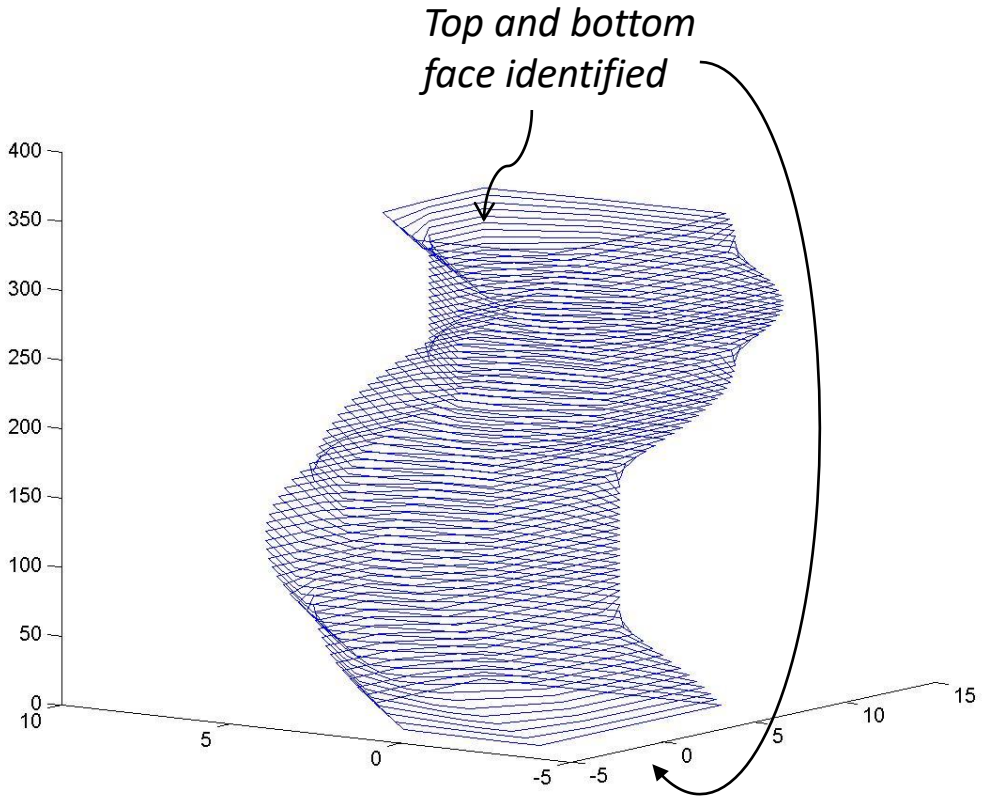






Each slice is a convex polygon of $\leq 4 + 5$ vertices

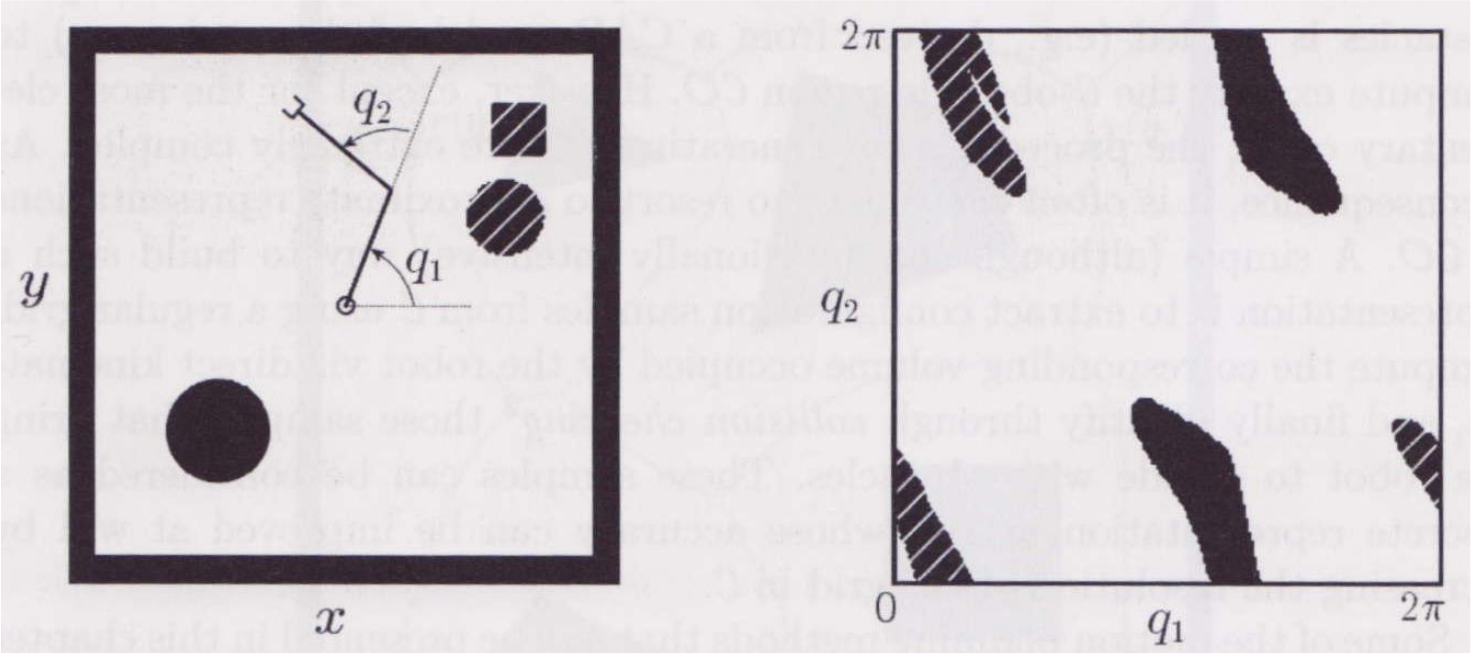
C^{obs} with Rotations



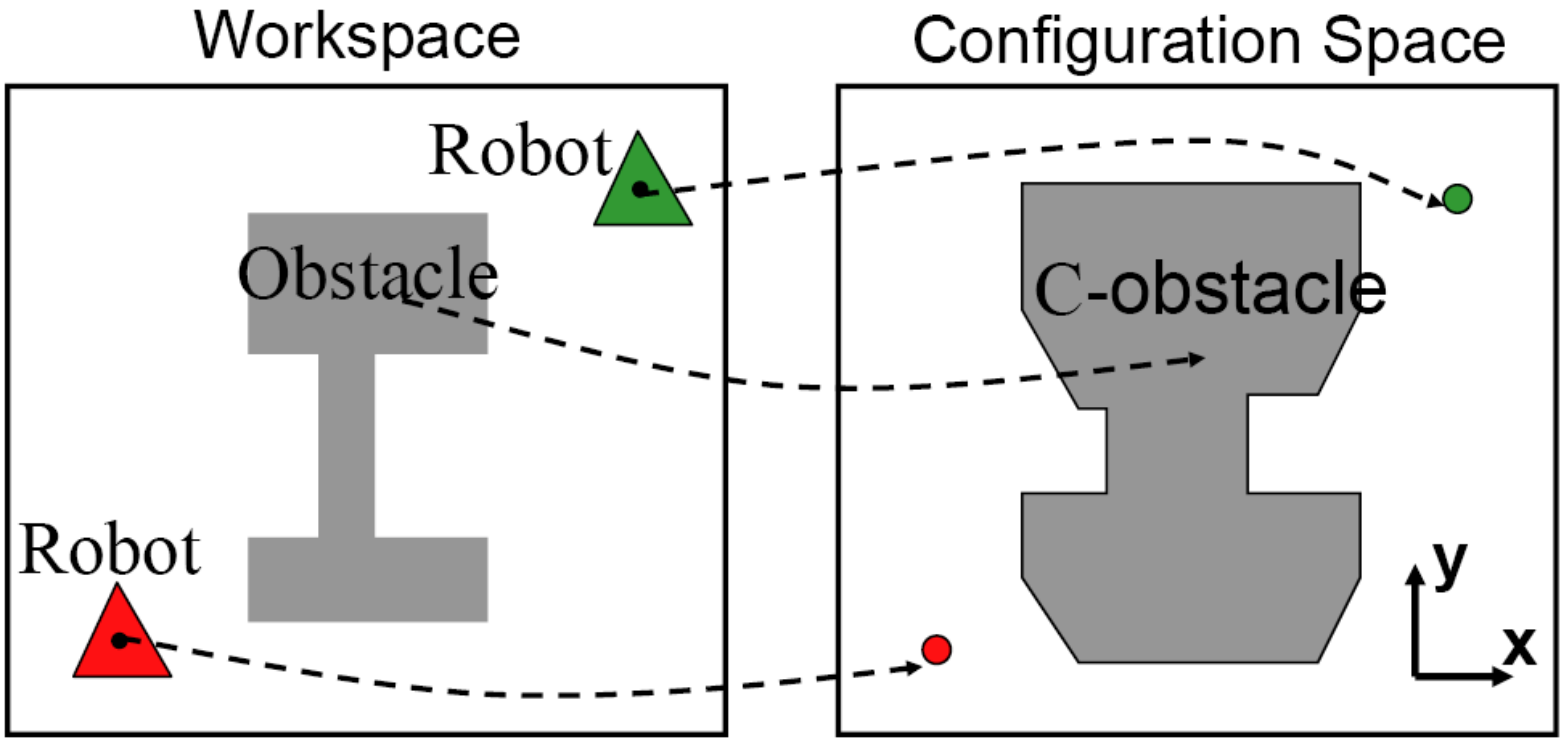
LaValle (3.1.1, 4.3)

C^{obs} of Robot Arms

- The configuration space obstacle region can take strange and complicated forms



[Siciliano, Sciavicco, Villani, Oriolo, 09]



Tasks are described in the workspace.

Plan in Configuration Space

Path Planning Approaches

Challenges

Designing a complete motion/path planning algorithm

- A **complete** algorithm finds a solution if one exists and reports no otherwise

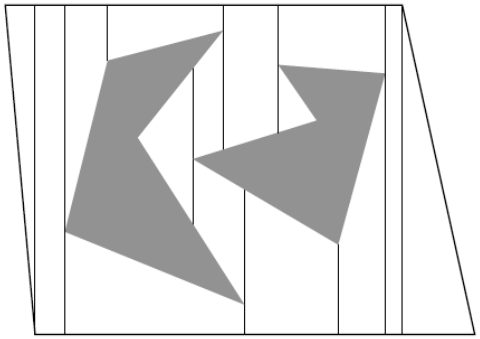
Many challenges

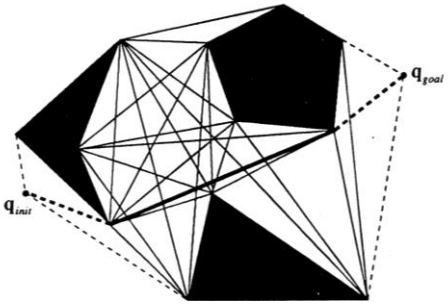
- Dimensionality (2/3 for point robots in Euclidean space, 3/6 with orientation)
- Obstacles – potentially narrow corridors
- Articulated chains (robot arms)
- Topology – C^{free} can be very complicated
- Representation – C^{free} must be stored with limited memory and computation
- Motion constraints – kinematic or dynamic
- Actuator limits
- ⋮

Three Techniques

Cellular methods

- Represent C^{free} as a set of non-overlapping cells
- Can be exact or approximate



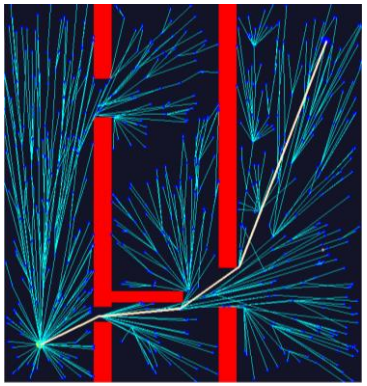


Roadmap methods

- Represent C^{free} using a roadmap

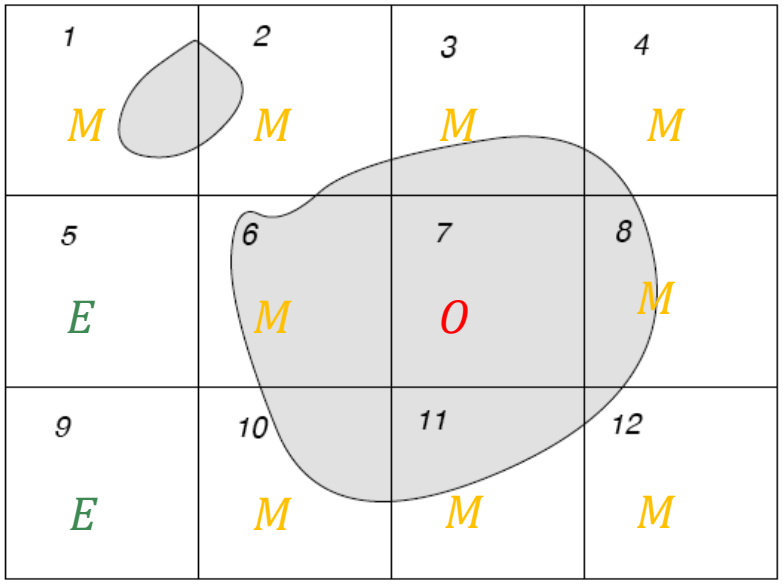
Probabilistic / Sampling methods

- Perhaps avoid explicit representation of C^{free} altogether.

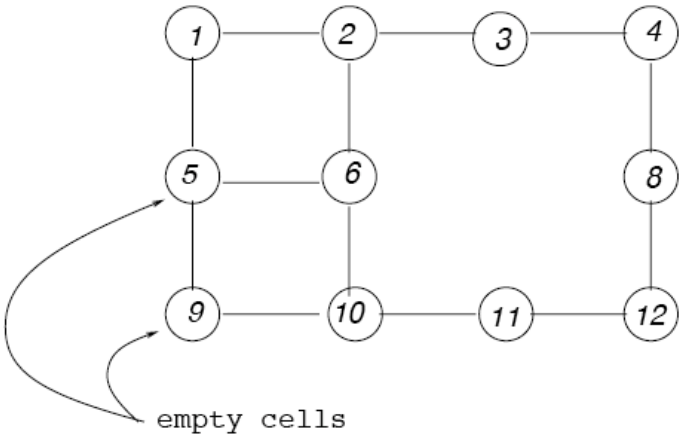


Cellular Methods

Approximate Cell Decomposition



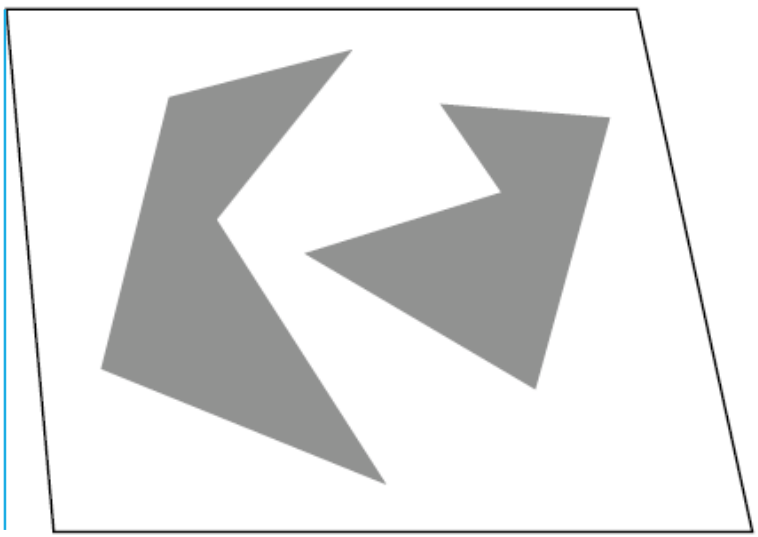
E – empty
M – mixed
O – occupied

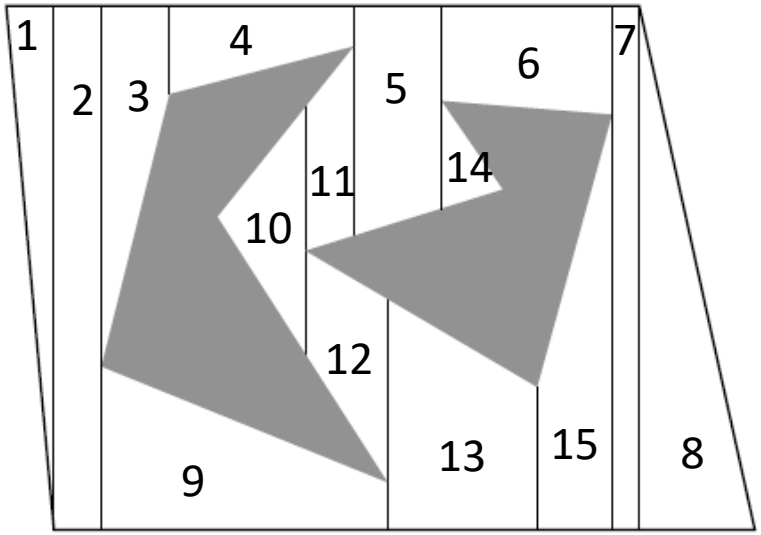


Example by Nancy Amato, Texas A&M

Exact Cell Decomposition

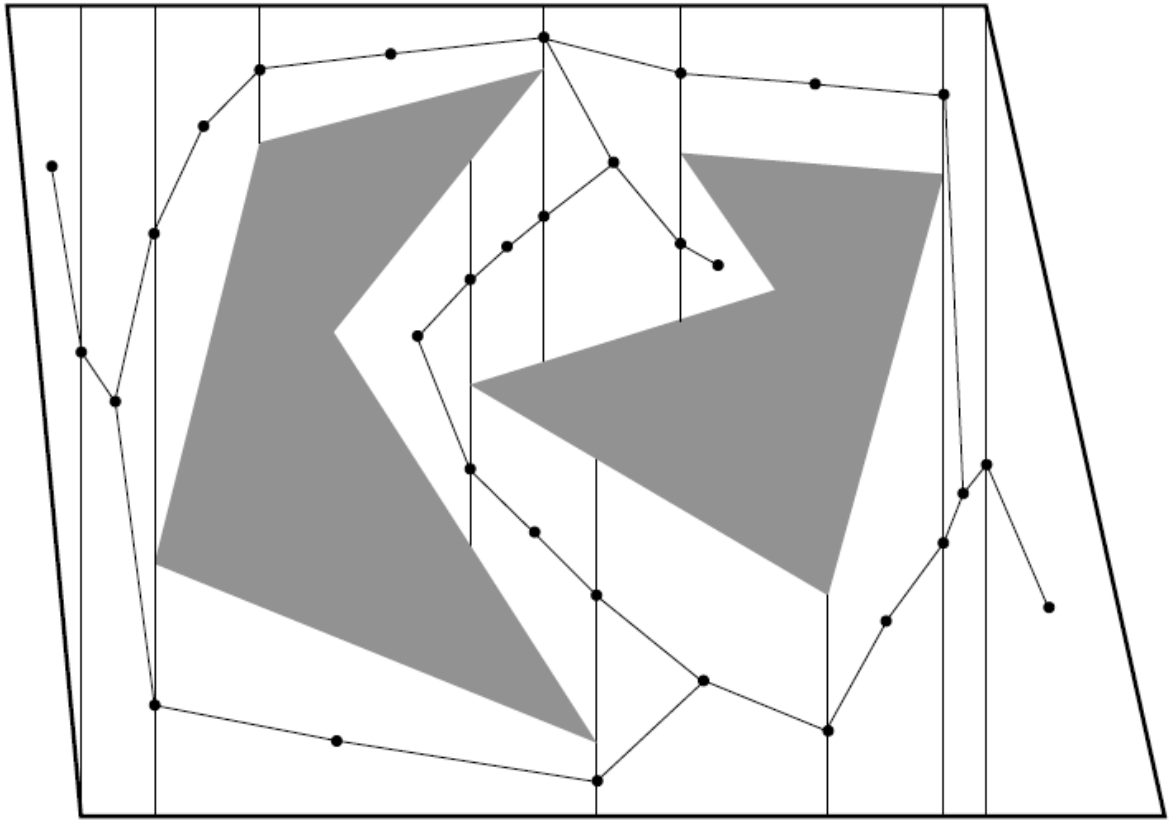
Example: Vertical Cell Decomposition in R^2





Roadmaps

Roadmap from Cells



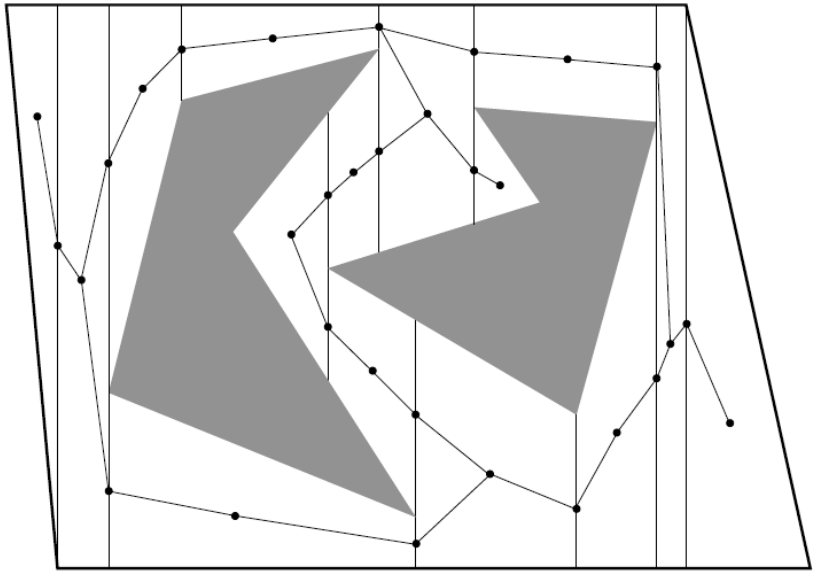
Roadmap Features

Accessibility: Point q_{start} can be connected to some point q' in the roadmap.

Departability: Some point q'' in the roadmap can be connected to q_{goal}.

Connectivity: There exists a path in the roadmap from q' to q''.

In this example, use of convex cells ensure accessibility and departability using straight line paths.

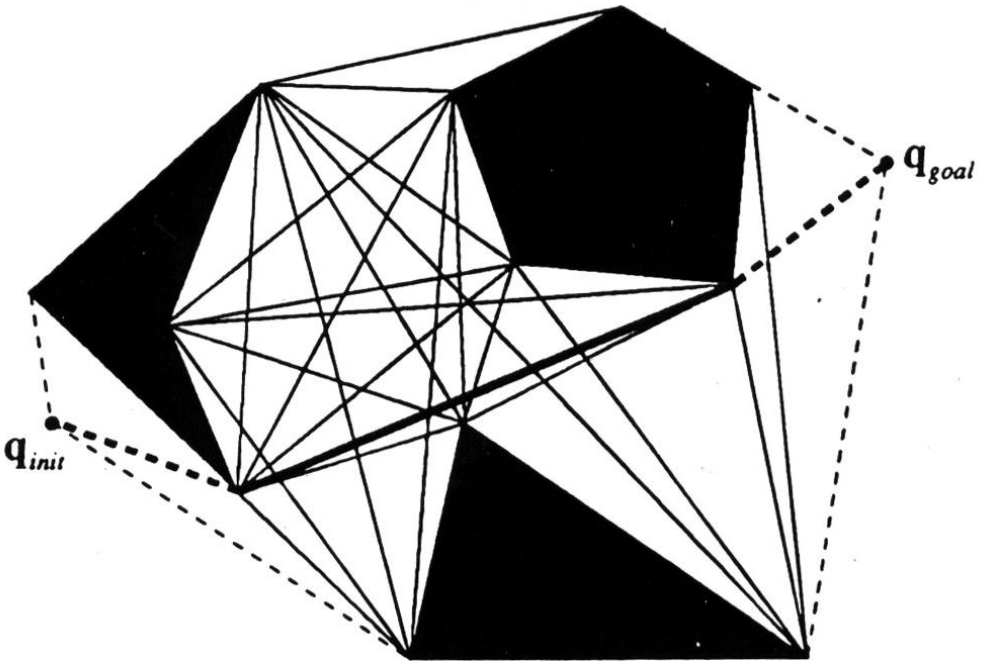


Visibility Roadmaps

Join all pairs of vertices, plus initial and final starting position.

Eliminate edges that intersect obstacles.

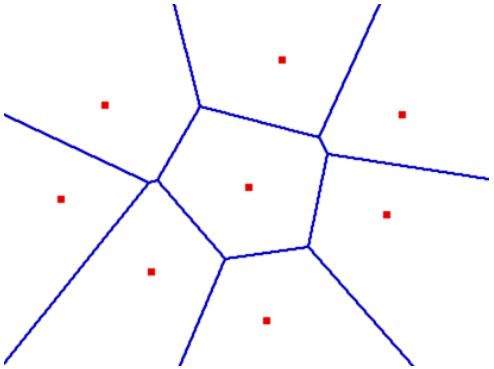
Roadmap always includes shortest path through the configuration space.



[Latombe 91]

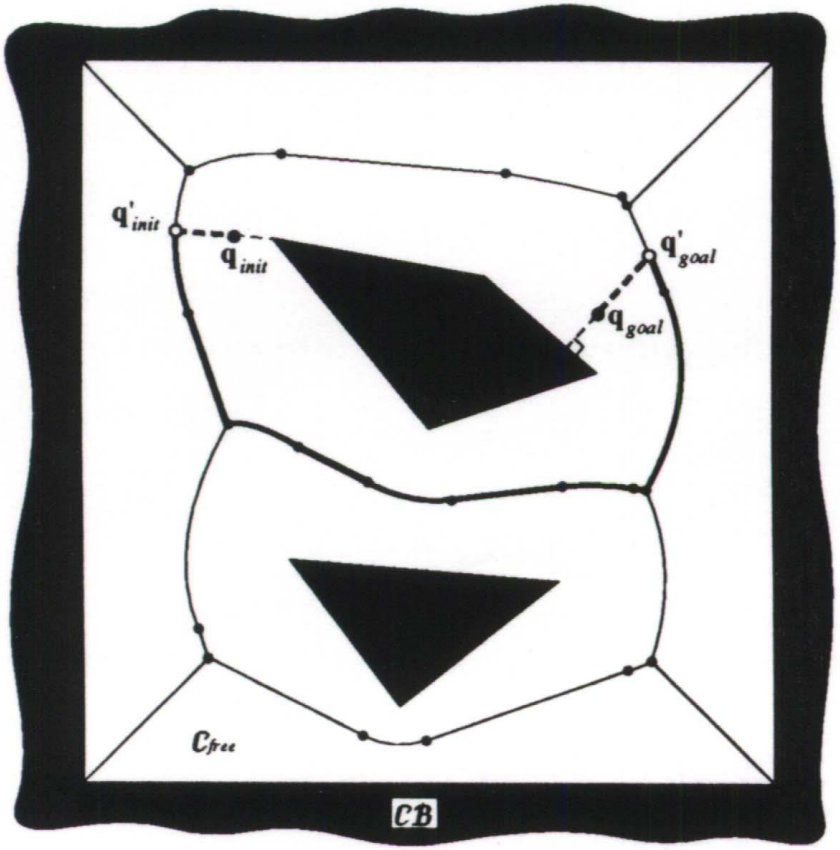
Voronoi Diagram Roadmaps

Inspiration: The Voronoi diagram depicts regions nearest to a given point.

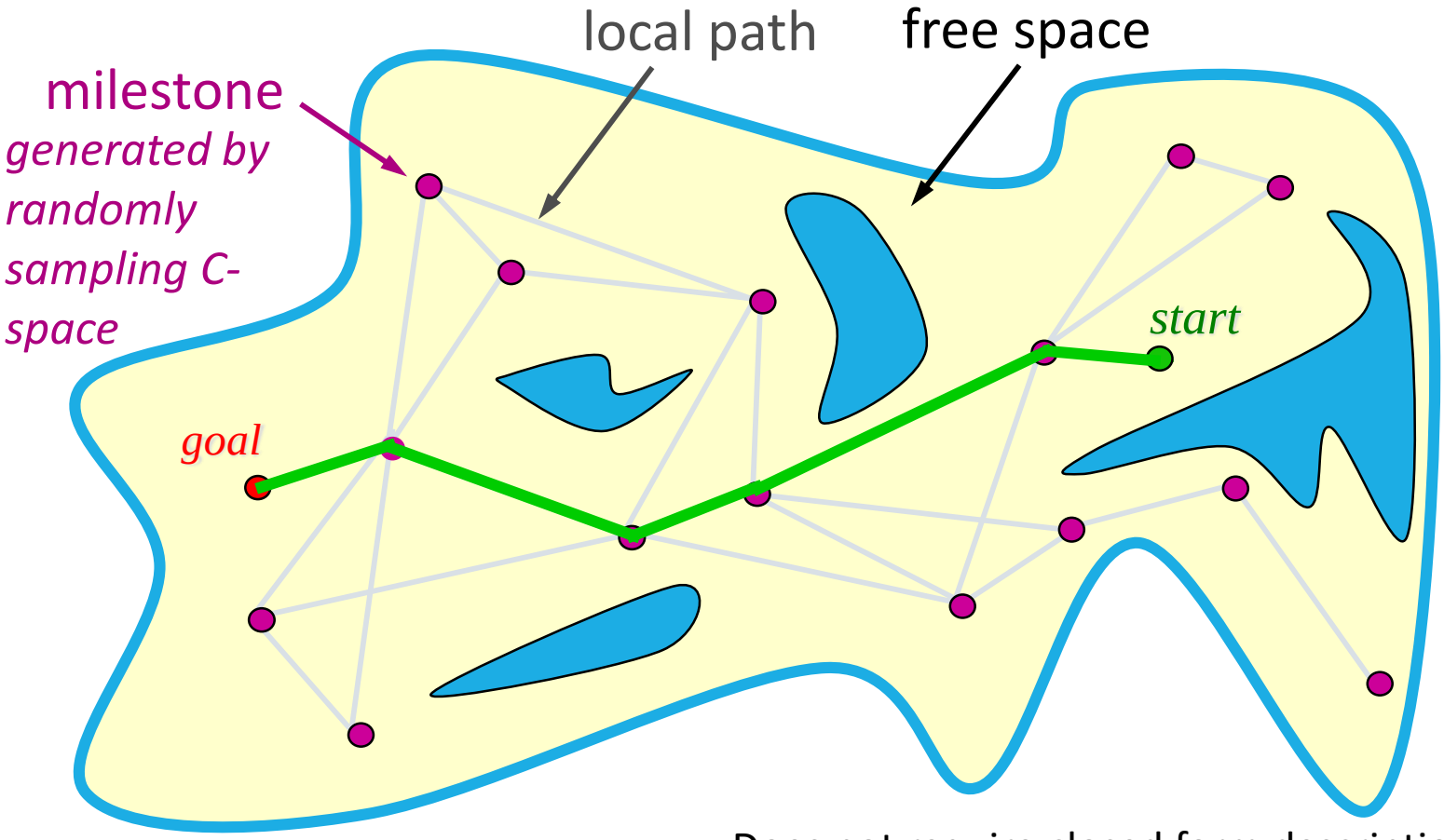


Points on the Generalized Voronoi Graph roadmap are equidistant from at least two obstacles.

Describes maximum margin paths.



Probabilistic Roadmap



Does not require closed form description of C^{obs}

Probabilistic Roadmap (PRM)

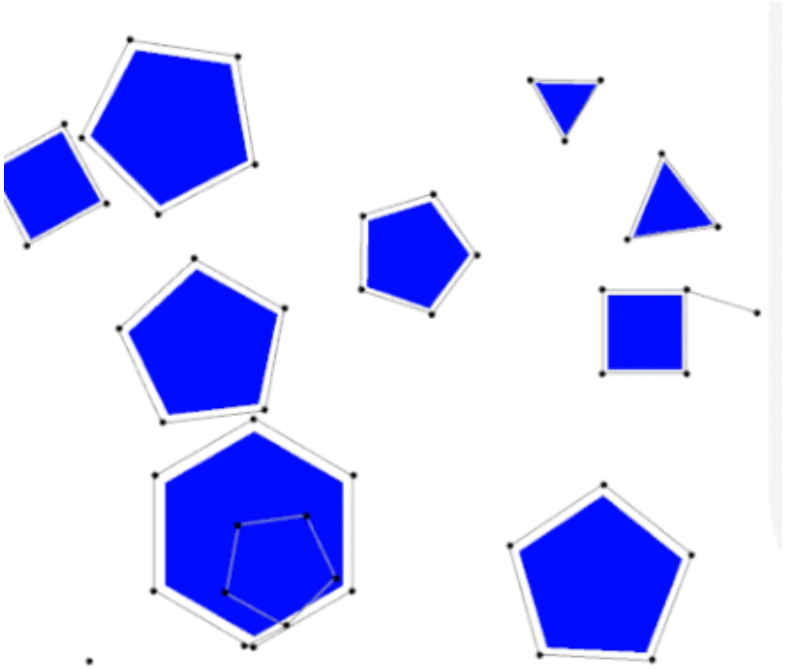
Developed by Kavraki and collaborators

- Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; Overmars, M. H. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces", 1996.

Build roadmap via random sampling

Add start and goal configurations

- Expand the roadmap if this is not possible.



https://en.wikipedia.org/wiki/Probabilistic_roadmap

Build PRM Algorithm

function $Build\_PRM(n, k, \delta)$:
$V \leftarrow \emptyset$ // vertices of PRM
$E \leftarrow \emptyset$ // edges of PRM

while $|V| < n$: // n is maximum number of nodes
    $q \leftarrow$ collision-free random configuration
    $V \leftarrow V \cup q$

for each $q \in V$:
    $N_q \leftarrow$ up to the k closest neighbors to q that are less than distance δ
    for each $q' \in N_q$:
        if $(q, q') \notin E$ and there exists a collision-free path from q to q':
            $E \leftarrow E \cup (q, q')$

return V, E

RRT

RAPIDLY-EXPLORING RANDOM TREES

Rapidly-exploring Random Trees (RRTs)

What if we build the graph incrementally?

What if we only use it once?

Developed by S. LaValle and collaborators, 2000-2003

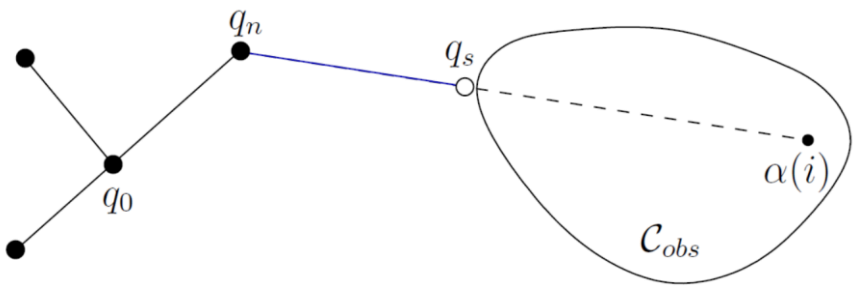
LaValle, *Planning Algorithms*, 2006.
http://planning.cs.uiuc.edu/

https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

Basic Algorithm

Called Rapidly-Exploring Dense Trees (RDT) or Rapidly-Exploring Random Trees (RRT) depending on how points are chosen.

```
RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      q_n ← NEAREST(S, α(i));
4      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
5      if q_s ≠ q_n then
6          G.add_vertex(q_s);
7          G.add_edge(q_n, q_s);
```



start graph from origin

sample a point α, and
find nearest point on the *swath* S
reach from q_n towards α

connect new point q_s to the graph

optional: every once in a while, try α = the goal point.

LaValle, *Planning Algorithms*, 2006

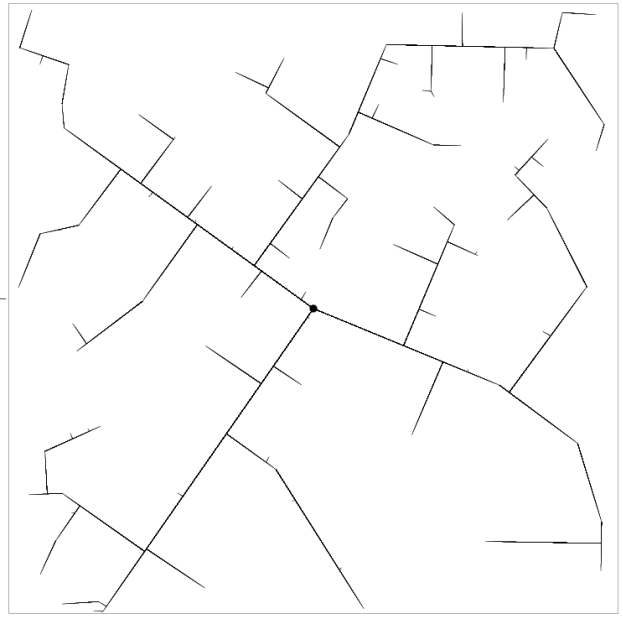
Properties / Variations

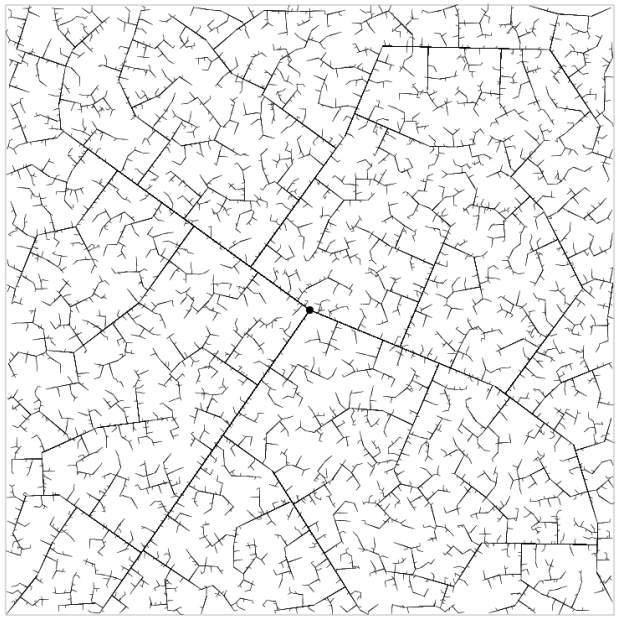
Characteristic fractal-like space filling pattern.

Probabilistically complete.

Lots of room for variations:

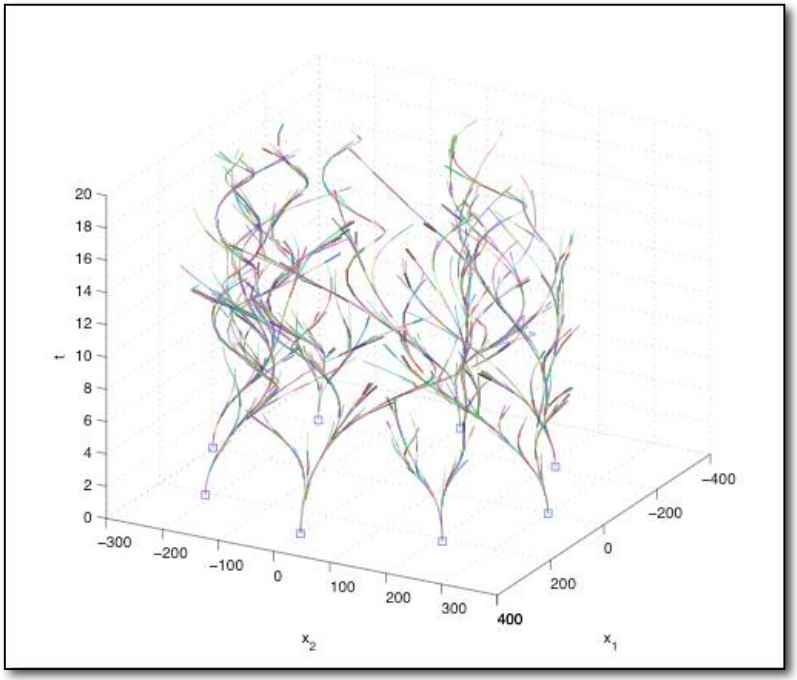
- What do we mean by “nearest?”
- How do we connect to the graph swath?
- Bi-direction search? Many trees?
- How are points sampled?





LaValle, *Planning Algorithms*, 2006

Multiple Trees

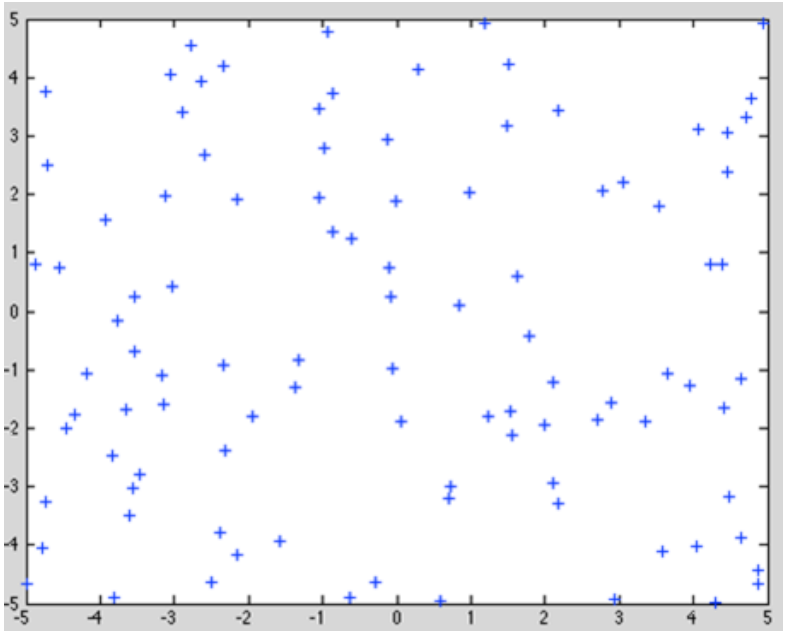


Can grow trees from:

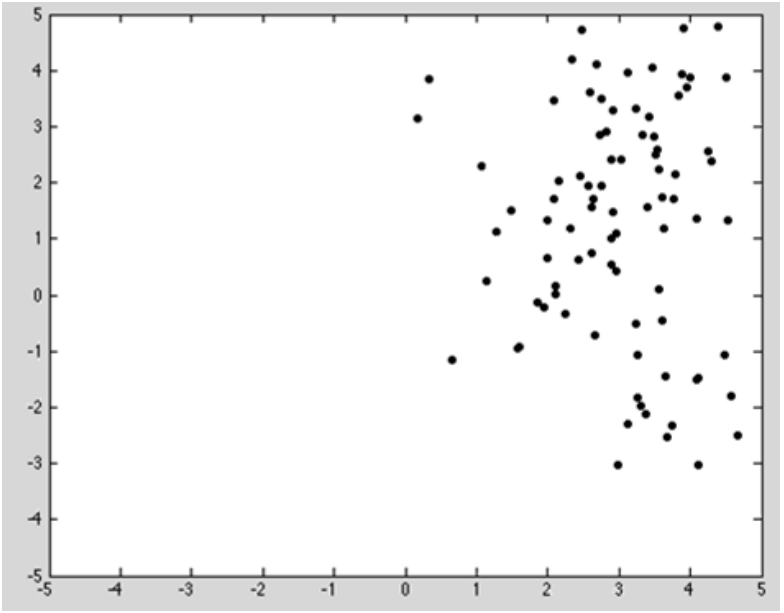
- Start
- Goal
- Start and goal
- Start, goal, and intermediate points
- Etc.

Sampling Strategy

UNIFORM



BIASED TO (3,2)



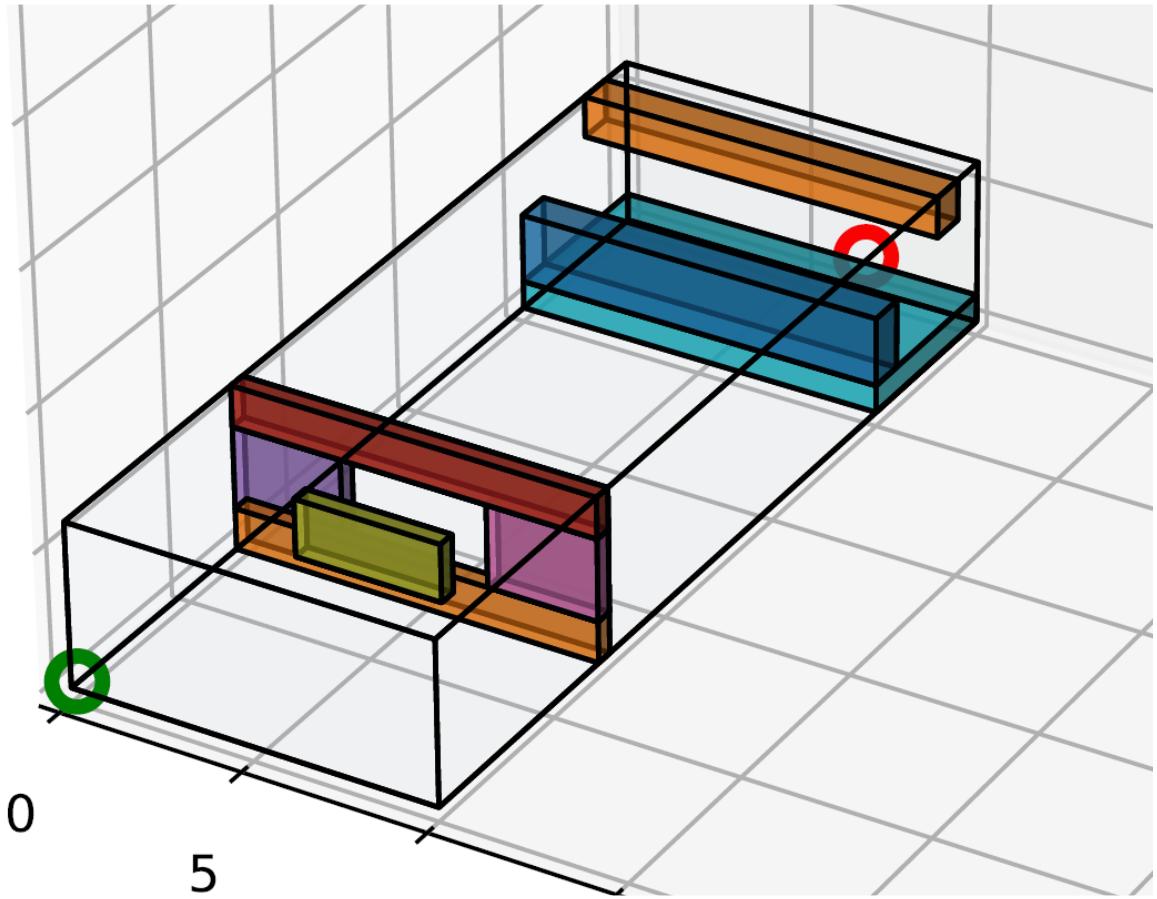
Comparison

	Exact Cell Decomposition	Approx Cell Decomposition	Roadmap methods	Probabilistic methods
Practical above 2 or 3 dimensions	Very slow, memory	Slow, memory (m^n requirement*)	Very slow	Fast
Practical above 5 dimensions	No	Perhaps not, not clear	No	Yes
Optimality	Yes	Yes, up to resolution	Yes	Probabilistic guarantees
Easy to implement	No	Yes	No	Yes
Completeness	Yes	Yes, up to resolution	Yes	Probabilistically complete

*m grid cells along each of n axes

If there is a solution path, the algorithm will find it with high probability

Project 1-2



Project-Inspired Tips

Avoid looping over items in an array.

Example: Replace all negative values in an array with zero.

Input:

```
x = np.array([1, 4, -3, 7])
```

Option #1 using numpy functions:

Idea: "I want x to be the maximum of each value in x and 0."

```
x = np.maximum(x, 0)
```

Option #2 using 'logical indexing.'

Idea: "Everywhere x is less than zero, replace it with zero."

```
x[x < 0] = 0
```

Example: Test if two points are not the same.

Input:

```
start = np.array([3, 7, 9])
goal  = np.array([3, 7, 9])
```

Option #1: Idea: Two points are the same if the distance between them is zero.

```
distance = np.linalg.norm(goal-start)
if distance > 0:
    # Point are not the same.
```

Option #2 Idea: Google the phrase "numpy check if two arrays are equal."

```
if not np.array_equal(start, goal):
    # Points are not the same.
```

Multiply matrices using the '@' symbol.

There is typically no need to explicitly use the the np.matmul function.

```
R = Rotation.from_euler('z', np.pi/4).as_matrix()
x = np.array([1,0,0])
y = R @ x
```

Note that typically you want 3D vectors to have shape=(3,). There is only one possible way to interpret the shape of x in y=Rx or y=xR. There is usually no need to add an extra dimension to x to "force" it to look like a row or column vector.

Print values using "f-strings"

```
x = np.array([1,2,3])
y = np.sum(x)
print(f"The sum of {x} is {y}.")

Output: The sum of [1 2 3] is 6.
```

Use assertions to check your assumptions.

```
while True:
    # Sample Gaussian distribution with mean 1 and stdev 1.
    x = np.random.normal(loc=1.0, scale=1.0)
    y = np.sqrt(x)
    assert not np.isnan(y), f'Square root of {x} is {y}'
```

```
AssertionError                            Traceback (most recent call last)
<ipython-input-12-e2ff553d6dde> in <module>
      3     x = np.random.normal(loc=1.0, scale=1.0)
      4     y = np.sqrt(x)
----> 5     assert not np.isnan(y), f'Square root of {x} is {y}.'
      6

AssertionError: Square root of -0.237540373310813346 is nan.
```

Read Your Tracebacks

```
======================================================================
FAIL: test_unit_traj_end_loc (proj1_1.util.test.TestBase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/autograder/source/meam620-2020/proj1_1/util/test.py", line 229,
    self.assertEqual(flat_outputs['x'].shape, (3,))
AssertionError: Tuples differ: (3, 1) != (3,)
```

Take advantage of basic data types.

- A '**np.array**' is great if you want to do math.
- A '**tuple**' is great for storing a short, constant ordered collections of things, like a shape or index for a numpy array.
- A '**list**' is great for storing an ordered collection of things that you want to add, remove, or modify.
- A '**dict**' is great for storing an unordered collections of things ('values') that you can access by name ('key').
- A '**set**' is great for storing an unordered collection of unique things, and testing if a particular thing is already in your set.