Lesson 8 Moving into 3D

1. **Sebastian Introduction**

[**https://www.youtube.com/watch?time\_continue=2&v=41DhFrk6m7s**](https://www.youtube.com/watch?time_continue=2&v=41DhFrk6m7s)

1. **Introduction**

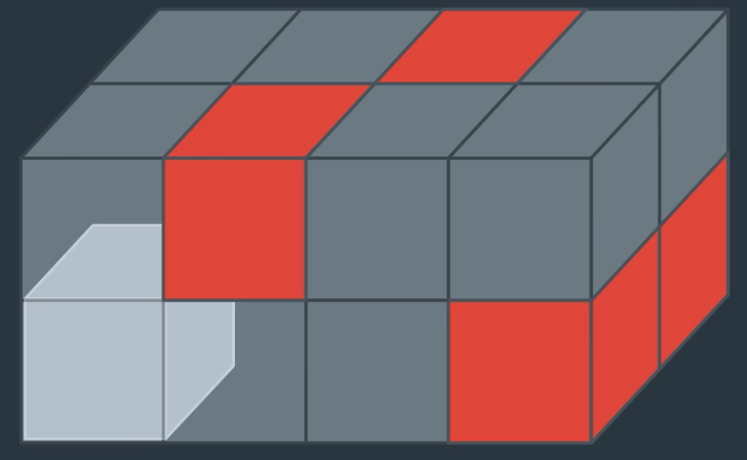
[**https://www.youtube.com/watch?v=x7aDcy9xxrQ**](https://www.youtube.com/watch?v=x7aDcy9xxrQ)

1. **3D Grids**

[**https://www.youtube.com/watch?v=SzNLVvlvOvs**](https://www.youtube.com/watch?v=SzNLVvlvOvs)

1. **Voxel Map Exercise**

**Voxel Map**

[****](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/05fb791c-c342-4f8a-8432-063e6eef021b)

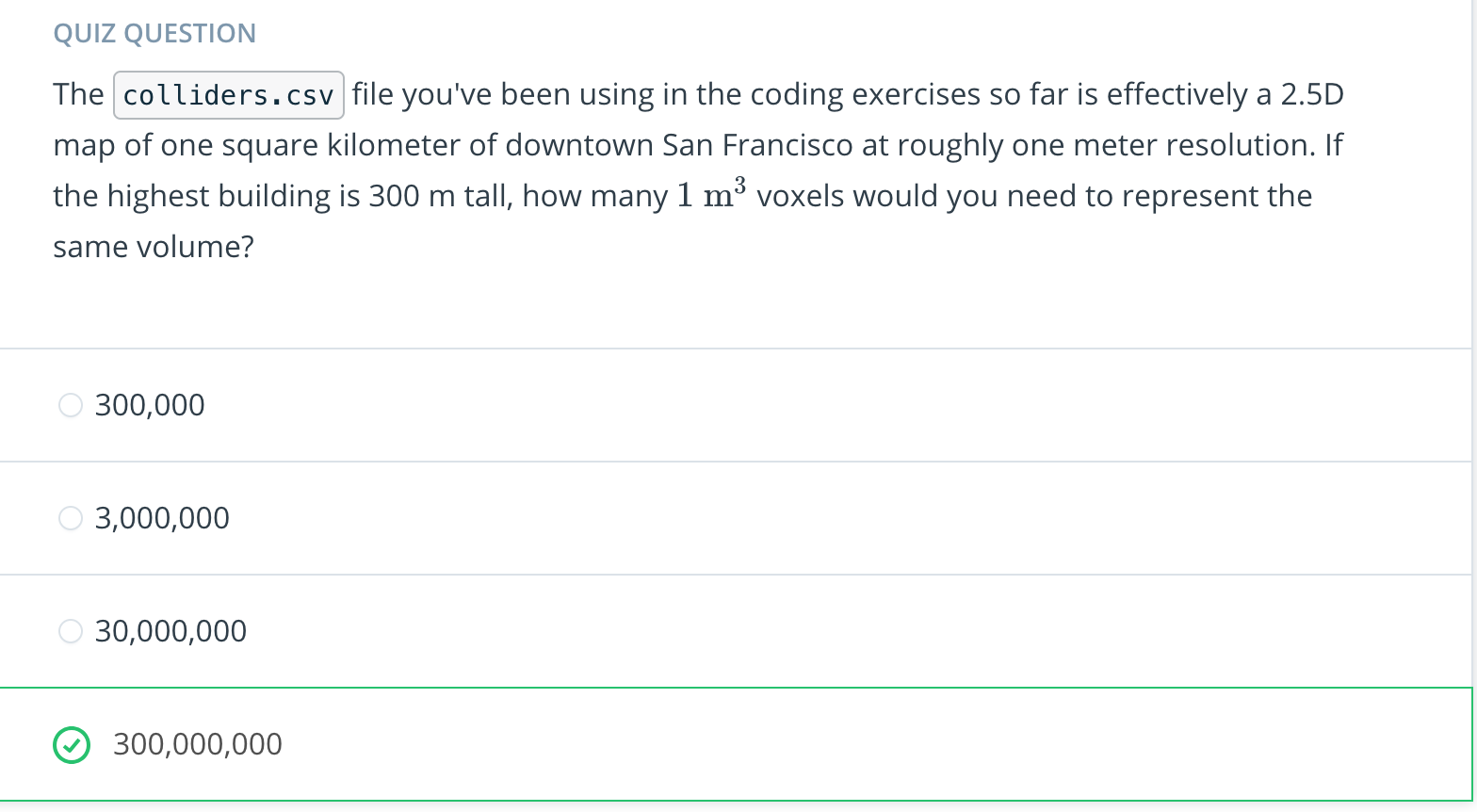
A voxel map is a 3D grid representation of the environment. For flying cars or drones in most cases you wouldn't want to represent your entire search space as a 3D grid, but it can be useful when you want to do higher resolution planning for a local region, which is what we're going to talk about later in this lesson.

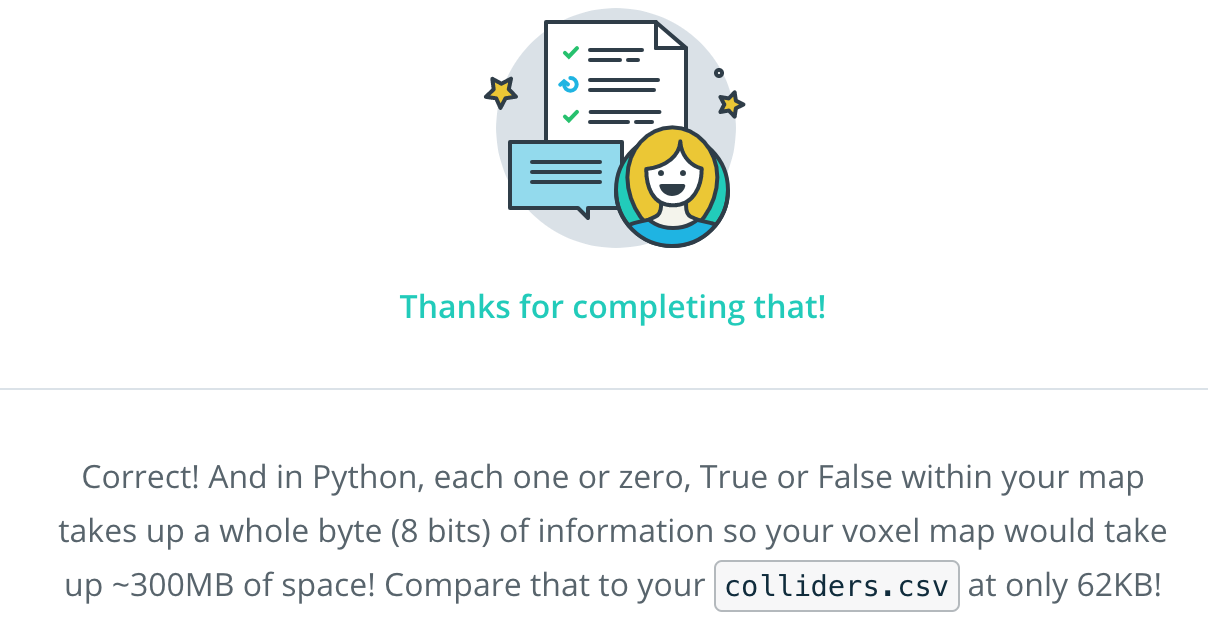
In this exercise, you'll convert the map of the city that you've been working with into a 3D voxel map for visualization.

[**Voxel-Map.ipynb**](https://view5be37991.udacity-student-workspaces.com/notebooks/Voxel-Map.ipynb)

1. **2.5D Maps**

[**https://www.youtube.com/watch?v=oWvl2yaoblY**](https://www.youtube.com/watch?v=oWvl2yaoblY)

****

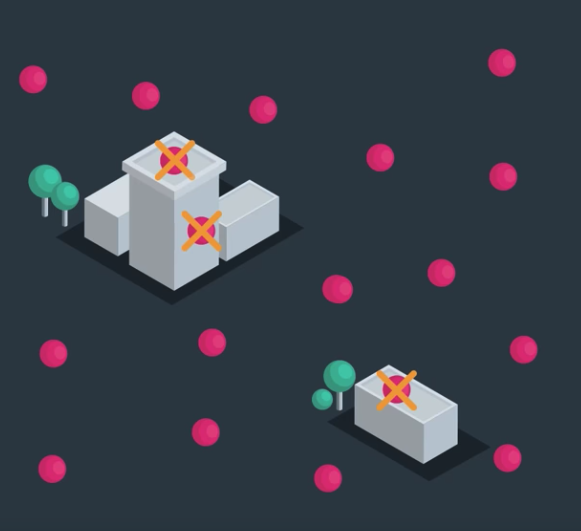
****

1. **Random Sampling**

[**https://www.youtube.com/watch?v=DtmF5xOx2P0**](https://www.youtube.com/watch?v=DtmF5xOx2P0)

1. **Random Sampling Exercise**

**Random Sampling**

[****](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/fc88ad3f-9ac2-4e13-a36e-dc76a509a17b)

Sampling your environment at random might seem like a strange idea, but it's actually a relatively efficient way to build up a set of feasible states through your free space. In this exercise you'll implement random sampling of your 3D environment to generate candidate nodes for a graph representation of your state space.

The process you'll go through to generate a set of feasible states will be to first scatter points at random throughout your environment over some range in x, y, and z. Then for each of those points, you'll test whether it lies inside the ground plane polygon of any obstacles, and if so, whether or not it is above or below the height of the obstacle.

You'll then discard points that are in collision with obstacles, or in other words, those that lie within the ground plane polygon of an obstacle and below the height of that obstacle. And what you're left with are a collection of states that lie in the free space!

**Python Shapely package**

In this exercise, you'll leverage a powerful Python package called [**Shapely**](http://shapely.readthedocs.io/en/stable/manual.html). With this package you can define a polygon object using a set of coordinates like this:

**from** shapely.geometry **import** Polygon

coords = [(0, 0), (1, 0), (1, 1), (0, 1)]

poly = Polygon(coords)

You now have a polygon object with various attributes

print(poly.area)

print(poly.length)

print(poly.bounds)

> 1.0

> 4.0

> (0.0, 0.0, 1.0, 1.0)

But more importantly, for our purposes here you can define a point using Shapely as well and then test whether that point lies within the polygon:

**from** shapely.geometry **import** Point

p1 = Point(0.5, 0.5)

p2 = Point(1.5, 1.5)

print(poly.contains(p1))

print(poly.contains(p2))

> **True**

> **False**

And simple as that you have a method for testing whether a point lies within the ground plane polygon of an obstacle! All you need to do is add in a check for height and you've got your random sampling test in order.

**Random Sampling Exercise**

In this exercise you'll read in the same colliders.csv file you've been using in previous exercises and cast each obstacle into a Shapely Polygon() object. You'll then generate a random bunch of points and test each one for collision with obstacle polygons. Think about what the best approach might be when it comes to testing points for collision. Do you need to test all points against all polygons?

Good luck! And for a peek at our solution you can scroll down to the link at the bottom of the notebook.

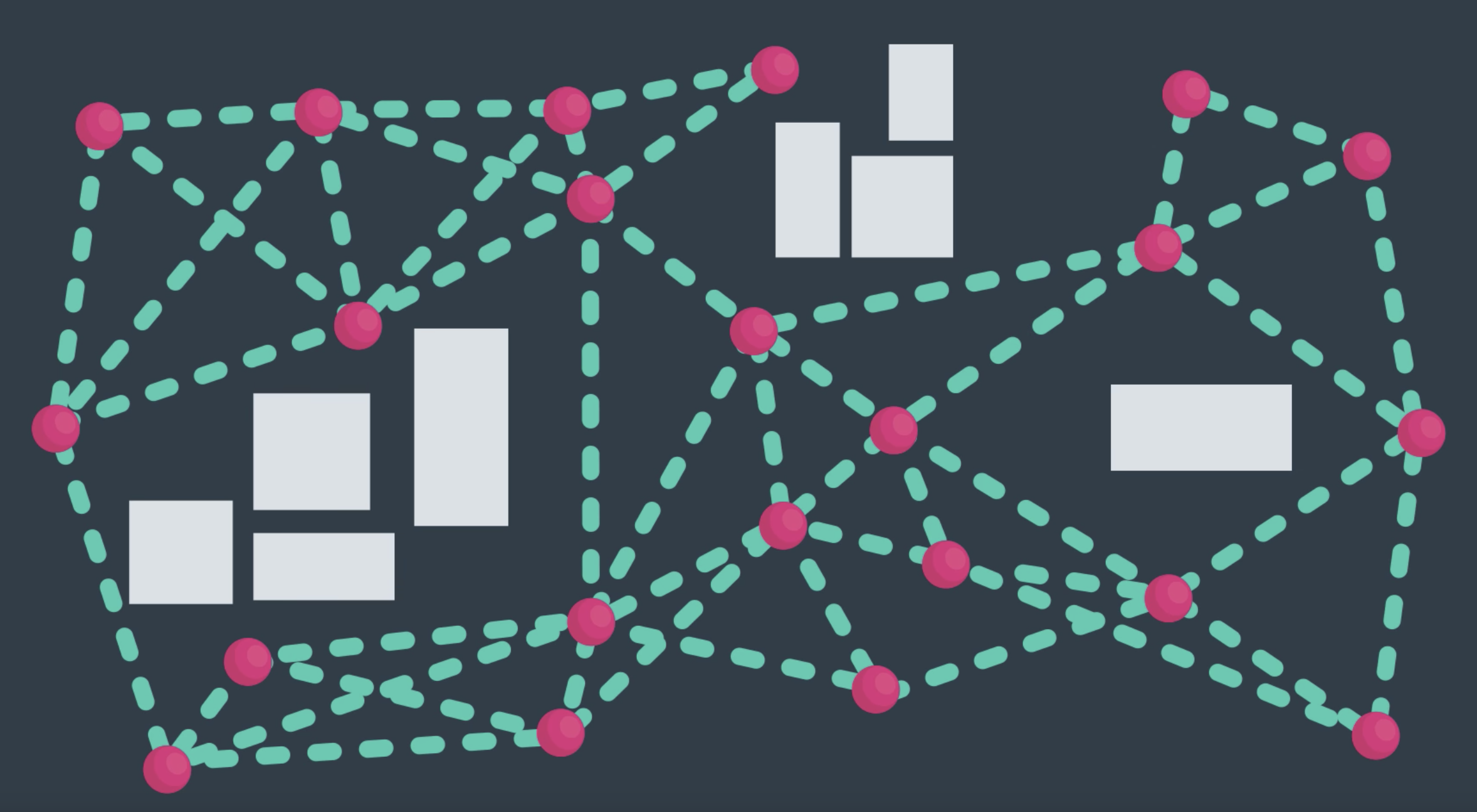
[Random-Sampling.ipynb](https://view10713e24.udacity-student-workspaces.com/notebooks/Random-Sampling.ipynb)

1. **Probabilistic Roadmap**

[**https://www.youtube.com/watch?time\_continue=2&v=DCSe-uBfBNY**](https://www.youtube.com/watch?time_continue=2&v=DCSe-uBfBNY)

1. **Probabilistic Roadmap Exercise**

**Probabilistic Roadmap**

[****](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/7f2d7afe-4775-4377-9fa4-167ce6431315)

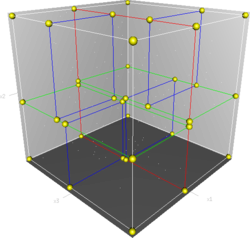
In the previous random sampling exercise, you saw that with the Python [**shapely library**](http://shapely.readthedocs.io/en/stable/manual.html) it's relatively easy to cast your obstacles as polygon objects and test for collision with points in a 2D plane. Then, by comparing the height of the obstacle with the z-dimension of the point in question, you could discard states in collision and retain all candidate states in free space.

In this exercise, you'll connect the states that you've randomly sampled to create a graph representation of the free space in the environment. After that, you'll run search to find a path through this graph from start to goal!

**KD Trees**

In the last exercise, you saw that naively checking for collision between all points and all object polygons can be slow. We hinted there that the [**KD Tree**](https://en.wikipedia.org/wiki/K-d_tree) data structure can help to alleviate this inefficiency by allowing you to quickly identify nearest neighbors to a point or polygon.

You'll be faced with the same problem here when you attempt to connect nodes together into a graph, namely, trying to connect all nodes with all other nodes is costly and not really what you want to do anyway. So now is a good time to consider KD Trees more closely!

[****](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/7f2d7afe-4775-4377-9fa4-167ce6431315)

**[Visual of the KD Tree in action](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/7f2d7afe-4775-4377-9fa4-167ce6431315)**

The KD Tree is a space-partitioning data structure, which allows for fast search queries. The KD Tree achieves this by cutting the search space in half on each step of a query. If you're familiar with [**"big O notation"**](https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/), this brings the total search time down to O(m \* log(n))*O*(*m*∗*log*(*n*)) from O(m\*n)*O*(*m*∗*n*), where m*m* is the number of elements to compare to and n*n* is the number of elements in the KD Tree. So for example, if you want to find the closest neighbor to a single point, m=1*m*=1 and n*n* is equal to the total number of potential neighbors.

The Python Scikit-Learn ([**sklearn**](http://scikit-learn.org/stable/)) library has an easy to use [**implementation of KD Trees**](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html) that we'll be introducing in this exercise. To find neighbors using this implementation, you'll use it like this:

*# Import KDTree and numpy*

**from** sklearn.neighbors **import** KDTree

**import** numpy **as** np

*# Generate some random 3-dimensional points*

np.random.seed(0)

points = np.random.random((10, 3)) *# 10 points in 3 dimensions*

*# Cast points into a KDTree data structure*

tree = KDTree(points)

*# Extract indices of 3 closest points*

*# Note: need to cast search point as a list*

*# and return 0th element only to get back list of indices*

idxs = tree.query([points[0]], k=3, return\_distance=**False**)[0]

*# indices of 3 closest neighbors (will vary due to random sample)*

print(idxs)

> [0 3 1]

**Probabilistic Roadmap Exercise**

In this exercise, you'll perform random sampling as before, but this time using a KDTree for finding the nearest polygon. Next, you'll use networkx to generate a graph of nodes from points that lie in free space. Then you'll test pairs of nodes for connectivity through free space and determine edges for the graph. Once you have a graph of nodes and edges, you'll perform search to find a path from start to goal!

This exercise is pretty involved, but using KDTrees along with everything else you've learned in previous lessons you have all the tools for success! If you want a peek at our solution check out the link at the bottom of the notebook.

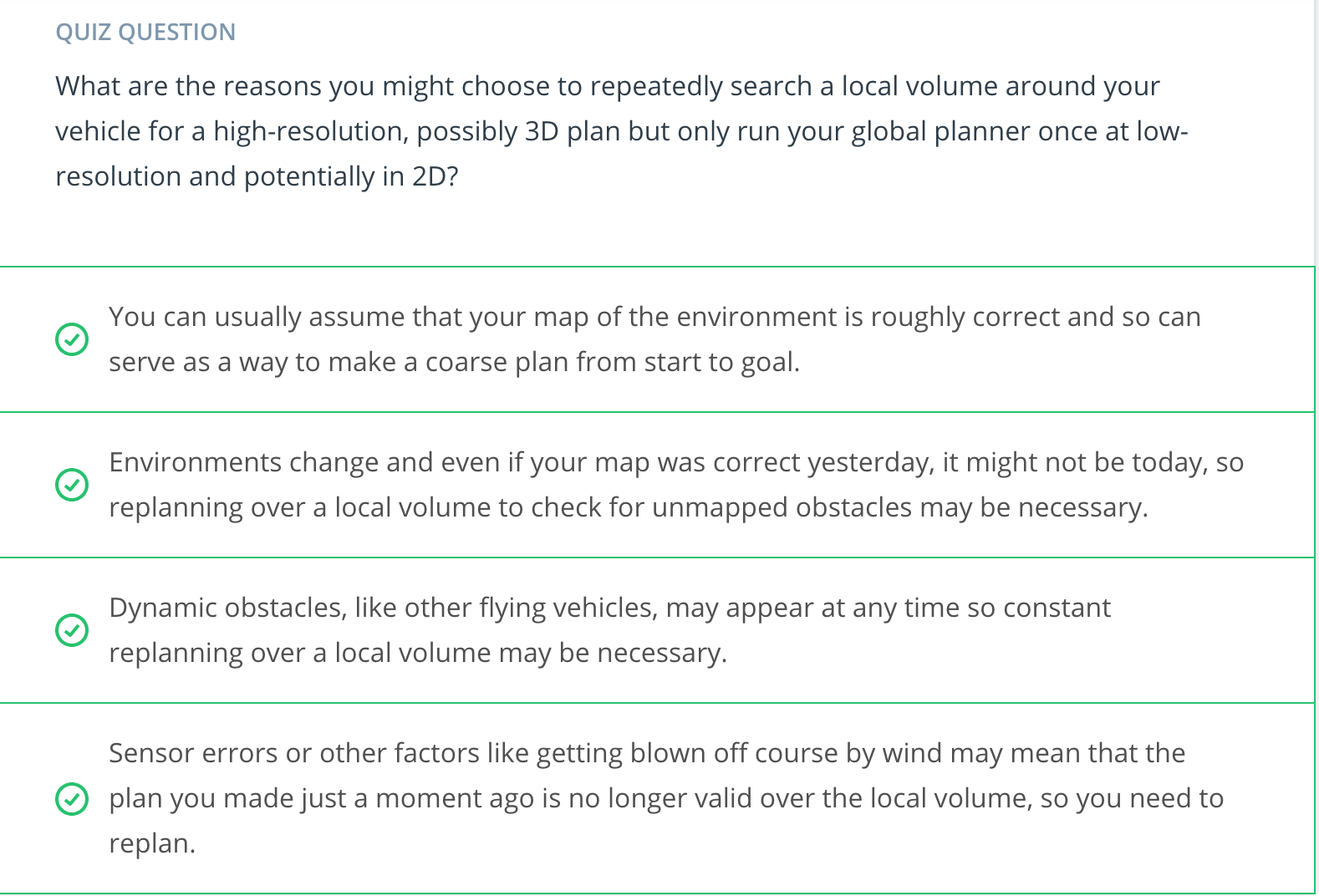
This is a modern technique and the relevant detail goes far beyond what's presented in this exercise. We wanted to give you a taste of it, but if you want to dig deeper, [**this article**](https://arxiv.org/abs/1105.1186) is a great starting point!

**Probabilistic-Roadmap Workspace**

[**Probabilistic-Roadmap.ipynb**](https://viewf0a46bdd.udacity-student-workspaces.com/notebooks/Probabilistic-Roadmap.ipynb)

[10. Local Planning](https://classroom.udacity.com/nanodegrees/nd787/parts/300ffbd4-f9f0-4524-84fb-0b24d9d04cc3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/9ebc0c34-bf32-4b56-bf76-21b14e752dd2)

[**https://www.youtube.com/watch?v=DsUE1rYvRyw**](https://www.youtube.com/watch?v=DsUE1rYvRyw)

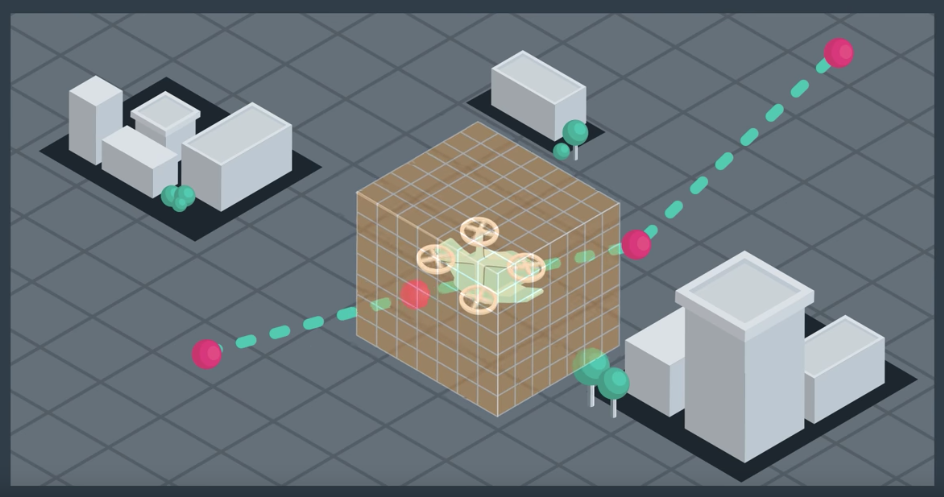
****

[**11. Receding Horizon**](https://classroom.udacity.com/nanodegrees/nd787/parts/300ffbd4-f9f0-4524-84fb-0b24d9d04cc3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/c8d724e0-59cf-404c-bc65-b34deb7c36c0)

[**https://www.youtube.com/watch?v=-wTe\_AFFT-I**](https://www.youtube.com/watch?v=-wTe_AFFT-I)

[**12. Receding Horizon Exercise**](https://classroom.udacity.com/nanodegrees/nd787/parts/300ffbd4-f9f0-4524-84fb-0b24d9d04cc3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/29ae6cac-904e-443c-a67f-9cc82da52f52)

# Receding Horizon Planning

****

**Receding horizon planning is a two-tiered approach to solving the planning problem. First, you find a coarse global plan all the way from the start to the goal. Then, as you execute that plan, you continuously replan in a local volume around the vehicle at a higher resolution. This approach allows for fine tuning your plan on the fly, reacting to obstacles that weren't on the map or other uncertainties, like sensor errors or wind. The edge or horizon of your local planner continuously moves out in front of you and hence the name, receding horizon planning.**

### Receding Horizon Exercise

**Given the dynamic nature of receding horizon planning, it's much more interesting to implement this technique in a simulator, where you can actually execute a plan, than in an exercise. And in the project at the end of these lessons, you'll get a chance to do just that!**

**In the notebook below, there is no simulator, but you can get a feel for receding horizon planning by combining the methods you've worked with up to this point to create a two-tiered solution to your planning problem.**

**First, you'll use the method of your choice to find a coarse global plan from start to goal. Next, you'll generate a plan through your local volume, at a higher resolution, that's not planning toward the goal, but rather, to the next waypoint in the global plan.**

**This exercise is relatively open-ended and, while it won't be required to implement receding horizon planning in the project, we encourage you to investigate the techniques you might use here and consider adding these to your project solution!**

[**Receding-Horizon.ipynb**](https://viewb0522b0d.udacity-student-workspaces.com/notebooks/Receding-Horizon.ipynb)

[**13. Replanning**](https://classroom.udacity.com/nanodegrees/nd787/parts/300ffbd4-f9f0-4524-84fb-0b24d9d04cc3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/83d7f684-2c5b-4a42-9452-37faf3323b84)

[**https://www.youtube.com/watch?time\_continue=5&v=zcZbKZJ3D6s**](https://www.youtube.com/watch?time_continue=5&v=zcZbKZJ3D6s)

[14. Summary](https://classroom.udacity.com/nanodegrees/nd787/parts/300ffbd4-f9f0-4524-84fb-0b24d9d04cc3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/6e2a0873-e549-4045-a877-394ef572f93e/concepts/50a6c40b-ccfa-41f6-9da3-aca3bdd3e039)

[**https://www.youtube.com/watch?v=ydWkeSB30zs**](https://www.youtube.com/watch?v=ydWkeSB30zs)