Lesson 7 From Grids to Graphs

1. **Sebastian Introduction**

[**https://www.youtube.com/watch?v=MYtJg8fe4U4**](https://www.youtube.com/watch?v=MYtJg8fe4U4)

1. **Introduction to Graphs**

[**https://www.youtube.com/watch?v=ikh\_CDbMwec**](https://www.youtube.com/watch?v=ikh_CDbMwec)

1. **Waypoint Extraction**

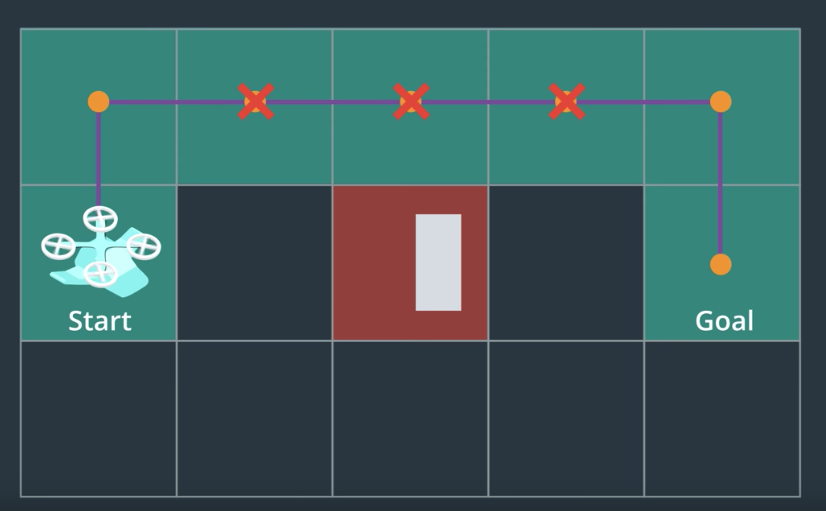
[**https://www.youtube.com/watch?v=rI5AJGHStxA**](https://www.youtube.com/watch?v=rI5AJGHStxA)

1. **Collinearity**

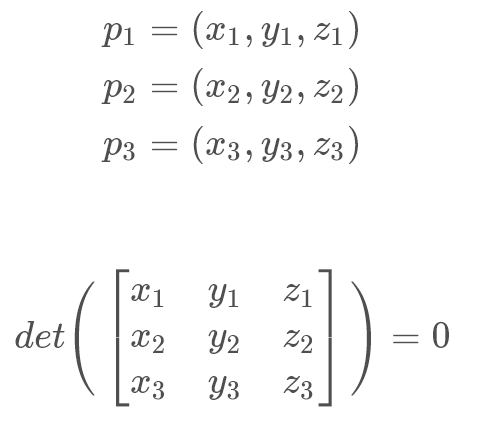
[**https://www.youtube.com/watch?v=NsmiEpZKYJU**](https://www.youtube.com/watch?v=NsmiEpZKYJU)

1. **Collinearity Exercise**

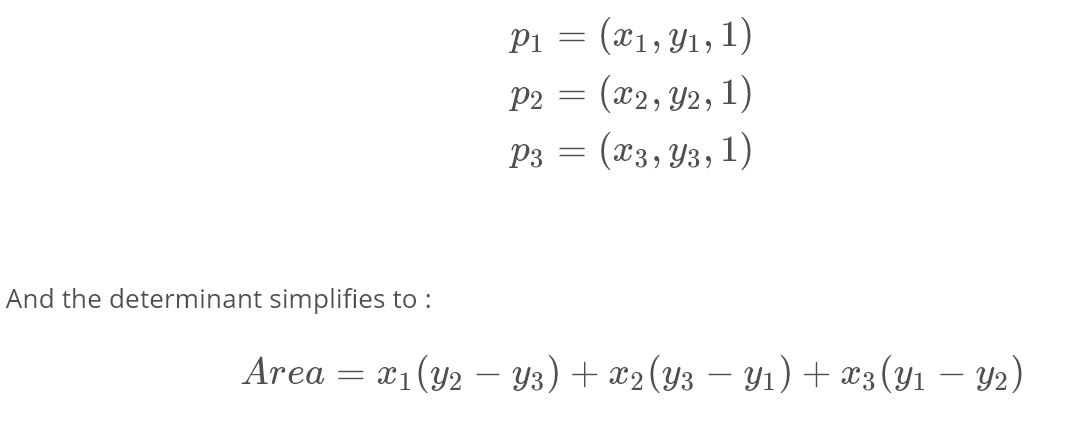
**Collinearity**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/286f789f-bf70-4896-b41c-68907a0c9825)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/286f789f-bf70-4896-b41c-68907a0c9825)**

One way of eliminating unnecessary waypoints in your path is to evaluate whether or not those points are collinear, and remove everything but the endpoints of each collinear series of points. For three points p\_1, p\_2*p*1​,*p*2​ and p\_3*p*3​ to be collinear, the [**determinant**](https://en.wikipedia.org/wiki/Determinant) of the matrix that includes the coordinates of these three points as rows must be equal to zero:



For points in three dimensions this is a [**necessary but not sufficient condition for collinearity**](http://mathworld.wolfram.com/Collinear.html). However, for two dimensional points, evaluating the same determinant with the z*z* coordinate simply set to 1, the determinant being equal to zero indicates that the area of the triangle described by those three points is zero and is a sufficient condition for collinearity:



With Python you can easily evaluate a determinant using numpy:

*# Import numpy*

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

*# Define some 2D points*

p1 = np.array([1, 2])

p2 = np.array([2, 3])

p3 = np.array([3, 4])

*# Define a simple function to add a z coordinate of 1*

**def** **point**(p):

**return** np.array([p[0], p[1], 1.])

*# Add points as rows in a matrix*

mat = np.vstack((point(p1), point(p2), point(p3)))

*# Compute and print the determinant of the matrix*

det = np.linalg.det(mat)

print(det)

>>> 0.0

So that works, but in the real world your points might not be exactly on a line, but for all intents and purposes you can treat them as collinear. For example, consider the case where:

*# Define some 2D points*

p1 = np.array([1.001, 2.002])

p2 = np.array([1.999, 3.001])

p3 = np.array([2.99, 4.002])

*# Add points as rows in a matrix*

mat = np.vstack((point(p1), point(p2), point(p3)))

*# Compute and print the determinant of the matrix*

det = np.linalg.det(mat)

print(det)

>>> 0.008989

If these values are in metres, then most likely you can accept these points as collinear for planning a path for your flying car. So a necessary addition if you're using this method is to introduce a threshold, let's call it epsilon, which indicates how close to zero the determinant must be in order to consider the points to be collinear. This allows you to impose a criterion for accepting points that are *almost* collinear. So you could add the following to your implementation:

collinear = **False**

epsilon = 1e-2

*# Compare the absolute value of the determinant with epsilon*

**if** np.abs(det) < epsilon:

collinear = **True**

So that all works just fine, but np.linalg.det() uses floating point math, which is necessary if your points are non-integer. If, however, you're running on a low-power flight computer and working with grid cell locations, which have integer values, you'd rather be using integer arithmetic.

**Collinearity Exercise**

In this exercise, you'll write a function to evaluate the determinant for three points using the np.linalg.det() function including a threshold epsilon for collinearity. You'll write another function to compute the determinant in the 2D case using only integer arithmetic and compare with the speed of the 3D implementation. Your takeaways here will be two choices for collinearity testing, one which optimizes for speed, and the other which allows you to set a collinearity threshold for ever-so-slightly wandering lines.

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

Collinearity Workspace

1. **Ray Tracing**

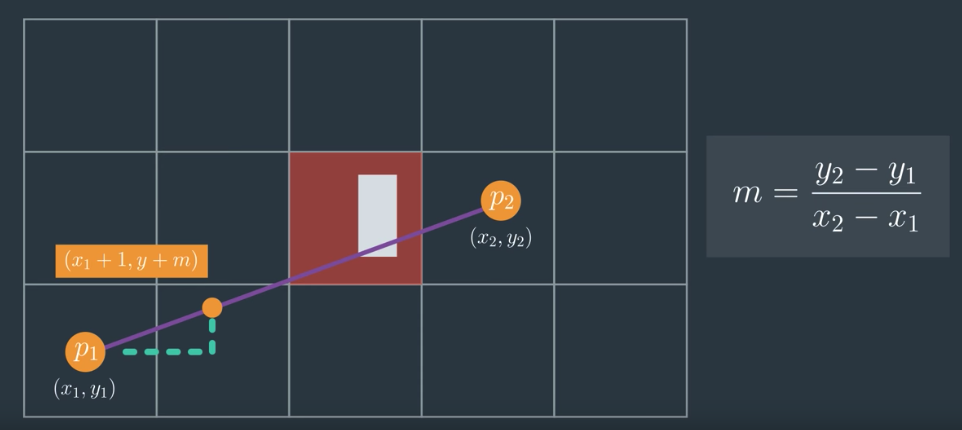
[**https://www.youtube.com/watch?v=BiPdcTmi5Yg**](https://www.youtube.com/watch?v=BiPdcTmi5Yg)

1. **Bresenham**

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

1. **Bresenham Exercise**

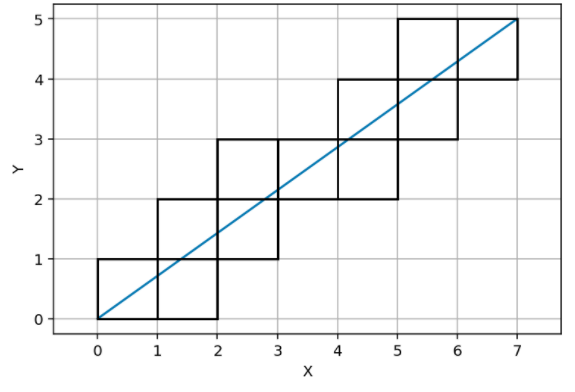
**Bresenham's Algorithm**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)**

The **[Bresenham algorithm](https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html" \t "_blank)** is a way of computing which cells are intersected by a line using only integer arithmetic.

Bresenham developed this algorithm to draw lines in a graphics environment. It is simple, fast and can show which pixels need to be illuminated on a screen to plot a particular line. In our case, the screen is our grid map and the pixels are nodes or grid cells. Because we're using this algorithm for collision detection instead of illuminating pixels, we'll implement it in a slightly different way.

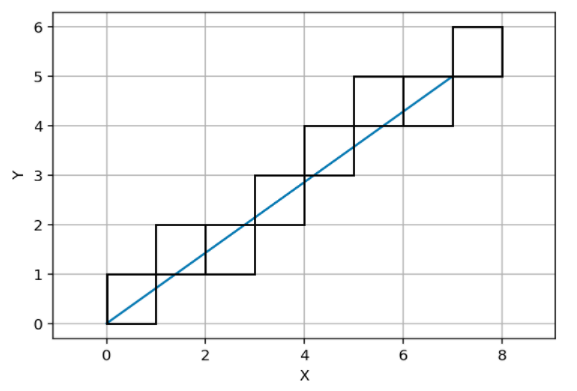
Your goal here is to figure out how to get from p\_1 = (x\_1, y\_1)*p*1​=(*x*1​,*y*1​) to p\_2 = (x\_2, y\_2)*p*2​=(*x*2​,*y*2​) by starting at p\_1*p*1​ and moving 1 unit in the x or y direction until you arrive at p\_2*p*2​. Whether you decide to move vertically or horizontally is determined by your current position with respect to the line connecting p\_1*p*1​ and p\_2*p*2​. For example, moving from the point (0, 0) to (7, 5) would look like this:

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)**

**[Identifying all possible cells in collision with the line](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)**

The difference between this method and the typical computer graphics implementation is that in computer graphics, it's about drawing lines with pixels rather than collision detection so the cells illuminated end up missing some potential collisions along the way from p\_1*p*1​ to p\_2*p*2​.

There's a [**Python package called Bresenham**](https://pypi.python.org/pypi/bresenham/0.1) that uses the computer graphics method. We can use this package and run the same process of identifying the cells that lie along a line from (0, 0) to (7, 5) as we did above and you'll see that the result looks a bit different:

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)**

**[Computer graphics Bresenham method, where some cells in collision may be missed.](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b05f865b-d12a-4e43-ae67-82bcbe04695b)**

With this second method, there are cells the line crosses that are not identified in the plot and this is because the restriction of only being able to move one step in either x or y is not imposed here. In other words, diagonal motions are allowed in the computer graphics version of this drawing method.

What that means is that if you're using the graphics method, you could potentially miss some cells that are in collision if you're planning a trajectory that flies close to obstacles. This is probably fine if you've padded obstacles with a safety margin in creation of your grid map.

**Bresenham Exercise**

In this exercise, you'll get a chance to both implement your own version of Bresenham's algorithm as well as play with the Python Bresenham package implementation. The reason for doing both is that sometimes just using the prebuilt version is fine, but you also need to understand how it can impact your planning solution and how to modify the algorithm to suit your needs if necessary.

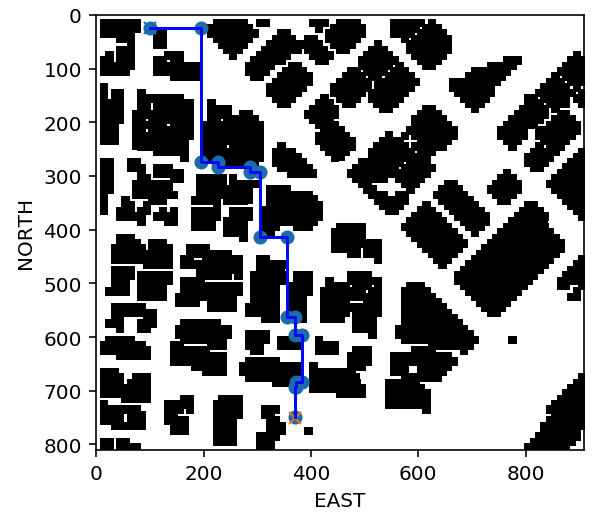
To keep things relatively simple, in the notebook below you'll implement a method for detecting cells that are in collision with a line from points p\_1 = (x\_1, y\_1)*p*1​=(*x*1​,*y*1​) and p\_2 = (x\_2, y\_2)*p*2​=(*x*2​,*y*2​), where x\_1 < x\_2*x*1​<*x*2​ and y\_1 < y\_2*y*1​<*y*2​. Extending this method to work with any p\_1*p*1​ and p\_2*p*2​ is left as an extra challenge to you!

After that you'll have a chance to play around with the package implementation of Bresenham's algorithm. Follow along with the TODOs in the notebook below and if you get stuck, you can check out our solution by scrolling down to the link at the bottom of the notebook.

**Bresenham Workspace**

1. **Putting it Together Exercise**

**Putting it all Together**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b5df5753-2223-49f4-b6a6-e636a317403e)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/b5df5753-2223-49f4-b6a6-e636a317403e)**

**A\* in the City**

At this point, you've got the tools to put together a real planning solution! In this exercise, you'll bring them together to find your way through the city. The steps you'll take include:

* Read in your map of the world (colliders.csv) and create a grid representation of the environment
* Define start and goal locations, as well as a heuristic, and run A\* through the grid to find a path of cells.
* Use collinearity and / or Bresenham to eliminate unnecessary waypoints.

And with that you have a planning solution! Follow along with the TODOs in the notebook below and if you get stuck you can always check out our solution by scrolling down to the link at the bottom of the notebook.

**A-Star-City Workspace**

1. **Grids to Graphs**

[**https://www.youtube.com/watch?time\_continue=4&v=DDamvBxXP3I**](https://www.youtube.com/watch?time_continue=4&v=DDamvBxXP3I)

1. **Graph Tradeoffs**

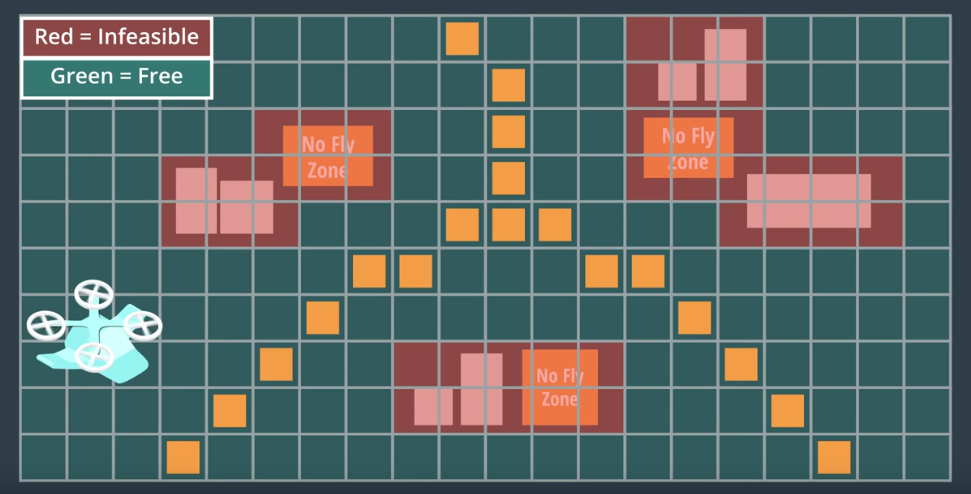
[**https://www.youtube.com/watch?v=gQvoAaAYP88**](https://www.youtube.com/watch?v=gQvoAaAYP88)

1. **Generating Graphs**

[**https://www.youtube.com/watch?time\_continue=1&v=Pj1U6VHs1w4**](https://www.youtube.com/watch?time_continue=1&v=Pj1U6VHs1w4)

1. **Medial Axis Exercise**

**Medial Axis Skeletonization**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/82275abd-86bc-43c0-a613-06728daffb22)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/82275abd-86bc-43c0-a613-06728daffb22)**

The medial axis method is an image processing technique for identifying a "skeleton" of a binary image, or in this case, your grid map of obstacles and free space. In this exercise, you'll be using the **[medial\_axis()](http://scikit-image.org/docs/0.10.x/auto_examples/plot_medial_transform.html" \t "_blank)** transform method within the the Scikit-Image library.

This method returns the one-pixel-wide skeleton of the image so for instance, when you run it on the grid you've created of the environment from the previous exercises, it looks like this:

**from** skimage.morphology **import** medial\_axis

**from** skimage.util **import** invert

*# Assuming you've already created the grid*

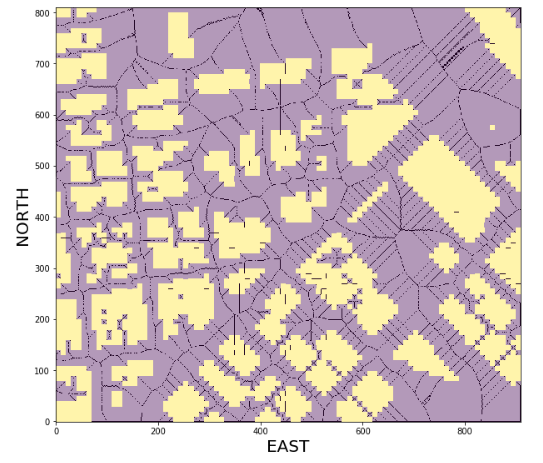
*# The medial\_axis() method requires that we invert the grid image*

skeleton = medial\_axis(invert(grid))

plt.imshow(grid, origin='lower')

plt.imshow(skeleton, origin='lower', alpha=0.7)

plt.show()

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/82275abd-86bc-43c0-a613-06728daffb22)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/82275abd-86bc-43c0-a613-06728daffb22)**

Now the black pixels represent the medial axis of the free space and now the entire connectivity of your free space is expressed as strings of adjacent pixels laid out along thees medial axes between obstacles. So how can you use this for planning?

**Medial Axis Exercise**

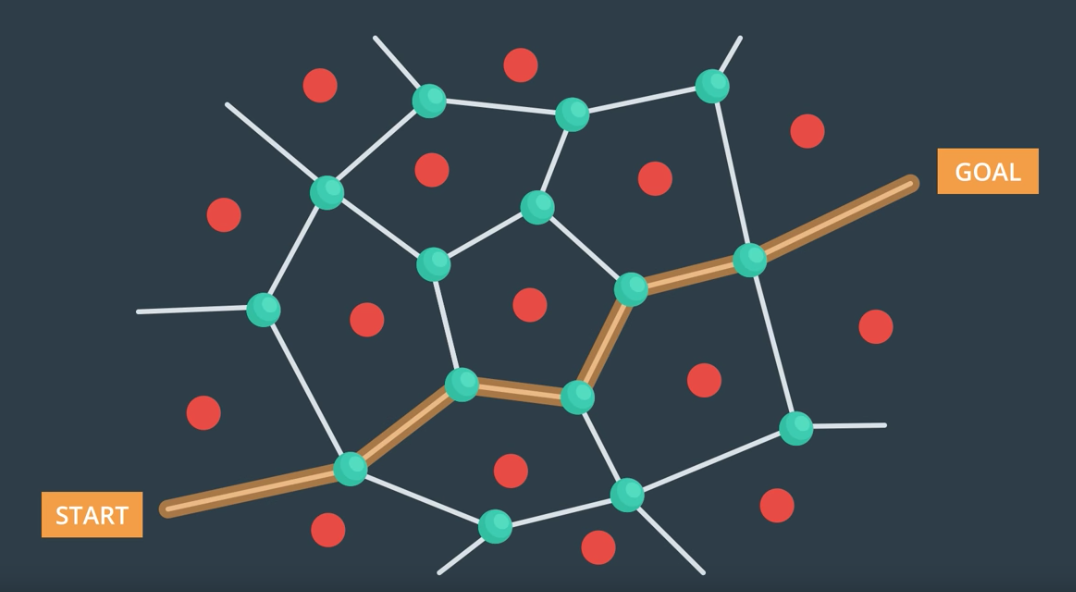
In this exercise, you'll extract a grid map of obstacles and free space as in previous exercises and run the medial axis transform on it to get at the underlying skeleton of the free space. There are a variety of options for how you could use this output for planning, but the way you'll do it in this exercise is to simply identify the points on the skeleton that are closest to your start and goal locations, then run A\* across the skeleton to find a path between these points.

Next, you can use collinearity or Bresenham (or maybe some other method) to reduce the path you found through the skeleton to a sparse collection of waypoints! If you get stuck you can have a look at our solution by scrolling down to the link at the bottom of the notebook.

**Medial-Axis Workspace**

1. **Voronoi Graph Exercise**

**Voronoi Graphs**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)**

Voronoi diagrams can be tricky to construct, but thankfully Python's **[SciPy library](https://www.scipy.org/" \t "_blank)** already has an implementation of the **[Voronoi method](https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.spatial.Voronoi.html" \t "_blank)** built in! The way this method works is that you need to feed in a set of points that, in our case, represent the obstacles, and create a Voronoi() object that contains a graph of the "ridges" that define the midline in free space between the obstacles.

For example, you can generate some random points and plot up the associated Voronoi graph like this:

*# Import numpy and Voronoi method and plotting routines*

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

**from** scipy.spatial **import** Voronoi, voronoi\_plot\_2d

**import** matplotlib.pyplot **as** plt

*# Generate 50 random points with integer values between 0 and 49*

points = np.random.randint(50, size=(50, 2))

*# Extract the Voronoi diagram*

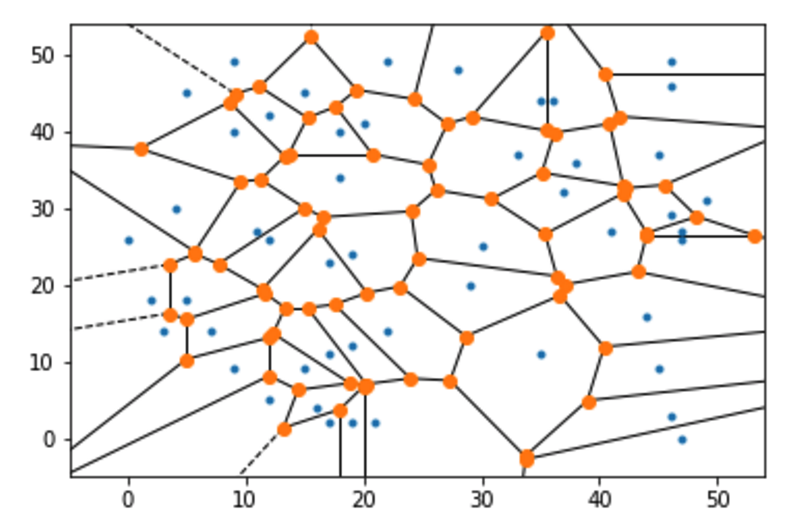
graph = Voronoi(points)

*# Plot it up!*

voronoi\_plot\_2d(graph)

plt.show()

And you get something that looks like this (will look different if you try this at home because values are random):

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)**

In this image, you can think of the blue points as obstacles (inputs to the Voronoi method), and orange points are the intersections of the Voronoi ridges, or in other words, the nodes in a graph that navigates between the obstacles!

In the real world, of course, obstacles are not points but they have some size and shape. Still, the Voronoi method can be a useful starting point for generating a planning graph around real world obstacles and that's what we're going to do next!

**Extracting Nodes and Edges**

For the example shown above, the edges in the graph or "ridges" as they're often called in the context of Voronoi diagrams, all represent feasible paths to navigate around those point obstacles. In the upcoming exercise, you'll start by creating a similar graph based on obstacle center points, but then you need to add an additional step to test whether edges in the graph are traversing free or infeasible space.

Within the graph object created in the code snippet above, you've got all the information you need to extract nodes and edges and test for feasibility. The graph.vertices attribute contains the x and y positions of each of the vertices (or nodes), and graph.ridge\_vertices contains the indices that identify each pair of points that define a ridge in the graph.vertices list. I think that will make more sense if I say it in code!

*# Assuming you've created a graph object as shown above*

*# You can step through each pair of points that define a ridge like this*

edges = [] *# Empty list to contain valid edges*

**for** v **in** graph.ridge\_vertices:

p1 = graph.vertices[v[0]]

p2 = graph.vertices[v[1]]

*# Then you can test each pair p1 and p2 for collision using Bresenham*

*# (need to convert to integer if using prebuilt Python package)*

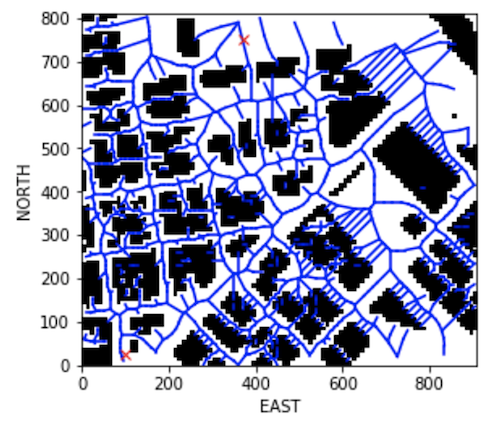
*# If the edge does not hit an obstacle*

*# add it to the list*

**if** **not** in\_collision:

edges.append((p1, p2))

Your goal is to plot up all the valid edges and the result should look something like this:

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/0f7d82e6-08ae-4b3f-bcc4-af4295b45acd)**

**Voronoi Exercise**

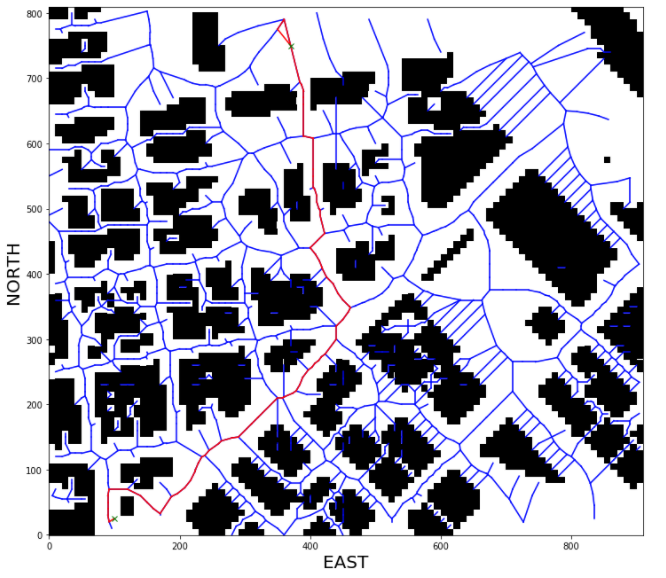
In the notebook below, you'll read in the map of the world that you've been using in previous exercises and generate a Voronoi graph like the one above for obstacles that are not simple points, but actual 2D polygons in the ground plane. To accomplish this task, you'll need to fill in the TODOs in the create\_grid\_and\_edges() function below.

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

**Voronoi Workspace**

1. **Graph Search Exercise**

**Graph Search**

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/c766ccda-fc5f-4137-8bcf-5cf288318141)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/0c12632a-b59a-41c1-9694-2b3508f47ce7/lessons/5a623140-3e28-4ef1-a0f2-104715381c54/concepts/c766ccda-fc5f-4137-8bcf-5cf288318141)**

Alright now things are getting interesting! You've got a couple different methods for building a graph, and you've got some tools you developed previously to search through grids. You may have started to notice that grids and graphs actually have a lot in common, and with the medial axis transform in particular, we're really sort of straddling the divide between grids and graphs.

The fact is that grids are really just a subset of graphs where all possible actions are of a fixed size and direction. Graphs on the other hand, are more flexible and allow you to define actions over any distance and in any direction. These similarities mean that you can take your grid-based search methods and easily modify them to work on a graph!

**Graph-based A\***

In the last two exercises you generated graph representations of your search space. The way A\* worked for a grid was that in expanding your list of partial plans, you considered the sum of the cost function G and the heuristic H in deciding where to move to next.

For graphs-based A\*, you'll do essentially the same thing. The "Manhattan distance" as a heuristic no longer makes sense, but Euclidean distance can still serve as a perfectly valid heuristic estimate. For the cost function, you can now think of the cost of moving between two nodes as a function of the distance between those nodes in the graph. And with that, you have your graph-based A\* implementation!

**NetworkX**

In order to make graph creation and manipulation simple, we're going to leverage a powerful package called **[NetworkX](https://networkx.github.io/" \t "_blank)**. With Networkx, it's easy to take the nodes and edges you found using the Voronoi and medial axis methods and arrange them into a single graph object. In this exercise you'll start with the Voronoi method to extract a graph where nodes and edges don't collide with obstacles. Then you'll use NetworkX to build up a graph object like this:

*# Assuming you have already extracted "edges"*

*# And edges is a list of tuples of the form (p1, p2)*

import networkx **as** nx

G = nx.Graph()

**for** e in edges:

p1 = e[0]

p2 = e[1]

dist = LA.norm(np.**array**(p2) - np.**array**(p1))

G.add\_edge(p1, p2, weight=dist)

**Graph-Search Workspace**

1. **Deadbands**

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

**QUESTION 1 OF 2**

Check out the local\_position\_callback() function in the [**backyard\_flyer.py**](https://github.com/udacity/FCND-Backyard-Flyer/blob/solution/backyard_flyer.py) script. What is the deadband radius being used there?

* 

0.1 m

* \* \* 1 m
* 

10 m

SUBMIT

**QUESTION 2 OF 2**

Try resetting the deadband radius in the [**backyard\_flyer.py**](https://github.com/udacity/FCND-Backyard-Flyer/blob/solution/backyard_flyer.py) script to 0.01, 0.1 and 5 and running the script with the simulator. Match each deadband radius with the with the appropriate description of the behaviour of the quad.

*Submit to check your answer choices!*

**BEHAVIOUR**

**DEADBAND RADIUS**

The quad executes the path quickly but it appears almost circular instead of square

5 m

The quad executes the path more precisely but more slowly than with a 1 m radius.

0.1 m

The quad executes the path precisely, but gets stuck at each waypoint struggling to land within the deadband radius.

0.01 m

SUBMIT

1. **Summary**

[**https://www.youtube.com/watch?v=RGkoXva9rSc**](https://www.youtube.com/watch?v=RGkoXva9rSc)