

CS221 Lecture notes #2

Robotics and motion planning

In the first lecture, we saw two examples of intelligent systems—Flakey the robot, that could move around an office environment and talk to people, and Stanley, Stanford’s autonomous car. An intelligent systems can be a robot and have a physical presence, or it can also entirely live within your desktop computer. Today, lets talk about what it takes to physically control a robot.

What is a robot? Informally, it is a physical system that interacts environment using physical sensors and effectors.

Some examples of robotic sensors include video cameras; sonar (which measures distances to obstacles by measuring how long it takes sound to bounce off the obstacle and return to the robot); laser range scanners (which works similarly to sonar, except it bounces light rather than sound off the obstacles); microphones; odometers (which measure distance traveled); GPS; accelerometers; and many more. These sensors give the robot information about the state of the environment around it, as well as information about the robot’s location and orientation within that environment. Another important type of sensor are “proprioceptive sensors,” which tells the robot about the position or changes in position of its own joints.

For most robots, their effectors can be divided into two categories based on their function:

- **Locomotors**, such as wheels or legs, to allow the robot to move itself around. Wheels are popular since they are easier to control, but there many other possibilities, such as legs.
- **Manipulators**, such as a robot arm and hand, which allow the robot to interact with the world and affect the world around it.

Given a robot, how can we get it to drive from one place to another, without hitting obstacles? Alternatively, given a robot arm, how can we generate a smooth motion for it to reach out and, say, pick up an object? At

its basic level, a robot consists of a set of motors, and we have to decide (say) what sequence of joint angles to command each motor to go to. We would like to find a sequence of joint angles that will cause the robot to follow some “path” from its initial position to some goal position. To develop algorithms to accomplish this, we will need to introduce the concept of a configuration space.

1 Configuration space

First consider a robot in a 2D plane. How would we describe its position? That depends on whether it can rotate, or just translate without rotating in the plane. In the latter case, we can describe the robot’s position with a pair of real numbers, such as its Cartesian coordinates (x, y) . In the former case, we would need three real-valued parameters (x, y, θ) , with θ giving the robot’s orientation.

This leads us to the important notion of **degrees of freedom (dof)**: A robot has k degrees of freedom if its current **configuration** can be fully described by a set of k real numbers. Thus, the robot which is allowed to translate and rotate has three degrees of freedom, while the one which is only allowed to translate in a plane has two.

How about the state of a helicopter, that can fly around in 3D? We would need three real numbers to give its position in space, as well as three more numbers (such as roll, pitch, and yaw) to specify its orientation. Thus, the helicopter has six degrees of freedom. Note that there are sometimes many possible choices for the parametrization; for instance, we could use either the Cartesian coordinates (x, y, z) , or latitude, longitude, and height above sea level to specify the location. The relative merits depend on the application, but any sensible parametrization should still result in six degrees of freedom.

What about the robot arm shown in Figure 1(a)? One (naive) parametrization would be to give the Cartesian coordinates of the lower-left corners of each of the two parts; thus we could specify the position of the robot using four real numbers. The problem with this parametrization is that almost all values of these four real numbers correspond to illegal configurations for the robot (and are not possible unless we break the arm); therefore, it doesn’t capture the true dimensionality of the space. A better parametrization would be two real numbers specifying the angle of each of the joints. This captures the allowable variation with the minimum number of dimensions.

Suppose a robot’s configuration can be specified via k real numbers. The set of all possible values for these k real numbers is called the **configuration**

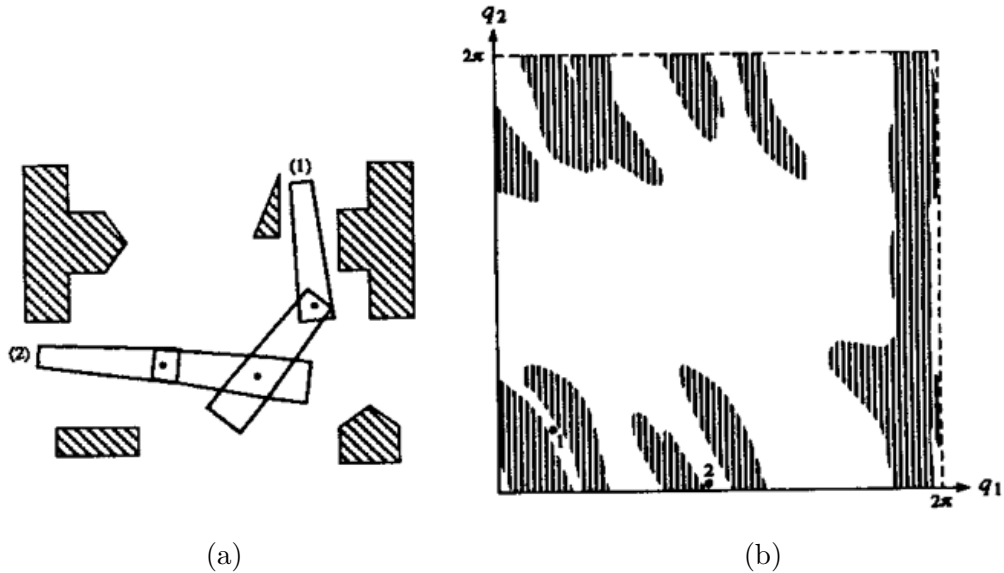


Figure 1: (a) The workspace for a robot arm with two degrees of freedom, one for each joint. (Assume the base of the arm is anchored to the wall.) (b) The configuration space for the robot. Images courtesy Jean-Claude Latombe.

space of the robot. Consider, for instance, a circular robot which moves in a 2D workspace as shown in Figure 2. The configuration space for this robot is the set of all (x, y) coordinates; thus its configuration space is \mathbb{R}^2 .

Our goal is path-planning—finding a legal path for the robot which it can physically follow. Therefore, we will divide the configuration space into two parts: **free space** and **obstacles**. Free space is the part of the configuration space that represents legal positions of the robot; obstacles correspond to the illegal positions, such as if the robot is physically embedded within some other object.¹

In point of terminology, we will use the term **workspace** to refer to the physical environment that the robot works in. Thus the workspace is always 2D or 3D, and corresponds directly to our physical world. The term “free space” will always refer to a subset of the configuration space. But we may use “obstacle” either to refer to the set of illegal positions in configuration space, or to physical obstacles that the robot may encounter in its workspace.

Previously, we described a 2D planar robot. What is its free space? That depends on how big the robot is. If the robot is a point, the configuration

¹Note that obstacles in the configuration space may not necessarily correspond only to physical obstacles in the world; for example, they might also correspond to states where one part of the robot tries to pass through another part.



Figure 2: (a) An example workspace with the circular robot shown in red. (b) The resulting configuration space, where the white areas are the free space. Coordinates are defined with respect to the center of the robot.

space looks the same as the workspace; but if the robot has shape, the two usually differ. In this case, we need to be precise about how we use a pair of cartesian coordinates (x, y) to describe “where the robot is.” We define the configuration of the robot by the location of some specific *reference point* on the robot. If the robot is round, for instance, we may choose its center as the reference point.

For the case of a non-rotating 2D robot, a point (x, y) in the configuration space is in the free space if, when the physical robot’s reference point is at position (x, y) in the workspace, the robot does not intersect with any obstacle in the workspace. The free space for our robot is shown in Figure 2(b).

What about our arm robot from before? There, we have to be even more careful. Obstacles in the configuration space include configurations where the robot overlaps an obstacle, but also configurations where one arm of the robot passes through the other. The resulting configuration space is shown in Figure 1(b).

2 Path planning

Recall that our task is to find a path moving the robot from an initial position to some goal position. The initial and goal positions of robot will correspond to a pair of initial and goal points in the configuration space. To accomplish our task, all we need to do therefore is find a path in configuration space going from the initial configuration to the goal configuration, that is contained entirely in the free space. Once we have found a path, the sequence of points along this path will correspond to a continuous sequence of valid positions

for the robot that take it from the initial position to the goal position.

This formulation of the problem makes several assumptions, namely, that:

- The obstacles are known in advance (and do not move).
- The robot can follow any path in free space.

These assumptions often hold, but let's discuss the second of these assumptions in more detail. In particular, let's discuss an example of when the second assumption is violated.

A car moving in a 2D plane has three degrees of freedom, since its state can be represented with the triple (x, y, θ) , where θ is its orientation. We really do need all three degrees of freedom to represent the car's position, since given a big enough plane, it can get itself into any position and orientation. Suppose now that we have a car in position (x, y, θ) , and we want it to wind up in $(x, y + \Delta y, \theta)$. Assuming there are no obstacles in the workspace, there is clearly a straight line in configuration space connecting these two points: one in which y varies, but x and θ remain constant. The car can follow the path if $\theta = 90^\circ$ or 180° , because the car is then aligned with the y axis. Otherwise it is impossible, because that would require it to move sideways.

In this particular case, we have three degrees of freedom, but we can't control them directly. In fact, we only have two independent controls: forward-backward motion and steering. We say that the car has only two **controllable degrees of freedom**. Thus, even given two adjacent points in configuration space, we may not be able to move directly from one of these points to another. A robot where the number of controllable degrees of freedom is equal to the number of total degrees of freedom is called **holonomic**. Otherwise, it's called **nonholonomic**. It is possible, albeit difficult, to design robots capable of holonomic locomotion in a plane. There are also techniques for controlling nonholonomic robots (some of which we'll see later this quarter). But for simplicity, we will assume here that our robot is holonomic.

3 Search space for motion planning

We have seen that the motion planning problem, under certain assumptions, can be reduced to the problem of finding a path from an initial configuration to a goal in the robot's configuration space. Note how elegant this formulation is—whereas the problem of moving a robot arm from one position to another seems to involve lots of complicated things such as the geometry of the arm, what obstacles there are around the robot, the specific shapes/positions of these obstacles, etc., the notion of a configuration

space takes all of the geometry and obstacles into account, and reduces the motion planning problem into that of planning a path in a k -dimensional configuration space.

Unfortunately, the configuration space is continuous, and it is usually very hard to reason directly with continuous spaces. Therefore, our approach to finding a path in configuration space will be based on formulating a discrete graph search problem, and applying standard discrete graph search algorithms to find a path. The problem of finding a path (or the shortest path) from an initial state to a goal in a discrete graph is well-studied in computer science.

There are many ways to discretize a continuous configuration space, and the relative merits of each depend on many factors of the particular task, such as the number of degrees of freedom, the complexity of the obstacles, and the computational resources available. We are particularly interested in three properties of a discretization method and/or search algorithm:

1. **Completeness:** if a path exists in continuous space, whether our algorithm will find a path.
2. **Optimality:** if a path exists, whether our algorithm will find the shortest one.
3. **Computational complexity:** how the running time grows as a function of the problem size. Many algorithms work fine for small numbers of dimensions (such as 3 or 4), but don't scale well to higher dimensions.

Our approach will be to apply a **roadmap** methods, also called **skeletonization** methods. Specifically, we will create a discrete graph where vertices in the graph correspond to points, called **landmarks**, in the configuration space. The edges in the graph will correspond to paths (such as straight lines in configuration space) between the landmarks.

3.1 Grid discretization

To apply a **grid discretization**, we choose the landmarks to be the set of points lying in a regular grid in the configuration space (discarding points that lie in obstacles rather than free space). One vertex of our discrete graph will correspond to each of these landmarks. In addition, two adjacent vertices in the grid are connected if there is a straight-line path between them that lies entirely in free space.

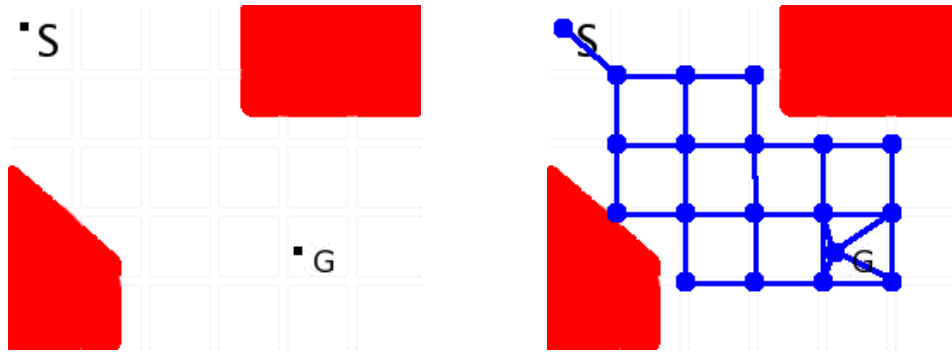


Figure 3: (left) An example of using a grid to select landmarks in a configuration space. The obstacles are shown in red. (right) The resulting graph is superimposed.

For any fixed resolution grid, this method is unfortunately not complete or optimal.² Grid based discretization pays a hefty price, however, in terms of computational complexity. For instance, if we have a 3D configuration space, and discretize each axis using 512 values, we get $512^3 \approx 1.3 \times 10^8$ vertices. For 10 dimensions, if we discretize each dimension with just 100 values, we get on the order of 10^{20} grid cells.

Grid based discretization often works fine for low dimensional problems (say 2D to 4D configuration spaces). For slightly higher dimensional problems (say 5D-6D) you can sometimes get it to work if you're clever and choose the grid very carefully. But for problems much higher dimensional than that, grid discretization usually does not work well, because of the exponential blowup in the number of dimensions.

3.2 Visibility graph

Particularly in high dimensional configuration spaces, using a regular grid results in choosing far too many landmarks. This and the following section describe two ways for choosing landmarks that result in a significantly smaller number of them.

²By modifying the algorithm to keep on trying finer and finer grids, it is possible to get the algorithm to be complete—since if we sample finely enough, we will get good enough resolution to find a path if one exists. By modifying the algorithm even further (to connect all pairs of cells for which the straight-line path between them is entirely in free space, rather than only immediately adjacent landmarks), it is also possible to get the algorithm to become arbitrarily close to optimal, since as we use finer and finer discretizations, there will exist a path arbitrarily close to the optimal one.

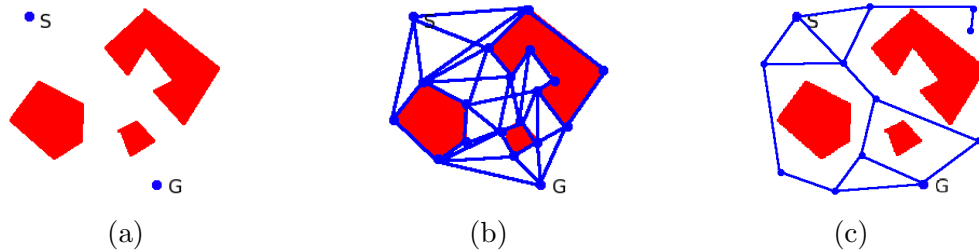


Figure 4: (a) A path planning problem, where we need to get from initial state S to goal state G . (b) The visibility graph for this planning problem. (c) An example of a probabilistic roadmap. For clarity, not all edges are shown.

In the **visibility graph** method we assume that all the obstacles in the configuration space are polygons.³ We choose as landmarks all of the vertices of all of the obstacles in the configuration space (plus the start and goal states).⁴ We then draw edges between all pairs of vertices such that one is “visible” from the other, i.e., that can be connected by a straight line. An example is shown in Figure 4 (b). It is possible to prove that the visibility graph method is complete.

In general, most obstacles are not, however, polygons (for example, see Figure 1b). So the visibility graph method is not very widely applicable.

3.3 Probabilistic Roadmap planning

The **Probabilistic Roadmap (PRM)** algorithm is due to Jean-Claude Latombe, and is widely used for many classes of robots, such as many robot arms.

It is a randomized planning algorithm, in which we choose points at random in the free space to be landmarks. Concretely, we sample points at random from the configuration space, and then discard the ones which lie within obstacles; this leaves us a set of points lying in free space, we will be the landmarks. We then check each pair of (say) nearby landmarks to see if

³If you really want to get the terminology right, polygons are 2D; their generalization to 3D is called polyhedrals; and their generalization to arbitrary numbers of dimensions is called polytopes.

⁴One problem with this approach is that the resulting path tends to follow the edges of the obstacles; this corresponds to having the robot just barely brush against the edges of the obstacles, and is undesirable, since robots can’t be controlled to perfect accuracy. In practice, for robustness, the obstacles are usually expanded slightly when the graph is generated.

they can be connected by a straight line that lies entirely in free space. An example is shown in 4 (c).

In the PRM algorithm, we can't guarantee completeness or optimality, because we could, in principle, get extremely unlucky and choose a very bad set of landmarks. However, it is possible to make probabilistic guarantees on completeness. For instance, assuming that the obstacles are widely spaced apart, it is possible to show that a path from the initial state to the goal will be found with very high probability (assuming one exists) so long as the number of samples is sufficiently large.

Finally, note that in all of these algorithms, we don't ever explicitly "compute" what the entire configuration space is or otherwise come up with a complete explicit representation of what are the free space and obstacles in configuration space. Thus, even though we've been drawing examples of 2D configuration spaces in these lecture notes and on the chalkboard during lectures in order to illustrate these ideas, you wouldn't actually write a program to "draw" out an entire k -dimensional configuration space (which is not realistic to do anyway when k is large, because of the exponential scaling in k). Specifically, note that in order to implement PRMs, all we need are (i) A way to sample points randomly in configuration space, (ii) A way to test if each of these points lies in free space or obstacle, and (iii) A way to test if the straight line between a pair of these points also lies entirely within free space. Step (i) is easily done using a random number generator, and steps (ii) and (iii) are standard geometric calculations (for example, in the case of a robot arm, step (ii) would typically require only checking whether the robot will intersect either with itself or with any obstacles in its workspace when all its joint angles are set to specific values); and there're many free software packages that you can use to perform these purely geometric computations.