Chapter

# 5 Navigation

*the process of directing a vehicle so as to reach the intended destination*
IEEE Standard 172-1983

Robot navigation is the problem of guiding a robot towards a goal. The human approach to navigation is to make maps and erect signposts, and at first glance it seems obvious that robots should operate the same way. However many robotic tasks can be achieved without any map at all, using an approach referred to as *reactive navigation*. For example, navigating by heading towards a light, following a white line on the ground, moving through a maze by following a wall, or vacuuming a room by following a random path. The robot is reacting directly to its environment: the intensity of the light, the relative position of the white line or contact with a wall. Grey Walter's tortoise Elsie from page 95 demonstrated "life-like" behaviors – she *reacted* to her environment and could seek out a light source. Today tens of millions of robotic vacuum cleaners are cleaning floors and most of them do so without using any map of the rooms in which they work. Instead they do the job by making random moves and sensing only that they have made contact with an obstacle as shown in Fig. 5.1.

Human-style *map-based navigation* is used by more sophisticated robots and is also known as motion planning. This approach supports more complex tasks but is itself more complex. It imposes a number of requirements, not the least of which is having a map of the environment. It also requires that the robot's position is always known. In the next chapter we will discuss how robots can determine their position and create maps. The remainder of this chapter discusses the reactive and map-based approaches to robot navigation with a focus on wheeled robots operating in a planar environment.

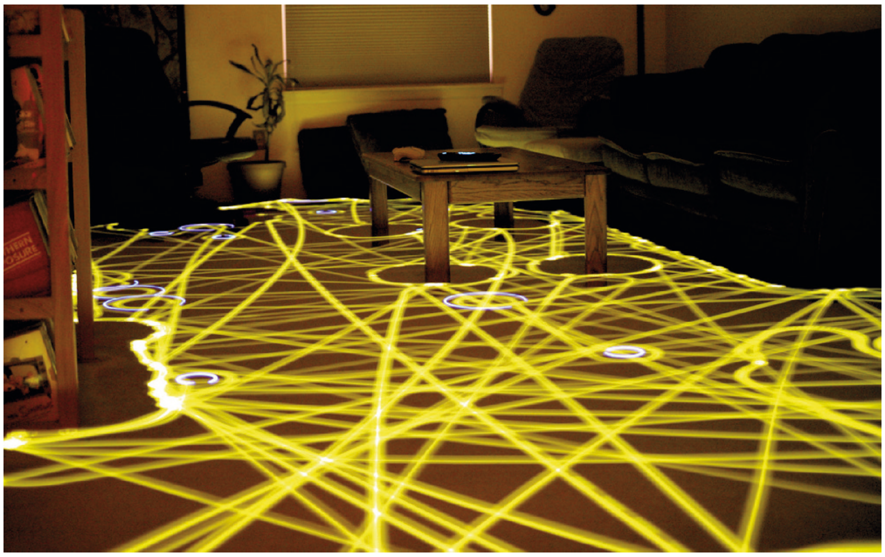
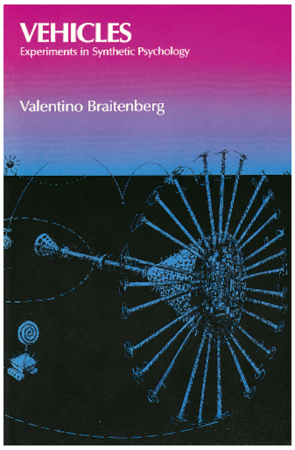


**Fig. 5.1.**
Time lapse photograph of a Roomba robot cleaning a room (photo by Chris Bartlett)

**Valentino Braitenberg (1926–2011)** was an Italian-Austrian neuroscientist and cyberneticist, and former director at the Max Planck Institute for Biological Cybernetics in Tübingen, Germany. His 1986 book "*Vehicles: Experiments in Synthetic Psychology*" (image on right is the cover of this book, published by The MIT Press, ©MIT 1984) describes reactive goal-achieving vehicles, and such systems are now commonly known as Braitenberg Vehicles.

A Braitenberg vehicle is an automaton which combines sensors, actuators and their direct interconnection to produce goal-oriented behaviors. In the book these vehicles are described conceptually as analog circuits, but more recently small robots based on a digital realization of the same principles have been developed. Grey Walter's tortoise predates the use of this term but was nevertheless an example of such a vehicle.



## 5.1 Reactive Navigation

Surprisingly complex tasks can be performed by a robot even if it has no map and no real *idea* about where it is. As already mentioned robotic vacuum cleaners use only random motion and information from contact sensors to perform a complex task as shown in Fig. 5.1. Insects such as ants and bees gather food and return it to their nest based on input from their senses, they have far too few neurons to create any kind of mental map of the world and plan paths through it. Even single-celled organisms such as flagellate protozoa exhibit goal-seeking behaviors. In this case we need to temporarily modify our earlier definition of a robot to

*a goal oriented machine that can sense, ~~plan~~ and act.*

Grey Walter's robotic tortoise demonstrated that it could moves toward a light source, a behavior known as phototaxis.▸ This was an important result in the then emerging scientific field of cybernetics.

More generally a *taxis* is the response of an organism to a stimulus gradient.

### 5.1.1 Braitenberg Vehicles

A very simple class of goal achieving robots are known as Braitenberg vehicles and are characterized by direct connection between sensors and motors. They have no explicit internal representation of the environment in which they operate and nor do they make explicit plans.▸

This is a fine philosophical point, the plan could be considered to be implicit in the details of the connections between the motors and sensors.

Consider the problem of a robot moving in two dimensions that is seeking the local maxima of a scalar field – the field could be light intensity or the concentration of some chemical.▸ The Simulink® model

This is similar to the problem of moving to a point discussed in Sect. 4.1.1.1.

```
>> sl_braitenberg
```

shown in Fig. 5.2 achieves this using a steering signal derived directly from the sensors.▸

This is similar to Braitenberg's Vehicle 4a.

**William Grey Walter (1910–1977)** was a neurophysiologist and pioneering cyberneticist born in Kansas City, Missouri and studied at King's College, Cambridge. Unable to obtain a research fellowship at Cambridge, he worked on neurophysiological research in hospitals in London and from 1939 at the Burden Neurological Institute in Bristol. He developed electro-encephalographic brain topography which used multiple electrodes on the scalp and a triangulation algorithm to determine the amplitude and location of brain activity.

Walter was influential in the then new field of cybernetics. He built robots to study how complex reflex behavior could arise from neural interconnections. His tortoise Elsie (of the species *Machina Speculatrix*) is shown, without its shell, on page 95. Built in 1948 Elsie was a three-wheeled robot capable of phototaxis that could also find its way to a recharging station. A second generation tortoise (from 1951) is in the collection of the Smithsonian Institution. He published popular articles in "Scientific American" (1950 and 1951) and a book "The Living Brain" (1953). He was badly injured in a car accident in 1970 from which he never fully recovered. (Image courtesy Reuben Hoggett collection)

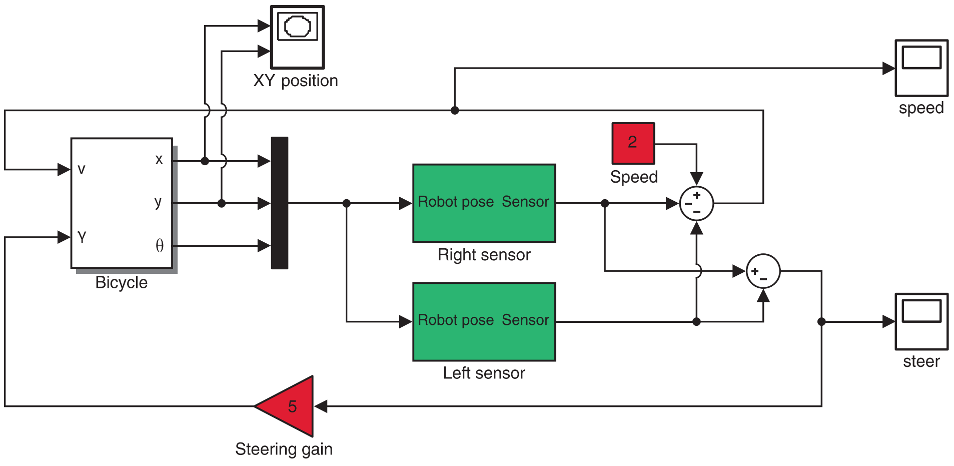


**Fig. 5.2.** The Simulink® model `sl_braitenberg` drives the vehicle toward the maxima of a provided scalar function. The vehicle plus controller is an example of a Braitenberg vehicle

To ascend the gradient we need to estimate the gradient direction at the current location and this requires at least two measurements of the field. In this example we use two sensors, bilateral sensing, with one on each side of the robot's body. The sensors are modeled by the green sensor blocks shown in Fig. 5.2 and are parameterized by the position of the sensor with respect to the robot's body, and the sensing function. In this example the sensors are at ±2 units in the vehicle's lateral or $y$-direction.

We can make the measurements simultaneously using two spatially separated sensors or from one sensor over time as the robot moves.

The field to be sensed is a simple inverse square field defined by

```
function sensor = sensorfield(x, y)
    xc = 60; yc = 90;
    sensor = 200./((x-xc).^2 + (y-yc).^2 + 200);
```

which returns the sensor value $s(x, y) \in [0, 1]$ which is a function of the sensor's position in the plane. This particular function has a peak value at the point (60, 90).

The vehicle speed is

$$v = 2 - s_R - s_L$$

where $s_R$ and $s_L$ are the right and left sensor readings respectively. At the goal, where $s_R = s_L = 1$ the velocity becomes zero.

Steering angle is based on the difference between the sensor readings

$$\gamma = k(s_R - s_L)$$

so when the field is equal in the left- and right-hand sensors the robot moves straight ahead.

Similar strategies are used by moths whose two antennae are exquisitely sensitive odor detectors that are used to steer a male moth toward a pheromone emitting female.

We start the simulation from the Simulink menu or the command line

```
>> sim('sl_braitenberg');
```

and the path of the robot is shown in Fig. 5.3. The starting pose can be changed through the parameters of the `Bicycle` block. We see that the robot turns toward the goal and slows down as it approaches, asymptotically achieving the goal position.

This particular sensor-action control law results in a specific robotic *behavior*. We could add additional logic to the robot to detect that it had arrived near the goal and then switch to a stopping behavior. An obstacle would block this robot since its only behavior is to steer toward the goal, but an additional behavior could be added to handle this case and drive around an obstacle. We could add another behavior to search randomly for the source if none was visible. Grey Walter's tortoise had four behaviors and switching was based on light level and a touch sensor.

Multiple behaviors and the ability to switch between them leads to an approach known as behavior-based robotics. The subsumption architecture was proposed as a

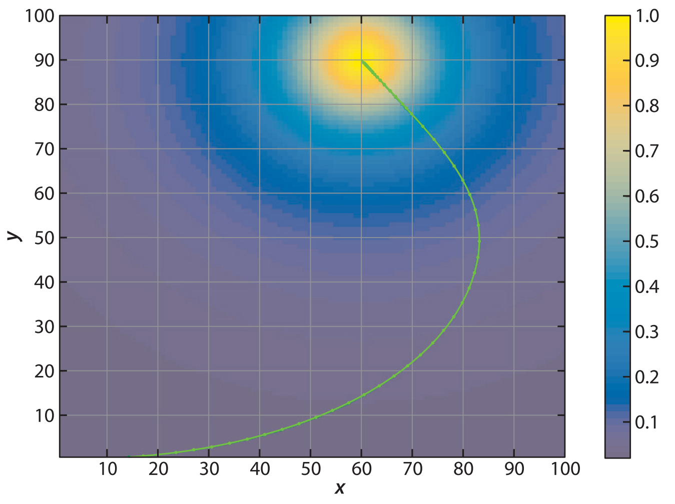


**Fig. 5.3.** Path of the Braitenberg vehicle moving toward the maximum of a 2D scalar field whose magnitude is shown color coded

means to formalize the interaction between different behaviors. Complex, some might say *intelligent looking*, behaviors can be manifested by such systems. However as more behaviors are added the complexity of the system grows rapidly and interactions between behaviors become more complex to express and debug. Ultimately the penalty of not using a map becomes too great.

### 5.1.2 Simple Automata

Another class of reactive robots are known as *bugs* – simple automata that perform goal seeking in the presence of nondriveable areas or obstacles. There are a large number of *bug* algorithms and they share the ability to sense when they are in proximity to an obstacle. In this respect they are similar to the Braitenberg class vehicle, but the *bug* includes a state machine and other logic in between the sensor and the motors. The automata have memory which our earlier Braitenberg vehicle lacked.▸ In this section we will investigate a specific *bug* algorithm known as *bug2*.

Braitenberg's book describes a series of increasingly complex vehicles, some of which incorporate memory. However the term *Braitenberg vehicle* has become associated with the simplest vehicles he described.

We start by loading an obstacle field to challenge the robot

```
>> load house
>> about house
house [double] : 397x596 (1.9 MB)
```

which defines a matrix variable `house` in the workspace. The elements are zero or one representing free space or obstacle respectively and this is shown in Fig. 5.4. Tools to generate such maps are discussed on page 131. This matrix is an example of an occupancy grid which will be discussed further in the next section. This command also loads a list of named places within the house, as elements of a structure

```
>> place
place =
     kitchen: [320 190]
      garage: [500 150]
         br1: [50 220]
            .
            .
```

At this point we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot is capable of omnidirectional motion and can move to any of its eight neighboring grid cells. Thirdly, it is able to determine its position on the plane which is a nontrivial problem that will be discussed in detail in Chap. 6. Finally, the robot can only sense its goal and whether adjacent cells are occupied.

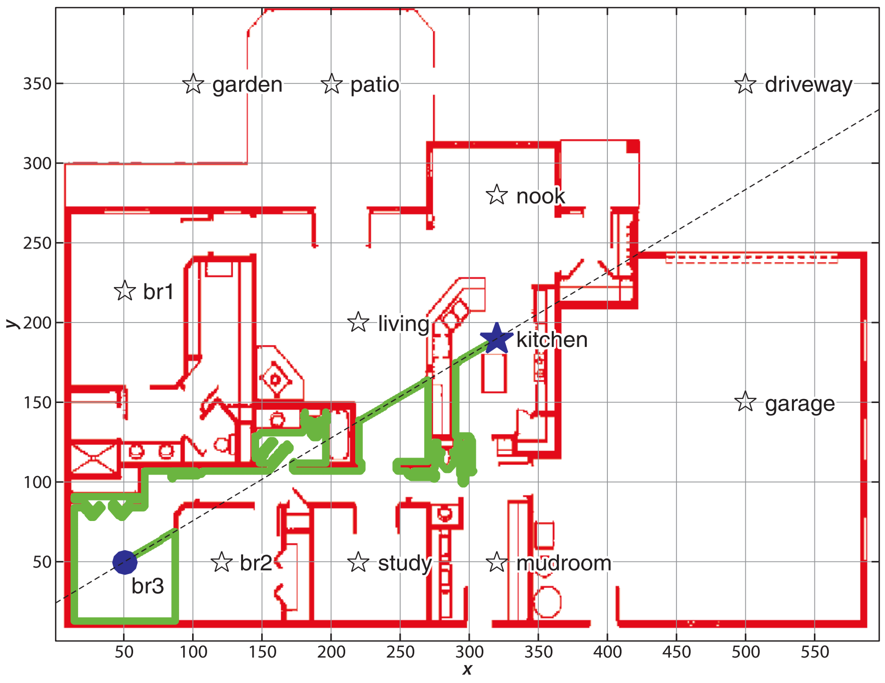


**Fig. 5.4.** Obstacles are indicated by *red pixels*. Named places are indicated by *hollow black stars*. Approximate scale is 4.5 cm per cell. The start location is a *solid blue circle* and the goal is a *solid blue star*. The path taken by the bug2 algorithm is marked by a *green line*. The *black dashed line* is the m-line, the direct path from the start to the goal

We create an instance of the `bug2` class

```
>> bug = Bug2(house);
```

and pass in the occupancy grid. The *bug2* algorithm does not use the map to plan a path – the map is used by the simulator to provide sensory inputs to the robot. We can display the robot's environment by

```
>> bug.plot();
```

The simulation is run using the `query` method

```
>> bug.query(place.br3, place.kitchen, 'animate');
```

whose arguments are the start and goal positions of the robot within the house.

The method displays an animation of the robot moving toward the goal and the path is shown as a series of green dots in Fig. 5.4.

The strategy of the *bug2* algorithm is quite simple. It is given a straight line – the m-line – towards its goal. If it encounters an obstacle it turns right and continues until it encounters a point on the m-line that is closer to the goal than when it departed from the m-line.◂

It could be argued that the m-line represents an explicit plan. Thus *bug* algorithms occupy a position somewhere between Braitenberg vehicles and map-based planning systems in the spectrum of approaches to navigation.

If an output argument is specified

```
>> p = bug.query(place.br3, place.kitchen)
```

it returns the path as a matrix `p`

```
>> about p
p [double] : 1299x2 (20.8 kB)
```

which has one row per point, and comprises 1 299 points for this example. Invoking the function with an empty matrix

```
>> p = bug.query([], place.kitchen);
```

will prompt for the corresponding point to be selected by clicking on the plot.

In this example the *bug2* algorithm has reached the goal but it has taken a very suboptimal route, traversing the inside of a wardrobe, behind doors and visiting two bathrooms. It would perhaps have been quicker in this case to turn left, rather than right, at the first obstacle but that strategy might give a worse outcome somewhere else. Many variants of the *bug* algorithm have been developed, but while they improve the performance for one type of environment they can degrade performance in others. Fundamentally the robot is limited by not using a map. It cannot see the big picture and therefore takes paths that are locally, rather than globally, optimal.

## 5.2 Map-Based Planning

The key to achieving the *best* path between points A and B, as we know from everyday life, is to use a map. Typically best means the shortest distance but it may also include some penalty term or cost related to traversability which is how easy the terrain is to drive over – it might be quicker to travel further but faster over better roads. A more sophisticated planner might also consider the size of the robot, the kinematics and dynamics of the vehicle and avoid paths that involve turns that are tighter than the vehicle can execute. Recalling our earlier definition of a robot as a

*goal oriented machine that can sense, plan and act,*

this section concentrates on planning.

There are many ways to represent a map and the position of the vehicle within the map. Graphs, as discussed in Appendix I, can be used to represent places and paths between them. Graphs can be efficiently searched to find a path that minimizes some measure or cost, most commonly the distance traveled. A simpler and very computer-friendly representation is the occupancy grid which is widely used in robotics.

An occupancy grid treats the world as a grid of cells and each cell is marked as occupied or unoccupied. We use zero to indicate an unoccupied cell or free space where the robot can drive. A value of one indicates an occupied or nondriveable cell. The size of the cell depends on the application. The memory required to hold the occupancy grid increases with the spatial area represented and inversely with the cell size. However for modern computers this representation is very feasible. For example a cell size $1 \times 1$ m requires▸ just 125 kbyte km$^{-2}$.

Considering a single bit to represent each cell. The occupancy grid could be compressed or could be kept on a disk with only the local region in memory.

In the remainder of this section we use code examples to illustrate several different planners and all are based on the occupancy grid representation. To create uniformity the planners are all implemented as classes derived from the `Navigation` superclass which is briefly described on page 133. The *bug2* class we used previously was also an instance of this class so the remaining examples follow a familiar pattern.

Once again we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot does not have any nonholonomic constraints and can move to any neighboring grid cell. Thirdly, it is able to determine its position on the plane. Fourthly, the robot is able to use the map to compute the path it will take.

In all examples we will use the house map introduced in the last section and find paths from bedroom 3 to the kitchen. These parameters can be varied, and the occupancy grid changed using the tools described above.

### 5.2.1 Distance Transform

Consider a matrix of zeros with just a single nonzero element representing the goal. The distance transform of this matrix is another matrix, of the same size, but the value of each element is its distance▸ from the original nonzero pixel. For robot path planning we use the default Euclidean distance. The distance transform is actually an image processing technique and will be discussed further in Chap. 12.

The distance between two points $(x_1, y_1)$ and $(x_2, y_2)$ where $\Delta_x = x_2 - x_1$ and $\Delta_y = y_2 - y_1$ can be Euclidean $\sqrt{\Delta_x^2 + \Delta_y^2}$ or CityBlock (also known as Manhattan) distance $|\Delta_x| + |\Delta_y|$.

**Making a map.** An occupancy grid is a matrix that corresponds to a region of 2-dimensional space. Elements containing zeros are free space where the robot can move, and those with ones are obstacles where the robot cannot move. We can use many approaches to create a map. For example we could create a matrix filled with zeros (representing all free space)

```
>> map = zeros(100, 100);
```

and use MATLAB operations such as

```
>> map(40:50,20:80) = 1;
```

or the MATLAB builtin matrix editor to create obstacles but this is quite cumbersome. Instead we can use the Toolbox map editor `makemap` to create more complex maps using an interactive editor

```
>> map = makemap(100)
```

that allows you to add rectangles, circles and polygons to an occupancy grid. In this example the grid is $100 \times 100$. See online help for details.

The occupancy grid in Fig. 5.4 was derived from a scanned image but online buildings plans and street maps could also be used.

Note that the occupancy grid is a matrix whose coordinates are conventionally expressed as (row, column) and the row is the vertical dimension of a matrix. We use the Cartesian convention of a horizontal $x$-coordinate first, followed by the $y$-coordinate therefore the matrix is always indexed as `y,x` in the code.

To use the distance transform for robot navigation we create a `DXform` object, which is derived from the `Navigation` class

```
>> dx = DXform(house);
```

and then create a plan to reach a specific goal

```
>> dx.plan(place.kitchen)
```

which can be visualized

```
>> dx.plot()
```

as shown in Fig. 5.5. We see the obstacle regions in red overlaid on the distance map whose grey level at any point indicates the distance from that point to the goal, in grid cells, taking into account travel *around* obstacles.

The hard work has been done and to find the shortest path from *any* point to the goal we simply consult or query the plan. For example a path from the bedroom to the goal is

For the bug2 algorithm there was no planning step so the query in that case was the simulated robot querying its proximity sensors.

```
>> dx.query(place.br3, 'animate');
```

which displays an animation of the robot moving toward the goal. The path is indicated by a series of green dots as shown in Fig. 5.5.

By convention the plan is based on the goal location and we query for a start location, but we could base the plan on the start position and then query for a goal.

The plan is the distance map. Wherever the robot starts, it moves to the neighboring cell that has the smallest distance to the goal. The process is repeated until the robot reaches a cell with a distance value of zero which is the goal.

If the `path` method is called with an output argument the path

```
>> p = dx.query(place.br3);
```

is returned as a matrix, one row per point, which we can visualize overlaid on the occupancy grid and distance map

```
>> dx.plot(p)
```

The path comprises

```
>> numrows(p)
ans =
   336
```

points which is considerably shorter than the path found by *bug2*.

This navigation algorithm has exploited its global view of the world and has, through exhaustive computation, found the shortest possible path. In contrast, *bug2* without

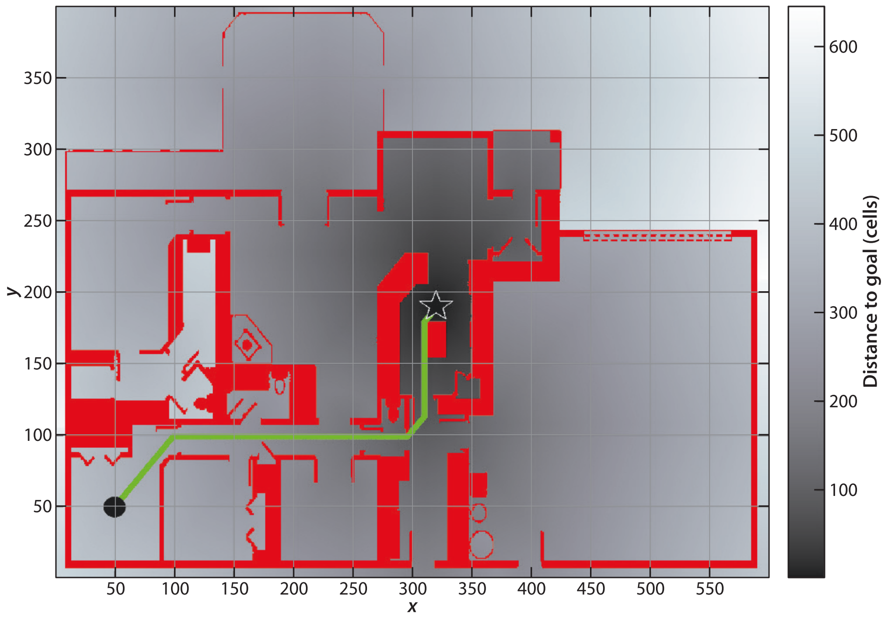


**Fig. 5.5.**
The distance transform path. Obstacles are indicated by *red cells*. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

the global view has just bumped its way through the world. The penalty for achieving the optimal path is computational cost. This particular implementation of the distance transform is iterative. Each iteration has a cost of $O(N^2)$ and the number of iterations is at least $O(N)$, where $N$ is the dimension of the map.

We can visualize the iterations of the distance transform by

```
>> dx.plan(place.kitchen, 'animate');
```

which shows the distance values propagating as a wavefront outward from the goal. The wavefront moves outward, spills through doorways into adjacent rooms and outside the house.▸ Although the plan is expensive to create, once it has been created it can be used to plan a path from *any* initial point to that goal.

More efficient algorithms exist such as fast marching methods and Dijkstra's method, but the iterative wavefront method used here is easy to code and to visualize.

We have converted a fairly complex planning problem into one that can now be handled by a Braitenberg-class robot that makes local decisions based on the distance to the goal. Effectively the robot is rolling *downhill* on the distance function which we can plot as a 3D surface

```
>> dx.plot3d(p)
```

shown in Fig. 5.6 with the robot's path and room locations overlaid.

For large occupancy grids this approach to planning will become impractical. The roadmap methods that we discuss later in this chapter provide an effective means to find paths in large maps at greatly reduced computational cost.

The scale associated with this occupancy grid is 4.5 cm per cell and we have assumed the robot occupies a single grid cell – this is a very small robot. The planner could therefore find paths that a larger real robot would be unable to fit through. A common solution to this problem is to *inflate* the occupancy grid – making the obstacles bigger is equivalent to leaving the obstacles unchanged and making the robot bigger. For example, if we inflate the obstacles by 5 cells

```
>> dx = DXform(house, 'inflate', 5);
>> dx.plan(place.kitchen);
>> p = dx.query(place.br3);
>> dx.plot(p)
```

the path shown in Fig. 5.7b now takes the wider corridors to reach its goal. To illustrate how this works we can overlay this new path on the inflated occupancy grid

```
>> dx.plot(p, 'inflated');
```

This is morphological dilation which is discussed in Sect. 12.6.

and this is shown in Fig. 5.7a. The inflation parameter of 5 has grown the obstacles by 5 grid cells in all directions, a bit like applying a very thick layer of paint. This is equivalent to growing the robot by 5 grid cells in all directions – the robot grows from a single grid cell to a disk with a diameter of 11 cells which is equivalent to around 50 cm.

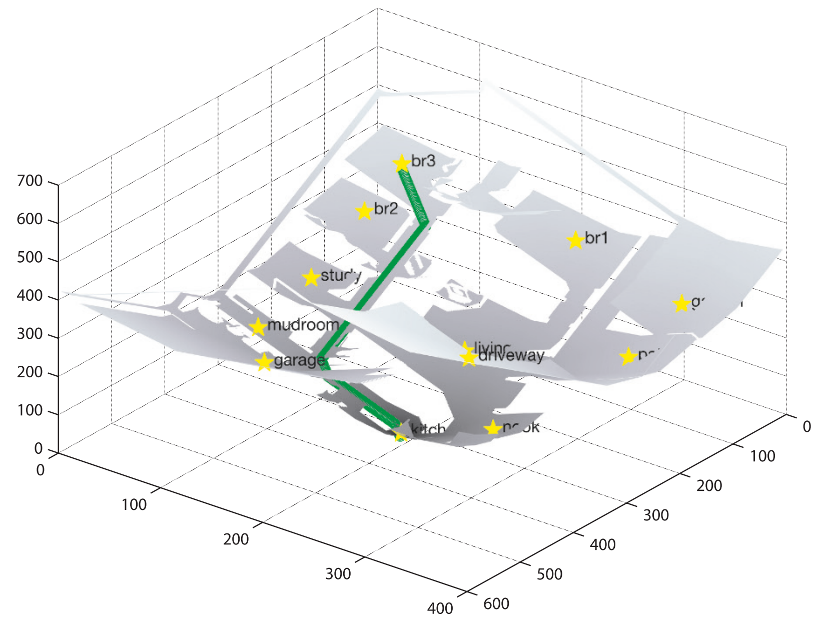


**Fig. 5.6.** The distance transform as a 3D function, where height is distance from the goal. Navigation is simply a downhill run. Note the discontinuity in the distance transform where the split wavefronts met

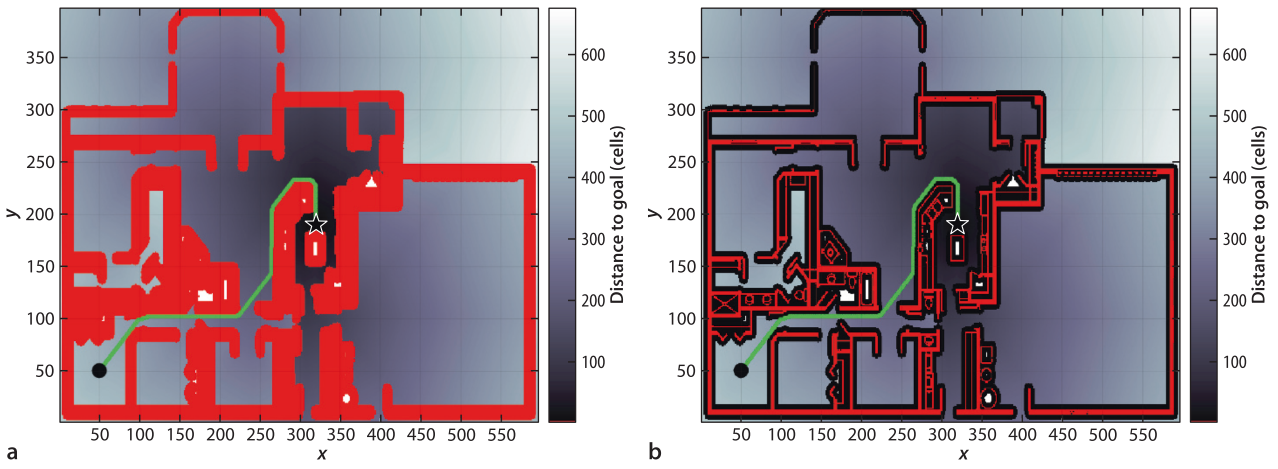


**Fig. 5.7.** Distance transform path with obstacles inflated by 5 cells. **a** Path shown with inflated obstacles; **b** path computed for inflated obstacles overlaid on original obstacle map, black regions are where no distance was computed

**Navigation superclass.** The examples in this chapter are all based on classes derived from the `Navigation` class which is designed for 2D grid-based navigation. Each example consists of essentially the following pattern. Firstly we create an instance of an object derived from the `Navigation` class by calling the class constructor.

```
nav = MyNavClass(map)
```

which is passed the occupancy grid. Then a plan is computed

```
nav.plan()
nav.plan(goal)
```

and depending on the planner the goal may or may not be required. A path from an initial position to the goal is computed by

```
p = nav.query(start, goal)
p = nav.query(start)
```

again depending on whether or not the planner requires a goal. The optional return value `p` is the path, a sequence of points from `start` to `goal`, one row per point, and each row comprises the *x*- and *y*-coordinate. If `start` or `goal` is given as [] the user is prompted to interactively click the point. The 'animate' option causes an animation of the robot's motion to be displayed.

The map and planning information can be visualized by

```
nav.plot()
```

or have a path overlaid

```
nav.plot(p)
```

### 5.2.2 D*

A popular algorithm for robot path planning is D* which finds the best path▶ through a graph, which it first computes, that corresponds to the input occupancy grid. D* has a number of features that are useful for real-world applications. Firstly, it generalizes the occupancy grid to a cost map which represents the cost $c \in \mathbb{R}$, $c > 0$ of traversing each cell in the horizontal or vertical direction. The cost of traversing the cell diagonally is $c\sqrt{2}$. For cells corresponding to obstacles $c = \infty$ (`Inf` in MATLAB).

D* is an extension of the A* algorithm for finding minimum cost paths through a graph, see Appendix I.

Secondly, D* supports incremental replanning. This is important if, while we are moving, we discover that the world is different to our map. If we discover that a route has a higher than expected cost or is completely blocked we can incrementally replan to find a better path. The incremental replanning has a lower computational cost than completely replanning as would be required using the distance transform method just discussed.

D* finds the path which minimizes the total cost of travel. If we are interested in the shortest time to reach the goal then cost is the time to drive across the cell and is inversely related to traversability. If we are interested in minimizing damage to the vehicle or maximizing passenger comfort then cost might be related to the roughness of the terrain within the cell. The costs assigned to cells will also depend on the characteristics of the vehicle: a large 4-wheel drive vehicle may have a finite cost to cross a rough area whereas for a small car that cost might be infinite.

To implement the D* planner using the Toolbox we use a similar pattern and first create a D* navigation object

```
>> ds = Dstar(house);
```

The D* planner converts the passed occupancy grid `map` into a cost map which we can retrieve

```
>> c = ds.costmap();
```

where the elements of `c` will be 1 or $\infty$ representing free and occupied cells respectively.

A plan for moving to the goal is generated by

```
>> ds.plan(place.kitchen);
```

which creates a very dense directed graph (see Appendix I). Every cell is a graph vertex and has a cost, a distance to the goal, and a link to the neighboring cell that is closest to the goal. Each cell also has a state $t \in$ {NEW, OPEN, CLOSED}. Initially every cell is in the NEW state, the cost of the goal cell is zero and its state is OPEN. We can consider the set of all cells in the OPEN state as a wavefront propagating outward from the goal.▶ The cost of

The distance transform also evolves as a wavefront outward from the goal. However D* represents the frontier efficiently as a list of cells whereas the distance transform computes the frontier on a per-cell basis at every iteration – the frontier is implicitly where a cell with infinite cost (the initial value of all cells) is adjacent to a cell with finite cost.

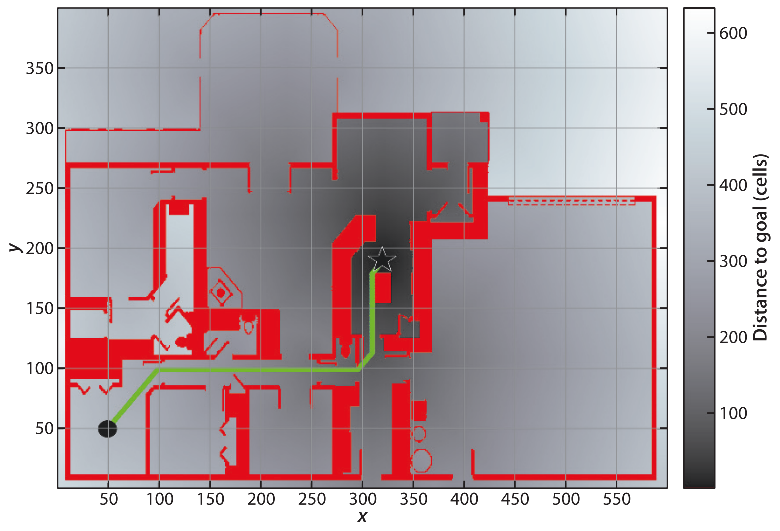


**Fig. 5.8.**
The D* planner path. Obstacles are indicated by *red cells* and all driveable cells have a cost of 1. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

reaching cells that are neighbors of an OPEN cell is computed and these cells in turn are set to OPEN and the original cell is removed from the open list and becomes CLOSED. In MATLAB this initial planning phase is quite slow◂ and takes over a minute and

D* is more efficient than the distance transform but it executes more slowly because it is implemented entirely in MATLAB code whereas the distance transform is a MEX-file written in C.

```
>> ds.niter
ans =
      245184
```

iterations of the planning loop.

The path from an arbitrary starting point to the goal

```
>> ds.query(place.br3);
```

is shown in Fig. 5.8. The robot has again taken a short and efficient path around the obstacles that is almost identical to that generated by the distance transform.

The real power of D* comes from being able to efficiently change the cost map during the mission. This is actually quite a common requirement in robotics since real sensors have a finite range and a robot discovers more of world as it proceeds. We inform D* about changes using the `modify_cost` method, for example to raise the cost of entering the kitchen via the bottom doorway

```
>> ds.modify_cost( [300,325; 115,125], 5 );
```

we have raised the cost to 5 for a small rectangular region across the doorway. This region is indicated by the yellow dashed rectangle in Fig. 5.9. The other driveable cells have a default cost of 1. The plan is updated by invoking the planning algorithm again

```
>> ds.plan();
```

and this time the number of iterations is only

```
>> ds.niter
ans =
      169580
```

which is 70% of that required to create the original plan.◂ The new path for the robot

The cost increases with the number of cells modified and the effect those changes have on the distance map. It is possible that incremental replanning takes more time than planning from scratch.

```
>> ds.query(place.br3);
```

is shown in Fig. 5.9. The cost change is relatively small but we notice that the increased cost of driving within this region is indicated by a subtle brightening of those cells – in a cost sense these cells are now further from the goal. Compared to Fig. 5.8 the robot has taken a different route to the kitchen and avoided the bottom door. D* allows updates to the map to be made at any time while the robot is moving. After replanning the robot simply moves to the adjacent cell with the lowest cost which ensures continuity of motion even if the plan has changed.

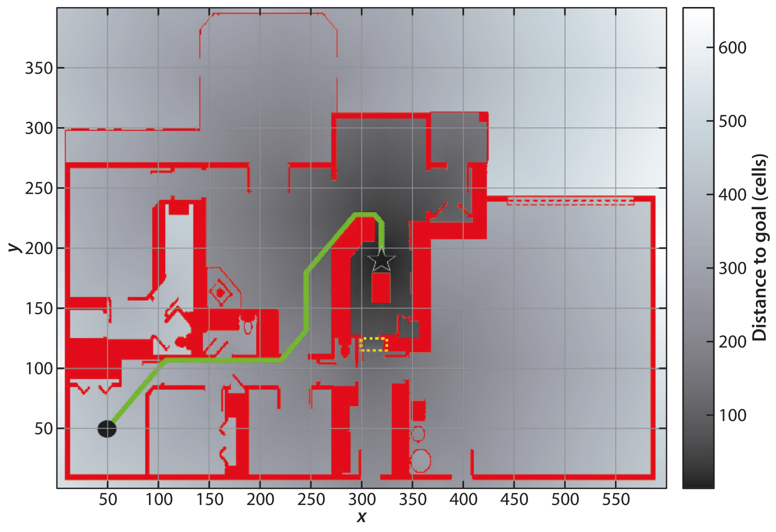
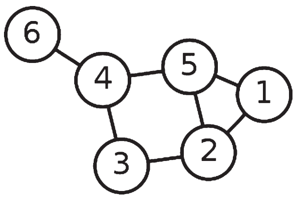


**Fig. 5.9.** Path from D* planner with modified map. The higher-cost region is indicated by the *yellow dashed rectangle* and has changed the path compared to Fig. 5.7

A graph is an abstract representation of a set of objects connected by links typically denoted $G(V, E)$ and depicted diagrammatically as shown to the right. The objects, $V$, are called vertices or nodes, and the links, $E$, that connect some pairs of vertices are called edges or arcs. Edges can be directed (arrows) or undirected as in this case. Edges can have an associated weight or cost associated with moving from one of its vertices to the other. A sequence of edges from one vertex to another is a path. Graphs can be used to represent transport or communications networks and even social relationships, and the branch of mathematics is graph theory. Minimum cost path between two nodes in the graph can be computed using well known algorithms such as Dijstrka's method or A*.
The navigation classes use a simple MATLAB graph class called `PGraph`, see Appendix I.



### 5.2.3 Introduction to Roadmap Methods

In robotic path planning the analysis of the map is referred to as the *planning phase*. The *query phase* uses the result of the planning phase to find a path from A to B. The two previous planning algorithms, distance transform and D*, require a significant amount of computation for the planning phase, but the query phase is very cheap. However the plan depends on the goal. If the goal changes the expensive planning phase must be re-executed. Even though D* allows the path to be recomputed as the costmap changes it does not support a changing goal.

The disparity in planning and query costs has led to the development of roadmap methods where the query can include both the start and goal positions. The planning phase provides analysis that supports changing starting points and changing goals. A good analogy is making a journey by train. We first find a local path to the nearest train station, travel through the train network, get off at the station closest to our goal, and then take a local path to the goal. The train network is invariant and planning a path through the train network is straightforward. Planning paths to and from the entry and exit stations respectively is also straightforward since they are, ideally, short paths. The robot navigation problem then becomes one of building a network of obstacle free paths through the environment which serve the function of the train network. In the literature such a network is referred to as a roadmap. The roadmap need only be computed once and can then be used like the train network to get us from any start location to any goal location.

We will illustrate the principles by creating a roadmap from the occupancy grid's free space using some image processing techniques. The essential steps in creating the roadmap are shown in Fig. 5.10. The first step is to find the free space in the map which is simply the complement of the occupied space

```
>> free = 1 - house
```

and is a matrix with nonzero elements where the robot is free to move. The boundary is also an obstacle so we mark the outermost cells as being not free

```
>> free(1,:) = 0; free(end,:) = 0;
>> free(:,1) = 0; free(:,end) = 0;
```

and this map is shown in Fig. 5.10a where free space is depicted as white.

The topological skeleton of the free space is computed by a morphological image processing algorithm known as thinning▸ applied to the free space of Fig. 5.10a

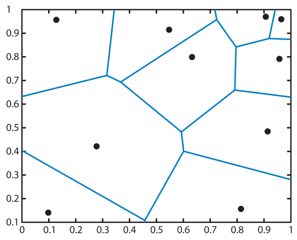
Also known as skeletonization. We will cover this topic in Sect. 12.6.3.

```
>> skeleton = ithin(free);
```

and the result is shown in Fig. 5.10b. We see that the obstacles have grown and the free space, the white cells, have become a thin network of connected white cells which are equidistant from the boundaries of the original obstacles.

Figure 5.10c shows the free space network overlaid on the original map. We have created a network of paths that span the space and which can be used for obstacle-free travel around the map.▸ These paths are the edges of a generalized Voronoi

The junctions in the roadmap are indicated by black dots. The junctions, or triple points, are identified using the morphological image processing function `triplepoint`.



The Voronoi tessellation of a set of planar points, known as sites, is a set of Voronoi cells as shown to the left. Each cell corresponds to a site and consists of all points that are closer to its site than to any other site. The edges of the cells are the points that are equidistant to the two nearest sites. A generalized Voronoi diagram comprises cells defined by measuring distances to objects rather than points. In MATLAB we can generate a Voronoi diagram by

```
>> sites = rand(10,2)
>> voronoi(sites(:,1), sites(:,2))
```

**Georgy Voronoi (1868–1908)** was a Russian mathematician, born in what is now Ukraine. He studied at Saint Petersburg University and was a student of Andrey Markov. One of his students Boris Delaunay defined the eponymous triangulation which has dual properties with the Voronoi diagram.

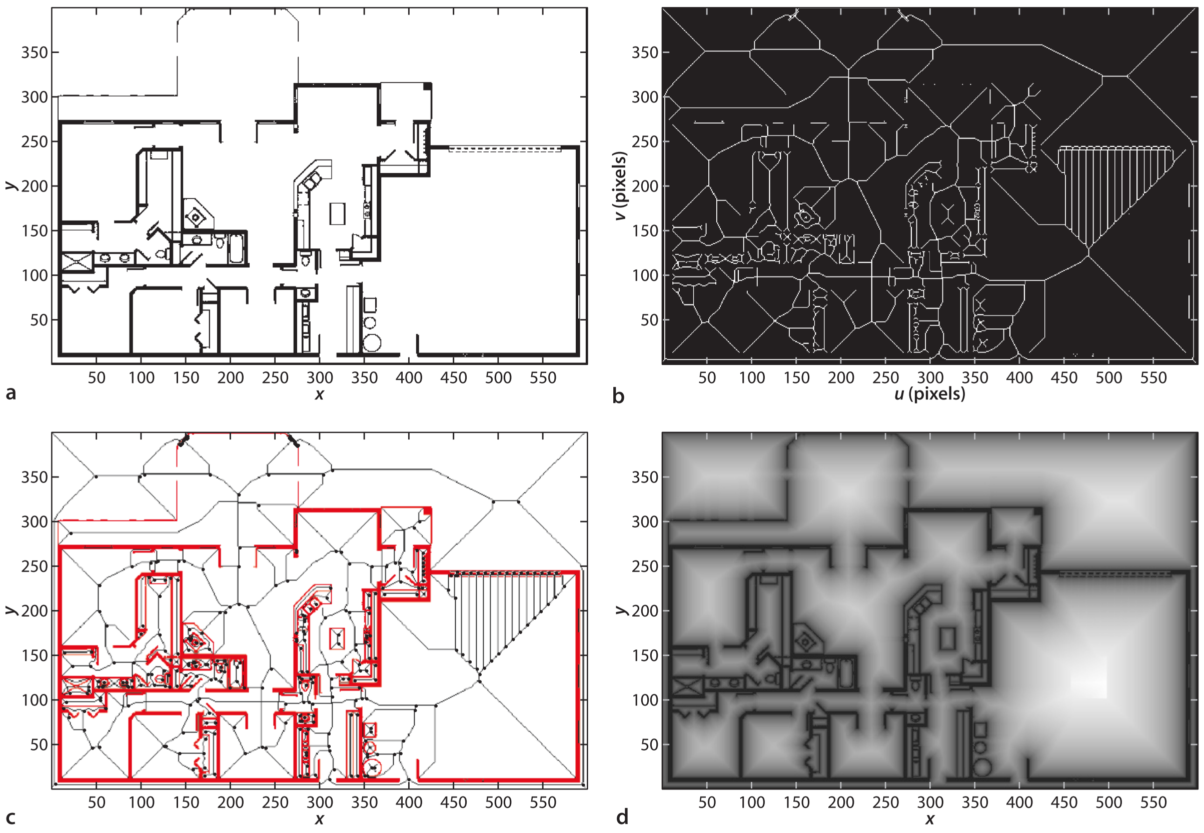


**Fig. 5.10.** Steps in the creation of a Voronoi roadmap. **a** Free space is indicated by *white cells*; **b** the skeleton of the free space is a network of adjacent cells no more than one cell thick; **c** the skeleton with the obstacles overlaid in red and road-map junction points indicated by *black dots*; **d** the distance transform of the obstacles, pixel values correspond to distance to the nearest obstacle

diagram. We could obtain a similar result by computing the distance transform of the obstacles, Fig. 5.10a, and this is shown in Fig. 5.10d. The value of each pixel is the distance to the nearest obstacle and the ridge lines correspond to the skeleton of Fig. 5.10b. Thinning or skeletonization, like the distance transform, is a computationally expensive iterative algorithm but it illustrates well the principles of finding paths through free space. In the next section we will examine a cheaper alternative.

### 5.2.4 Probabilistic Roadmap Method (PRM)

The high computational cost of the distance transform and skeletonization methods makes them infeasible for large maps and has led to the development of probabilistic methods. These methods sparsely sample the world map and the most well known of these methods is the probabilistic roadmap or PRM method.

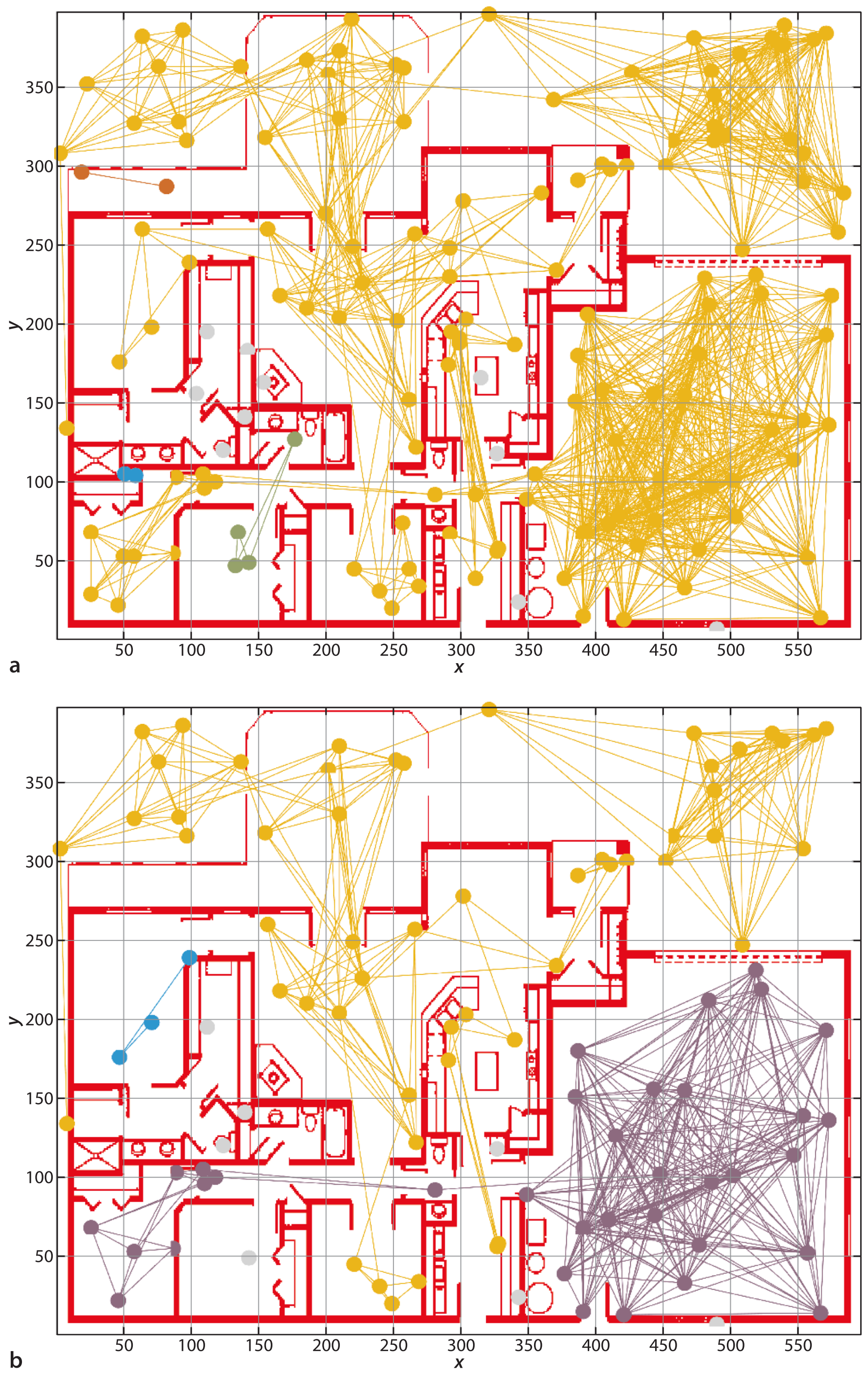


**Fig. 5.11.**
Probablistic roadmap (PRM) planner and the random graphs produced in the planning phase. **a** Well connected network with 150 nodes; **b** poorly connected network with 100 nodes

To use the Toolbox PRM planner for our problem we first create a `PRM` object

```
>> prm = PRM(house)
```

and then create the plan

```
>> prm.plan('npoints', 150)
```

To replicate the following result be sure to initialize the random number generator first using `randinit`. See page 139.

with 150 roadmap nodes. Note that we do not pass the goal as an argument since the plan is independent of the goal. Creating the path is a two phase process: planning, and query. The planning phase finds *N* random points, 150 in this case, that lie in free space. Each point is connected to its nearest neighbors by a straight line path that does not cross any obstacles, so as to create a network, or graph, with a minimal number of disjoint components and no cycles. The advantage of PRM is that relatively few points need to be tested to ascertain that the points and the paths between them are obstacle free. The resulting network is stored within the `PRM` object and a summary can be displayed

```
>> prm
prm =
PRM navigation class:
  occupancy grid: 397x596
  graph size: 150
  dist thresh: 178.8
  2 dimensions
  150 vertices
  1223 edges
  14 components
```

which indicates the number of edges and connected components in the graph. The graph can be visualized

This is derived automatically from the size of the occupancy grid.

```
>> prm.plot()
```

**Random numbers.** The MATLAB random number generator (used for `rand` and `randn`) generates a very long sequence of numbers that are an excellent approximation to a random sequence. The generator maintains an internal state which is effectively the position within the sequence. After startup MATLAB always generates the following random number sequence

```
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
>> rand
ans =
    0.1270
```

Many algorithms discussed in this book make use of random numbers and this means that the results can never be repeated. Before all such examples in this book is an invisible call to `randinit` which resets the random number generator to a known state

```
>> randinit
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
```

and we see that the random sequence has been restarted.

as shown in Fig. 5.11a. The dots represent the randomly selected points and the lines are obstacle-free paths between the points. Only paths less than 178.8 cells long are selected which is the distance threshold parameter of the `PRM` class. Each edge of the graph has an associated cost which is the distance between its two nodes. The color of the node indicates which component it belongs to and each component is assigned a unique color. In this case there are 14 components but the bulk of nodes belong to a single large component.

The query phase finds a path from the start point to the goal. This is simply a matter of moving to the closest node in the roadmap (the start node), following a minimum cost $A^*$ route through the roadmap, getting off at the node closest to the goal and then traveling to the goal. For our standard problem this is

```
>> prm.query(place.br3, place.kitchen)
>> prm.plot()
```

and the path followed is shown in Fig. 5.12. The path that has been found is quite efficient although there are two areas where the path doubles back on itself. Note that we provide the start and the goal position to the query phase. An advantage of this planner is that once the roadmap is created by the planning phase we can change the goal and starting points very cheaply, only the query phase needs to be repeated. The path taken is

```
>> p = prm.query(place.br3, place.kitchen);
>> about p
p [double] : 9x2 (144 bytes)
```

which is a list of the node coordinates that the robot passes through – via points. These could be passed to a trajectory following controller as discussed in Sect. 4.1.1.3.

There are some important tradeoffs in achieving this computational efficiency. Firstly, the underlying random sampling of the free space means that a different roadmap is created every time the planner is run, resulting in different paths and path lengths. Secondly, the planner can fail by creating a network consisting of disjoint components. The roadmap in Fig. 5.11b, with only 100 nodes has several large disconnected components and the nodes in the kitchen and bedrooms belong to different components. If the start and goal nodes are not connected by the roadmap, that is, they are close to different components the `query` method will report an error. The only solution is to rerun the planner and/or increase the number of nodes. Thirdly, long narrow gaps between obstacles such as corridors are unlikely to be exploited since the probability of randomly choosing points that lie along such spaces is very low.

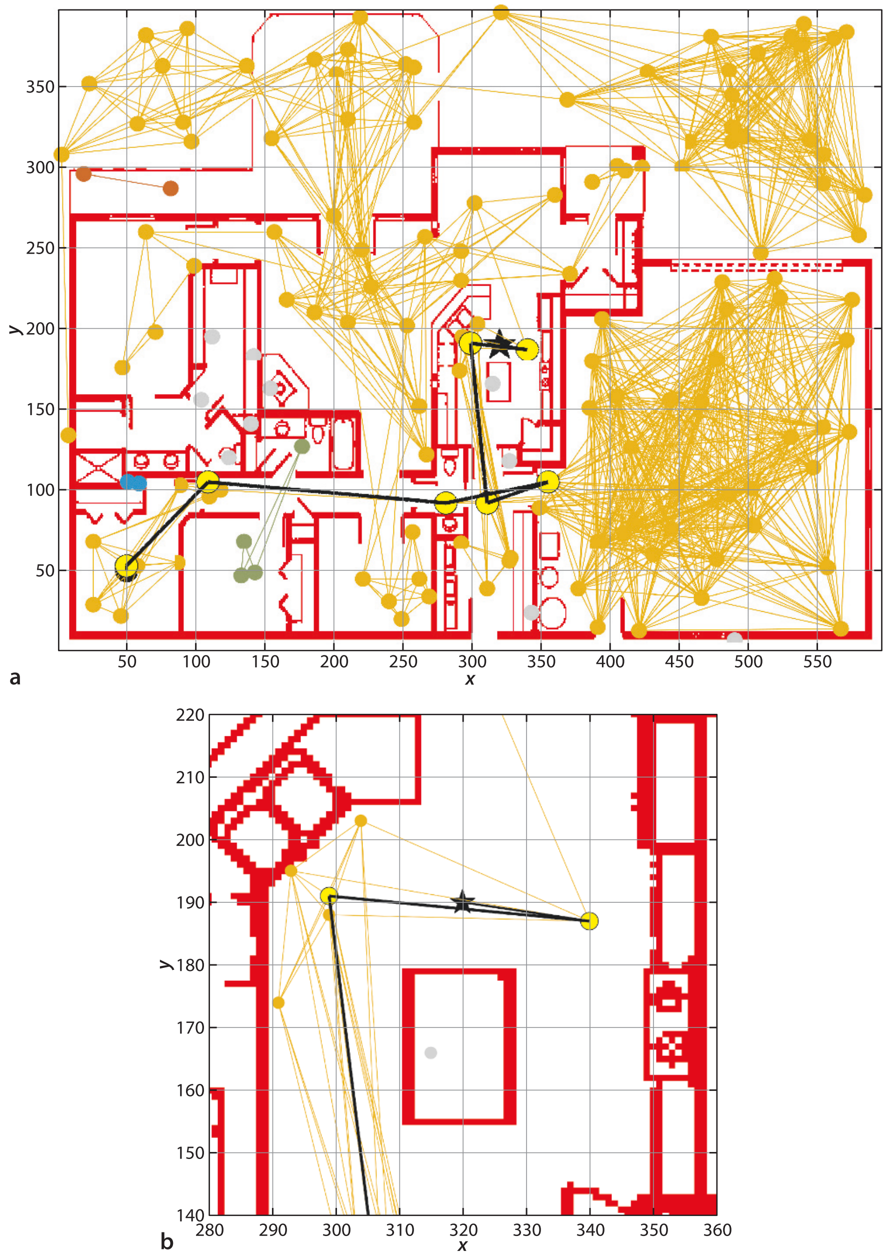


**Fig. 5.12.** Probablistic roadmap (PRM) planner **a** showing the path taken by the robot via nodes of the roadmap which are highlighted in yellow; **b** closeup view of goal region where the short path from the final roadmap node to the goal can be seen

## 5.2.5 Lattice Planner

The planners discussed so far have generated paths independent of the motion that the vehicle can actually achieve, and we learned in Chap. 4 that wheeled vehicles have significant motion constraints. One common approach is to use the output of the planners we have discussed and move a point along the paths at constant velocity and then follow that point, using techniques such as the trajectory following controller described in Sect. 4.1.1.3.

An alternative is to *design* a path from the outset that we know the vehicle can follow. The next two planners that we introduce take into account the motion model of the vehicle, and relax the assumption we have so far made that the robot is capable of omnidirectional motion.

We consider that the robot is moving between discrete points in its 3-dimensional configuration space. The robot is initially at the origin and can drive forward to the three points shown in black in Fig. 5.13a. Each path is an arc which requires a constant steering wheel setting and the arc radius is chosen so that at the end of each arc the robot's heading direction is some multiple of $\frac{\pi}{2}$ radians.

The pitch of the grid is dictated by the turning radius of the vehicle.

Sometimes called Dubins curves.

At the end of each branch we can add the same set of three motions suitably rotated and translated, and this is shown in Fig. 5.13b. The graph now contains 13 nodes and represents 9 paths each 2 segments long. We can create this lattice by using the `Lattice` planner class

```
>> lp = Lattice();
>> lp.plan('iterations', 2)
13 nodes created
>> lp.plot()
```

which will generate a plot like Fig. 5.13b. Each node represents a configuration $(x, y, \theta)$, not just a position, and if we rotate the plot we can see in Fig. 5.14 that the paths lie in the 3-dimensional configuration space.

While the paths appear smooth and continuous the curvature is in fact discontinuous – at some nodes the steering wheel angle would have to change instantaneously from hard left to hard right for example.

A real robot would take a finite time to adjust its steering angle and this would introduce some error in the robot path. The steering control system could compensate for this by turning harder later in the segment so as to bring the robot to the end point with the correct orientation.

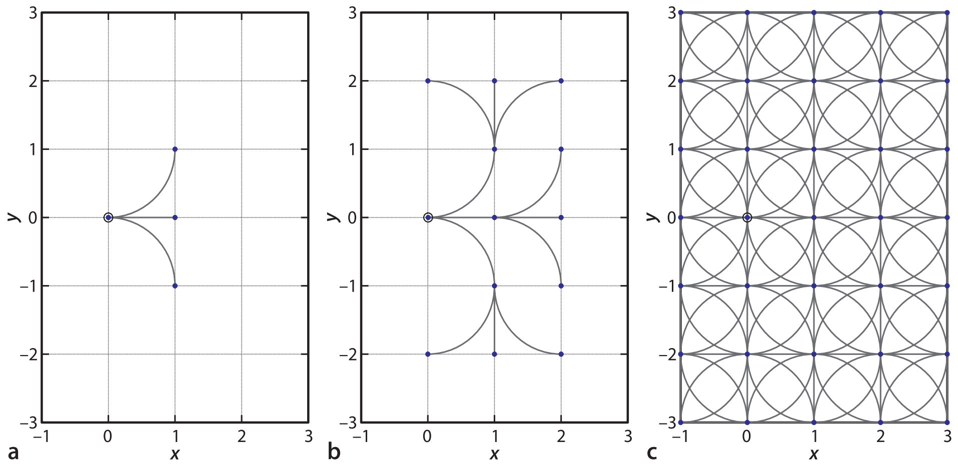


**Fig. 5.13.** Lattice plan after 1, 2 and 8 iterations

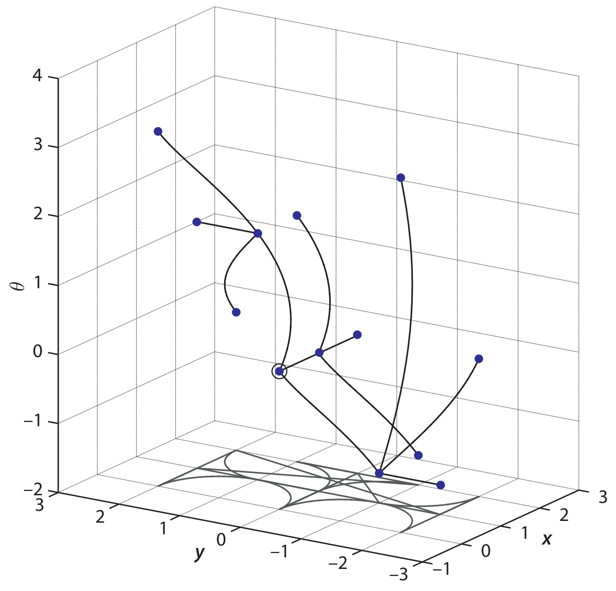


**Fig. 5.14.** Lattice plan after 2 iterations shown in 3-dimensional configuration space

By increasing the number of iterations

```
>> lp.plan('iterations', 8)
780 nodes created
>> lp.plot()
```

we can fill in more possible paths as shown in Fig. 5.13c and the paths now extend well beyond the area shown.

Now that we have created the lattice we can compute a path between any two nodes using the query method

```
>> lp.query( [1 2 pi/2], [2 -2 0] );
A* path cost 6
```

where the start and goal are specified as configurations $(x, y, \theta)$ and the lowest cost path found by an A$^*$ search is reported. We can overlay this on the vertices

Every segment in the lattice has a default cost of 1 so the cost of 6 simply reflects the total number of segments in the path. A* search is introduced in Appendix I.

```
>> lp.plot
```

and is shown in Fig. 5.15a. This is a path that takes into account the fact that the vehicle has an orientation and preferred directions of motion, as do most wheeled robot platforms. We can also access the configuration-space coordinates of the nodes

```
>> p = lp.query( [1 2 pi/2], [2 -2 0] )
A* path cost 6
>> about p
p [double] : 7x3 (168 bytes)
```

where each row represents the configuration-space coordinates $(x, y, \theta)$ of a node in the lattice along the path from start to goal configuration.

Implicit in our search for the lowest cost path is the cost of traversing each edge of the graph which by default gives equal cost to the three steering options: straight ahead, turn left and turn right. We can increase the cost associated with turning

```
>> lp.plan('cost', [1 10 10])
>> lp.query(start, goal);
A* path cost 35
>> lp.plot()
```

and now we now have the path shown in Fig. 5.15b which has only 3 turns compared to 5 previously. However the path is longer – having 8 rather than 6 segments.

Consider a more realistic scenario with obstacles in the environment. Specifically we want to find a path to move the robot 2 m in the lateral direction with its final heading angle the same as its initial heading angle

```
>> load road
>> lp = Lattice(road, 'grid', 5, 'root', [50 50 0])
>> lp.plan();
```

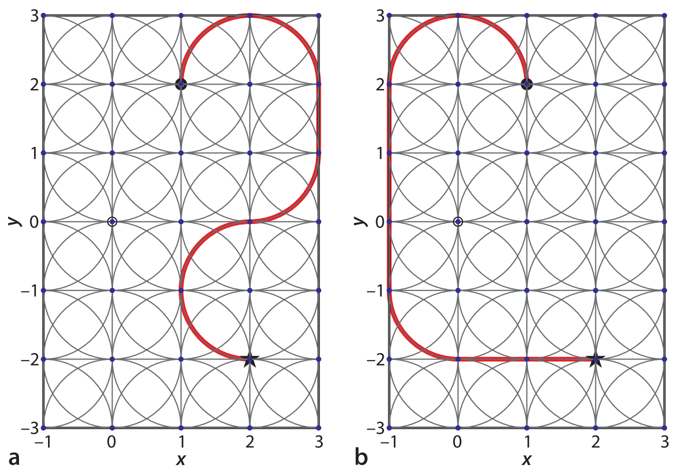


**Fig. 5.15.** Paths over the lattice graph. **a** With uniform cost; **b** with increased penalty for turns

where we have loaded an obstacle grid that represents a simple parallel-parking scenario and planned a lattice with a grid spacing of 5 units and the root node at a central obstacle-free configuration. In this case the planner continues to iterate until it can add no more nodes to the free space. We query for a path from the road to the parking spot

```
>> lp.query([30 45 0], [50 20 0])
```

and the result is shown in Fig. 5.16.

Paths generated by the lattice planner are inherently driveable by the robot but there are clearly problems driving along a diagonal with this simple lattice. The planner would generate a continual sequence of hard left and right turns which would cause undue wear and tear on a real vehicle and give a very uncomfortable ride. More sophisticated version of lattice planners are able to deal with this by using motion primitives with hundreds of arcs, such as shown in Fig. 5.17, instead of the three shown in these examples.

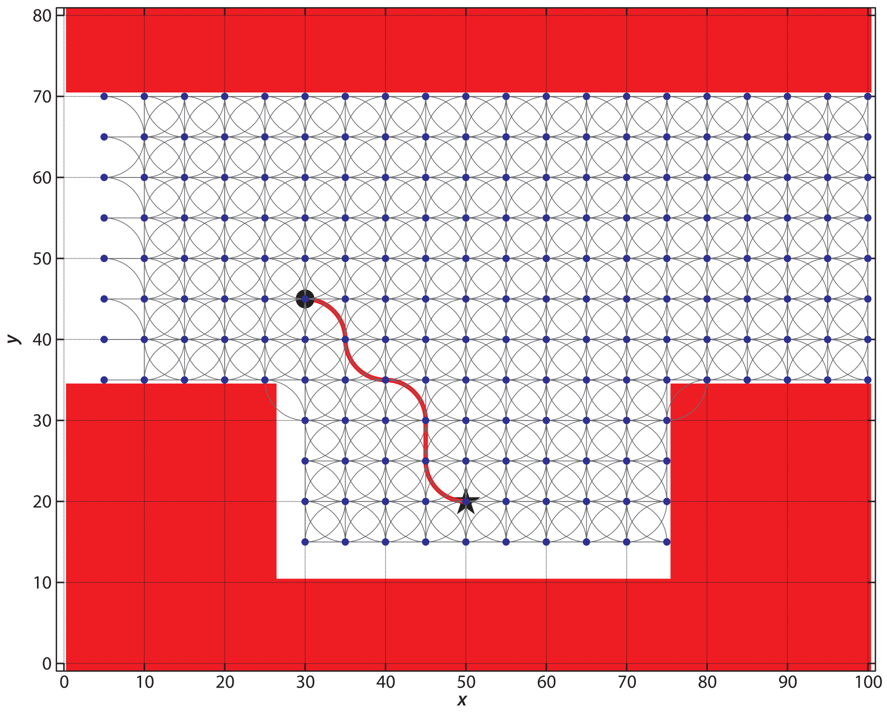


**Fig. 5.16.** A simple parallel parking scenario based on the lattice planner with an occupancy grid (cells are 10 cm square)

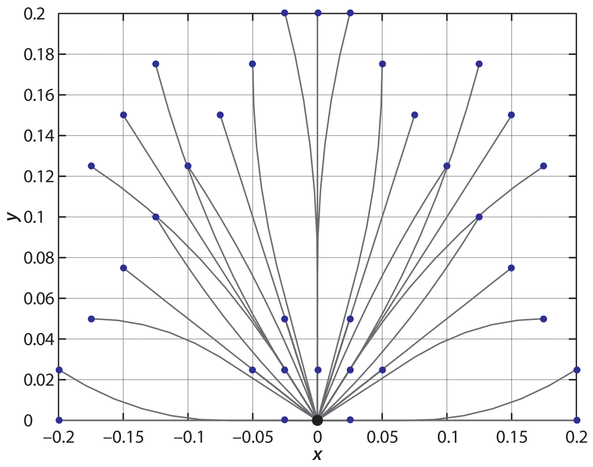
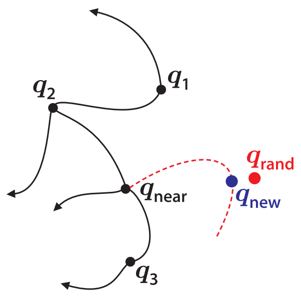


**Fig. 5.17.** A more sophisticated lattice generated by the package **sbpl** with 43 paths based on the kinematic model of a unicycle

### 5.2.6 Rapidly-Exploring Random Tree (RRT)

The final planner that we introduce is also able to take into account the motion model of the vehicle. Unlike the lattice planner which plans over a regular grid, the RRT uses probabilistic methods like the PRM planner.

The underlying insight is similar to that for the lattice planner and Fig. 5.18 shows a family of paths that the bicycle model of Eq. 4.2 would follow in configuration space. The paths are computed over a fixed time interval for discrete values of velocity, forward or backward, and various steering angles. This demonstrates clearly the subset of all possible configurations that a nonholonomic vehicle can reach from a given initial configuration.



The main steps in creating an RRT are as follows, with the notation shown in the figure to the right. A graph of robot configurations is maintained and each node is a configuration $\boldsymbol{q} \in \mathbb{R}^2 \times \mathbb{S}^1$ which is represented by a 3-vector $\boldsymbol{q} \sim (x, y, \theta)$. The first, or root, node in the graph is the goal configuration of the robot. A random configuration $\boldsymbol{q}_{\mathrm{rand}}$ is chosen, and the node with the closest configuration $\boldsymbol{q}_{\mathrm{near}}$ is found – this configuration is near in terms of a cost function that includes distance and orientation.▸ A control is computed that moves the robot from $\boldsymbol{q}_{\mathrm{near}}$ toward $\boldsymbol{q}_{\mathrm{rand}}$ over a fixed path simulation time. The configuration that it reaches is $\boldsymbol{q}_{\mathrm{new}}$ and this is added to the graph.

The distance measure must account for a difference in position and orientation and requires appropriate weighting of these quantities. From a consideration of units this is not quite proper since we are adding meters and radians.

For any desired starting configuration we can find the closest configuration in the graph, and working backward toward the starting configuration we could determine the sequence of steering angles and velocities needed to move from the start to the goal configuration. This has some similarities to the roadmap methods discussed previously, but the limiting factor is the combinatoric explosion in the number of possible poses.

We first of all create a model to describe the vehicle kinematics

```
>> car = Bicycle('steermax', 0.5);
```

and here we have specified a car-like vehicle with a maximum steering angle of 0.5 rad. Following our familiar programming pattern we create an RRT object

```
>> rrt = RRT(car, 'npoints', 1000)
```

for an obstacle free environment which by default extends from –5 to +5 in the $x$- and $y$-directions and create a plan

```
>> rrt.plan();
>> rrt.plot();
```

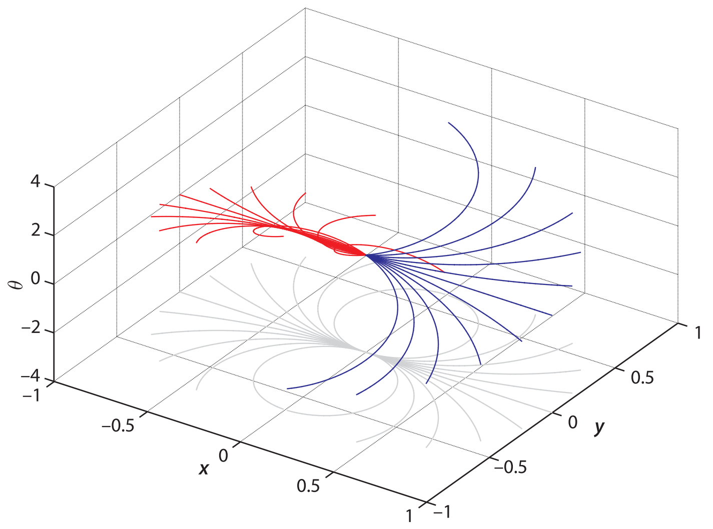


**Fig. 5.18.** A set of possible paths that the bicycle model robot could follow from an initial configuration of (0, 0, 0). For $v = \pm 1$, $\alpha \in [-1, 1]$ over a 2 s period. *Red lines* correspond to $v < 0$

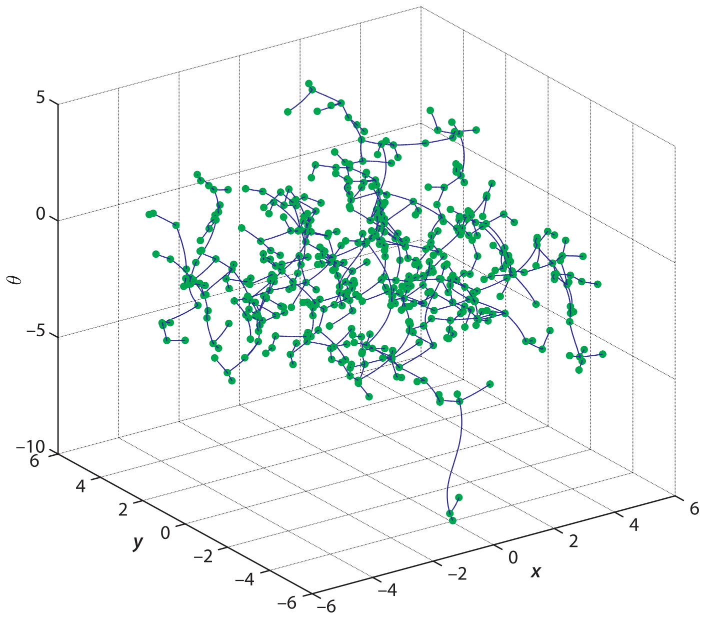


**Fig. 5.19.** An RRT computed for the bicycle model with a velocity of $\pm 1\ \text{m s}^{-1}$, steering angle limits of $\pm 0.5$ rad, integration period of 1 s, and initial configuration of (0, 0, 0). Each node is indicated by a *green circle* in the 3-dimensional space of vehicle poses $(x, y, \theta)$

The random tree is shown in Fig. 5.19 and we see that the paths have a good coverage of the configuration space, not just in the $x$- and $y$-directions but also in orientation, which is why the algorithm is known as *rapidly exploring*.

An important part of the RRT algorithm is computing the control input that moves the robot from an existing configuration in the graph to $\boldsymbol{q}_{\text{rand}}$. From Sect. 4.1 we understand the difficulty of driving a nonholonomic vehicle to a specified configuration. Rather than the complex nonlinear controller of Sect. 4.1.1.4 we will use something simpler that fits with the randomized sampling strategy used in this class of planner. The controller randomly chooses whether to drive forwards or backwards and randomly chooses a steering angle within the limits◂. It then simulates motion of the vehicle model for a fixed period of time, and computes the closest distance to $\boldsymbol{q}_{\text{rand}}$. This is repeated multiple times and the control input with the best performance is chosen. The configuration on its path that was closest to $\boldsymbol{q}_{\text{rand}}$ is chosen as $\boldsymbol{q}_{\text{near}}$ and becomes a new node in the graph.

Uniformly randomly distributed between the steering angle limits.

Handling obstacles with the RRT is quite straightforward. The configuration $\boldsymbol{q}_{\text{rand}}$ is discarded if it lies within an obstacle, and the point $\boldsymbol{q}_{\text{near}}$ will not be added to the graph if the path from $\boldsymbol{q}_{\text{near}}$ toward $\boldsymbol{q}_{\text{rand}}$ intersects an obstacle. The result is a set of paths, a roadmap, that is collision free and driveable by this nonholonomic vehicle.◂

We have chosen the first node to be the goal configuration, and we search from here toward possible start configurations. However we could also make the first node the start configuration. Alternatively we could choose the start node to be neither the start or goal position, the planner will find a path through the RRT between configurations close to the start and goal.

We will repeat the parallel parking example from the last section

```
>> rrt = RRT(car, road, 'npoints', 1000, 'root', [50 22 0], 'simtime', 4)
>> rrt.plan();
```

where we have specified the vehicle kinematic model, an occupancy grid, the number of sample points, the location of the first node, and that each random motion is simulated for 4 seconds. We can query the RRT plan for a path between two configurations

```
>> p = rrt.query([40 45 0], [50 22 0]);
```

and the result is a continuous path

```
>> about p
p [double] : 520x3 (12.5 kB)
```

which will take the vehicle from the street to the parking slot. We can overlay the path on the occupancy grid and RRT

```
>> rrt.plot(p)
```

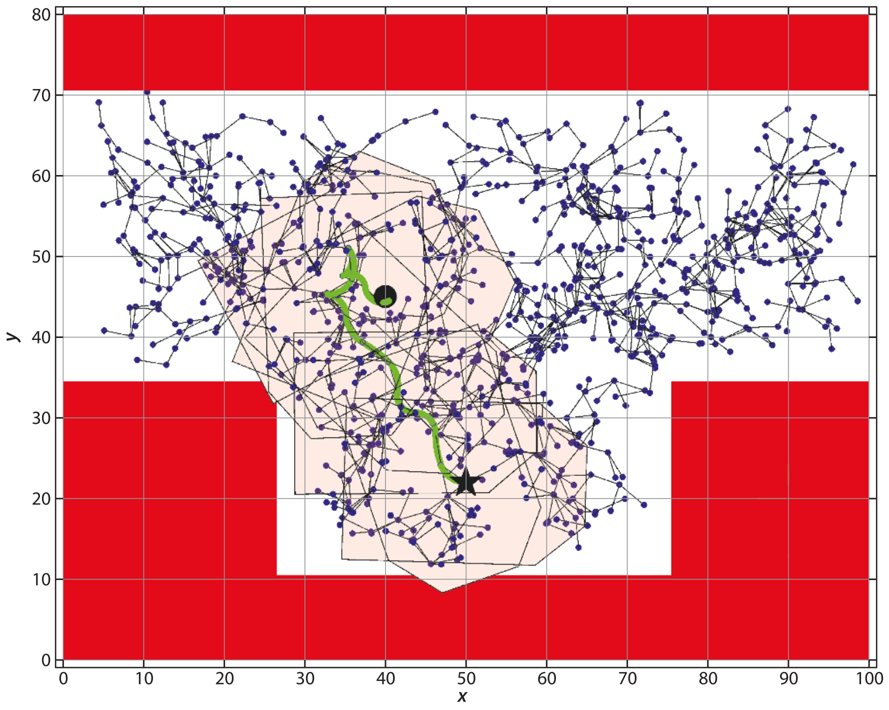


**Fig. 5.20.** A simple parallel parking example based on the RRT planner with an occupancy grid (cells are 10 cm square). RRT nodes are shown in *blue*, the initial configuration is a *solid circle* and the goal is a *solid star*. The path through the RRT is shown in *green*, and a few snapshots of the vehicle configuration are overlaid in *pink*

and the result is shown in Fig. 5.20 with some vehicle configurations overlaid. We can also animate the motion along the path

```
>> plot_vehicle(p, 'box', 'size', [20 30], 'fill', 'r', 'alpha', 0.1)
```

where we have specified the vehicle be displayed as a red translucent shape of width 20 and length 30 units.

This example illustrates some important points about the RRT. Firstly, as for the PRM planner, there may be some distance (and orientation) between the start and goal configuration and the nearest node. Minimizing this requires tuning RRT parameters such as the number of nodes and path simulation time. Secondly, the path is feasible but not quite optimal. In this case the vehicle has changed direction twice before driving into the parking slot. This is due to the random choice of nodes – rerunning the planner and/or increasing the number of nodes may help. Finally, we can see that the vehicle body collides with the obstacle, and this is very apparent if you view the animation. This is actually not surprising since the collision check we did when adding a node only tested if the node's position lay in an obstacle – it should properly check if a finite-sized vehicle with that configuration intersects an obstacle. Alternatively, as discussed on page 132 we could inflate the obstacles by the radius of the smallest disk that contains the robot.

## 5.3 Wrapping Up

Robot navigation is the problem of guiding a robot towards a goal and we have covered a spectrum of approaches. The simplest was the purely reactive Braitenberg-type vehicle. Then we added limited memory to create state machine based automata such as *bug2* which can deal with obstacles, however the paths that it finds are far from optimal.

A number of different map-based planning algorithms were then introduced. The distance transform is a computationally intense approach that finds an optimal path to the goal. D* also finds an optimal path, but supports a more nuanced travel cost – individual cells have a continuous traversability measure rather than being considered as only free space or obstacle. D* also supports computationally cheap incremental replanning for small changes in the map. PRM reduces the computational burden significantly by probabilistic sampling but at the expense of somewhat less optimal paths. In particular it may not discover narrow routes between areas of free space. The lattice planner takes into account the motion constraints of a real vehicle to create paths which are feasible to drive, and can readily account for the orientation of the vehicle as well as its position. RRT is another random sampling method that also generates kinematically feasible paths. All the map-based approaches require a map and knowledge of the robot's location, and these are both topics that we will cover in the next chapter.

## Further Reading

Comprehensive coverage of planning for robots is provided by two text books. Choset et al. (2005) covers geometric and probabilistic approaches to planning as well as the application to robots with dynamics and nonholonomic constraints. LaValle (2006) covers motion planning, planning under uncertainty, sensor-based planning, reinforcement learning, nonlinear systems, trajectory planning, nonholonomic planning, and is available online for free at http://planning.cs.uiuc.edu. In particular these books provide a much more sophisticated approach to representing obstacles in configuration space and cover potential-field planning methods which we have not discussed. The powerful planning techniques discussed in these books can be applied beyond robotics to very high order systems such as vehicles with trailers, robotic arms or even the shape of molecules. LaValle (2011a) and LaValle (2011b) provide a concise two-part tutorial introduction. More succinct coverage of planning is given by Kelly (2013), Siegwart et al. (2011), the Robotics Handbook (Siciliano and Khatib 2016, § 7), and also in Spong et al. (2006) and Siciliano et al. (2009).

The *bug1* and *bug2* algorithms were described by Lumelsky and Stepanov (1986). More recently eleven variations of Bug algorithm were implemented and compared for a number of different environments (Ng and Bräunl 2007). The distance transform is well described by Borgefors (1986) and its early application to robotic navigation was explored by Jarvis and Byrne (1988). Efficient approaches to implementing the distance transform include the two-pass method of Hirata (1996), fast marching methods or reframing it as a graph search problem which can be solved using Dijkstra's method; the last two approaches are compared by Alton and Mitchell (2006). The A* algorithm (Nilsson 1971) is an efficient method to find the shortest path through a graph, and we can always compute a graph that corresponds to an occupancy grid map. D* is an extension by Stentz (1994) which allows cheap replanning when the map changes and there have been many further extensions including, but not limited to, Field D* (Ferguson and Stentz 2006) and D* lite (Koenig and Likhachev 2002). D* is used in many real-world robot systems and many implementations exist including open source.

The ideas behind PRM started to emerge in the mid 1990s and it was first described by Kavraki et al. (1996). Geraerts and Overmars (2004) compare the efficacy of a number of subsequent variations that have been proposed to the basic PRM algorithm. Approaches to planning that incorporate the vehicle's dynamics include state-space sampling (Howard et al. 2008), and the RRT which is described in LaValle (1998, 2006) and related resources at http://msl.cs.uiuc.edu. More recently RRT* has been proposed by Karaman et al. (2011). Lattice planners are covered in Pivtoraiko, Knepper, and Kelly (2009).

**Historical and interesting.** The defining book in cybernetics was written by Wiener in 1948 and updated in 1965 (Wiener 1965). Grey Walter published a number of popular articles (1950, 1951) and a book (1953) based on his theories and experiments with robotic tortoises.

The definitive reference for Braitenberg vehicles is Braitenberg's own book (1986) which is a whimsical, almost poetic, set of thought experiments. Vehicles of increasing complexity (fourteen vehicle families in all) are developed, some including nonlinearities, memory and logic to which he attributes anthropomorphic characteristics such as love, fear, aggression and egotism. The second part of the book outlines the factual basis of these machines in the neural structure of animals.

Early behavior-based robots included the Johns Hopkins Beast, built in the 1960s, and Genghis (Brooks 1989) built in 1989. Behavior-based robotics are covered in the book by Arkin (1999) and the Robotics Handbook (Siciliano and Khatib 2016, § 13). Matarić's Robotics Primer (Matarić 2007) and associated comprehensive web-based resources is also an excellent introduction to reactive control, behavior based control and robot navigation. A rich collection of archival material about early cybernetic machines, including Grey-Walter's tortoise and the Johns Hopkins Beast can be found at the Cybernetic Zoo http://cyberneticzoo.com.

## Resources

A very powerful set of motion planners exist in OMPL, the Open MotionPLanning Library (http://ompl.kavrakilab.org) written in C++. It has a Python-based app that provides a convenient means to explore planning problems. Steve LaValle's web site http://msl.cs.illinois.edu/~lavalle/code.html has many code resources (C++ and Python) related to motion planning. Lattice planners are included in the sbpl package from the Search-Based Planning Lab (http://sbpl.net) which has MATLAB tools for generating motion primitives and C++ code for planning over the lattice graphs.

## MATLAB Notes

The Robotics System Toolbox™ from The MathWorks Inc. includes functions `BinaryOccupancyGrid` and `PRM` to create occupancy grids and plan paths using probabilistic roadmaps. Other functions support reading and writing ROS navigation and map messages. The Image Processing Toolbox™ function `bwdist` is an efficient implementation of the distance transform.

## Exercises

1. Braitenberg vehicles (page 127)
   a) Experiment with different starting configurations and control gains.
   b) Modify the signs on the steering signal to make the vehicle light-phobic.
   c) Modify the `sensorfield` function so that the peak moves with time.
   d) The vehicle approaches the maxima asymptotically. Add a stopping rule so that the vehicle stops when the when either sensor detects a value greater than 0.95.
   e) Create a scalar field with two peaks. Can you create a starting pose where the robot gets confused?
2. Occupancy grids. Create some different occupancy grids and test them on the different planners discussed.
   a) Create an occupancy grid that contains a maze and test it with various planners. See http://rosettacode.org/wiki/Maze_generation.
   b) Create an occupancy grid from a downloaded floor plan.
   c) Create an occupancy grid from a city street map, perhaps apply color segmentation (Chap. 13) to segment roads from other features. Can you convert this to a cost map for D* where different roads or intersections have different costs?
   d) Experiment with obstacle inflation.
   e) At 1 m cell size how much memory is required to represent the surface of the Earth? How much memory is required to represent just the land area of Earth? What cell size is needed in order for a map of your country to fit in 1 Gbyte of memory?

3. Bug algorithms (page 128)
   a) Using the function `makemap` create a new map to challenge *bug2*. Try different starting points.
   b) Create an obstacle map that contains a maze. Can *bug2* solve the maze?
   c) Experiment with different start and goal locations.
   d) Create a bug trap. Make a hollow box, and start the bug inside a box with the goal outside. What happens?
   e) Modify *bug2* to change the direction it turns when it hits an obstacle.
   f) Implement other bug algorithms such as *bug1* and *tangent bug*. Do they perform better or worse?
4. Distance transform (page 132)
   a) Create an obstacle map that contains a maze and solve it using distance transform.
5. D* planner (page 134)
   a) Add a low cost region to the living room. Can you make the robot prefer to take this route to the kitchen?
   b) Block additional doorways to challenge the robot.
   c) Implement D* as a mex-file to speed it up.
6. PRM planner (page 138)
   a) Run the PRM planner 100 times and gather statistics on the resulting path length.
   b) Vary the value of the distance threshold parameter and observe the effect.
   c) Use the output of the PRM planner as input to a pure pursuit planner as discussed in Chap. 4.
   d) Implement a nongrid based version of PRM. The robot is represented by an arbitrary polygon as are the obstacles. You will need functions to determine if a polygon intersects or is contained by another polygon (see the Toolbox `Polygon` class). Test the algorithm on the piano movers problem.
7. Lattice planner (page 140)
   a) How many iterations are required to completely fill the region of interest shown in Fig. 5.13c?
   b) How does the number of nodes and the spatial extent of the lattice increase with the number of iterations?
   c) Given a car with a wheelbase of 4.5 m and maximum steering angles of $\pm30$ deg what is the smallest possible grid size?
   d) Redo Fig. 5.15b to achieve a path that uses only right hand turns.
   e) Compute curvature as a function of path length for a path through the lattice such as the one shown in Fig. 5.15a.
   f) Design a controller in Simulink that will take a unicycle or bicycle model with a finite steering angle rate (there is a block parameter to specify this) that will drive the vehicle along the three paths shown in Fig. 5.13a.
8. RRT planner (page 144)
   a) Find a path to implement a 3-point turn.
   b) Experiment with RRT parameters such as the number of points, the vehicle steering angle limits, and the path integration time.
   c) Additional information in the node of each graph holds the control input that was computed to reach the node. Plot the steering angle and velocity sequence required to move from start to goal pose.
   d) Add a local planner to move from initial pose to the closest vertex, and from the final vertex to the goal pose.
   e) Determine a path through the graph that minimizes the number of reversals of direction.
   f) The collision test currently only checks that the center point of the robot does not lie in an occupied cell. Modify the collision test so that the robot is represented by a rectangular robot body and check that the entire body is obtacle free.