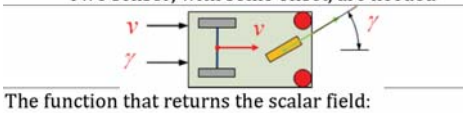
Reactive Navigation – Braitenberg Vehicles

Braitenberg vehicles are characterised by direct connection between sensors and motors. They have no explicit internal representation of the environment in which they operates and nor do they make explicit plans.

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

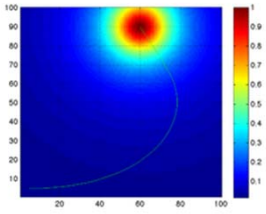
In this behaviour the sensor are connected directly to the motors:
no memory is needed, no map, just the capacity to sense a scalar field.

Two sensor, with some offset, are needed



The function that returns the scalar field:

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



Reactive Navigation – Braitenberg Vehicles

Understanding the control

As sensor value $\mathbf{s(x, y)} \in [0, 1]$ which is a function of the sensor's position in the plane. This particular function has a peak value at the point (60, 90). It makes sense that the vehicle speed has to be like:

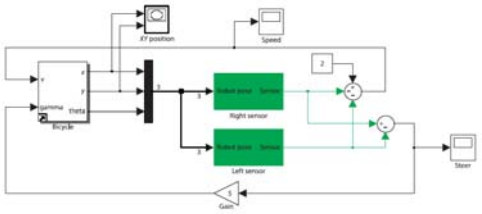
$v = 2 - S_R - S_L$

When the vehicle arrive to the target; $v = 2 - 1 - 1 = 0$

The vehicle steering:

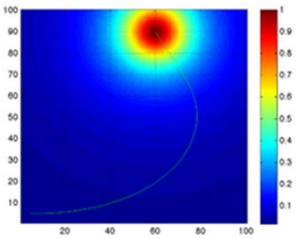
$\gamma = K(S_R - S_L)$

If both sensor has the same value the robot moves straight ahead



The '.m' code used to plot the results

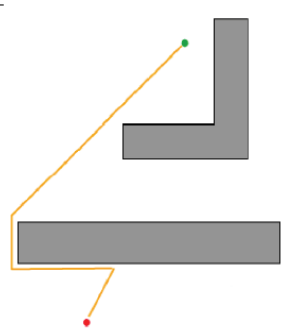
```
[x,y] = meshgrid([0:3:100]);
xc = 60; yc = 90 % coordinates of the peak
Intensity = 200./((x-xc).^2 + (y-yc).^2 + 200);
surf(x,y,Intensity)
view (0,90)
colorbar
hold on
sim('sl_braitenberg')
dimension=size (xygamma.signals.values,1)
plot3(xygamma.signals.values(:,1),xygamma.signals.
values(:,2),ones(dimension,1),...
    'LineWidth',2,...
    'Color',[0.8 0.8 0.8]
```



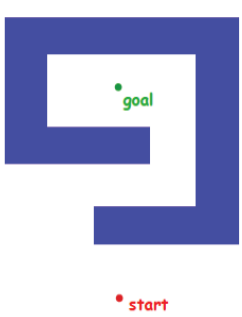
Reactive Navigation – Bug 0

Bug 0 Algorithm:

1. Head toward goal
2. Follow obstacle until you can head toward the goal again
3. Continue



Bug 0 Success!

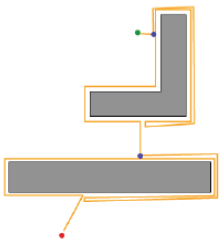


Bug 0 Fail!

Reactive Navigation – Bug 1

Bug 1 Algorithm:

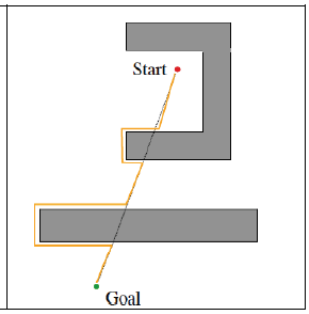
1. Head toward goal
2. If an obstacle is encountered, circumnavigate it and remember how close you get to the goal
3. Return to that closest point (wall-following) and continue



Reactive Navigation – Bug 2

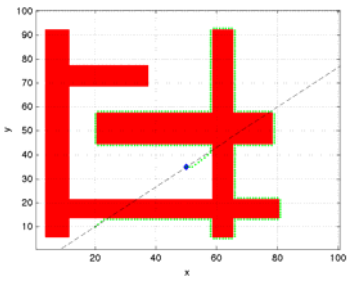
Bug 2 Algorithm:

1. Head toward goal on the m-line
2. If an obstacle is in the way, follow it until you encounter the m-line again closer to the goal.
3. Leave the obstacle and continue toward the goal



Play with the RTB implemented BUG2 Algorithm

```
>> load map1
>> bug = Bug2(map);
>> bug.goal = [50; 35];
>> bug.path([20; 10]);
```



Map-based Navigation

The key to achieving the *best* path between points A and B, is to use a map.

"*Best* path" means
- shortest distance
- shortest time
- penalty term or cost related to traversability
- feasibility due to kinematics & dynamics restrictions

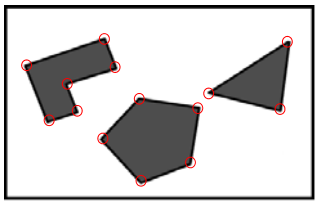
Requirements / Step
- Map building / upgrading
- Robot localization
- Path planning

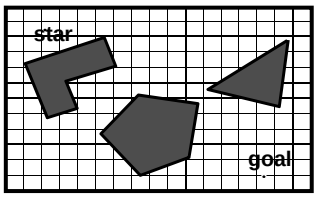
Map Building

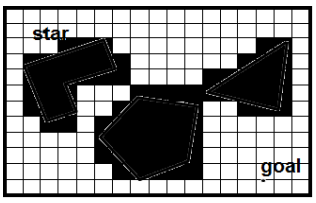
A map can be

- A list of driveable regions or obstacles, as polygons, each defined by their vertex or edges.
  Compact representation, Computational expensive.
- A occupancy grid. The world is treated 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 non-driveable cell.
  Memory expensive, Fast computation.

Map dimension and resolution are relevant criteria.







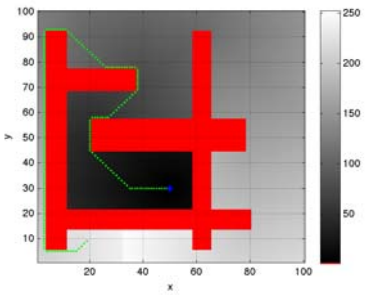
Distance Transform

Consider a matrix of zeros with just a single non-zero 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 non-zero pixel.

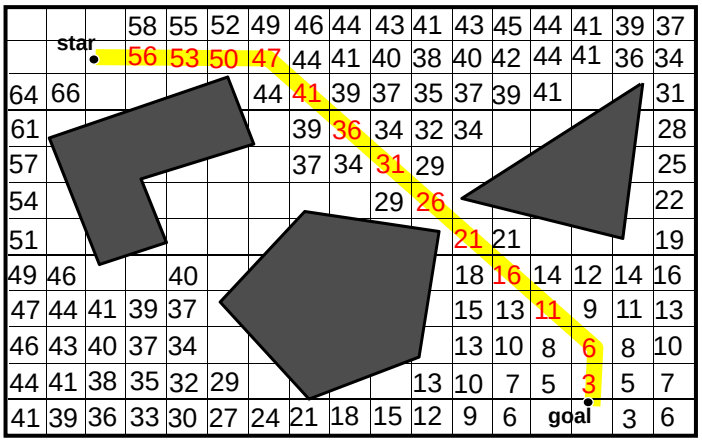
Red cells are obstacle. The background grey intensity represents the cell's Euclidean distance from the goal in units of cell size. Goal is represented by a zero value.

For robot path planning we use the default Euclidean distance.

```
>> goal = [50; 30]
>> start = [20; 10]
>> load map1
>> dx = DXform(map)
>> dx.plan(goal)
>> p = dx.path(start)
>> dx.plot(p)
```



Distance Transform. Exemple



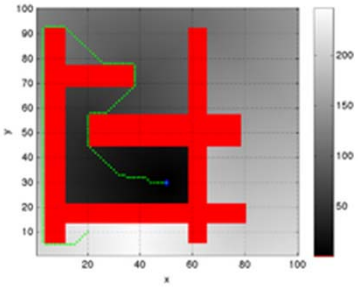
D* Path Planning

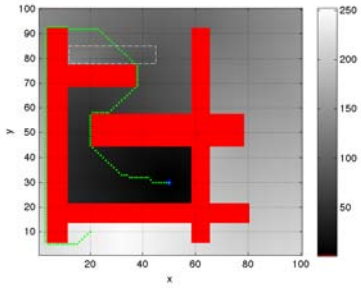
- D* generalizes the occupancy grid to a cost map.
- D* finds the path which minimizes the total cost of travel (from start to goal positions)
- A plan for moving to the goal is generated by creating a very dense directed graph. Every cell is a graph vertex and has a cost, a distance to the goal, and a link to the neighbouring cell that is closest to the goal.
- The key feature of D* is that it supports incremental replanning. This is important if, while we are moving, we discover that the world is different to our map.

Path Planning using RTB D*

```
% Initial path planning
    >> ds = Dstar(map);
    % Default cost map
    >> c = ds.costmap();
    >> ds.plan(goal);
    >> ds.path(start);
```

```
for y=78:85     % Modifiying cost map
  for x=12:45
    ds.modify_cost([x,y], 2);
  end
end
ds.plan()
ds.path(start)
```

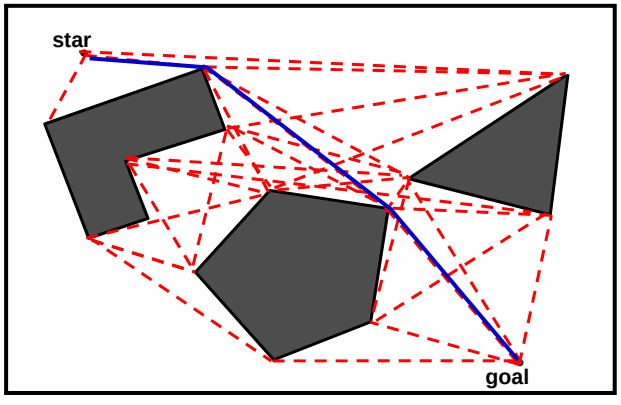




Visibility Graph

The visibility graph for a polygonal configuration space consists of edges joining all pairs of vertices that can see each other (including both the initial and goal positions as vertices as well). The unobstructed straight lines (roads) joining those vertices are obviously the shortest distances between them.

The task of the path planner is to find the shortest path from the initial position to the goal position along the roads defined by the visibility graph.



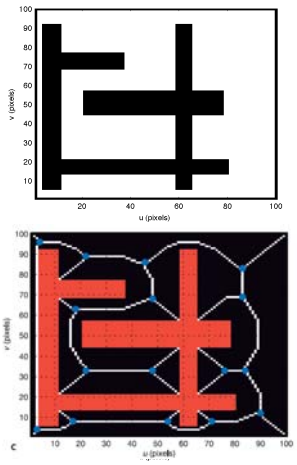
Voronoi Roadmap

The robot navigation problem deals with to build a network of obstacle free paths through the environment which serve the function of the “basic” road network (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.

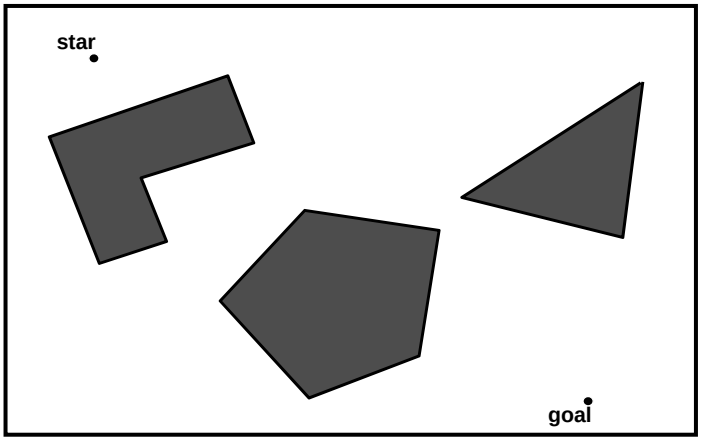
Computing the Voronoi Roadmap generation

1. Generate the occupancy matrix
   >> free = 1 - map;
   >> free(1,:) = 0; free(100,:) = 0;
   >> free(:,1) = 0; free(:,100) = 0;
2. Thinning free regions
   >> skeleton = ithin(free);

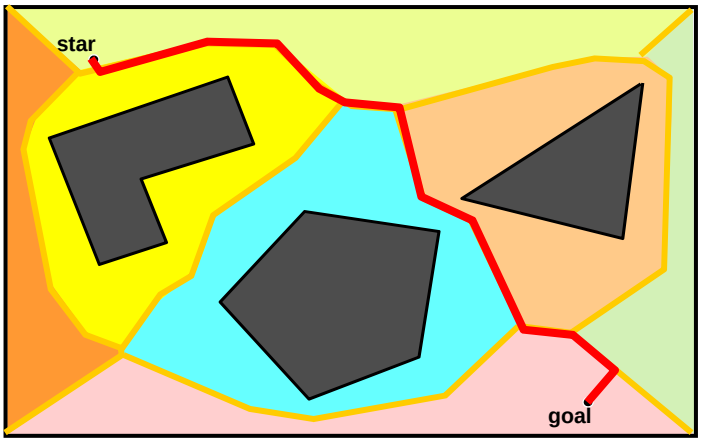
Obstacles have grown and the free space have become a thin network of connected white cells which are equidistant from the boundaries of the original obstacles.



Voronoi Roadmap. Exemple



Voronoi Roadmap. Exemple

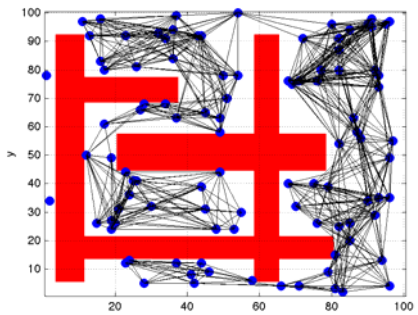


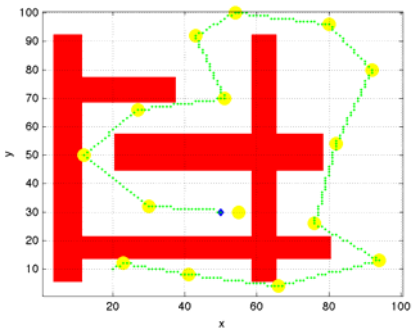
Probabilistic Roadmaps

To reduce computing time, a reduced set of points area taking into account to generate the roadmap. N *random points* that lie in free space are generated.

Each point is connected to its *nearest neighbours* 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.

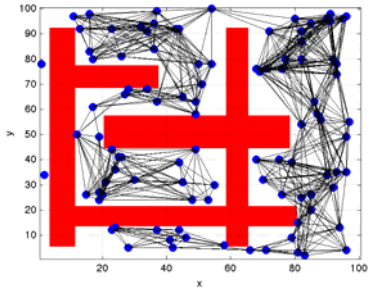


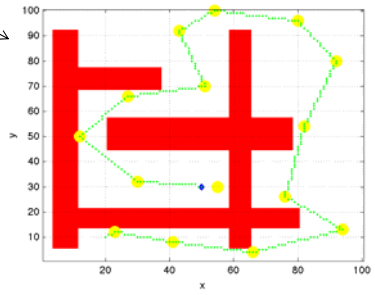


Probabilistic Roadmaps using RTB

```
>> goal = [50; 30];
>> start = [20; 10];
>> prm = PRM(map);
>> prm.plan();
>> prm.visualize()
>> p= prm.path(start, goal)

>> goal = [90; 90];   %  New path
>> start = [15; 50];
%  We can use the same roadmap
>> p = prm.path(start, goal);
```

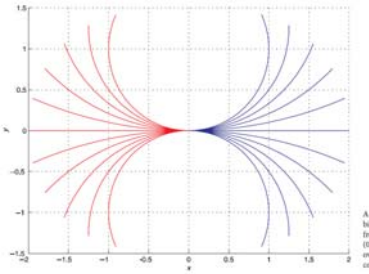


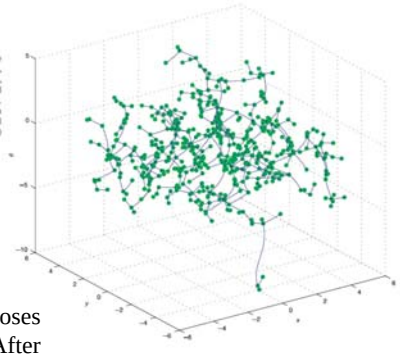


Rapidly-exploring Random Tree (RRT)

This planner is able to take into account the motion model of the vehicle, relaxing the assumption that the robot is capable of omni-directional motion.

Left figure shows a family of paths that the bicycle model would follow for discrete values of velocity, forward and backward, and steering wheel angle over a fixed time interval. It's the subset of all possible configurations that a non-holonomic vehicle can reach from a given initial configuration.





From each of these poses, we could compute another 22 poses that the vehicle could reach after two periods, and so on. After just a few periods we would have a very large number of possible poses.

Rapidly-exploring Random Tree (RRT)

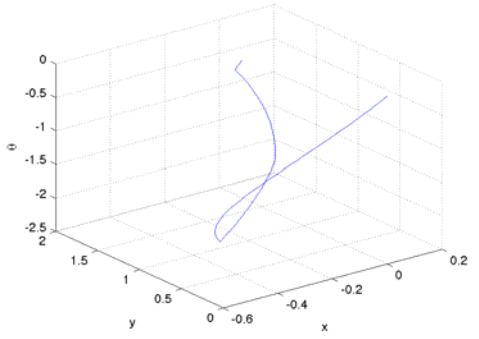
For any desired goal pose we could find the closest precomputed pose, and working backward toward the starting pose we could determine the sequence of steering angles and velocities needed to move from initial to the goal pose.

We create an RRT roadmap for an obstacle free environment using RTB. We create an RRT object

```
>> rrt = RRT()
```

which includes a default bicycle kinematic model, velocity and steering angle limits. We create a plan and visualize the results

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




```
>> p = rrt.path([0 0 0], [0 2 0]);
>> about(p)
    p [double] : 3x126 (3024 bytes)
>> plot2(p')
>> print('rrt_path2')
```


Cell Decomposition

The main idea of cell decomposition is to discriminate between geometric areas, or cells, that are free and areas that are occupied by objects.

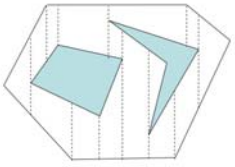
The basic cell decomposition path-planning algorithm can be summarized as follows:

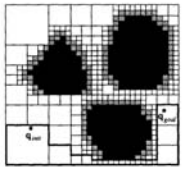
- Divide M into simple, connected regions called “cells”.
- Determine which opens cells are adjacent and construct a “connectivity graph”.
- Find the cells in which the initial and goal configurations lie and search for a path in the connectivity graph to join the initial and goal cell.
- From the sequence of cells found with an appropriate searching algorithm, compute a path within each cell, for example, passing through the midpoints of the cell boundaries or by a sequence of wall-following motions and movements along straight lines.

Two approaches are used;

Exact cell decomposition

Approximate cell decomposition



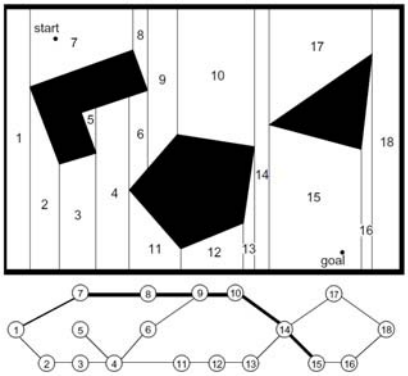


Exact Cell Decomposition

The boundary of cells is based on geometric criticality. The resulting cells are each either completely free or completely occupied. The decomposition is based on element's vertices.

The basic abstraction behind such a decomposition is that the particular position of the robot within each cell of free space does not matter; what matters is rather the robot's ability to traverse from each free cell to adjacent free cells.

The key disadvantage of exact cell decomposition is that the number of cells and, therefore, overall path planning computational efficiency depends upon the density and complexity of objects in the environment, just as with road mapbased systems. In environments that are extremely sparse, the number of cells will be small, even if the geometric size of the environment is very large.



Example of exact cell decomposition.

Approximate Cell Decomposition

It's a popular technique because grid-based representations are themselves fixed grid-size decompositions and so they are identical to an approximate cell decomposition of the environment.

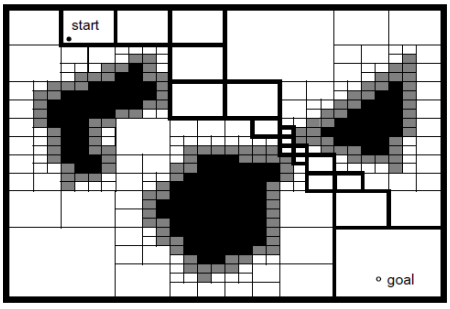
Distance transform can be see as a cell decomposition technique, using fixed grid-size.

A variable-size approximate cell decomposition method. The free space is externally bounded by a rectangle and, when needed, is recursively decomposed into smaller rectangles.

Each decomposition generates four identical new rectangles. At each level of resolution only the cells whose interiors lie entirely in the free space are used to construct the connectivity graph.

The resolution is reduced until either the path planner identifies a solution or a limit resolution is attained.

Less memory, More computing.



Conclusions

Robot Navigation == How to reach the goal position ?

- *Reactive Navigation*
- *Map-based Navigation*

Robot sensors are required

- *To compute current robot position (localization)*
- *To detect obstacles (obstacle avoidance)*
- *To guide the robot towards its goal (reactive approach)*
- *To upgrade map (map-based approach)*

Important tasks related with navigation are

- *Localization*
- *Environment perception*
- *Motion Control*