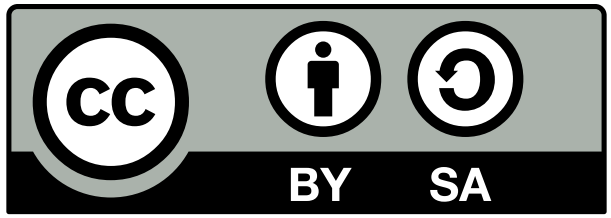
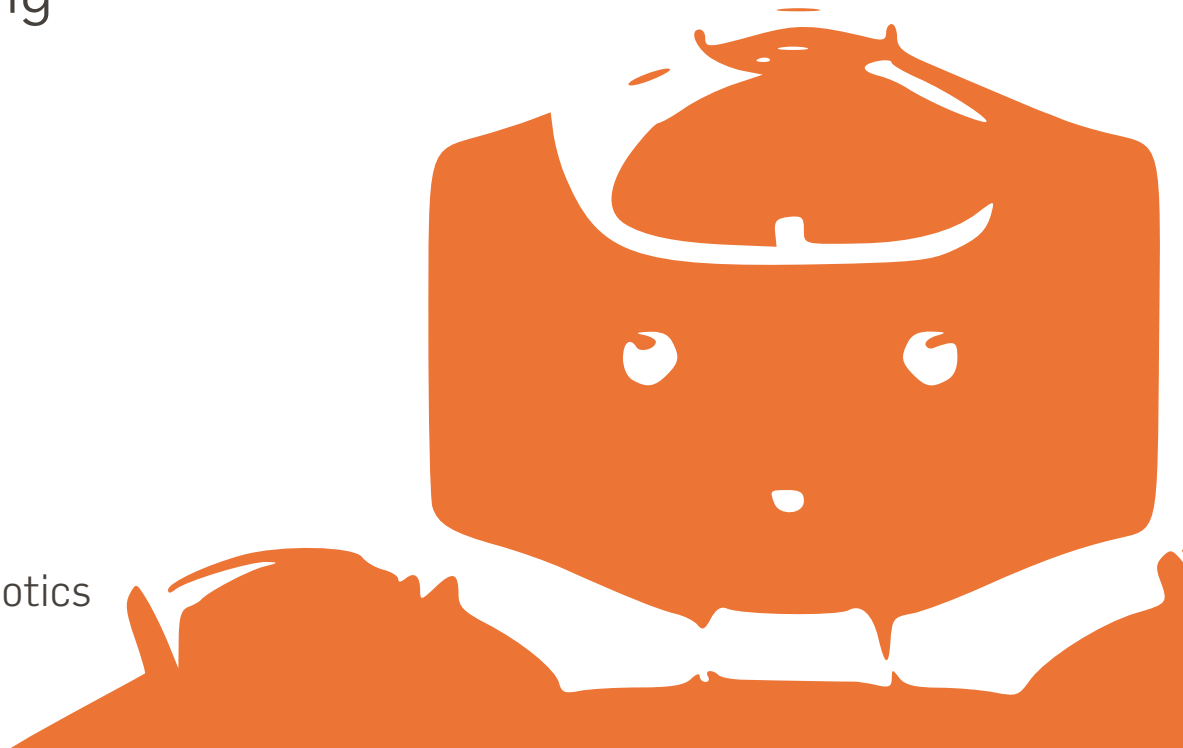


ROBOTICS WITH PLYMOUTH UNIVERSITY

ROCO318
Mobile and Humanoid Robots
Localisation and Planning



Séverin Lemaignan

Centre for Neural Systems and Robotics
Plymouth University

POSE AND NAVIGATION

Pose maintenance

- Pose (= position and orientation) maintenance is keeping an as precise as possible estimate of the robot's pose.

POSE AND NAVIGATION

Pose maintenance

- Pose (= position and orientation) maintenance is keeping an as precise as possible estimate of the robot's pose.

Planning: getting from A to B

- A cognitive skill: given partial knowledge of the environment, a goal and sensor readings, decide what to do to reach the goal as efficiently and reliably as possible.
- Different from inverse kinematics: IK ***locally*** maps a target state (position and orientation) in a global space (e.g. Cartesian space) to the robot's internal space (joint space, or generally ***configuration space***)
- $\rightarrow$ IK work on a small scale (< meter) while planning/navigation works on larger scale (room, building, roadmap, …)

PLANNING AND NAVIGATION – TRADITIONAL APPROACH

Today's industrial robots can operate **without any intelligence** because their environment is static and very structured.

In mobile robotics, cognition and reasoning is primarily of geometric nature, such as picking a safe path or determining where to go next.

$\rightarrow$ already been largely explored in literature for cases in which complete information about the current situation and the environment exists (e.g. travelling sales man problem or route inspection problem).

KIVA SYSTEMS: BEACON BASED LOCALISATION

http://www.youtube.com/watch?v=lWsMdN7HMuA

MONTE-CARLO LOCALISATION

MONTE CARLO METHODS

Repeated random sampling to compute results

Used when it to difficult to calculate exact results:

- the system is complex and calculating it through is too computationally expensive,
- or, too many variables and it is impossible to calculate all possibilities (*combinatorial explosion*)

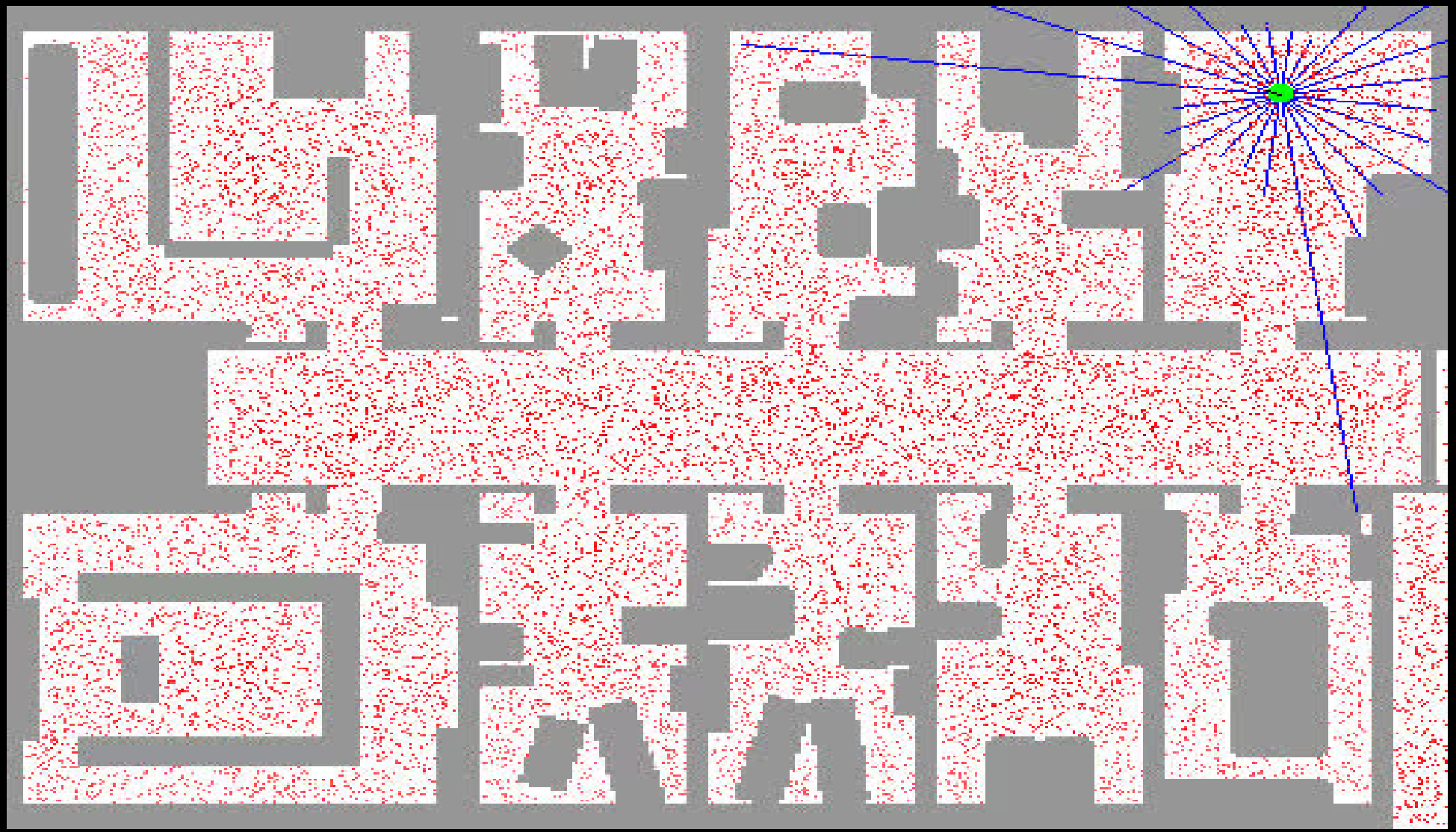
MONTE CARLO METHODS

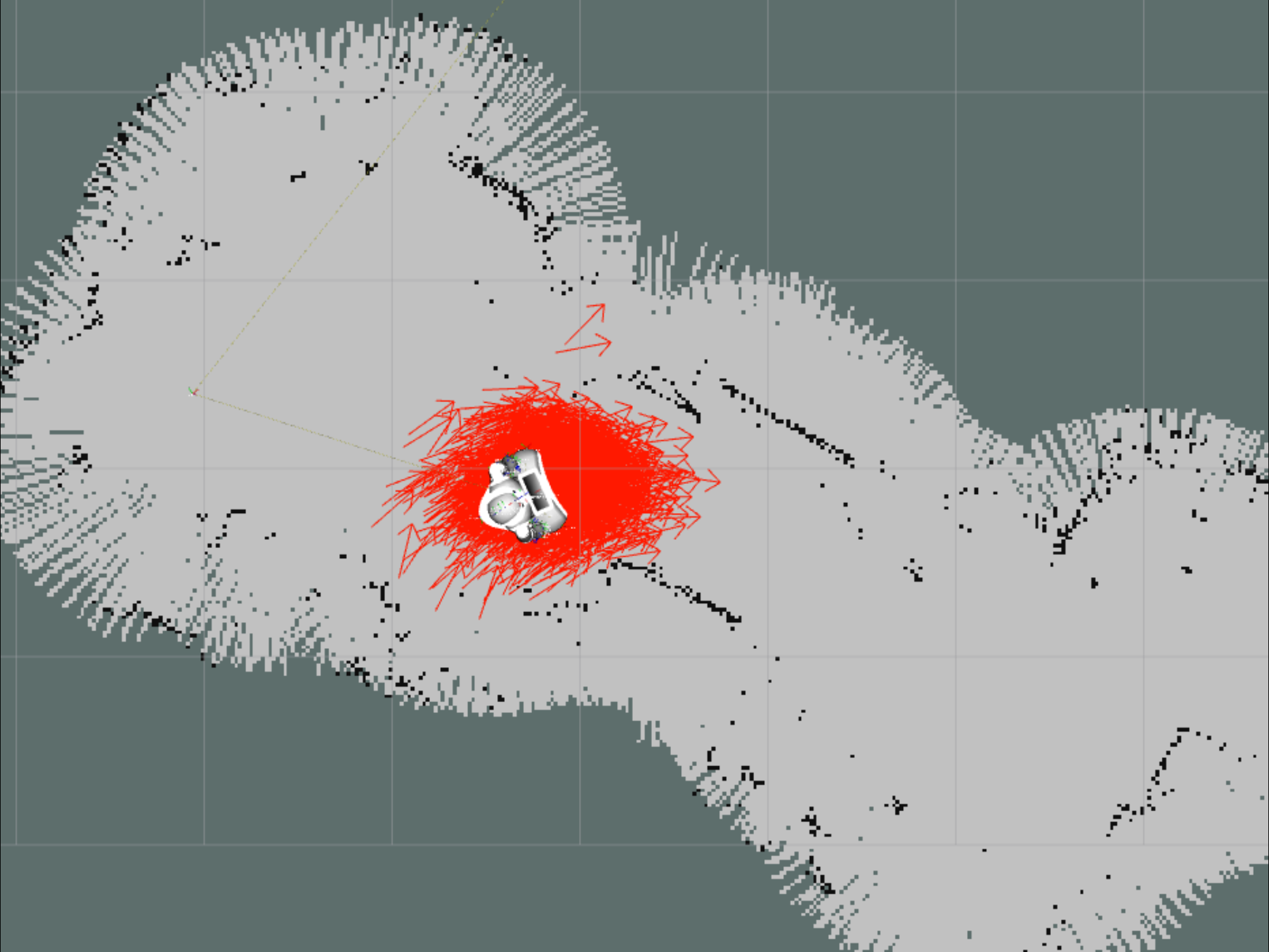
Repeated random sampling to compute results

Used when it to difficult to calculate exact results:

- the system is complex and calculating it through is too computationally expensive,
- or, too many variables and it is impossible to calculate all possibilities (***combinatorial explosion***)

Monte Carlo ***localisation*** is MC methods applied to robot localisation.





MONTE-CARLO LOCALISATION – INTRODUCTION

- Monte-Carlo Localisation (MCL) is a **particle filter**, which uses a set S_t of N particles at time t.

$$S_t = \{s_t^i \quad | \quad i = 1 \dots N\}$$

$$s^i = \{\mathbf{x}^i, w^i\} \text{ with } \mathbf{x}^i = \begin{bmatrix} x^i \\ y^i \\ \theta^i \end{bmatrix}$$

- Each particle s^i contains the pose of the particle $\mathbf{x}^i$ and a weight w^i.

MONTE-CARLO LOCALISATION – INTRODUCTION

- Monte-Carlo Localisation (MCL) is a **particle filter**, which uses a set S_t of N particles at time t.

$$S_t = \{s_t^i \quad | \quad i = 1 \dots N\}$$

$$s^i = \{\mathbf{x}^i, w^i\} \text{ with } \mathbf{x}^i = \begin{bmatrix} x^i \\ y^i \\ \theta^i \end{bmatrix}$$

- Each particle s^i contains the pose of the particle $\mathbf{x}^i$ and a weight w^i.
- The sum of all weights is 1:

$$\sum_{i=1}^{N} w^i = 1$$

MONTE-CARLO LOCALISATION – INTRODUCTION

- Monte-Carlo Localisation (MCL) is a **particle filter**, which uses a set S_t of N particles at time t.

$$S_t = \{s_t^i \quad | \quad i = 1 \dots N\}$$

$$s^i = \{\mathbf{x}^i, w^i\} \text{ with } \mathbf{x}^i = \begin{bmatrix} x^i \\ y^i \\ \theta^i \end{bmatrix}$$

- Each particle s^i contains the pose of the particle $\mathbf{x}^i$ and a weight w^i.
- The sum of all weights is 1:

$$\sum_{i=1}^{N} w^i = 1$$

- For small maps, N is in the hundreds, for larger maps you can have thousands of particles.

MONTE-CARLO LOCALISATION – INITIALISATION

The N particles are distributed on the map

- If the robot's position is not known, they are distributed randomly across the map with a probability $p = \frac{1}{N}$ (*uniform distribution*).

MONTE-CARLO LOCALISATION – INITIALISATION

The N particles are distributed on the map

- If the robot's position is not known, they are distributed randomly across the map with a probability $p = \frac{1}{N}$ (*uniform distribution*).
- If the robot's position is available, the initial particle distribution could be Gaussian around the robot (*normal distribution*).

MONTE-CARLO LOCALISATION – INITIALISATION

The N particles are distributed on the map

- If the robot's position is not known, they are distributed randomly across the map with a probability $p = \frac{1}{N}$ (*uniform distribution*).
- If the robot's position is available, the initial particle distribution could be Gaussian around the robot (*normal distribution*).

Then , Monte Carlo Localisation runs through three steps:

1. Prediction phase (when the robot takes an action)
2. Update phase (when sensor data comes in)
3. Resampling phase

MONTE-CARLO LOCLAISATION - PREDICTION PHASE

The prediction phase **essentially moves all particles** along with the robot.

Example: if the robot moves 10cm right and turn 10° counterclockwise, then all particles are updated:

$$s_t^i = \{\{x_{t-1}^i + 0.1, y_{t-1}^i, \theta_{t-1}^i - 10°\}, w_{t-1}^i\}$$

Note, the weight is not changed

MONTE-CARLO LOCLAISATION - PREDICTION PHASE

The prediction phase **essentially moves all particles** along with the robot.

Example: if the robot moves 10cm right and turn 10° counterclockwise, then all particles are updated:

$$s_t^i = \{\{x_{t-1}^i + 0.1, y_{t-1}^i, \theta_{t-1}^i - 10°\}, w_{t-1}^i\}$$

This allows for a **motion model** to be implemented, by adding some noise.

$$s_t^1 = \{\{x_{t-1}^1 + 0.11, y_{t-1}^1, \theta_{t-1}^1 - 10.3°\}, w_{t-1}^1\}$$
$$s_t^2 = \{\{x_{t-1}^2 + 0.09, y_{t-1}^2, \theta_{t-1}^2 - 10.2°\}, w_{t-1}^2\}$$
$$\ldots$$

MONTE-CARLO LOCLAISATION - PREDICTION PHASE

The prediction phase **essentially moves all particles** along with the robot.

Example: if the robot moves 10cm right and turn 10°

counterclockwise, then all particles are updated:

$$s_t^i = \{\{x_{t-1}^i + 0.1, y_{t-1}^i, \theta_{t-1}^i - 10°\}, w_{t-1}^i\}$$

This allows for a **motion model** to be implemented, by adding some noise.

This motion model should be based on the kinematics of your robot. e.g. for a differential drive robot you can have a good idea of how imprecise it is.

MONTE-CARLO LOCLAISATION - PREDICTION PHASE

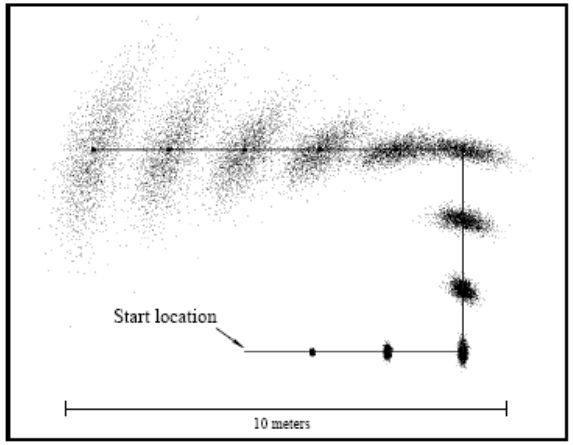
The prediction phase **essentially moves all particles** along with the robot.

Formally, if the robot performs an action a, all N particles are updated as follows:

$$\mathbf{x}_t^i = p(\mathbf{x}_t{}^i | \mathbf{x}_{t-1}{}^i, a)$$

Motion model: what is the probability of ending up in the location of particle s_t after doing action a at the location of particle s_{t-1}

Example: repeated application of the prediction rule leads to gradual loss of pose accuracy:



MONTE-CARLO LOCALISATION – UPDATE PHASE

- The robot gets **new sensor data** in, all N particles updated as follows:

$$w_t^i = \alpha \cdot p(z_t | \mathbf{x}_t^i) \cdot w_t^i$$

Normalisation constant: keeps the sum of w^i equal to 1

Sensor model: what is the probability of seeing sensor reading z_t at position $\mathbf{x}_t$

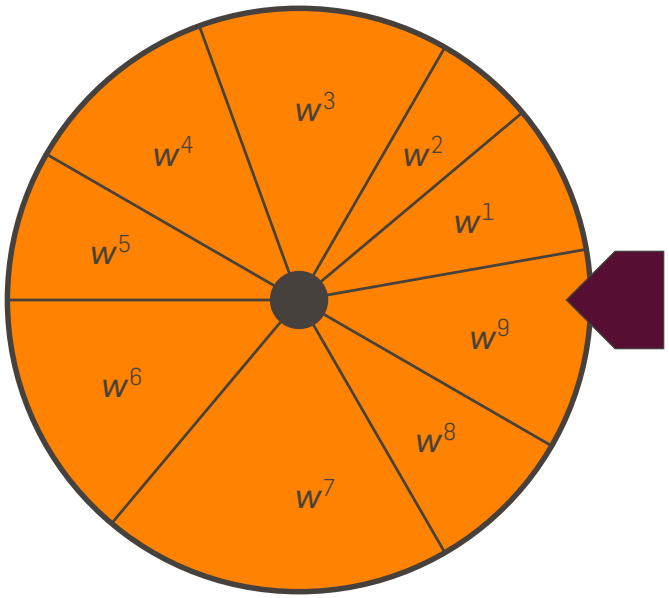
- The normalising constant α is calculated afterwards to make

$$\sum_{i=1}^{N} w^i = 1$$

MONTE-CARLO LOCALISATION – RESAMPLING PHASE

Generate a new set of particles by drawing randomly from the current set. Likelihood is determined by the weights values.

You can think of drawing particles as a roulette wheel selection. Particles with a higher weight have a higher chance of being selected into the new set. Also, a particle can be selected **more than once**.



MONTE-CARLO LOCALISATION – RESAMPLING PHASE

Generate a new set of particles by drawing randomly from the current set. Likelihood is determined by the weights values.

In addition, it is useful to add a **small number of uniformly distributed particles** to the set of particles.
This helps the algorithm recover when it loses track of the robot's position.

This is know as the *"kidnapped robot"* problem, and occurs when the robot loses track of its position, due to a malfunction or power outage.

PSEUDOCODE

```
# initialise all N particles
for i in range(1, N):
    S[i].x = random(x_max)
    S[i].y = random(y_max)
    S[i].θ = random(2 * π)
    S[i].w = 1/N
```

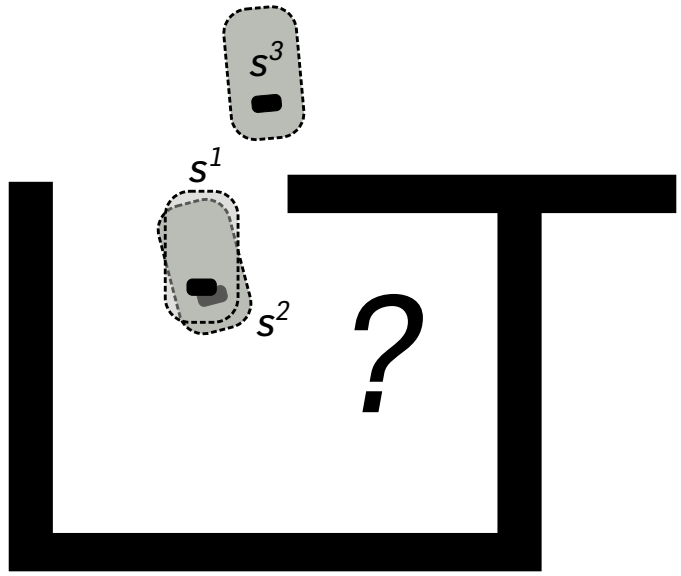
PSEUDOCODE

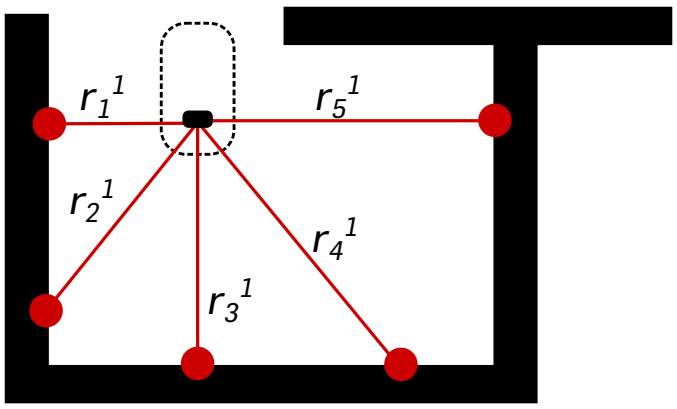
```
while True:
    # 1- prediction phase
    x, y, θ = estimate_robot_position()
    for i in range(1,N):
        move_particle(S[i], x, y, θ) # adds noise as well -> motion model

    # 2- update phase (z: sensor readings, m: map)
    n = 0
    for i in range(1,N):
        S[i].w = likelihood(z, (S[i].x, S[i].y, S[i].θ), m) * S[i].w
        n = n + S[i].w
    # normalise
    for i in range(1,N):
        S[i].w = S[i].w/n

    # 3- resample phase
    S1 = best_particles() # particles with high S.w
    S2 = worst_particles() # particles with low S.w
    S = S - S2 + S1 # effectively duplicates good particles
    # optionally, add a few new random particles
    for i in range(1, N): # reset the weights
        S[i].w = 1/N
```

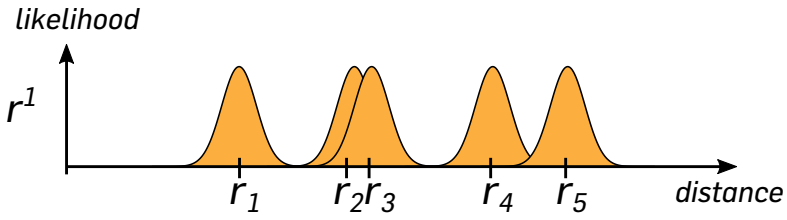
LIKELIHOOD FUNCTION

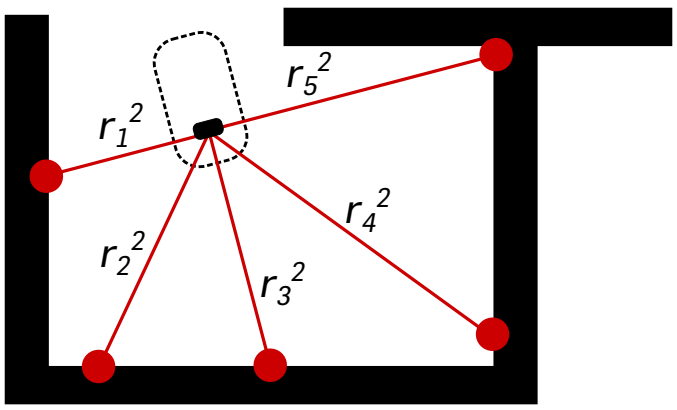
How is `likelihood(z, x, m)` implemented?

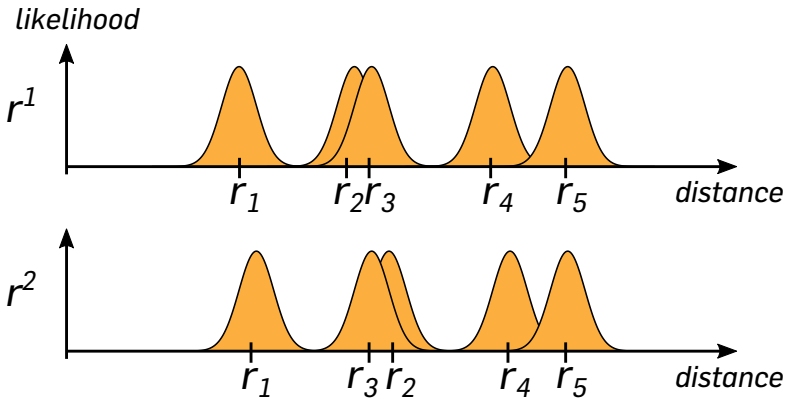
- `z`: the actual range measurements (sensor reading)
- `x`: the candidate position of the robot (particle location)
- `m`: the map

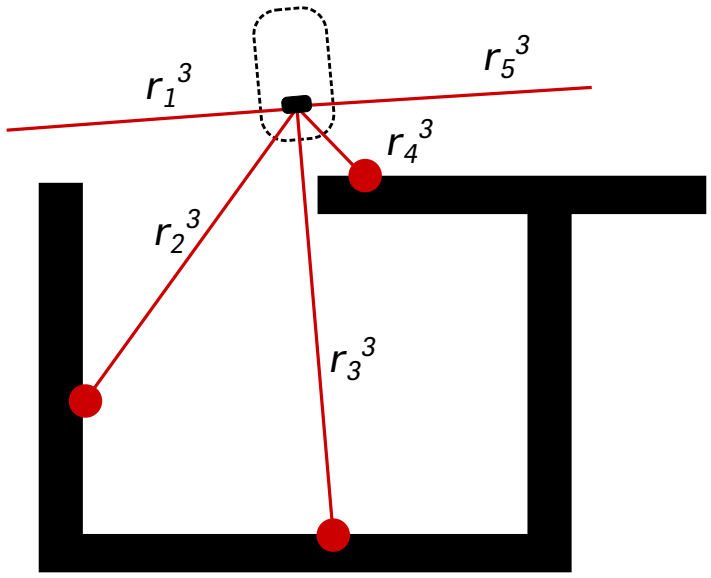

s^3
s^1
s^2
?

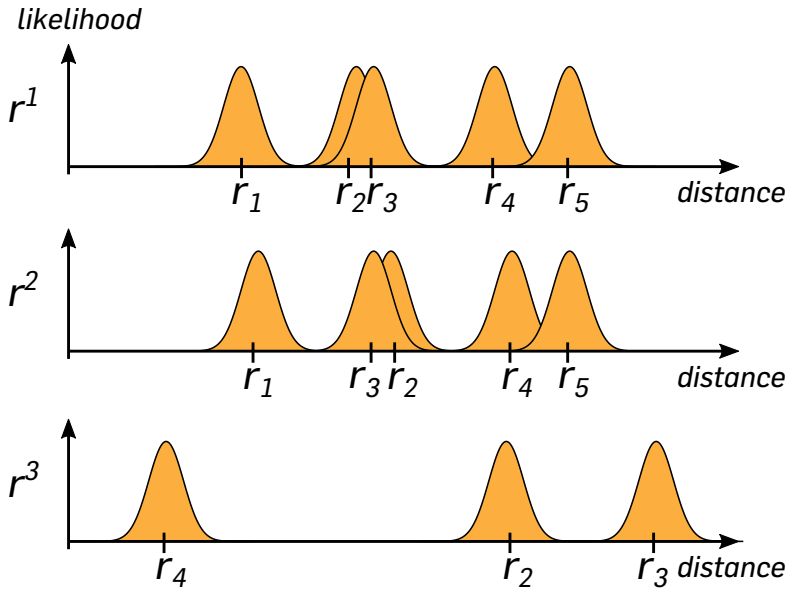

r_1^1
r_5^1
r_2^1
r_4^1
r_3^1

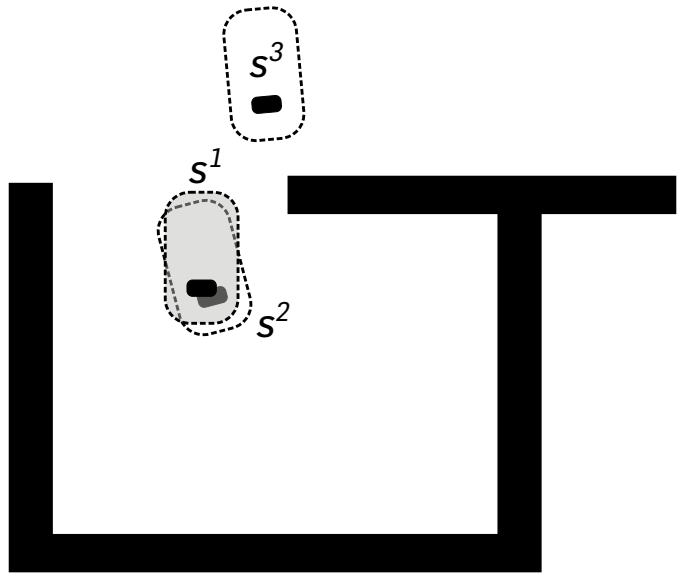

likelihood
r^1
r_1
r_2 r_3
r_4
r_5
distance

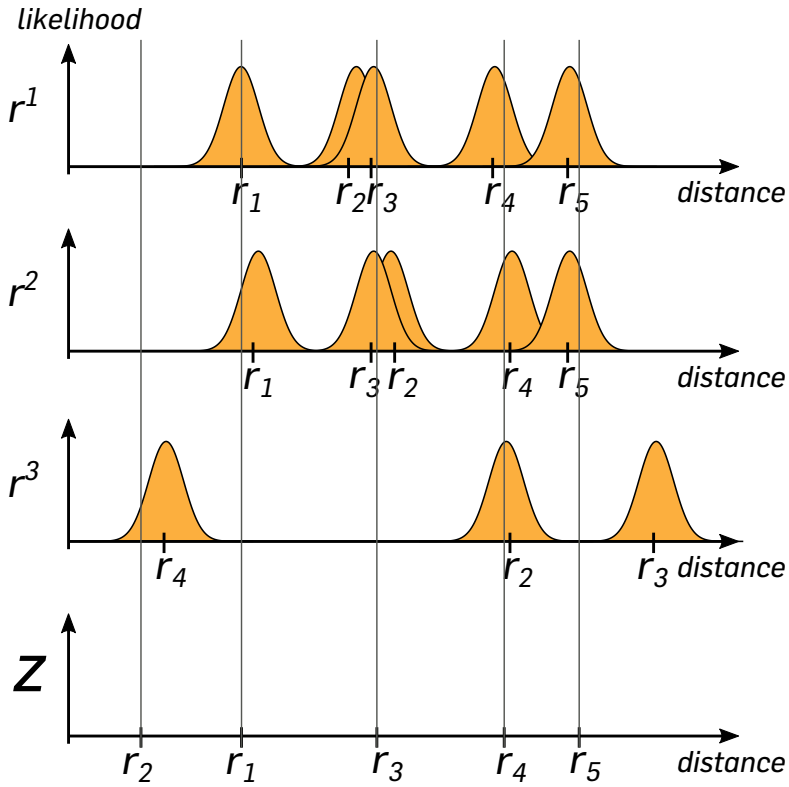

r_1^2
r_2^2
r_3^2
r_4^2
r_5^2

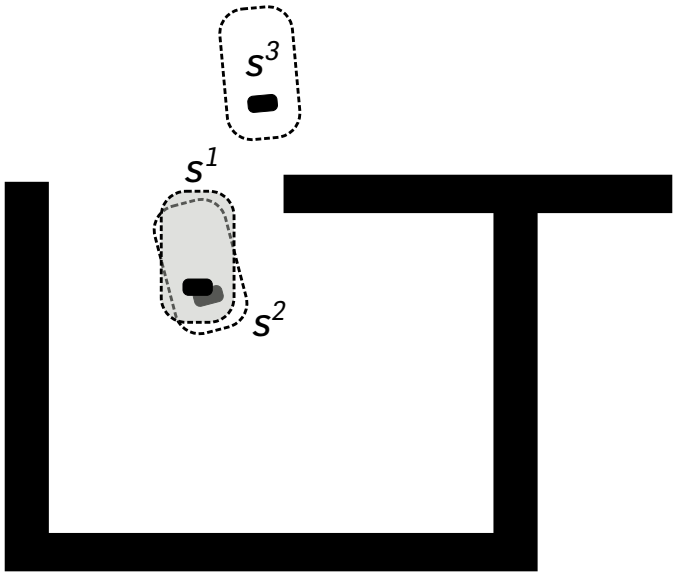

likelihood
r^1
r_1
r_2 r_3
r_4
r_5
distance
r^2
r_1
r_3 r_2
r_4
r_5
distance

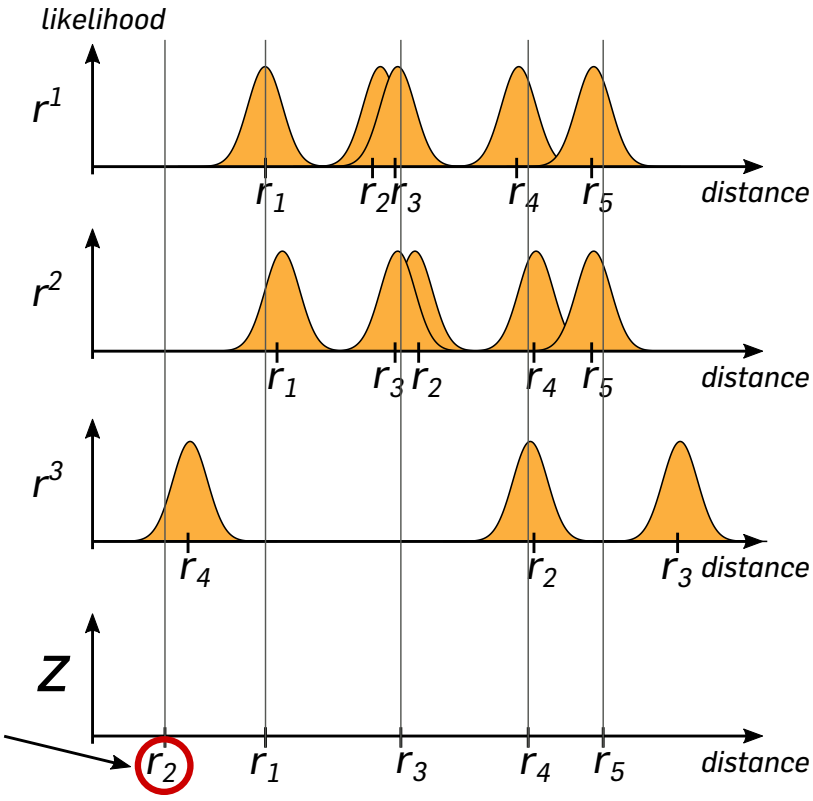

r_1^3
r_5^3
r_4^3
r_2^3
r_3^3


likelihood
r^1
r_1
r_2 r_3
r_4
r_5
distance
r^2
r_1
r_3 r_2
r_4
r_5
distance
r^3
r_4
r_2
r_3 distance


s^3
s^1
s^2


likelihood
r^1
r_1 r_2 r_3 r_4 r_5 distance
r^2
r_1 r_3 r_2 r_4 r_5 distance
r^3
r_4 r_2 r_3 distance
z
r_2 r_1 r_3 r_4 r_5 distance





short reading?

someone walking in front of the robot?

LIKELIHOOD FUNCTION

```
def likelihood(z, x, m):

    p = 1 # likelihood

    for obs_range in z:
        expected_range = raycast(x, bearing(z), m)

        pz = 0.0

        # good, but noisy, hit -> normal distribution
        err = obs_range - expected_range
        pz += z_hit * exp(-err^2 / (2 * sigma_hit^2))

        # + other types of measurement errors:
        #    - likelihood of a short reading from unexpected objects
        #    - likelihood of failures (black surface...)
        #    - likelihood of random measurements (reflections...)
        #
        # => pz += p_short + p_failure + p_rand

        p = p * pz

    return p
```

POSITION ESTIMATE

The position estimate at time t is the mean of the weighted particles locations, i.e. the centre of the weighted particle cloud.

$$\bar{\mathbf{x}}_t = \sum_{i=1}^{N} w_t^i \cdot \mathbf{x}_t^i$$

GLOBAL VS. CONTINUOUS LOCALISATION

The N particles are distributed on the map

- If the robot's position is not known, they are distributed randomly across the map with a probability $p = \frac{1}{N}$ (*uniform distribution*). **Global localisation**.
- If the robot's position is available, the initial particle distribution could be Gaussian around the robot (*normal distribution*). **Continuous localisation** $\equiv$ **Tracking**.

MONTE-CARLO LOCALISATION – SOME PROPERTIES

- Light on memory and computational resources.

  MCL draws inspiration from ***Markov Localisation***, but Markov Localisation uses much more memory as it needs to keep track of belief at each location of the map even the ones that have a low belief. This also results in a computationally expensive algorithm.

- "Any time" algorithm: you can interrupt the algorithm and still get an estimate out.
- Powerful, yet efficient.
- Easy to implement.

MONTE-CARLO LOCALISATION – SOME PROPERTIES

Compared to Kalman filtering:

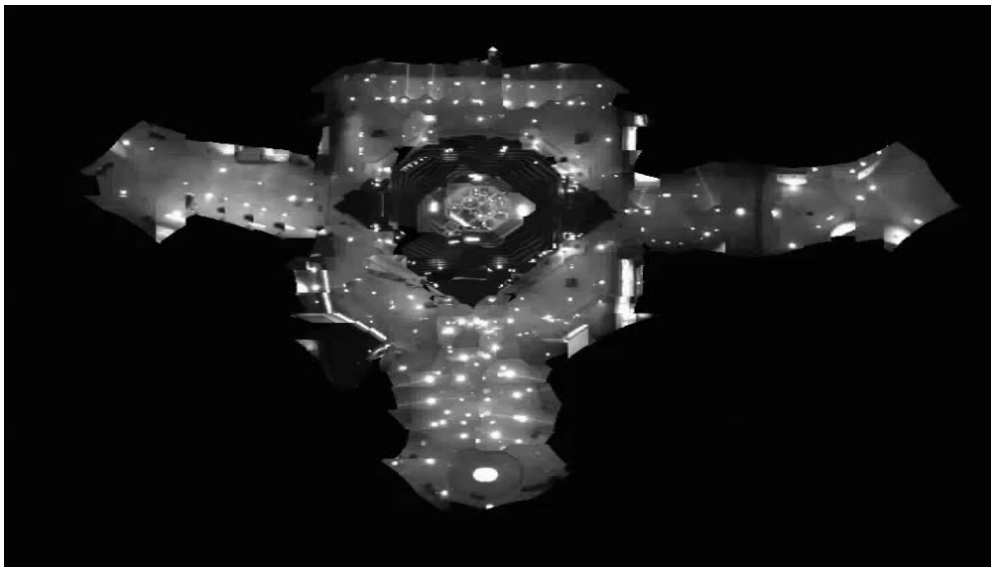
- Kalman filtering does not work for multi-modal distributions of the position estimate (***data association problem***)
- Kalman filtering assume linear sensors/motion models
- Kalman filtering is proved to be optimal

Example of bi-modal distribution arising from the symmetry of the environment

Source: Fox et al., Monte Carlo Localization: Efficient Position Estimation for Mobile Robots

EXAMPLE: MUSEUM POSE MAINTENANCE

Monte Carlo using the ceiling of the Smithsonian museum



MONTE-CARLO LOCALISATION – FURTHER READING

- Thrun, Burgard, Fox, *Probabilistic Robotics*
- Dellaert, Fox, Burgard, Thrun, Monte Carlo Localization for Mobile Robots, *IEEE International Conference on Robotics and Automation (ICRA99)*, May, 1999.

Videos of examples of MCL

- **http://www.youtube.com/watch?v=uU_1c_CxB1g**
- **http://www.youtube.com/watch?v=7K8dZwqBSSA**
- **https://www.youtube.com/watch?v=oKUYj1FWzN4**
- **https://www.youtube.com/watch?v=lCXv4yOcwf8**

SLAM

WHAT IS SIMULTANEOUS LOCALISATION AND MAPPING?

Given an unknown environment and vehicle pose:

- Move through the environment
- Estimate the robot pose
- Generate a map of environmental features

WHAT IS SIMULTANEOUS LOCALISATION AND MAPPING?

Given an unknown environment and vehicle pose:

- Move through the environment
- Estimate the robot pose
- Generate a map of environmental features

1. Use the robot pose estimate to improve the map landmark position estimates
2. Use the landmark estimates to improve the robot pose estimate
3. Repeat

PROBABILISTIC BASIS OF SLAM

Given:

- Robot control signal u_k (or measurement $\rightarrow$ odometry)
- A set of feature observations z_k (sensor measurements)

Estimate:

- Map of landmarks M_{k+1}
- Robot Pose V_{k+1}

Sources of Error:

- Control signal
- Motion model
- Sensor model

Source: Jack Collier

PROBABILISTIC BASIS OF SLAM

$$p(V_{k+1}, M_{k+1} | z_k, u_k)$$

$\Rightarrow$ estimate the joint probability of V_{k+1} and M_{k+1} conditioned on z_k and u_k.

Source: Jack Collier

PROBABILISTIC BASIS OF SLAM

$$p(V_{k+1}, M_{k+1}|z_k, u_k) =$$
$$\eta p(z_{k+1}|V_{k+1}, M_{k+1}) \int p(V_{k+1}|V_k, u_k) p(V_k, M_k|z_k, u_k) \mathrm{d}v_k$$

Source: Jack Collier

PROBABILISTIC BASIS OF SLAM

$$p(V_{k+1}, M_{k+1}|z_k, u_k) =$$
$$\eta p(z_{k+1}|V_{k+1}, M_{k+1}) \int p(V_{k+1}|V_k, u_k)\, p(V_k, M_k|z_k, u_k)\, \mathrm{d}v_k$$

Prior state and covariance estimates from the last filter iteration

Source: Jack Collier

PROBABILISTIC BASIS OF SLAM

$$p(V_{k+1}, M_{k+1}|z_k, u_k) =$$

$$\eta p(z_{k+1}|V_{k+1}, M_{k+1}) \int p(V_{k+1}|V_k, u_k)\, p(V_k, M_k|z_k, u_k) \mathrm{d}v_k$$

Probabilistic motion model estimates the new vehicle pose covariance estimates from the prior estimate and the control

Source: Jack Collier

PROBABILISTIC BASIS OF SLAM

$$p(V_{k+1}, M_{k+1}|z_k, u_k) =$$

$$\eta\, p(z_{k+1}|V_{k+1}, M_{k+1}) \int p(V_{k+1}|V_k, u_k) p(V_k, M_k|z_k, u_k) \mathrm{d}v_k$$

Measurement model gives the expected value of the feature observations

Source: Jack Collier

SLAM FAMILIES

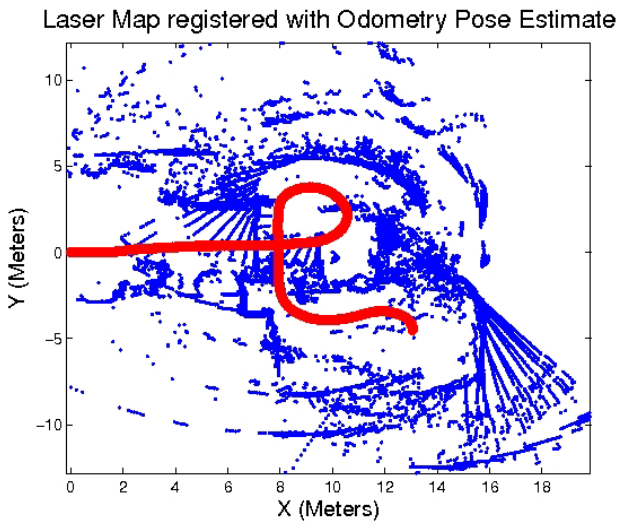
- *Kalman filtering*: **Extended KF (EKF) SLAM** + a few others
- *particle filtering*: Rao-Blackwellized particle filter (**FastSLAM**)

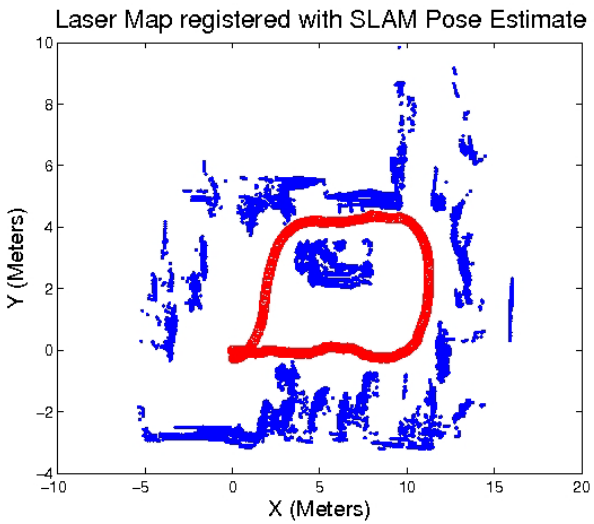
WHY IS SLAM HARD?

Chicken and egg problem

If there is no map, how the robot localise itself. If the robot doesn't know its location, how can it build up a map?

Odometry is not to be trusted:



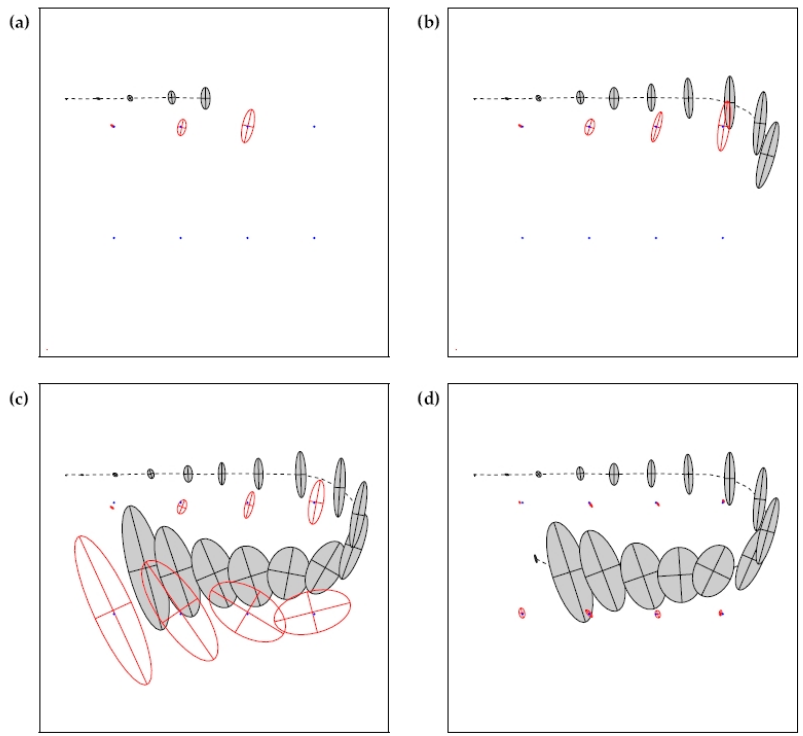


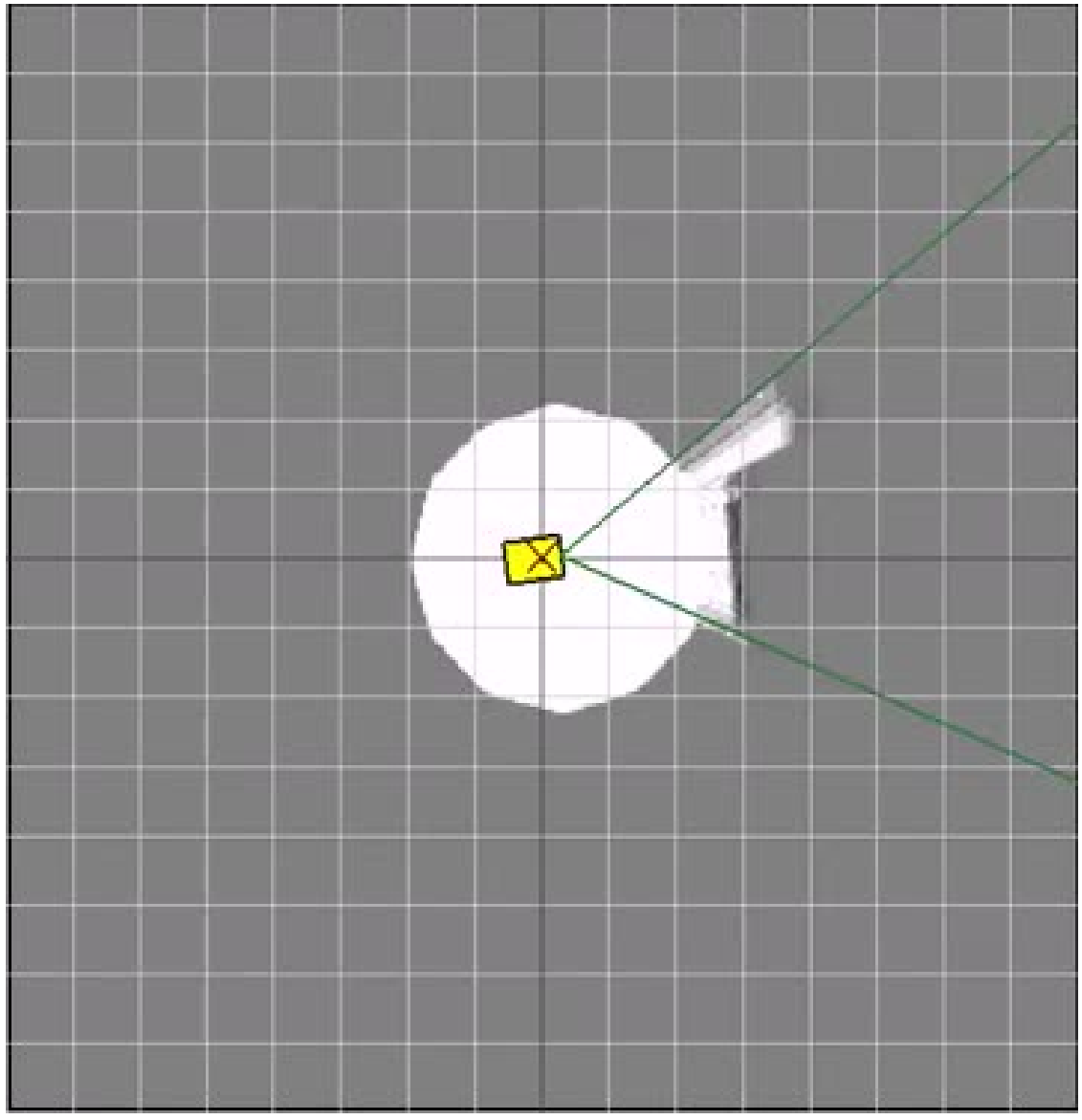
Source: Jack Collier

WHY IS SLAM HARD?

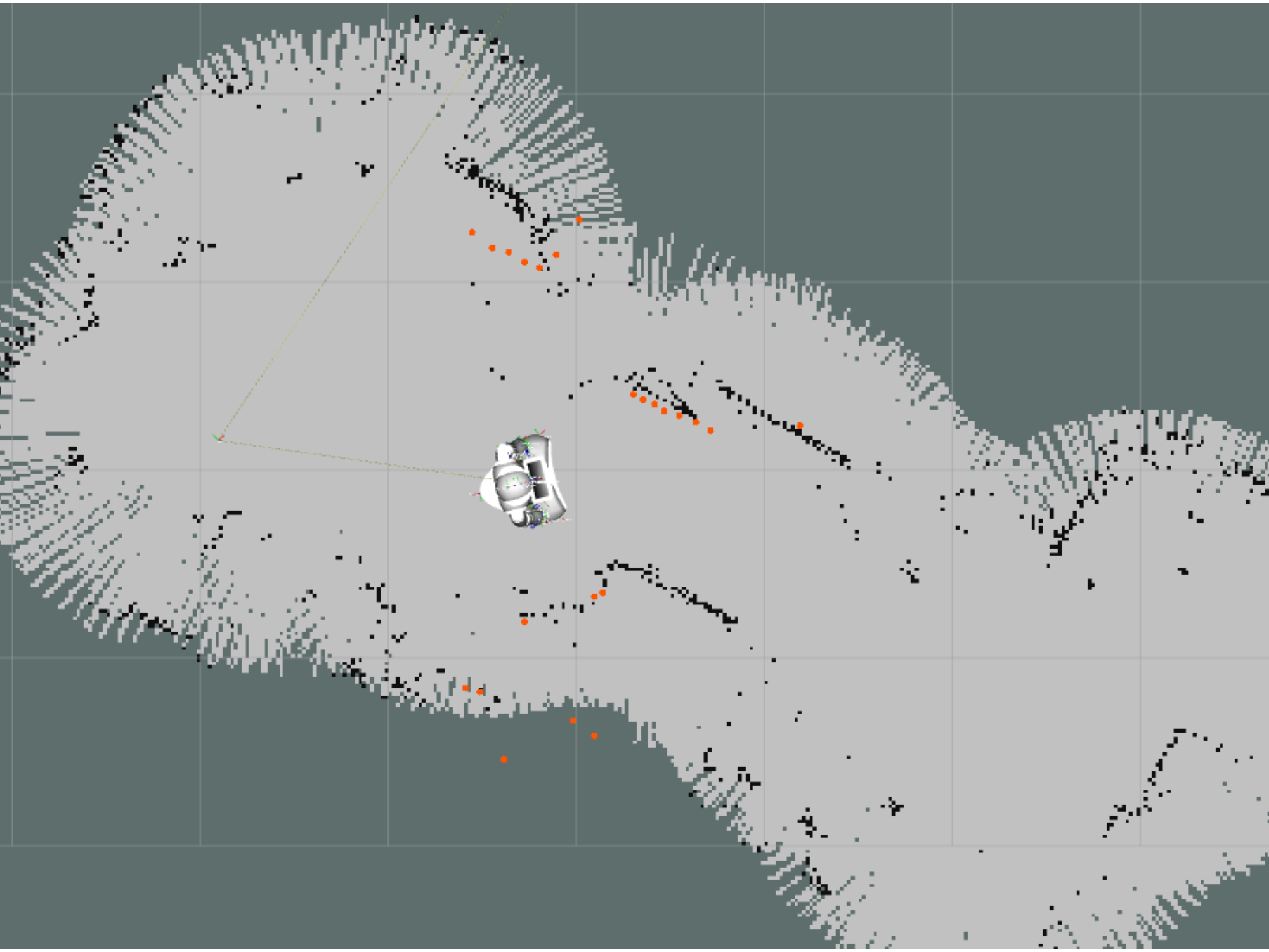
Loop closure

- Even with the best SLAM algorithm, pose uncertainty will increase as the vehicle moves
- This pose uncertainty means that landmark locations further from the map origin have a higher uncertainty
- **Revisiting a previously observed landmarks significantly reduces uncertainty in robot and landmark pose estimates**





Source: Robert Sim, University of British Columbia



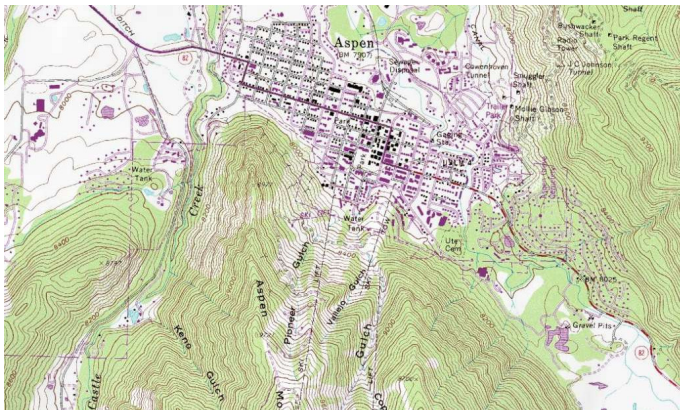

2x

SLAM – FURTHER READING

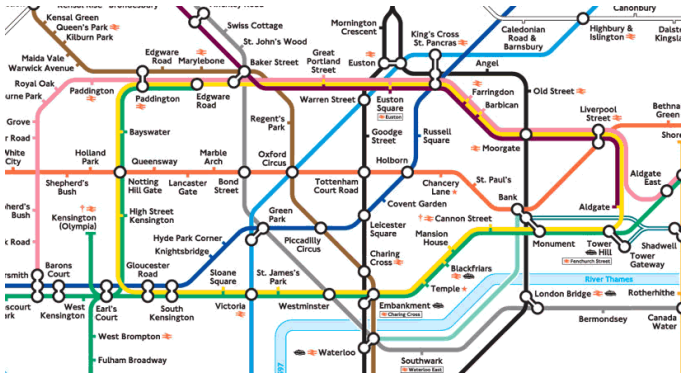
- Take a MSc Robotics next year!
- Jack Collier introduction (!!) to SLAM
- Thrun, Burgard, Fox, *Probabilistic Robotics*

PATH PLANNING

METRIC VS TOPOLOGICAL MAPS

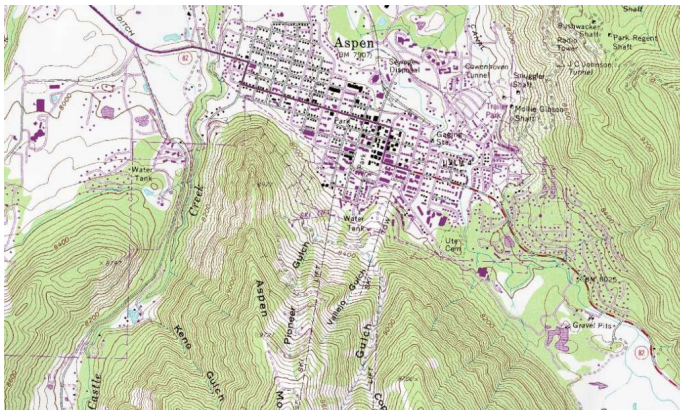


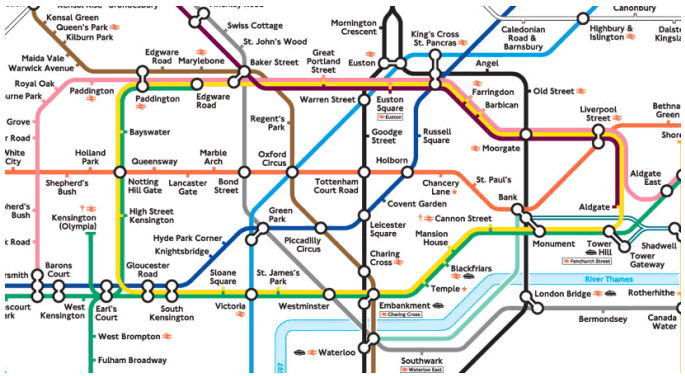
A metric map considers a two-dimensional space in which it places objects. The objects are placed with precise coordinates. This representation is convenient and natural, but ***sensitive to noise*** and ***difficult to calculate the distances precisely***.



A topological map only considers places and relations between them. Often, the distances between places are stored.
The map is typically a **graph**, in which the nodes corresponds to places and edges correspond to the paths.

METRIC VS TOPOLOGICAL MAPS

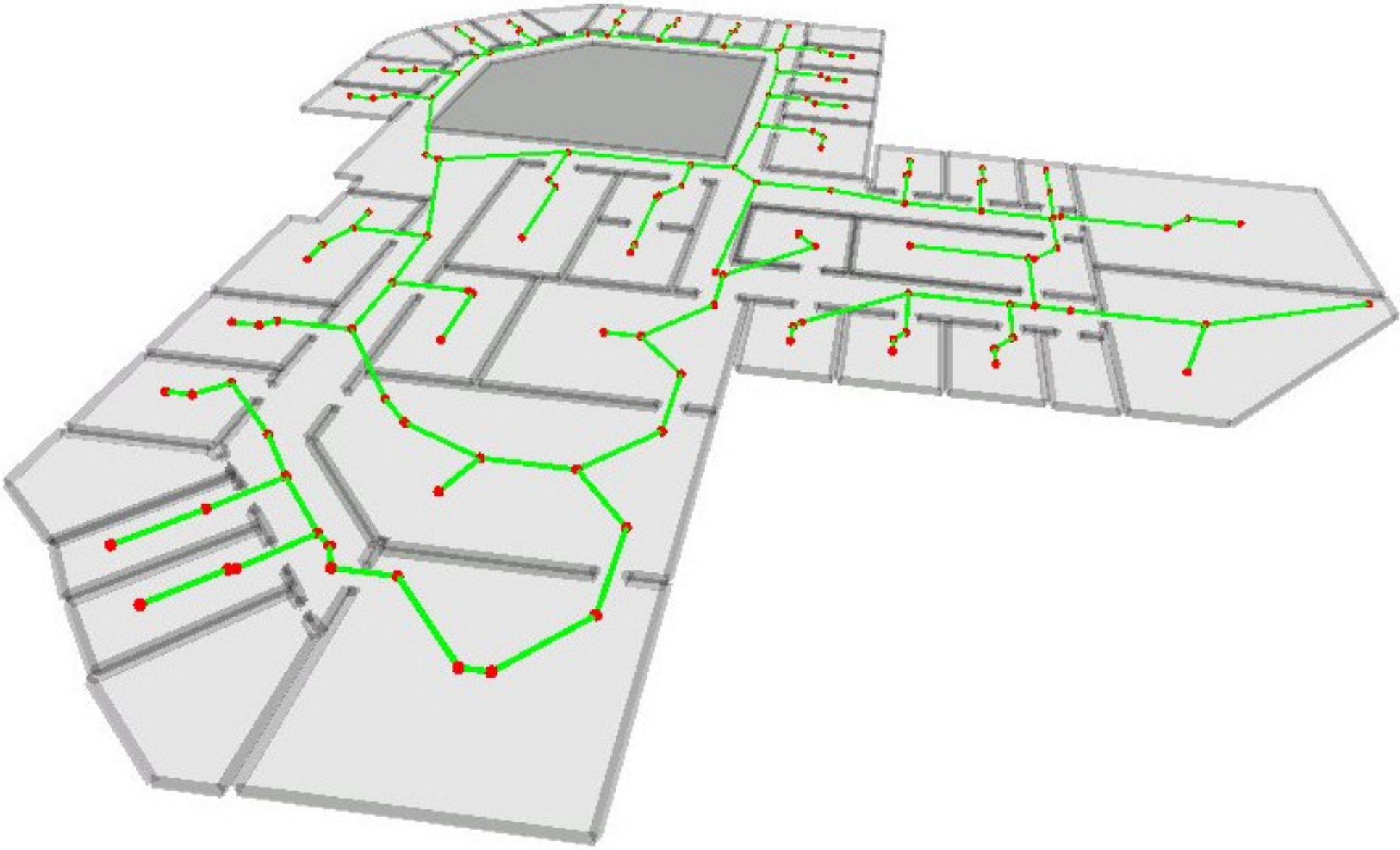




A metric map considers a two-dimensional space in which it places objects. The objects are placed with precise coordinates. This representation is convenient and natural, but *sensitive to noise* and *difficult to calculate the distances precisely*.

A topological map only considers places and relations between them. Often, the distances between places are stored.
The map is typically a **graph**, in which the nodes corresponds to places and edges correspond to the paths.

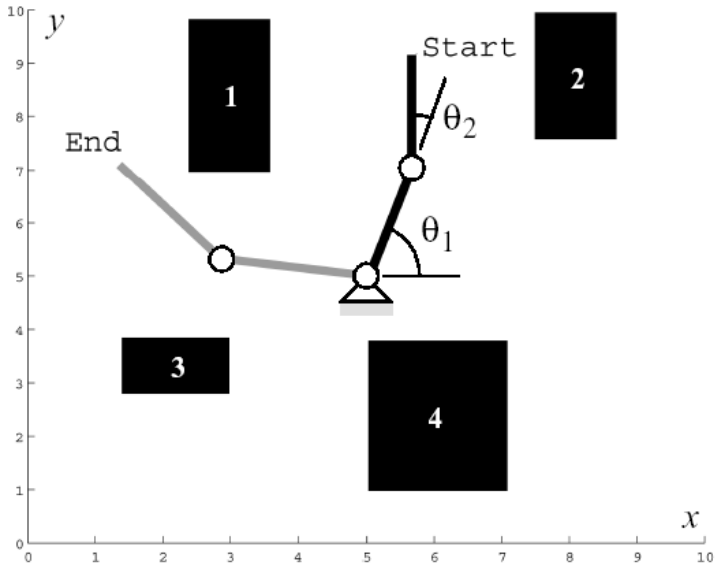
Cell decomposition can be used to transform a metric map into a topological one. More on that in a moment.




Queens Park
Kilburn High Road
Kilburn Park
St John's Wood
Regents Park
Maida Vale
Mornington Crescent
King's Cross
St Pancras
Euston
Angel
SHOREDITCH
Hoxton
BETHNAL GREEN
King's Cross Thameslink
FINSBURY
King's Cross St Pancras
Old Street
Great Portland Street
Euston Square
Warren Street
Baker Street
Marylebone
PADDINGTON
Warwick Avenue
Regent's Park
Russell Square
Bishopsgate
Shoreditch
Barbican
Liverpool Street
Edgware Road
Goodge Street
Farringdon
Royal Oak
Edgware Road
Chancery Lane
Moorgate
Aldgate East
Holborn
MARYLEBONE
Tottenham Court Road
City Thameslink
CITY OF LONDON
Aldgate
Paddington
Oxford Circus
Bank
St Paul's
Fenchurch Street
Tower Gateway
Marble Arch
Covent Garden
Temple
Cannon Street
Bayswater
Bond Street
Blackfriars
Mansion House
Tower Hill
Lancaster Gate
Leicester Square
Monument
Notting Hill Gate
Queensway
Piccadilly Circus
Charing Cross
Embankment
CITY OF WESTMINSTER
Green Park
Waterloo East
London Bridge
Holland Park
Hyde Park
Hyde Park Corner
Westminster
Waterloo
Southwark
Borough
Knightsbridge
Green Park
St James's Park
Lambeth North
Bermondsey
KENSINGTON
High Street Kensington
St James's Park
WESTMINSTER
Victoria
Elephant & Castle
BERMONDSEY
Kensington Olympia
Gloucester Road
South Kensington
Sloane Square
Earl's Court
LAMBETH
Kennington
KENSINGTON AND CHELSEA
Pimlico
West Kensington

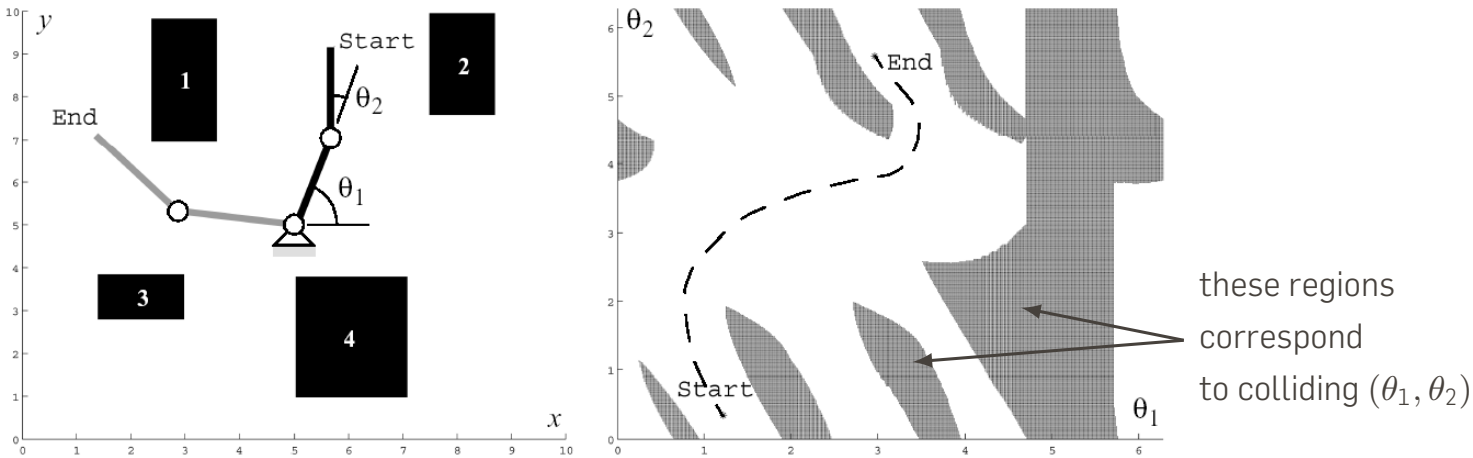
PATH PLANNING: CONFIGURATION SPACE

The state or configuration q of a robot can be described with k values q_i. For example, for a two-link planar robot, θ_1 and θ_2:



PATH PLANNING: CONFIGURATION SPACE

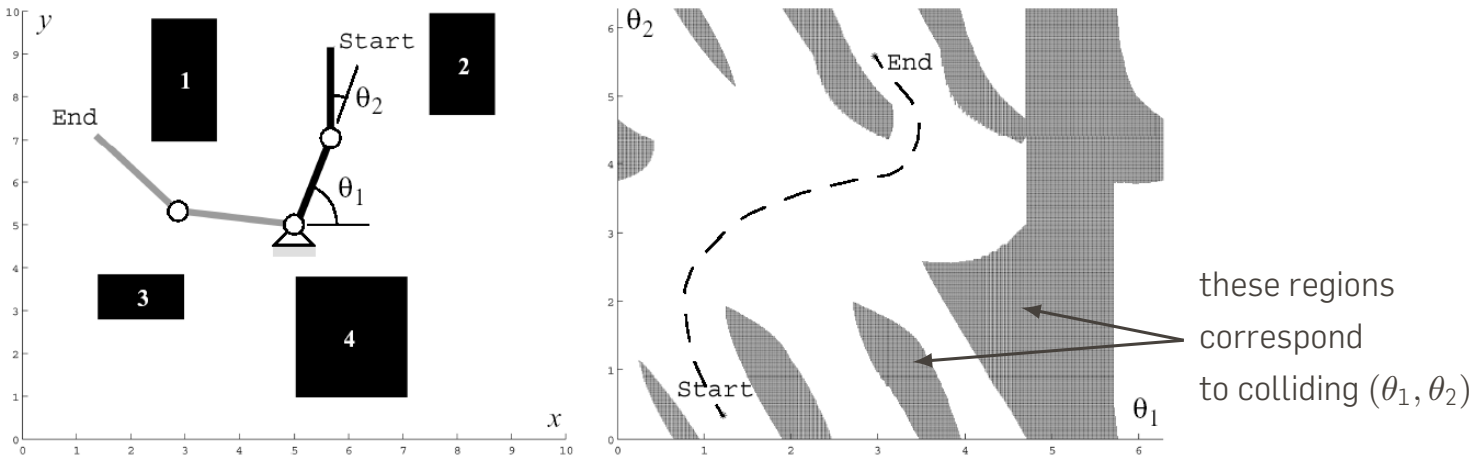
The state or configuration q of a robot can be described with k values q_i. For example, for a two-link planar robot, θ_1 and θ_2:



Each q is a point in a k-dimensional space called the **configuration space** C of the robot.

PATH PLANNING: CONFIGURATION SPACE

The state or configuration q of a robot can be described with k values q_i. For example, for a two-link planar robot, θ_1 and θ_2:

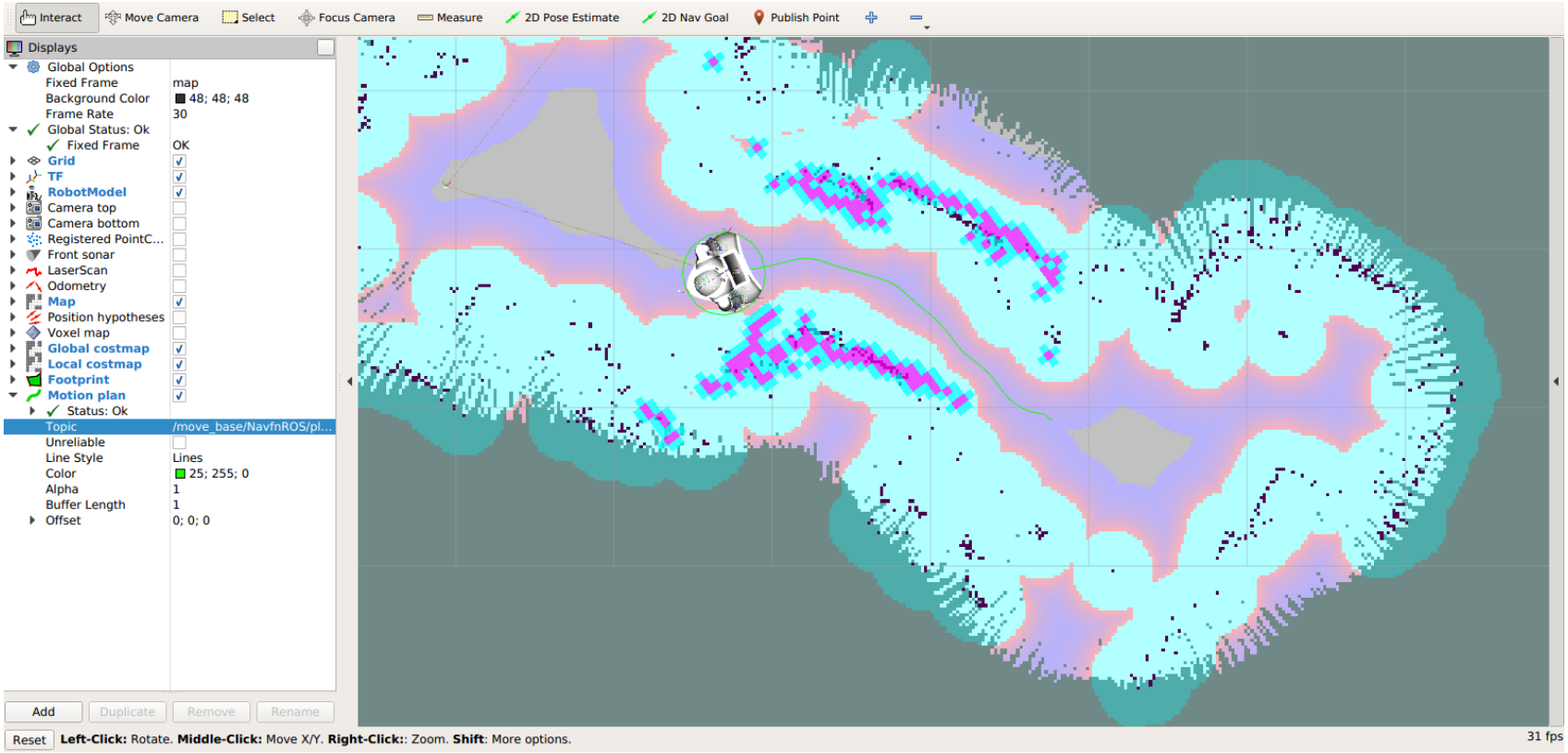


Each q is a point in a k-dimensional space called the **configuration space** C of the robot.

We can map as well obstacles to ***configuration space obstacles*** O.

The ***free space*** (where the robot is safe to move) is then $F = C - O$.

GLOBAL VS LOCAL PATH PLANNING



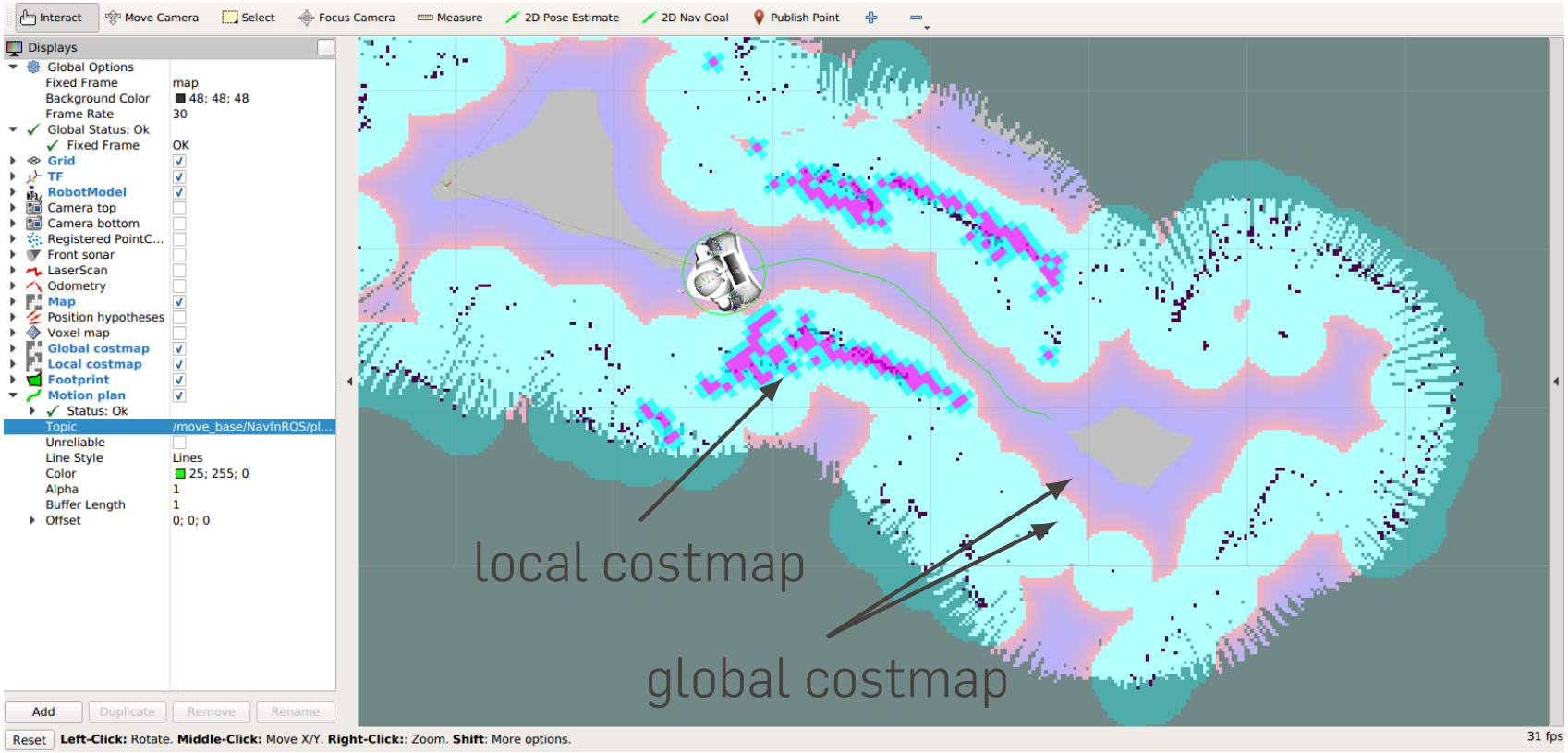
Global path planning

- long distances (i.e. large map, slower calculations)
- static environment

Local path planning

- short horizon
- dynamic environment (use of sensors)

GLOBAL VS LOCAL PATH PLANNING



Global path planning

→ global (static) **costmap**

Local path planning

→ local (dynamic) **costmap**

GLOBAL PATH PLANNING

Assumptions:

- A good enough map of the environment is available for navigation (topological or metric or a mixture between both).
- The robot knows where it is on the map (using for example GPS or Monte Carlo localisation).

GLOBAL PATH PLANNING

Assumptions:

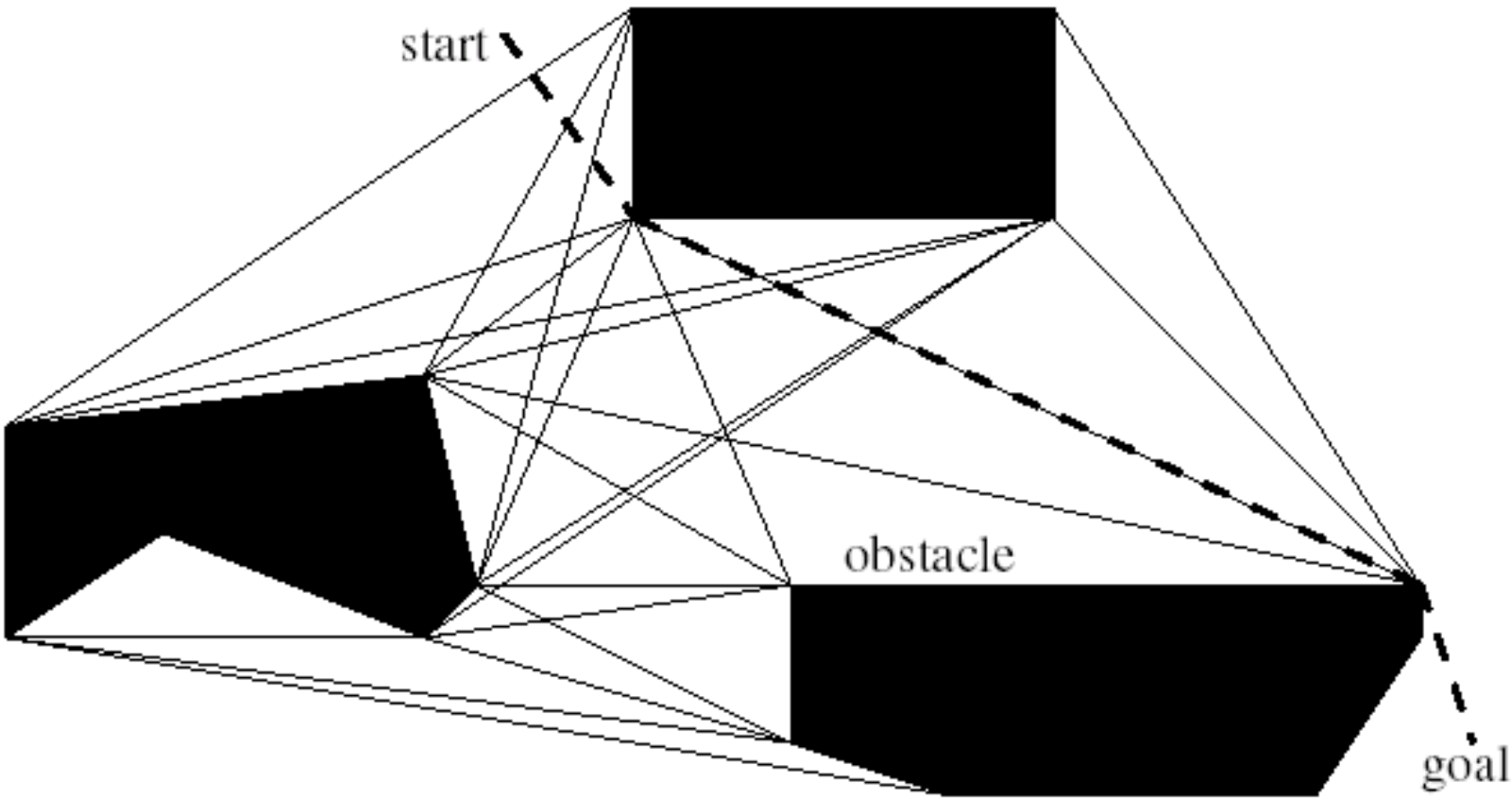
- A good enough map of the environment is available for navigation (topological or metric or a mixture between both).
- The robot knows where it is on the map (using for example GPS or Monte Carlo localisation).

First step: Build a representation of the environment using a **road-map (graph)**, **cells** or a **potential field**. The resulting *discrete* locations or cells allow then to use standard planning algorithms.

Possible algorithms:

- Visibility graph
- Voronoi diagram
- Cell decomposition $\rightarrow$ connectivity graph
- Potential field

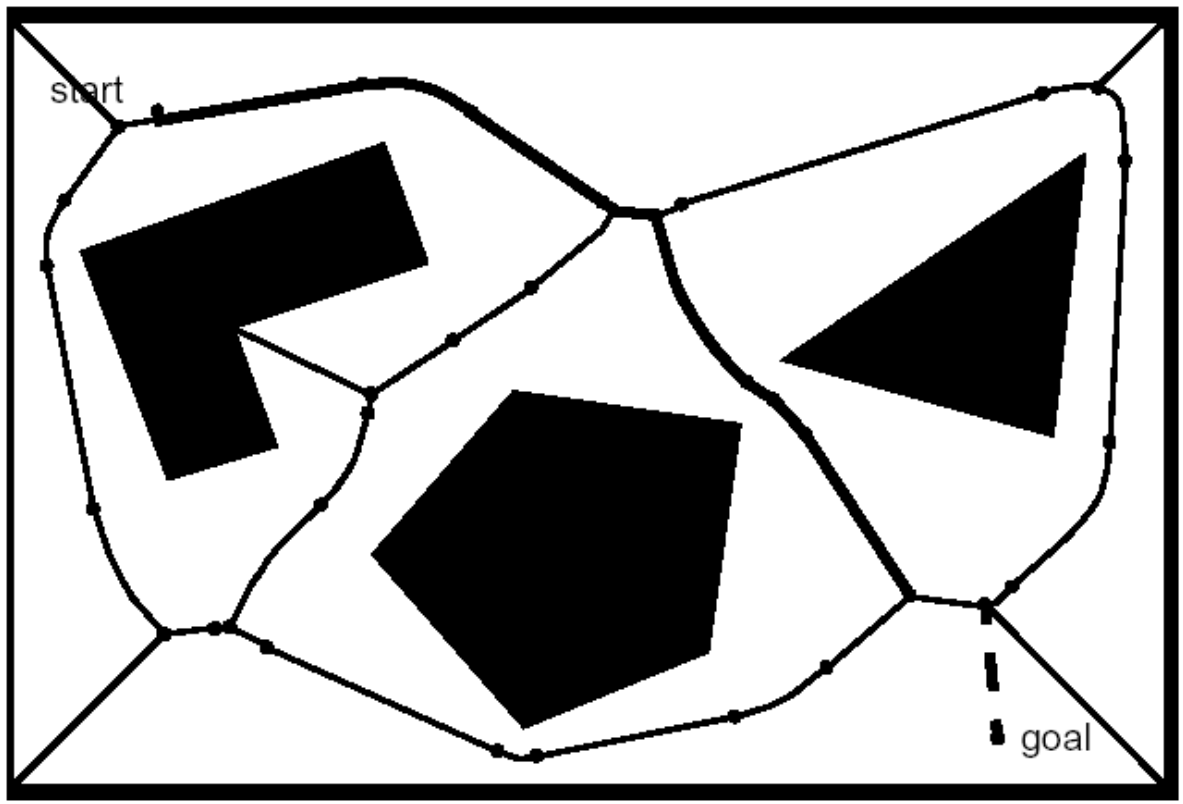
GRAPHS: VISIBILITY GRAPH



Edges between nodes are the lines of sight from each corner to an obstacle to every other visible corner.

GRAPHS: VORONOI GRAPHS

Voronoi graph: distance to obstacles from every nodes is maximal.



(by moving along the edges of a Voronoi graph, the robot ensures it is always as far as possible from obstacles)

GRAPHS: CELL DECOMPOSITION

Divide space into simple, connected regions called **cells**.

Determine which open cells are adjacent and construct a **connectivity graph**.

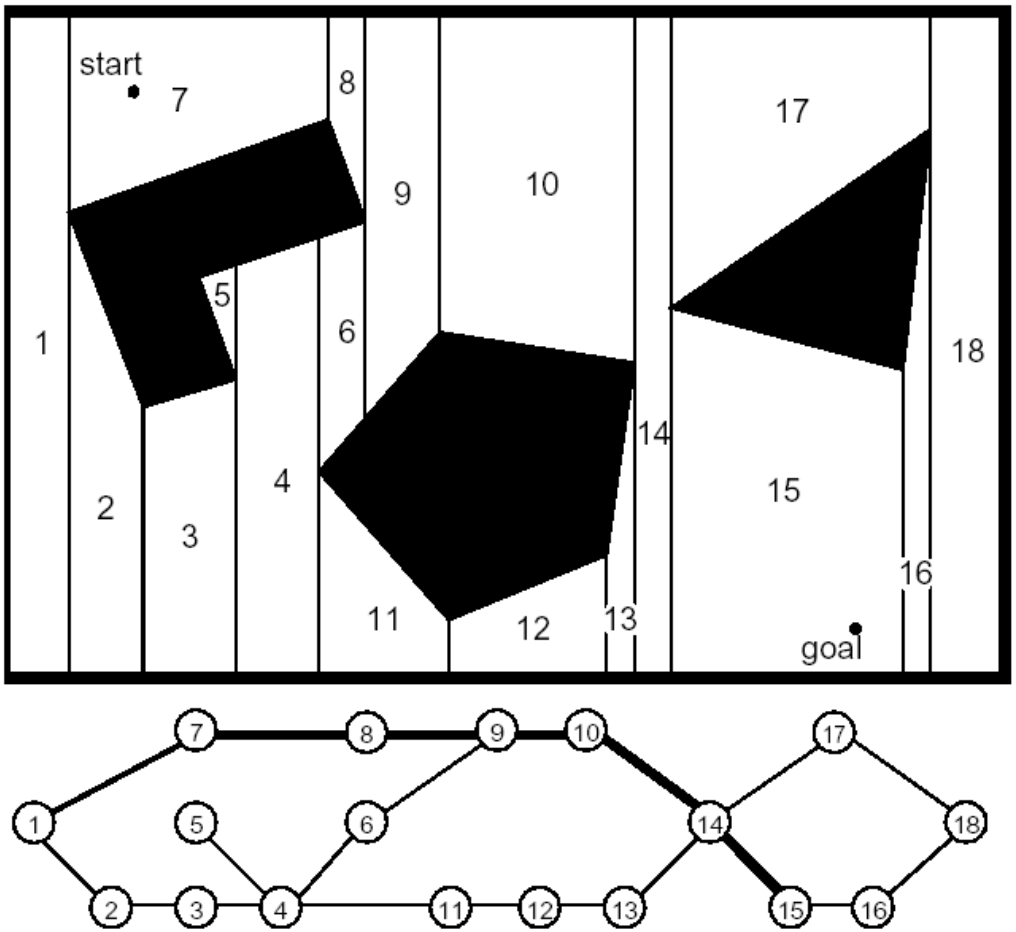
Find cells in which the initial and goal configuration (state) lie and search for a path in the connectivity graph to join them.

From the sequence of cells found with an appropriate search algorithm, compute a path within each cell.

e.g. passing through the midpoints of cell boundaries or by sequence of wall following movements.

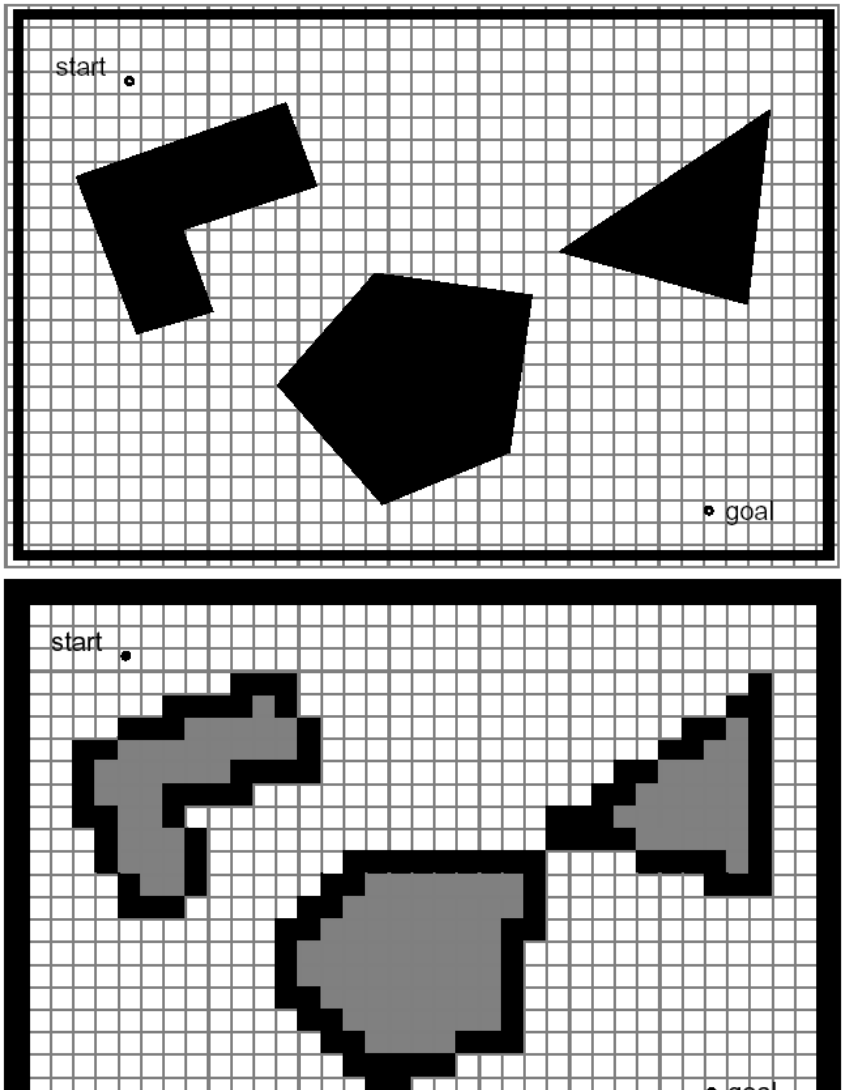
GRAPHS: CELL DECOMPOSITION

Example: exact cell decomposition



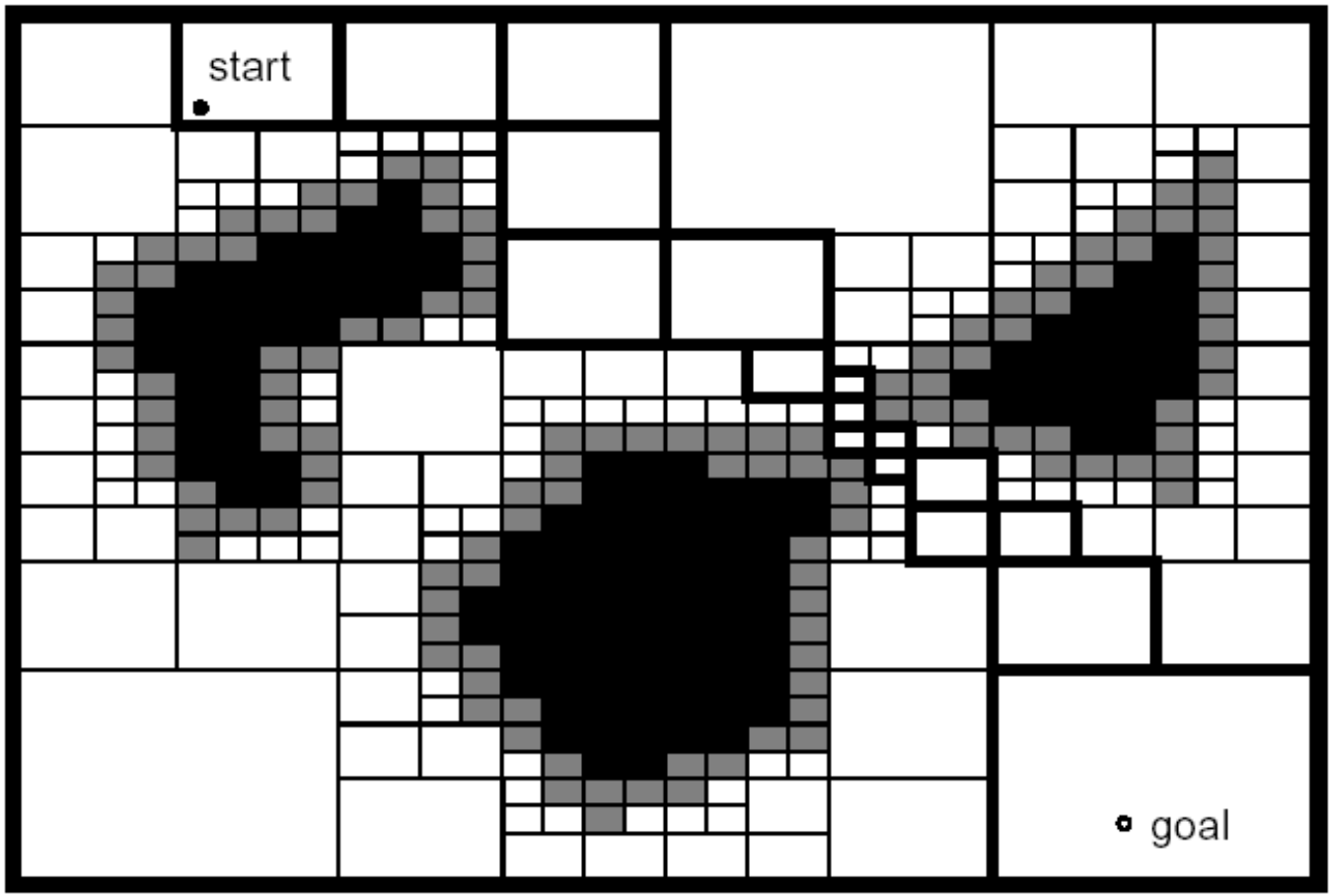
GRAPHS: CELL DECOMPOSITION

Example: approximate cell decomposition



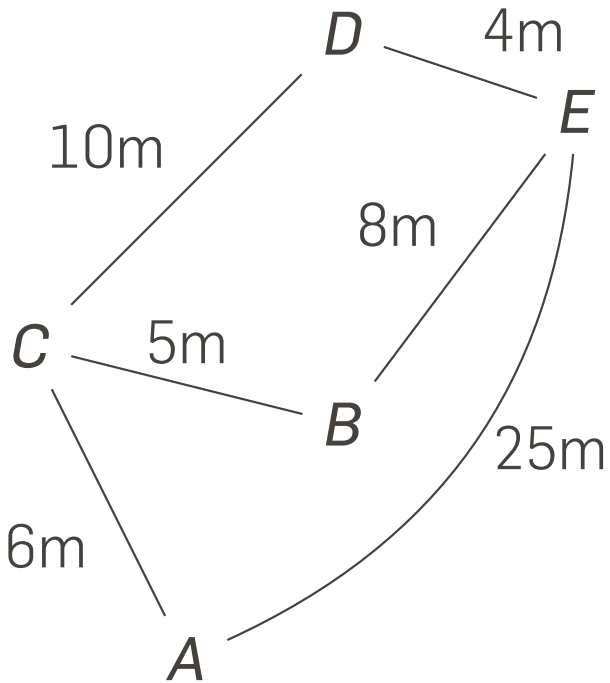
GRAPHS: CELL DECOMPOSITION

Adaptive cell decomposition: changes the size of each cell according to the detail needed.



REPRESENTING GRAPHS

- How is a graph represented in a computer? Graphical representation is intuitive, but not straightforward to implement in a program.
- Graphs can be represented as a matrix:



	A	**B**	**C**	**D**	**E**
A	0	-	6	-	25
B	-	0	5	-	8
C	6	5	0	10	-
D	-	-	10	0	4
E	25	8	-	4	0

SEARCHING GRAPHS

SEARCHING GRAPHS

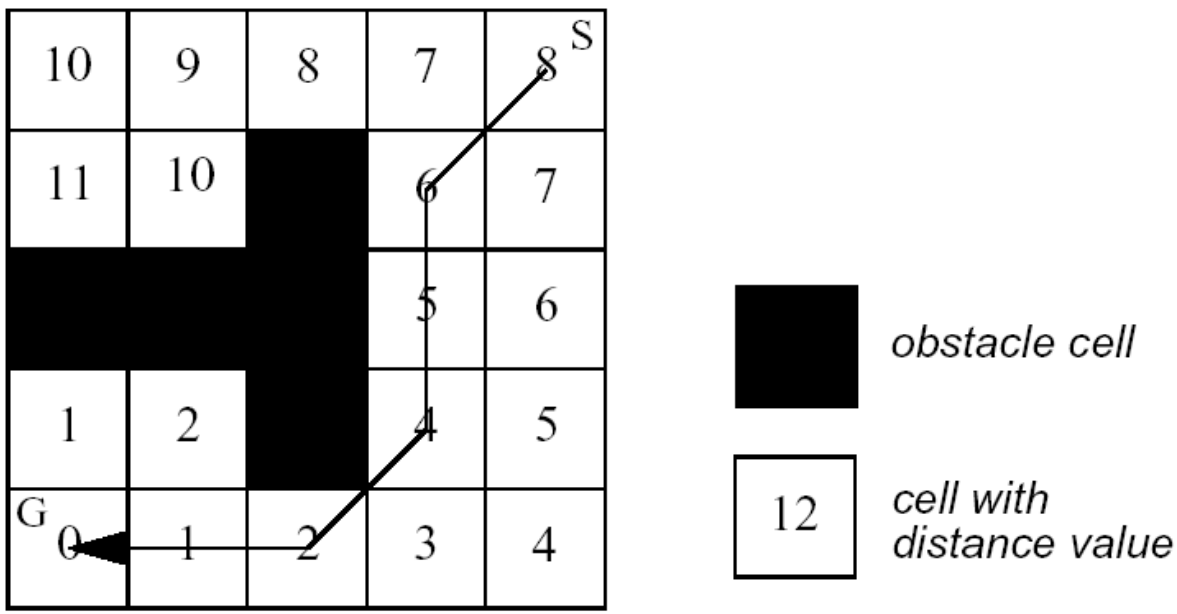
Graphs can be searched, for example to find the shortest path between two nodes.

Many different search algorithms exist

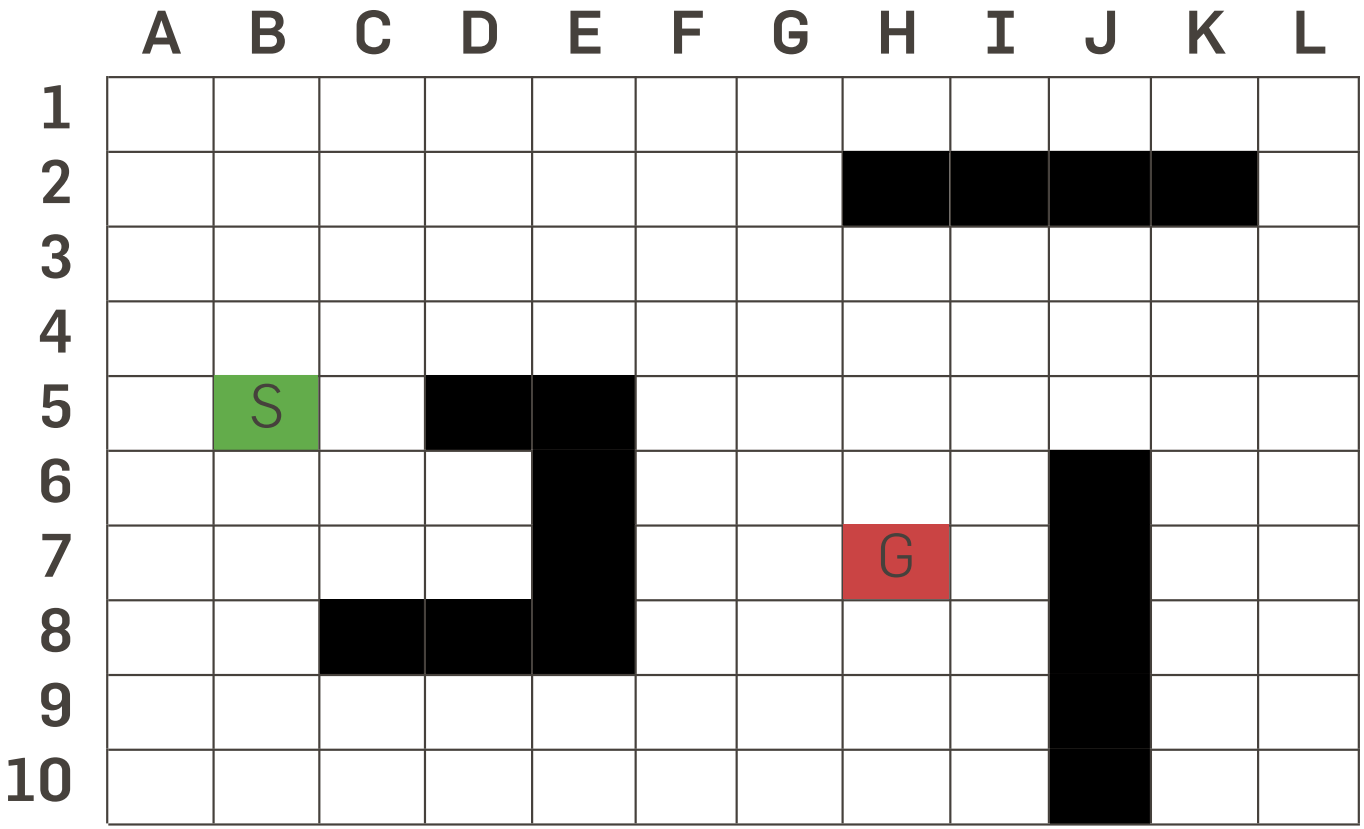
- Breadth first
- Depth first
- A*
- Dijkstra's shortest path

GRASSFIRE ALGORITHM

- Starting from a metric map, a **wave front** is expanded around the goal.
- Goal is reached by travelling on descending values.

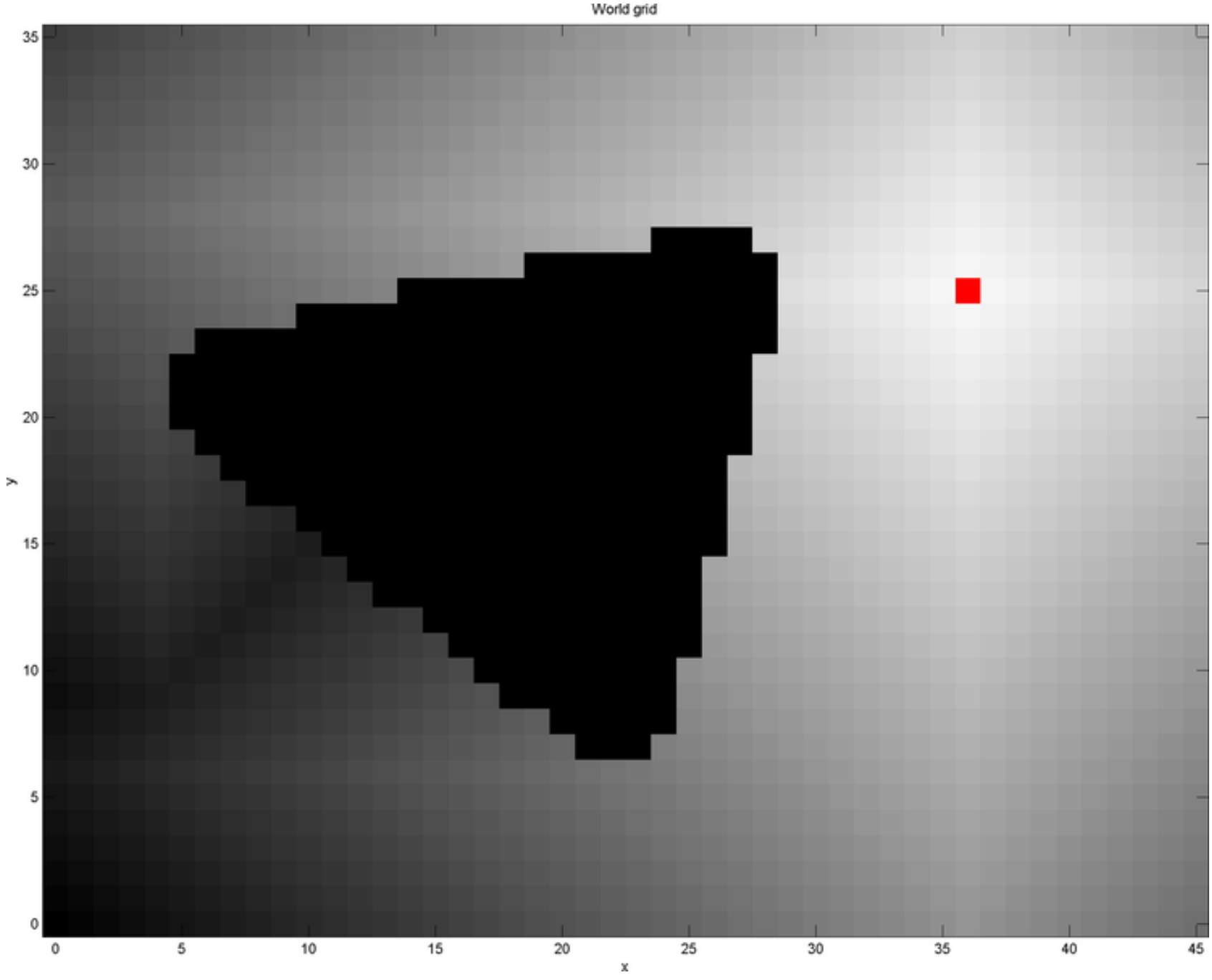


GRASSFIRE ALGORITHM



GRASSFIRE ALGORITHM

	A	B	C	D	E	F	G	H	I	J	K	L
1	13	12	11	10	9	8	7	8	9	10	11	10
2	12	11	10	9	8	7	6					9
3	11	10	9	8	7	6	5	4	5	6	7	8
4	10	9	8	7	6	5	4	3	4	5	6	7
5	11	S	9			4	3	2	3	4	5	6
6	12	11	10	11		3	2	1	2		6	7
7	11	10	11	12		2	1	G	1		7	8
8	10	9				3	2	1	2		8	9
9	9	8	7	6	5	4	3	2	3		9	10
10	10	9	8	7	6	5	4	3	4		10	11


World grid
35
30
25
20
15
10
5
0
y
0
5
10
15
20
25
30
35
40
45
x

GRASSFIRE ALGORITHM

```
def grassfire(M, goal):
    Q.push(goal)
    M[goal] = 0 # set goal value to 0

    while not Q.empty(): # loop until map filled
        a = Q.pop()

        for n in neighbours(a):
            if not n in M and not is_obstacle(n):
                Q.push(n)
                M[n] = M[a] + 1
    return M
```

Q is a queue data structure, a first-in first-out (fifo) list. The queue keeps track of which locations on the map still need to be visited.

DIJKSTRA'S ALGORITHM: ILLUSTRATION

Source: Wikipedia

Dijkstra search between start (red) and goal (green) positions.

Note how Dijkstra is expanding its search out from the starting position, without knowledge about which nodes could bring it closer to the goal.

SEARCHING GRAPHS - DIJKSTRA

```
def dijkstra(graph, weights, start):

  cost_to = {} # maps nodes to cost to 'start'
  come_from = {} # needed to reconstruct shortest path

  for node in graph:
    cost_to[node] = math.inf # initial cost from 'start' to 'node'

  cost_to[start] = 0

  frontier = PriorityQueue()
  frontier.put(graph.start, 0)

  while not frontier.empty():
    u = frontier.get_cheapest() # remove best node

    for v in u.neighbours: # iterate over nodes connected to u
      if cost_to[u] + weights(u, v) < cost_to[v]: # new shorter path to v!
        cost_to[v] = cost_to[u] + weights(u, v)
        frontier.put(v, cost_to[v])
        come_from[v] = u

  return cost_to, come_from
```

PRIORITY QUEUE

This `PriorityQueue` is based on a ***heap***.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children.

```
class PriorityQueue:
    """ Simple priority queue: get_cheapest always returns
    the node with the lowest 'priority' value.

    Implementation taken from
    https://www.redblobgames.com/pathfinding/a-star/implementation.html
    """
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get_cheapest(self):
        return heapq.heappop(self.elements)[1]
```

SEARCHING GRAPHS – DIJKSTRA

Reading out shortest path from the start node to goal:

```
path = []
node = goal
while node in come_from:
  path = [node] + path # append at the front of our path
  node = come_from[node]
```

Returns the shortest path (if there are more than one, only one is returned).

Returns empty if no path exists.

THE COST OF SEARCH

Maps can be huge

- For example, 10,000 m^2 shopping mall ($\sim$ Marshmills' Sainsburys), with a 10x10cm map resolution, requires 10^{10} = 10 billion nodes on the graph!

Search time can be problematic:

- Dijkstra has a complexity of $O(N + V^2)$. For example, for a map with 10^7 nodes and 10^8 vertices, it can take $10^7 + 10^{2\times 8} \sim 10^{16}$ calculations to find a shortest path between two points.
- Some implementations are much more efficient. A* search is at worst the same as Dijkstra, but can be polynomial in the number of nodes $O(N^k)$ when a good heuristic is used. For example, if you can choose between two nodes to explore, take the node closer to the goal.

THE COST OF SEARCH

Maps can be huge

- For example, 10,000 m^2 shopping mall ($\sim$ Marshmills' Sainsburys), with a 10x10cm map resolution, requires 10^{10} = 10 billion nodes on the graph!

Search time can be problematic:

- Dijkstra has a complexity of $O(N + V^2)$. For example, for a map with 10^7 nodes and 10^9 vertices, it can take $10^7 + 10^{2\times 8} \sim 10^{16}$ calculations to find a shortest path between two points.
- Some implementations are much more efficient. A* search is at worst the same as Dijkstra, but can be polynomial in the number of nodes $O(N^k)$ when a good heuristic is used. For example, if you can choose between two nodes to explore, take the node closer to the goal.

10^{16} sums $\approx$ 50h on a modern i7 @3.4GHz

A* ALGORITHM - ILLUSTRATION

Source: Wikipedia

A* search for finding path between start and goal position.

A* uses a **heuristic** (here the distance to the goal) to guide the exploration (first expand to nodes closer to the goal). This gives a chance of finding the solution in less time.

SEARCHING GRAPHS - A*

```
def a_star(graph, weights, start):

  cost_to = {} # maps nodes to distance to 'start_node'
  come_from = {} # needed to reconstruct shortest path

  for node in graph:
    dist[node] = math.inf # initial cost from 'start' to 'node'

  cost_to[start] = 0

  frontier = PriorityQueue()
  frontier.put(start, 0)

  while not frontier.empty():
    u = frontier.get_cheapest()

    for v in u.neighbours:
      if cost_to[u] + weights(u, v) < cost_to[v]: # new shorter path to v!
        cost_to[v] = cost_to[u] + weights(u, v)
        frontier.put(v, cost_to[v] + heuristic(v, goal))
        come_from[v] = u

  return came_from, cost_so_far
```

A* HEURISTICS

The heuristic is task-specific. If the map is known, a simple L1 distance is a common choice:

```
def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)
```

You can read more on A* heuristics here.

POTENTIAL FIELD PATH PLANNING

POTENTIAL FIELD PATH PLANNING

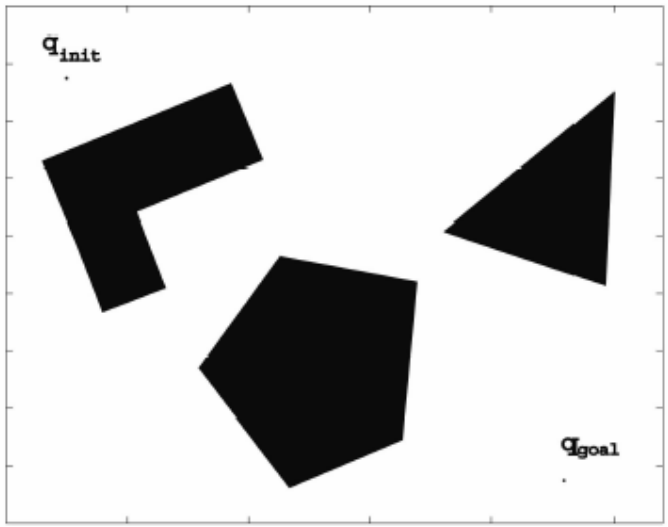
Robot is treated as a ***point under the influence*** of an artificial potential field.

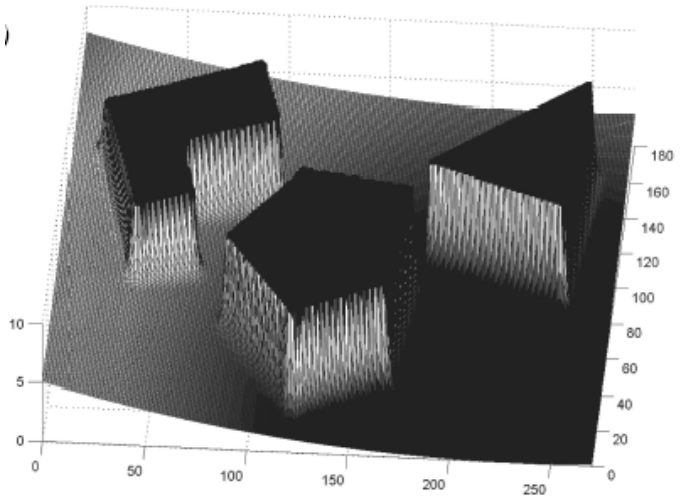
- Generated robot movement is similar to a ball rolling down the hill
- Goal generates attractive force
- Obstacle are repulsive forces

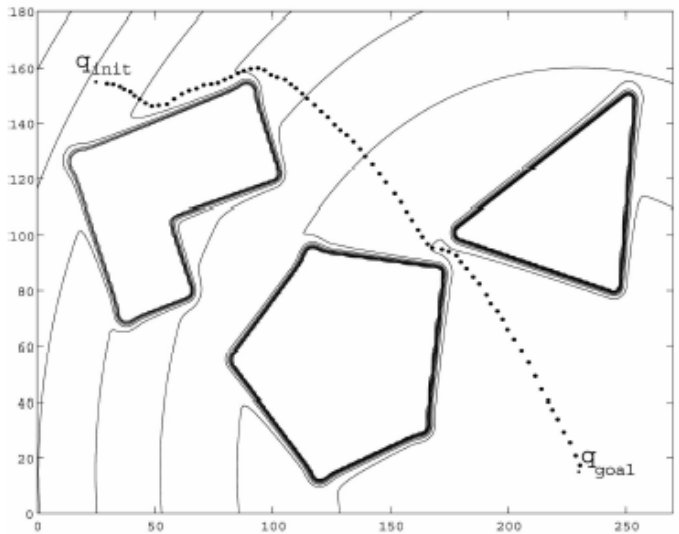
Needed:

- Location of robot
- Location of obstacles
- $\Rightarrow$ a metric map is well suited

POTENTIAL FIELD PATH PLANNING







POTENTIAL FIELD GENERATION

Generation of potential field function $U(q)$

- attracting (goal) and repulsing (obstacle) fields
- summing up the fields
- functions must be differentiable

POTENTIAL FIELD GENERATION

Generation of potential field function $U(q)$

- attracting (goal) and repulsing (obstacle) fields
- summing up the fields
- functions must be differentiable

Generate artificial force field $F(q)$

$$F(q) = -\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) = \begin{bmatrix} \frac{\partial U}{x} \\ \frac{\partial U}{y} \end{bmatrix}$$

POTENTIAL FIELD GENERATION

Generation of potential field function $U(q)$

- attracting (goal) and repulsing (obstacle) fields
- summing up the fields
- functions must be differentiable

Generate artificial force field $F(q)$

$$F(q) = -\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) = \begin{bmatrix} \frac{\partial U}{x} \\ \frac{\partial U}{y} \end{bmatrix}$$

Set robot speed (v_x, v_y) proportional to the force $F(q)$ generated by the field

- the force field drives the robot to the goal
- if robot is assumed to be a point mass

POTENTIAL FIELD PATH PLANNING

Attractive potential field

- For example, a parabolic function representing the Euclidean distance $\rho = ||q - q_{goal}||$ to the goal:

$$U_{att}(q) = \frac{1}{2} k_{att} \cdot \rho_{goal}^2(q)$$

- Attracting force converges linearly towards 0 (goal)

$$\begin{aligned} F_{att}(q) &= -\nabla U_{att}(q) \\ &= -k_{att} \cdot \rho_{goal}(q) \nabla \rho_{goal}(q) \\ &= -k_{att} \cdot (q - q_{goal}) \end{aligned}$$

POTENTIAL FIELD PATH PLANNING

Repulsing potential field

Should generate a barrier around all the obstacle

- strong if close to the obstacle
- no influence if far from the obstacle:

distance to obstacle

$$U_{rep}(q) = \begin{cases} \frac{1}{2} k_{rep} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right)^2 & \text{if } \rho(q) < \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases}$$

- field is positive or zero and ***tends to infinity*** as q gets closer to the object:

$$F_{rep}(q) = -\nabla U_{rep}(q) = \begin{cases} k_{rep} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right) \cdot \frac{1}{\rho(q)^2} & \text{if } \rho(q) < \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases}$$

POTENTIAL FIELD PATH PLANNING

Notes:

- **Local minima problem** exists: robot can get stuck in places on the map where the potential field has a local dip.
- problem is getting more complex if the robot is not considered as a point mass
- If objects are convex there exists situations where several minimal distances exist $\rightarrow$ can result in oscillations
- Robot path oscillates, e.g. in narrow passages.

Videos:

- **http://www.youtube.com/watch?v=DxzRYYMjxKY**
- **http://www.youtube.com/watch?v=JnPfu7xHNt4**

EXTENDED POTENTIAL FIELD PATH PLANNING

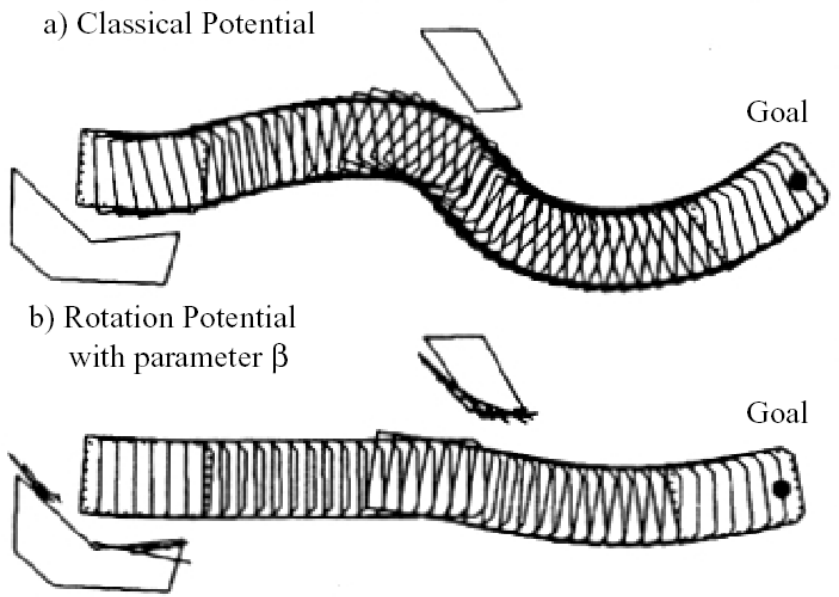
Additionally a **rotation potential field** and a **task potential field** are introduced.

Rotation potential field

- force is also a function of robots orientation to the obstacle

Task potential field

- filters out the obstacles that should not influence the robots movements, i.e. only the obstacles in front of the robot are considered



That's all, folks!

Questions:
Portland Square B316 or **severin.lemaignan@plymouth.ac.uk**

Slides:
github.com/severin-lemaignan/module-mobile-and-humanoid-robots