Robotics
Robot Navigation (1)

Tullio Facchinetti
<tullio.facchinetti@unipv.it>

Tuesday 3rd October, 2023

http://robot.unipv.it/toolleeo

Robot navigation

Robot navigation

Robot's ability to determine its own position in its frame of reference and then to plan a path towards some goal location.

Source: Wikipedia

sub-problems to address:

- localization
- path planning
- mapping

Problems to address

localization

- determination of the current robot configuration/position

path planning

- find a collision-free path to move from a starting configuration to a destination configuration

mapping

- environment exploration to build a map of the configuration space; useful for path planning, coverage and localization

Example applications requiring navigation

manipulation and grasping

- manufacturing
- tele-medicine (e.g. remote surgery)

assembly planning

- manufacturing
- coverage: let a sensor or an actuator to cover the working space
- special interventions (e.g. space stations)

multi-robot coordination

- object transportation
- improvement in area coverage
- wireless connectivity preservation

Basic terminology

system

- set of particles composing the moving object (the robot)

configuration

- the position of each point composing the system

configuration space

- set of all the possible configurations

degree of freedom

- the dimension of the configuration space

Obstacles and free space

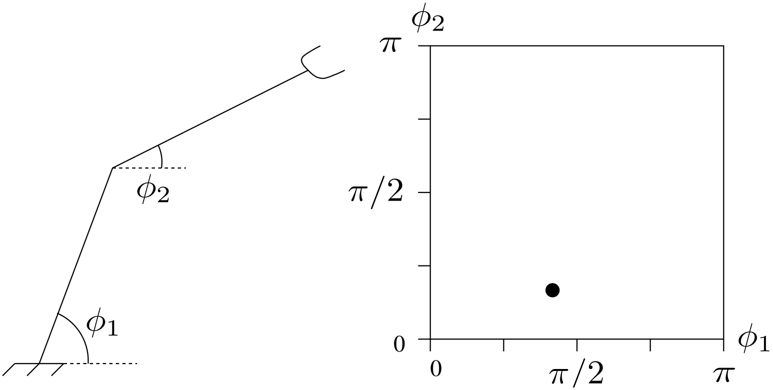
working space

- working space W
- the i-th obstacle is denoted as WO_i
- the free space is $W_{free} = W \setminus (\bigcup_i WO_i)$

configuration space

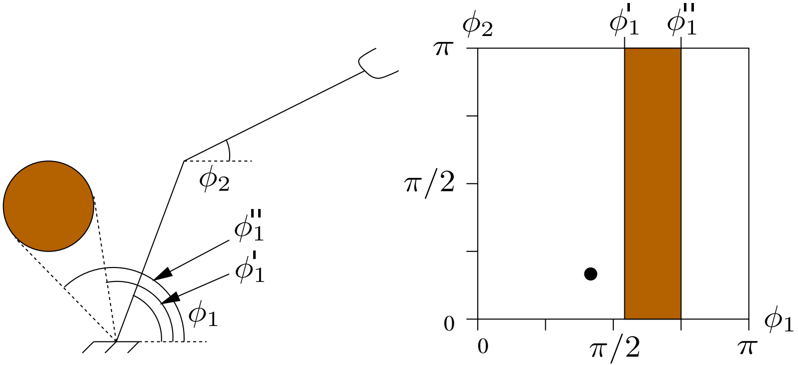
- configuration space Q
- $R(q)$: points occupied by the robots at configuration q
- the i-th obstacle is denoted as QO_i
- the free configuration space is $Q_{free} = Q \setminus (\bigcup_i QO_i)$

Configuration space: an example



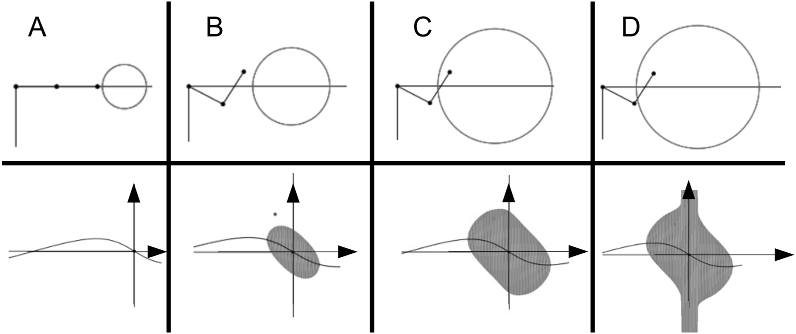
configuration space of a two-arm robot moving in the 2-dimensional plane

Path planning with obstacles: modeling



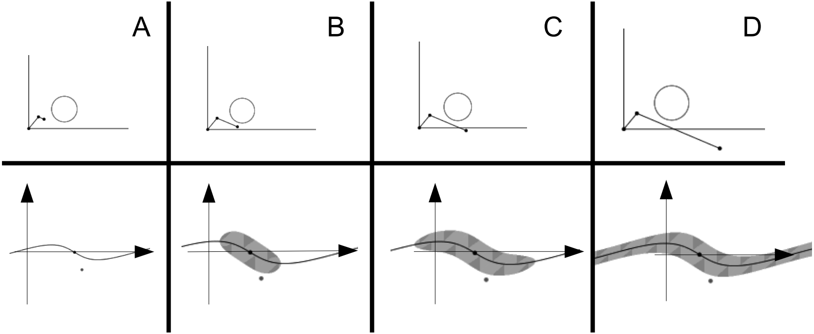
- $\phi_1, \phi_2 \in [0, \pi]$
- an obstacle in the working space corresponds to a set of non-allowed configurations in the configuration space (the above is a sub-set that is easy to draw by hand)

Configuration space with obstacles: an example



changing the obstacle radius

Configuration space with obstacles: an example



changing the link length

Path planning: lesson learned

the configuration of a robot can be represented as one point in a n-dimensional configuration space

- the value of n depends on the mechanical structure of the robot (degree of freedom)
- the representation of an obstacle in the configuration space depends both on the shape of the object **AND** the structure of the robot

the motion of a complex robot (several degrees of freedom) in the working space is mapped into the motion of one point in a complex (several dimensions) configuration space

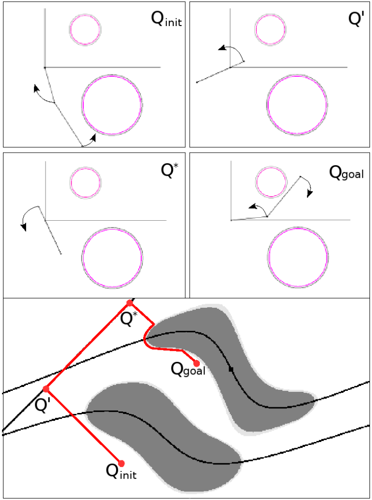
Path planning: the goal

the goal of the path planning is to let a point move in the configuration space

- the movement goes from a starting point q_{start} to a destination point q_{goal}
- configurations QO_i are present in the configuration space that are not allowed
- an obstacle in the operating space is associated with configurations that are not allowed in the configuration space
- the path planning shall avoid obstacles

Path planning: example

the motion of a point in the configuration space is associated with the motion of an arm in the workspace



The path/trajectory planning

Path

A continuous curve in the configuration space

Trajectory

A continuous curve in the configuration space parameterized by time

in the remainder, the focus will be on path planning, thus the term "navigation" will be (mostly) restricted to that topic

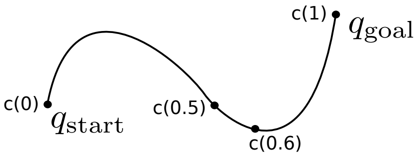
The path/trajectory planning

path

$$c : [0,1] \to Q$$

where

- $c(0) = q_{start}$, and
- $c(1) = q_{goal}$, and
- $c(s) \in Q_{free} \forall s \in [0,1]$

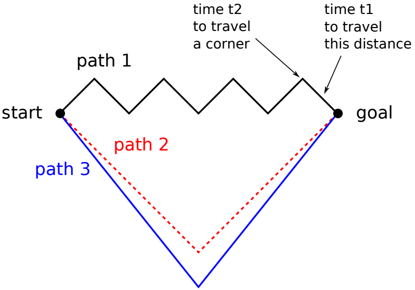


when c is parametrized by t it becomes a trajectory

Properties of a path planning algorithm

optimality: is it the best algorithm?

Performance evaluation can be based on: path length, required time, consumed energy



1. L_i = length(path i)
2. $L_1 = L_2 < L_3$
3. N_i = corners(path i)
4. $N_1 > N_2 = N_3$ $(7 > 1 = 1)$
5. T_i = time(path i)
6. $T_2 < T_3$, $T_2 < T_1$, T_1 ? T_3

While the comparison among lengths is straigthforward, the comparison among times depends from the time t_1 to cover the straight lines and the time t_2 to handle the corners

Properties of a path planning algorithm

computational complexity:
(how long does it take to find a path?)

- constant, polynomial or exponential complexity as a function of the problem size
- the problem size can be expressed in terms of degree of freedom, number of obstacles, etc.
- evaluate the average complexity and the worst case complexity

Complexity: example

an algorithm requires 50 ms to execute the instruction that processes 1 single datum

supposing that we have 50 data to process, the required time is:

- $O(1)$: e.g. 80 ms, which does not depend on the number of data
- $O(\log n)$: in the order of 195.6 ms
- $O(n)$: in the order of 2.5 sec
- $O(n^3)$: in the order of 125 sec
- $O(2^n)$: in the order of 1.12×10^{12} sec, i.e., 35.702.000 year

Properties of a path planning algorithm

completeness

- a **complete** algorithm finds a solution if one exists
- resolution completeness: a solution can be found only above a given resolution of the problem representation
- probabilistic completeness: the probability p to find a solution tends to 100% as $t \to \infty$

> optimality, completeness and complexity are
> trade-off parameters

e.g. the complexity may increase if optimality or completeness is required

Offline/online execution

offline

- given all the necessary information, a path is calculated in advance
- later, the robot will follow the pre-computed path
- the environment must be known in advance to obtain a correct/safe/reliable path

online

- the path is generated while the robot is moving
- the information required for the navigation are collected during the motion (i.e., online), using the information gathered by sensors
- do not require the a-priori knowledge of the environment

Two-dimensional motion: the bugs algorithms

a family of 3 algorithms based
on similar strategies

features:

- designed to manage the presence of obstacles
- work for 2-dimensional configuration spaces
- do not work for higher dimensional spaces

requirements:

- self localization (can use maps, GPS, etc.)
- coordinates of the start and destination points
- proximity sensing

Bugs algorithms

complete algorithms: a solution is found, if one exists

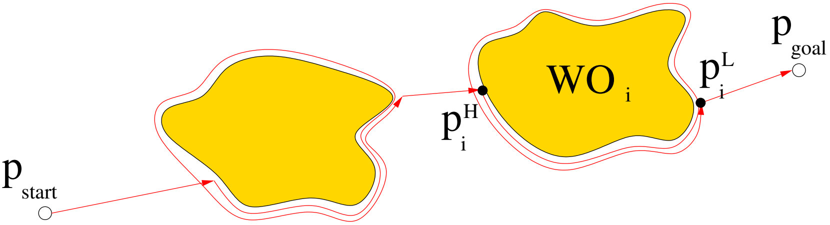
combination of 2 motion strategies:

- motion-to-goal: move towards the goal point
- boundary-following: run along the border of an obstacle

Bug 1

essentials:

- motion-to-goal until an obstacle is detected (hit point)
- complete circumnavigation of the obstacle to find the point p_i^L closest to the goal (leave point)
- return to p_i^L **along the shortest path** and back to motion-to-goal

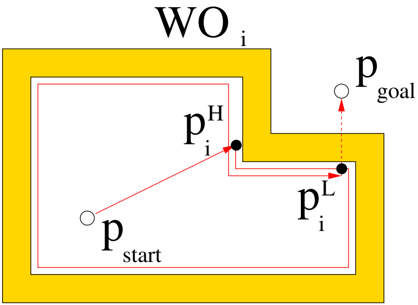


Bug 1 pseudo-code

```
i = 1
p^L_{i-1} = p_start
while forever do
    repeat
        move from p^L_{i-1} to p_goal
    until (p_goal is reached → path found) or (WO_i encountered in p^H_i)
    select a direction (left or right)
    repeat
        follow the boundary of WO_i
    until (p_goal is reached → path found) or (p^H_i is encountered)
    determine the closest point p^L_i ∈ ∂WO_i to p_goal
    boundary following towards p^L_i, along the shortest path
    move towards the goal
    if WO_i is encountered then
        p_goal is not reachable
        stop
    end if
    i = i + 1
end while
```

Bug 1: no path to goal

example where a path to goal
can not be found



Bug 1: proof of completeness

> an algorithm is complete if, in finite time, it finds a path if such a path exists or terminates with failure if it does not

suppose Bug 1 were incomplete

this means that

- there is a path from start to goal
- by assumption, it has finite length, and intersects obstacles a finite number of times
- Bug 1 does not find it
  - either it spends an infinite amount of time looking for the goal (it never terminates), or
  - it terminates incorrectly (determines that there are not paths to goal)

Bug 1: proof of completeness

suppose it never terminates

- but each leave point is closer to p_{goal} than corresponding hit point
- each hit point is closer than the previous leave point
- thus, there are a finite number of hit/leave pairs
- after exhausting them, the robot will proceed to the goal and terminate

Bug 1: proof of completeness

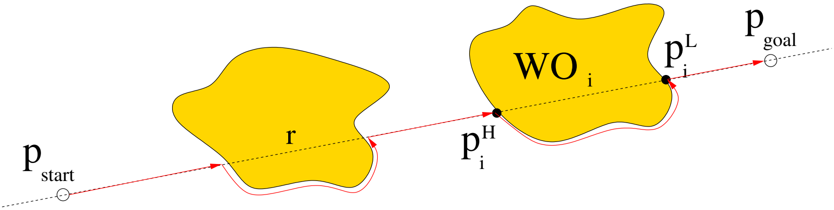
suppose it terminates with no path found (incorrectly)

- then, the closest point after a hit must be a leave point where the robot would have to move into the obstacle
- but, then line from robot to goal must intersect the object an even number of times (Jordan curve theorem)
- but then there is another intersection point on the boundary that is closer to the goal
- since we assumed there is a path, we must have crossed this point on the boundary, which contradicts the above assumption about the leave point

Bug 2

essentials:

- motion-to-goal until an obstacle is encountered
- obstacle circumnavigation until the *r* straight line is encountered in a point that is closer to the goal than the previous hit point
  - the *r* straight line is the line passing through the starting point and the goal
- at that point, back to motion-to-goal along the *r* straight line

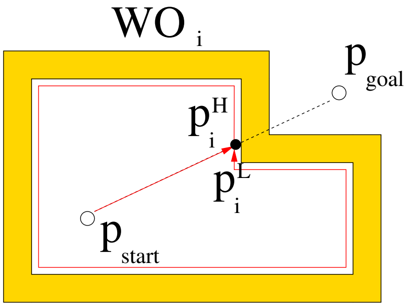


Bug 2: pseuso-code

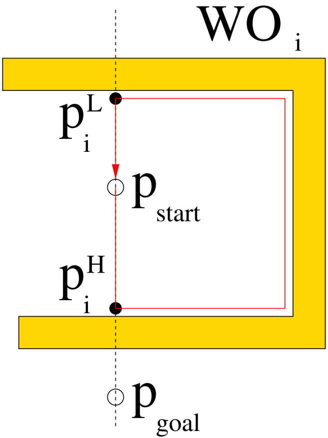
i = 1
$p^L_{i-1} = p_{\text{start}}$
while forever **do**
  repeat
    move from p^L_{i-1} to p_{goal}
  until (p_{goal} is reached → path found) **or** ($\mathcal{WO}_i$ encountered in p^H_i)
  select a direction (left or right)
  repeat
    follow the boundary of $\mathcal{WO}_i$
  until (p_{goal} is reached → path found) **or**
    (p^H_i is encountered again → no path exists) **or**
    r is crossed in point m **such that**
      $m \neq p^H_i$ (the robot did not get back to the hit point)
      $d(m, p_{\text{goal}}) < d(p^H_i, p_{\text{goal}})$ (the robot got closer to the goal)
      if the robot moves towards p_{goal} it does not encounter an obstacle
  set $p^L_i = m$
  i = i + 1
end while

Bug 2: no path to goal

example where no path exists
connecting the starting point and the goal



Bug 2: odd condition



- may this situation happen?
- if not, which is the condition that prevents it?

- when r is intersected during the boundary following, the path goes down r only if the intersection point is closer to the goal than the hit point
- when in motion-to-goal (i.e., moving along r), the point never goes in a direction that takes it farther from the goal

Bug 1 and 2: performance comparison

performance indicator: path length

which is the method that achieves the shortest path in the worst-case?

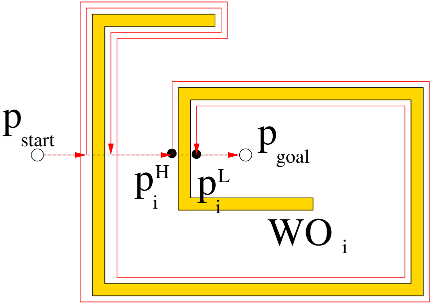
(qualitative observations)

- Bug 1 always goes through the entire perimeter o_i of the i-th obstacle once

instead...

- Bug 2 may cross the straight line r several (n_i) times for the i-th obstacle
- this fact may lead to cover the obstacle perimeter o_i several times

Bug 2: example of bad case



- the r straight line can intersect n_i times the boundary of the i-th obstacle
- therefore, there are $n_i/2$ pairs of hit/leave points
- in the worst case, this leads to cover many times the same parts of the perimeter

Performance comparison

a more accurare comparison of the worst case can be done considering that n obstacles are encountered by both algorithms

path length generated by Bug 1:

$$L_{\text{bug1}} \leq d(p_{\text{start}}, p_{\text{goal}}) + 1.5 \sum_{i=1}^{n} o_i$$

path length generated by Bug 2:

$$L_{\text{bug2}} \leq d(p_{\text{start}}, p_{\text{goal}}) + \frac{1}{2} \sum_{i=1}^{n} n_i o_i$$

Performance comparison

- in the worst case, the path generated by Bug 2 may quickly increase
- with Bug 2, the path length depends on how many times an obstacle is crossed by the r straight line
- an obstacle can be arbitrary complex, such that it is crossed by r an high number of times
- the performance of the algorithm strongly depends from the complexity of the environment

Two approaches, different features

Bug 1 and Bug 2 implement two common approaches available in operational research

- Bug 1 performs an exaustive research to (locally) find the optimal leave point
- Bug 2 uses heuristic research to limit the search time
- the heuristic adopted by Bug 2 is said greedy, i.e., the first option that promise good results is selected

as a consequence:

- Bug 2 provides good performance in case of simple obstacles
- generally, Bug 1 performs better in case of complex scenarios

A model for a range sensor

- the distance is given by the function $\rho : \mathbb{R}^2 \times S^1 \to \mathbb{R}$
- given a position $x \in \mathbb{R}^2$ and an orientation $\theta \in S^1$, the function is

$$\rho(x,\theta) = \min_{\lambda \in [0,\infty]} d(x, x + \lambda[\cos\theta, \sin\theta]^T)$$

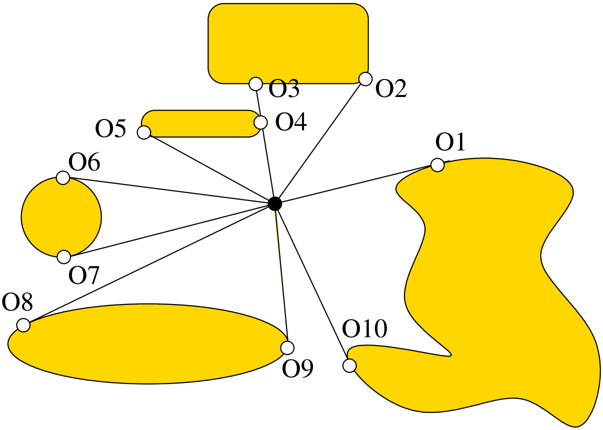
$$\text{such that } x + \lambda[\cos\theta, \sin\theta]^T \in \bigcup_i \mathcal{WO}_i$$

Discontinuity of ρ

points of discontinuity of the ρ function are especially relevant: they indicate the presence of a passage between two obstacles

- a continuity interval is defined as a connected interval $x + \rho(x, \theta)[\cos\theta, \sin\theta]$ such that $\rho(x, \theta)$ is finite and continuous w.r.t. θ
- the limits of continuity intervals compose the set O_i

Example of sensor with infinite sensing range



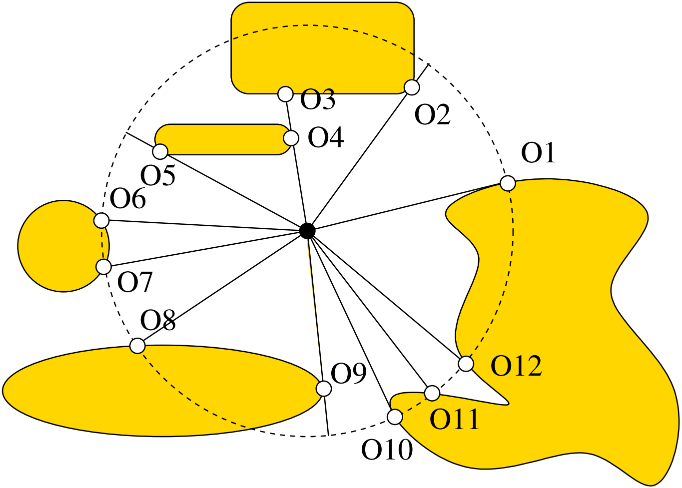
connected interval $x + \rho(x, \theta)[\cos\theta, \sin\theta]$
such that $\rho(x, \theta)$ is finite and continuous w.r.t. θ

Model of a real range sensor

- a real range sensor has a finite sensing range
- being R the sensing range, the function $\rho_R : \mathbb{R}^2 \times S^1 \to \mathbb{R}$ is said saturated distance

$$\rho_R(x,\theta) = \begin{cases} \rho(x,\theta), & \text{if } \rho(x,\theta) < R \\ \infty, & \text{otherwise} \end{cases}$$

Example of sensor with finite sensing range



Tangent Bug

still uses the two motion modes, namely motion-to-goal and boundary-following

however, differently from Bug 1 and Bug 2:

- in motion-to-goal the robot can run along the obstacle border
- in boundary-following mode the robot may travel without considering the obstacle border

the names of the strategies may be misleading: they are only used to identify a motion state

Tangent Bug

- during the motion-to-goal the robot moves along the direction that minimized a cost function, such as $d(x, O_i) + d(O_i, p_{\text{goal}})$
- when a local minimum of the cost function is found, it switches to the boundary-following mode
- in boundary-following mode 2 values are considered:
  - d_{followed}, which is the minimum distance from the goal registered during the current boundary-following motion
  - the value d_{reach} calculated ad follows:

$$\Lambda = \{y \in \partial \mathcal{WO}_f : \lambda x + (1 - \lambda) y \in \mathcal{Q}_{\text{free}} \forall \lambda \in [0, 1]\}$$

$$d_{\text{reach}} = \min_{c \in \Lambda} d(p_{\text{goal}}, c)$$

- the robot switches back to motion-to-goal when $d_{\text{reach}} < d_{\text{followed}}$

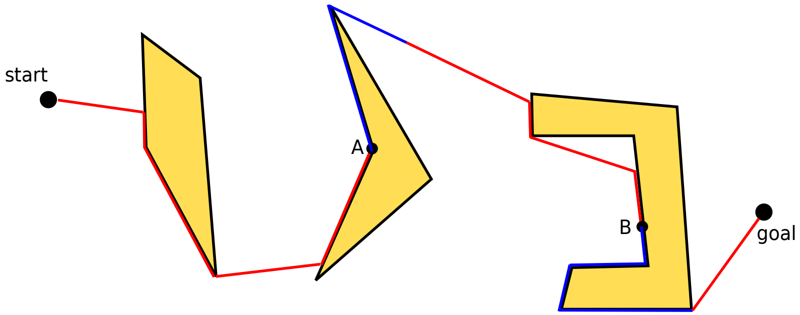
Tangent Bug

the Tangent Bug algorithm behavior depends on the sensing range of the range sensor

there are 3 cases:

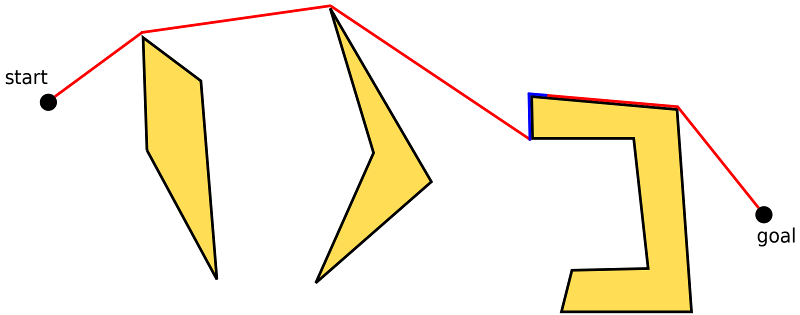
- range $R = 0$ (typical of a tactile sensor)
- range $R = \infty$ (the ideal situation)
- range $R > 0$ but finite (real range sensor)

Tangent Bug with $R = 0$



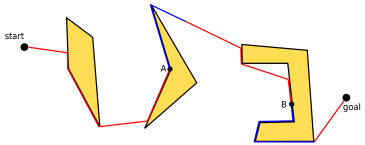
- the red line represents the motion-to-goal, while the blue line indicates the boundary-following
- points A and B indicate two local minima of the cost function

Tangent Bug with $R = \infty$

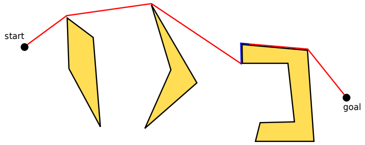


- the red line represents the motion-to-goal, while the blue line indicates the boundary-following

Tangent Bug: comparison between $R = 0$ and $R = \infty$



$R = 0$



$R = \infty$

the higher the sensing range, the better the performance of the algorithm in terms of length of the generated path

Potential fields method

pros

- does not require global information
- works in n-dimensional configuration spaces
- easy to implement and to visualize; this latter improves the predictability of the motion
- efficient implementation: fields are independent from each others, each field can be independently computed
- possibility to add custom parameters to tweak the desired behavior, both at design time and runtime
- the approach can be extended to non-Euclidean spaces

cons

- suffers of the local minima problem
- lack of completeness: may not find a path even if one exists

Potential fields and gradient

it is based on a potential field function such as

$$U(p) : \mathrm{R}^n \to \mathrm{R}$$

$p \in \mathrm{R}^n$ is the point where the potential is calculated

the gradient function can be obtained as

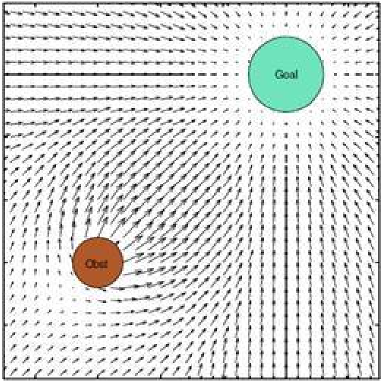
$$\nabla U(p) = DU(p)^T = \left[\frac{\partial U}{\partial p_1} \dots \frac{\partial U}{\partial p_n}\right]^T$$

physical meaning:

- the potential can be considered as the energy level in the point p
- its gradient has the features of a force applied on the moving point when located in p

Comparison with a physical system

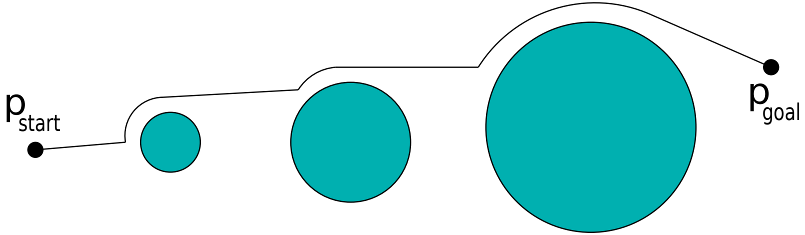
the point moving in the configuration space can be seen as a particle moving in a force field, which tends to a state of minimum energy



Comparison with a physical system

the point moving in the configuration space can be seen as a particle moving in a force field, which tends to a state of minimum energy

in presence of obstacles:



Attraction and repulsion

the overall potential is composed by the sum of 2 components:

$$U(p) = U_{att}(p) + U_{rep}(p)$$

- the attraction potential $U_{att}(p)$ attracts the particle; it is associated with the goal
- the repulsion potential $U_{rep}(p)$ repulses the particle; it is associated with obstacles

Attraction and repulsion

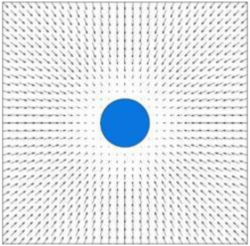
the force acting on the moving point is

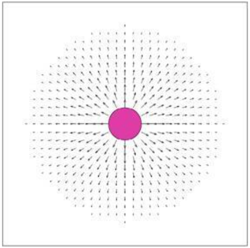
$$F(p) = F_{att}(p) + F_{rep}(p)$$

where

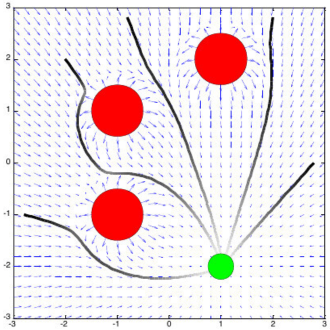
$$F_{att}(p) = -\nabla U_{att}(p)$$

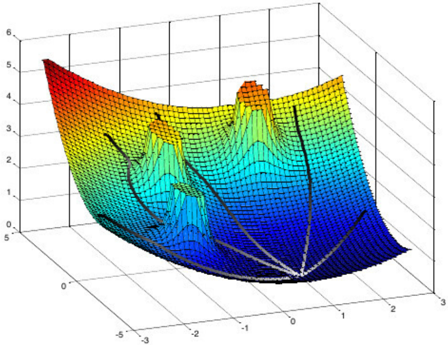
$$F_{rep}(p) = -\nabla U_{rep}(p)$$





Example of motion in the potential field





Attraction potential

the attraction potential has the following features:

- it must be a monotone function that increases with the distance from the goal
- as a consequence, it is non-null everywhere but in the goal point

one of the most trivial function having such features increases quadratically with the distance from the goal:

$$U_{att}(p) = \frac{1}{2} k_{att} d^2(p, p_{goal})$$

Attraction potential

the gradient of the attraction potential is

$$\nabla U_{att}(p) = \nabla \left(\frac{1}{2} k_{att} d^2(p, p_{goal}) \right)$$
$$= \frac{1}{2} k_{att} \nabla d^2(p, p_{goal})$$
$$= k_{att}(p - p_{goal})$$

- the gradient converges to 0
- can become arbitrary large if p is far from p_{goal}
- thresholds can be introduced on the distance to limit its value

Repulsion potential

the repulsion potential can be defined as follows:

$$U_{rep}(p) = \begin{cases} \frac{1}{2} k_{rep} \left(\frac{1}{D(p)} - \frac{1}{P^*} \right)^2, & \text{if } D(p) \leq P^* \\ 0, & \text{if } D(p) > P^* \end{cases}$$

where:

- $D(p)$ is the distance of p from the closest point q of the closest obstacle
- P^* is the threshold value that allows to discard obstacles that are too far

its gradient is

$$\nabla U_{rep}(p) = \begin{cases} k_{rep} \left(\frac{1}{P^*} - \frac{1}{D(p)} \right) \frac{(p-q)}{D^3(p)}, & \text{if } D(p) \leq P^* \\ 0, & \text{if } D(p) > P^* \end{cases}$$

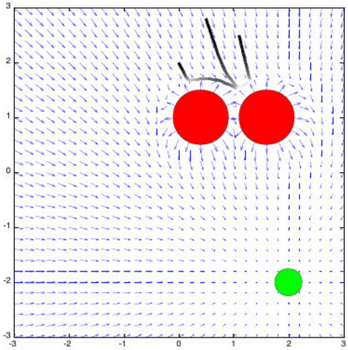
The gradient descent

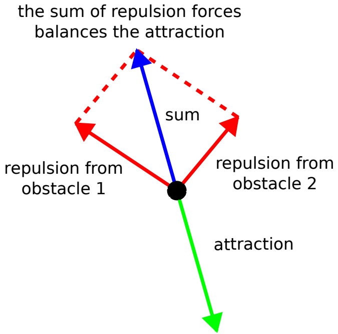
$p(0) = p_{start}$
while $|\nabla U(p(i))| > \epsilon$ **do**
  $p(i+1) = p(i) + \alpha \nabla U(p(i))$
  $i = i + 1$
end while

where

- $p(i)$ is the sequence of locations generated by the algorithm
- α is the motion step; while it should not be too large to avoid "jumping inside" an obstacle, it should not be too short to limit the execution time
- ϵ is the precision required to match the goal

The local minima problem





- The moving point gets stuck due to the balance of attraction and repulsion forces
- The point can get stuck even if there is a passage between the obstacles