Robot Motion Control
Matteo Matteucci – matteo.matteucci@polimi.it
Open loop control

A mobile robot is meant to move from one place to another

- Pre-compute a smooth trajectory based on motion segments (e.g., line and circle segments) from start to goal
- Execute the planned trajectory along the way till the goal

Disadvantages:

- Not an easy task to pre-compute a feasible trajectory
- Limitations and constraints of the robots velocities and accelerations
- Does not handle dynamical changes in the environment
- Resulting trajectories are usually not smooth
Feedback control (diff drive example)

With feedback control the trajectory is recomputed and adapted online.

We can design a simple control schema for path following:

- First we close a speed control loop on the wheels.
- Then divide the problem in:
  - Control of the orientation
  - Control of the distance

Control orientation acting on angular velocity
Control distance acting on linear velocity
Feedback control (diff drive example)

With feedback control the trajectory is recomputed and adapted online.

We can design a simple control schema for path following:

- First we close a speed control loop on the wheels.
- Then divide the problem in:
  - Control of the orientation
  - Control of the distance

Control orientation acting on angular velocity
Control distance acting on linear velocity

A simple logic handles the next point.
Feedback control (diff drive example)

Planned Trajectory

Next Point Sequencing

Desired path

Inverse kinematics

Direct kinematics (Odometry)

What about obstacles?
Who does provide the trajectory?
Do you trust direct kinematics?
A Two Layered Approach

Trajectory Planning

Lower Frequency

Trajectory Following
(and Obstacle Avoidance)

Higher Frequency

Motion Commands

Current Position

Goal Position

Map

Sensors

Local planner
Obstacle Avoidance (Local Path Planning)

Obstacle avoidance should:

- Follow the planned path
- Avoid unexpected obstacle, i.e., those that were not in the map

Several proposed methods in the literature:

- Potential field methods [Borenstein, 1989]
- Nearness diagram [Minguez & Montano, 2000]
- Curvature-Velocity [Simmons, 1996]
- Dynamic Window Approach [Fox, Burgard, Thrun, 1997]
- …
“Bugs” have little if any knowledge …
  • known direction to the goal
  • only local sensing (walls/obstacles + encoders)
… and their world is reasonable!
  • finite obstacles in any finite range
  • a line intersects an obstacle finite times

Switch between two basic behaviors
  1. head toward goal
  2. follow obstacles until you can head toward the goal again
Each obstacle is fully circled before it is left at the point closest to the goals

- **Advantages**
  - No global map required
  - Completeness guaranteed

- **Disadvantages**
  - Solution are often highly suboptimal
Use a local map of the environment and evaluate the angle to drive towards

- Environment represented in a grid (2 DOF) with
- The steering direction is computed in two steps:
  - all openings for the robot to pass are found
  - the one with lowest cost function $G$ is selected

$$G = a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_direction}$$

- target\_direction = alignment of the robot path with the goal
- wheel\_orientation = difference between the new direction and the current wheel orientation
- previous\_direction = difference between the previously selected direction and the new direction
CVMs add physical constraints from the robot and the environment on \( (v, w) \):
- Assumption that robot is traveling on arcs \( (c = \frac{w}{v}) \) with acceleration constraints
- Obstacles are transformed in velocity space
- An objective function to select the optimal speed

Simmons et al. [1996]
Vector Field Histogram+ (VFH+) \cite{Borenstein et al. 1998}

VFH+ accounts also in a very simplified way for vehicle kinematics
- robot moving on arcs or straight lines
- obstacles blocking a given direction also blocks all the trajectories (arcs) going through this direction like in an Ackerman vehicle
- obstacles are enlarged so that all kinematically blocked trajectories are properly taken into account

However VFH+ as VHF suffers
- Limitation if narrow areas (e.g. doors) have to be passed
- Local minima might not be avoided
- Reaching of the goal can not be guaranteed
- Dynamics of the robot not really considered
The kinematics of the robot are considered via local search in velocity space:

- Consider only circular trajectories determined by pairs $V_s=(v,\omega)$ of translational and rotational speeds.
- A pair $V_a=(v, \omega)$ is considered admissible, if the robot is able to stop before it reaches the closest obstacle on the corresponding curvature.
- A dynamic window restricts the reachable velocities $V_d$ to those that can be reached within a short time given limited robot accelerations.

$$V_d = \left\{ \begin{array}{l}
  v \in [v - a_{tr} \cdot t, v + a_{tr} \cdot t] \\
  \omega \in [\omega - a_{rot} \cdot t, \omega + a_{rot} \cdot t]
\end{array} \right\}$$

DWA Search Space

$$V_r = V_s \cap V_a \cap V_d$$
How to choose \((v, \omega)\)?

Steering commands are chosen maximizing a heuristic navigation function:

- Minimize the travel time by “driving fast in the right direction”
- Planning restricted to \(V_r\) space [Fox, Burgard, Thrun ‘97]

\[
G(v, \omega) = \sigma(\alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{dist}(v, \omega) + \gamma \cdot \text{velocity}(v, \omega))
\]

- Global approach [Brock & Khatib 99] in \(<x, y>\)-space uses

\[
NF = \alpha \cdot \text{vel} + \beta \cdot nf + \gamma \Delta nf + \delta \text{goal}
\]

Forward robot velocity

Follows global path

Cost to reach the goal

Goal nearness
The basic idea of the Dynamic Window Approach (DWA) algorithm follows ... 

1. Discretely sample robot control space
2. For each sampled velocity, perform forward simulation from current state to predict what would happen if applied for some (short) time.
3. Evaluate (score) each trajectory resulting from the forward simulation
4. Discard illegal trajectories, i.e., those that collide with obstacles, and pick the highest-scoring trajectory

What about non circular kinematics?

**Clothoid:** \[ S(x) = \int_0^x \sin(t^2) \, dt, \quad C(x) = \int_0^x \cos(t^2) \, dt. \]
A Two Layered Approach

Global planner

Trajectory Planning

Goal Position

Lower Frequency

Trajectory

Current Position

Higher Frequency

Trajectory Following
(and Obstacle Avoidance)

Motion Commands

Map

Sensors
“...eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world.”

J.-C. Latombe (1991)

Robot Motion Planning Goals
- Collision-free trajectories
- Robot should reach the goal location as fast as possible

Information available
- Map with obstacles
- Robot shape and kinematics
Different possible maps representations exist for path planning

- Paths (e.g., probabilistic road maps)
- Free space (e.g., Voronoi diagrams)
- Obstacles (e.g., geometric obstacles)
- Composite (e.g., grid maps)
What a Planner?

Search Based Planning Algorithms

- A*
- ARA*
- ANA*
- AD*
- D*
- ...

Random Sampling

- PRMs
- RRT
- T-RRT
- SBL
- ...

Search Based Planning Library

Open Motion Planning Library
### Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Based Planning</td>
<td>• Finds the optimal solution • Possible to assign costs • Use of Heuristics • Can state if a solution exists (complete)</td>
<td>• High computational cost</td>
</tr>
<tr>
<td>Random Sampling Planning</td>
<td>• Fast in finding a feasible solution</td>
<td>• Hard to assign costs • Only probably complete (cannot be used to test for existence)</td>
</tr>
</tbody>
</table>

Let's have a look at Search Based Methods (SBPL) because of

- Their simplicity (at least in description)
- The generality of approaches
- Their theoretical guarantees (if connectivity assumptions hold)
Different possible maps representations exist for path planning

- Paths (e.g., probabilistic road maps)
- Free space (e.g., Voronoi diagrams)
- Obstacles (e.g., geometric obstacles)
- Composite (e.g., grid maps)

Kinematics approximation in grid maps

- 4-orthogonal connectivity
- 4-diagonal connectivity
- 8-connectivity
The overall idea:

- Generate a discretized representation of the planning problem
- Build a graph out of this discretized representation (e.g., through 4 neighbors or 8 neighbors connectivity)
- Search the graph for the optimal solution

- Can interleave the construction of the representation with the search (i.e., construct only what is necessary)
A real mobile robot should not be modeled as a point; to take into account its shape obstacles are enlarged.

This might generate some issues and a trade-off is between memory requirements and performance.
For non-circular robots, collisions might depend on the orientation.

The C-Space is used to speed up collision detection:

- A configuration of an object is a point \( q = (q_1, q_2, \ldots, q_n) \).
- Point \( q \) is free if the robot in \( q \) does not collide.
- \( C\)-obstacle = union of all \( q \) where the robot collides.
- \( C\)-free = union of all free \( q \).
- \( C\)space = \( C\)-free + \( C\)-obstacle.

Planning can be performed in C-Space.
Mobile robots C-Space

A robot can translate in the plane and/or rotate

Obstacles should be expanded according to the robot orientation
Different algorithms are available

- Returning the optimal path (e.g., Dijkstra, A*, …)
- Returning an $\epsilon$ sub-optimal path (e.g., weighted A*, ARA*, AD*, R*, D* Lite, …)
Searching Graphs for Least Cost Path

Given a graph search for the path that minimizes costs as much as possible

Many search algorithms compute optimal $g$-values for relevant states

- $g(s)$–an estimate of the cost of a least-cost path from $s_{\text{start}}$ to $s$
- optimal values satisfy: $g(s) = \min_{s'' \in \text{pred}(s)} g(s'') + c(s'',s)$

Least-cost path is a greedy path computed by backtracking:

- start with $s_{\text{goal}}$ and from any state $s$ move to the predecessor state $s'$ such that

$$s' = \arg\min_{s'' \in \text{pred}(s)} (g(s'') + c(s'',s))$$
A* Search

A* speeds up search by computing g-values for relevant states as

\[ g(s) \]

Heuristic function must be

- admissible: for every state \( s \), \( h(s) \leq c^*(s,s_{goal}) \)
- consistent (satisfy triangle inequality):
  - \( h(s_{goal},s_{goal}) = 0 \)
  - for every \( s \neq s_{goal} \), \( h(s) \leq c(s,succ(s)) + h(succ(s)) \)

Admissibility follows from consistency and often consistency follows from admissibility
A* Search Algorithm

Main function

- \( g(s_{\text{start}}) = 0 \); all other \( g \)-values are infinite;
- \( OPEN = \{ s_{\text{start}} \} \);
- ComputePath();

ComputePath function

- while \( s_{\text{goal}} \) is not expanded
  - remove \( s \) with the smallest \( f(s) = g(s) + h(s) \) from \( OPEN \);
  - expand \( s \);

Set of candidates for expansion

For every expanded state \( g(s) \) is optimal (if heuristics are consistent)
A* Search Algorithm

Main function
- $g(s_{start}) = 0$; all other $g$-values are infinite;
- $OPEN = \{s_{start}\}$;
- ComputePath();

ComputePath function
- while($s_{goal}$ is not expanded)
  - remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  - insert $s$ into $CLOSED$;
  - for every successor $s'$ of $s$ such that $s'$ not in $CLOSED$
    - if $g(s') > g(s) + c(s,s')$
      - $g(s') = g(s) + c(s,s')$;
      - insert $s'$ into $OPEN$;

Set of candidates for expansion
Set of states already expanded

Tries to decrease $g(s')$ using the found path from $s_{start}$ to $s$
A* Search Algorithm

**ComputePath function**

- while \( s_{\text{goal}} \) is not expanded
  - remove \( s \) with the smallest \([f(s) = g(s) + h(s)]\) from OPEN;
  - insert \( s \) into CLOSED;
  - for every successor \( s' \) of \( s \) such that \( s' \) not in CLOSED
    - if \( g(s') > g(s) + c(s, s') \)
      - \( g(s') = g(s) + c(s, s') \);
      - insert \( s' \) into OPEN;

\[
\begin{align*}
\text{CLOSED} &= \{\} \\
\text{OPEN} &= \{s_{\text{start}}\} \\
\text{next state to expand: } s_{\text{start}}
\end{align*}
\]
A* Search Algorithm

ComputePath function

• while ($s_{goal}$ is not expanded)
  • remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from OPEN;
  • insert $s$ into CLOSED;
  • for every successor $s'$ of $s$ such that $s'$ not in CLOSED
    • if $g(s') > g(s) + c(s,s')$
      • $g(s') = g(s) + c(s,s')$;
      • insert $s'$ into OPEN;

$CLOSED = \{s_{start}\}$

$OPEN = \{s_2\}$

next state to expand: $s_2$
A* Search Algorithm

ComputePath function

- while($s_{goal}$ is not expanded)
  - remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from OPEN;
  - insert $s$ into CLOSED;
  - for every successor $s'$ of $s$ such that $s'$ not in CLOSED
    - if $g(s') > g(s) + c(s,s')$
      - $g(s') = g(s) + c(s,s')$;
      - insert $s'$ into OPEN;

$CLOSED = \{s_{start}, s_2\}$

$OPEN = \{s_1, s_4\}$

next state to expand: $s_1$
A* Search Algorithm

**ComputePath function**

- while \( s_{goal} \) is not expanded
  - remove \( s \) with the smallest \( f(s) = g(s) + h(s) \) from OPEN;
  - insert \( s \) into CLOSED;
  - for every successor \( s' \) of \( s \) such that \( s' \) not in CLOSED
    - if \( g(s') > g(s) + c(s,s') \)
      - \( g(s') = g(s) + c(s,s') \);
      - insert \( s' \) into OPEN;

\[
\begin{align*}
CLOS\text{ED} &= \{s_{start}, s_2, s_1\} \\
OPEN &= \{s_4, s_{goal}\}
\end{align*}
\]

next state to expand: \( s_4 \)
A* Search Algorithm

ComputePath function

- while \( s_{goal} \) is not expanded
  - remove \( s \) with the smallest \( f(s) = g(s) + h(s) \) from OPEN;
  - insert \( s \) into CLOSED;
  - for every successor \( s' \) of \( s \) such that \( s' \) not in CLOSED
    - if \( g(s') > g(s) + c(s,s') \)
      - \( g(s') = g(s) + c(s,s') \);
      - insert \( s' \) into OPEN;

\[
\begin{align*}
\text{CLOSED} &= \{s_{start}, s_2, s_1, s_4\} \\
\text{OPEN} &= \{s_3, s_{goal}\}
\end{align*}
\]

next state to expand: \( s_{goal} \)
A* Search Algorithm

ComputePath function

- while ($s_{goal}$ is not expanded)
  - remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from OPEN;
  - insert $s$ into CLOSED;
  - for every successor $s'$ of $s$ such that $s'$ not in CLOSED
    - if $g(s') > g(s) + c(s,s')$
      - $g(s') = g(s) + c(s,s')$;
      - insert $s'$ into OPEN;

$CLOSED = \{S_{start}, S_2, S_1, S_4, S_{goal}\}$
$OPEN = \{S_3\}$

DONE!
A* Properties

A* is guaranteed to
- return an optimal path in terms of the solution
- perform provably minimal number of state expansions

Algorithms state expansion:
- Dijkstra’s: expands states in the order of \( f = g \) values (roughly)
- A* Search: expands states in the order of \( f = g + h \) values
- Weighted A*: expands states in the order of \( f = g + \varepsilon \) \( h \) values, \( \varepsilon > 1 \) = bias towards states that are closer to goal

Weighted A* Search in many domains, it has been shown to be orders of magnitude faster than A*
Algorithms state expansion:

- Dijkstra’s: expands states in the order of $f = g$ values (roughly)
Algorithms state expansion:

- Dijkstra’s: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values
A* Properties

Algorithms state expansion:

- Dijkstra’s: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values
- Weighted A*: expands states in the order of $f = g + \varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal

key to finding solution fast: shallow minima
Planning Problem Ingredients

Typical components of a Search-based Planner

- Graph search algorithm (for example, A* search)
- Graph construction (given a state what are its successor states)
- Cost function (a cost associated with every transition in the graph)
- Heuristic function (estimates of cost-to-goal)

The graph can take into account robot dynamics/kinematics constraints

- Domain Independent
- Domain Dependent
Lattice Based Graphs for Navigation

Graph can be constructed by using motion primitives

- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness
Lattice Based Graphs for Navigation

Graph can be constructed by using motion primitives
- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness
A Two Layered Approach

Global planner

Trajectory Planning

Lower Frequency

Trajectory Following (and Obstacle Avoidance)

Higher Frequency

Motion Commands

Where from?

Map

Sensors

Goal Position

Current Position
Cognitive Robotics – Robot Motion Planning

Matteo Matteucci – matteo.matteucci@polimi.it