Ch. 10 Motion Planning 26.5.23

Finding robot motion from a start state to a goal state, avoiding obstacles and satisfy other constraints.

Overview

Key concept: · configuration space (C-space) C

· free C-space C_{free}: robot neither penetrates an obstacle nor violates joint limits

· config of a robot can be represented as list of joint positions $q = (\theta_1, \ldots, \theta_n)$ (n: number of joints)
↳ $q \in \mathbb{R}^n$, $C \subset \mathbb{R}^n$
· concepts could be generalized to apply to other (non-Euclidian) C-spaces e.g. $C = SE(3)$

· control inputs for robot (driving the robot): $u \in U \subset \mathbb{R}^m$ (m: number of control inputs)

· for typical robot arm: $m = n$

· if robot has second-order dynamics (e.g. robot arm) and control inputs are forces ($\hat{=}$ accelerations):
state of robot: $x = (q, v) \in X$ (q: configuration, v: velocity)
- for $q \in \mathbb{R}^n$: typically $v = \dot{q}$

· if control inputs are velocities:
· state of robot: $x = q$

· $q(x)$: configuration q corresponding to state x
$X_{free} = \{x \mid q(x) \in C_{free}\}$

· equation of motion of robot:

$$\dot{x} = f(x, u)$$

or (integral form): (state, control input)

$$x(T) = x(0) + \int_0^T f(x(t), u(t))\,dt$$

Types of Motion Planning Problems

Def:

Given initial state $x(0) = x_{start}$ and final state x_{goal}

find a time T and a set of controls $u: [0,T] \to U$

such that motion $x(T) = x(0) \int_0^T f(x(t), u(t))\, dt$

satisfies $x(T) = x_{goal}$ and $q(x(t)) \in C_{free}$ for all $t \in [0,T]$.

A feedback controller is assumed to ensure the planned motion $x(t)$, $t \in [0,T]$ is followed closely

Also an accurate model of the robot and the environment is assumed to evaluate C_{free} during motion planning

Path planning vs. motion planning:

- Path planning is a subproblem of motion planning
- Path planning:
  - purely geometric problem
  - finding a collision-free path $q(s)$, $s \in [0,1]$ from start $q(0) = q_{start}$ to goal $q(1) = q_{goal}$
  - without concerns for dynamics, duration or constraints
  - it's assumed that the path returned by the path planner can be time-scaled to create feasible trajectory (piano mover's problem)

Control inputs: $m = n$ vs. $m < n$

- if $m < n$: robot can not follow all path even if they are collision free
  - e.g. car (Ackerman-steering) cannot slide sideways

Online vs offline:

- immediate results are required in dynamic environments (online)
- static environments: slower (offline) planners may suffice

Optimal vs satisficing:

- reducing cost to reach goal
- minimizing (or approximately minimizing) a cost

e.g. $J = \int_0^T L(x(t), u(t))\, dt$

- minimizing for $L = 1$: time-optimal motion
- minimizing for $L = u^T(t) u(t)$: "minimal effort"

Types of Motion Planning Problems (cont.)

26.5.23

Exact vs. approximate:

- approximate final state $x(T)$ sufficiently close to x_{goal} e.g: $\|x(T) - x_{goal}\| < \varepsilon$
- or $x_{goal} \doteq x(T)$: needs to exactly reach goal state

With or without obstacles:

- motion planning can be challenging even without obstacles particularly if $m < n$ or if optimality is required

Properties of Motion Planners

Multiple-query vs single-query planner

- in an unchanging environment (multiple-query):
  - build data structure that accurately represents C_{free}
  - search data structure to solve multiple planning queries
- Single-query: solve each new problem from scratch (ad-hoc)

„Anytime" planning

- search for better solutions after first solution is found
- stop when given criteria is reached (e.g. after given time)

Completeness:

- complete: . guaranteed to find a solution (in finite time) if one exists
  . report failure if no solution exists
- resolution complete: . weaker than 'complete'
  . guaranteed to find solution (if it exists) at the resolution of a discretized representation of problem
  . e.g. resolution of grid representation of C_{free}
- probabilistically complete: if probability of finding solution (if it exists) tends to 1 as planning time $t \to \infty$ goes to infinity

Computational complexity:

- amount of time or memory needed
- big-O notation
- average case or worst case

Motion Planning Methods

27.5.23

Complete Methods:

- exact representation of geometry/topology of C_{free}
- ensuring completeness
- only for simple or low-degree-of-freedom problems

Grid Methods:

- discretize C_{free} into grid
- search grid from q_{start} to goal region
- may use multi-scale grid for refinement of C_{free} near obstacles
- relatively easy to implement
- can return optimal solutions
- memory required to store grid and time to search for solution grow exponentially with number of dimensions of the space
- only for low-dimensional problems

Sampling Methods:

- choose samples from C-space or state space
  - random or
  - deterministic
- evaluate whether sample is in X_{free}
- determine "closest" previous free-space sample
- local planner: try to connect or move towards new sample
- easy to implement
- tend to be probabilistically complete
- can solve high-degree-of-freedom motion planning problems
- tend to be satisficing, not optimal
- difficult to characterize computational complexity

Motion Planning Methods (cont.)

27.5.23

Virtual potential fields:
- create forces on the robot that
  - pull it toward goal
  - push it away from obstacles
- relatively easy to implement
- suitable for high-degree of-freedom systems
- fast to evaluating, often allowing online implementation
- drawback: robot may get stuck in local minima of potential function

Nonlinear optimization:
- convert motion planning problem to nonlinear optimization problem
- representing path or control by a finite numbers of parameters such as coefficients of a polynomial or a Fourier series
- solve for parameters that minimize a cost function
- constraints on controls, obstacles and goal need to be satisfied
- can produce near-optimal solutions
- require initial guess at solution
- can get stuck far away from a feasible solution

Smoothing:
A smoothing algorithm can be run on the solution of a motion planner to improve smoothness (less jerky motions)

A major trend has been towards sampling methods. They are easy to implement and can handle high-dimensional problems.

Configuration Space Obstacles 29.5.23

Workspace obstacles partition configuration space C into freespace C_{free} and obstacle space C_{obs}

$$C = C_{free} \cup C_{obs}$$

Joint limits are treated as obstacles in the configuration space.

Planning Problem: finding path for a point-robot among obstacles C_{obs}.
If obstacles break C_{free} into separate connected components, and q_{start} and q_{goal} do not lie in same connected component: there is no collision-free path

Distance to Obstacles and Collision Detection

Given obstacle B and configuration q:
let $d(q, B)$ be distance between robot and obstacle:

- $d(q, B) > 0$: no contact with obstacle
- $d(q, B) = 0$: contact
- $d(q, B) < 0$: penetration

Distance $d(q, B)$ could be Euclidian distance.
- distance-measurement algorithm: determine $d(q, B)$
- collision-detection routine: determine whether $d(q, B) \leq 0$ for any C-obstacle B_i

One popular distance-measurement algorithm is the Gilbert-Johnson-Keerthi (GJK) algorithm.

A simpler approach is to approximate robot and obstacles as unions of overlapping spheres. Approximations must be conservative.

Graphs and Trees

29.5.23

Many motion planners represent the C-space or the state-space as a graph.

A graph:
- N: nodes
- E: edges (each edge connects two nodes)

A graph can be
- directed (digraph)
- undirected
- weighted
- unweighted

Tree:
- graph with no cycles
- each node at most one parent
- a root (with no parent)
- leafes (with no children)

Graph search : A* search

2.6.23

Free space (free represented as graph.
Searching from start to goal.
Find minimum-cost path. Cost: sum of positive edge costs along path.
Given: • set of nodes $\mathcal{N} = \{1, \dots, N\}$, node 1 is start
• set of edges $\mathcal{E}$ (between nodes)

Data structures:
- OPEN list · sorted by estimated cost
  · nodes that still need to be explored
- CLOSED list: · nods that have been fully explored
- cost[node1, node2]: matrix with encoding of cost to move from one node (node1) to another (node2)
  → negative value : no edge
- past_cost[node]: minimum cost path so far found from start to node
- parent[node]: parent for shortest path from node
- heuristic_cost_to_go[node]: optimistic (underestimating) estimate of the 'distance' from node to goal

optimistic straight line distance to goal

OPEN ← {1}
past_cost[1] ← 0, past_cost[node] ← ∞ for node ∈ {2, ..., N}
while OPEN not empty do:
  current ← first in OPEN, remove from OPEN
  add current to closed
  if current in goal return path to current (calculated with parent[node])
  endif
  for each nbr in current.neighbors not in CLOSED:
    tentative_past_cost ← past_cost[current] + cost[current, nbr]
    if tentative_past_cost < past_cost[nbr] then
      past_cost[nbr] ← tentative_past_cost
      parent[nbr] ← current
      put (or move) nbr in OPEN sorted by est_total_cost[nbr] = past_cost[nbr] + heuristic_to_go(nbr)
    endif
  endif
endwhile
return FAILURE

10.3 Complete Path Planners

9.6.23

Needed: exact representation of the free c-space (C_{free})
Mathematically and algorithmically sophisticated.
But impractical for many real systems.

One aproach:

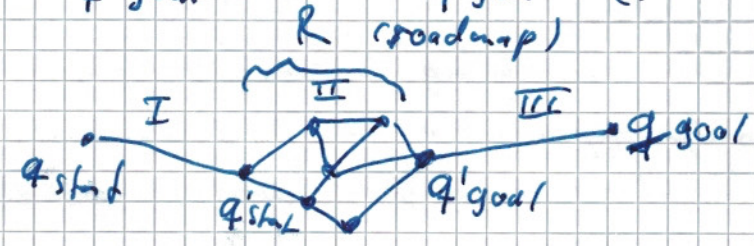
Represent complex, high-dimensional space C_{free} by a 1-dimensional <u>roadmap</u> R with properties

a) Reachability: from every point $q \in C_{free}$ to a point $q' \in R$ can be found trivially (e.g. straight-line path)

b) Connectivity: for each connected component of C_{free} there is one connected component of R

Planner can find a path between q_{start} and q_{goal} (in the same connected component of C_{free}) by:
finding paths:

I. from q_{start} to $q'_{start} \in R$ (from start to roadmap)
II. from $q'_{start} \in R$ to $q'_{goal} \in R$ (on roadmap)
III. from $q'_{goal} \in R$ to q_{goal} (from roadmap to goal)



If a path can be found trivially between q_{start} and q_{goal} the roadmap may not even be used.

Roadmap: Suitable roadmap: weighted undirected <u>visibility graph</u>:

- nodes at vertices of obstacles
- edges between nodes that can "see" each other
  - weight: euclidian distance

10.4. Grid Methods

Algorithms like A* require a <u>discretization</u> of the search space.
Simplest discretization: grid
If C-space n-dimensional and k grid points required along each dimension:

C-space is represented by k^n grid points

A* for a C-space grid:

- definition of "neighbor" of a grid point:
  - axis aligned (Manhattan distance)
  - multiple dimensions simultaneously
  - to 2-dimensional C-space:
    - 4-connected (north, south, east, west)
    - 8-connected (also diagonals allowed)
- heuristic cost-to-go:
  - Manhattan distance for axis aligned
  - Euclidian distance for multiple dimensions simultaneously
- other optimizations / constraints could be added

An A* grid-based path planner is <u>resolution complete</u>:
it will find a solution (if one exist) at level of discretization of the C-space.

Grid-based path planning is easy to implement.
Bad it's only appropriate for low-dimensional C-spaces
Computational complexity increases exponentially with number of dimensions n.

10.4.1 Multi-Resolution Grid Representations

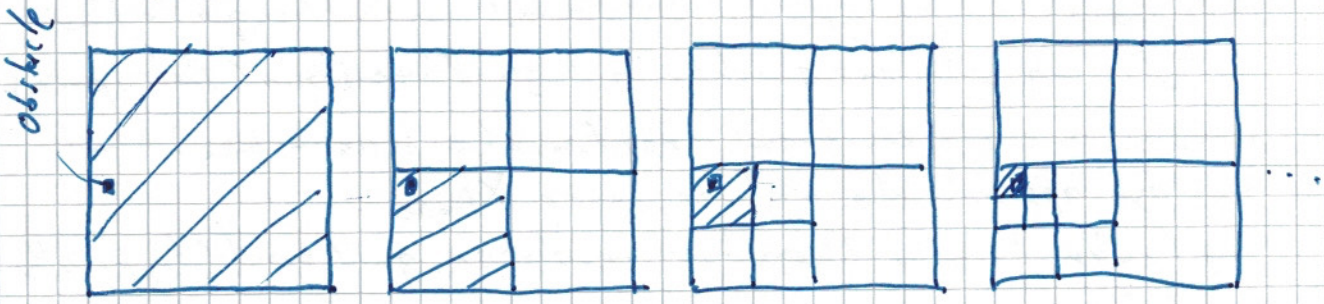
9.6.23

One way to reduce the computational complexity of a grid-based planner:

Multi resolution grid representation of C_{free}

A grid-point is considered an obstacle if the cell centered at the grid-point touches an C-obstacle. Obstacle cells are subdivided into smaller cells. Any cells still in contact with an obstacle are subdivided further up to a max resolution.

This allows the planner to find paths through cluttered environments.



10.4.2. Grid Methods with Motion Constraints

Some robots might not be able to go from one cell to any neighboring cell in a regular C-space grid.

e.g. car cannot park sideways

Also a fast moving robot arm should be planned in the state space not just in the C-space to take <u>dynamics</u> into account. In the state space the arm cannot move in certain directions.

Constraint may result in a <u>directed</u> graph.

One aproach:

- discretize robot control's
- still make use of a grid on the C-space or state space

Sampling Methods: - optimal solutions subject to chosen discretization
- but high computational complexity
  - only suitable for systems with few degrees of freedom

10.5 Sampling Methods

Sampling methods rely on:
- random or deterministic function to choose sample from C-space or state space
- function to evaluate if a sample or motion is in X_{free}
- function to determine near previous free-space samples
- simple local planner to try to connect (or move to) the new sample

These functions are used to build a graph or tree to represent feasible motions of robot.

Give up resolution-optimal solutions in exchange to find satisficing solutions quickly in high-dimensional state spaces.
Samples are chosen from a roadmap (or search tree) that quickly approximate the free space X_{free}.
Uses fewer samples than high-resolution grid would typically need.
Most sampling methods are probabilistically complete:
- probability of finding a solution (if one exists) $\to 100\%$ as the number of samples $\to \infty$

Two major classes of sampling methods:

- RRTs: rapidly exploring random trees
  - tree representation
  - single query
  - C-space or state space
- PRMs: probabilistic roadmaps:
  - roadmap graph
  - multiple query
  - primarily C-space

10.5.1 The RRT Algorithm (rapidly-exploring random trees) 10.6.2

Search a collision-free motion from initial state x_{start} to goal set X_{goal}.

Can be applied to
- kinematic problems: state x is configuration q
- dynamic problems: state includes velocity

Grows a single tree from x_{start}

Algorithm:

initialize search tree T with x_{start}
while T is less than max. tree size do
① $x_{samp} \leftarrow$ sample from X
② $x_{nearest} \leftarrow$ nearest node in T to x_{samp}
③ employ local planner to find a motion from $x_{nearest}$ to x_{new} in direction of x_{samp}
if the motion is collision-free then
add x_{new} to T with edge from $x_{nearest}$ to x_{new}
if x_{new} in X_{goal} then
return SUCCESS and motion to x_{new}
endif
endif
end while
return FAILURE

The algorithm leaves the programmer with different choices:

① how to sample from X
② how to define "nearest" node in T
③ how to plan motion towards x_{samp}

① The Sampler

Most obvious:
- sampling randomly from uniform distribution over X
- Straight-forward for euclidian C-spaces $\mathbb{R}^n$ as well as for n-joint robot C-spaces $T^n = S^1 \times \dots \times S^1$ (n times)
  - ↳ choose uniform distribution over each joint angle

  as well as for a mobile robot with C-space $\mathbb{R}^2 \times S^1$ (in a plane)
  - ↳ choose uniform distribution over $\mathbb{R}^2$ and S^1 individually
- Other curved C-spaces e.g. $SO(3)$ are not so straight-forward
- dynamic systems:
  uniform distribution over state space
  - ↳ cross product of uniform distribution over C-space and uniform distribution over bounded velocity set
- sample often implemented with a bias toward states in X_{goal}
- Sampler does not need to be random
  - ↳ rapidly-exploring dense trees (RDTs)

② Defining the nearest Node

Finding the "nearest" Node depends on a definition of distance on X.

For an unconstraint kinematic robot: Euclidian distance.

For other spaces defining notion of distance:
- combining components of different units (deg, meter, deg/s, m/s ...) into a single distance measure
- taking into account motion constraint of the robot

Could compute which node is reached fastest from x_{samp}. But this can be expensive to calculate

simple variant: weighted sum of components (weight: importance)

Nearest node need to be computed fast.

Common problem in computational geometry: kd-trees, hashing

(3) The localPlanner

11.6.23

Find a motion from $x_{nearest}$ to some x_{new} that is nearer to x_{samp}.
The planner should be simple and fast.

Examples:

- Straight-line planner: straight-line with distance α towards x_{samp} and directly to x_{samp}
  suitable for kinematic systems with no motion constraints
- Discretized controls planner: discretize controls in a discrete set $\{u_1, u_2, \ldots\}$. Integrate each control from $x_{nearest}$ for fixed time Δt using $\dot{x} = f(x, u)$. Among new states choose the one closest to x_{samp} as x_{new} if it is collision-free
  Suitable for systems with motion constraints, such as wheeled mobile robots or dynamic systems
- Wheeled robot planner: local plans can be found using Reeds-Shepp curves.

Other RRT Variants

- Bidirectional RRT: · grow two trees, one from x_{start} to x_{goal} and one from x_{goal} to x_{start}
  · try to connect trees
- RRT*: "local" planner tries to minimize total cost from x_{start} to x_{new}. Uses multiple nearest neighbors.

10.5.2 The PRM Algorithm 12.6.23

Use sampling to build a roadmap (of C_{free}) then answer specific queries. Roadmap: undirected graph, robot can move in either direction from one node to another

Primarily for kinematic problems with exact local planners (e.g. straight-line planner for robot without kinematic constraints.

When map is built:

- connect q_{start} with nearest node on map
- connect q_{goal} with nearest node on map
- Search path with A* (or similar) algorithm

Sampling can be random (uniform distribution) or deterministic. (multi-resolution)
Sampling more densly around obstacles improves planning around narrow passages.

Sampling: Deterministic Methods (for different planning methods)

Examples:

- Van der Corput sampling (1d)
- Halton sequence (higher dimensions)

10.6. Virtual Potential Fields

13.6.23

Inspired by potential energy fields in nature (gravitation, magnet...)

Physics:

$P(q)$: Potential field over C

↳ force: $F = -\partial P / \partial q$

F drives an object from high to low potential

Motion control:

- goal configuration q_{goal} is assigned a low virtual potential
- obstacles are assigned a high virtual potential

↳ apply force to robot proportional to negative gradient of virtual potential ⇒ pushes robot toward goal and away from obstacles

- Gradient of the field can usually be calculated quickly
  - ↳ real time: reactive control
  - don't need to be planned in advance
- can even handle moving obstacles or obstacles that appear unexpected
  - needs appropriate sensors
- robot can get stuck in local minima of field
- in some cases the potential field can be designed to only contain the goal as a minimum

A Point in C-Space

Assuming a point robot in its C-space

Goal configuration is typically encoded as a ("quadratic") "bowl" with zero energy at the goal

$$P_{goal}(q) = \frac{1}{2} (q - q_{goal})^T K (q - q_{goal})$$

with K: weighting matrix (symmetric positive-definite) i.e. identity matrix

Force induced by this potential:

$$F_{goal}(q) = -\frac{\partial P_{goal}}{\partial q} = K (q_{goal} - q)$$

↳ attractive force proportional to the distance to the goal.

Obstacles: repulsive potential induced by c-obstacle B

calculate from distance $d(q, B)$: use appropriate distance algorithm

↳ $P_B(q) = \frac{k}{2 d^2(q, B)}$

with $k > 0$: scaling factor

$d(q, B) > 0$: otherwise potential is not properly defined

Force induced by obstacle potential:

$$F_B(q) = -\frac{\partial P_B}{\partial q} = \frac{k}{d^3(q, B)} \frac{\partial d}{\partial q}$$

Total potential:

- summing attractive goal potential and all repulsive obstacles potentials

↳ $P(q) = P_{goal}(q) + \sum_i P_{B_i}(q)$

yielding a force

$$F(q) = F_{goal}(q) + \sum_i F_{B_i}(q)$$

Note: the sum might not give a minimum (exactly zero) force at q_{goal}

Useful to put a bound on maximum potential an force!

Controlling robot with calculated $F(q)$: there are several options

- Apply calculated force and damping

$$u = F(q) - B\dot{q}$$

↳ B: positive definite dissipates energy for all $\dot{q} \neq 0$ reducing oscillation ⇒ robot will come to rest

- Treat calculated force as commanded velocity

$$\dot{q} = F(q)$$

↳ eliminates oscillation

Distant obstacles should be ignored: define range of influence of obstacles $d_{range} > 0$: $U_B(q) = \begin{cases} \frac{k}{2}\left(d_{range} - d(q, B)\right)^2 & \text{if } d(q, B) < d_{range} \\ 0 & \text{otherwise: } d(q, B) \end{cases}$

10.6.2 Navigation Functions

15.6.23

One method to avoid local minima in the potential field is the wavefront planner.

It creates a local-minimum-free potential function of every cell reachable from the goal (grid representation of C_{free}).
Uses breadth-first traversal of cells.
Moving „downhill" at every step garantees to bring the robot to the goal.

Another method to avoid local minima: navigation function
A navigation function $\varphi(q)$ is a type of virtual potential field that:

1. is smooth on q (at least twice differentiable)
2. has a bounded max. value (e.g. 1) on boundaries of obstacles
3. has a single minimum at q_{goal}
4. has full-rank Hessian $\frac{\partial^2 \varphi}{\partial q^2}$ at all critical points q where $\frac{\partial \varphi}{\partial q} = 0$

⇒ $\varphi(q)$ is a Morse function

Condition 1. guarantees that the Hessian $\frac{\partial^2 \varphi}{\partial q^2}$ exists (4.)
Condition 2. puts an upper bound on robots virtual potential energy
Condition 3. ensures there is only one minimum
Condition 4. ensures that almost every initial state converges to the unique minimum q_{goal}

Constructing a navigation potential function with only one minimum is nontrivial.

See book for method using sphere world / star world for navigation functions.

10.6.3 Workspace Potential

Difficult to obtain distance to obstacle $d(q, B)$ for calculating repulsive force.

One approach to avoid exact calculation:

- represent boundary of an obstacle as set of point obstacles
- represent robot as small set of control points

- Cartesian location of control point i on robot: $f_i(q) \in \mathbb{R}^3$
- Boundary point j of the obstacle be: $c_j \in \mathbb{R}^3$

↳ distance between the two points:

$$\| f_i(q) - c_j \|$$

potential at control point i due to obstacle point j:

$$P_{ij}^t(q) = \frac{k}{2\| f_i(q) - c_j \|^2}$$

yielding repulsive force at control point:

$$F_{ij}^t(q) = \frac{\partial P_{ij}^t(q)}{\partial q} = \frac{k}{\| f_i(q) - c_j \|^4} \left(\frac{\partial f_i}{\partial q} \right)^T (f_i(q) - c_j) \in \mathbb{R}^3$$

↳ Turn linear force $F_{ij}^t(q) \in \mathbb{R}^3$ into generalized force $F_{ij}(q) \in \mathbb{R}^4$ acting on robot arm on the mobile robot:

- find Jacobian $J_i(q) \in \mathbb{R}^{3 \times 4}$
- relating $\dot{q}$ to linear velocity of control point $\dot{f}_i$:

$$\dot{f}_i = \frac{\partial f_i}{\partial q} \dot{q} = J_i(q) \cdot \dot{q}$$

Generalized force $F_{ij}(q) \in \mathbb{R}^4$ due to repulsive linear force $F_{ij}^t(q) \in \mathbb{R}^3$:

$$F_{ij}(q) = J_i^T(q) \cdot F_{ij}^t(q)$$

↳ principle of virtual work

The total force acting on robot:

- sum of attractive force $F_{goal}(q)$ and repulsive forces $F_{ij}(q)$ for all i and j.

10.7. Nonlinear Optimization (for Motion Planning) 16.6.23

Different approach to motion planning

Goal: find: $u(t)$, $q(t)$, T

minimizing: $J(u(t), q(t), T)$

subject to: $\dot{x}(t) = f(x(t), u(t))$ — dynamic equations need to be satisfied for all t

$u(t) \in \mathcal{U}$: controls are feasible

$q(t) \in \mathcal{C}_{free}$: motion is collision free

$x(0) = x_{start}$

$x(T) = x_{goal}$ } start at start state, end at goal

constraints

$u(t)$: control history

$q(t)$: trajectory

T: trajectory duration

J: cost functions to minimize (like energy consumed, duration of motion...)

Control history needs an initial guess and finite parameter representation of the controll:

- set points interpolated by polynomials
- splines
- piecewise constant or linear controls
- Fourier series (truncated)

...

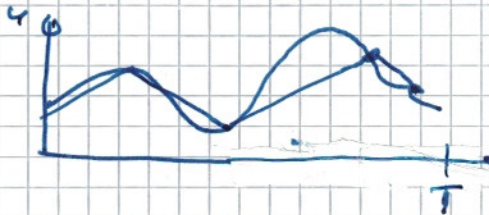
(can get stuck in local minima)

Shooting

- Piecewise linear controls
- Integrating equations of motion → robots trajectory
- The trajectory might not end at goal configuration or intersect obstacles
- To evaluate constraints due to obsacles:
  - choose finite set of test point alon the trajectory
  - evaluate if test points are collision-free
- Update guess at controll history:
  - calculate gradients of constraints wrt the trajectory and gradient for cost wrt controls
  - move test points to satisfy constraints
  - update control history, integrate control to get updated trajectory
- repeat

$\frac{\partial \text{constraints}}{\partial \text{trajectory}} \Rightarrow \frac{\partial \text{trajectory}}{\partial \text{controls}}$

$\Rightarrow$ modify testpoints



other nonlinear methods: collocation, transcription...