School of Mechanical Engineering

FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

MECH5170M
Connected and Autonomous Vehicles Systems

Path planning for AV

Kris Kubiak (k.kubiak@leeds.ac.uk)

Outline

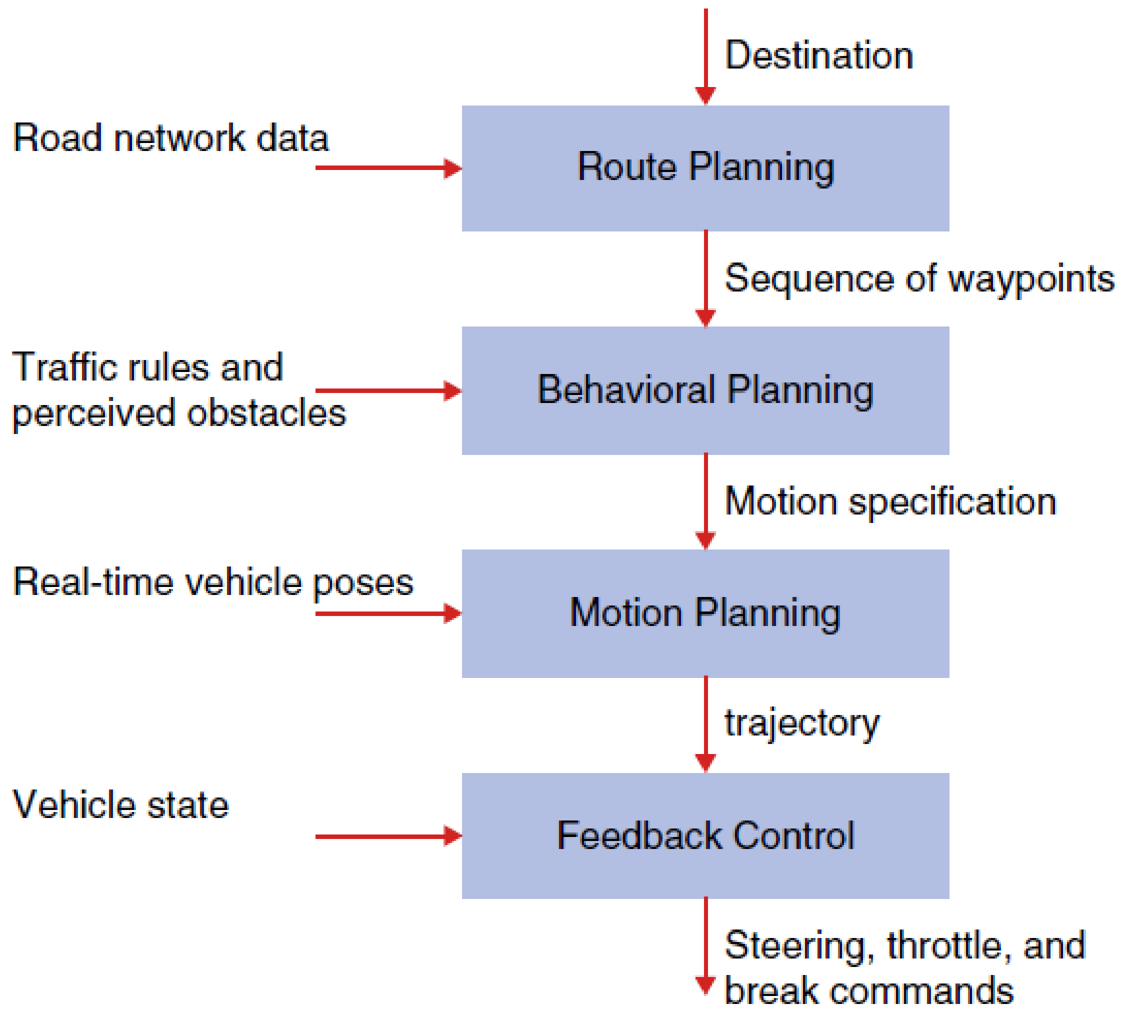
- Path Planning
- Route Planning algorithms
- Behavioural Planning
- Motion Planning (RTT)
- Feedback control

Path Planning

Planning and control module architecture

Path Planning

- Route Planning
- Behavioural Planning
- Motion Planning (RTT)
- Feedback control



Planning and control module architecture

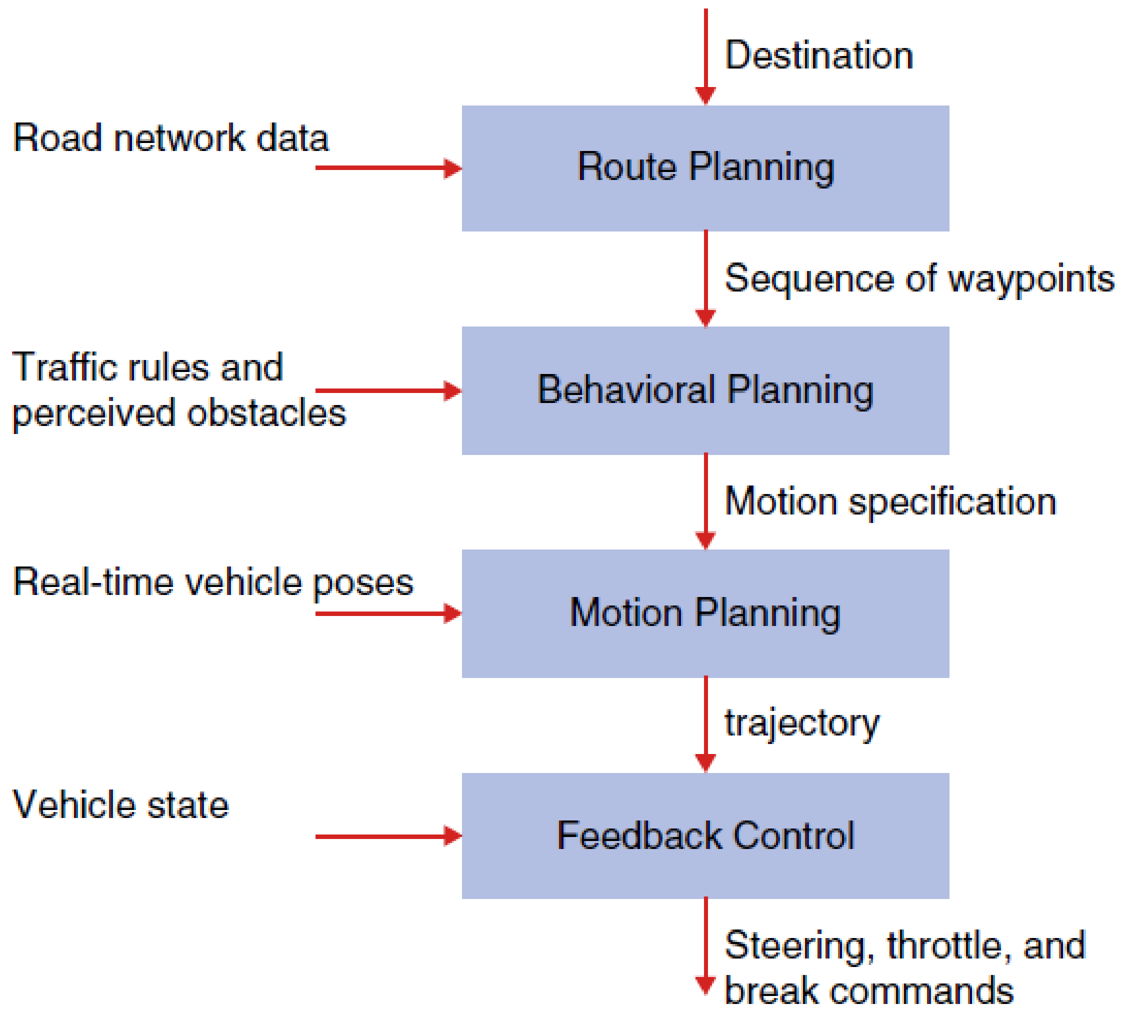
UNIVERSITY OF LEEDS

Route Planning

The first submodule is the route planner, which selects an optimal route by checking the road network information from the map.

Most common algorithms:

- Weighted Directed Graph
- Dijkstra’s Algorithm
- A* Algorithm

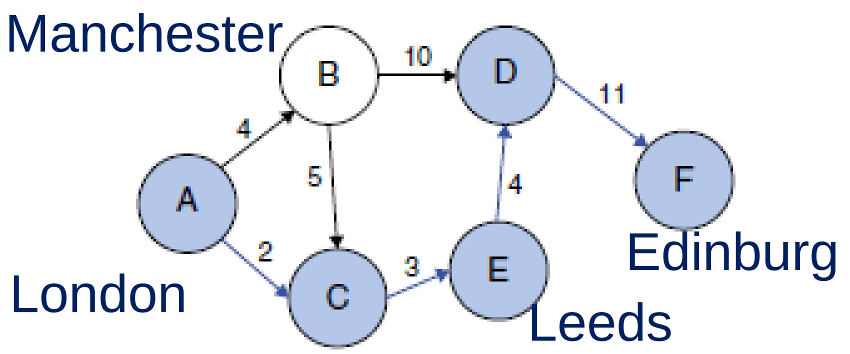


Weighted directed graphs

Weighted directed graphs can be used to represent road networks

For instance, a vertex can represent London, another vertex can represent Edinburg, and an edge connecting these two vertices records the distance between these two cities.

A routing algorithm can then be applied to this road network graph to search for the shortest route between the two cities.

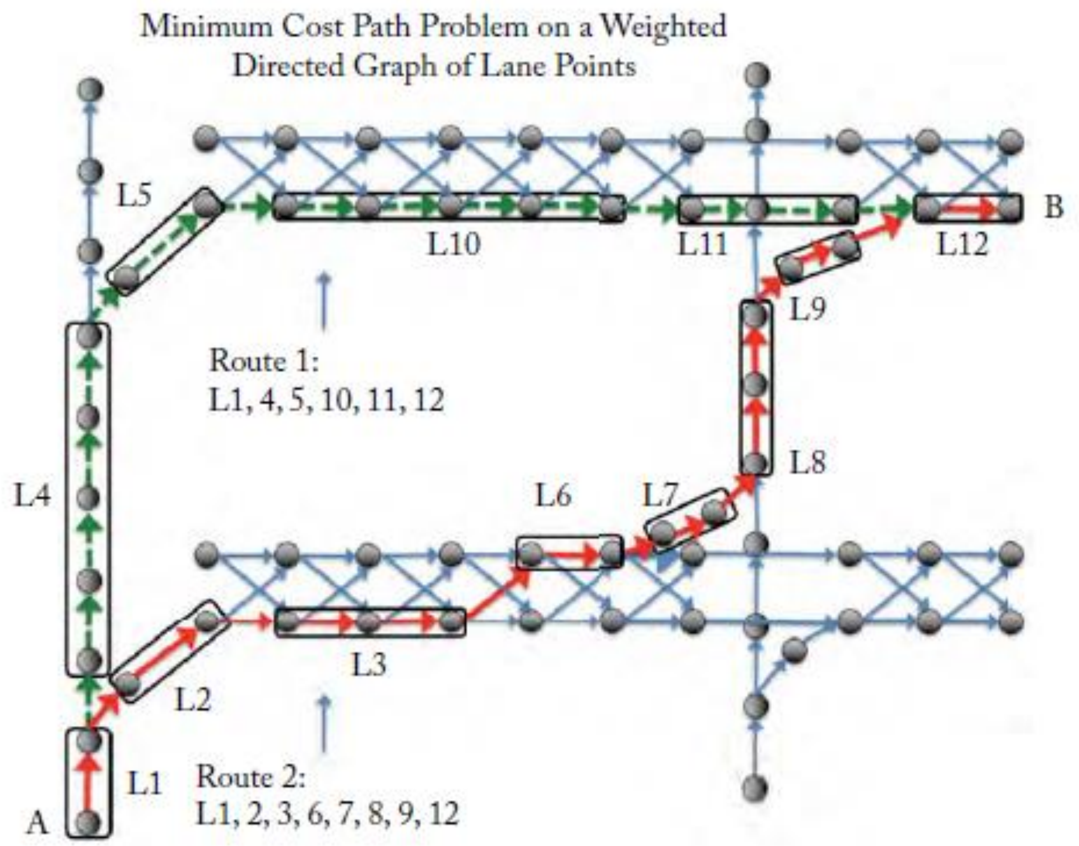


Dijkstra Algorithm on Autonomous Vehicle Routing

Dijkstra [dajkstra] is a very common shortest path algorithm in graph theory.

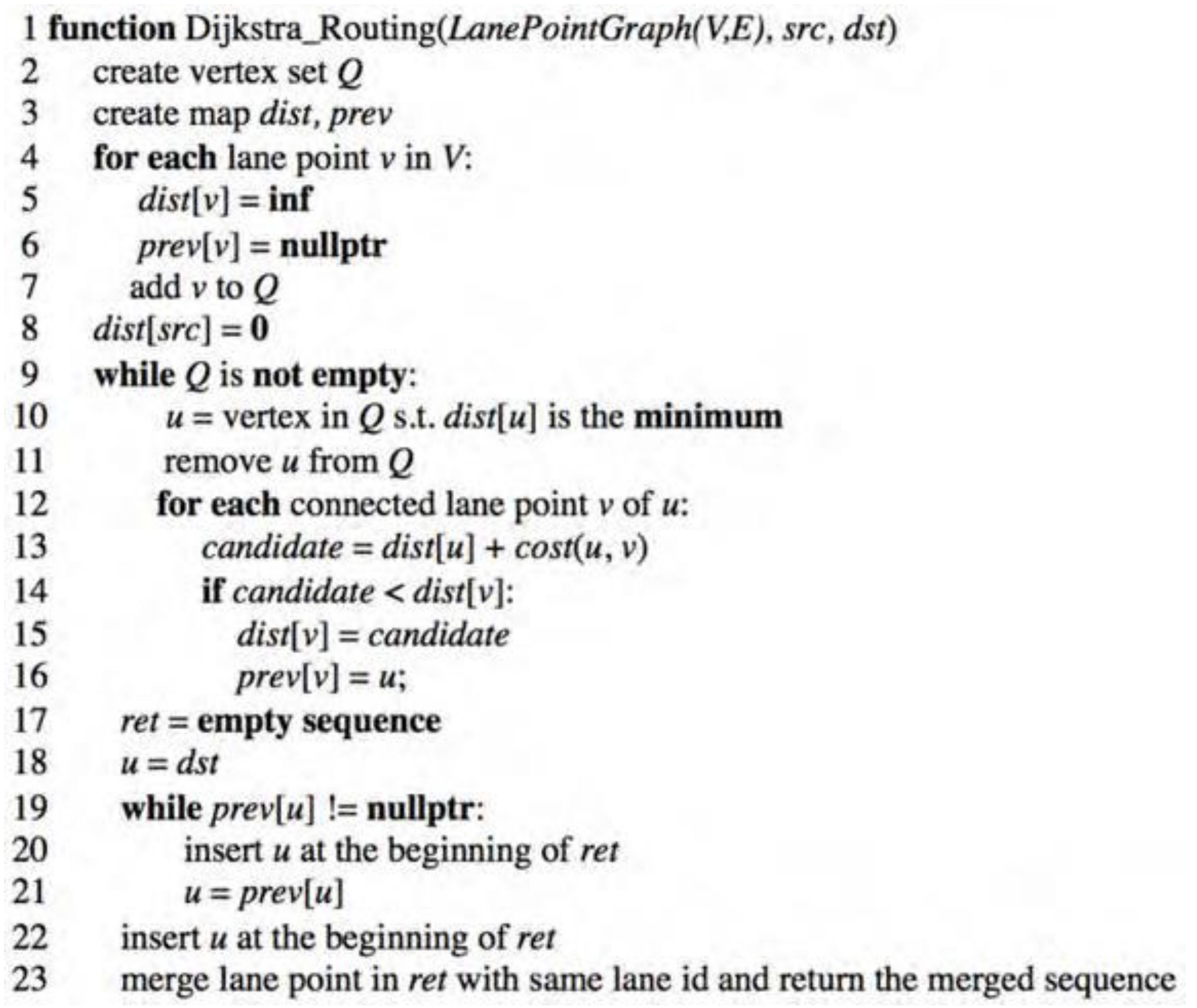
Proposed by Edsger W. Dijkstra in 1959, the algorithm solves the shortest path from a source node to a destination node on a weighted graph.

- Read the connected lane graph information From the HD map.
- Set the current node to the *Source Lane Point*
- Consider all the adjacent lane points which are unvisited and compute distance to reach these unvisited lane points.
- Repeat previous point until destination is reached.
- Shortest distance path has been found.



Pseudo code for Dijkstra Algorithm

Although Dijkstra’s algorithm guarantees to find a shortest path, when there is a large graph, Dijkstra’s algorithm can be extremely computationally expensive.



```
function Dijkstra_Routing(LanePointGraph(V,E), src, dst)
    create vertex set $Q$
    create map $dist$, $prev$
    for each lane point $v$ in $V$:
        $dist[v]$ = inf
        $prev[v]$ = nullptr
        add $v$ to $Q$
    $dist[src] = 0$
    while $Q$ is not empty:
        $u$ = vertex in $Q$ s.t. $dist[u]$ is the minimum
        remove $u$ from $Q$
        for each connected lane point $v$ of $u$:
            $candidate = dist[u] + cost(u, v)$
            if $candidate < dist[v]$:
                $dist[v] = candidate$
                $prev[v] = u$;
    $ret$ = empty sequence
    $u = dst$
    while $prev[u]$ != nullptr:
        insert $u$ at the beginning of $ret$
        $u = prev[u]$
    insert $u$ at the beginning of $ret$
    merge lane point in $ret$ with same lane id and return the merged sequence
```

A* Algorithm

A much faster algorithm, called the **Greedy Best-First-Search** algorithm works in a similar way: instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. However, Greedy Best-First-Search relies on a heuristic function and is not guaranteed to find the shortest path.

The A* algorithm, combines the **benefits of Dijkstra's algorithm** and the **Greedy Best-First-Search** algorithm.

A* is like Dijkstra's algorithm in that it can be **used to find a shortest path**.

A* is also like Greedy Best-First-Search in that **it can use a heuristic function to guide itself**.

Dijkstra's algorithm wastes time exploring in directions that are not promising.

A* Algorithm

Greedy Best-First-Search explores in promising directions but it may not find the shortest path.

The A* algorithm uses both the actual distance from the start and the estimated distance to the goal.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Behavioural Planning

After a route plan has been found, the autonomous vehicle must be able to navigate the selected route and interact with other traffic participants according to driving conventions and rules of the road.

1. **The routing output**: A sequence of lanes along with the desired starting and ending position (where to enter and leave along the lane).

2. **The attributes about the autonomous vehicle itself**: Current GPS position, current lane, current relative position given the lane, speed, heading, as well as what is the current target lane given the autonomous vehicle location.

3. **The historical information about the autonomous vehicle**: In the previous frame or cycle of behavioural decision, what is the decision output? Is it to follow, stop, turn, or switch lanes?

4. **Obstacle information around the autonomous vehicle**: All the objects within a range radius of the autonomous vehicle. Each perceived object contains attributes

5. **Traffic and map objects information**, Traffic rules.

Markov Decision Process (MDP)

UNIVERSITY OF LEEDS

By using MDP to formalise the decision-making process, a number of algorithms can be used to automatically solve the decision problem. The four components of an MDP model are: a set of **states**, a set of **actions**, the **effects** of the actions, and the immediate value (**rewards**) of the actions.

S: a set of **states**. The state is the way the world currently exists.

A: a set of **actions**. The problem is to know which of the available actions to take in for a particular state of the world.

T: **transitions**. The transitions specify how each of the actions change the state. We need to specify the action's effect for each state in the MDP.

R: immediate **rewards**. Measure of an action's value so that we can compare different actions. We specify the immediate value for performing each action in each state.

Motion Planning

When the **behavioural layer decides on the driving** behaviour to be performed in the current context, which could be, e.g. **cruise-in-lane, change lane, or turn right**, the selected behaviour has to be translated into a **path or trajectory** that can be tracked by the **low-level feedback controller**.

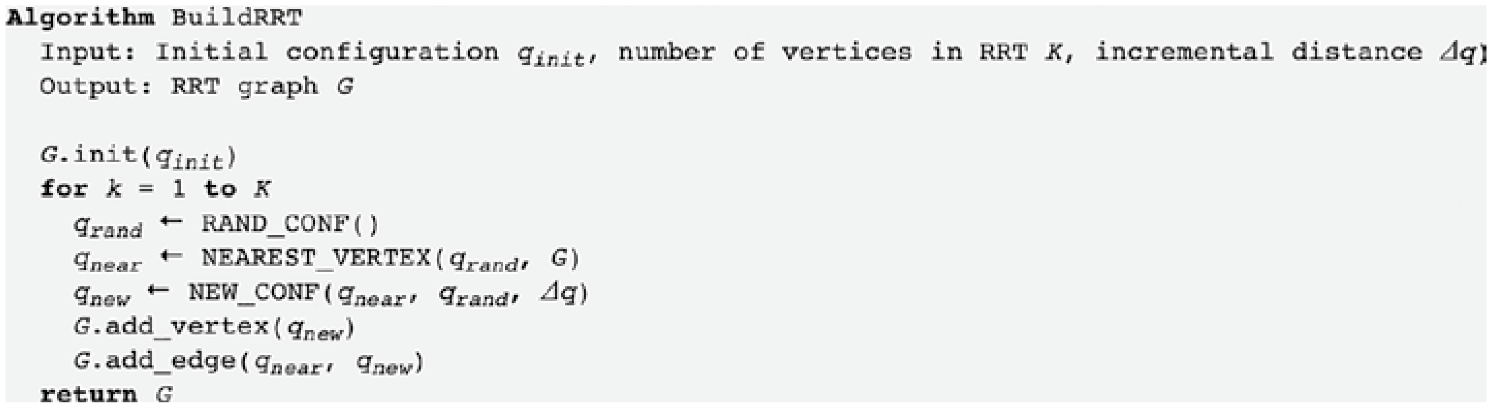
Exact solutions to the motion planning problem are in most cases computationally expensive. Thus, numerical approximation methods are typically used in practice.

Rapidly Exploring Random Tree - incremental tree-based approaches are widely used in motion planning.

Rapidly Exploring Random Tree

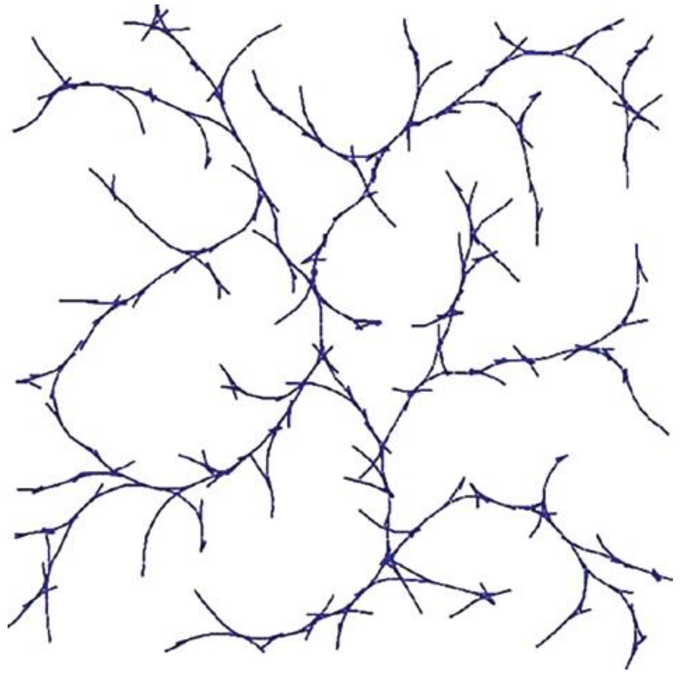
RRT **grows a tree rooted at the starting configuration** by using random samples from the search space. As each sample is drawn, a connection is attempted between it and the nearest state in the tree.

If the connection is feasible, this results in the addition of the new state to the tree.



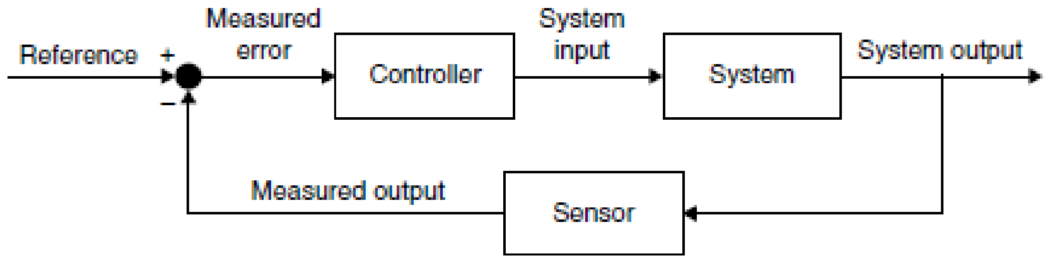
```
Algorithm BuildRRT
  Input: Initial configuration $q_{init}$, number of vertices in RRT $K$, incremental distance $\Delta q$
  Output: RRT graph $G$

  $G$.init($q_{init}$)
  for $k$ = 1 to $K$
    $q_{rand}$ ← RAND_CONF()
    $q_{near}$ ← NEAREST_VERTEX($q_{rand}$, $G$)
    $q_{new}$ ← NEW_CONF($q_{near}$, $q_{rand}$, $\Delta q$)
    $G$.add_vertex($q_{new}$)
    $G$.add_edge($q_{near}$, $q_{new}$)
  return $G$
```

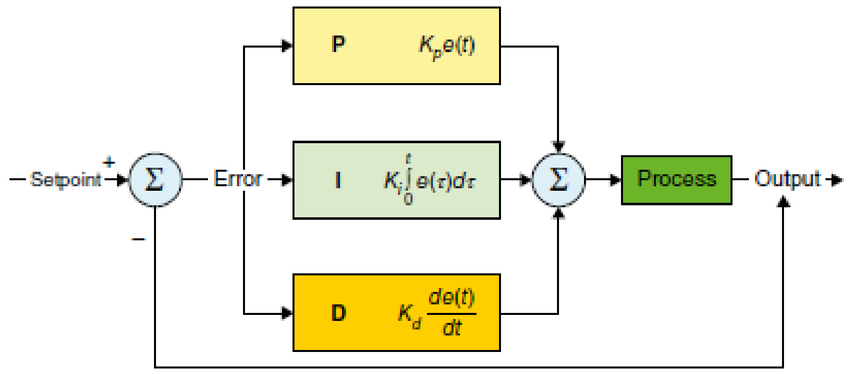


Feedback Control

In order to **execute the reference path or trajectory** from the motion planning system **a feedback controller is used** to select appropriate actuator inputs to carry out the planned motion and correct tracking errors.



Proportional–Integral–Derivative (PID) Controller



```
previous_error = 0
integral = 0
Start:
        error = setpoint – input
        integral = integral + error*dt
        derivative = (error – previous error)/dt
        output = Kp*error + Ki*integral + Kd*derivative
        previous_error = error
        wait (dt)
Goto Start
```

Conclusions

UNIVERSITY OF LEEDS

- Path Planning
  - **Route Planning**
    - Weighted Directed Graph
    - Dijkstra’s Algorithm
    - A* Algorithm
  - **Behavioural Planning**
  - **Motion Planning (RTT)**
  - **Feedback control**

References

1. Shaoshan, Engineering Autonomous Vehicles and Robots, 2020

2. Shaoshan, Creating autonomous vehicle systems, 2018

Q&A

ANY QUESTIONS

???