Hybrid A*

A*
- ✓ — A continous method: the configuration that make up the trajectory are not from a predefined discrete set.
- ✓ ✓ — Uses an optimistic heuristic like A*
- ✓ ✗ — It is not complete: does not always find a solution when a solution exists: The completeness is sacrificed because we only allow one continous configuration per diserete cell.
- ✗ ✓ — The solutions it finds are guaranteed to be drivable, though not neccessirily optimal
- ✓ ✗ — The solutions might not be optimal.

- Compared to A*, with Hybrid A* we've lost completeness and optimality in favor of drivenbility
- I practice Hybrid A* is very efficient at finding good path almost all the time.

Practical Considerations for Hybrid A*

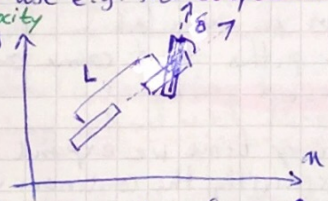
- check the video!
- The update equations we used above were generic to most (x,y) field configuration spaces.
- The ω is the rate of change of heading
- For some robots we might be able to specify ω idependant of the robot state
  → i.e. the robot can turn around its z axis without constraint (like a robot vacum cleaner)

$$x(t+\Delta t) = x(t) + v\,\Delta t \cos\theta$$
$$y(t+\Delta t) = y(t) + v\,\Delta t \sin\theta$$
$$\theta(t+\Delta t) = \theta(t) + \omega\,\Delta t$$

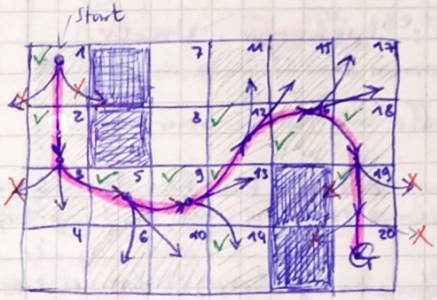
- For car this is not the case. We have to use e.g. the bicycle model

$$\omega = \frac{v}{L}\tan(\delta(t))$$

(v: constant positive velocity; $\delta(t)$: steering angle; L: distance between front and rear axis)



- now, for (hybrid) A* algorithm we need to figure out which configs can be reached from the current node, so that we add them → to the open set (the frontier)
- in practice we would use a number of steering angles between max steering angle and min steering angle (left and right)
  → but the more δs we have, the longer it takes for the algorithm to complete.
- For simplicity, in this example, we only choose three angles to choose δ from: { max_left_steering_angle, max_right_steering_angle, 0 }
  e.g: $\{-35°, 0°, 35°\}$
- now we can expand the search tree with three new motion primitives (actions). For a given speed, we assemble the path out of three components:



- Actions (motion primitives): go left, go straight, go right
- open cell (Frontier)
- closed cell (explored)
- At each step we discard the trajectories that go off the map or collide.
- at each step, we keep track of not just the cell we are in, but also "where in that cell we are, including the orientation"
- at each step the best cell to expand is chosen based on our heuristic function (like A*)
- At the step in cell 9, the heuristic will probably suggest to expand cell 14 (it is nearer to goal). But all the trajectories from there collide or are off the map → so close it and take the next best for expansion, .., and so on, ..
- what we finally get is a smoth path :)
  - But it is not always optimal (and sometimes very worse then what a human driver would do)
  - There are situations where this continuous nature of Hybrid A* will lead to no path being found
    → a solution could be increasing the resolution of the grid; or adding a third dimension to the search space for branching

Environment Classification :

- Hybrid A* is one of the best algorithms for trajectory generation in unstructured environments
- Unstructured Environments : e.g. a parking lot or a maze
  - → Less specific rules compared to highways or streets
  - → lower speeds
  - → no obvious reference path or trajectory that guide us ; because they change so much
- Structured Environments : e.g. highways , streets
  - → predefined rules govern how to move on the road
    e.g
    - direction of traffic
    - lane boundaries
    - speed limits
  - → these rules impose constraints that have to be satisfied but also offer guidance regarding what the trajectory should look like
- A* is great at finding solutions everywhere, it does not take advantage of this information. For example the road structure itself can be used as a reference path.

- Frenet Coordinates (reminder) : (s,d)
  - → longitudinal (s) distance and lateral distance (d) instead of cartesian (x,y) coordinates
  - → Note : instead of simply taking the center of the ~~road~~ lane as the reference line we could also take an actual nice and feasible trajectory computed with an offline planning algorithm like A* or hybrid A* ⟹ we would have a nominal path that we know the vehicle should follow if possible
- Need for Time

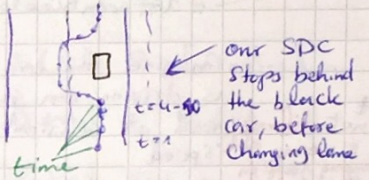
  - it is important to not just compute a sequence of configurations, but also to decide when we're going to be in each config.

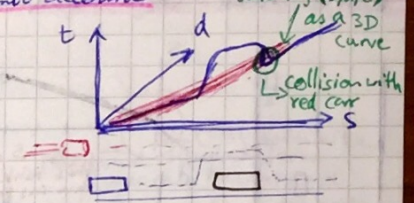
  → we need to incorporate time in trajectory planning ; because otherwise we don't when should the vehicle be at each → point of trajectory, which is necessary for dynamic environments

⟹ driving in traffic (and robot motion planning) is 3D by nature : (s,d,t) (t: time)

Time

- so far our path planning was generating a sequence of configurations using A* or Hybrid A*
- The paths generate using A* and Hybrid A* however, don't take predictions into account





⟹ we have to wait for the red car to pass us, before we change lanes.

imagine we perceive a fast moving vehicle behind us

⟹ Having only a path is not sufficient to solve the planning problem when there are dynamic objects around us.

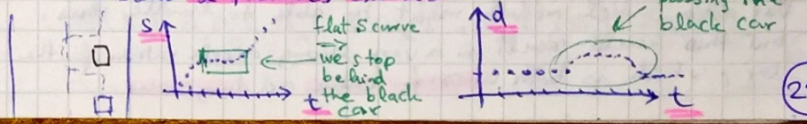
⟹ We have to plan not only a sequence of configurations (s,d,Θ), but also how these configurations are setup in time

⟹ The trajectory is a function, which for any time t associates a configuration of the vehicle.

$$\text{Trajectory } \sigma : [0, t_f] \longrightarrow \mathbb{R}^3$$
$$t \longrightarrow (s, d, \Theta)$$

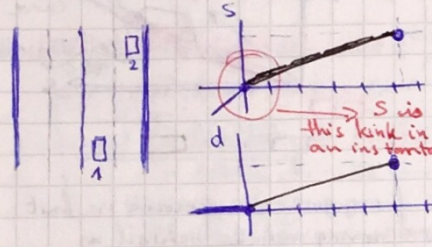
- We generate such trajectory by separating the planning we do in the s dimension from the planning we do in the d dimension

→ instead of thinking about 1 picture in 3D, we reason about 2 pictures in 2D



- Structured Trajectory Generation Overview:
  ↳ useful for structured environments like highways.
  - Boundary conditions
  - Polynomial Trajectories
  - Example - highway trajectory
  - Trajectory Generation algorithm:
    generate many similar jerk minimizing trajectories;
    compare them, and select the best one for our situation
  - Jerk Minimization
  - Deriving coefficients
  - Evaluating feasibility and cost

- We need continous and smooth trajectories: ← ≈ differentiable!



s is not differentiable: this kink in s means an instantaneous jump in speed, which would require infinite acceleration.

- imagine the car is driving on the road at constant speed
- the s and d graphs are shifted a bit for making computations simpler

- The s-t graph is a straight line, since the velocity is constant
- The d-t graph is flat, since the car stays at the center of the lane
- We want to leave the highway and be at position 2 after some Δt, say 10 sec.
- Start position (1) and goal position (2) are fixed. They define the boundary conditions of the trajectory at $t=0$ and $t=10$
- if those were the only boundary conditions we might consider s and d trajectories to be straight lines connecting start and end
  ↳ But these aren't physically possible
- the kink in slope, would mean an instantaneous jump in speed which would require infinite acceleration. ✗
- Our motion control module might be able to handle this, but this would result in a very strong acceleration of the car, which is both uncomfortable and dangerous.

⇒ The trajectory needs to be both continous and smooth.
But how much continuity and smoothness?

- Position must be continous → we cannot teleport :D
- Velocity needs to be continous → we cannot have instantaneous change in speed
- Acceleration must be continous → sudden changes in accel. (Jerks) are dangerous and uncomfortable.

Position $\xrightarrow{d}$ Velocity $\xrightarrow{d}$ Accel. $\xrightarrow{d}$ Jerk $\xrightarrow{d}$ snap $\xrightarrow{d}$ Crackle $\xrightarrow{d}$ Pop

- Jerk is directly related to the sense of comfort
  ↳ the rate of change in accel.

⇒ We design to minimize jerk. ⇒ A jerk minimizing trajectory

$s(t) \in [0, t_f]$

$\text{Jerk} = \dddot{s}(t)$

$\text{Total Jerk (squared)} = \int_0^{t_f} \dddot{s}(t)^2 \, dt$

(we want to minimize the total squared jerk (squared, because we consider both positive and negative))

Find $s(t)$ that minimizes ↑

- if we go through the math, we find that all functions that minimize that integral (the total squared jerk) must have all the time derivatives of s of order 6 or higher equal to 0:
- $\frac{d^m s}{dt} = 0 \quad \forall m \geq 6 \qquad (*)$
- also using Taylors series, we can write (any functions) $s(t)$ like this:

$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \dots + \alpha_n t^n + \dots$$

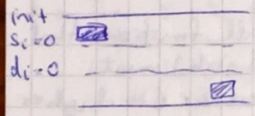
$$s(t) = \sum_{n=0}^{\infty} \alpha_n t^n$$

using $(*)$ $\alpha_6, \alpha_7, \dots = 0$ ⇒ All minimum Jerk Trajectories can be written as: $(a := \alpha)$

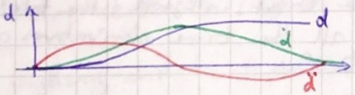
$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

← $s(t) = a_0 + a_1 t^1 + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$

- we have an equation with 6 coefficients / tuneable parameters
  → 6 coefficients → 6 tuneable param. → 6 Boundary Cond.
- We use the params to define the boundary conditions of the trajectory
  => we want to constrain
  → $[s_i, \dot{s}_i, \ddot{s}_i, s_f, \dot{s}_f, \ddot{s}_f]$ for logitudental displacement s
  → $[d_i, \dot{d}_i, \ddot{d}_i, d_f, \dot{d}_f, \ddot{d}_f]$ for lateral displacement d
  => we need to pick 12 variables in order to fully define the motion of our vehicle in S&D over time.
- The initial state of our vehicle, gives us the longitudinal and lateral boundary conditions for $t = 0$
- we need to pick the end boundary conditions for $t = t_f$
- For example for the Lateral movement simple example with simplifying assumptions that $\dot{d}_i, \ddot{d}_i, \dot{d}_f, \ddot{d}_f$ all $= 0$



for $d: [0, 0, 0, 10, 0, 0]$



- Note however that although the trajectories we generate using this method are Jerk optimal, but they depend heavily on the choice of boundary conditions.

Derivation:

$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$

We want to find the six coefficients $\alpha_0 - \alpha_5$ that define this polinomial.

We differentiate $s(t)$ to get equations for the velocity and acceleration:

$$\dot{s}(t) = \alpha_1 + 2\alpha_2 t + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4$$

$$\ddot{s}(t) = 2\alpha_2 + 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3$$

→

- We could now just plugin our 6 boundary conditions to get a system of 6 equations and solve for $\alpha_0 \dots \alpha_5$
- But we make it simpler by choosing $t_i = 0$ (initial time = 0)

$t_i = 0$ => 6 equation turn to 3 equations

(known)

$$s_i = s(0) = \alpha_0 \qquad \dot{s}_i = \dot{s}_i(0) = \alpha_1 \qquad \ddot{s}_i = \ddot{s}_i(0) = 2\alpha_2$$

=> 3 of the unknowns don't need to be identified anymore.
=> we now need to find 3 left unknowns

- let's gather the known terms, into functions of start boundary conditions and simplify the equations:

$$s(t) = \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5 + C_1$$
$$\dot{s}(t) = 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4 + C_2$$
$$\ddot{s}(t) = 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3 + C_3$$

(function of the initial condition)

- Pluging in the end boundary conditions:
  ($\alpha_3, \alpha_4, \alpha_5$ only depend on end boundary conditions)

$$s(t_f) = s_f = \alpha_3 t_f^3 + \alpha_4 t_f^4 + \alpha_5 t_f^5 + C_1$$
$$\dot{s}(t_f) = \dot{s}_f = 3\alpha_3 t_f^2 + 4\alpha_4 t_f^3 + 5\alpha_5 t_f^4 + C_2$$
$$\ddot{s}(t_f) = \ddot{s}_f = 6\alpha_3 t_f + 12\alpha_4 t_f^2 + 20\alpha_5 t_f^3 + C_3$$

- we know $s_f, \dot{s}_f, \ddot{s}_f$ and t_f, because we picked these values ourself as end boundary conditions

=> restructing the equation, we get a system of 3 equations for 3 unknowns

$$\begin{bmatrix} t_f^3 & t_f^4 & t_f^5 \\ 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix} = \begin{bmatrix} s_f \\ \dot{s}_f \\ \ddot{s}_f \end{bmatrix} - \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

→

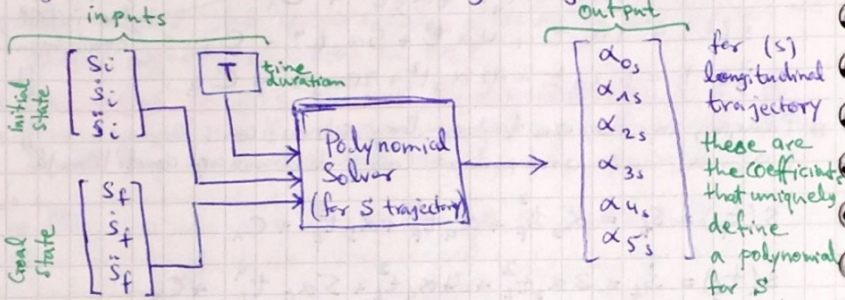
← summary:

1. The Jerk minimizing polynomials are degree 5 polynomials with 6 tuneable parameters $\alpha_0 \dots \alpha_5$

2. We find the coefficient $\alpha_0 \dots \alpha_5$ of this trajectory polynomial by using the initial ~~boundary~~ and end boundary conditions
   $[s_i, \dot{s}_i, \ddot{s}_i, s_f, \dot{s}_f, \ddot{s}_f]$ for s and
   $[d_i, \dot{d}_i, \ddot{d}_i, d_f, \dot{d}_f, \ddot{d}_f]$ for d

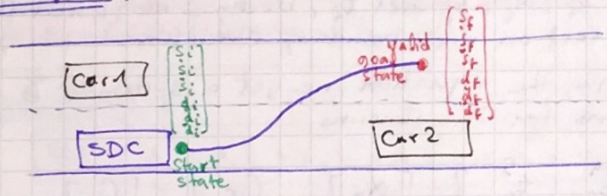
---

- Using the polynomial Solver to generate Jerk minimizing Trajectories:



- Same for (d) lateral displacement

Example:



- assume the requested behavior is to pass the car 2 in front of us
- we take the current config of the car including its velocity and accel. as start state. We also specify a valid end state that lead our ~~vehicle~~ vehicle to the other lane.
- We feed these states into our polynomial solver, →

← along with the desired duration that would allow for the maneuver.

⟹ and we get a jerk minimizing trajectory to the goal.

Evaluating the feasibility of a potential trajectory:

things that needs to be checked:

- max velocity
- min velocity (also backwards (negative velocity))
- max accel. (both lateral and longitudinal; max lateral accel. needs to be checked to avoid roll-overs or skid; max longitudinal accel. needs to be checked with power train capabilities)
- min accel. would be negative and corresponds to max breaking force
- Steering angle

- We don't cover the details of checking the feasibility, but here only some hints on initial validation:
  - Assumption: Neglect the curvature of the road and assume it is locally straight.
  - Longitudinal accel: assume the heading is aligned with the road

  ⟹ $\ddot{s}$ is the longitudinal accel of the car

$a_{max\ breaking} < \ddot{s} < a_{MAX}$ ← max accel. that the engine can supply

max deceleration when breaking: For now we assume it is constant. In real life, it should be computed using info about friction of the road.

- similarly for Lateral accel. we can check

  $|\ddot{d}| < a_y$ ← all $\ddot{d}$ values must be less than a fixed lateral accel. value that is set for comfort and to avoid any risk of rollover

- Steering Angle:

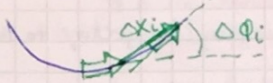
  from bicycle model: the relationship between steering angle and radius of the curvature of the road

  $\tan\delta = \frac{L}{R}$ ← L: distance between wheel axels; R: radius of road curve circl

  $(\kappa = \frac{1}{R})$ curvature

  $\Rightarrow \kappa_{max} = \frac{\tan(\delta_{max})}{L}$ gives us the max curvature allowed at any point in trajectory

← curvature of the path is then defined like this



$\Delta\varphi_i$ = heading difference between two points in the trajectory

Δx_i = distance between them

$$\kappa_i = \frac{\Delta\varphi_i}{\Delta x_i}$$

- Finally for velocity, we check it against values given by the map or the behavioral layer. $\dot{s}$

$$v_{min} < \dot{s} < v_{speed\ limit}$$

e.g. in highways ← e.g. of the road

• Putting it all together:

- the prediction/behavior layer in most cases does not send us an exact and config to reach
  But rather an approximate one
  ⇒ we need to identify a valid goal state from that approximation

- So we use our sampling-based method:
  - sample a large number of goal positions around the approximate desired config
  - ~~discard all trajectories~~
  - for each goal config, generate the corresponding jerk minimizing trajectories
  - discard all non-drivable trajectories, all that collide with the road boundaries or other vehicles or pedestrains as predicted by the prediction module
  - now we have a nice set of drivable trajectories which are collision free and jerk optimal.
  - we then define a cost function to rank the remaining trajectories and choose the best one

• Cost Functions:

the □ ones are more important to minimize

Jerk

J_d vs. J_s

Distance to Obstacles

near vs. far

Distance to center line

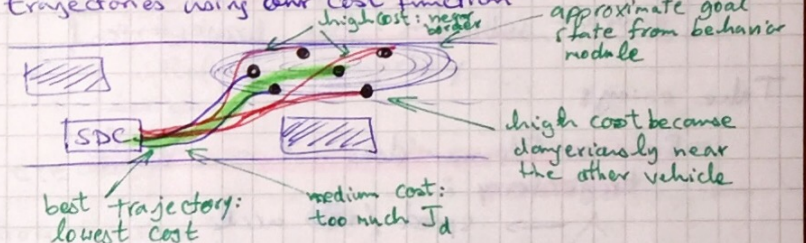
near center vs. far from center

Time to Goal

earlier vs. later

- the weighted cost function is continuously determined based on the situation.
- in practice balancing the combined cost function too is tricky (because they are most of the time contradicting)

- Now we rank the remaining jerk minimizing drivable trajectories using our cost function



- Most SDCs have several Trajectory Planners that they use depending on situation: e.g.
  → A Hybrid A* in parking lots
  → Polynomial Traj. Generation on low traffic highways
  → and maybe several others for situations like intersections, high traffic, etc.

09.06
22.06
14. Days!

Project 7: Path Planning (highway driving)

- The behavior planner does not give a target accel/decel (speed, etc.).
  It only tells us where to go (s, d)
  The rest is task of trajectory planner. (i.e. how to go there)
- Behavior Planner shouldn't also care about Collision avoidance.
  It is the job of trajectory planner to generate collision-free trajectories!

Take away:

- Behavior Planner does not care about trajectory:
  - X → speed / accel
  - X → collision avoidance
- Designing cost functions is hard

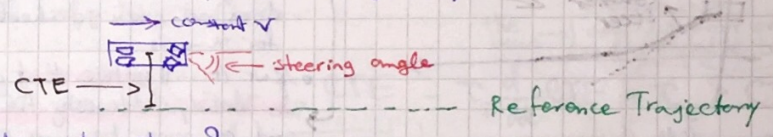
Lesson 12: PID Controller (2h)

26. June

- control:
  use { - Steering, - throttle, - breakes } to move the car to where we want it to go.
- Drew Gray: Director of Engineering Uber ATG (former); Voyage (CTO); Aquabink (founder)
  - Ph.D. Control Theory / Berkeley

• PID-Control

- CTE: Cross-track-Error
  - Lateral distance between Vehicle and Reference Trajectory



- How to steer?
  - constant steering angle → bad; drives in circle
  - random steering angle → crazy :))
  - Steer in PROPORTION to CTE.

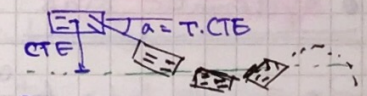
(inverse)

P: proportional

• P-controller.

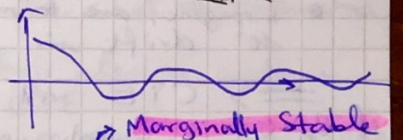
- What if now if we steer proportional to CTE?

control response to error → $\alpha = -\tau \cdot CTE$ ← gain control parameter, ← error



→ car overshoots

- A P-controller applied to a car will always slightly overshoot.
  → that could be ok; but never really converges



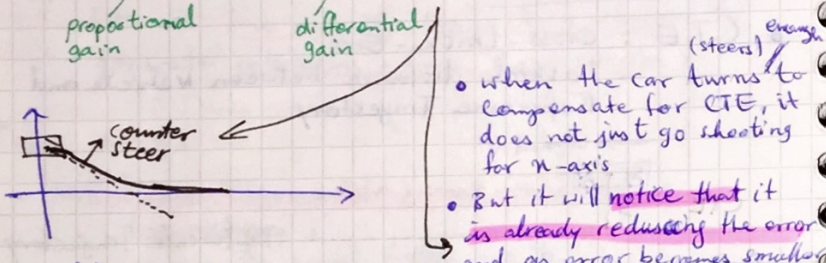
← if τ is larger, then it oscilates faster

PD - Control

- In PD-control the response is not just proportional to error using the gain parameter (τ). But it also depends on the temporal derivative of the error.

$$\alpha = -\tau_p \cdot CTE - \tau_D \frac{d}{dt} CTE$$

proportional gain — differential gain



- when the car turns (steers) to compensate for CTE, it does not just go shooting for x-axis
- But it will notice that it is already reducing the error and as error becomes smaller over time the car counter steers

⇐ gracefully approach the target trajectory (assuming appropriate settings for the diff. gain τ_D versus proportional gain τ_p)

- $\frac{d}{dt} CTE = \frac{CTE_t - CTE_{t-1}}{\Delta t}$

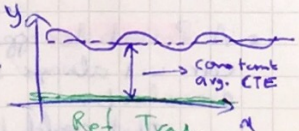
Systematic Bias

- Imagine, due to construction mechanical reasons there is 10° drift in steering

⟹ CTE will be high.

if $\tau_D = 0$

↳ interesting, also the bias is in steering, the error shows still in y-

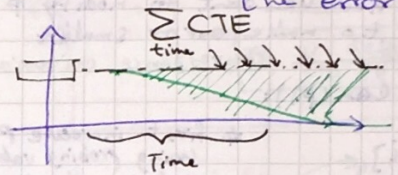


← • Can the differential term (D-term) ($\tau_D \cdot \frac{d}{dt} CTE$) solve this?

NO

PID - control

- Intuition: to compensate the systematic bias, we have to observe and consider the error over longer time!



- like a human: if you notice your car is drifting, (over a period of time observed), you steer against.

⟹ we need to consider the sum (aka integral) of the Error

the bias → compensate constant error over a long period of time

⟹ $\alpha = -\tau_p CTE - \tau_D \frac{d}{dt} CTE - \tau_I \sum CTE$

PID: proportional term (P) — differential term (D) — integral term (I)

- Twiddle (Coordinate Ascent) more, tuning, params

τ_p, τ_D, τ_I

- How can we find good control gains?
- see the video
- it is a local hill-Climber alg.

⟶ we want to optimize a set of parameters

- let our run (params) return also a goodness measure, e.g. avg. CTE

⇒ Twiddle wants to minimize the error. avg. CTE

R41 →

← run(params) ⟶ goodness

⇒ output of run() depends on params (say three params)

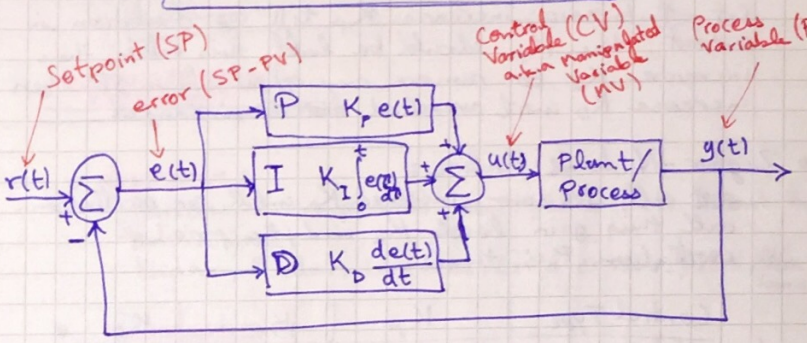
```
p = [0, 0, 0]      <- our initial vector of params
dp = [1, 1, 1]     <- vector of potential change that we want to probe

best_err = run(p)
while sum(dp) > 0.0001:        # now we want to modify p to make error smaller  <- convergence criteria
    for i in range(len(p)):
        p[i] += dp[i]          # first increase p using probing value and compute the new error
        err = run(p)
        if err > best_err:
            best_err = err
            dp[i] *= 1.1       # it worked good increase dp
        else:
            p[i] -= 2 * dp[i]  # try the other way -> subtract p by dp   (2 because we added once above)
            err = run(p)
            if err > best_err:
                best_err = err
                dp[i] *= 1.1
            else:
                p[i] += dp[i]
                dp[i] *= 0.9   # if neither adding or subtracting worked: decrease probing interval
```

- **Coordinate descent (ascent)** ⟶ from Wikipedia

⟶ minimize a multivariate $F(X)$ by ~~minimizing~~ minimizing along one coordinate at a time.

P8 : PID Controll

29. June • 19:30
30. June



Example: A robotic arm lifted or lowered by an electric motor to reach a desired position

- $y(t)$: the sensed position; Process Variable (PV)
- $r(t)$: desired position; Setpoint (SP)
- $e(t)$: error; difference between SP and PV
- $u(t)$: input to the process (e.g. the electric current in the motor) is the output of PID controller; called Manipulated Variable (MV) or Control Variable (CV)

K_p, K_I, K_D: Process gains

gain ↑	Risetime	Overshoot	SettlingTime	Steady state error	Stability
K_P ↑	↓	↑	small change ≈	↓	↓
K_I ↑	↓	↑	↑	↓↓	↓
K_D ↑	≈↓ minor decrease	≈↓	≈↓	no effect	↑

P: Oscilates around SP; make response fast and swift →

D: Dampes; makes response slow

I: Compensate steady error (systematic bias)

← Tunning Strategies for Grains:

- Manual; Online:
  Set all to zero; increase K_p till see oscilation in output; the K_p should be half this value; then increase K_I to remove any offset from SP; then increase K_D until overshoot ~~and dies~~ is damped
- Zigler-Nichols; Online:
  set all to zero; increase K_p until see oscillation. call this gain level K_u and the period of oscillation P_u; then:

Control Type	K_p	K_I	K_d
P	$0.5\,K_u$	–	–
PI	$0.45\,K_u$	$1.2\,\frac{K_p}{P_u}$	–
PID	$0.6\,K_u$	$2\,\frac{K_p}{P_u}$	$K_p\,\frac{P_u}{8}$

- coordinate descent
- SGD (grad. desc.)

{Github}

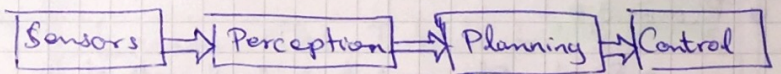
{LinkeIn}

01 July

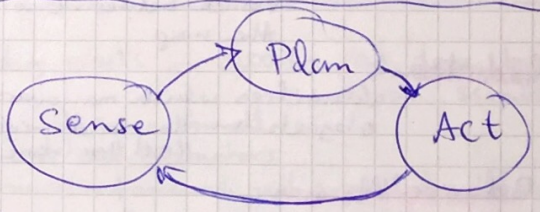
Lesson 16: Autonomous Vehicle Architecture

30m

- **Subsystems:**
  - **Sensor:**
    Hardware components that gather data about the environment
    Lidar, Radar, Cameras, GPS sensors, IMU, Ultrasound
  - **Perception:**
    software that process the sensor data and combine them into meaningful information
  - **Planning:**
    uses the output of perception for behavior planning and for short range and long range path planning
  - **Control:**
    ensures the vehicle follows the path provided by the planning subsystem and sends control commands to the vehicle



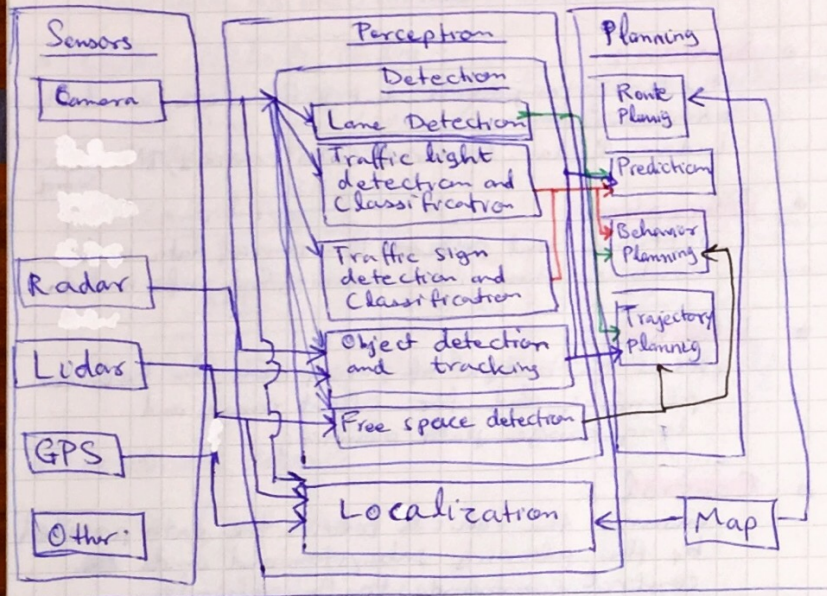
- SPA



245

- was the Predominent robot control methodology through 1985
- OODA: Observe; Orient; Decide; Act [John Boyd]
- PDCA: Plan; Do; Check; Act (Adjust) • Continuous improvement Process

- **Perception**
  - **Detection:**
    - Lane detection
    - Traffic sign & traffic light detection
    - Object detection and tracking
    - Free space detection
  - **Localization:** uses sensor and map data to determine the vehicle's precise location



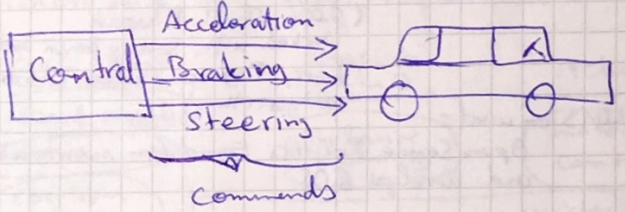
- **Planning:**
  - **Route Planning:** high level path of the vehicle between two points on the map
  - **Prediction:** identifies which manovers other objects on the road might take (and their future trajectory)
  - **Behavior Planning:** decides which manover our vehicle should take
  - **Trajectory Generation:** Plots the precise path we'd like our vehicle to follow.

- **Control:**
  - once the vehicle has a planned trajectory, the next step is to execute that trajectory
  - ensures that the vehicle follows the path specified by the planning subsystem

PID Controllers

Model-Predictive Controllers (MPC)

Other controllers



01 July

Lesson 17: Introduction to ROS

1 h

- Building robots in 2007: all done by yourself from scratch!
  - device drivers for all sensors and actuators
  - communication framework with support for different protocols
  - algorithms for perception, navigation, motion planning
  - logging
  - control
  - error handling — ...

- ROS:

- an opensource SW Framework for robotic dev.

- device drivers for HW
- message passing for IPC
- package mgmt. sys. / easy deployment
- visualization
- simulation and analysis
- extensive community support
- interface to numerous powerful SW libraries

- History of ROS:

- mid 2000's as project Switch Yard at Stanford AI Labs
- 2007: ROS → Willow Garage (2006 Scott Hassan worked with Sergey Brain and Larry Page on Google's Search algorithm)
- 2013 onward: Open Source Robotics Foundation maintains and develops ROS

Motivation:

- constantly reinventing the wheel by researchers
- different groups all working on own custom solution difficult to share code and ideas
- difficult to compare results

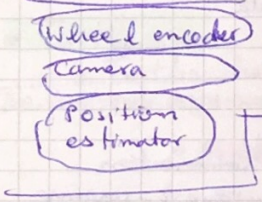
Used by: research & industry

drones | kinematic arms | wheeled robot | Bi-pedal

- Pretty much many robots do the following general tasks:

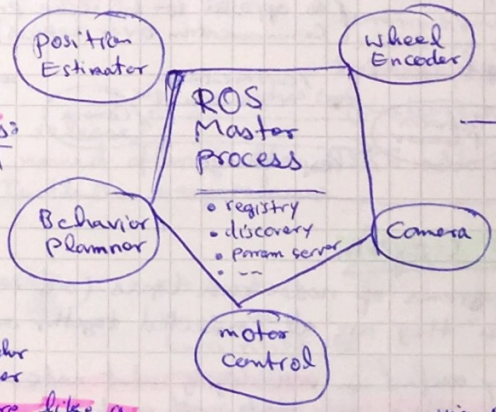
Perception — using sensors to perceive the world → **Decision Making** — making high level decision based on perceived info → **Actuation** — motors and controllers for performing the decisions

- ROS provides a powerful communication system, allowing these components to communicate with eachother
- ROS manages these three complex steps by breaking each of them down into **many small Unix processes** called **Nodes**
- Each node on a system is typically responsible for one small and relatively specific portion of the robots overall functionality; e.g. there maybe nodes for



ROS Master:

- registry of all active nodes
- discovery
- communication
- Parameter server



Communication between nodes:

- topic-based Pub-sub architecture
- passing messages using topics
- each node can be publisher and subscriber
- topics are like a named Bus

*** Note:**

ROS master allows nodes to discover one another, but after that they communicate with one another directly via pub-sub messages

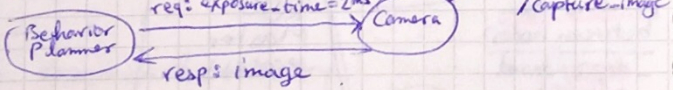
Messages: topic-based pub-sub communication

- Each ROS distribution comes with a variety of predefined message types
- e.g. message types for communicating physical quantities
  - position
  - velocities
  - accelerations
  - rotations
  - duration
  - ...
- messages for communicating sensor readings
  - laser scans
  - images
  - point clouds
  - inertial measurements
  - ...
- The number of message types on a full ROS installation is actually quite a lot, e.g. ≈ 200 different message types
- you can also define your own message types
- Messages can contain any type of data e.g. image, rotation, ...

In addition to passing messages using pub-sub

Services:

- provide request-response type communication
- 1 to 1 basis communication using request-response (as opposed to bus like pub-sub communication over topics)
- example:



Compute Graphs

- diagrams of nodes and topics (and services) and how they are all connected together and communicate
- very useful in visualizing what nodes exist, and how they communicate with eachother

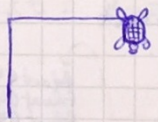
→

←
- ROS provides a tool called RQT Graph for showing the compute graph of the system
- Example:
  EMBot: a mobile base demonstration platform developed by Electric Movement co.
  - Relatively simple:
    - a lidar
    - a depth camera
    - four independently driven omni directional wheels
  - Although relatively simple, still the compute graph is quite big and complex

  ⇒ RQT Graph is very useful handy tool
  - allows zooming in and panning around the graph
  - allows filtering

Turtlesim

- Turtles have a long history in AI, neurobiology robotics and education
  - William Gray Walter (turtoises) - 1948! ← Elmer & Elsie
  - Seymour Papert (1960's turtles, LOGO 1980's programing language constructionism, experiential hands-on learning) turtle graphics (vector graphics) (turtleart.org)
- ROS releases are named after turtles; OSRF adds a new turtle to Turtlesim every release



```
put pen down
move 40
turn right 90°
move 40
turn right 90°
```

:)

- /opt/ros/kinematics/setup.bash for env vars
- env , ./ , source {ubuntu}
- roscore ; rosrun turtlesim turtlesim_node (package / node executable name)
  rosrun turtlesim turtle_teleop_key
- **Comms:** List all active nodes
  topics
  Getting info about topics
  Showing message info
  Echoing messages in realtime

- rosnode list
  - /rosout ← automatically launched by ROS master logging to file and terminal
  - /turtlesim
  - /teleop_turtle

  (node broadcast names)

- rostopic list
  - /rosout
  - /rosout_agg
  - /turtle1/cmd_vel
  - /turtle1/color_sensor
  - /turtle1/pose

  (turtle1 namespace)

- rostopic info /turtle1/cmd_vel
  Type: geometry_msgs/Twist ← (message type; part of geometry_msgs package)
  Publishers:
  - /teleop_turtle (http://10.0.2.15:42923)

  (one publisher)

  Subscribers:
  - /turtlesim (http://10.0.2.15:32971)

  (one subscriber)

- rosmsg show geometry_msgs/Twist

  Twist message definition — Two Vector3 messages
  - geometry_msgs/Vector3 linear
    - built-in types → float64 x, float64 y, float64 z
  - geometry_msgs/Vector3 angular
    - float64 x, float64 y, float64 z

→

- rosed : simple bash command for viewing and editing files in ROS environment
- rosed geometry_msgs/Twist.msg [opens a nano editor]

```
# This expresses velocity ...
Vector3  linear
Vector3  angular
```

Twist message itself consists of two Vector3 messages

comments and a lot of useful info on how the msg is meant to be used

- rosed geometry_msgs Vector3.msg
  → def. of Vector3 msg

When in doubt:
Check the source code! :)

- Echoing Topics:
  see the messages themselves as they are being published to a topic
  e.g. which axis is turtle rotating about?
  what are units of rotation? radians? degrees?

- rostopic echo /turtle1/cmd_vel [waits for msgs to be published and then echos them]

```
---
linear
  x: 0.0
  y: 0.0
  z: 0.0
angular
  x: 0.0
  y: 0.0
  z: 2.0
```

[issue a turn command in terminal for teleop]

---

next: How to build a basic ROS package

July 02 20:0

Lesson 18: Packages & Catkin Workspace

1h

- On an SDC you might have up to 50 different processes running and interacting with eachother
  - some might even crash and restart as you drive

  => having a flexible loosly coupled architecture like ROS around them is supremely helpful to build scalable software

---

- Catkin ⟶ powerfull build and package mgmt of ROS
- Catkin Workspace
  - a directory where catkin packages are built, modified and installed.
- Typically a ROS-based robot project is contained in one single catkin workspace (wps)
  - the wps holds a wide variety of catkin packages
- All ROS software and components are organized into and distributed as catkin packages
- A Catkin package is simply a directory containing a variety of resources that together build up a useful coherent module
  - src code for nodes
  - useful scripts
  - cfg files
  - assets
  - ... ...

⟶

Outline: Create Catkin wps; add pkgs; manage inter-pkg dependencies; compile everything together.

- mkdir -p ~/catkin_ws/src (ws)
  cd ~/catkin_ws/src
  - catkin_init_workspace
  - catkin_make
- ws/devel/setup.bash
  ↳ must be sourced befor using catkin ws.

- ROS has a large community of devs
- clone an existing pkg into our ws:
  - catkin_make
- install missing packages using apt-get install ...
  or
  - source devel/setup.bash
  - rosdep install simple_arm

ROS Packages have two types of dependencies: build & runtime

- ros launch : running all packages one by one by rosrun would be too cumbersome
  - launch multiple packages in one command
  - set default params in param server
  - auto-matically respond processes that have died
  - ...
- Gazebo simulator
- simple 2DOF robotic arm (simple_arm)
  - revolt joint

- rosdep check simple_arm
- rosdep install -i simple_arm

Creating packages:

- catkin_create_package <pkg_name> [dep1 dep2 ...]

/src/first_package/
CMakeLists.txt
package.xml
} describe dependencies and the package

- typical pkg dir structure

scripts (python executables)
src (C++ src files)
msg (message defs)
srv (service defs)
include (headers / libraries that are needed as dependencies)
config (config files)
launch (launch file, more automated way of starting nodes)

other files / folds

urdf (universal robot description files)
meshes (CAD files in .dae (Collada) or .stl (StereoLitography) format)
worlds (XML like files that are used for Gazebo simulation env)

- ros.org / browse ⟶ find packages

July 03 – July 03

Lesson 19: Writing ROS Nodes
1 h

- Writing nodes in Python that publish and subscribe to topics, and a ROS service that can be called from other nodes or from the command line.

- simple_mover node:
  - publish join angle commands to simple_arm

- arm_mover node:
  - provides a service called safe_move: allows the arm to move to any position within the workspace that is safe
  - safe zone is bounded by min and max joint angles; configurable via the ROS param server

- look_away node:
  - subscribes to a topic where camera data is being published
  - when the camera detects an image with uniform color (meaning it is looking at the sky) the node will call the safe_move service to move the arm to a new position

Script / Publisher

- ROS Publishers: (Python)

pub1 = rospy.Publisher("/topic_name", msg_type, queue_size=size)

- ROS Publishing:
  - Synchronous: (default behavior of a Publisher if queue_size not set or None)
    Block if another publisher is publishing to a topic. Block till all messages of the other publisher are written to buffer and buffer has written all messages to subscribers.
  - Asynchronous:
    queue_size ← Publisher can store messages in the queue until it can be sent. if queue full, oldest msgs are dropped

← publishing message to topic:

```
pub1.publish(msg)
```

- messages are usually picked up from queue with the same rate as they are being written, example:
  - publisher publishes at 10 Hz
  - message typically picked up at 0.1 sec with a std dev of 0.05 sec.
  - what is a good queue_size?
    - X 10? too large
    - ✓ 2? ok. gives enough room for msgs to be queued if not picked in time
    - X 0? too small all msgs will be dropped

- simple_mover : publish joint movement commands to simple_arm

  Topic name : /simple_arm/joint1_position_controller/command
  Msg type : std_msgs/Float64
  Description : Commands Joint1 to move center clockwise (units radians!)

  Topic name : /simple_arm/joint2_position_controller/command
  Msg Type : std_msgs/Float64
  Description : " Joint2 " " " " " "

- create a new script in ws/src/simple_arm/scripts
  scripts must be executable to run with rosrun

```
chmod u+x hello
```

catkin_make
source ~/...

```
rosrun simple_arm hello
```

... check the code ...

- import rospy : official Python client for ROS
- rospy.init_node("arm_mover") : init node and register with ROS master

def mover():

- rospy.Rate(10) : are used to limit the frequency of loops

```
rate.sleep()
```

- rospy.Time.now().to_sec() : initially returns zero till first msg is received on the /clock topic

continuously poll till a non zero value is returned:

```
start_time = 0
while not start_time:
    start_time = rospy.Time.now().to_sec()
```

- main loop:

```
while not rospy.is_shutdown():
    elapsed = rospy.Time.now().to_sec() - start_time
    pub1.publish(math.sin(2*pi*0.1*elapsed)*(pi/2))
    pub2.publish(math.sin(2*pi*0.1*elapsed)*(pi/2))
    rate.sleep()
```

sample joint angles from a sin wave with period of 10 sec and magnitude $[-\frac{\pi}{2}, \frac{\pi}{2}]$

- main method (node entry point):

```
if __name__ == "__main__":
    try:
        mover()
    except rospy.ROSInterruptException:
        pass
```

important

raised on shutdown signal

any cleanup code needed

- rospy uses exceptions intensively!

- arm_mover node / safe_move service
- rospy.Service("service_name", srvClsName, handler) incoming serv. reqs.

name of function that handles

filename where serv. def is

Service

serv. def. : .srv file defining message types for req and resp.

← Using Services :

- Services can be called directly from cmd line

or

- using `rospy.ServiceProxy("ser_name", srvClsName)`

```
# update msg attrs to have correct data
req = ~~req()~~ srvClsNameRequest()
resp = proxy(req)
```

given by ROS by appending a Request() to srvClsName

- example uses of services
  - fold a robotic arm and kill request to stop all processes related to it
  - set param values ; e.g. pen color of turtle
  - executing movements ; e.g safe-move srvs

- next steps:

  what we did so far:
  - writing a ROS node in a Package with a Publisher

  - custom msg generation
  - Services
  - Parameters
  - Launch Files
  - Subscribers
  - Logging

- arm_mover node
  - like simple_mover node, arm_mover also moves the robot arm
  - but instead of commanding the arm to follow a predefined trajectory, the arm_mover node provides the service move_arm
  - the move_arm service allow other nodes in the system to send movement_commands
  - in addition to moving the arm via a service interface arm_mover also allows for configurable min and max joint angles by using parameters

- Service def. :

  Add service

  - .srv file in /srv/ dir in package →

← • Create a GoToPosition srvice:

- create a file ws/srv/GoToPosition.srv :

.srv always two sections separated by ---

```
req  { float64 joint_1
       float64 joint_2
       ---
resp { duration time_elapsed
```

built-in type duration

[Note: defining a custom msg is very similar with following differences:
- in ws/msg dir
- .msg extention
- no section divider ---
]

- Modify CMakeLists.txt : (ws/src/simple_arm/CMakeLists.txt)

- ensure find_package() macro has the required packages

Contains all basic msg types → std_msgs and generate → message_generation

generate message libraries for all supported languages (cpp, python, lisp, js)

- uncomment add_service_files() macro
  (this tells catkin which files to generate code for)
- ensure that generate_messages() macro is uncommented
  (this macro is actually responsible for generating the code)

- Modify package.xml
  - responsible for defining many of the pkg properties e.g name, version, authors, maintainer, dependencies, ...
  - rosdep uses package.xml to search for dependencies (runtime and build-time)
  - add build dep. message_generation and runtime dep. message_runtime

→

- now build the package
  → catkin generates a python package containing a module for the new service GoToPosition
  ws/devel/lib/python2.7/dist-packages/simple_arm/srv/
  ↳ _GoToPosition.py

- now source the setup.bash
  → the new dist-packages is added to pythonpath env variable
  - env | grep PYTHONPATH

- Create the empty **arm_mover node script**
  - touch ws/src/simple_arm/scripts/arm_mover
  - chmod u+x ——— /arm_mover

Check code

- from sensor_msgs.msg import JointState
  from simple_arm.srv import *
  (published to /simple_arm/joint_state topic
  * check launch/robot_description.xml)
  (generated in site-packages)

- at_goal (pos_j1, goal_j1, pos_j2, goal_j2)
  # check if position is at goal (± noise)

- clamp_at_boundaries (requested_j1, requested_j2)
  # check and enforce min/max joint angles
  Clamp if requested angles are outside allowable bounds
  - rospy.get_param("~min_joint_1_angle", 0)
    (~ namespace qualifier: indicates that this param name is within the node's namespace (private)
    default if param was not found
    e.g. /arm_mover/min_joint_1_angle)
  - rospy.logwarn (——)
    # warning if joint angle was clamped

- move_arm (pos_j1, pos_j2)
  # command the arm to move by publishing the position and return the amount of time elapsed
  #publish position → j1_publisher.publish (pos_j1) (defined in —main—)
  - rospy.wait_for_message ("/simple_arm/joint_states", JointState) (topic where joint state is published; msg type)
    (blocking function call: does not return until a msg has been received on the topic)
    # get the current joint state to check with at_goal()

(in general you should not use wait_for_message()!
(used here for clarity and became move_arm() is being called by handle_safe_move request function which demands that the response message is passed back as the return param)

- handle_safe_move_request (req)
  # service handler function
  When a service client sends a GoToPositionRequest to safe_move service, this function is called with the request (type of req is GoToPositionRequest)
  # the service response is of type GoToPositionResponse

Note: move_arm() function is blocking and will not return until the arm has finished its movement. Incomming messages to service handler cannot be processed and no other useful work can be done in the Python script, while the arm is performing its move command!
In this example it is not a big problem
But this practice (blocking) should generally be avoided!

One greate way to avoid blocking the thread of execution would be to use an Action rather than a service. Action vs. Service vs. Topic

- Action vs. Service vs. Topic

- mover_service()

```
# initialize arm_mover node and
  Create a service with name safe_move
- rospy.init_node("arm_mover")          # "arm_mover" = node name
- service = rospy.Service("safe_move",  # "safe_move" = service name
                          GoToPosition, # GoToPosition = service class name
                          handle_safe_move_request)
                                        # handle_safe_move_request = service handler function
```

current node's private namespace (/arm_mover/safe_move)

- rospy.spin() ← blocks till the shutdown request is received by the node

if forgotten mover_service() simply returns and the script completes execution

- Finally the __main__ :

```
if __name__ == __main__:
    j1_publisher = rospy.Publisher(----)   # same topic as simple_mover node
    j2_publisher = ---
    try:
        mover_service()
    except rospy.ROSInterruptException:
        pass
```

- to get the arm_mover node and its safe_move service to launch along with all other nodes, we modify robot_spawn.launch launch file

add the following to launch file

```
<node name="arm_mover" type="arm_mover" pkg="simple_arm">
  <rosparam>
    min_joint_1_angle: 0
  </rosparam>
</node>
```

- rosservice list
- rqt_image_view [rqt and its associated tools]
  - a Qt-based framework for GUI development for ROS
- rosservice call /arm_mover/safe_move "joint_1: 1.57 newline! join_2: 1.57"
  - auto completes even the service req params :)

to get a newline in bash: ctrl+v ctrl+j

- rosparam set /arm_mover/max_joint_2_angle 1.57

What have we learned so far:
- Custom msg generation, publishing to a topic, ROS services, parameters, launch files
- next: ROS subscribers

- ROS subscribers:
  - enable a node to read msgs from a topic allowing useful data to be streamed into the node

```
sub1 = rospy.Subscriber("/topic_name", msg_type, callback_function)
```

called with msg as argument

- Look_away node (defined in simple_arm.gazebo.xacro (plugin camera_controller) -->)
  - subscribes to the /rgb_camera/image_raw topic which has image data from the camera mounted at the end of robotic arm

Check Code

- from sensor_msgs.msg import Image, JointState (Image: message type)

- rospy.init_node("look_away")
- self.sub1 = rospy.Subscriber("/simple_arm/joint_states", JointState, self.joint_state_callback)
  # listen to joint states
  # check camera only when the arm is not moving (by checking the joint state)
- self.sub2 = rospy.Subscriber("rgb_camera/image_raw", Image, self.look_away_callback)
- self.safe_move = rospy.ServiceProxy("/arm_mover/safe_move", GoToPosition)
  for calling a service from a node
- uniform_image(self, image)
  # take an image and check if all color values are the same as the value of the first pixel (is image uniform, like the gray sky)
- coord_equal(coord1, coord2)
  # check if coords are equal (+ noise tolerance)
- joint_state_callback(data)
  # check if arm is moving by comparing the data with last position (using the coord_equal())
- look_away_callback(data)
  # if arm is not moving and image is uniform:
  try:
    rospy.wait_for_service("/arm_mover/safe_move")
    msg = GoToPositionRequest()
    msg.joint_1 = 1.57
    msg.joint_2 = 1.57
    resp = self.safe_move(msg)
  except rospy.ServiceException

- now launch and interact with simple_arm as before
  - move arm to a sky looking position
    rosservice call /arm_mover/safe_move "- - - - 0 - - - - 0"
  - arm moves and looks at sky and then automatically moves back to 1.57 1.57 looking at boxes

Logging

- rospy.logwarn(" ")
- rospy.loginfo(" ")
- log are by default written to the node's log file in ~/.ros/log or ROS_ROOT/log
- roscd log ← requires a running roscore!
  - goes to log dir (latest run/launch)
  - log dir contains directories for runs of ROS code
  - the latest dir is a symlink to log dir of the latest run

- log debug(), loginfo(), logwarn(), logerr(), logfatal()
- all levels are written to log file
- loginfo messages are also written to pythons stdout
- warn, err, fatal are also written to Pythons stderr
- log fatals are also written to /rosout topic

	Debug	Info	Warn	Error	Fatal
stdout		x			
stderr			x	x	x
logfile	x	x	x	x	x
/rosout		x	x	x	x

- Filtering and saving log messages from /rosout topic:
  - rostopic echo /rosout
    - → see log messages of /rosout in real-time as the program is running
    - → sometimes too many messages
  - filter by piping to grep

```
rostopic echo /rosout | grep <pattern>
```

Save to file:

```
rostopic echo /rosout | grep <pattern> > outputslog.
```

- Modify log level for /rosout:

```
rospy.init_node("my_node", log_level = rospy.DEBUG)
```

- Modify display of messages sent to stdout and stderr
  - in launch file, set the "output" attribute of the node to "screen" or "log" (default is log)

	stdout	stderr
"screen"	screen	screen
"log"	log	screen & log

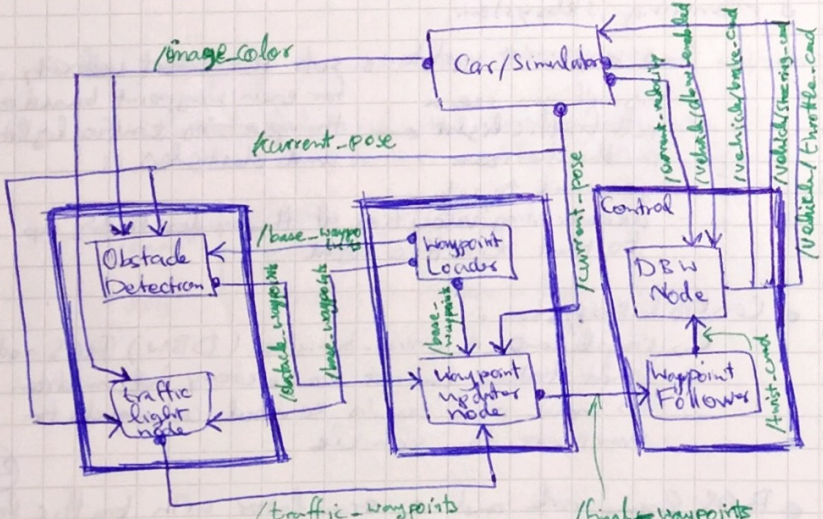
08. July
14. July

PG: Final Project
Program an Autonomous Vehicle

- waypoint navigation
  - → each done an associated target velocity
- Carla's control subsystem
  - → throttle, steering, break
- Project tasks: implementation of → perceception, planning, control subsystems
- perception subsystem
  - → traffic light detection
  - → obstacle detection
- planning subsystem
  - → waypoint updater: sets the target velocity for each waypoint based on the upcoming traffic lights and obstacles
    - e.g. if you see a red traffic light in the horizon you want to set decelerating velocities at the nodes leading up to that traffic light
- Control subsystem:
  - → implement a drive-by-wire (DBW) ROS node that takes target trajectory information as input and sends control commands to navigate the vehicle
- ROS framework and a simulator with traffic lights and obstacles is provided →

- Ubuntu 14.04 → ROS Indigo
  Ubuntu 16.04 → ROS Kinetic
- Dataspeed DBW
- Code on VM / simulator local / port forwarding 4567
- Simulator: two tracks
  - high way with traffic lights
  - testing ~~parking~~ lot track of the real Carla
    - needs updating for a new set of way points
- Latency problem
  - ~~leave~~ remove data off when developing car's ~~control~~ controllers
- Simulator shows mph
  - but all units including incoming v from simulator are metric.

System Architecture



- proj/ros/src

/tl_detector/

- tl_detector.py

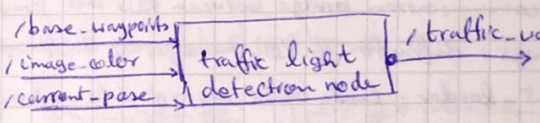
vehicle's current position

input: /image_color, /current_pose, /base_waypoints

complete list of waypoints the car will be following.

publish: the location to stop for red tl to /traffic_waypoints

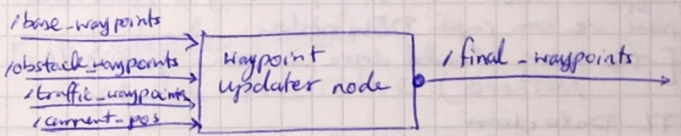
- build both a tl detection (within tl_detector.py) and a tl classification node within ../tl_detector/light_classification_model/tl_classifier.py



- proj/ros/src/waypoint_updater/
  - waypoint_updater.py

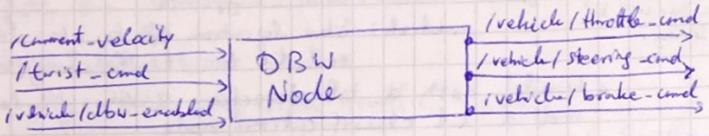
update the target velocity of each wp based on traffic light and obstacle detection

- subscribe: /base_waypoints, /current_pose, /obstacle_waypoints, /traffic_waypoints
- publish: a list of waypoints ahead of car with target velocities to /final_waypoints



- proj/ros/src/twist_controller
  - dbw_node.py, twist_controller.py responsible for control of the AV
  - a PID and low pass filter that we can use
  - subscribe: /current_velocity, /twist_cmd receive target linear and angular velocities →

- subscribe : /vehicle/dbw_enabled
- publish:
  throttle, break, steering commands to
  /vehicle/throttle_cmd, /vehicle/brake_cmd,
  /vehicle/steering_cmd



Additional Packages:

- Styx : communication bridge between ROS and simulator
- styx_msgs : definition of ROS messages use for project
- waypoint_loader : loads static waypoint data and publishes to /base_waypoints
- waypoint_follower :
  contains code from Autoware, subscribes to /final_waypoints and publishes target vehicles linear and angular velocities in form of twist commands to /twist_cmd topic

Suggested work plan:

1. (Partial) Waypoint updater : read /base_waypoints and /current_pose and publish to /final_waypoints
2. As soon as Waypoint Updater publishes to /final_waypoints, Waypoint Follower can start publishing to /twist_cmd. Now we can impel DBW node
   → [The car should drive in sim - ignoring TLs] ✓
3. TL Detection
   - Detect TL and color from /image_color; test using ground truth from /vehicle/traffic_lights (has exact loc and status of all TLs in sim)
   - Waypoint publishing : convert position of Detected TL to wp index and publish it.
4. Waypoint Updater (Full) : use /traffic_waypoint to change waypoint target velocity before publishing to /final_waypoints
   → [car stops at red TL and start moving on Green]

Finished on 14. July

Check Notes in Folder for Extra-Cricular stuff:

- Object Detection using classic CV
- Fully Convolutional Networks
- Scene Understanding
- Inference Performance
- Semantic Segmentation
- Unscented Kalman Filter
- Vehicle Models
- Model Predictive Control

- Functional Safety
- Vehicle Models
- Model Predictive Control