**A\* Reminder**

If you'd like to refamiliarize yourself with the A\* search algorithm (covered in lesson 1), feel free to watch the video below. The question at the bottom of the page will be asked again later in the lesson in regards to Hybrid A\*.

## Note Regarding A\* Heuristics and Optimal Solutions:

A\* will find an optimal solution if the heuristic is [Admissible](https://en.wikipedia.org/wiki/Admissible_heuristic). A\* may find a suboptimal path if the heuristic function is not Admissible.

# Hybrid A\* Heuristics

The paper [Junior: The Stanford Entry in the Urban Challenge](https://d17h27t6h515a5.cloudfront.net/topher/2017/July/595fe838_junior-the-stanford-entry-in-the-urban-challenge/junior-the-stanford-entry-in-the-urban-challenge.pdf) is a good read overall, but **Section 6.3 - Free Form Navigation** is especially good and goes into detail about how the Stanford team thought about heuristics for their Hybrid A\* algorithm (which they tended to use in parking lots).

Read section 6.3 of [the paper](https://d17h27t6h515a5.cloudfront.net/topher/2017/July/595fe838_junior-the-stanford-entry-in-the-urban-challenge/junior-the-stanford-entry-in-the-urban-challenge.pdf) and then answer the following question:

## Quiz: Valid Heuristics

If you have two individually admissible heuristic functions h\_1*h*1​ and h\_2*h*2​, which of the following **combinations** are also valid?

# Hybrid A\* Pseudocode:

The pseudocode below outlines an implementation of the A\* search algorithm using the bicycle model. The following variables and objects are used in the code but not defined there:

* State(x, y, theta, g, f): An object which stores x, y coordinates, direction theta, and current g and f values.
* grid: A 2D array of 0s and 1s indicating the area to be searched. 1s correspond to obstacles, and 0s correspond to free space.
* SPEED: The speed of the vehicle used in the bicycle model.
* LENGTH: The length of the vehicle used in the bicycle model.
* NUM\_THETA\_CELLS: The number of cells a circle is divided into. This is used in keeping track of which States we have visited already.

The bulk of the hybrid A\* algorithm is contained within the search function. The expand function takes a state and goal as inputs and returns a list of possible next states for a range of steering angles. This function contains the implementation of the bicycle model and the call to the A\* heuristic function.

**def** **expand**(state, goal):

next\_states = []

**for** delta **in** range(-35, 40, 5):

*# Create a trajectory with delta as the steering angle using*

*# the bicycle model:*

*# ---Begin bicycle model---*

delta\_rad = deg\_to\_rad(delta)

omega = SPEED/LENGTH \* tan(delta\_rad)

next\_x = state.x + SPEED \* cos(theta)

next\_y = state.y + SPEED \* sin(theta)

next\_theta = normalize(state.theta + omega)

*# ---End bicycle model-----*

next\_g = state.g + 1

next\_f = next\_g + heuristic(next\_x, next\_y, goal)

*# Create a new State object with all of the "next" values.*

state = State(next\_x, next\_y, next\_theta, next\_g, next\_f)

next\_states.append(state)

**return** next\_states

**def** **search**(grid, start, goal):

*# The opened array keeps track of the stack of States objects we are*

*# searching through.*

opened = []

*# 3D array of zeros with dimensions:*

*# (NUM\_THETA\_CELLS, grid x size, grid y size).*

closed = [[[0 **for** x **in** range(grid[0])] **for** y **in** range(len(grid))]

**for** cell **in** range(NUM\_THETA\_CELLS)]

*# 3D array with same dimensions. Will be filled with State() objects*

*# to keep track of the path through the grid.*

came\_from = [[[0 **for** x **in** range(grid[0])] **for** y **in** range(len(grid))]

**for** cell **in** range(NUM\_THETA\_CELLS)]

*# Create new state object to start the search with.*

x = start.x

y = start.y

theta = start.theta

g = 0

f = heuristic(start.x, start.y, goal)

state = State(x, y, theta, 0, f)

opened.append(state)

*# The range from 0 to 2pi has been discretized into NUM\_THETA\_CELLS cells.*

*# Here, theta\_to\_stack\_number returns the cell that theta belongs to.*

*# Smaller thetas (close to 0 when normalized into the range from 0 to*

*# 2pi) have lower stack numbers, and larger thetas (close to 2pi when*

*# normalized) have larger stack numbers.*

stack\_num = theta\_to\_stack\_number(state.theta)

closed[stack\_num][index(state.x)][index(state.y)] = 1

*# Store our starting state. For other states, we will store the previous*

*# state in the path, but the starting state has no previous.*

came\_from[stack\_num][index(state.x)][index(state.y)] = state

*# While there are still states to explore:*

**while** opened:

*# Sort the states by f-value and start search using the state with the*

*# lowest f-value. This is crucial to the A\* algorithm; the f-value*

*# improves search efficiency by indicating where to look first.*

opened.sort(key=**lambda** state:state.f)

current = opened.pop(0)

*# Check if the x and y coordinates are in the same grid cell*

*# as the goal. (Note: The idx function returns the grid index for*

*# a given coordinate.)*

**if** (idx(current.x) == goal[0]) **and** (idx(current.y) == goal.y):

*# If so, the trajectory has reached the goal.*

**return** path

*# Otherwise, expand the current state to get a list of possible*

*# next states.*

next\_states = expand(current, goal)

**for** next\_s **in** next\_states:

*# If we have expanded outside the grid, skip this next\_s.*

**if** next\_s **is** **not** **in** the grid:

**continue**

*# Otherwise, check that we haven't already visited this cell and*

*# that there is not an obstacle in the grid there.*

stack\_num = theta\_to\_stack\_number(next\_s.theta)

**if** closed[stack\_num][idx(next\_s.x)][idx(next\_s.y)] == 0

**and** grid[idx(next\_s.x)][idx(next\_s.y)] == 0:

*# The state can be added to the opened stack.*

opened.append(next\_s)

*# The stack\_number, idx(next\_s.x), idx(next\_s.y) tuple*

*# has now been visited, so it can be closed.*

closed[stack\_num][idx(next\_s.x)][idx(next\_s.y)] = 1

*# The next\_s came from the current state, and is recorded.*

came\_from[stack\_num][idx(next\_s.x)][idx(next\_s.y)] = current

# Trajectory Matching

Below you will see s vs. t and d vs. t graphs for four different trajectories (labeled Option A, B, C, and D). Match each set of graphs to the corresponding verbal description of the trajectory.

Chart

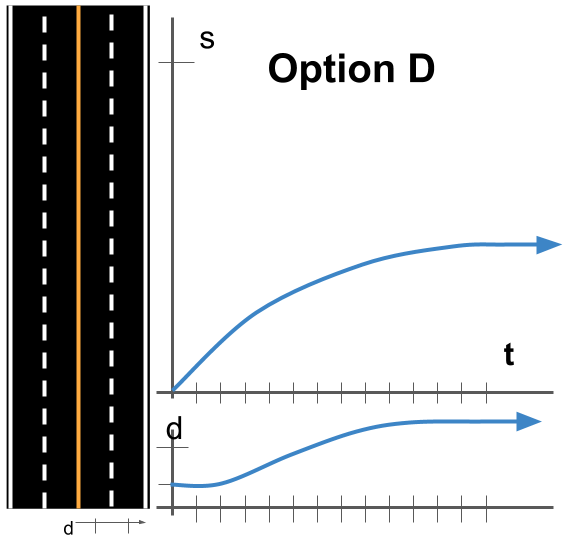
Description automatically generated

Chart

Description automatically generated

Chart

Description automatically generated



**Derivation Details**

In the previous videos we have:

1. Stated that Jerk Minimizing Polynomials are degree 5 polynomials with 6 tunable parameters.
2. Given an overview of how to calculate the coefficients for such a polynomial given the boundary conditions [s\_i, \dot{s}\_i, \ddot{s}\_i][*si*​,*s*˙*i*​,*s*¨*i*​] and [s\_f, \dot{s}\_f, \ddot{s}\_f][*sf*​,*s*˙*f*​,*s*¨*f*​] (or the corresponding d coordinates).

# Implement a Quintic Polynomial Solver

In this exercise you will implement a quintic polynomial solver. This will let you take boundary conditions as input and generate a polynomial trajectory which matches those conditions with minimal jerk.

### Inputs

Your solver will take three inputs.

1. start - [s\_i, \dot{s\_i}, \ddot{s\_i}][*si*​,*si*​˙​,*si*​¨​]
2. end - [s\_f, \dot{s\_f}, \ddot{s\_f}][*sf*​,*sf*​˙​,*sf*​¨​]
3. T - the duration of maneuver in seconds.

### Instructions

1. Implement the JMT(start, end, T) function in main.cpp
2. Hit Test Run and see if you're correct!

### Tips

Remember, you are solving a system of equations: matrices will be helpful! The Eigen library used from Sensor Fusion is included.

The equations for position, velocity, and acceleration are given by:

s(t) = s\_i + \dot{s\_i}t + \frac{\ddot{s\_i}}{2}t^2 + \alpha\_3t^3 + \alpha\_4t^4 + \alpha\_5t^5*s*(*t*)=*si*​+*si*​˙​*t*+2*si*​¨​​*t*2+*α*3​*t*3+*α*4​*t*4+*α*5​*t*5

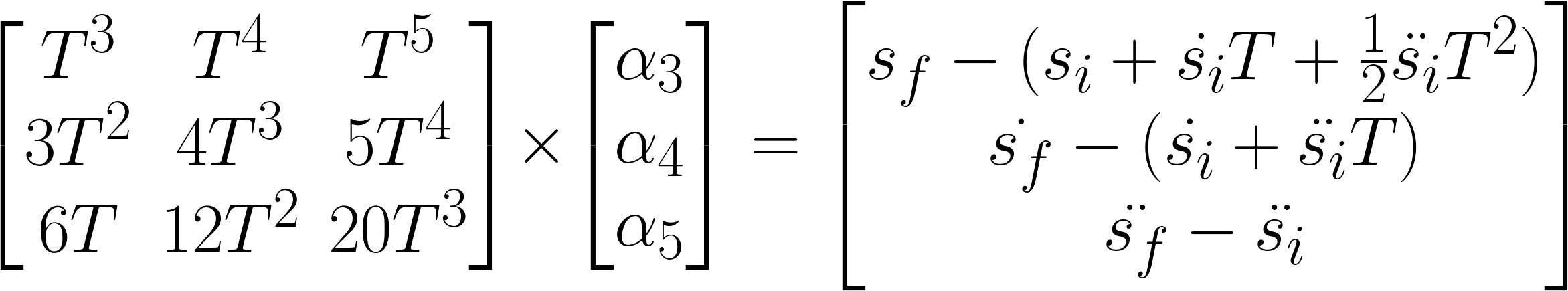
\dot{s}(t) = \dot{s\_i} + \ddot{s\_i}t + 3 \alpha\_3t^2 + 4\alpha\_4t^3 + 5\alpha\_5t^4*s*˙(*t*)=*si*​˙​+*si*​¨​*t*+3*α*3​*t*2+4*α*4​*t*3+5*α*5​*t*4

\ddot{s}(t) = \ddot{s\_i} + 6 \alpha\_3t + 12\alpha\_4t^2 + 20\alpha\_5t^3*s*¨(*t*)=*si*​¨​+6*α*3​*t*+12*α*4​*t*2+20*α*5​*t*3

and if you evaluate these at t=0*t*=0 you find the first three coeffecients of your JMT are:

[\alpha\_0, \alpha\_1, \alpha\_2] = [s\_i, \dot{s\_i}, \frac{1}{2}\ddot{s\_i}][*α*0​,*α*1​,*α*2​]=[*si*​,*si*​˙​,21​*si*​¨​]

and you can get the last three coefficients by evaluating these equations at t = T*t*=*T*. When you carry out the math and write the problem in matrix form you get the following:



All these quantities are known except for \alpha\_3, \alpha\_4, \alpha\_5*α*3​,*α*4​,*α*5​

# Polynomial Trajectory Generation

If you are interested in learning more about PTG, I've included a link to a paper (below) titled "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame". It is short and discusses some interesting (and potentially useful) topics like:

1. Cost Functions.
2. Differences between high speed and low speed trajectory generation.
3. Implementation of specific maneuvers relevant to highway driving like following, merging, and velocity keeping.
4. How to combining lateral and longitudinal trajectories.
5. A derivation of the transformation from Frenet coordinates to global coordinates (in the appendix).

#### **Supporting Materials**

[**Optimal Trajectory Generation For Dynamic Street Scenarios In A Frenet Frame**](http://video.udacity-data.com.s3.amazonaws.com/topher/2017/July/595fd482_werling-optimal-trajectory-generation-for-dynamic-street-scenarios-in-a-frenet-frame/werling-optimal-trajectory-generation-for-dynamic-street-scenarios-in-a-frenet-frame.pdf)

# Polynomial Playground (making PTG work)

Before you begin the final project we'd like to give you a chance to play around with cost function design and weight tweaking. In the Python code you will be provided, you will have a working Polynomial Trajectory Generator. You can see it in action by doing the following:

### Getting Started

1. Download the project code by clicking **TrajectoryExercise2** at the bottom of this page.
2. From the project's directory, run python evaluate\_ptg.py. You should see a plot similar to the one below. This plot shows the s (x-axis) and d(y-axis) trajectories followed by a vehicle in traffic (red) and a self driving car (blue).

Chart, scatter chart

Description automatically generated

In this situation, the self driving car was trying to get behind the target vehicle, but the cost functions it was using weren't weighted appropriately and so it didn't behave as expected.

### Fixing the Problem(s)

There are 5 files in the provided code. You'll probably want to start by modifying cost function weights in ptg.py but may also want to add cost functions of your own. As a bonus challenge try implementing this same code in C++.

**File Descriptions**

1. ptg.py - The primary code for generating a polynomial trajectory for some constraints. This is also where weights are assigned to cost functions. Adjusting these weights (and possibly adding new cost functions), can have a big effect on vehicle behavior.
2. cost\_functions.py - This file contains many cost functions which are used in ptg.py when selecting the best trajectory. Some cost functions aren't yet implemented...
3. evaluate\_ptg.py - This file sets a start state, goal, and traffic conditions and runs the PTG code. Feel free to modify the goal, add traffic, etc... to test your vehicle's trajectory generation ability.
4. constants.py - constants like speed limit, vehicle size, etc...
5. helpers.py - helper functions used by other files.

#### **Supporting Materials**

[**Trajectoryexercise2**](http://video.udacity-data.com.s3.amazonaws.com/topher/2017/October/59d5bfcb_trajectoryexercise2/trajectoryexercise2.zip)

[**Trajectoryexercise2\_python3**](https://d17h27t6h515a5.cloudfront.net/topher/2017/December/5a3ea459_trajectoryexercise2-python3/trajectoryexercise2-python3.zip)

## Additional Resources on Path Planning

Nice work reaching the end of the path planning content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to learn by doing - if you find a paper that really excites you, try to implement it (or even something better) yourself!

##### Optional Reading

All of these are completely optional reading - you could spend hours reading through the entirety of these! We suggest moving onto the project first so you have what you’ve learned fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well including their Abstract section, which summarizes the paper.

### Indoors

[Intention-Net: Integrating Planning and Deep Learning for Goal-Directed Autonomous Navigation](https://arxiv.org/abs/1710.05627) by S. W. Gao, et. al.

***Abstract:****How can a delivery robot navigate reliably to a destination in a new office building, with minimal prior information? To tackle this challenge, this paper introduces a two-level hierarchical approach, which integrates model-free deep learning and model-based path planning. At the low level, a neural-network motion controller, called the intention-net, is trained end-to-end to provide robust local navigation. The intention-net maps images from a single monocular camera and "intentions" directly to robot controls. At the high level, a path planner uses a crude map, e.g., a 2-D floor plan, to compute a path from the robot's current location to the goal. The planned path provides intentions to the intention-net. Preliminary experiments suggest that the learned motion controller is robust against perceptual uncertainty and by integrating with a path planner, it generalizes effectively to new environments and goals.*

### City Navigation

[Learning to Navigate in Cities Without a Map](https://arxiv.org/abs/1804.00168) by P. Mirowski, et. al.

***Abstract:****Navigating through unstructured environments is a basic capability of intelligent creatures, and thus is of fundamental interest in the study and development of artificial intelligence. Long-range navigation is a complex cognitive task that relies on developing an internal representation of space, grounded by recognizable landmarks and robust visual processing, that can simultaneously support continuous self-localization ("I am here") and a representation of the goal ("I am going there"). Building upon recent research that applies deep reinforcement learning to maze navigation problems, we present an end-to-end deep reinforcement learning approach that can be applied on a city scale. [...] We present an interactive navigation environment that uses Google StreetView for its photographic content and worldwide coverage, and demonstrate that our learning method allows agents to learn to navigate multiple cities and to traverse to target destinations that may be kilometers away. [...]*

### Intersections

[A Look at Motion Planning for Autonomous Vehicles at an Intersection](https://arxiv.org/abs/1806.07834) by S. Krishnan, et. al.

***Abstract:****Autonomous Vehicles are currently being tested in a variety of scenarios. As we move towards Autonomous Vehicles, how should intersections look? To answer that question, we break down an intersection management into the different conundrums and scenarios involved in the trajectory planning and current approaches to solve them. Then, a brief analysis of current works in autonomous intersection is conducted. With a critical eye, we try to delve into the discrepancies of existing solutions while presenting some critical and important factors that have been addressed. Furthermore, open issues that have to be addressed are also emphasized. We also try to answer the question of how to benchmark intersection management algorithms by providing some factors that impact autonomous navigation at intersection.*

### Planning in Traffic with Deep Reinforcement Learning

[DeepTraffic: Crowdsourced Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation](https://arxiv.org/abs/1801.02805) by L. Fridman, J. Terwilliger and B. Jenik

***Abstract:****We present a traffic simulation named DeepTraffic where the planning systems for a subset of the vehicles are handled by a neural network as part of a model-free, off-policy reinforcement learning process. The primary goal of DeepTraffic is to make the hands-on study of deep reinforcement learning accessible to thousands of students, educators, and researchers in order to inspire and fuel the exploration and evaluation of deep Q-learning network variants and hyperparameter configurations through large-scale, open competition. This paper investigates the crowd-sourced hyperparameter tuning of the policy network that resulted from the first iteration of the DeepTraffic competition where thousands of participants actively searched through the hyperparameter space.*