PART A

Q1. TRAINING SET(S) AND TEST SET FORMULATION

Q1(a). LEARNING AIM:

The chosen learning aim is to tune the parameters of a dual-PID controller to follow the groundtruth trajectory from ds1 (used in hw0). The duality of the PID controller comes from the six terms included-- the proportional, integral, and derivative terms for the linear and angular velocities, respectively. The robot used is the two-wheel mobile robot (used in hw0 and hw1), which utilizes a differential-drive model in conjunction with the dual-PID controls to follow a trajectory in a point-wise manner.

The motion model is mathematically the same used in hw1, though it is worth highlighting the implementation of the PID control and changes for the GA. Illustrated in the sketch below, every point in the trajectory that the GA is intended to tune the robot to follow has a number of sub-cycles. These sub-cycles are the most computationally intensive (read: lengthy) portion of the code, and so the inclusion of a threshold on total sub-cycles per trajectory step (**ct_treshold**) as well as a **cutoff** were employed to limit the time spend on populations with PID values that diverge. The ct_threshold will cut off the sub-cycles from trying to recover, and thus must be chosen large enough to allow small deviations to recover while also large enough to prevent large, divergent deviations from wandering. The cutoff helps with the latter issue, ending the sub-cycle if the trajectory is too far away from the intended target. Different values for the ct_threshold and cutoff were used for different training sets (e.g. sinusoidal vs. groundtruth trajectory), though the threshold was always kept between 0.05 and 0.1 for good (convergent) results.

```
------------------------------------------------------------------------------------------
motion_plan():
    trial_index = 0
    while flag != 1:
        calc_target_distance(x,y,theta,x',y') → r, ΔΦ
        one_step_controls() → vel_command, w_command
        <impose limits on acceleration>
        kinematic_model() → x, y, theta
        append(vehicle_traj, (x, y, theta)) → vehicle_traj
        integral_error_lin += r
        integral_error_ang += ΔΦ
        deriv_error_lin = r
        deriv_error_ang = ΔΦ
        trial_index += 1
        if (trial_index = ct_threshold) or ((x or y) < threshold) or ((x or y) > cutoff):
            flag = 1
    return vehicle_traj, v_curr, w_curr
------------------------------------------------------------------------------------------
```

In the sketch above, note also that integral and derivative error is updated at each step. This is fed into the PID controller, shown below in sketch format:

```
------------------------------------------------------------------------------------------
one_step_controls(r, ΔΦ, kp, ki, kd, deriv_error, integral_error):
    kpv*r + kiv*integral_error_lin + kdv*deriv_error_lin → vel_command
    kpw*ΔΦ + kiw*integral_error_ang + kdw*deriv_error_ang → w_command
    return vel_command, w_command
------------------------------------------------------------------------------------------
```

Mathematically, the linear and angular PID controllers can be respectively represented as:

$$u_{linear}(t) = k_{p,lin} * e(t) + k_{i,lin} * \int_0^t e(t)dt + k_{d,lin} * \frac{d}{dt}e(t)$$

$$u_{angular}(t) = k_{p,ang} * e(t) + + k_{i,ang} * \int_0^t e(t)dt + k_{d,ang} * \frac{d}{dt}e(t)$$

…with the terms represented in code and sketches as [kpv, kpw, kiv, kiw, kdv, kdw].

Q1(b). DATASET:

For Q1, the script *generate_ds.py* was employed. This script has three functions:

1. *linear_trajectory(start_loc, distance, variation, length)*
   1. this function takes a starting (x,y) tuple, a variation term, and a length (defines the number of points in the dataset). The chosen length was 1000 points.
   2. the output from this function is a series of linear paths at random angles. For 0 variation, the result is an entirely straight-line trajectory. For > 0 variation, the output is a series of linear components that begin to resemble a random walk.

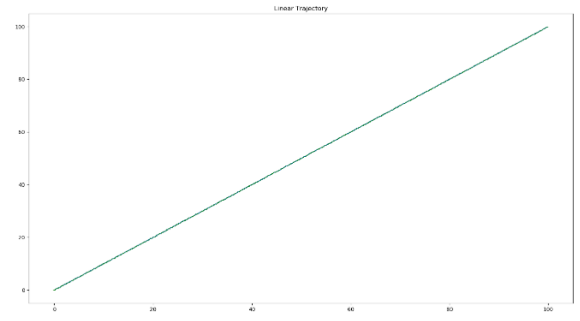


Fig. 1: Small amount of noise (+/- 0.05) linear trajectory.

2. *sin_trajectory(start_loc, distance, variation, length)*
   1. this function takes a starting (x,y) tuple, a variation term, and a length (defines the number of points in the dataset). The chosen length was 1000 points.
   2. the output from this function is a series of points in a sinusoidal pattern. For 0 variation, the result is a standard sine wave trajectory. For > 0 variation, the sine wave will have variation in: (1) amplitude, (2) frequency, and (3) smoothness.

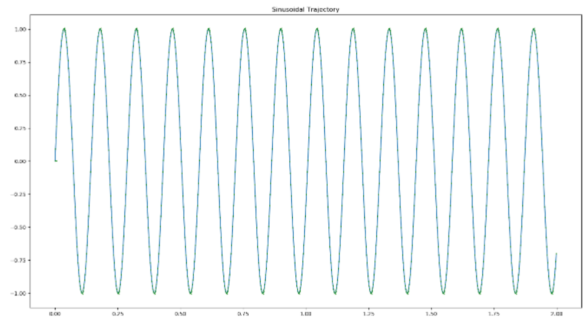


Fig. 2: Small amount of noise (+/- 0.05) sinusoidal trajectory.

3. *ds1_training_and_test_data(groundtruth_ds, percent_total)*
   1. this function takes in the ds1 groundtruth dataset as well as a total percentage (decimal). The total percentage defines the cutoff point between where the training data and test data sets will be constructed. For this assignment, I chose 0.7, resulting in 70% of the ds1 dataset being used for training and testing on the remaining 30%.
   2. the resultant training and test datasets are of the form [x, y, theta].

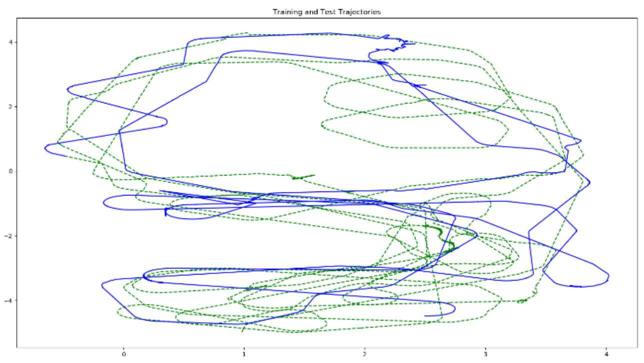


Fig. 3: ds1 groundtruth trajectory; Green is first 70% (training set), Blue is remaining 30% (test set).

Q2. GENETIC ALGORITHM IMPLEMENTATION v1

Q2(a). GENETIC ALGORITHM

A genetic algorithm (GA) is a machine learning technique inspired by the evolutionary process in biology. The tunable parameters of the learning aim are considered **genes**. For this assignment, the genes in question would be any the six controller terms for the linear and angular PID controllers: [kpv, kpw, kiv, kiw, kdv, kdw]. Because these values are floats, I chose a simple value-representation for the parameters (other popular parameter representations include binary and hexadecimal representation, which can facilitate simple crossover/mutation mechanics).

A collection of all the genes required for the learning aim is called a **chromosome** but is often referred to as an individual member of the population. Again, for this assignment one chromosome/individual would be a six randomly chosen PID tuning parameters in an array. For conventional GA implementations, a chromosome will take the form of a 1x(number of parameters) array.

For each cycle of the GA we will have (population size) chromosomes, resulting in the **population** array of size (population size)x((number of parameters)+1). The +1 on the array's width is sometimes used (and is used in my GA implementation) to store fitness values directly in the population array. Setting up the initial population is fairly important, as it requires a number of design decisions including:

1. population size
    1. a smaller population size will result in faster runtime, but typically will come at the cost of slower (if any) convergence.
    2. This is because the number of exploratory values distributed across the population will decrease, resulting in less chances for crossover and mutation (discussed below).
2. initial mean values (IMV)
    1. these are the values that each parameter in the initial population is assigned prior to the addition of random values.
    2. This is an exceptionally important parameter in that a rough estimate of 'good values' will result in significantly shorter runtime than randomly chosen bad values. Essentially, the initial mean values determine the starting point for the GA to explore
3. random distribution (RD)
    1. the random distribution from which you pull values to add (or subtract) from the initial mean values will determine the initial spread of chromosomes.
    2. A smaller spread can be beneficial if your initial mean values are already performing well, but a larger spread (especially in conjunction with a large population size) can capture a good deal of variation that the programmer might otherwise overlook.

The initialization for the population list is shown below:

```
------------------------------------------------------------------------------------------
initial_pop():
        zeros(popsize, param_length + 1) → pop_list
        for i in range(0, size(pop_list)):
```

```
        for j in range(0, param_length):
            pop_list[i, j] = IMV + random.uniform(-RD, RD)
    return pop_list
```

After the population list is initialized (corresponding to the first **generation**), we begin the GA's cyclic portion. At its core, the GA takes the following format:

```
initial_pop() → pop_list (#gen X #pop array)
for i = [0,#gen]:
    training_fn(pop_list, training_set) → pop_list
    new_gen(pop_list) → pop_list
```

The GA has a number of cycle equal to the number of generations (a design decision), and during each cycle it updates the population with fitness scores via *training_fn()*, wherein each chromosome is evaluated over a training set and a fitness score is returned. The cycle then takes the population (with fitness scores) and creates the next generation of chromosomes via *new_gen()*.

Q2(b). FITNESS FUNCTION

The fitness function employed was standard root mean-squared error (RMSE). Whereas fitness scores are most often meant to be maximized in a GA (to line up with biology's notion of 'greatest fitness in an individual'), in this implementation the raw RMSE is used and thus the GA trends towards minimizing the fitness score *F(n)*:

$$F(n) = RMSE \stackrel{\text{def}}{=} \sqrt{\frac{\sum_{i=1}^{n}(Prediction_i - Actual_i)^2}{n}}$$

The training function, outlined below, calls *motion_plan()* (which includes the motion model for the robot and the IK PID controller) to establish a step in the trajectory given an individual's genes, and then calculates a running fitness value (*calc_fitness()* calculates differences at each timestep, *fitness_metric()* calculates the RMSE for all differences for a single individual) for that individual based on the RMSE between the groundtruth trajectory and the motion model's trajectory. For each individual, these two steps run the length of the training set. At the end, the population list with fitness scores is returned:

```
training_fn(pop_list, training_set):
    for j = [0, size(pop_list[:,0])]
        fitness_score_list = (0,0,0)
        for i in range(0, size(training_set)):
            motion_plan() → trajectory, v_curr, w_curr
            calc_fitness() → fitness_score_list
        fitness_metric(fitness_score_list) → pop_list[j,6]
    return pop_list
```

The function *calc_fitness()* takes in the generated pose as well as the groundtruth pose and compares them, adding their difference to the array 'fitness_score_list.'

```
calc_fitness(x, x', y, y', theta, theta', fitness_score_list):
    abs(x – x') = dx
    abs(y – y') = dy
    abs(theta – theta') = dtheta
    fitness_score_list = np.append(fitness_score_list, (dx,dy,dtheta))
    return fitness_score_list
```

Once the differences are computed for every step in the training set, the RMSE of differences are calculate for the entire run, and the fitness score is returned.

```
-----------------------------------------------------------------------------------------
fitness_metric(fitness_score_list):
        lin_norm_x = 1
        lin_norm_y = 1
        angular_norm = 1
        unnormalized_fitness = 0
        for i = [0, size(fitness_score_list[:,0])]:
                unnormalized_fitness + fitness_score_list[i,0]/lin_norm_x
                                     + fitness_score_list[i,1]/lin_norm_y
                                     + fitness_score_list[i,2]/angular_norm
                                    → unnormalized_fitness
        sqrt(unnormalized_fitness/size(fitness_score_list[:,0])) → fitness_score
        return fitness_score
-----------------------------------------------------------------------------------------
```

Q2(c). MATING FOR NEW GENERATIONS

After fitness scores are computed, the GA requires the creation of **children** (next-generation chromosomes) from the **parents** (current-generation chromosomes). There are a number of design decisions that may be taken into account here:

1. culling percentage
    1. This determines how many parents are left in the population. For this assignment, I chose to keep half of the parents each step. The ½ population cull (i.e. the deletion of the least fit half of the population) is a fairly standard convention that speeds up successive fitness score computation (since all subsequent cycles of the GA will only have to compute for ½ the population size).
    2. The parents chosen will be the ones deemed fittest (via fitness score). In my implementation, fitness values were minima.
    3. The remaining parents will be combined via crossover and mutation (the two forms of mating employed in my implementation; there are many others and variations thereof). Every two children will be the product of two parents.
        1. Thus, in choosing to keep ½ of the parents, we ensure that the remaining half of the population (the new half, i.e. children) will each be the product of two unique parents.
            1. That is to say-- no parent's genes will be used to mate more than once in this formulation.
2. The mutation rate (mut_rat in the algorithm sketch below)
    1. This determines how much any gene can be randomly changed (via the addition or subtraction of a float in range [-mut_rat, +mut_rat]).

```
-----------------------------------------------------------------------------------------
new_gen(pop_list):
        fittest = (0, 0, 0)
        for i = [0, size(pop_list)/2]:
                append(fittest, min(pop_list)) → fittest
                delete(population, min(pop_list)) → population
        population = fittest
        for i = [0, size(pop_list)/4]:
                crossover(fittest[i,:], fittest[-i,:]) → child1, child2
                mutation(child1) → child1
                mutation(child2) → child2
                append(population, child1) → population
                append(population, child2) → population
        population[size(pop_size/2):end, 6] = 0.0
        return population
-----------------------------------------------------------------------------------------
```

The crossover component takes in two parents, chooses a random integer within the range of parameters (chromosome length), and then splits each parent at that index. The splits are shuffled and concatenated, creating two children that are each taking a segment of genes directly from each parent:

```
-----------------------------------------------------------------------------------------
crossover(parent1, parent2):
    random.randint(0, parameter_length) → crossover_index
    concatenate(parent1[:crossover_index], parent2[crossover_index:]) → child1
    concatenate(parent2[:crossover_index], parent1[crossover_index:]) → child2
    return child1, child2
-----------------------------------------------------------------------------------------
```

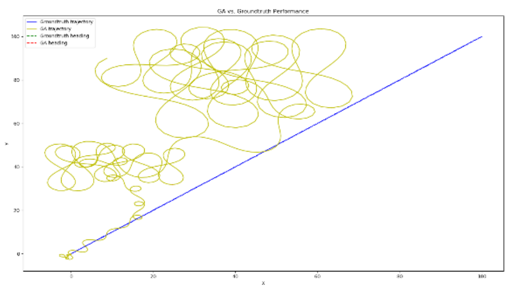
Mutation is a process which introduces further randomness to the exploration of different parameter values between generations. In my implementation, *mutation()* takes a child as argument, chooses a random number of mutations within the gene length (i.e. it is possible for 0 or 6 mutations to occur on each child), and then randomly chooses which gene is to be mutated as well as by how much. One feature of this implementation is that a single gene can undergo multiple mutations between generations. Following the process, the child is returned.

```
-----------------------------------------------------------------------------------------
mutation(child):
    random.randint(0, parameter_length) → num_mutations
    for i = [0, num_mutations]:
        random.randint(0, parameter_length) → mutation_index
        child[mutation_index] + random.uniform(0, mut_rate) → child[mutation_index]
    return child
-----------------------------------------------------------------------------------------
```

At the end of this process of crossovers and mutations, the population will maintain its size and be composed of ½ the past generation's fittest individuals and ½ new children created by the combination and randomization of that fittest ½. This ensures that the best fitness values throughout the entirety of the GA will be maintained between generations, as the top ½ is never deleted.

Q3(d). LINEAR TRAJECTORY

The linear trajectory had issues when the angular parameters would get larger than their linear counterparts. Thus, the better GA-tuned dual-PID for the linear example had relatively low angular parameter values and large linear parameter values.



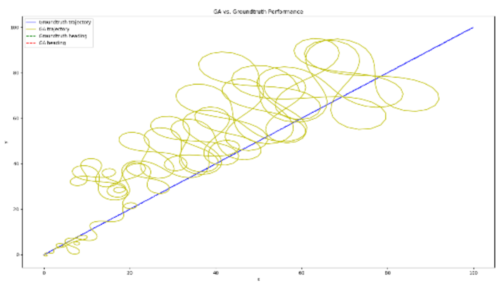
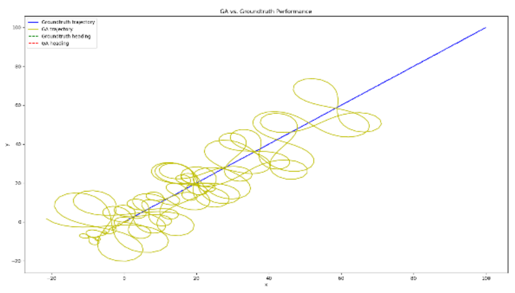


Fig. 4, 5: linear trajectory for bad initial population genes [0.01,0.8,0.0,0.0,0.1,0.1]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (75 generations, population size of 40). Note that the GA did manage to 'pull' the values closer to the trajectory despite the obvious issues in linear motion. This is an example of initial population values being poor and limiting search space.



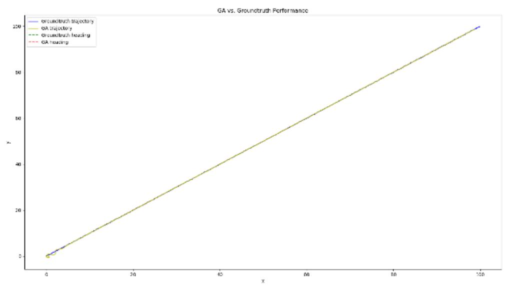
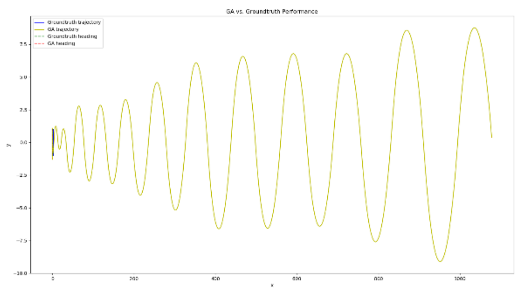


Fig. 6, 7: linear trajectory for good initial population genes [0.5,0.5,0.5,0.5,0.5,0.5]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (75 generations, population size of 40). Note that the GA was able to track the noisy linear trajectory quite well. Here the initial population values were 'fairly good' and could be tuned over the GA to be quite good.

Q3(e). SINUSOIDAL TRAJECTORY

In opposition to the linear trajectory, angular parameters had relatively high values and low linear parameter values for good fitness.



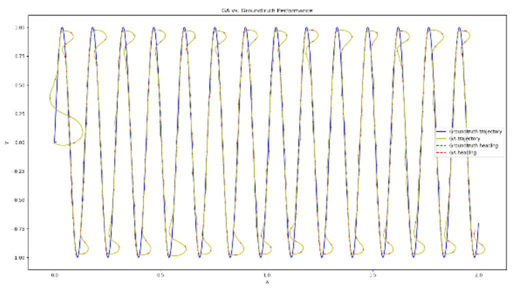


Fig. 8, 9: noiseless sinusoidal trajectory for good initial population genes [0.5,0.5,0.5,0.5,0.5,0.5]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (100 generations, population size of 40). Note that the GA was able to recover from issues in increasing amplitude over time as well as with distance scaling.

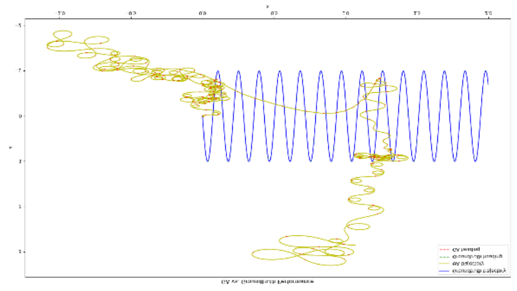
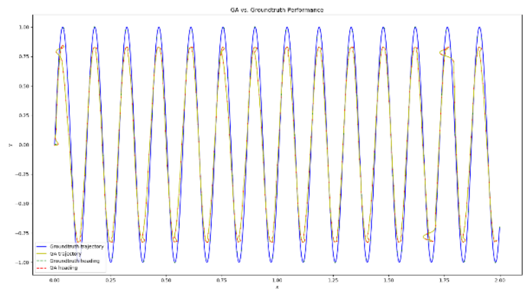



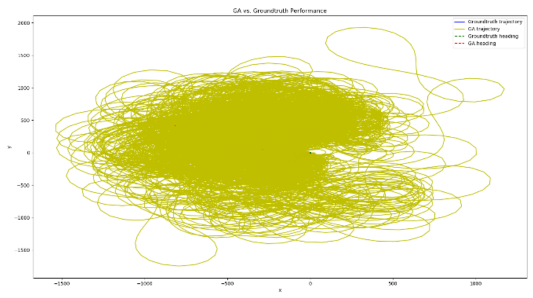
Fig. 10, 11: noisy sinusoidal trajectory for good initial population genes [0.5,0.5,0.5,0.5,0.5,0.5]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (100 generations, population size of 40). Note that the GA was able to recover from the extremely divergent values and to fit to the noisy sine wave almost exactly. NOTE: there are still issues in amplitude, as the fittest member does not ever reach the maximum or minimum values of the sine wave's y-values.

PART B

Q3. GENETIC ALGORITHM IMPLEMENTATION v2

The GA trained over the first 70% of the groundtruth trajectory from dataset1 (hw0, hw1) and was subsequently analyzed on the remaining 30%. The GA was tested for a wide range of values (best values highlighted):

1. Population size: [10, 20, 25, 40, 75, 150]
2. Number of generations: [5, 10, 20, 25, 100, 1500]
3. Mutation rate (mut_rate): [0.0, 0.05, 0.1, 1.0, 5.0, 10.0]
4. ct_thresh: [2, 5, 10, 25, 100]
5. threshold: [0.01, 0.05, 0.1]
6. cutoff: [2, 5, 10, 100]



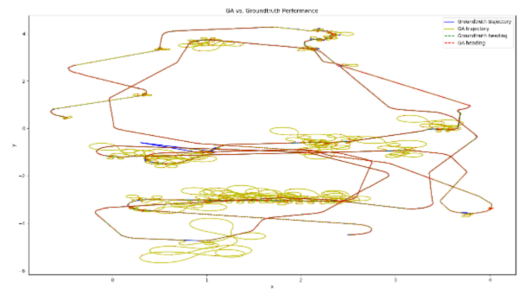
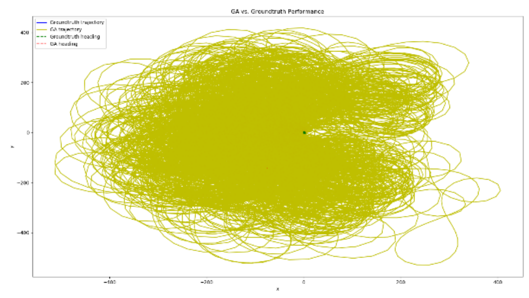
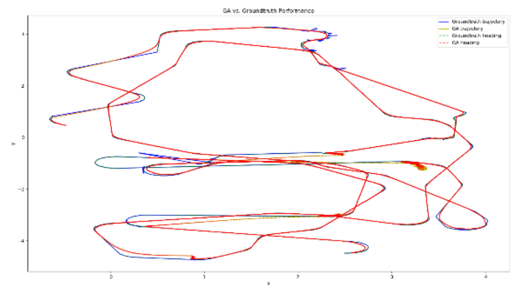
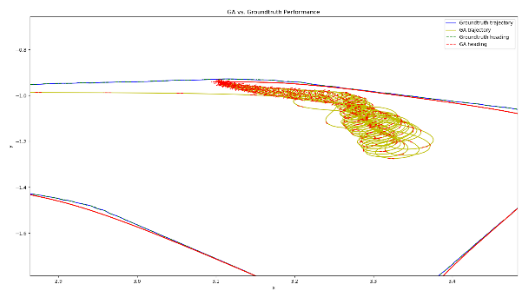


Fig. 12, 13: training set trajectory for good initial population genes [0.01,0.8,0.0,0.0,0.1,0.1]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (10 generations, population size of 20). Note that the GA was able to recover from the extremely divergent values and to fit to the test trajectory almost exactly. NOTE: there were issues through where the GA-tuned dual-PID velocities would go off-track at small inconsistencies in the initial trajectory. At these points we see the swirling loops as the controller tries to recover, which it manages to do (but not quickly).







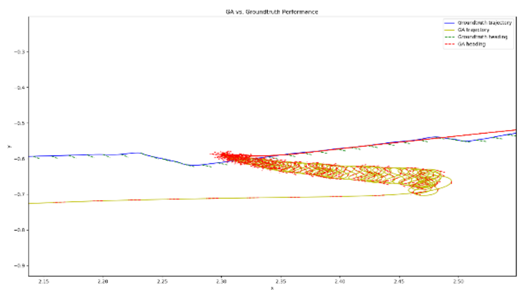
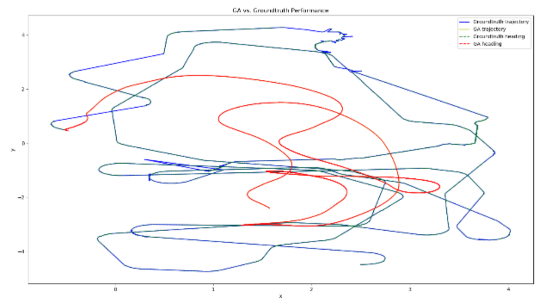


Fig. 14, 15, 16, 17: training set trajectory for good initial population genes [0.01,0.8,0.0,0.0,0.1,0.1]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (10 generations, population size of 25). Note that the GA was able to recover from the extremely divergent values and to fit to the noisy sine wave almost exactly. NOTE: the same issues appear in the form of the swirling oscillations when the path gets off track, but they are reduced in scope and recover much then in Fig. 12,13's GA run (see detail images in Fig. 16,17). This is surely due to the increased population, as all other instances are parameters and metaparameters are the same.



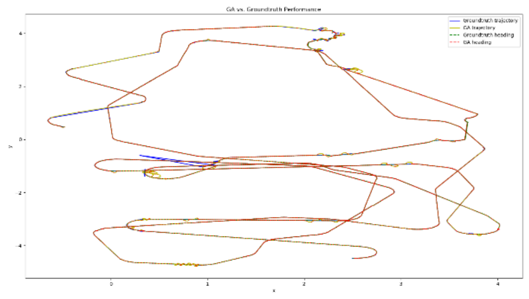
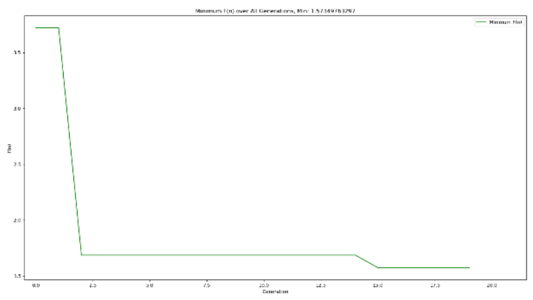


Fig. 18, 19: BEST test trajectory for good initial population genes [0.01,0.8,0.0,0.0,0.1,0.1]; Left is the fittest individual in the population of the first generation, right shows the fittest individual of the final generation (20 generations, population size of 20). Note that the GA was able to recover from the extremely divergent values and to fit to the noisy sine wave almost exactly. NOTE: there are still issues in a few small segments of the test set where the data rapidly oscillates (likely due to noise in the sensors when this dataset was initially recorded—the sharpness of those segments seem unlikely to reflect actual robot motion), but the GA-tuned dual-PID recovers quickly.



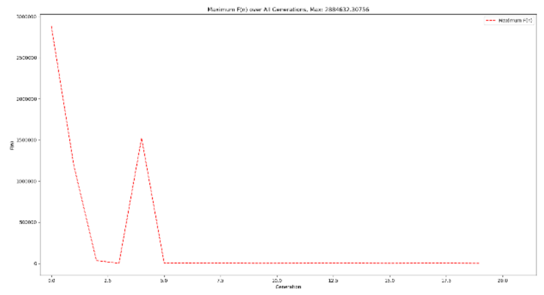


Fig. 20, 21: minimum and maximum for the GA run shown in Fig. 18,19; Left is the minimum (best) fitness score for each generation. Note the large initial drop-off and series of plateaus as the ½ cull preserves highest fitness. Plateaus are for generations where the best fitness score is not beaten. On the right is the maximum (worst) fitness score for each generation. Not the initial spikes followed by much smaller variation as the crossover step overtakes the mutation step by limiting the range of values for children's genes.

NOTE: for Fig. 12-21, 'good' initial population genes are good only for the training set; as evidenced in Fig. 4,5, those same initial population genes perform very poorly as starting points in the GA for a different trajectory.

DISCUSSION:

1. The major issue in this implementation is the runtime. The entire groundtruth trajectory is over 125,000 datapoints. Even throwing away the first ~5000 points where the robot is stationary (with some minimum variation due to sensor error), this still results in a very long run time checking every point in the trajectory.
    a. In future implementations, I would like to try to run the GA over smaller segments of the groundtruth trajectory (e.g. 10% per segment), then run a separate GA incorporating the fittest individuals from each 10% segment.
    b. Ideally a way to incrementally try different metaparameters would also be good, though due to the length of time the training set took to run I changed them by hand. Finding ideal initial population values and the range over which to vary those values was a tedious process of informed guess-and-check.
    c. The runtime for the 1000 points of the sinusoidal and linear trajectory was very fast, however.
2. The problem of 'when to stop a motion planning sub-cycle' discussed earlier was a large struggle in getting values to converge in a timely manner and warrants further exploration.

CITATIONS:

[1] Thrun, S., Burgard, W., & Fox, D. (2010). *Probabilistic robotics*. Cambridge, MA: MIT Press.
[2] Russell, S. J., & Norvig, P. (2018). *Artificial intelligence: a modern approach*. Noida, India: Pearson India Education Services Pvt. Ltd.
[3] "Introduction to Optimization with Genetic Algorithm", https://www.kdnuggets.com/2018/03/introduction-optimization-with-genetic-algorithm.html