Improving Mobile Robot Perception Using a Kalman Filter and Machine Learning

Kevin Robb
kevin@js-x.com

Engineering Physics Capstone 2020-2021
May 12, 2021

Contents

Abstract

The most essential aspect of any mobile robotics application is its perceptive ability, especially if it will run autonomously. A common method for using data from a suite of sensors to keep track of a robot's state is the Kalman Filter. This iterative process combines sensor fusion with constant feedback to keep an accurate internal model of the real world. The catch is that the filter must be setup and tuned relatively precisely to obtain the best functionality; if we could minimize this phase, it would be far easier to implement with a variety of applications. We propose an evolutionary approach to automatically find the best settings and initialization parameters to maximize the performance of the Kalman Filter. We find in this project that perception is equally improved by the filter with manual or evolutionary tuning, and becomes even better with a combination of the two.

1 Introduction

Perception and localization are important areas in the realm of robotics, and are absolutely necessary for any sort of autonomous or intelligent behavior. With the advent of self-driving cars, warehouse robots, floor cleaners, and many more direct robotics applications, it is becoming increasingly necessary for machines to be able to recognize details about their environment and act on that information to the greatest extent possible.

Robots, UAVs, and any other sort of intelligent equipment benefit from increased knowledge of the world, because an accurate and constantly updating internal world map allows extremely efficient behavior, and prevents actions that would be unsafe to themselves or to nearby humans. The best way to ensure that technological developments in these areas over the next several decades can take hold and keep pushing the field forward is to create and necessitate a baseline of safe functionality, which not only improves the performance of these devices, but keeps public opinion and lawmakers from pushing against further progress.

Shifting to the focus of this project, we deal with a simulated robot performing tasks associated with autonomous path planning in unknown, random environments. This can be thought of as a potential search and rescue or exploration application. To gain the best performance in this environment, it is very important for the robot to have as good an idea as possible about its exact position. To obtain this, we use a Kalman Filter (KF), an iterative sensor-fusion algorithm that keeps track of an internal model of the robot's state, and updates it with new measurements at each timestep. The KF has the potential to optimize the robot's performance, achieving the best localization possible given the environmental parameters; unfortunately, the KF is hard and tedious to precisely tune manually, and this tuning does not hold up to much change in the environment. To address this, we propose the use of machine learning, specifically evolutionary computation, to automatically tune the KF to dynamically changing environments without user intervention.

We are also interested in removing the complexity associated with the robot's motion policy, so in this project, we implement an extremely simple control structure in which the robot directly pursues the heading to its next goal waypoint. We are interested in using similar techniques as appear in this paper to attempt evolving control parameters instead of or in addition to the perception parameters that we focus on, but this will require continuation or further experimentation by others, as we did not get to it in the course of this project. To allow for this future expansion, we have setup the entire project to be very malleable in terms of what variables are part of the evolution aspect, and where they can be used throughout the ROS architecture.

1.1 Hypotheses

We hypothesize the following.

H_0: The agent's localization will be improved by the addition of the Kalman Filter.

H_1: The agent's localization will be further improved after tuning the KF with machine learning.

2 Project Components

This project has consisted of several modules working together and building on each other. This section details each of these modules.

2.1 Simulator

As cited in [5], we are using the Software Challenge Simulator created for Sooner Competitive Robotics in April 2020. We have received permission to use it for this project.

This simulator models a competition in which our team has competed in the past, the RoboGames RoboMagellan challenge. This involves sending an autonomous robot through terrain completely autonomously to hit a few waypoints, avoid obstacles, and ultimately arrive at the final destination. The robot uses Ackermann steering, and accepts a commanded velocity and heading. We can assume that these commands are transformed into individual motor controls by the firmware at a lower level of software than this project focuses on.

For any given run, we can specify a seed to determine the generation of the waypoints and obstacles, or we can leave it out to have the simulator choose a random seed every time it is launched. The simulator will write to a text file the results of a given run, including a score and the seed. A lower score is better. Many of the details of the simulator's operation and results generation are explained further in Appendix A.

The rest of our code relies on the fact that the simulator closes when the robot has either completed the course (reached the final waypoint) or timed out. We have added some more functionality that detects if the robot has driven very far out of bounds and kills the simulator process, keeping the program moving forwards and avoiding the wait for a timeout.

The goal of this project was to evolve optimal functionality in an environment with randomized obstacles, but for the most part we have worked to demonstrate performance simply reaching all waypoints quickly in an obstacle-free environment, and had plans to revisit reactive obstacle avoidance when satisfactory performance had been achieved. This decision is further explained at the end of the next subsection.

2.2 Sensor Suite

The simulator has the ability to publish data from several different sensors. The ones we are using are detailed herein.

The *Inertial Measurement Unit (IMU)* is a gyroscope, magentometer, and accelerometer all in one. It is able to collect and send a variety of data, including orientation data (i.e., robot pitch, roll, and yaw), dead-reckoning position data, acceleration, and temperature. We primarily use the orientation data to extract the robot's current global heading (yaw).

The *GPS* unit tracks the robot's current latitude and longitude, which we convert into a position in meters on our relative map.

The *encoders* track each motor's turning speed and distance, and publish the angular velocities of both the left and right drive wheels. The encoders could also be used for simple dead-reckoning localization, but we do not use them for this.

The *LiDAR* sensor uses laser light to collect distances to obstacles in a full circle around the robot. We convert the laserscan message into meters, and use it to identify ranges of our vision that contain obstructions.

The *bumper* emits a message containing "True" when the robot is in contact with an obstacle, and emits a follow-up message containing "False" when the robot comes out of contact with an obstacle.

The *Waypoints* service is a one-time message containing the GPS coordinates for all five waypoints in this run of the simulation. This includes the starting and goal waypoints, which are always at the same location, as well as the randomized three intermediary waypoints.

There are other sensors available, namely the camera that can be utilized for computer vision input, but we do not utilize any other sensors than those mentioned in this section.

2.3 Simple ROS Agent

The first module of this project is the main "robot" that interacts with the simulator. We used Robot Operating System (ROS Melodic) to create a simple agent which can make it to the goal most of the time. The main deliverable we need from this module is an agent which moves around in a non-random way, sends data to the KF, and receives and uses meaningful data from the KF to affect its behavior. We will add the KF in the following subsection, so for now an agent that can just follow a general path towards the end of the course is a solid benchmark performance. This will of course be improved with each later stage of the project. Even before the KF is added, this agent has several ROS nodes whose architecture is detailed in FIG. 1. A "node" refers simply to one code file in a ROS package which can publish messages and subscribe to those published by the other nodes.

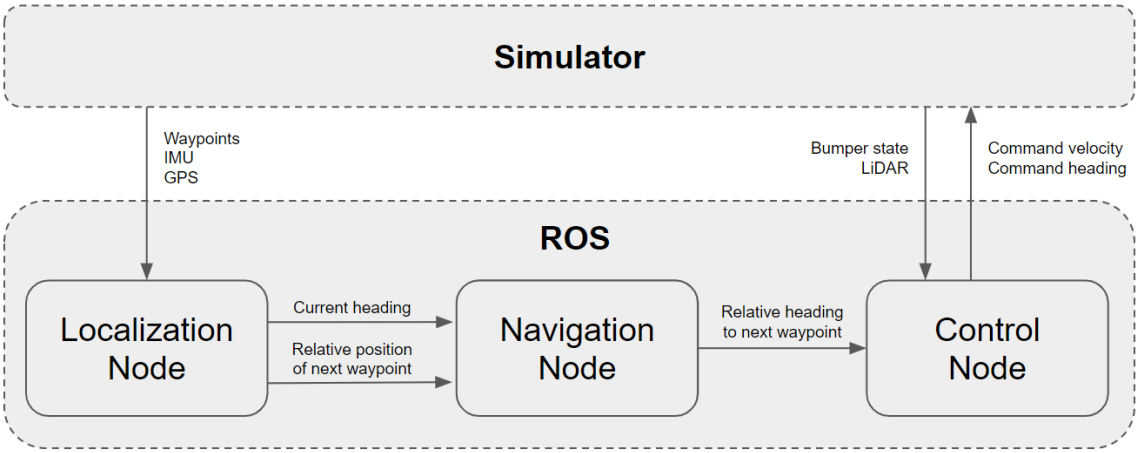

FIG. 1: The architecture for a simple agent with no intelligence who blindly pursues a direct heading to the next waypoint. Dashed boxes represent main modules, while solid boxes each signify a single script or node. Solid arrows denote values published and subscribed to.

This simple flow chart starts with sensor values read by the Localization Node that are parsed to determine the robot's current position in meters and the current heading (i.e., yaw). The Localization Node is also where we subscribe to the GPS coordinates of the three bonus waypoints and the final goal, so we use these two sets of information to define a new coordinate system in the robot's frame of reference, and we publish the relevant resulting information to be picked up by the Navigation Node. Since this is a simple agent with no complex path planning, the Navigation Node is rather straightforward, simply publishing the necessary heading to pursue the next waypoint in the robot's reference frame. We normalize all headings in degrees

to stay in the range $(0°, 360°)$. This heading is obtained by the Control Node, where it is used in Equations (1) and (2) to generate the commands that are sent back to the robot.

$$v = 5 \cdot \left(1 - \frac{|\theta|}{30}\right)^5 + \frac{1}{2} \tag{1}$$

$$\phi = \theta \cdot P = \theta \cdot \left\{ \begin{array}{ll} 0.3, & d \geq 6 \\ 0.6, & \text{else} \end{array} \right\} \tag{2}$$

where θ is the target heading (in degrees) published by the Navigation Node, and d is the relative distance in meters to the target waypoint.

We setup the Control Node in this way to modulate the robot's forward speed in order to drive as fast as possible when perfectly aligned, and to stay mostly in place while turning if its heading is way off. Our simple two-state P-controller allows the robot to turn faster when very close to a waypoint to help make sure it doesn't miss. The positive shift to v ensures the robot doesn't get stuck somewhere and keep commanding a velocity of 0.

If the simulator has obstacles enabled, the Control Node will occasionally stop following commands from the Navigation Node while avoiding or recovering from a collision. This avoidance is handled purely reactively; if the bumper is activated or the LiDAR detects an obstacle very close in our path, we stop, back up, and turn for a set amount of time. The direction to turn is based on which way brings us more in line with the waypoint we are targeting. After this "back up and turn" maneuver has finished, control is released back to the commands from the Navigation Node. The reorientation of the robot helps us to get around obstacles, and because our commanded heading at any given time is determined only by our current position (rather than any more intelligent plan or trajectory), we don't need to do anything else besides release control back to the Navigation Node in order to continue on our journey through the course. It is possible (and frequent) for this avoidance to trigger repeatedly on the same obstacle, but the agent almost always manages to make it past and finish the course.

The robot does take damage when colliding with obstacles, and if enough damage is taken, it "dies," ending the run. Referring back to our intention of creating a source of data for the rest of our project without regard to its actual scored performance, a situation that increases the likelihood that an agent will fail to finish the course goes against our goals. This is why we disabled the obstacles for the remainder of the project at this stage, at least while we are focusing on perception and not performance.

2.4 The Kalman Filter

A *Kalman Filter* is a moderately well-known method for combining sensor values and keeping an internal estimation for the true state over many timesteps. The general process involves predicting what the state will look like at the next timestep, performing a measurement, and comparing them in a smart way to update our estimations. This process is depicted in FIG. 2.

This iterative process can be thought of as a sort of "weighted average over time," where we weight the measurements and predictions differently based on our confidence in the accuracy of each. This can provide both rapid convergence to the truth, and quick recognition of mistakes and adjustment back on track. The KF is implemented alongside the previous architecture as an additional ROS node; the updated diagram is shown in FIG. 3.

For this project, the *state* is simply a column vector containing the x position, y position, and x and y velocities. These are all expressed in a global, right-handed coordinate system, assuming the robot is initialized at the origin facing along the positive x-axis. We assume a constant velocity model for simplicity, so we use basic Newtonian mechanics to predict updates to the position, and accept that our velocities will not be tracked extremely well. This is okay,

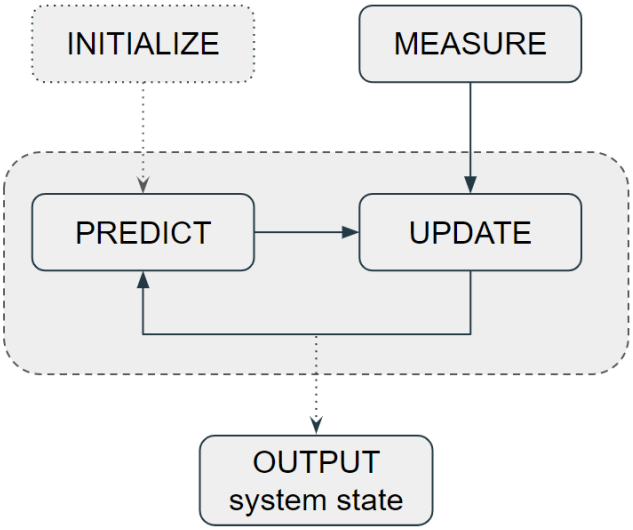

FIG. 2: After initializing with some guesses for measurements, all other steps happen during every timestep to produce a constantly updating "state" at the output.

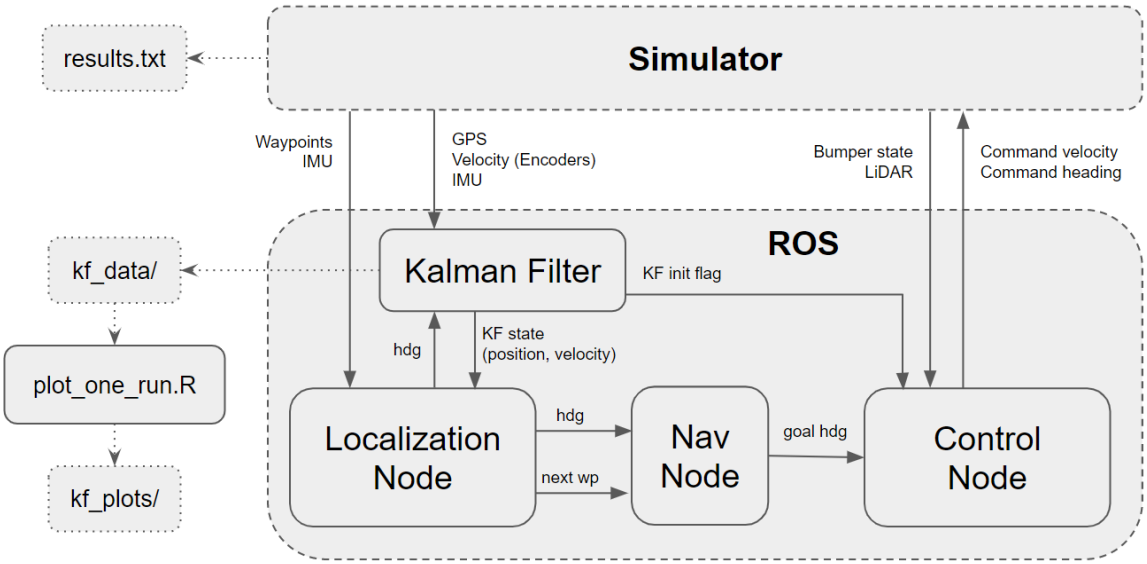

FIG. 3: The KF integrated with the existing agent's architecture. In addition to the previous notation, dotted boxes represent files or directories, and dotted arrows denote information transfer that occurs by writing to and reading from a file.

because we are only concerned with tracking the position, and the velocities serve their goal of helping us to compute that.

Our full derivation of the Kalman Filter is contained in Appendix B. The aspect most relevant to the rest of this paper is the following: there are several *hyperparameters* of the KF that must be manually tuned by the user for a particular application. For this step, we performed this tuning until the agent did acceptably well, and the next stage of this project will automatically optimize these values and allow for dynamic corrections to changing environments.

2.5 Evolutionary Computation

Evolutionary Computation (EC) is a form of machine learning that mimics a "survival of the fittest" selection approach. We form a generation of agents, each of which is defined by a unique set of values called its *genome*. For this project, the genome contains several hyperparameters

of the Kalman Filter, and could be easily expanded to include motion parameters or anything else used by the ROS architecture. We will describe the specific choice of genome in more detail in the next section.

With this choice of genome, we can evolve and optimize the perception abilities of the robot while keeping the simplistic motion model. Our project architecture with the EC included is shown in FIG. 4.

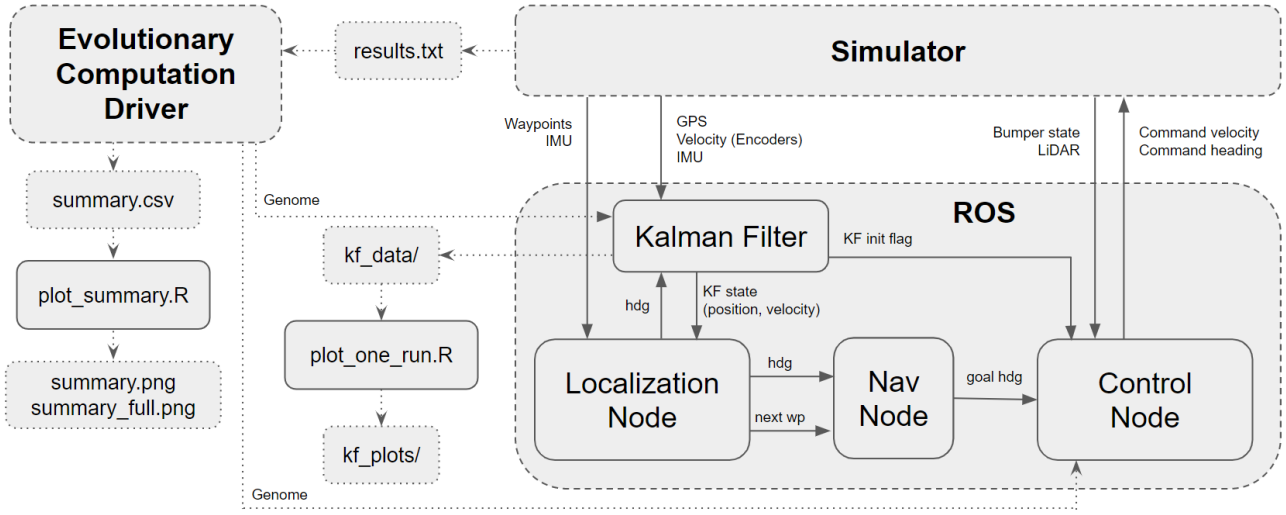

FIG. 4: The Evolutionary Computation module operating with the rest of our architecture to perform many runs of the simulation with different configuration values.

We begin by instantiating a set of agents with randomized genomes, and we perform a full run of the simulation for each one to record its score and other performance results. The agent is assigned a *cost* based on Equation (3). Because the most important job of the KF is to accurately track the true positions, our cost function will be computed using the differences between the KF state and the ground truth for x and y. We normalize each run by dividing out the number of timesteps taken to reach the goal. This gives us the cost function

$$C = \frac{1}{N}\sum_{i=0}^{N}\Big(|x_{i,state} - x_{i,true}| + |y_{i,state} - y_{i,true}|\Big) \tag{3}$$

where N is the total number of timesteps the simulation took to complete.

As a clarifying point, evolutionary computation and genetic algorithms usually have an evaluative metric called a *fitness function*. In our case, we want our metric to evolve downwards toward zero rather than upwards, so we refer to it as a cost function. It is essentially an arbitrary matter of semantics, so feel free to think of the "cost" as a fitness if it aids understanding.

After the simulation has been run for every agent in the generation, we select agents to reproduce with a linear weight function, where the most likely agent to be selected for reproduction is the one with the lowest cost. Agents are selected two at a time (allowing multiple selection) and we perform crossover and mutation on their genomes to form an agent for the new generation.

Crossover involves randomly selecting from which parent each gene will be copied. After being copied from a parent, each gene has a 5% chance to be mutated. A mutation consists of adding a random value selected from a Gaussian centered at 0 with standard deviation 0.01. All genes are capped afterwards to ensure they remain in the allowable $(0, 1)$ interval.

Once the new generation is formed, we repeat the process for as many generations as requested. Data for the KF in every run is saved and plotted as it was before, and evolutionary progress is saved in aggregate for a summary of the cost improvement over time.

3 Procedure

This project involved incrementally increasing the complexity of the overall framework with the addition of each component outlined in Section 2. The most fundamental layers were implemented with rather straightforward reasoning: the behavior should be as simple as possible while keeping reasonably consistent completion of the course. To this end, we use the extremely basic control policy described previously in Equations (1) and (2). Implementation follows fairly clearly from the descriptions in the previous section.

Keep in mind that the actual performance of the agent, in terms of time to finish the course, waypoints hit or missed, and overall score at the end, is separate from our actual evaluation of the project's success. The point is to have something to interact back-and-forth with the Kalman Filter. As long as something is moving in the environment and changing its position, velocity, and heading in a manner similar to an actual robot, we are happy. All of the data regarding the true state of these variables, as well as the measurements, predictions, and estimations used by the Kalman Filter, are saved each timestep to a data file. When a run concludes, this is automatically plotted using a custom R script.

For each agent, we plot each of its four KF state variables as a function of time, and create a map of x vs y showing the real path forged by the robot in the simulation. This track also shows the five waypoints, in purple if the robot succeeded in passing through them, and in yellow if the robot failed to do so. The first and last waypoint will always be visited, since they are required to begin and end the run. An example run is shown in FIG. 5; the actual quality of the results depicted in such a plot will be described in the next section.

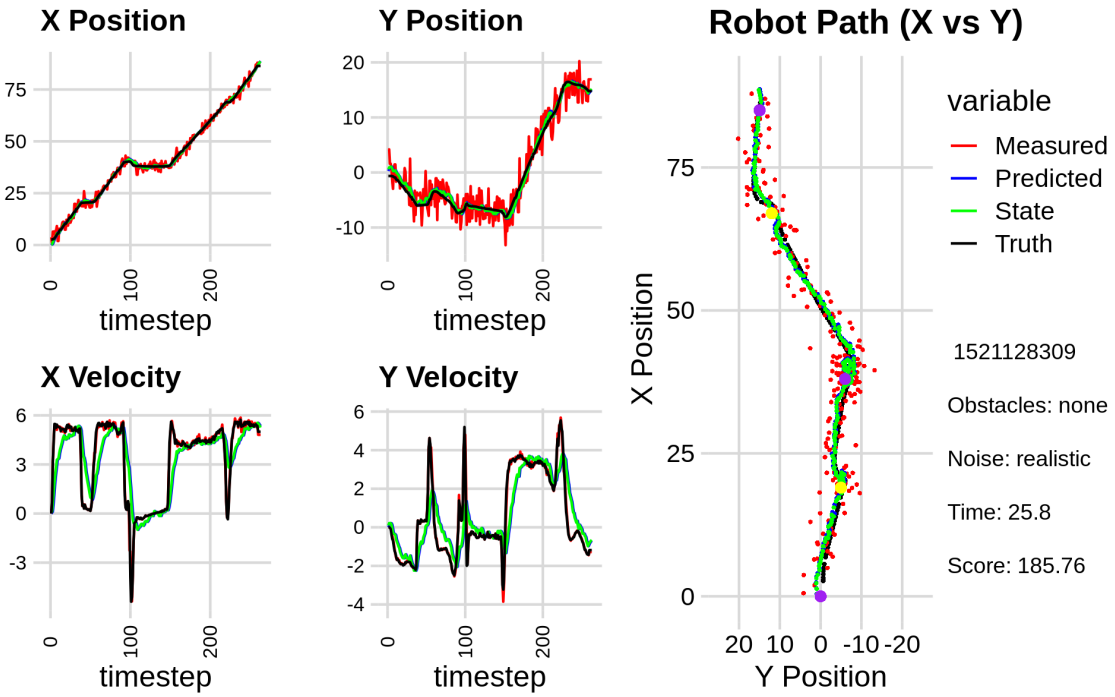

FIG. 5: Plot generated to show data for a single run of the simulation, with the results and settings shown in the bottom right corner for the sake of reproducibility and comparison.

The Kalman Filter not only receives data from the agent and simulator, but also sends its estimate of the current state to the Localization Node, which is used in place of the raw sensor values. This trivially improves the localization, but it's hard to tell whether improving the localization will lead to improvements in the scores of agents overall. This is because any given run has a randomized environment, and it seems that an agent's success, or even ability to

reach the goal, is more dependent on this randomness than the slight variations in localization precision. This is important to bring up here, because it leads into our choice of cost function.

Initially, we used the score determined by the simulator for a particular run directly as the cost function for our agents, since a lower score is best. This was highly variant, and failed to isolate important characteristics enough to have any meaningful evolution. The most significant detriment is that the score is not directly dependent on the Kalman Filter, and is thus a poor proxy for the kind of cost function we really need. This realization led to changing to the cost function we now use, which relies only on the robot's localization success.

Recalling that the presence of obstacles doesn't really affect our perception one way or the other (since we aren't using the LiDAR data in our KF inputs), and that contrary to our goals, obstacles provide more variance and less certainty that an agent will be able to run for sufficient time to gather data for its cost calculation, it is clear that they should remain disabled for this portion of the project.

Now with a cost function defined, it was important to decide what variables would form the "genome" for our evolutionary module. We originally included some parameters of the motion, such as from the P-controller in Equation (2), but removed them to focus solely on perception. With this in mind, we used 12 values for the genome, which make up the diagonal entries of the three KF matrices $\boldsymbol{P}_{initial}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$. These are each 4x4 matrices representing the initial covariance, the process noise, and the measurement uncertainty used by the KF, and are the "hyperparameters" discussed throughout this paper.

Since the first part of this project involved manually implementing and optimizing the Kalman Filter, we already had some reasonable values for a "default genome." To see if this could be improved upon, some runs of our program were initialized with slightly mutated versions of this default genome. Then, to test our program's ability to find an optimum on its own, we did some more runs with a much more random first generation, which understandably performed worse at the outset than our hand-selected genome. The results of these two types of tests are described in the next section.

Our initial attempts at evolving towards an optimal genome encountered some problems. The first of these problems is the randomness in our "fitness gathering process," i.e., when the simulator is run by a given agent. This type of randomness is not uncommon in reinforcement learning applications, but in our case, a single run of the simulation can take up to 40 seconds, which means that we cannot overcome this by running large numbers of agents for a large number of generations. The changes discussed so far have mitigated these issues to a point where we can run the program for a reasonable number of agents and generations, and actually see results.

4 Results and Discussion

4.1 Kalman Filter Results

We can see from the plot in FIG. 6, as we do for the graph of any given run, that the positions are tracked by the KF fairly well, and the velocities are pretty bad. This is due to our constant velocity model. The KF is using the velocities to make predictions of future positions, but is assuming there is no acceleration, and is thus making poor predictions for future velocities. We would need our state to contain one level of precision higher than desired, and the last layer will be imprecise no matter what. Here, we only care about the positions.

It is also apparent from FIG. 6 that the state is able to converge on the truth when there is little change in heading and velocity, but it shoots slightly off track and must rebalance when major changes occur, such as after rounding each waypoint in this example. This demonstrates the handoff between measurements and predictions in the KF. If the robot has little changes to

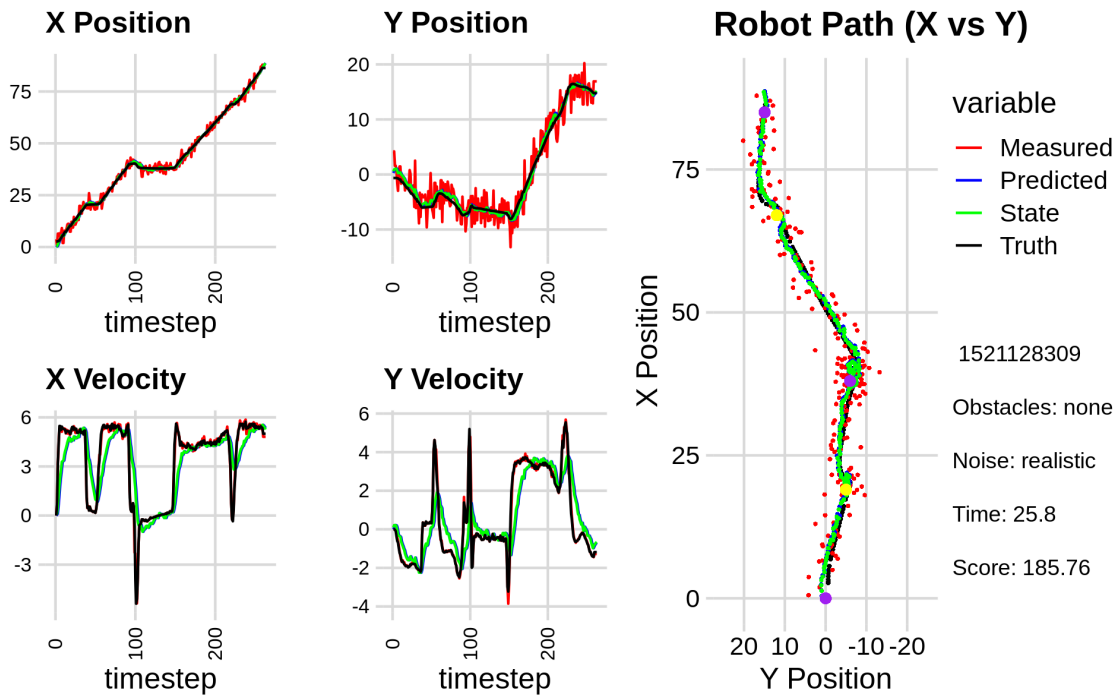

FIG. 6: Plot generated to show data for a single run of the simulation.

other variables, it will become more and more confident in its predictions, since they are very accurate. When a sudden untracked change occurs, such as when the robot abruptly makes a large turn, the KF takes a few cycles to switch its confidence over to the measurements and get back on track. This happens over and over again, keeping the estimated state as accurate as possible for as much of the runtime as it can.

4.2 Machine Learning Results

Our cost function was described in Section 2, and it bears repeating that our sensors have a precision of ± 5 meters, so the raw cost is around 10m $(x + y)$; with our manually tuned KF, we are able to average around 1.75 meters, representing over 5 times improvement. This section examines further or parallel improvements to this caused by our use of evolutionary computation.

The results of running our program for many large generations are promising. We see that on its own, the evolutionary tuning is able to take the genome from a random start to a similar level as our manual tuning. This progress is shown in FIG. 7. This "random start" consists of each gene being chosen from a Gaussian distribution centered at 0.5 with a standard deviation of 0.1; all genes are normalized within the range $(0, 1)$. As expected, the first few generations perform worse than the manually tuned KF, but are still mostly better than our initial agent without the Kalman Filter. After a few generations, the mean cost of each population tends to converge on 1.75 meters (close to the mean of our manually tuned KF) and after this point, the variance within each generation continues to decrease. At the end of our full run, we have consistent performance which rivals that of the KF results described in the previous section.

When initializing the genomes not at random, but as slight mutations of the result of our manual tuning, we see a similar trend; however, this trend results in a better mean cost than the random runs. We see this very consistently over several runs; generations based on our manual trend are better than in the random runs, and they are still able to improve and coalesce in a similar way. A run with this process is shown in FIG. 8.

In either use case, we see generations condensing down to very low variance, after which

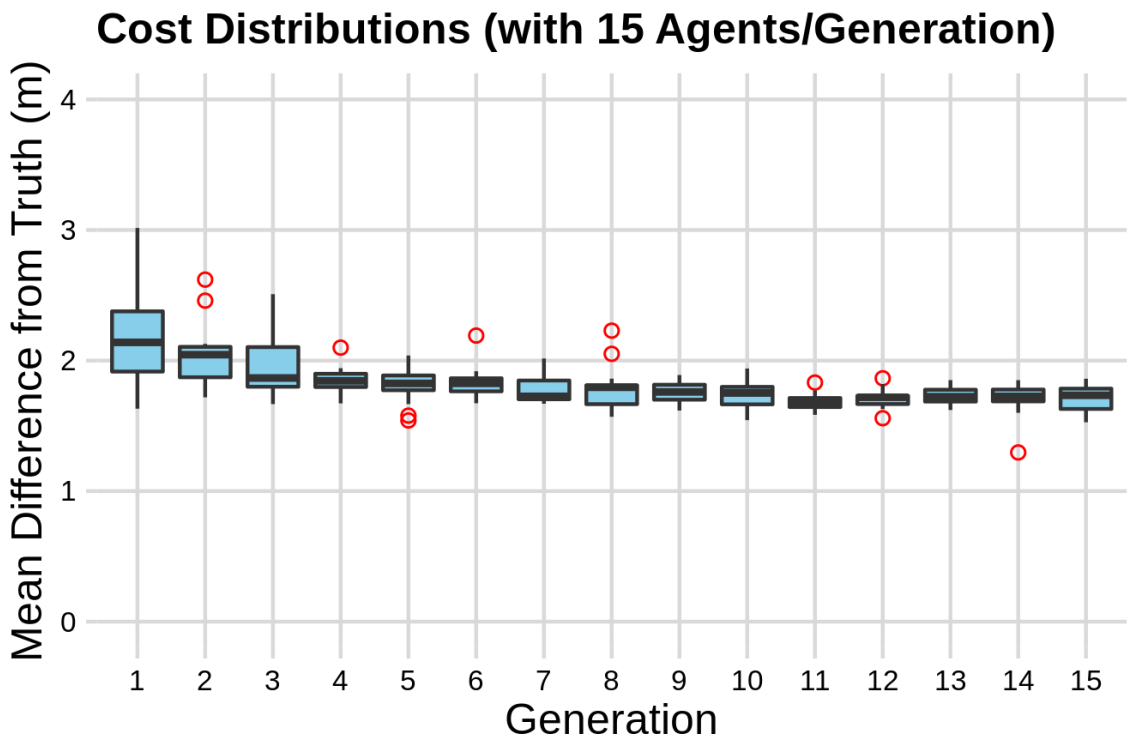

FIG. 7: This plot shows the cost function's results for a 15 agent population over 15 generations, in which the first generation is created at random. There is clear improvement from the starting generation's 2.2m to the final generation's consistent 1.75m.

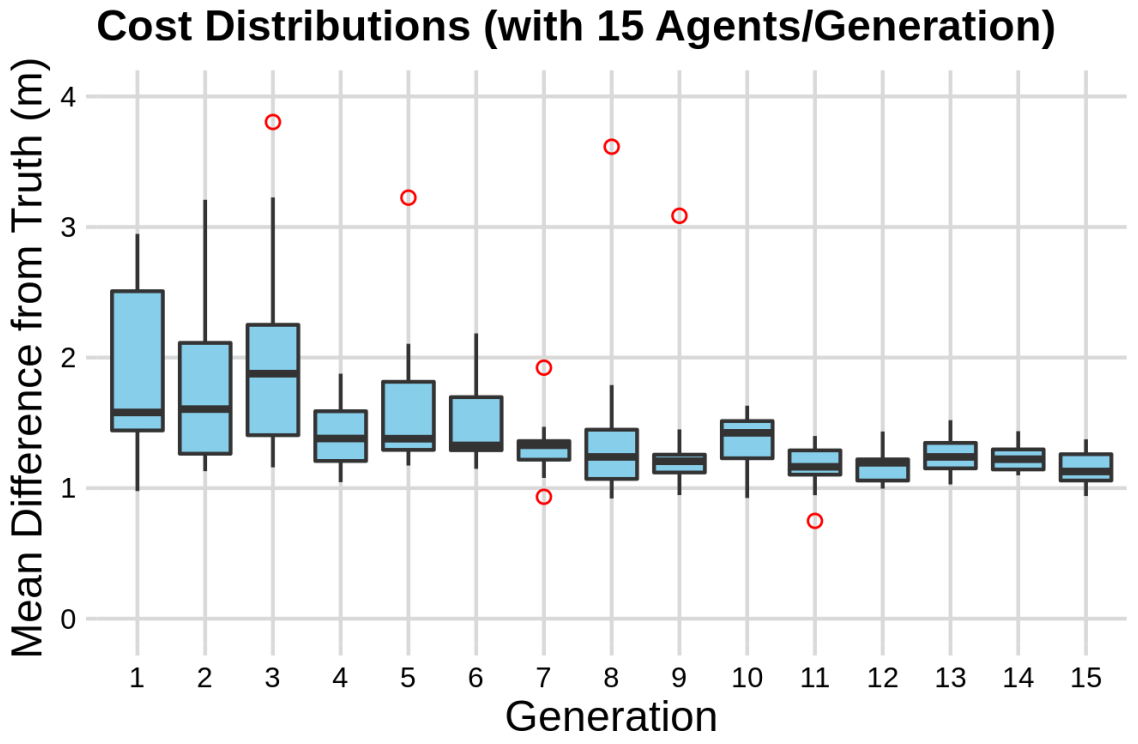

FIG. 8: This plot shows the cost function's results for a 15 agent population over 15 generations, in which the first generation is based on our manually tuned KF. There is clear improvement from the starting generation's 1.75m to the final generation's consistent 1.2m.

the mean cost stays very consistent. We discuss the implications of this in the next section.

5 Conclusions and Future Work

This project demonstrated several methods of affecting the performance of an autonomous mobile robot. We saw that the introduction of a manually-tuned Kalman Filter led to improvements in state tracking over our raw estimates, so our first hypothesis can be confirmed. We also saw that rather than manually tuning the KF, we can use its parameters as the genome to an evolutionary algorithm, which allows agents to learn how to improve its performance over several generations; thus our second hypothesis is also confirmed. We further found that combining the two methods – that is, performing some manual tuning and then using that as a jumping-off point for our evolutionary tuning – leads to even better performance than either method alone.

Although our attempt to introduce machine learning into this project was a success, we did discover some limitations; namely, it seems the variance of each generation decreases to a minimal value within ten or so generations. This could be a benefit or a hindrance; it does entail enhanced consistency within the entire population around a mean value that has been optimized locally, but is likely limiting exploration more than necessary. This reduction in variance often implies a reduction in the genetic diversity of our agents, meaning less of the fitness landscape is available for exploration; this would explain why randomized runs settle down to a local optimum and do not continue improving their mean after the first few generations. It's possible this behavior could be different with modifications to the crossover and mutations procedures, such as the likelihood any given gene is mutated, and how aggressive a mutation can be. There is certainly a balance to be found, since higher mutations encourage more exploration, but also induce more randomness. This also could be due to our process for selecting agents for reproduction, since the weight linearly decreases and is minimal for "poor" agents. Changing this to allow say, a 5% chance the agent is selected at random rather than weighted by cost, could preserve exploration abilities of a population even into later generations. We can see that even in a longer run with more agents, as in FIG. 9, improvement levels off relatively early on, and we know from some other runs in the Results section that this value is not optimal. A future project, or future continuation of this project, could seek to address this failure to preserve evolutionary ability after a few generations of high improvement.

We spent considerable time tweaking the parameters of this process to obtain the caliber of performance we now see, but we do not assume that we found the perfectly optimal conditions. All we can say on this front is that the machine learning aspect of this project is fairly effective with a little effort, and that if it can be improved, it would take enough effort to negate the time saved from simply tuning the KF optimally in the first place. It would be interesting to see how different distributions for creating the first generation of genomes would affect the results. A future project could modify this part of the procedure and record how the evolution and the final result are affected.

Some more limitations of the evolutionary computation aspect came with mixing different components of the project architecture into the same genome. When attempting to evolve both the KF parameters and a motion policy, the results were garbled together, and there was very little consistency in which one, if either, was improved via evolution. After moving to a genome and a cost function which both deal only with the perceptive abilities of the agents, we saw much more significant progress. It could certainly be possible to evolve things like the motion policy, navigation procedure, behavior for obstacle avoidance and recovery, and more, but we would specifically attempt evolving one feature at a time, with a fitness function that measures the effect of changes in the genome as directly as possible. Future projects could use this base designed for perception as a foundation for evolving other components.

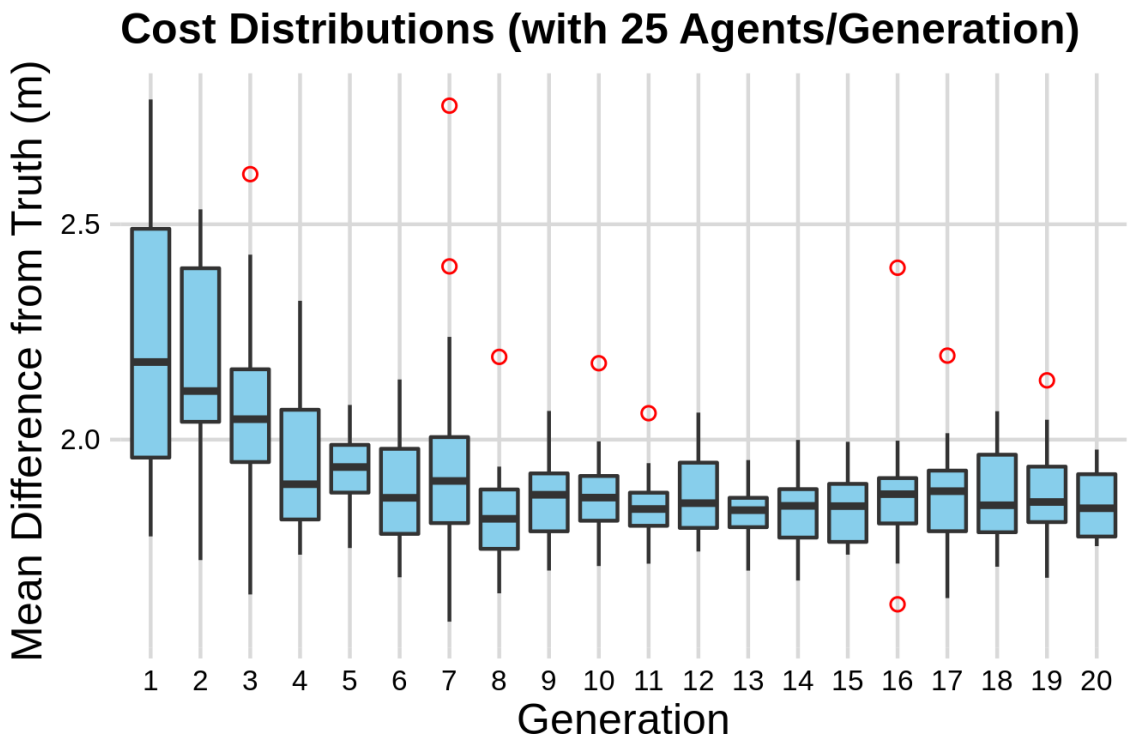

FIG. 9: This plot shows the cost function's results for a 25 agent population over 20 generations, in which the first generation is created at random. There is clear improvement from the starting generation's 2.3m to the consistent 1.75m, but it levels off and does not reach costs as low as we know to be possible. (Notice the y-axis scale is much more compressed than in our previous plots to show much more granularity.)

Our original intention with this project was to optimize our procedure in simulation before moving on to attempt something similar with physical robots. We did not come out of the simulation, but a future project could adapt this one by using a simulator specifically made to mimic an available physical robotics scenario as closely as possible.

We invite any other parties to make their own changes or additions by forking the GitHub repository for this project, available at github.com/kevin-robb/capstone-kf-ml.

6 References

[1] Alex Becker, *Kalman Filter Tutorial*, Last modified 2018, www.kalmanfilter.net.

[2] Michael Nielson, *Neural Networks and Deep Learning (2019)*, Chapter 1, www.neuralnetworksanddeeplearning.com.

[3] Jianguo Jack Wang, Weidong Ding, and Jinling Wang, *Improving Adaptive Kalman Filter in GPS/SDINS Integration with Neural Network*, School of Surveying and Spatial Information Systems, University of New South Wales, Australia. ResearchGate, 2013, www.researchgate.net/publication/251439107_Improving_Adaptive_Kalman_Filter_in_GPSSDINS_Integration_with_Neural_Network.

[4] Justin Kleiber, *IGVC Kalman Filter Derivation*, Sooner Competitive Robotics, Intelligent Ground Vehicle Competition design document submission, March 16th 2020. Available at https://github.com/SoonerRobotics/igvc_software_2020/blob/master/docs/IGVC_Kalman_Filter_Derivation.pdf.

[5] Noah Zemlin, *SCR Software Challenge Simulator*, Sooner Competitive Robotics, 2020, https://github.com/SoonerRobotics/SCR-SWC-20/releases.

[6] Dean Hougen, *CS 4023: Intro to Intelligent Robotics* course material, Spring 2020, https://www.cs.ou.edu/~hougen/classes/Spring-2020/Robotics/.

A Simulator Details

The simulator I am using is specified in citation [5]. It was created by Noah Zemlin in May 2020 for an internal software competition for our student organization, Sooner Competitive Robotics. He has given me full permission to use it for this project.

The score is determined by many performance factors as follows:

- Time to reach the goal is the main score, which is multiplied by the other factors. The simulator will time out at 300 seconds.
- Simulator settings give multipliers which are a benefit (< 1) or penalty (>1).
  - Obstacles (none:15x, normal:1x, hard:0.1x)
  - Noise (none:15x, reduced:1x, realistic:0.6x)
- The robot takes damage upon colliding with obstacles. A penalty is given in the form of a multiplier proportional to the damage taken, up to a maximum of 10x. If the robot is fully destroyed, it is instead given a multiplier of 100x and the time is assigned the maximum value of 300 seconds.
- A benefit of 0.8x is given for hitting each of the optional waypoints on the way to the goal. There are always three of them in addition to the required final waypoint.

The simulator settings affecting the score are further explained here:

- Obstacles
  - none: There will be no obstacles.
  - normal: There will be obstacles.
  - hard: There will be 3 times more obstacles.
- Noise
  - none: No noise on any sensors.
  - reduced: Half the standard deviation on the noise as realistic on all sensors.
  - realistic: Realistic noise profiles on all sensors.

Example outputs for two runs are shown below. The first is a moderately successful run, and the second is a failure in which the robot was destroyed by repeated obstacle collisions.

```
Team KevinRobbbotics
Date: 2021-02-19 19:03:42 -06
Seed: 600319646

Time: 25.600
Multiplier: x3.127
Score: 80.062

-- Multipliers --
Obstacles Bonus: x1.00
Noise Bonus: x0.60
Waypoint 1 reached: x0.80
61% Damaged: x6.52
```

```
Team KevinRobbbotics
Date: 2020-11-03 13:39:42 -06
Seed: 679245648

Time: 300.000
Multiplier: x64.000
Score: 19200.000

-- Multipliers --
Obstacles Bonus: x1.00
Noise Bonus: x1.00
Waypoint 1 reached: x0.80
Waypoint 2 reached: x0.80
Robot Destroyed: x100.00
```

B Kalman Filter Derivation

The Kalman Filter (KF) is essentially a loop of computations based on five main equations, separated into the "Predict" (time update) and "Correct" (measurement update) modules. This process is shown in FIG. 10. The KF requires an initial estimate, which need not be perfect, but is simply a starting point from which to attempt convergence on the truth.

The main five KF equations are:

1. *State Extrapolation Equation* - prediction of the state at the next timestep based on the (known) estimation for the current timestep.
2. *Covariance Extrapolation Equation* - measure of the uncertainty in the prediction.
3. *Kalman Gain Equation* - required for computation of the update equations.
4. *State Update Equation* - estimation of the current state based on the (known) estimation from the previous timestep and the measurement for the current timestep.
5. *Covariance Update Equation* - measure of the uncertainty in our estimation.

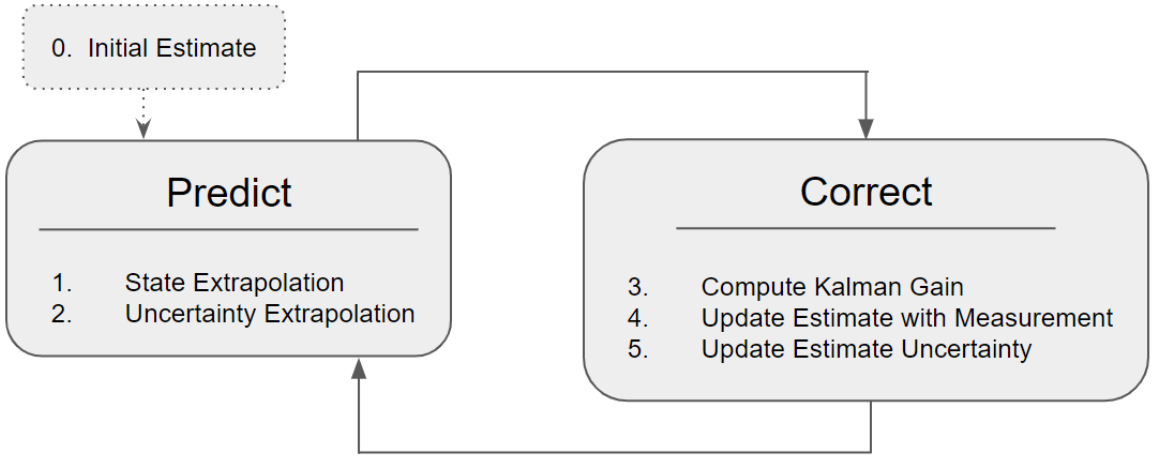

FIG. 10: Outline of the Kalman Filter process as it relates to the equations. Based on a figure from [1]

B.1 Notation

The equations throughout this section will commonly have expressions such as $\hat{\boldsymbol{x}}_{\boldsymbol{n+1,n}}$. The $\hat{\boldsymbol{x}}$ should be read as "the estimate of $\boldsymbol{x}$." Since it is bold and lowercase, it is a column vector. Bold, uppercase symbols represent matrices, and non-bold symbols are scalars. Finally, the subscripts denote which timesteps this expression is applicable to. Here, $\hat{\boldsymbol{x}}_{\boldsymbol{n+1,n}}$ should be read as "the prediction of $\boldsymbol{x}$ at timestep $n+1$ performed at timestep n." Conversely, $\hat{\boldsymbol{x}}_{\boldsymbol{n,n}}$ is an estimate for $\boldsymbol{x}$ at timestep n performed at timestep n; since these are the same timestep, this is the estimate of our current state, rather than a prediction of a future state.

These two alternatives will be the only categories of subscripts that appear, so just remember that if the numbers are the same, it is an estimate of the current state, and if they are different, it is a prediction for the next state. It would not make sense for the first value to be less than the second, because if we are at timestep n, we already know the state estimations for all timesteps $m < n$. It would also not make sense for the difference between these two numbers to be larger than 1, because we can only predict a state given the previous state; we would not skip timesteps.

B.2 The Equations

B.2.1 State Extrapolation Equation

This equation predicts the system state at the next timestep given the current state estimate, the control signals, and a process noise.

$$\hat{\boldsymbol{x}}_{\boldsymbol{n+1,n}} = \boldsymbol{F} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{n,n}} + \boldsymbol{G} \cdot \hat{\boldsymbol{u}}_{\boldsymbol{n,n}} + \boldsymbol{w}_{\boldsymbol{n}} \tag{4}$$

where $\boldsymbol{x}$ is the system state, $\boldsymbol{F}$ is the state transition matrix, $\boldsymbol{G}$ is the control matrix ("input transition matrix"), and $\boldsymbol{w}$ is the process noise (unmeasurable input that affects the state); $\boldsymbol{u}$ is a control variable or input variable (a measurable or deterministic input to the system). Since we are using a constant velocity (zero acceleration) model, we take $\boldsymbol{u}$ to be zero, simplifying the equation to

$$\hat{\boldsymbol{x}}_{\boldsymbol{n+1,n}} = \boldsymbol{F} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{n,n}} + \boldsymbol{w}_{\boldsymbol{n}} \ . \tag{5}$$

B.2.2 Covariance Extrapolation Equation

This equation predicts the estimate uncertainty (i.e., covariance) of the next state given the covariance of the current state and the process noise matrix.

$$\boldsymbol{P}_{\boldsymbol{n+1,n}} = \boldsymbol{F} \cdot \boldsymbol{P}_{\boldsymbol{n,n}} \cdot \boldsymbol{F}^{\boldsymbol{T}} + \boldsymbol{Q} \tag{6}$$

where $\boldsymbol{P}$ is the covariance matrix, and $\boldsymbol{Q}$ is a process noise matrix.

B.2.3 Kalman Gain Equation

The *Kalman Gain* is the relative weight of the predictions versus the measurements, which we recalculate every timestep and use to update our state estimation.

$$\boldsymbol{K}_{\boldsymbol{n}} = \boldsymbol{P}_{\boldsymbol{n,n-1}} \boldsymbol{H}^{\boldsymbol{T}} (\boldsymbol{H} \boldsymbol{P}_{\boldsymbol{n,n-1}} \boldsymbol{H}^{\boldsymbol{T}} + \boldsymbol{R}_{\boldsymbol{n}})^{-1} \tag{7}$$

where $\boldsymbol{K}$ is the Kalman Gain, $\boldsymbol{H}$ is the observation matrix, and $\boldsymbol{R}$ is the measurement uncertainty (i.e., measurement noise covariance) matrix.

Note the Kalman Gain is a matrix, not a column vector.

B.2.4 State Update Equation

This equation creates our estimate for the state at the current timestep by using the Kalman Gain to compare the previous prediction with the measurements.

$$\hat{\boldsymbol{x}}_{\boldsymbol{n,n}} = \hat{\boldsymbol{x}}_{\boldsymbol{n,n-1}} + \boldsymbol{K}_{\boldsymbol{n}} \cdot (\boldsymbol{z}_{\boldsymbol{n}} - \boldsymbol{H} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{n,n-1}}) \tag{8}$$

where $\boldsymbol{z}$ is the measurement vector.

B.2.5 Covariance Update Equation

This equation uses the Kalman Gain to update our covariance, based on the prediction at the previous timestep.

$$\boldsymbol{P}_{\boldsymbol{n,n}} = (\boldsymbol{I} - \boldsymbol{K}_{\boldsymbol{n}} \boldsymbol{H}) \boldsymbol{P}_{\boldsymbol{n,n-1}} (\boldsymbol{I} - \boldsymbol{K}_{\boldsymbol{n}} \boldsymbol{H})^{T} + \boldsymbol{K}_{\boldsymbol{n}} \boldsymbol{R}_{\boldsymbol{n}} \boldsymbol{K}_{\boldsymbol{n}}^{\boldsymbol{T}} \ . \tag{9}$$

B.2.6 Measurement Equation

This auxiliary equation is used to relate measurements and state values. This is important because we likely cannot directly measure our system variables. An example of this is measuring GPS coordinates, acceleration, and heading to determine a system state of position (in meters) and velocity (in meters per second). This is why we have $\boldsymbol{H}$, the observation matrix. It relates the measurements to the state as follows:

$$\boldsymbol{z_n} = \boldsymbol{H}\boldsymbol{x_n} + \boldsymbol{v_n} \tag{10}$$

where $\boldsymbol{v}$ is a random noise vector.

$\boldsymbol{H}$ can be thought of as "recreating the measurements from the system state." Thus, our state must have sufficient information such that we can use only the values in a single timestep to compute what the sensors' measurements would have been to create that state. I.e., if our state were $\begin{pmatrix} x & y & |v| \end{pmatrix}^T$, we could not then include $\dot{x}$ and $\dot{y}$ in our measurement vector, since we cannot recreate these values from our state. If our state also contained the robot's heading, or the component velocities themselves, then this would be allowed.

B.3 Our Application

We use a 4D system state, $\boldsymbol{x} = \begin{pmatrix} x & y & \dot{x} & \dot{y} \end{pmatrix}^T$. These are the x and y component positions (m) and velocities (m/s) in the global frame of reference, with the robot beginning at $\begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}^T$ and facing along the positive x-axis.

Since we are using a constant velocity model, our equations of motion are rather simple:

$$\begin{cases} \frac{dx}{dt} = v_x = \dot{x} \\ \frac{dy}{dt} = v_y = \dot{y} \\ \frac{dv_x}{dt} = \ddot{x} = 0 \\ \frac{dv_y}{dt} = \ddot{y} = 0 \end{cases} \tag{11}$$

We can derive our state transition matrix, $\boldsymbol{F}$, by first writing these differential equations in matrix form, $\dot{\boldsymbol{x}} = \boldsymbol{A} \cdot \boldsymbol{x}$:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{v_x} \\ \dot{v_y} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix} \tag{12}$$

Solving this matrix equation, we see that $\boldsymbol{A}^2 = 0$, and thus all higher powers of $\boldsymbol{A}$ will drop out. This gives us the simple equation $\boldsymbol{F} = \boldsymbol{I} + \boldsymbol{A} \cdot \Delta t$, so

$$\boldsymbol{F} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

We can check the resultant system of equations produced by this matrix to see that these are some of the simple Newtonian kinematics equations.

$$\begin{cases} \hat{x}_{n+1,1} = \hat{x}_{n,n} + \hat{\dot{x}}_{n,n} \cdot \Delta t \\ \hat{y}_{n+1,1} = \hat{y}_{n,n} + \hat{\dot{y}}_{n,n} \cdot \Delta t \\ \hat{\dot{x}}_{n+1,1} = \hat{\dot{x}}_{n,n} \\ \hat{\dot{y}}_{n+1,1} = \hat{\dot{y}}_{n,n} \end{cases} \tag{14}$$

Now that we've settled $\boldsymbol{F}$, and by our simple model assumption we do not need $\boldsymbol{G}$, we can move on to deriving $\boldsymbol{H}$, the observation matrix, and $\boldsymbol{z}$, the measurement vector.

For this project, the measurements we will be able to receive include:

- λ - latitude, from GPS
- Λ - longitude, from GPS
- ϕ - global heading (yaw), from IMU
- v - forward linear speed, from encoders

Since we are not including ϕ in our system state, we cannot have it in $\boldsymbol{z}$ either. Because of this, we will perform some small pre-computations as soon as we receive new measurements, which will allow us to satisfy this requirement. We can also make some of the KF equations simpler by forcing $\boldsymbol{H}$ to be $\boldsymbol{I}$, the identity matrix. This is actually fairly straightforward to do, and only requires a few more pre-computations. The following definition of $\boldsymbol{z}$ allows us to define $\boldsymbol{H}$=$\boldsymbol{I}$.

$$\boldsymbol{z} = \begin{bmatrix} (\lambda - \lambda_{start}) \cdot \text{lat\_to\_m} \\ (\Lambda - \Lambda_{start}) \cdot \text{lon\_to\_m} \\ v \cdot \cos\phi \\ -v \cdot \sin\phi \end{bmatrix} = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix} \tag{15}$$

Since the region of the simulator is so small (<100 m x 100 m), we can linearly translate GPS coordinates to meters from the starting position, with the following conversion factors calculated for the same location as the simulator assumes, which is the football stadium at the University of Oklahoma.

- lat_to_m = 110944.33
- lon_to_m = 91058.93

We can verify that this method works with Equation (10):

$$\begin{gathered} \boldsymbol{z} = \boldsymbol{H} \cdot \boldsymbol{x} + \boldsymbol{v} \\ \begin{bmatrix} (\lambda - \lambda_{start}) \cdot \text{lat\_to\_m} \\ (\Lambda - \Lambda_{start}) \cdot \text{lon\_to\_m} \\ v \cdot \cos\phi \\ -v \cdot \sin\phi \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} + \text{ noise} \\ \boldsymbol{z} = \boldsymbol{z} + \text{ noise} \end{gathered} \tag{16}$$

The main aspect remaining to be set for our application is $\boldsymbol{P}$, the covariance matrix. This represents our uncertainty in our knowledge of the true values, and is how we keep from assuming our predictions equal the ground truth. If we were claiming to know the exact positions and velocities, we would have $\boldsymbol{P}$ as a 4D square matrix filled with only zeroes. Since our measurements have noise, and our process itself has noise, we don't know the precise state values, so we will choose suitable variances for our initial definition of $\boldsymbol{P}$. $\boldsymbol{P}$ will be modified throughout the operation of the filter by Equations (6) and (9), so we only need to ensure that we initialize it with values that are close enough to optimum that the filter can self-correct before forcing its way off the course and becoming lost.

Filling in $\boldsymbol{P}$ with the normalized variances of our measurements along the diagonal is a good starting strategy, since this will allow the filter to prioritize measurements at first until running for enough timesteps to have a reasonably accurate estimation of the state. We don't

know the exact variances of our sensors, so some rough experimentation yields the following matrix as a good enough starting point for the KF to take over.

$$\boldsymbol{P}_{initial} = \begin{pmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0.5 \end{pmatrix} \tag{17}$$

The last undefined parameters of the KF are the "hyperparameters" that we mention throughout the main paper. These are values chosen by the user that are not modified during the operation of the program. As such, these can be thought of as a manual tuning requirement. Our aim with the latter half of this project is to use machine learning to do this tuning for us, so Equation (18) is used as a placeholder for the first generation of our evolutionary computation module, and is used as the default when our program is run without the learning aspect. It's worth noting that ordinarily the values of $\boldsymbol{R}$ would be set based on the real precision values of the sensors used, but we use it differently to emphasize simplicity in our model.

We start with $\boldsymbol{Q}$ and $\boldsymbol{R}$ set to:

$$\boldsymbol{Q} = \boldsymbol{R} = \begin{pmatrix} 0.01 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0.01 \end{pmatrix} = 0.01 \cdot \boldsymbol{I} \tag{18}$$

We evolve the cell values for $\boldsymbol{Q}$, the process noise, $\boldsymbol{R}$, the measurement uncertainty, and the initial values of $\boldsymbol{P}$ that we just described.