Architecture for Testing Learning-Based Autonomous Vehicle Control Design

Michael Kogan
Department of Electrical and Computer Engineering
Royal Military College of Canada
Kingston, Ontario, Canada
Email: Michael.Kogan@forces.gc.ca

Peter T. Jardine
Department of Electrical and Computer Engineering
Royal Military College of Canada
Kingston, Ontario, Canada
Email: peter.jardine@rmc.ca

Sidney N. Givigi
Department of Electrical and Computer Engineering
Royal Military College of Canada
Kingston, Ontario, Canada
Email: sidney.givigi@rmc.ca

***Abstract*—This paper presents the architecture for the testing of autonomous vehicle controllers designed using machine learning. A localization system feeds measurement data into a ground station. The ground station processes this data and provides the position of the vehicle to the controller, which has been tuned offline via machine learning. Control inputs are then generated and provided to the vehicle in order to accomplish the desired mission. The performance of the learned controller is compared to a nominal case. During the offline tuning process, a vehicle model is used to simulate the actual vehicle. Once tuning is complete, the parameters are used to control a real vehicle in order to accomplish the desired mission. The proposed architecture was tested by tuning a model predictive controller to guide a differential drive robot to a series of waypoints using a Reinforced Learning technique known as Learning Automata. The controller was then tested online with a similar but different problem on a real differential drive robot. The results showed that after tuning, the vehicle performed significantly better. This demonstrated that offline learning techniques can be used to optimally select controller parameters.**

I. Introduction

Architectures present a streamlined approach to solving a problem, and therefore exist for many different applications. One notable field where different architectures are used is in software engineering [1]. Another prominent use of architecture arises in the field of robotics, where different architectures exist for the design of robot behaviour [2]. The purpose of developing a testing architecture is to streamline the testing of new ideas so that the tests are fast and results are easily comparable. An architecture allows a designer to run the same tests in a controlled environment for each of his or her designs. This ensures that the test results are a good measure of performance, leading to the best design being chosen. The primary goal of an architecture is to be as modular as possible while still providing enough detail to streamline the process, in order to allow for a broad number of use cases.

This paper presents the architecture that was developed at the Royal Military College of Canada in order to test vehicles whose controllers have been designed using machine learning. The architecture involves a localization system, a ground station, a controller, a vehicle, and a learning environment. The model of the vehicle is used to simulate the vehicle's behaviour so that the controller can be tuned offline in a realistic environment. The tuned controller is then used to control the real vehicle. The localization system provides measurement data to the ground station, which provides accurate position and orientation feedback to the controller in real-time. The controller sends commands to the vehicle to ensure that it completes the mission. These commands are equivalent to the inputs that we would pass to the vehicle according to the vehicle model. Low level controllers on the robot are used to actuate the vehicle in accordance with these commands.

The use of machine learning techniques to tune controllers is a relatively recent innovation in control engineering. Classically, controllers are tuned via trial-and-error [3] and experience by taking into account the dynamics of the control problem [4]. More recent studies have proposed auto-tuning approaches based on particle swarm optimization [5], [6] and genetic algorithms [7]. A Reinforcement Learning (RL) technique known as Learning Automata (LA) has been shown to be an effective way to tune the gains of a Proportional-Integral-Derivative (PID) controller [8]. Recently, this learning technique has been applied to linear [9], [10] and non-linear Model Predictive Control (MPC) [11].

With machine learning quickly becoming a heavily-researched topic in controller design, a testing architecture is required to provide a way for fast and effective comparison between different controllers. This architecture needs to be general enough to handle a variety of learning techniques, controllers, and vehicles. Additionally, a demonstration using a specific learning technique, controller, and vehicle needs to be performed to prove the efficacy of the testing architecture.

In order to demonstrate the effectiveness of the testing architecture, an MPC was designed using LA to train a differential drive robot to follow a series of waypoints. LA was chosen as the training method because it has been demonstrated in the past that this technique works well for vehicle control tuning [8], [9], [10], [11]. A differential drive robot was used because it is a commonly used platform in robotics research. Lastly, MPC was chosen because it is currently at the forefront of controls research, being an advanced control technique that can make efficient use of energy resources while considering constraints [12], [13], [14]. It was important to demonstrate that the architecture can handle a controller which is actively being researched. A real differential drive robot was then given a moving target to track in a real environment, and the performance was compared with a nominal case.

The remainder of this paper is organized as follows: Sections II describes the notation used throughout the paper; Section III describes the system architecture; Section IV develops the vehicle model; Section V addresses the control aspect of this architecture; Section VI presents machine learning with a focus on reinforcement learning; Section VII describes our implementation of the architecture; Section VIII presents the results; and Section IX concludes the paper.

II. Notation

In this paper, scalars are represented by lowercase letters from the Greek and Latin alphabets. The dot notation ($\dot{x}$) is used for derivatives with respect to time. Vectors are written in bold lowercase font ($\mathbf{x}$), and matrices are represented by uppercase Latin letters (A). Sets are represented by uppercase symbols written in calligraphy mode ($\mathcal{A}$), and uppercase Greek letters (Γ) represent tuples. Bold uppercase Greek letters ($\boldsymbol{\Gamma}$) represent groups of tuples. Subscripts (a_i) are used to denote individual elements in a vector, or when applied to matrices denote matrices in a series. Superscripts (Γ^i) are used to represent an element in a tuple that corresponds to the superscript.

III. System Architecture

This section describes the proposed system architecture. The architecture consists of a localization system, a ground station, a controller, a vehicle, and a learning environment. The vehicle itself consists of a vehicle computer taking command inputs and turning them into low level control signals, as well as actuators reacting to the low level control signals. The controller is tuned using a vehicle model to simulate the actual vehicle in the learning environment. In a simulator, the vehicle model and the controller can run much faster than real time, making offline learning substantially faster than online learning. Once the parameters have been determined, they are used to design the actual controller. The controller feeds control commands to the robot and receives feedback on the position of the robot via the localization system. The system can be seen in Fig. 1.

The localization system feeds measurement data to the ground station. The ground station is responsible for processing this data and sending the position and orientation data to the controller. The ground station is also responsible for saving relevant test data so that it may be compared with other results and used to validate the controller. The controller is tuned offline in the learning environment using a machine learning technique. The learning environment uses a model to simulate the vehicle during offline learning. Once the controller is tuned, it is brought online to control the vehicle. The learning environment itself is therefore not part of the online architecture, but rather a subsystem that tunes the controller prior to it coming online. The controller then generates control commands based on the position of the vehicle, the mission, and the parameters. These commands are sent to the vehicle wirelessly, in order to simulate a real-world environment where the controller and the vehicle could be running on different nodes of the network. The vehicle computer processes these commands and transforms them into low level control signals for the actuators of the vehicle.

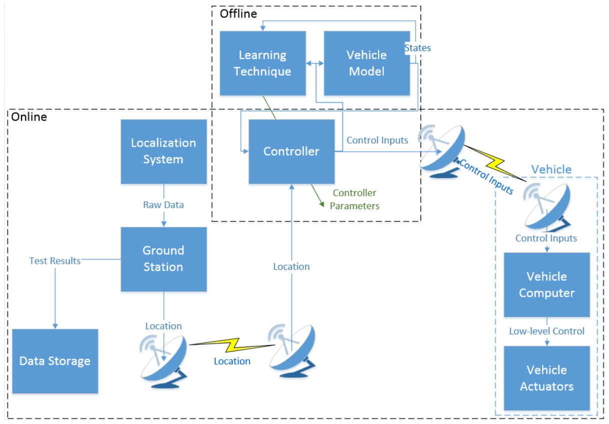

Fig. 1. Illustration of System Architecture for Vehicle Controller Testing

The architecture presented is suitable for use with most common controllers, learning techniques, and vehicles. For the purposes of our experiment, we used the Optitrack localization system to provide data to the ground station, which used QUARC software to provide accurate position feedback to the controller. The controller was an MPC tuned with an RL technique known as LA. The vehicle was a differential drive robot, a Turtlebot running the Robot Operation System (ROS). ROS provided the functionality of the vehicle computer, taking commands and turning them into low level control signals for the Turtlebot's actuators.

IV. Vehicle Modeling

A vehicle model is required for the controller tuning process. We can define the movement of any vehicle using the following equation: $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$. This equation states that the change in the states of a vehicle is always a function of its current states $\mathbf{x}$ and the control inputs $\mathbf{u}$.

A. Overview

The equation above is the most general description of an undisturbed vehicle with states dependent on previous states and inputs. The vehicle's model is used in the learning environment to simulate the behaviour of the vehicle.

B. Differential Drive Robot

Let us now explore a specific example to see how the general vehicle equation can be applied. The kinematics of a differential drive robot can be described in terms of three states: position in Cartesian coordinates (x, y) and orientation θ. The evolution of these states is controlled by two inputs, linear velocity v and angular velocity ω. The change of the orientation of the robot will depend solely on the angular velocity, and the change in x and y position will be a function

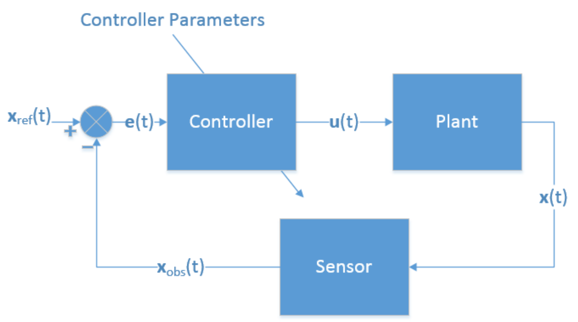

Fig. 2. Generic Controller

of the linear velocity and the current orientation. To get the change in position in each axis, we project the velocity v unto the x and y axes. This leads to the kinematics model for the differential drive robot [15]:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}v\cos(\theta)\\ v\sin(\theta)\\ \omega\end{bmatrix}$$

V. Vehicle Control

A controller is any system that generates signals to a plant in order to achieve the desired plant behaviour. A controller can also be subjected to a set of constraints in order to reflect the real world considerations of the system. The states of the system are observed via some sort of sensor, and then fed back to the controller in order to make the next control decision.

A. Overview

In case of the presented architecture, the controller can be any controller with a feedback loop and the plant can be any vehicle. The sensor is a localization system, which provides position and orientation data about the vehicle. If other states are required, an observer can be used to estimate them. The key difference between this architecture and typical controllers is that here the parameters of the controller may be modified offline through a learning process, which is represented in the drawing by a diagonal arrow going through the controller itself. Fig. 2 demonstrates the architecture of a generic feedback controller in this system.

As we can see, the controller is given the measured error vector $\mathbf{e(t)} = \mathbf{x_{ref}(t)} - \mathbf{x_{obs}(t)}$ between the desired states and observer states. It provides control inputs via the vector $\mathbf{u(t)}$ to the plant (vehicle). The plant's states $\mathbf{x(t)}$ are measured by the sensor, in our case the localization system, and fed back to the controller. The control inputs are based off of the tracked error and the previous inputs, so $\mathbf{u(t)} = \mathbf{f_c}(\mathbf{e(t)}, \mathbf{u(t)})$.

As already mentioned, this architecture allows for the modification of the parameters of the controller. A controller's parameters are values that modify how the output signal $\mathbf{u(t)}$ will depend on the tracked error and the previous inputs. In essence, they define the control function, $\mathbf{f_c}$. Control parameters differ depending on the type of controller being used. The following subsection will provide an example of a controller and the description of its specific parameters. In any case, in this architecture these parameters are learned offline in the learning environment through the use of a model and a learning algorithm, after which the controller is brought online to control an actual vehicle.

B. Model Predictive Control

In this subsection, an MPC will be explored. An MPC may be applied to a Multi-Input Multi-Output (MIMO) system. This kind of controller produces a set of control inputs over finite prediction horizon p. It uses a discrete model of the system to describe the evolution of the states over this prediction horizon. To obtain a discrete linear model, the continuous model from Section IV needs to be discretized. MPC works by taking the tracking error and a generic control input vector, $\mathbf{u(t)}$, and constructing an objective function. It then minimizes this objective function with respect the the control input $\mathbf{u(t)}$. The $\mathbf{u(t)}$ value that corresponds to the global minimum of the objective function is used as the control input. The objective function is built as follows:

$$J(\mathbf{e(t)}, \mathbf{u(t)}) = \sum_{i=0}^{p} (\mathbf{e(i)}^T Q \mathbf{e(i)} + \mathbf{u(i)}^T R \mathbf{u(i)})$$

In the function above, the limit of the summation p represents the prediction horizon, or how many timesteps into the future the controller is projecting. The control output vector can be written as:

$$\mathbf{u(t)} = \underset{\mathbf{u(t)}}{\operatorname{argmin}}\ J(\mathbf{e(t)}, \mathbf{u(t)})$$

The parameters of this controller are the diagonal elements of the symmetric matrices Q and R and the prediction horizon p. If we want to simplify the tuning of the controller, we can choose a constant value for the prediction horizon and only tune the matrices Q and R. The Q and R matrices are usually diagonal matrices with only positive numbers to ensure that they are positive definite matrices. The diagonal entries Q matrix represents the weights assigned to the different elements in the error vector. The higher the weight, the more costly that aspect of the tracking error is, and therefore the more the controller will seek to minimize that error. It is important that the relative weights of the different elements in the error vector reflect our intention, so that the controller is focusing on the most relevant aspects of the problem. The diagonal entries in the R matrix represent the weights assigned to the different elements of the control vector. These weights should reflect the relative cost of responding to different control inputs, so that the more costly inputs have higher weights and are therefore less used. For example, if turning is very costly, and angular velocity is one of the inputs, the weight corresponding to angular velocity should be high to represent that it is not desirable to turn.

VI. Reinforcement Learning

Machine learning is a technique whereby computers are taught to perform certain actions without being explicitly

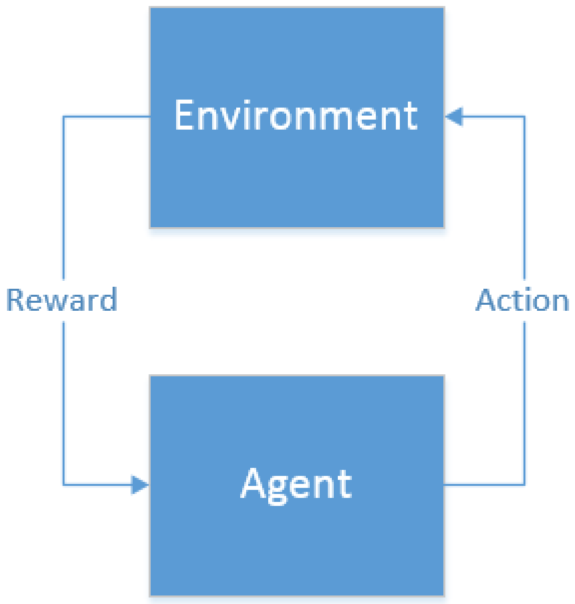

Fig. 3. Reinforcement Learning

programmed to do so. One learning technique that has been applied to control system design is known as RL.

A. Overview

RL is a learning paradigm whereby the agent takes actions in an environment in order to maximize the reward. This type of learning allows the agent to automatically determine the ideal behaviour within a specific context (the environment). In the case of the presented architecture, the agent is the controller and the environment is the vehicle model. The learning algorithm monitors the environment and the results of the actions taken by the agent, and modifies the agent by reinforcing positive results and penalizing negative results as observed in the environment. Essentially, if the control signals being sent to the vehicle model result in performance that is considered desirable, then the current parameters of the controller are reinforced. A high level illustration of this architecture can be seen in Fig. 3. A more detailed, low level architecture can be seen in Fig. 6, which will be presented in Section VII.

B. Learning Automata

LA is one learning technique that has been previously applied to controller design. When the number of actions that the agent can take in the environment is finite, the technique is referred to as Finite Action-set Learning Automata (FALA). A detailed explanation of this can be found in [8]. The timestep n is used to denote one learning trial. The classical FALA problem can be described by the following quadruple:

$$\Gamma = (\mathcal{A}, r(n), \tau, \mathbf{p}(\mathbf{n}))$$

In the formulation above,

- $\mathcal{A} = \{\alpha_1, \alpha_2, ..., \alpha_{N_r}\}$ is the set of all possible actions, known as the action set
- $r(n)$ is the reinforcement function that specifies rewards after executing an action from the action set
- τ is the algorithm that is used to update the action probabilities at time $n+1$
- $\mathbf{p}(\mathbf{n}) = [p_1(n), p_2(n), ..., p_{N_r}(n)]$ is the vector that holds the probabilities that correspond to an action being taken at a given time, so $p_i(n)$ corresponds to the probability that action α_i will be taken at time n

The algorithm that updates the probability vector $\mathbf{p}(\mathbf{n})$, is usually formulated as follows:

$$\mathbf{p}(\mathbf{n+1}) = \tau(\mathbf{p}(\mathbf{n}), \alpha(n), r(n))$$

In the formulation above, $\mathbf{p}(\mathbf{n+1})$ represents the updated probability vector, which depends on $\alpha(n)$ (the action chosen at timestep n), $r(n)$, and $\mathbf{p}(\mathbf{n})$, which is a random process who's evolution is governed by the algorithm. When there is more than one FALA, we get a game of FALA [16], [17]. In this case, we get a list of the different Γs that represent each individual FALA: $\mathbf{\Gamma} = (\Gamma^1, \Gamma^2, ...\Gamma^{N_a})$. This is important because the tuning of each controller parameter is a FALA problem, and since almost all controllers have multiple parameters, there will almost always be a game of FALA used to tune a controller, instead of just a single FALA.

VII. Architecture Implementation

This section will detail our specific implementation of the architecture. An MPC was designed using a game of FALA to control a differential drive robot, the Turtlebot as seen in Fig. 4.

We have seen the illustration of the architecture in Fig. 1. Fig. 5 shows the implementation used in this experiment. The remainder of this section will describe each component in more detail.

A. Optitrack System

The localization system used for our implementation is the Optitrack System. This system uses 24 cameras arranged in a pattern around a floor space to track distinct patterns of reflectors. This system connects to the ground station via a Universal Serial Bus connection in order to feed the measurement data in for processing. To track the robot, the system was first calibrated. Calibration determines the position of each camera relative to the other cameras and the floor (ground plane). The result of this calibration is that the cameras are able to provide the position and orientation of any object relative to the centre of the ground plane established during the calibration phase.

Once the system was calibrated, a unique pattern of reflectors was glued on top of the robot. The pattern had to be unique to ensure that the system did not confuse the robot with any other object that was being tracked. The reflectors can be seen on top of the Turtlebot in Fig. 4.

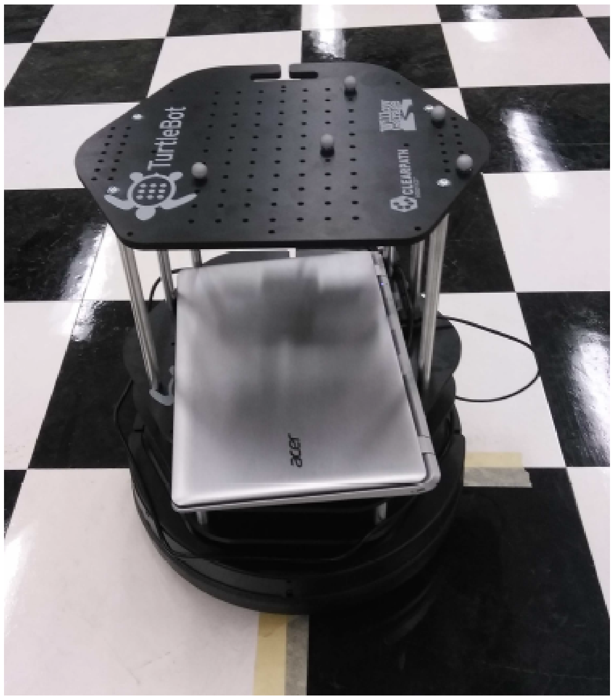

Fig. 4. Turtlebot

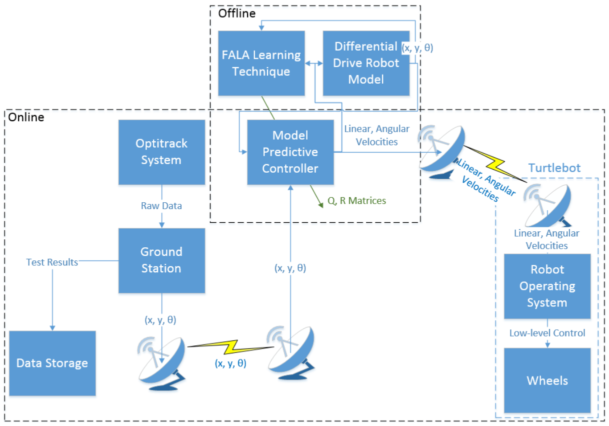

Fig. 5. Illustration of Our Implementation of the Architecture

B. Ground Station

The ground station is a computer that reads data from the cameras, transform sit, and then sends the position information to the controller. It also saves the data so that the controller may be validated. The ground station runs Motive, a software published by Optitrack, and a Quanser tool called Quarc to read the cameras and transform the measurement data into position and orientation information. The required information is then recorded into a file so that the performance of the controller may be validated later on, and is also sent to the controller so that it may generate control commands for the robot. The commands were sent via UDP.

C. Model Predictive Controller

The controller used is an MPC, which has already been described in Section V. It is important to note is that the controller needs a linear discrete model of the robot. In order to linearize and discretize the model of the differential drive robot described in Section IV, a linearization technique known as feedback linearization was used [18]. A change of variable was done so that now $\gamma_1 = x$, $\gamma_2 = y$, $\gamma_3 = \dot{x}$, $\gamma_4 = \dot{y}$, and δt is the timestep of the discretized model. The resulting equation is:

$$\begin{bmatrix}\gamma_1(k+1)\\ \gamma_2(k+1)\\ \gamma_3(k+1)\\ \gamma_4(k+1)\end{bmatrix} = \begin{bmatrix}1 & 0 & \delta t & 0\\ 0 & 1 & 0 & \delta t\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\gamma_1(k)\\ \gamma_2(k)\\ \gamma_3(k)\\ \gamma_4(k)\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta t^2 & 0\\ 0 & \delta t^2\\ 2\delta t & 0\\ 0 & 2\delta t\end{bmatrix}\begin{bmatrix}u_1(k)\\ u_2(k)\end{bmatrix}$$

The equation above is the state-space model used to represent the vehicle in the controller. The control commands, u_1 and u_2 are the accelerations in x and y respectively. Due to the linearization method used, it is only valid for $v \neq 0$. The control commands, u_1 and u_2 are translated to linear and angular velocities before being sent to the robot via UDP.

D. Offline Learning Environment

The previous subsections have discussed the online components of the system. These are used once the controller has been tuned. Before being brought online, the controller is tuned offline in the learning environment. The learning environment uses the vehicle model from Section IV to simulate the behaviour of the vehicle. The learning technique, a game of FALA, is used to observe the environment and modify the controller parameters. The parameters are the diagonal elements of the Q and R matrices, and are modified until the algorithm converges on parameters that achieve the desired behaviour in the model. Fig. 6 shows exactly how the learning was implemented. This is a more detailed breakdown of the architecture seen in Fig. 1 in the ”Offline” box.

For each set of parameters, we run a simulation for N_t steps, and calculate the total tracking error $e(t)$ by adding up all the euclidean errors between the model’s behaviour (location) and the desired location. This total cost is calculated using the error function:

$$J_{LA}(n) = \sum_{t=1}^{N_t}[k_p(e(t))^2 + k_d\frac{d}{dt}e(t)]$$

and is a scalar value that represents the error. The constants k_p and k_d define the relative importance of the different error components. This then feeds into the reinforcement signal computer, which calculates the reinforcement signal

$$r(n) = R_b \min(\max(0, \frac{J_{LA,med} - J_{LA}(n)}{J_{LA}(n) - J_{LA,min}}), 1)$$

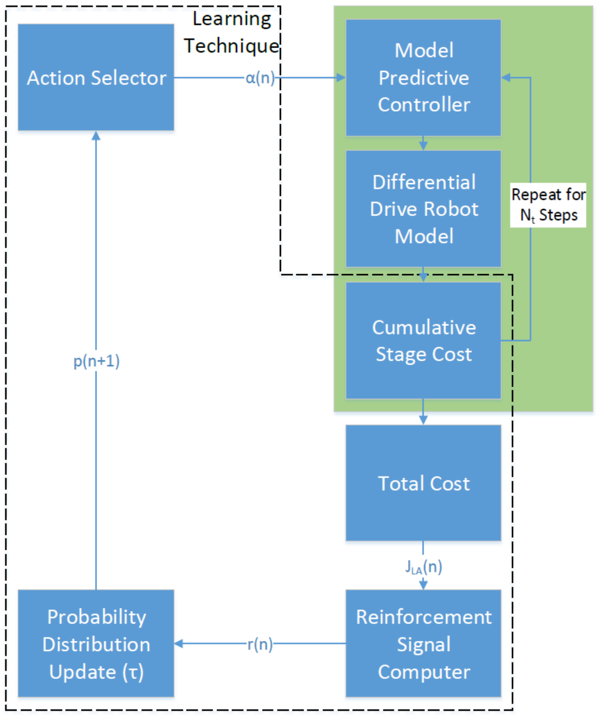
Learning Technique
Action Selector
α(n)
Model Predictive Controller
Differential Drive Robot Model
Repeat for N_t Steps
Cumulative Stage Cost
Total Cost
$J_{LA}(n)$
p(n+1)
Probability Distribution Update (τ)
r(n)
Reinforcement Signal Computer

Fig. 6. Detailed Learning Architecture

based off of the error. $J_{LA,med}$ represents the mean error over the iterations, $J_{LA,min}$ represents the minimum error, and R_b is the upper limit of a term used to accelerate the training. Lastly, this is fed into the probability distribution update function τ, which updates the probability vector

$$\mathbf{p^i(n+1)} = \frac{\mathbf{p^i(n)} + \lambda r(n)\mathbf{p^i(n)}}{|\mathbf{p^i(n)} + \lambda r(n)\mathbf{p^i(n)}|}$$

using the learning rate λ. This is the passed to the action selector, which modifies the parameters in the controller via the signal $\alpha(\mathbf{n})$, and the steps are repeated until the best results are achieved.

As already mentioned, the controller was tuned by having the robot follow a series of waypoints. The set of waypoints that the robot was trained on, the progression of the path, and the visualization of this environment can be seen in Fig. 7.

E. Turtlebot

In our implementation of the architecture we used the Turtlebot, which is a differential drive robot controlled through a middleware framework known as the Robot Operation System (ROS). ROS allows for communication and computation distribution.

VIII. Results

In order to verify the testing environment and to demonstrate that the learned parameters of the controller performed better than the designed ones, the robot was given a moving reference in the shape of a figure eight. The learned parameters performed measurably better than the designed parameters.

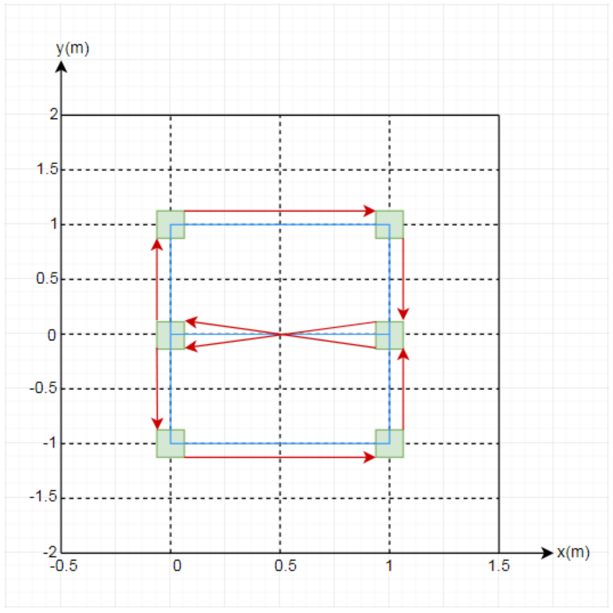

Fig. 7. Waypoints Used to Tune the Controller

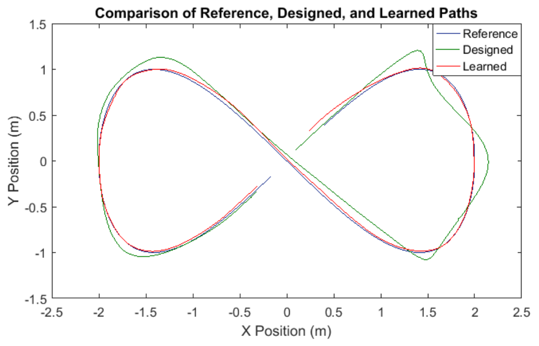

Fig. 8. Illustration of the Paths Taken by the Designed and Learned Parameters

Additionally, the testing environment allowed for fast and easy comparison of the parameters. Fig. 8 shows the reference, the path as performed with the designed parameters, and the path as performed with the learned parameters.

While it may appear evident from the illustration in Fig. 8 that the learned parameters perform better than the designed parameters, it is important to demonstrate this mathematically. Ten runs were done for each type of tuning. Then, the Mean Square Error was calculated for each run. Each type of tuning now had a sample set of ten Mean Square Error values. In order to show that the two sample sets were not drawn from the same distribution a Student's t-test was done. The null hypothesis was rejected with $p < 0.01$, which meant that the two sample sets were demonstrably different. Lastly, box plots for the two sample sets were made to visualize the difference between the two. These box plots can be seen in Fig. 9.

As we can see from the plot in Fig. 9, the learned parameters did indeed perform much better than the designed parameters. This demonstrates the efficacy of FALA as an MPC weight tuning technique. Furthermore, all of the data necessary to

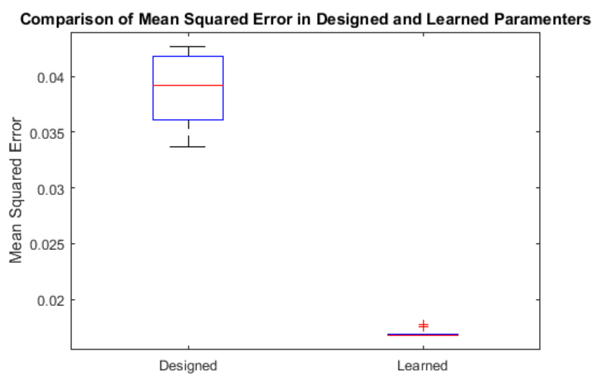

Fig. 9. Comparison of Designed and Learned Parameters

compare the two controllers was readily available due to the architecture providing constant position information about the robot, allowing us to see how well each controller performed tracking the moving figure eight reference. This demonstrates that the testing architecture presented in this paper is a good way to compare different controllers.

IX. Conclusion

To conclude, this paper presents an architecture that provides a way for fast and effective comparison between different controllers. The architecture relies on a localization system combined with a ground station to provide feedback to a generic controller which has been tuned offline using a machine learning technique. The controller then generates commands which are fed to a vehicle computer that turns them into low level control signals for the actuators on the vehicle. This architecture was implemented by tuning an MPC through FALA to control a differential drive robot. Two controllers were easily compared through the data obtained from the testing architecture. This demonstrated the efficacy of the testing architecture presented in this paper. This architecture may be used to test any controller, tuned with any machine learning technique, and controlling any vehicle.

References

[1] X. Yu, L. Liang, R. Zhou, and R. Sinha, "A software architecture for energy consumption optimization in location-based mobile applications," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct 2017, pp. 5400–5405.

[2] R. R. Murphy, *Introduction to AI Robotics*, 1st ed. Cambridge, MA, USA: MIT Press, 2000.

[3] E. C. Suicmez and A. T. Kutay, "Optimal path tracking control of a quadrotor UAV," in *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, May 2014, pp. 115–125.

[4] E. F. Camacho and C. Bordons, *Model predictive control*. Berlin Heidelberg: Springer-Verlag, 1999.

[5] K. Han, J. Zhao, and J. Qian, "A novel robust tuning strategy for model predictive control," in *2006 6th World Congress on Intelligent Control and Automation*, vol. 2, 2006, pp. 6406–6410.

[6] R. Suzuki, F. Kawai, H. Ito, C. Nakazawa, Y. Fukuyama, and E. Aiyoshi, "Automatic tuning of model predictive control using particle swarm optimization," in *2007 IEEE Swarm Intelligence Symposium*, April 2007, pp. 221–226.

[7] J. van der Lee, W. Svrcek, and B. Young, "A tuning algorithm for model predictive controllers based on genetic algorithms and fuzzy decision making," *ISA Transactions*, vol. 47, no. 1, pp. 53 – 59, 2008.

[8] S. R. B. dos Santos, S. N. Givigi, and C. L. Nascimento, "Autonomous construction of multiple structures using learning automata: Description and experimental validation," *IEEE Systems Journal*, vol. 9, no. 4, pp. 1376–1387, Dec 2015.

[9] P. T. Jardine, S. N. Givigi, and S. Yousefi, "Experimental results for autonomous model-predictive trajectory planning tuned with machine learning," in *2017 Annual IEEE International Systems Conference (SysCon)*, April 2017, pp. 1–7.

[10] P. T. Jardine, S. Givigi, and S. Yousefi, "Parameter tuning for prediction-based quadcopter trajectory planning using learning automata," in *20th World Congress The International Federation of Automatic Control*, July 2017, pp. 2377–2382.

[11] K. M. Cabral, S. R. B. dos Santos, S. N. Givigi, and C. L. Nascimento, "Design of model predictive control via learning automata for a single uav load transportation," in *2017 Annual IEEE International Systems Conference (SysCon)*, April 2017, pp. 1–7.

[12] M. Choi and S. Choi, "Model predictive control for vehicle yaw stability with practical concerns," *Vehicular Technology, IEEE Transactions on*, vol. 63, no. 8, pp. 3539–3548, Oct 2014.

[13] J. Zhao and J. Wang, "Integrated model predictive control of hybrid electric vehicle coupled with aftertreatment systems," *Vehicular Technology, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[14] M. Ramana, S. A. Varma, and M. Kothari, "Motion planning for a fixed-wing UAV in urban environments," in *4th IFAC Conference on Advances in Control and Optimization of Dynamical Systems*, February 2016, pp. 419 – 4249.

[15] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. Scituate, MA, USA: Bradford Company, 2004.

[16] K. S. Narendra and M. A. L. Thathachar, *Learning Automata: An Introduction*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[17] M. A. L. Thathachar and P. S. Sastry, *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. New York, NY, USA: Kluwer Academic Publishers, 2004.

[18] S. He, "Feedback control design of differential-drive wheeled mobile robots," in *ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, 2005.*, July 2005, pp. 135–140.