Adaptive Control of a Self Balancing Robot with Varying Payloads

Samuel Bednarski, Michael Dermksian, Alanna Mitchell, and Vybhav Murthy
College of Engineering, Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213

***Abstract*—For systems with time-varying parameters, adaptive control methods can be used to ensure stability even when these parameters are unknown. Here we present two adaptive methods, self tuning regulation and approximate dynamic programming, to stabilize the two-wheeled self-balancing robot, Tumbller. The robot is modified to carry a varying number of quarters offset from its center of gravity. The self tuning regulator with equilibrium estimation is capable of stabilizing the robot with up to 21 quarters, both in simulation and experimentally. The approximate dynamic programming method implemented with a deep neural network as its value function approximator is currently capable of stabilizing the robot with up to 8 quarters, though further model training could improve these results.**

I. Introduction

Adaptive control techniques present an opportunity for control system engineers to design intelligent systems capable of handling time-varying changes to the dynamics of the controlled system. In the real world, many systems have unpredictable time varying components that may reduce the effectiveness of or completely invalidate a controller that is designed assuming time-invariant behavior. As a result, it is imperative that we have tools that can reduce our reliance on the time-invariant assumption.

In particular, we address in this paper two techniques that can be utilized to estimate and appropriately redesign controllers for unpredictable variations to the parameters of a system. We investigate the effectiveness of two unique adaptive control techniques for balancing an inverted cart-pendulum system with an unknown time-varying offset mass. The first technique uses recursive least-squares (RLS) to iteratively identify the model parameters of the robot. A state feedback gain found through the linear quadratic regulator (LQR) algorithm is then designed using the parameterized model to optimally stabilize the unstable system. We refer to this technique as the Self-Tuning Regulator (STR). The second technique, which we refer to as Approximate Dynamic Programming (ADP), assumes that the measured dynamics will fall within a predetermined set. We utilize a deep neural network to identify which element of this set most accurately describes the behavior of the system, and apply an LQR state feedback controller based on this identification.

Classical approaches to adaptive control are indirect methods which rely on a system model. These algorithms generally have two components, model estimation and controller design. Though this is a widely studied area, applications for self-balancing robots appear to be limited. Zad and Ulasyar [1] combine model predictive control with a static Kalman filter for parameter estimation to achieve adaptive regulation control of a self-balancing robot. However, their analysis is limited to a time-invariant system with unknown system parameters, a limitation imposed by the estimation method. Wu et al. [2] designed a fuzzy PD controller to achieve robust stabilization of an uncertain plant model. Robust control in general is an alternate field of approaches to this problem, but is likely to have a limited range of feasibility.

More closely related to the approach derived here, Kim and Ahn [3] apply self-tuning control to achieve adaptive tracking of a self balancing robot. However, this is only applied as an outer loop controller, the inner loop is a fixed-gain LQR controller. This limits how much uncertainty the system is able to handle. Finally, Anninga [4] implemented a self tuning regulator by combining pole placement with RLS to stabilize a self balancing robot with time-varying parameters such as added mass. However, this implementation is limited to systems with consistent equilibria, i.e. the additional mass must be aligned with the existing center of gravity.

ADP has grown over the last two decades with the increasing popularity of neural networks. ADP can be categorized into four main schemes: a heuristic dynamic programming (HDP), an action dependent HDP based on Q learning, dual HDP, and action-dependent dual HDP [5]. Most approximate dynamic programming approaches considered often take on the form of an action dependent HDP, with an action-critic neural network [5] [6]. The actor network approximates the mapping between states and control input, while the critic network takes the system as an input and outputs the estimate value function [6].

Li and Dong [6] propose a data-based scheme to solve the optimal tracking problem of switching between autonomous systems. To find the action value function, or Q function, an iterative algorithm based on ADP is formulated to optimally determine which modes to switch between [6]. They use a critic only, linear-in-parameter neural network, to implement the proposed algorithm. This paper follows suit in using a critic-only approach to optimally switch between subsystems and apply corresponding control inputs.

Heydari and Balakrishnan [7] propose a solution to the problem of optimal switching and control of a non-linear system

using approximate dynamic programming. They propose an algorithm for switching that determines the optimal cost-to-go as a function of the current state and the switching times. The neural network is trained to pre-determine and estimate a value function for optimal switching through dynamic programming. A second neural network is used to calculate the optimal control to be applied for a given region; the conjunction of the two neural networks being an actor and critic. Similarly, this paper follows the idea of optimally switching between subsystems based on the identification of the system, but explores the use of deep learning to optimally switch between pre-determined LQR state feedback controllers based on the identified system dynamics.

II. System Modeling

The system is modeled first in its nonlinear form using two dimensional rigid body dynamics. The simplifying assumption is made that the robot can only tilt and translate, described by ψ and x, respectively. The process begins with attaching reference frames to key portions of the robot as shown in Fig. 1, where W is the world frame, C the cart frame, P the pendulum frame, and D the disturbance mass frame. Nonlinear equations for the dynamics can be derived as the second-order differential equation

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + N(q,\dot{q}) = \Upsilon$$

with $q = \begin{bmatrix} x & \psi \end{bmatrix}^\top$, $M(q)$ is the mass matrix, $C(q,\dot{q})$ is the Coriolis matrix, $N(q,\dot{q})$ is the potential energy terms matrix, and $\Upsilon = \begin{bmatrix} F & 0 \end{bmatrix}^\top$ the generalized forces matrix containing the linear force generated by the motors F. The conversion between motor voltage and force is made by the linear approximation

$$F = \frac{2k_T}{Rr}V$$

where k_T is the motor torque constant, R is the motor resistance, r is the wheel radius, and V is the supplied motor voltage.

Finally, the differential equation is rewritten in the state space representation $\dot{z} = f(z,u)$ by first inverting the mass matrix M such that

$$\ddot{q} = M^{-1}(\Upsilon - C(q,\dot{q})\dot{q} - N(q,\dot{q}))$$

The states z are then defined

$$z = \begin{bmatrix} x & \dot{x} & \psi & \dot{\psi} \end{bmatrix}^\top$$

Though this nonlinear system definition is useful for realistic simulations, it is too complex to be used for controller design. As the fundamental control approach for this project is the LQR design, the linearized model of the system is more useful. Using a first-order Taylor approximation, a linearized model of the form

$$\dot{z} = Az + Bu$$

is found at the unstable equilibrium point, where u is the motor voltage. Full-state feedback is assumed for this system, i.e. $y = z$. The structure of the linearized model is the same for any equilibrium tilt angle, though the values themselves will differ. For the RLS design, the physical parameters making up each element of the matrices A and B are grouped together to form generic parameters θ_i. Since RLS is a discrete-time algorithm, a discrete model of the system is needed. To maintain the simplicity of the parameterized linear model, a backward difference method is used to convert to a discretized model.

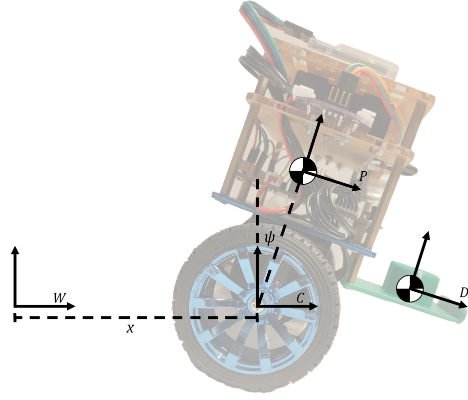

Fig. 1. Coordinate frames, center of mass locations, and system state definitions for the robot. The dynamic model of the system is derived using these definitions.

$$\dot{z} \approx \frac{z_k - z_{k-1}}{T}$$

$$z_k = (I + TA)z_{k-1} + TBu_{k-1}$$

One advantage of the backward difference approximation is that linearity in the generalized parameters is preserved. This means the discretized system can be equivalently represented as

$$z_k = \phi^\top(z_{k-1}, u_{k-1})\theta + \delta(z_{k-1}) = \phi_{k-1}^\top\theta + \delta_{k-1}$$

$$\phi_{k-1} = \begin{bmatrix} 0 & Tz_2[k-1] & 0 & 0 \\ 0 & Tz_3[k-1] & 0 & 0 \\ 0 & 0 & 0 & Tz_2[k-1] \\ 0 & 0 & 0 & Tz_3[k-1] \\ 0 & Tu[k-1] & 0 & 0 \\ 0 & 0 & 0 & Tu[k-1] \end{bmatrix}$$

$$\delta_{k-1} = \begin{bmatrix} z_1[k-1] + Tz_2[k-1] \\ z_2[k-1] \\ z_3[k-1] + Tz_4[k-1] \\ z_4[k-1] \end{bmatrix}$$

III. Controller Design

A. Self-Tuning Regulator

The first adaptive control method developed for stabilizing the robot under time-varying payloads is the self tuning regulator. This is an indirect adaptive method which relies on an estimate of the plant model to design an appropriate control method. As the true plant changes over time, the model

should update to reflect this, and a more appropriate controller be designed. The self tuning regulator is composed of model estimation and control design components. For this, RLS is used for model estimation and LQR is used for controller design.

RLS aims to approximate the solution of the least squares problem of minimizing the prediction cost in terms of the model parameters

$$J(\theta) = \frac{1}{2}\sum_{i=0}^{N} \lambda^{N-i}(y_i - \hat{y}_i)^2$$

where

$$\hat{y}_i = \phi^\top(y_{i-1}, u_{i-1})\theta + \delta(y_{i-1})$$

and $\lambda \in (0, 1]$ is an exponential forgetting factor. The term δ is an extension of the original algorithm taken from Astrom and Wittenmark [8] simply accounting for the bias terms which are not multiplied by any parameter, and is an artifact of maintaining the continuous-time parameters through the backward difference approximation. Computing the exact solution to the least squares problem becomes computationally expensive as data accumulates, making it infeasible for real-time implementation. However, it can be recursively approximated with a constant computational cost through the following algorithm.

$$K_k = P_{k-1}\phi_{k-1}(\lambda I + \phi_{k-1}^\top P_{k-1}\phi_{k-1})^{-1}$$
$$P_k = (I - K_k\phi_{k-1}^\top)P_{k-1}\lambda^{-1}$$
$$\hat{\theta}_k = \hat{\theta}_{k-1} + \rho K_k(y_k - \phi_{k-1}^\top\hat{\theta}_{k-1} - \delta_{k-1})$$

The scalar ρ is an additional learning rate extension from the original derivation used to tune how fast or slow the parameters converge. The algorithm is initialized with $\hat{\theta}_0$ set to the nominal linearized system parameters and $P_0 = p_0 I$ where p_0 is an arbitrarily large constant.

Using the continuous-time linear model estimated through RLS, a state feedback control law is generated using LQR. This LQR design step occurs at every sampling period along with the RLS model update. Every time the parameters are updated, they are used to generate a new state feedback matrix which is immediately applied to the online control system for the next sampling step.

Even though the linear model is a simplification of the true nonlinear dynamics, the difference is negligible when the robot is operating correctly near the equilibrium. While the RLS algorithm is able to estimate the linear model well, it does not provide any insight as to what the actual equilibrium point is. Specifically, the equilibrium tilt angle will vary since the additional weight is offset from the robot's center of mass, causing the combined center of mass to shift parallel to the ground. Even with a perfect system model, LQR alone is incapable of stabilizing the system if the equilibrium tilt angle is too far from the center.

To account for this, the equilibrium is separately estimated from the other system parameters and is used to define the tilt angle error for state feedback control. Intuitively, assuming the equilibrium is at the center, then additional mass will cause the robot to drift forwards. As the robot drifts, the equilibrium reference can be adjusted until the robot comes to a halt. This concept is manifested through the addition of reference dynamics

$$\dot{r}_\psi = -\alpha\bar{\dot{x}}$$

where $\dot{r}_\psi$ is the equilibrium reference for the tilt angle, α is an integration rate, and $\bar{\dot{x}}$ is an average over a fixed window of linear velocity samples.

B. Approximate Dynamic Programming

The second adaptive control method implemented to stabilize the Tumbller uses approximate dynamic programming to accurately predict the robot's current state in order to apply a precomputed controller to the feedback system. This direct adaptive control method incorporates a value function approximator to predict the current system dynamics. A neural network consisting of two 2D convolutional layers followed by two linear layers outputs a probability distribution across the precomputed system dynamics and tuned controller pairs. The most likely pair is selected and the appropriate control value is sent to the motors.

The goal of the neural network, as an approximate value function, is to accurately predict the correct controller to apply to the current system without explicit information about the current mass in the tray. In order to gather additional information from the four state measurements it sees at each time step, an additional time dimension is added to the data and states at the previous 299 time steps are stored and used as an input to the neural network. Fig. 3 shows the network architecture. Two-dimensional convolutional layers in the model use a filter of size (2 x 20) to convolve over the input. These layers use the time-dimension information to help determine the most probable current state based on how the Tumbller's four states change over time, in addition to their relationship with each other. Linear layers at the end of the network reduce the dimensions of the data to the number of controller state pairs available on the Tumbller which are considered the output classes. The cross entropy loss combines softmax and negative log likelihood. Softmax is used to normalize the final outputs and create a probability distribution across the output classes. Negative log likelihood calculates the loss value which is back-propagated through the model weights using gradient descent during training time. The size of each layer in the neural network as well as its depth is limited by the Raspberry Pi's 1GB RAM. When trained and validated on data collected on the hardware, the model achieved a final training accuracy of 97.38% and validation accuracy of 95.34% after training for 35 epochs, shown in Fig. 2.

Paired with the output predictions of the neural network are LQR controllers tuned in MATLAB. With the goal of robot stabilization, the Q and R matrices are chosen to place importance on the wheel angle θ. Six LQR controllers are computed on the continuous state-space, nonlinear-offset

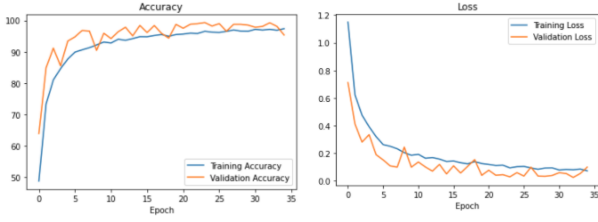

Fig. 2. Neural network training statistics. When trained over 35 epochs, the network reached a final training accuracy of 97.38% and validation accuracy of 95.34% which was calculated on data samples the network did not evaluate during training.

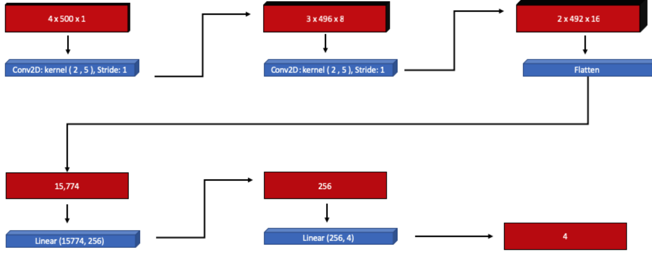

Fig. 3. The architecture of the neural network. The front of the network consists of two 2D convolutional layers to convolve over the time dimension of the inputs and extract information about the change in states overtime. The back of the network reduces the dimensions of the data to the output prediction size of 3. These 3 classes correspond to the 0, 4, and 8 quarter controllers respectively.

dynamics of the tumbller robot for a given mass in the tray container (i.e, 0,2,4,6,8,10 quarters in the tray) in MATLAB. The LQR controllers for each mass evaluate to

$$K_{0\ Quarters} = \begin{bmatrix} -0.316 & -2.598 & 113.942 & 4.866 \end{bmatrix}$$

$$K_{4\ Quarters} = \begin{bmatrix} -0.316 & -2.600 & 114.451 & 5.409 \end{bmatrix}$$

$$K_{8\ Quarters} = \begin{bmatrix} -0.316 & -2.602 & 114.937 & 5.847 \end{bmatrix}$$

It is evident that the gains are quite similar to one another, but the controllers themselves would be unable to stabilize the robot with any mass above 6 quarters. The weight offsets the robot's center of mass and thus, new equilibrium points are needed to balance about. The new equilibrium points were estimated through tuning, i.e. placing the respective amount of quarters in the mass tray, finding where the robot is in equilibrium, and writing down the value for theta that it is at rest. This is beneficial for developing stabilizing controllers for masses above 6 quarters, as it takes into consideration both the quarters and the added weight of the Raspberry Pi, which sits anterior to the mass tray.

IV. Simulations

For the STR method, the control algorithm was applied in MATLAB/Simulink to the nonlinear system described above. This was useful for verifying the original nonlinear model as well as how additional offset masses affect the dynamics, and in particular how the equilibrium point shifts. Beyond verification of the dynamic model, MATLAB simulations were not utilized for the ADP method. Since ADP is a direct control method trained on a finite set of LTI system models, this method was only simulated to verify correctness of implementation. All results for ADP are from hardware.

A. Self-Tuning Regulator

For the STR method, MATLAB simulations were used to verify the correctness of implementation as well as determine how effectively the algorithm is under pseudo-real world conditions. In addition to the complete nonlinear dynamic model, other physical nonlinearities including quantization of the control signal, control saturation, motor dead-band, and measurement noise are simulated. Measurement noise is added to each output as white noise. Since STR is an indirect method dependent upon the plant model, this simulation environment can provide a baseline for how effective the algorithm should perform.

Figs. 4, 5, and 6 show the results of applying the STR algorithm on the nonlinear robot dynamics. At different points of the simulation, masses simulating those of quarters are added and removed from the robot at an offset. The simulation parameters are $\lambda = 1$, $\rho = 1$, $\alpha = 0.4$, $Q = diag(1, 100, 10000, 100)$, and $R = 1$. By including the reference integrator, the robot is able to stabilize around each equilibrium point. Due to the noise and dead-band included in the simulation, the robot does not asymptotically converge to the equilibrium, but rather oscillates within some ultimate bound.

In simulation, any forgetting factor $\lambda < 1$ would destabilize the system after some time. Even in the real system, the smallest forgetting factor which preserves stability is around $\lambda = 0.999$. So, the parameters tend to converge close to their initial values and not change much as masses are added. Interestingly, some of the parameters tend to converge to

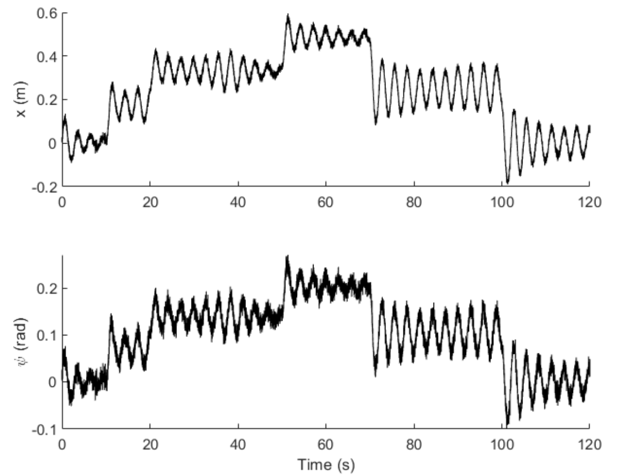

Fig. 4. States x and ψ of the STR system in simulation when applying periodic changes in mass. The reference integrator allows the system to find new equilibria with each change in mass.

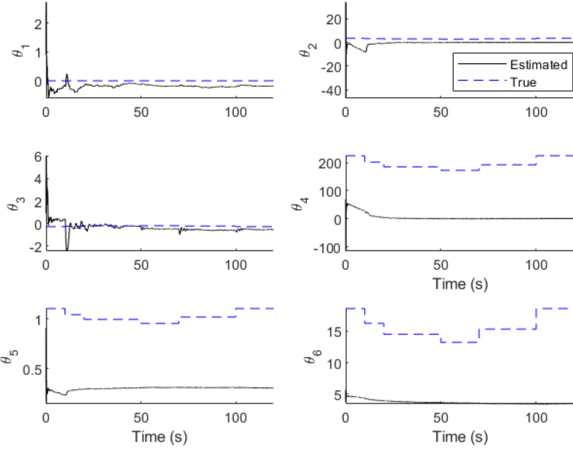

Fig. 5. Parameters θ of the system in simulation when applying periodic changes in mass. The estimated parameters rarely converge to the true parameter values.

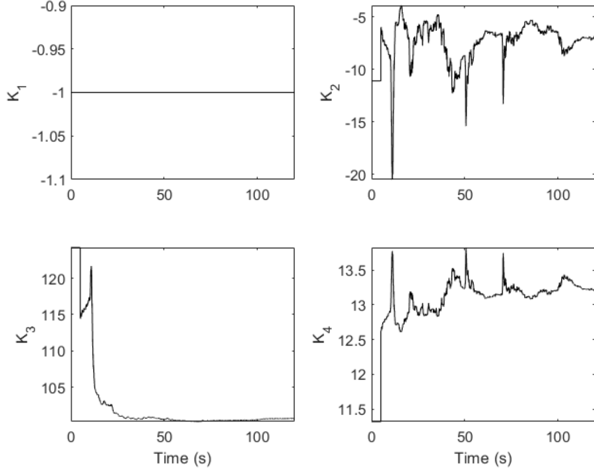

Fig. 6. State feedback gains K of the STR system in simulation when applying periodic changes in mass. The feedback gains K_2 and K_4 show sharp spikes throughout the simulation.

values very far from the true system parameters even though the system is still performing well. This is due to the accumulation of approximations made for the RLS estimation including linearization and backward difference discretization. Additionally, RLS is not intended to deal with measurement noise, which could also be contributing.

Some of the controller gains vary from their original values, but are all within the same orders of magnitude. When the offset mass is changed discontinuously, it causes a temporary spike in the controller gains before they settle again. The RLS estimation works well for slowly-varying parameters, but has unpredictable behaviour for fast and discontinuous changes. Yet, it seems that the controller should still be able to recover and stabilize the system.

B. Approximate Dynamic Programming

Simulation of the neural network output was performed in Python using experimentally collected data that the network had not been exposed to during training, i.e. a test subset. Separate runs with different mass values were concatenated together to simulate a step mass change to the system. A sequence of 300 time steps was taken as the input to the network and a moving window captures the sequence, including the transition states at the simulated step points. The network's predictions along with the correct labels corresponding to the actual number of quarters and corresponding controller applied were plotted. Every 700 time steps, a new mass and controller pair were simulated as the robot system. Fig. 7 shows the network's predictions to the step data. The network maintains a 99.1% prediction accuracy over these samples and has difficulty classifying during the transition from one mass to another, which is illustrated by the blue prediction dots that are offset from the actual controller trajectory.

V. Hardware Implementation and Testing

A. Sensing and Actuation

Several sensors and actuators are used on the Tumbller to ensure complete controllability and observability over the robot. To control the robot, the original pair of DC gearhead motors provide torques to the wheels, which directly correspond to friction forces along the ground. Even though there are two motors enabling the robot to travel anywhere on the ground, it is assumed that the robot is constrained to a straight line corresponding to the planar dynamics modeling.

An inertial measurement unit (IMU) containing a triaxial accelerometer and gyroscope is used to estimate the robot tilt angle and angular velocity. The angular velocity is measured directly from the corresponding gyroscope axis, and the tilt angle is estimated using a complementary filter combining the accelerometer axes as gravity vectors. The low-pass filtering of the accelerometer and gyroscope on the IMU is sufficient, so no additional measurement filtering is used. The IMU communicates as an I^2C slave device.

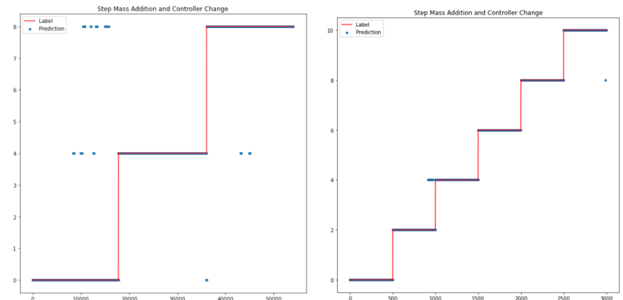

Fig. 7. Networks trained for differentiating between 0, 2, 4, 6, 8, and 10 quarters as well as 0, 4, and 8 quarters showed high accuracy when predicting the step change produced by adding either 2 or 4 quarters to the tray for respective networks. Both networks misclassified points around the step, but remained accurate for the unchanging systems. The blue dots/lines are the predictions of the neural network and the red line is the correct corresponding labels.

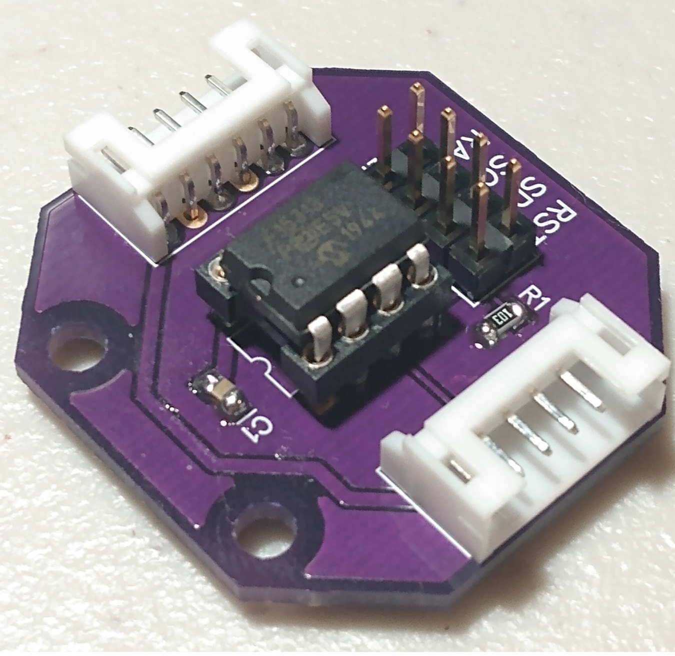

Fig. 8. Custom PCB with PIC16F15313 8-bit microcontroller used to read the full motor quadrature. The microcontroller interfaces over I^2C.

To measure the linear position and velocity of the robot, the built-in quadrature encoders on each motor are used. Since the original robot hardware does not support the full quadrature interface, a separate interface using a PIC microcontroller was custom designed to read the full quadrature. This doubles the measurement resolution from 780 CPR to 1560 CPR and also provides the true motor direction. The linear velocity is estimated as the backward difference of encoder counts between samples, and the position is estimated as the Euler integration of velocity. The PIC encoder interfaces communicate as I^2C slave devices.

B. Computation

The Arduino Nano on board the Tumblller is used to read the sensor measurements, calculate the state feedback control law, and apply the control output signal. It acts as the I^2C master for interfacing with the sensors. For both the STR and ADP methods, the remaining computations are too complex to be completed on the Arduino. Therefore, a Raspberry Pi is added to the system to perform larger computations. The Arduino sends the measured states and applied controls to the Pi, and the Pi computes the state feedback matrices to be sent back to the Arduino. The Arduino and Pi communicate over UART.

For the STR method, the Arduino first reads in a new set of state feedback gains which the Raspberry Pi has previously computed. It then reads the sensors and sends the measurements along with the previously applied control value back to the Pi. During the remainder of the 15 ms sampling period, the Arduino applies the new state feedback control law while the Pi is processing the new data. On the Pi, the NumPy Python library is used to iterate through the RLS parameter update. This new model is used with the Python Control Systems Library to solve the infinite-horizon continuous-time LQR problem to generate a new state feedback gain matrix. These gains are queued to be read by the Arduino during the next sampling period. The Arduino also applies the reference integrator dynamics during this time.

Before the full STR algorithm is run online, the initially designed LQR state feedback gains are used for some time while the RLS algorithm updates the parameters. Doing this allows the model parameters to begin converging asymptotically to their final values before using them to redesign the control law. This is necessary since for the first few seconds of operation the parameters tend to change rapidly in an extremely large range. Giving time for these to converge before using them for controller updates prevents the system from going unstable during this time. Additionally, a sinusoidal signal is applied as a control disturbance during this phase to provide persistent excitation, ensuring the data is rich enough for the model parameters to converge quickly.

When testing the STR method on the physical robot, the controller parameters are set to $Q = diag(1, 1, 2.5e6, 1000)$, $R = 2$, $\lambda = 0.999$, $\rho = 0.1$, $\alpha = 4e-5$, and a velocity averaging window length of 8 samples.

For the ADP method, the Arduino initializes the system by applying the zero-quarter LQR controller to stabilize the system. The Arduino continuously relays the current state measurements (linear position, linear velocity, tilt angle, and tilt velocity) to the Pi at a sampling rate of 10 ms. This ensures that there is enough time to communicate and allow for the model to run through the neural network that is on the Pi. On the Pi, the system initially gathers 300 points of data, the states of the robot, that act as the sequence that is then fed into the neural network. The model update is done every 10 timesteps to reduce computational intensity and lag, with the newest 300 points of data being used for every model update. After the 300 points of data are collected, the model outputs a set of prediction probabilities that correspond to the probability that the system is in a given set of dynamics. The highest prediction probability is taken and then used to select an LQR controller and reference point adjustment; where the LQR controller and equilibrium reference points are correlated to one another. The chosen LQR controller is then multiplied across the states to retrieve a control input. The control input is then multiplied by $\frac{256}{8}$ in order to convert it into PWM values, and sent back to the arduino to execute the low level hardware command.

C. Offset Mass Application

The offset mass is applied to the system by placing United States quarter dollar coins in a receptacle that is attached to the front of the robot. It can hold up to 21 coins simultaneously, corresponding to a total of roughly 0.119kg (5.67g per coin).

VI. Results

A. Self-Tuning Regulator

Using a combination of the self-tuning regulator and the equilibrium estimator, the robot is successfully able to balance upright with 0 - 21 quarters in its tray. This corresponds to 0 - 0.119kg, neglecting the mass of the tray itself. The robot iteratively adapts the parameters of the system on-line and simultaneously adapts its equilibrium point to avoid falling.

We find that the parameters to which the RLS algorithm converges to are highly unpredictable and often nonsensical. Fig. 9 illustrates this result, showing $\theta_n(t)$ for the system when zero, four, and eight quarters are placed in the tray. In each case, the parameters converge to drastically different values many orders of magnitude higher than their starting estimate. More surprising, θ_2 and θ_4 converge to positive values when zero quarters are applied, but converge to negative values in the cases of four and eight quarters.

A potential cause of the unpredictable steady-state parameters may lie in the fact that parameters are treated as independent values, despite being highly interdependent. Each parameter represents a non-zero element of the linearized dynamic matrices A and B, which in turn are complex polynomial functions of the true system parameters (masses, moments of inertia, and center of mass positions). The values of θ should therefore be constrained to change together, rather than separately. Additionally, unmodeled dynamics such as motor backlash and sensor biases are likely contributing to the amount of uncertainty in the measurements. Since RLS is a deterministic algorithm, the accumulation of uncertainties can lead the parameter estimation to yield inaccurate results.

It is also apparent from Figs. 10 that during initial estimation, the parameters exhibit transient spikes before settling at a steady value. This behavior is expected as RLS typically performs best when parameters change smoothly with time, so step changes cause unpredictable behavior during parameter

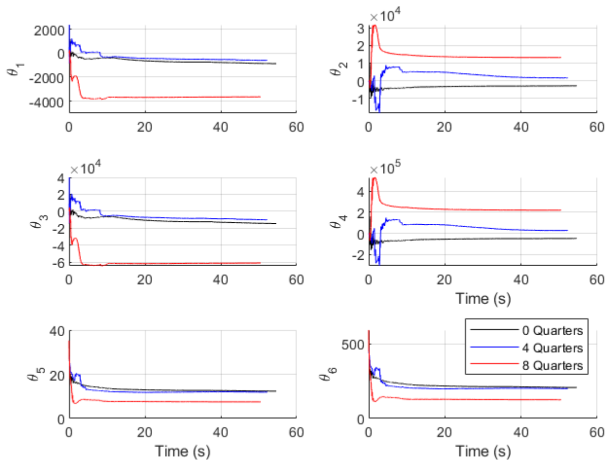

Fig. 9. Parameter estimates computed on the robot for 0, 4, and 8 quarters in the tray. The parameter estimates converge, but often to nonsensical values.

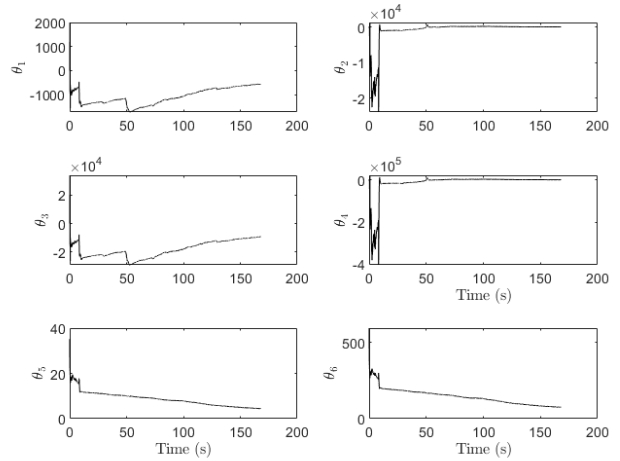

Fig. 10. Parameter estimates over time for the STR with the reference integrator. Parameters exhibit transient spikes initially before settling. Parameters continue to update as coins are added

estimation. In our experiments, coins are added in steps, giving the time-varying parameters sharp discontinuities. To minimize the impact of this transient behavior, the parameter learning rate ρ is set to slow down the parameter convergence, which has been seen to significantly improve the transient performance. Keeping the state feedback gains constant for the initial period of time and providing an initial disturbance to enforce persistent excitation both ensure the robot remains stable while the parameters settle through the transient stage.

From Figs. 11 and 12 elements of the LQR gain matrix K also exhibit unpredictable behavior, but are nonetheless capable of stabilizing the system. Likely, the sharp discontinuities happen with a short enough duration that it does not immediately destabilize the robot. However, this indicates that the system, as currently designed, cannot be relied on to robustly stabilize the system.

We also note that the inclusion of a forgetting factor less than 1 in the physical robot causes the system to be unstable. The smallest forgetting factor achieved experimentally was 0.999, which does not allow for very fast exponential decay. These results can be further supported by the work done by Fortescue, et al. [9] which indicates that stability is not guaranteed when RLS utilizes a forgetting factor.

To evaluate the performance of the self-tuning regulator, we perform two comparisons. First, the system is compared to an unmodified LQR controller tasked with regulating the system to the unstable equilibrium $z = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^\top$. As quarters were added one-by-one, the robot began to show larger oscillation magnitudes. For five or more quarters, the robot was no longer able to stabilize and crashed forwards. The standalone LQR controller is incapable of stabilizing an uncertain mass when this causes the equilibrium itself to change. Clearly our STR algorithm is able to provide more stability coverage than the standard LQR controller.

Second, the system is compared to the same LQR controller

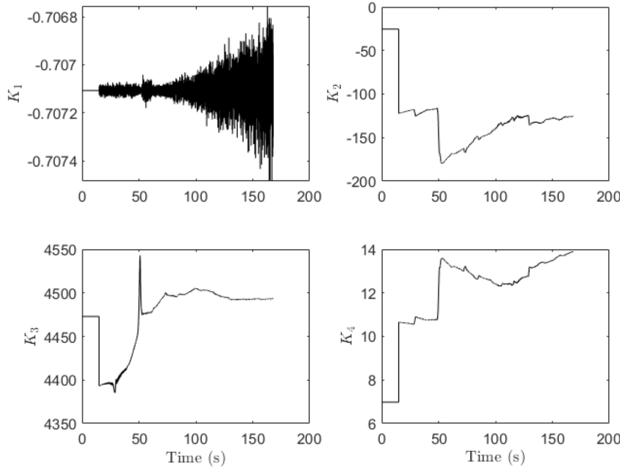

Fig. 11. Controller gains over time for the the STR with the reference integrator. The gains exhibit erratic behavior but are nonetheless capable of stabilizing the system.

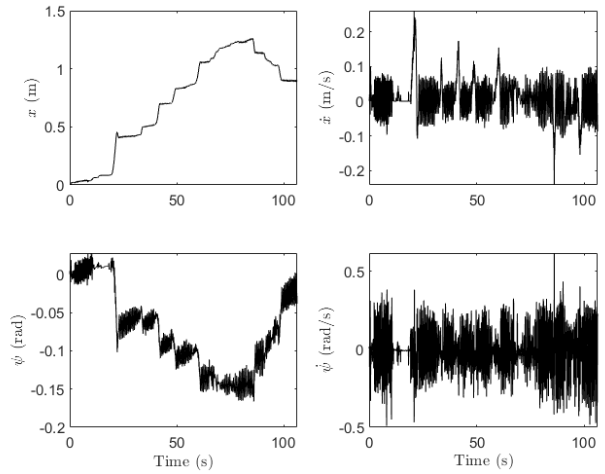

Fig. 13. States over time for the non-adaptive LQR controller with the reference integrator. The system is capable of stabilizing for up to 21 coins without adapting its parameters.

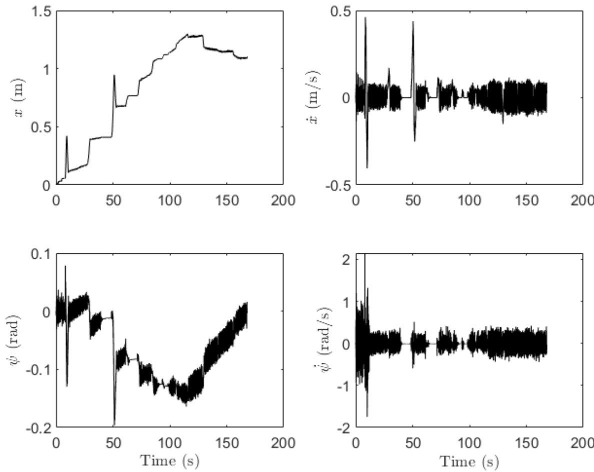

Fig. 12. States over time for the STR with the reference integrator. The system remains stable as the parameter estimates and controller gains adapt. The system identifies and regulates to a changing reference angle

but with the reference integrator to identify the time-varying equilibrium point. The state trajectories over time for this controller can be seen in Fig. 13. With the inclusion of the reference integrator, the controller was capable of stabilizing with any number of quarters. As quarters are added, the robot would drift forward until the new equilibrium was found, tilting the robot backward until it stabilized. It is expected that as mass is added, the frequency of oscillation would decrease. However, the relative mass of the quarters compared to that of the robot is not enough to observe this.

From this experiment, we notice that the performance of the fixed-gain LQR compared to the full STR method is nearly identical. For some quarter values the STR method was able to truly stabilize the system without any oscillations, however this is likely due to chance since in general it exhibited similar oscillations to which the fixed-gain method did. Therefore, it seems that since the additional masses are offset from the robot centroid, it is the reference integrator which is providing a majority of the benefits. Though the LQR redesign based on RLS parameter estimation is updating the state feedback gains over time, it seems to have much less impact on the stabilization performance.

B. Approximate Dynamic Programming

Using the pre-trained network of weights for the optimal switching between subsystems, the robot was able to switch between subsystems, although inconsistently. The robot was loaded at the 40th timestep and the 90th time step with 4 quarters each time. The neural network's task was to identify and switch between the controllers and reference point of the 0, 4, and 8 quarter dynamics. It should be noted that the predictions are made every 10 steps of the 10ms main time loop, so a controller update occurs every 100ms.

The ADP method does not perform as well as expected given the high validation accuracy of the neural network outputs. This is most likely due to the hardware and the inconsistent state information for the same mass/controller pairs used for training and testing. The difficulty of duplicating the same scenarios the robot will see at run time was difficult, even with keeping all conditions the same (i.e. same place on the carpet, same controllers, same references, and a fully charged battery). Although there were difficulties, it can be seen in Fig. 14 that in fact when there were quarters added to the tray, it was able to identify the gap and be able to switch. It did not respond as consistently when another 4 quarters were added for a total of 8 quarters. It was able to recognize the change in system dynamics and switch to the correct 8 quarter subsystem temporarily, but ultimately reverted back to predicting the 4 quarter subsystem.

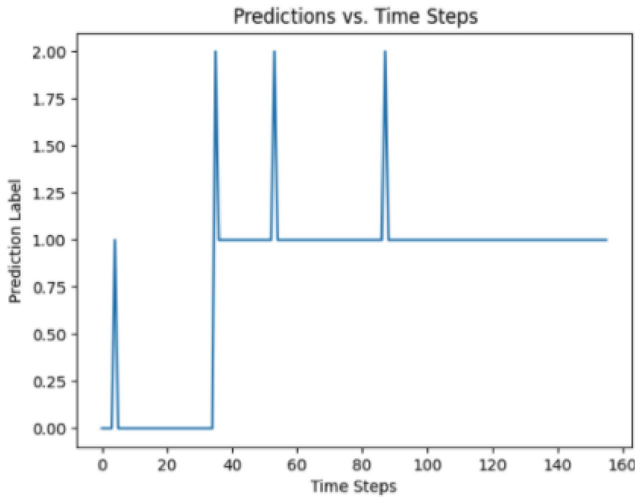

Fig. 14. Graph illustrates the prediction labels over the timesteps taken. ADP was applied to the robot to switch between 0 quarters, 4 quarters and 8 quarters. Circa timestep 40 is when 4 quarters were placed and timestep 90 is when an additional 4 quarters were placed.

In order to get a better look at exactly how the states were affected and how the stability of the system reacts to ADP, the states of the system were plotted over the time steps. Illustrated in Fig. 15, are the side by side profiles of the states from the implementation of the ADP controller and the non ADP controller. For the non ADP controller, a consistent zero-quarter LQR controller and reference was used. Compared to the ADP controller, the non-ADP controller was more unstable and oscillated back and forth with larger amplitudes. The wheel velocity state, which represents how fast the tumbller is rocking back and forth, on the non-ADP controller is much more sporatic vs. the ADP controller tumbller. The ADP controller was able to minimize the rocking back and forth by being able to switch to a more stable controller, but more importantly, shift its equilibrium point back so that it would not oscillate as much.

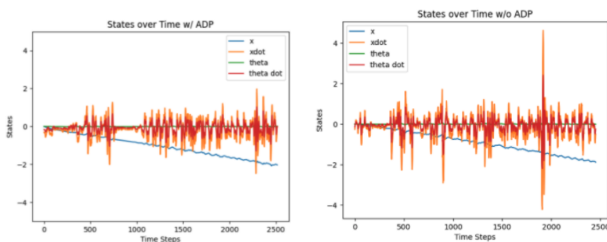

Fig. 15. The side by side graphs depict the control implementation with ADP (on the left) and without ADP (on the right) of the states (linear position, linear velocity, tilt angle, tilt velocity) over the given timesteps. The controller without ADP is the zero-quarter LQR controller. ADP seems to be able to switch subsystems enough where it is able to stabilize the changing weights (0,4,8 quarters) better than a constant LQR controller. Four quarters were dropped at the 400th timestep and the 900th timestep.

VII. Conclusion

At the outset of this investigation, the goal was to implement and compare the performance of two unique adaptive control strategies for stabilizing the inverted cart-pendulum system with an arbitrary offset mass. STR is an indirect method involving continuous tuning of an optimal regulator by iteratively estimating parameters of the system, while ADP is a direct method which bypasses parameter estimation and directly chooses an optimal controller based on the dynamics of the system.

The two adaptive algorithms have been evaluated largely independently, but the results from testing indicate several key differences in the methods. Both show promise for being able to stabilize the unstable system in the face of time-varying parameter changes. However, the recursive least squares algorithm is better suited to slowly changing system dynamics. As a result, it suffers from transients in plant estimation that may produce unstable behavior. Additionally, even when it settles at parameter estimations, it may take many time steps to do so. In comparison, the ADP approach makes a set of assumptions about how the dynamics may change in the future, which leaves it vulnerable to inconsistencies that plague the system between training data and testing implementation. In contrast to the STR method, these anticipated changes are capable of switching control strategies within a single timestep.

Both strategies introduce an added layer of tuning to the overall control strategy. In the case of the STR, there are several learning rates, the forgetting factor, and initializations which have a significant impact on how well the system performs. Additionally, extra care must be taken with transient behavior in the parameters and providing persistent excitation to ensure the parameters used for LQR design are accurate enough to stabilize the system. For the ADP method, the use of a neural network for approximating the value function introduces many weights and network parameters which must be tuned to ensure accurate training of the model.

Our current set of experiments demonstrate that the STR strategy with a reference integrator is capable of stabilizing for the full range of 0-21 coins, while the ADP strategy can only stabilize for a range of 0-8 coins. However, we believe that slight changes to the data collection method for neural network training can increase the stabilizing range of the ADP method. Training the neural network to classify or predict that the system has seen a change in mass could improve its performance. This can be implemented by collecting experimental data as mass is added to the tray, as mass in the tray remains constant, and as mass is removed from the tray and labeling the data as separate classes "increasing mass", "constant mass", and "decreasing mass". During runtime, these predictions can be used to adjust the current controller values appropriately. Since the ADP method uses LQR controllers tuned for predetermined system dynamics based on the expected mass in the tray, tuning stable controllers for the Tumbller system with up to 21 quarters could serve as an additional improvement, allowing it to successfully control this larger mass range.

References

[1] H. S. Zad and A. Ulasyar, "Adaptive control of self-balancing two-wheeled robot system based on online model estimation," in *10th International Conference on Electrical and Electronics Engineering)*, Bursa, Turkey, Nov. 2017, pp. 876–880.

[2] W. Z. J. Wu and S. Wang, "A two-wheeled self-balancing robot with the fuzzy pd control method," *Mathematical Problems in Engineering*, vol. 2012, 2012.

[3] S. Kim and C. K. Ahn, "Self-tuning position-tracking controller for two-wheeled mobile balancing robots," vol. 66, pp. 1008–1012, 2019.

[4] J. R. Anninga, "Implementation of an adaptive controller on a ball balancing robot," Master's thesis, Delft University of Technology, Delft, NL, 2019.

[5] A. Al-Tamimi and F. Lewis, "Discrete-time nonlinear hjb solution using approximate dynamic programming: Convergence proof," in *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, Honolulu, USA, 2007.

[6] X. Li and C. Sun, "Data-based optimal tracking of autonomous nonlinear switching systems," *IEEE/CAA Journal of Automatica Sinica*, vol. 8, no. 1, 2021.

[7] A. Heydari and S. N. Balakrishnan, "Optimal switching and control of nonlinear switching systems using approximate dynamic programming," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 6, p. 1106–1117, 2014.

[8] K. H. Astrom and B. Wittenmark, *Adaptive Control*, 2nd ed. Mineola, NY: Dover, 2008.

[9] L. S. K. T. R. Fortescue and B. E. Ydstie, "Implementation of self-tuning regulators with variable forgetting factors," *Automatica*, vol. 17, pp. 831–835, 1981.