Platform for Learning Robot Dynamics and Comparing Methods

Nathan Lambert; University of California, Berkeley

***Abstract*—This project creates a model-based reinforcement learning framework for simulating robots and learning dynamics. The contributions are simple dynamics models, controllers for exploring state spaces, learning classes for modeling the system dynamics, and controllers for performing tasks on the learned dynamics. The dynamics files included are for a Crazyflie and the Ionocraft [1], [2], with a class structure for adding more. The system learns dynamics by exploring around hover for these flying robots. We compare the effectiveness of simple least squared (LSTSQ) and neural network (NN) solutions for modeling dynamics used to accomplish simple tasks with Model Predictive Control (MPC). This simulation platform is the basework for testing models to learn dynamics to train the currently uncontrollable Ionocraft to hover and explore model-based reinforcement learning for robots without access to high-fidelity state variables. We also qualitatively describe initial evaluations of the dynamics models presented here.**

I. Introduction

Reinforcement learning (RL) conveys the opportunity for new agents to accomplish tasks in unknown environments without pre-described instructions, which has clear value to numerous applications. RL is often broken down into two categories: model-free learning, where the system is given goals and attempts to learn a policy via a reward function directly and model-based learning, where a system attempts to learn a dynamics model for future use. Current state of the art algorithms in model-free RL, such as OpenAI's Proximal Policy Optimization (PPO) [3], often focus on digital environments where the cost of obtaining data is low. Alternatively, model-based learning has been explored in environments with higher costs-per-run. Deisenroth and Rasmussen showed the feasibility of learning simple physical tasks in under a minute using Gaussian Processes (GP) with their algorithm Probabilistic Inference for Learning Control (PILCO) [4].

Building on PILCO and algorithms for low sample dynamics learning, numerous applications in exploring new platforms for these algorithms exists, such as gait discovery on quadrupedals [5], [6] and hexapods [7]. Work in quadrotors is of interest to see how new algorithms can perform on tasks that have been extensively studied in control. Bansal et al. shows the capabilities of learning generalized trajectories on quadrotors without explicitly training on trajectories of the same complexity [8]. Online Bayesian optimization can be used to learn dynamics in real time that give best control performance, proposed as Dynamics Optimization via Bayesian Optimization (aDOBO) [9]. The standard approach for control on the learned dynamics of these systems is with MPC [10].

Our contribution is in the platform to explore above algorithms. The simulation environment makes is simple to experiment with different objective functions on the MPC or different state variables to learn the dynamics. Changing the objective function explores if the learned dynamics suitably cover the state spaces to accomplish diverse tasks. Changing the state variables allows users to explore the feasibility of accomplishing tasks on learned dynamics without highly accurate state feedback systems such as VICON.

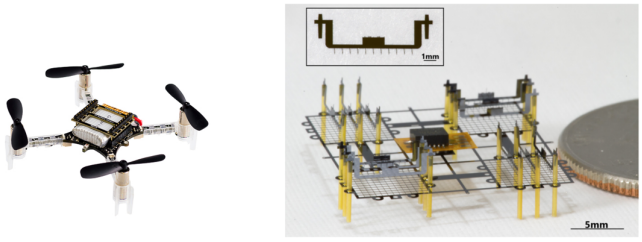


Fig. 1. Physical systems modeled - left: Crazyflie 2.0 quadrotor, right: ionocraft [1]

II. Robotics Simulation

A. Ground Truth Dynamics Model

The dynamics models (as in [11], [12]) we use to simulate the ionocraft and the crazyflie in Python are based on the following equations, where $\vec{p_g}$ is the position of the ionocraft with respect to the global frame, $\mathbf{R}$ is a matrix that maps vectors from the body fixed frame to the global frame, $\vec{v_b}$ is the velocity of the ionocraft with respect to the body fixed frame, $\vec{F}_{ext}$ and $\vec{M}_{ext}$ are the external forces and moments on the ionocraft with respect to the body fixed frame, $\mathbf{I}_b$ is the moment of inertia matrix, $\vec{w}_b$ are the angular velocities of the ionocraft with respect to the body fixed frame, $\vec{\xi}$ are the Euler angles, and $\mathbf{W}$ is a matrix mapping the angular velocities to Euler angle rates.

$$\dot{\vec{p_g}} = \mathbf{R}\vec{v_b} \tag{1}$$

$$\dot{\vec{v_b}} = \frac{1}{m}\vec{F}_{ext} - \vec{w_b} \times \vec{v_b} \tag{2}$$

$$\mathbf{I}_b\dot{\vec{w}_b} = \vec{M}_{ext} - \vec{w_b} \times \mathbf{I}_b\vec{w_b} \tag{3}$$

$$\dot{\vec{\xi}} = \mathbf{W}\vec{w_b} \tag{4}$$

The four individually controllable thrusters each generate a force $F_{i,ion}$ where $i \in \{1, 2, 3, 4\}$. This allows roll and pitch

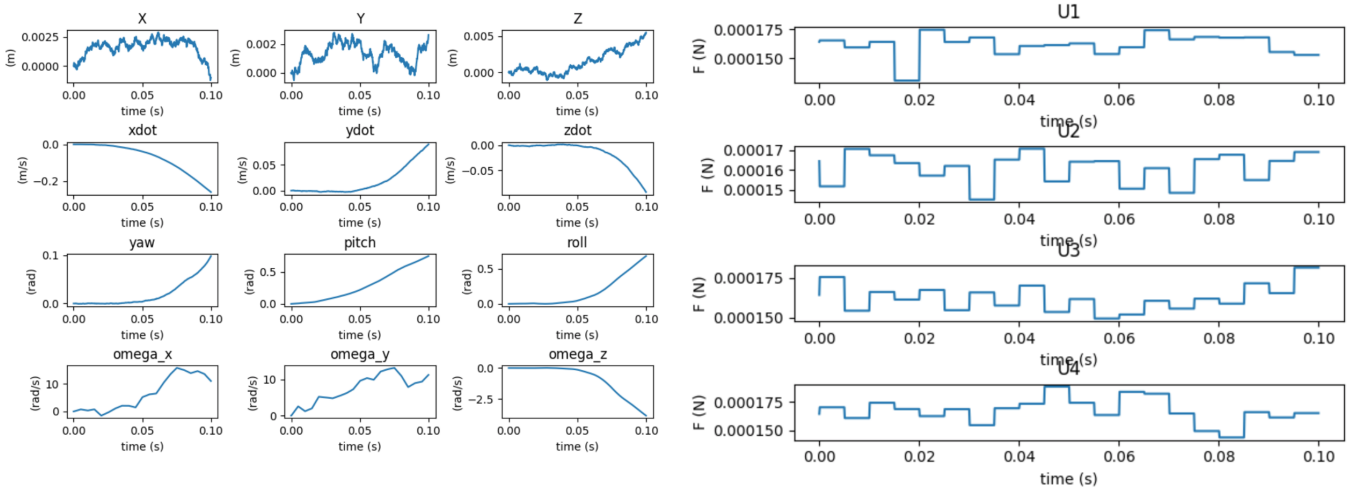


Fig. 2. Example of simulated ionocraft dynamics: Left, dynamics of a trajectory hovering. Right, inputs supplied to do so.

to be independently controlled [11]. Traditional quadrotors are able to control yaw by individually changing the angular momentum of each of the spinning rotors; a current limitation of the ionocraft design is that yaw cannot be controlled independently of roll and pitch because the thrusters do not intrinsically generate angular momentum. This lack of yaw control yields a rank deficient controllability Gramian.

Using the dynamics model above, a linear operator relating the thrust forces F_i to body fixed-frame z-axis thrust T and torques τ_x, τ_y, and τ_z can be generated. For a standard quadrotor, where body forces are proportional to the angular velocity squared of each motor, the map is:

$$\begin{bmatrix} T_z \\ \tau_z \\ \tau_y \\ \tau_x \end{bmatrix} \propto \begin{bmatrix} 1 & 1 & 1 & 1 \\ -c & c & -c & c \\ lx & lx & -lx & -lx \\ -ly & ly & ly & -ly \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \tag{5}$$

For an ionocraft with four thrusters pointing in the body frame's z direction, this becomes the following equality:

$$\begin{bmatrix} T_z \\ \tau_z \\ \tau_y \\ \tau_x \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ lx & lx & -lx & -lx \\ -ly & ly & ly & -ly \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \tag{6}$$

It can be seen that the map for the ionocraft prevents control authority over τ_z, which corresponds to yaw. The parameters lx and ly are defined as the distance from the thruster's center of mass to the body fixed-frame's center of mass in the x and y directions, respectively.

B. Hovering Controller Design

In order to hover, the controller stays around zero centered inputs for $\ddot{z}$, pitch, and roll. The equilibrium input for the Ionocraft is $F_{i,e} = \frac{mg}{4}$ for all $i \in \{1, 2, 3, 4\}$, and for the Crazyflie it is similar. This provides a stable hover equilibrium point. Each of the three PID controllers (shown in Fig. 3)

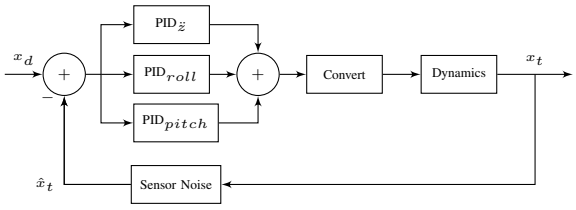


Fig. 3. Overall structure of the hovering control scheme. The difference between the estimated state at time t, $\hat{x}_t$ and the global desired state x_d is fed through a single PID loop with 3 independently tuned controllers.

regulates the error between the current state and the desired state, and has a z-transform of the form:

$$\frac{\tilde{u}_i(z)}{e_i(z)} = k_p + k_i T_s \frac{1}{z-1} + k_d \frac{N}{1 + NT_s \frac{1}{z-1}} \tag{7}$$

For the Ionocraft, The scalar PID output is transformed into a vector of desired input voltages by three individual K transformation matrices.

$$u_i = \begin{bmatrix} \delta F_1 \\ \delta F_2 \\ \delta F_3 \\ \delta F_4 \end{bmatrix} = K\tilde{u}_i \tag{8}$$

$$K_z = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; K_{pitch} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \; K_{roll} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \tag{9}$$

These three K matrices map a PID output to a change in input for all four thrusters. The output of the three transformed PID outputs, u_1, u_2, u_3 are summed together to form the input into the conversion block. Each thruster has a maximum force it can provide (approximately 0.25mN); this saturation makes the inputs nonlinear.

C. Learned Dynamics Models

Here, the state at time t is $x_t \in \mathbb{R}^d$, where d is a design parameter, the dimension of the state space, and $u_t \in \mathbb{R}^l$ being the inputs, which will depend on the robot and it's control scheme. The ionocraft and quadrotor dynamics model will begin as the following 12-dimensional state vector:

$$x := \begin{bmatrix} p \\ v \\ \zeta \\ \omega \end{bmatrix} \tag{10}$$

Where $p := (X, Y, Z)$ is the inertial reference position, $v := (\dot{x}, \dot{y}, \dot{z})$ is the body frame velocity vector, $\zeta := (\phi, \theta, \psi)$ is attitude Euler angle vector, and $\omega := (\omega_x, \omega_y, \omega_z)$ is the body frame rotational velocity vector.

We will formulate the problem as fitting a discrete time, potentially non-linear, state update function f

$$x_{t+1} = x_t + f(x_t, u_t) \tag{11}$$

In order to accurately model this function, the exploration of the state space when learning dynamics must be complete, otherwise the function $f(\cdot, \cdot)$ will model a subset of the dynamics. For example, if you only take random actions around hover, called bubbling, the learned model may only function around hovering. If a weak controller can be used to explore the state space slowly, it can improve the expression of the learned dynamics towards the true dynamics. The randomness in actions boils down to a bias verses variance trade-off in the training data. A biased model will be task specific. This illustration is seen below in Fig. 6, and is more important when training robots to accomplish diverse tasks on a single set of training data. This project focuses on simple objective functions, so this is not a limiting factor.

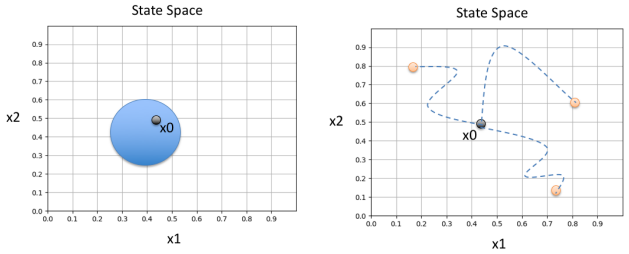


Fig. 4. Conceptual illustration of the importance of explicitly exploring state spaces (illustrated in lower dimensions). Left: random exploration around equilibrium, right: naive planed trajectories. The blue area or dashed lines show how the systems could explore state spaces. With a even poorly tuned controller, more of the dynamically feasible region can be explored. As dimensions scale, the volume of states explored around hover decreases exponentially, so the importance of improved exploration increases.

Code for this section is found in the files: dynamics.py, dynamics-ionocraft.py, dynamics-crazyflie-linearized.py, controllers.py

III. Model Learning

The premise of this project is to train different f to fit the dynamics while trying to achieve the control discussed below. When training these models, they will be minimizing a cost to best fit the training data. If one passes easy to model data in, such as the relationship between global velocity and position at the next time step, the model will waste expression on the relationship we already know. The importance of the model is to find the interaction between forces and states that we may not easily be able to model with disturbances or highly nonlinear dynamics. A accurate learned dynamics will incorporate our known knowledge of dynamical systems with learned non-linearities. An example of this would be when training for hover, the user may not want to include variations on z because we are only considering stability and the inputs will be around the hover point for z regardless. Formulation for a dynamics model learning only force and torque interactions will follow the setup in [8] below in (12), where α_i are separate learned parameters:

$$\dot{x} = \begin{bmatrix} \dot{p} \\ \dot{v} \\ \dot{\zeta} \\ \dot{\omega} \end{bmatrix} = f(x, u; \alpha) = \begin{bmatrix} v \\ f_v(x, u; \alpha_1) \\ R\omega \\ f_\omega(x, u; \alpha_2) \end{bmatrix} \tag{12}$$

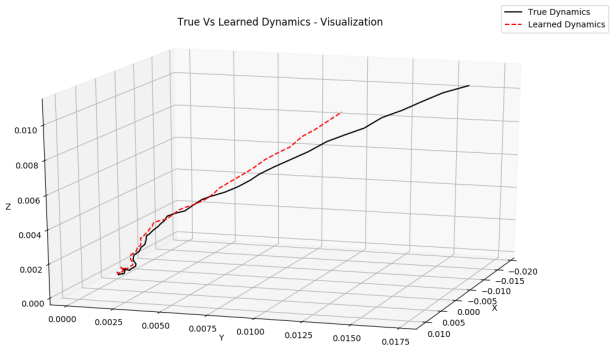


Fig. 5. An illustration of the accuracy of a learned dynamics model. For certain tasks, the divergence of the learned model would not guarantee failure.

A. Least Squares

Using least squares, one can quickly linearize a dynamics model based on the training data to form a linear system of the form

$$\dot{x}_t = Ax_t + Bu_t \tag{13}$$

for constant matrices $A \in \mathbb{R}^{d \times d}$ and $B \in \mathbb{R}^{d \times l}$. This follows from the closed form solution of least squares

$$\underset{w}{\operatorname{argmin}} \|Xw - y\|_2^2 \tag{14}$$

Here, X is the stacked training data vectors of the shape $\begin{bmatrix} x_t, u_t \end{bmatrix}^T$, y is the change in state $\begin{bmatrix} x_{t+1} - x_t \end{bmatrix}$ and w is the stacked matrices $\begin{bmatrix} A, B \end{bmatrix}^T$. Least squares has no strengths other than it's ease of implementation. It does not encompass nonlinearities or probabilistic outputs.

B. Neural Networks

Neural networks were the primary method employed for this project. The main reasons behind this design choice are the relatively rapid computation, current research interest within academic, and ability for generic function approximation. While the neural network may be able to fit a simple next state update model with high training or cross validation accuracy, when the model encounters states not trained for, it can capitulate completely. This is because NNs have complete confidence in it's output and has no way on indicating that a passed state was not trained for. This drawback is why GP have been successful and re-iterates the importance of diverse state space exploration when training for varied tasks. NNs are also very prone to overfitting and do not have explicit gradient information for control tools (MPC, iLQR, nonlinear methods). NNs also scale relatively well for higher dimensional systems, so computation of dynamics within MPC is less of a concern. In addition, NNs have been shown to have better generalization capabilities than linear least squares methods.

1) Learning Dynamics: The Neural Network was used to learn system dynamics, the neural network input were chosen to be:

$$\big[X, Y, Z, v_x, v_y, v_z, cos(\phi_r), sin(\phi_r), cos(\phi_p), sin(\phi_p), cos(\phi_y), sin(\phi_y), \omega_x, \omega_y, \omega_z, \vec{u_t}\big] \tag{15}$$

We chose to change raw angles to cos, and sin, as they better represent angles that are close to one another (i.e. $cos(0) = cos(2\pi)$) and they normalize the angles to appropriate values. The inputs to the neural network (during training and testing) were normalized to zero mean and unit variance. This allows for better training, as the neural network will not be forced to weight certain state variables higher than others, making all of the weights in the neural network of similar order. While $\big[x, y, z\big]$ were not all necessary as state variables (as we can approximate the dynamics of a flying object as spatially invariant), we chose to include them in our neural network as many people in the community got better results while using them, and they can be used to include more subtle dynamics in later models. [5], [6]. The output of our neural network were the normalized change in states $x_{t+1} - x_t$ which helps keep bounded outputs and allows us to easily tune the system based on the length of the time step.

2) Neural Network Model: We chose to use a fully connected feedforward neural network with ReLU activation functions to map from normalized input states to the expected change in state. Following [5] we optimized over the following loss function:

$$\underset{\theta}{\operatorname{argmin}} L(\theta) = \frac{1}{|D|} \sum_{x_t, u_t, x_{t+1} \in D} \frac{1}{2} ||(x_{t+1} - x_t - f_\theta(x_t, u_t)||_2^2 \tag{16}$$

Where our model is parameterized by θ, which is just the weight and bias matrices for the neural network. This loss function is the same as mean squared error where the target $y = x_{t+1} - x_t$. We used fully connected units with ReLU activation functions, as these units do not suffer from the vanishing gradient problem present in other sigmoid activation neural networks. Minimization of error was done using the ADAM optimization method [13] which has been shown to have good performance.

3) Hyperparameter tuning: To effectively train the neural network, we partitioned the training data and cross validated the data with during training. We tuned for number of neural network layers, learning rate, size of neural network layers, and training epochs. Our training set was 200 iterations of a random 50 step sequence with gaussian noise added to each training point. The results of these parameter sweeps can be seen in Figures 6 and 7. From these, we can see that more than 3 layers does not provide any benefit to training as we tend to overfit the data to the training data, while less than 2 layers does not provide enough dimensionality to capture the data. This is corroborated by Figure 7, where we see 3-5 layers is the optimum layer number for lowest test loss. For training rate, after 200 epochs of training (which we deemed sufficient to fully train a neural network based on Figure 7), we found that 10^{-3} provided the lowest test loss while still fitting with the minimal amount of data. For the size of the neural network, we found a generally decreasing trend with with number of hidden units, but the model seemed to hit a low point at 100 units. Finally, looking at the test error vs. epoch, 100 epochs of training was sufficient to train the model no matter the number of layers we used. This led to our decision for using 3 hidden layers, 100 units per layer, and training with 100 epochs at 10^{-3} training rate.

C. Gaussian Processes

Gaussian processes form each state variable as a multivariate Gaussian random variable and track correlations between variables. The main strength of this approach is the posterior confidence incorporated into state outputs. As the dynamics are more accurately learned, the system can gain more confidence on it's transitions. This probabilistic model allows the controller to supply $u_t \propto \frac{1}{\sigma}$ where σ is the uncertainty, so it will not dive into unexplored dynamics regions. GP scale poorly with higher number of states, so for complex state systems can be unfeasible.

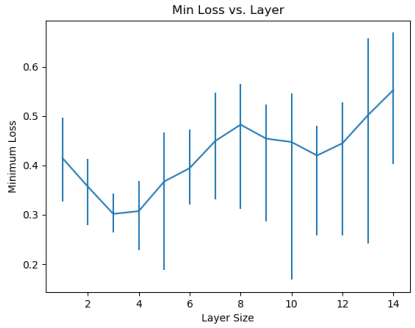
D. Advanced Methods

Especially for tasks looking to accomplish goals, training a loss function on a objective function rather than dynamical accuracy can be useful. Imagine using LQR to control a model to a specific goal with a cost function derived as J. One can tune their (A, B) matrices to iteratively minimize the global cost function. This would no longer be looking for the smallest error rate of the model $x_{t+1} = f(x_t, u_t)$, but for minimum cost. This has been shown to work in some cases, and is primarily used for experiments rather than simulation. This section would be a future area of research exploration

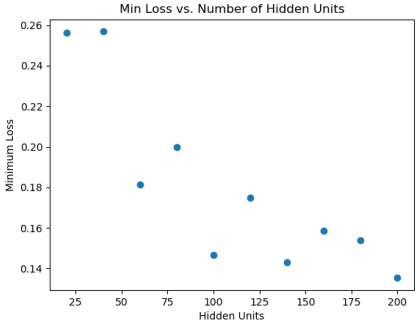
Code for this section is found in the files: models.py

IV. Control

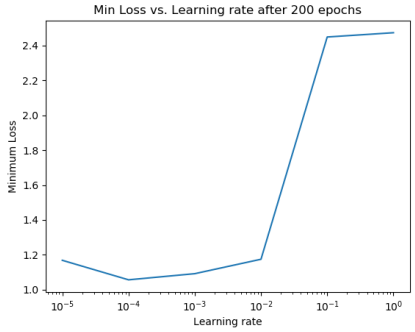
There are multiple directions of control to explore in this project. The standard, while computationally intensive, is



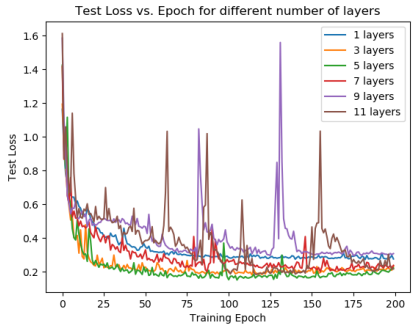
(a)



(b)



(c)



(d)

Fig. 6. Optimizing the number of layers, number of hidden units, learning rate, and number of epochs of training. a) Optimizing Layer size was done using hidden layers of 100 units and a 10^{-3} training rate. b) Hidden unit optimization was done using 3 hidden layers, and a 10^{-3} training rate. c) Learning rate optimization was done using 3 hidden layers, 100 units each. d) Epoch optimization was done using a 10^{-3} learning rate and 100 hidden units per layer. Number of layers is also swept to validate results in a) where 3-5 layers seems optimal to reduce test error.

model based control (MPC) based on the learned dynamics model. This works by looking through a range of random actions in the learned dynamics state, and taking the action that minimizes the desired cost function. Alternatively, we can employ a second round of learning to generate a policy for minimizing a cost function. Good resources for other control methods for learned dynamics on the crazyflie, such as LQR and feasibility reference can be found here [8], [9]. Many methods for control other than MPC require differentiating the learned dynamics, so advanced convex optimization techniques must be used for convergence of NN dynamics.

A. Model Predictive Control

MPC generally is the optimization of a cost or reward function over a finite time-horizon subject to dynamics constraints. A general formulations could be seen in a general form as

$$u^* = \underset{u}{\operatorname{argmin}} \sum_{i=1}^{T} J(x_t, u_t) \qquad (17)$$
$$\text{subject to } x_{t+1} = f(x_t, u_t)$$

This formulation can be expanded as a quadratic program for linear, and linearizable systems to be solved as other convex problems. Using NNs to model the dynamics, the optimal solution is not feasible with any global optimality guarantees. To alleviate this lack of optimally, we utilize the power of computation to simulate many actions and take the locally optimal solution. This random, 'shooter' approach has been shown to be successful in previous works [6], [8].

Our solution is to update the control at each iteration as the best solution to the objective function as a sequence of a single repeated action. We must sample individual random actions and repeat them T times because the dynamics update rate is faster than the control update rate, and we want the action that will perform best until the next control update. The time horizon T is balanced with the number of random actions N to maximize computational performance. The last parameters: J is the objective function and σ_u is the variance of random actions to take around equilibrium, which can be seen as an *exploration* parameter.

Algorithm 1 MPC(x_t, N, T, σ_u, J)

for $i = 1$ to N **do**
  $u[i] = randController(\sigma_u)$
end for
Tile u to form array U of depth T
for $i = 1$ to N **do**
  $Seqs[i] = simSeq(x_t, U[i, :])$
  $ObjVal[i] = \sum_{l=1}^{T} J(Seqs[i], U[i, :])$
end for
$actionIdx = \text{argmin}_i\, ObjVal$
return $u[actionIdx]$

B. Learned Policies

With model-based control, it is possible to separated learn a control policy π after training dynamics. This is done by

training some model to minimize and objective function with inputs being the current state and output being a desired next input.

C. Objective Functions

Our implementation has the ability to design arbitrary objective functions J for the MPC to optimize. As the MPC evaluates the objective function on each individual state vector of the trajectories, trajectory optimization is not currently implemented. To maximize certain variables and minimize others within the same 'max' or 'min' objective function, the function passed into the objective class object should return the sum of $-x_i$ or $\frac{1}{x_i}$ to actively minimize individual state values.

As an initial exploration, our cost function for control was centered around hovering an ionocraft. Without loss of generality, if the quadrotor is hovering around the origin, a quadratic cost function would be:

$$J(x_t, u_t) = \|p_t\|_2^2 \tag{18}$$

This would penalize the distance from the origin in Euclidean distance. We found more stable results in hovering when the cost function included minimizing the angles yaw, pitch, and roll.

Our code allows general creation of objective functions as any mapping $J : \mathbb{R}^{d+l} \mapsto \mathbb{R}^n$ being a vector of the length of the states and the inputs to a scalar value, which can be minimized or maximized. Future work will expand object functions as combinations of maximization and minimization, or max and then min as in many game scenarios. We explain in the following section that more advanced objective functions did not produce results as we hoped.

Code for this section is found in the files: controllers.py

V. Results

1) Ionocraft: The primary results we achieved is an ionocraft learning to hover on the full stack of implemented code. This goes through the dynamics file, to generating sample trajectories, to fitting a model, to generating MPC, and finally to simulate the new trajectory. Bridging this result from simulation to experiment would be a substantial research contribution to the field of micro-robotics and model-based reinforcement learning on novel robotics. As in most work with new implementations of machine learning, substantial parameter tuning is needed to optimize our learning process. A good use of time would be to examine the individual trajectories of the training data to see if we can identify trends and potential eliminate redundancy and or outliers corrupting the model. A visualization of the environment used to visualize animations of learned flight is found in Fig. 7.

The best parameters for learning hovering flight of the ionocraft from modeling the 12-state dynamics function are found below in the table below.

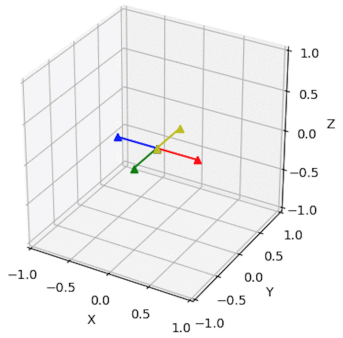


Fig. 7. Screen-shot of an animation of the quadrotor hovering in simulation found here. It was trained on random trajectories near hover and is running model predictive controller.

θ	val
dynamics update step	.0001 (s)
controller update step	.001 (s)
training samples	250
training sequence len	100
training collection time	2.5s
MPC num sim	50
MPC time horizon	5
MPC control variance	25 μN
NN num layers	2
NN hidden nodes	100

TABLE I
IONOCRAFT HOVER LEARNING PARAMETERS.

Our model is less successful when attempting to fly the ionocraft in more complicated trajectories. We attempted to maximize the x and y values of the state while minimizing all other values (z, yaw, pitch, etc.). In terms of axes in Figure 7, this would be the equivalent of the device flying out of the at a fixed height. We attempted a variety of techniques to train this trajectory, including: (a) maximizing the norm of $[x, y]$; (b) minimizing the inverse norm of $[x, y]$ summed with the norm of the rest of the vector; and (c) maximizing the norm of $[x, y]$ summed with the inverse norm of the rest of the state. We also ran all techniques with both the 1-norm and the 2-norm.

All methods showed incredibly high variability. Most experiments *do* in fact successfully maximize x and y. However, we were unable to minimize yaw, pitch, and roll in any simulation, and the device oscillates wildly around a fixed point. Most interestingly, it seems that our model consistently learns a coupling between x and z. As a result, even though we actively minimize z, the device climbs in height due to the x maximization resulting in a z increase. There is clearly significant room for improvement.

A. CrazyFlie

We also attempted to learn the dynamics of a CrazyFlie. The CrazyFlie is a heavier device than the Ionocraft and thus has more variable movements. With the same neural network architecture, we are able to learn the dynamics of the CrayzFlie

well enough to hover with high variance. In particular, over the course of the hover action, the device begins to rotate in terms of yaw & pitch though it does not change location. Due to time limitations, we were not able to attempt to learn more complex trajectories for the CrazyFlie.

Code for this section is found in the files: utils-plot.py

VI. Conclusion

This project lays the ground work for using model-based learning to model dynamics for robots with new dynamics and or less precise state variables. New dynamics could easily be modeled in a simple dynamics file and state variables can be adjusted by passing specific variables into the learning model. The current standard for model-based learning in robotics has been learning tasks for known robots with high-fidelity state feedback and tracking systems such as VICON.

Being able to train novel robotics with lower accuracy state variables paves the way for more robots to learn tasks in non-laboratory environments. This project allows one to explore the minimum quality and quantity of state variables needed to perform basic tasks by passing specific variables into the model with different noise levels. The lower bound on state information will provide insight into the sensors needed for training novel robotics platforms with potentially unmodeled dynamics to learn tasks.

As demonstrated by our results, the simplistic models trained here are functional for basic tasks. However, they do not scale to more complex tasks. We are interested in exploring more complex network architectures as well as different learning strategies to more effectively understand physical dynamics for real devices.

The work with quadrotors will build the foundation for first training hovered flight of the ionocraft [1], [2], and then the framework can be used to investigate other robots. The current test setup for flying the ionocraft uses a 9-axis IMU to return $\ddot{x}$, $\ddot{y}$, $\ddot{z}$, yaw, pitch, roll at 100Hz with low measured noise. Initial work on this dynamics file is underway, and simulation is under way to estimate the best NN structure for training tethered hovering. For this experiment, we will use external computation, so updating the MPC once per 10ms should be an attainable goal. Future balance between update rate and model accuracy will be needed to train an onboard controller. Specific values for number of sequences, sequence length, random action variance, MPC number of iterations, MPC simulation length, and best objective function for hovering should be found before testing on hardware.

Our goal is to have the ionocraft hovering by mid-summer and submit to either the Science Robotics special issue on learning beyond imitation or the RA-L/ICRA dual submission. Thank you for another great course.

Acknowledgements

The code base for this project can be found on GitHub at https://github.com/natolambert/dynamics-learn. We would like to thank Somil Bansal and Dr. Roberto Calandra for the guidance on the project development.

Appendix
Code

Github: https://github.com/natolambert/dynamics-learn

- *controllers.py*: File containing controller class, random controller for exploration, PID controllers for hovering.
- *dynamics.py*: File containing Dynamics class and functions to generate simulated trajectory data
- *dynamics-crazyflie-linearized.py*: Dynamics file for the crazyflie
- *dynamics-ionocraft.py*: Dynamics file for the ionocraft
- *models.py*: Contains the code for the least squares and neural network models. These work stand alone and do processing internally.
- *testing.py*: Script for testing the full stack of the project, with comments.
- *testing-nn*: Script for testing nueral network specficis.
- *utils-data.py*: Utilities for reshaping data.
- *utils-plot.py*: Utilities for generating plots to show results.

The following packages are required for use:

- python3
- math
- matplotlib
- numpy
- pytorch
- sklearn

References

[1] D. Drew, D. S. Contreras, and K. S. Pister, "First thrust from a microfabricated atmospheric ion engine," in *Micro Electro Mechanical Systems (MEMS), 2017 IEEE 30th International Conference on*. IEEE, 2017, pp. 346–349.

[2] D. S. Drew and K. S. Pister, "First takeoff of a flying microrobot with no moving parts," in *Manipulation, Automation and Robotics at Small Scales (MARSS), 2017 International Conference on*. IEEE, 2017, pp. 1–5.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[4] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 2011, pp. 465–472.

[5] A. Nagabandi, G. Yang, T. Asmar, G. Kahn, S. Levine, and R. S. Fearing, "Neural network dynamics models for control of under-actuated legged millirobots," *CoRR*, vol. abs/1711.05253, 2017. [Online]. Available: http://arxiv.org/abs/1711.05253

[6] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning," *ArXiv e-prints*, Aug. 2017.

[7] B. Yang, G. Wang, R. Calandra, D. Contreras, S. Levine, and K. Pister, "Learning flexible and reusable locomotion primitives for a microrobot," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1904–1911, July 2018.

[8] S. Bansal, A. K. Akametalu, F. J. Jiang, F. Laine, and C. J. Tomlin, "Learning quadrotor dynamics using neural network for flight control," in *2016 IEEE 55th Conference on Decision and Control (CDC)*, Dec 2016, pp. 4653–4660.

[9] S. Bansal, R. Calandra, T. Xiao, S. Levine, and C. J. Tomiin, "Goal-driven dynamics learning via bayesian optimization," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, Dec 2017, pp. 5168–5173.

[10] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic mpc for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 1714–1721.

[11] E. Altuğ, J. P. Ostrowski, and C. J. Taylor, "Control of a quadrotor helicopter using dual camera visual feedback," *The International Journal of Robotics Research*, vol. 24, no. 5, pp. 329–341, 2005.
[12] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles," *IEEE Robotics and Automation magazine*, vol. 20, no. 32, 2012.
[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980