1 Table of Notation

Table 1: Table of Model Elements

z_k	Vector of sensed values
$z_{0:k}$	History of sensed values to date
u_k	Vector of control inputs
$u_{0:k}$	History of control inputs to date
$y_{k+\Delta T}$	Performance metrics over time k to $k + \Delta T$
J	Reward function maps performance metrics to a scalar
θ_k	System parameters, intrinsic and extrinsic (unknown)
τ_k	Control objective derived from current task
g_k	Vector of control parameters
C	Control law $u_k = C(g_k, \tau_k, z_{0:k}, u_{0:k})$ (grey-box)
ΔT	Frequency at which OCCAM adapts and computes new gains

2 Details on Simulated Platforms

In this section we provide additional details about each of our simulated evaluation platforms, including two benchmark functions which are commonly used to test global functional optimization algorithms.

2.1 Benchmark Functions

We first validate our method on randomized variations of two common global optimization benchmark functions [1]. The first is the Branin function, which has a 2D input space and 1D output space:

$$f(x) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1 - t)\cos(x_1) + s$$

We treat as system parameters the six constants that parameterize the shape of the Branin function: $\theta = [a, b, c, s, t, r]$.

The second is the Hartmann function, which has a 6D input space and 1D output space:

$$f(x) = -\sum_{i}^{4} \theta_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)$$

Where A and P are constant matrices, and we randomize over the 4-dimensional vector θ as system parameters.

For these benchmark functions, there are no measured quantities $z_{0:k}$ or control actions $u_{0:k}$. We consider the inputs to the benchmark functions to be the "gains" g_k, and the outputs of the functions to be the performance measures y_k. Therefore the data tuples for these functions consist of only the inputs x and scalar "metrics" $y = f(x)$. For these functions, the reward function is simply set to the negative of the scalar function values: $J(y) = -y$. Because there is no history context, the `context-only` baseline in these two examples is simply our method without weight adaptation.

For the benchmark functions, we use F-PACOH [2], which is based on training neural networks with regularization to serve as mean and kernel functions in a GP. F-PACOH is ill-suited to our robotic tests due to the high dimensionality of the full input space to the networks, so we use the LK-GP baseline in our robotic experiments instead.

2.2 2D Race Car

Our first simulated robotic system is a 2-dimensional car racing around a track, modified from the OpenAI Gym "Car Racing" environment [3]. The environment models a powerful rear-wheel-drive car with sliding friction, making control nontrivial while trying to maximize speed on track. The system has three control inputs: $u_k = [u_s,\, u_g,\, u_b]$. The controller C of the car consists of a proportional-plus-derivative (PD) controller that computes steering input u_s to steer the car towards the centerline of the track and a simple control law that accelerates the car by force u_g on straightaways up to a maximum speed, or brakes the car by force u_b for corners above a certain curvature threshold. The sensor measurements of this system are $z_k = [v,\, \omega_k,\, e_{\text{lat}}]$, where v is the forward velocity of the car, ω_k is the angular velocity, and e_{lat} is the lateral distance between the car and the track centerline. The controller C uses z_k and an estimate for track curvature, c, derived from a vector of upcoming track waypoints τ_k to compute u_k as follows:

$$u_s = k_{ps} e_{\text{lat}} + k_{ds} \dot{e}_{\text{lat}}$$
$$u_g = \begin{cases} k_{pg}, & \text{if } v \le v_{\max} \\ 0, & \text{otherwise} \end{cases}$$
$$u_b = \begin{cases} k_{pb}, & \text{if } c \ge c_{\text{thresh}} \\ 0, & \text{otherwise} \end{cases}$$

The tunable parameters of this controller are

$$g = [k_{ps}\ k_{ds}\ k_{pg}\ k_{pb}\ v_{\max}\ c_{\text{thresh}}]$$

The racing car environment has three unknown system parameters $\theta = [m,\, p,\, \mu]$, which are respectively the mass of the car, the car's engine power, and the friction between the tires and track.

For the racing car, the evaluation function computes the vector of performance metrics

$$y_{k:k+\Delta T} = \frac{1}{\Delta T} \left[\frac{1}{1 + \sum_i e_{\text{lat}i}},\, \frac{1}{1 + \sum w_i},\, \sum v_i \right]$$

These are respectively the inverse average lateral tracking error, inverse of total number of timesteps during which a wheel was slipping, and average velocity over a fixed evaluation horizon. We invert tracking error and wheelslip since, in general, they ought to be minimized. In this case, the evaluation horizon is not a fixed ΔT but instead is however long it takes for the car to traverse a fixed distance on track. For online testing of this system, we set the reward function to be a weighted combination of the reward terms: $J(y) = \sum_{j=0}^{3} r_j y[j]$.

2.3 Quadrotor with Model-Based Controller

Our second simulated platform is a quadrotor MAV equipped with a geometric trajectory tracking controller defined on SE(3) [4]. This controller takes in a reference trajectory τ_k defined in the quadrotor's flat output space: position (p_x, p_y, p_z) and yaw. The controller computes a feedforward motor speed command based on τ_k using the quadrotor's nominal mass, inertial tensor, thrust and drag torque coefficients. It then uses measurements from the quadrotor $z_k = [p_x,\, p_y,\, p_z,\, v_x, v_y,\, v_z,\, R]_k$, where R is the rotation matrix representation of attitude, to compute feedback commands to correct tracking errors. The controller is parameterized by PD gains on the 3D position and PD gains on the attitude: $g_k = [k_x, k_v, k_R, k_\Omega]$ (following the convention given by [4]). The quadrotor has five unknown system parameters which are the quadrotor's mass, principal moments of inertia, and thrust coefficient: $\theta = [m, I_{xx}, I_{yy}, I_{zz}, k_\eta]$. The baseline controller is only aware of the nominal parameters, which are centered around those of the Crazyflie platform [5], and not the actual values. Thus, the feedback gains must be used to compensate for this parametric error. For more detailed information about the quadrotor's dynamics and the controller derivation, see [4].

For this system, the four performance measures y are the inverted average positional tracking error, inverted average yaw tracking error, inverted average pitch and roll, and inverted average commanded thrust over the episode. Following the racing car example, we choose the reward function to be a weighted combination of the terms of y: $J(y) = \sum_{j=0}^{4} r_j y[j]$. For this system, we set $\Delta T = 4$ seconds.

For our quadrotor experiments, the commanded trajectories τ consist of 3-dimensional ellipsoidal trajectories of varying radii and frequencies. Because of the simplicity of these trajectories, we do not have to provide information about τ as input to the network for this system. We leave the incorporation of more general and complex trajectories to future work. We use RotorPy [6] and its included SE(3) controller for all quadrotor simulations. For this environment, we also evaluate our framework on a physical quadrotor with the same controller and performance measures.

2.4 Quadrupedal Robot with Learned Locomotion Policy

Our third simulated robotic platform is a quadrupedal robot equipped with a static pretrained locomotion policy π trained using model-free RL [7]. π outputs joint angles such that the torso of the robot follows a velocity twist command $c_k = (\dot{x}_{\text{des}}, \dot{y}_{\text{des}}, \dot{\omega}_{\text{des}})$. The policy takes as high-dimensional input measurements z_k the joint positions and velocities $q_k, \dot{q}_k$, previous joint angle commands a_{k-1}, commands c_k, timing reference variables, and estimated base velocity and ground friction. We treat π as our controller C for this system.

Although π is parameterized by a deep neural network, it is also conditioned on an additional command that allows the user to specify high-level behaviors that the policy should follow:

$$b_k = \left[\theta_1^{\text{cmd}},\, \theta_2^{\text{cmd}},\, \theta_3^{\text{cmd}},\, f^{\text{cmd}},\, h^{\text{cmd}},\, h_f^{\text{cmd}},\, s^{\text{cmd}}\right]$$

The three terms $\left[\theta_1^{\text{cmd}},\, \theta_2^{\text{cmd}},\, \theta_3^{\text{cmd}}\right]$ jointly specify the quadrupedal gait, f^{cmd} is the commanded stepping frequency, h^{cmd} is the commanded body height, h_f^{cmd} is the commanded footswing height, and s^{cmd} is the commanded stance width. Thus, the policy tries to follow the velocity command c_k while satisfying the behavior constraints. In the original work b_k is a quantity to be selected by a human operator, while in this work we treat b_k as the controller parameters to be tuned automatically based on the quadruped's randomized parameters and the task c_t. For details on how the learned policy is trained, see [7].

The randomized system parameters θ_k for the quadruped are added mass payloads to the robot base, motor strengths, and the friction and restitution coefficients of the terrain. Although the π contains an estimator module to regress the ground friction, it does not receive direct observations of any of these parameters.

For use in our method, we input only a reduced-dimension subset of z_k into our prediction model network consisting of the estimated base linear and angular velocities and joint torques applied by the motors.

The four performance measures for the quadruped are the inverted average velocity errors along each axis of the command and inverted total commanded torque over the evaluation horizon. For this system, we set the evaluation horizon $\Delta T = 3$ seconds.

The reward function for the quadruped has the same form as the quadrotor system: $J(y) = \sum_{j=0}^{4} r_j y[j]$. All simulations are done using code and pretrained models from [7] and the Isaac Gym simulator [8].

	History Size	Encoder Layers	Encoded Dim	Network Layers	Nonlinearity	Basis Size	Phase 1 epochs	Phase 2 epochs
Branin	-	-	-	[16,16,16]	ReLU	5	50	45
Hartmann	-	-	-	[32, 32, 32]	ReLU	15	75	45
Racing Car	25	[32, 32]	15	[32, 32, 32]	ReLU	5	40	55
Quadrotor	25	[64, 64]	15	[64, 64, 64]	ReLU	15	50	40
Quadruped	20	[64, 64]	15	[64, 64, 64]	ReLU	15	50	15

Table 2: Architecture and Training Hyperparameters for OCCAM Basis Function Network for all tested systems

	History Buffer Size	Encoder Layers	Encoded Dim	Network Layers	Nonlinearity	Meta Training Epochs	Inner Loop Steps
Branin	-	-	-	[16,16,16]	ReLU	35	10
Hartmann	-	-	-	[32, 32, 32]	ReLU	70	20
Racing Car	25	[32, 32]	15	[32, 32, 32]	ReLU	70	10
Quadrotor	25	[64, 64]	25	[64, 64, 32]	ReLU	25	20
Quadruped	20	[64, 64]	15	[64, 64, 64]	ReLU	35	20

Table 3: Architecture and Training Hyperparameters for Reptile baseline for all tested systems

	Network Layers	Num fitting iters	Weight Decay	Prior Factor	Feature Dim
Branin	[32,32,32]	2500	3e-5	0.06	5
Hartmann	[32, 32, 32]	2500	0.03	0.23	6

Table 4: Training Details for F-PACOH baseline for all tested systems

3 Model Training and Testing Details

The datasets for the robotic systems each consist of $N = 1500$ batches of $N_B = 64$ datapoints each. The hyperparameters of each dataset and network are provided in the supplementary material. Note that our method does not require sampling only optimal or high-performing gains to generate data - only random ones. Thus, the dataset for each system consists of N batches of datapoints: $[(g, \tau, z, u, y)_{0:N_B}]_{0:N}$. Each of these batches is used as a "task" for a single inner loop during the meta-training process.

We find that we are able to use small networks to model each system; the networks are all fully-connected networks that consist of 3 hidden layers with fewer than 64 hidden units, outputting between 5-20 bases, indicating that many of the robotic systems that we are interested in controlling can be effectively modeled with a relatively small number of parameters. The exact network layer sizes and training hyperparameters are given in the supplementary material. All models are implemented and trained in PyTorch [9].

Architectural details and training hyperparameters for OCCAM's basis function network, Reptile, and F-PACOH are presented in Tables 2, 3, and 4 respectively. The F-PACOH training hyperparameters were chosen in accordance with experiments conducted in the original paper.

Training and testing parameter ranges for each system evaluated in this work are shown in Tables 5, 6, 7, 8, and 9. For the reward curves and tables shown in the main submission, test system parameters were sampled exclusively from the set difference of the test parameter range and training parameter range.

	Training		Testing	
Parameter	low	high	low	high
a	0.8	1.2	0.5	1.5
b	0.11	0.13	0.1	0.15
c	1.2	1.8	1	2
r	5.5	6.5	5	7
s	9	11	8	12
t	0.035	0.045	0.03	0.05

Table 5: Parameter ranges for Branin experiments

	Training		Testing	
Parameter	low	high	low	high
θ_1	1.0	1.5	0.5	1.5
θ_2	1.0	1.2	0.6	1.4
θ_3	2.4	3.0	2.0	3.0
θ_4	3.0	3.4	2.8	3.6

Table 6: Parameter ranges for Hartmann experiments

	Training		**Testing**	
Parameter	low	high	low	high
Size	0.01	0.03	0.005	0.04
Engine Power	2.5e4	4.5e4	2e4	5e4
Friction Limit	250	450	200	500

Table 7: Parameter ranges for Racing Car. Note that these quantities are given in internal units used by the simulator, not SI units.

	Training		**Testing**	
Parameter	low	high	low	high
Mass (kg)	0.02	0.09	0.01	0.1
I_{xx} (kg · m^2)	2e-6	9e-4	1e-6	1e-3
I_{yy} (kg · m^2)	2e-6	9e-4	1e-6	1e-3
I_{zz} (kg · m^2)	2e-6	9e-4	1e-6	1e-3
k_η (N/(rad/s)^2	2e-8	8e-7	1e-8	1e-6

Table 8: Parameter ranges for Quadrotor

	Training		**Testing**	
Parameter	low	high	low	high
Added Payload (kg)	-0.8	2.5	-1.0	4.0
Motor Strength Factor	0.9	1.0	0.8	1.1
Friction Coefficient	0.25	1.75	0.2	2.0
Restitution Coefficient	0.1	0.3	0.05	5.0

Table 9: Parameter ranges for Quadruped

4 Benchmark Function Results

Table 10: Average Final Obtained Value on Benchmark Systems

	Average Value over Last 5 Trials (↓)	
	Branin	Hartmann ($\times 10^{-4}$)
F-PACOH [2]	2.26 ± 0.70	3.30 ± 4.55
Reptile [10]	3.47 ± 11.79	$\mathbf{1.14 \pm 1.98}$
OCCAM (no-meta)	1.80 ± 0.77	7.42 ± 11.4
OCCAM (context-only)	4.25 ± 3.92	12.83 ± 15.7
OCCAM (Ours)	$\mathbf{1.65 \pm 0.49}$	3.14 ± 5.97

We report the average final reward obtained by all methods on the Branin and Hartmann benchmarks in Table 10, and show minima obtained by each method over time in Figure 1. Notably, our method performs well in both settings. In the Branin setting, OCCAM learns a good initialization and finds the best final minimum. In the Hartmann setting, even though OCCAM learns a relatively poor prior, it is able to adapt and find the same final minimum as F-PACOH.

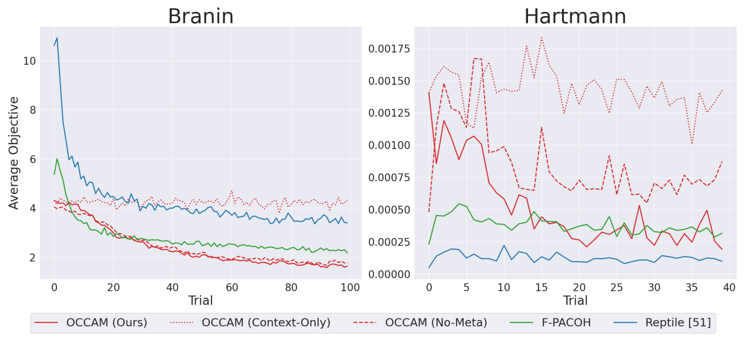


Figure 1: Minima found on each benchmark function (Lower is Better)

5 Raw Performance Metrics

Figures 2, 3, and 4 show the raw performance metrics obtained by each method on each system in the trials in which they did not crash. We note that each method is not directly optimizing for these raw metrics, but instead a weighted combination of their normalized versions, so good or bad performance in an individual metric in these plots does not necessarily translate to high or low reward in the plots reported in the paper. For example, in the Racing Car example, our method obtains a lower average speed than many other methods; however, this makes sense as, in the scalarized objective the model was optimizing for, the speed metric was weighted much lower than the tracking error metric. Also to faithfully report the raw metrics without the crashes skewing the averages, we filter out the runs that crashed. For example, in the quadrotor example, although Reptile performs well when it selects gains that don't result in crashes, its higher crash rate brings down its overall average reward.

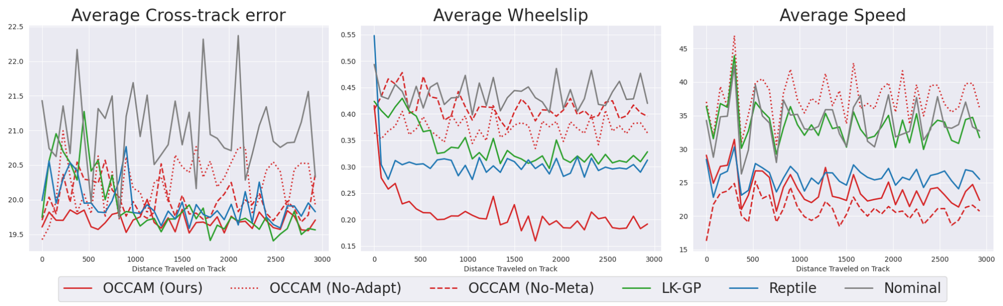


Figure 2: Raw performance metrics obtained by each method on our out-of-distribution racing car test set in successful runs.

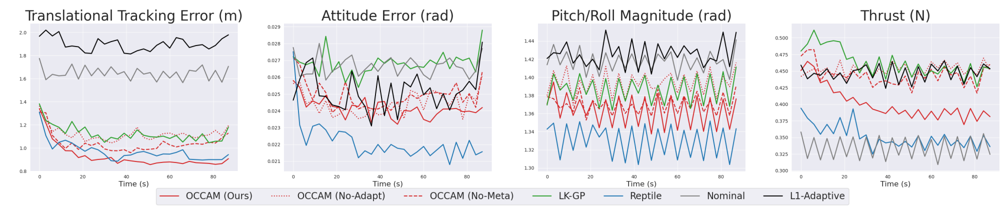


Figure 3: Raw Performance metrics obtained by each method on our out-of-distribution quadrotor test set in successful runs.

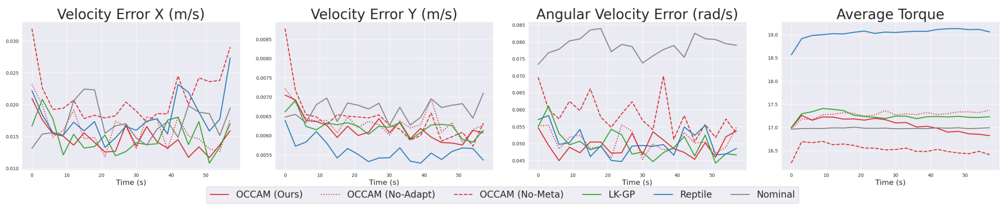


Figure 4: Raw Performance metrics obtained by each method on our out-of-distribution quadruped car test set in successful runs.

6 Additional Simulation Experiments

6.1 In-Distribution Experiments

Table 11: Average Final Reward and Crash Rate on In-Distribution Robotic Systems

	Race Car		Quadrotor		Quadruped	
Method	Avg Final Rwd (↑)	Crash % (↓)	Avg Final Rwd (↑)	Crash % (↓)	Avg Final Rwd (↑)	Crash % (↓)
Nominal	0.50 ± 0	0	1.15 ± 0.13	47.9	0.66 ± 0.9	14.7
LK-GP	0.49 ± 0.08	0	1.79 ± 0.38	37.7	0.74 ± 0.08	8.3
Reptile	0.42 ± 0.13	2.7	1.19 ± 0.37	33.8	0.72 ± 0.1	9.4
$\mathcal{L}1$-Adaptive	-	-	1.37 ± 0.55	57.5	-	-
OCCAM (context-only)	0.47 ± 0.06	2	1.94 ± 0.26	32.5	0.76 ± 0.07	5.6
OCCAM (Ours)	0.44 ± 0.19	4	1.82 ± 0.40	37.5	0.74 ± 0.09	8.7

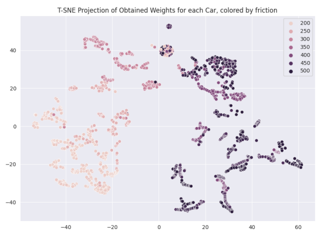
We also run our method and each baseline on test sets randomly sampled from the training distributions for each of the robotic systems (see Tables 7, 8, and 9). The average final obtained reward and crash rates are reported in Table 11. The performances of each method naturally improve in this setting as the sampled system parameters lie closer to the nominal parameters, but in particular the `context-only` baseline, which only uses the fixed context encoder for sysid, and the LK-GP baseline both obtain amongst the highest rewards and perform similarly to OCCAM, showing, within the training distribution, these approaches perform well.

Also notable in this setting is that the $\mathcal{L}1$-Adaptive controller obtains higher reward than the Nominal controller, demonstrating that the adaptive control does indeed improve performance when the deviation from the nominal dynamics is smaller. However, when the parametric error grows larger in the out-of-distribution experiments in the main paper, the adaptive controller becomes unstable and reduces performance.

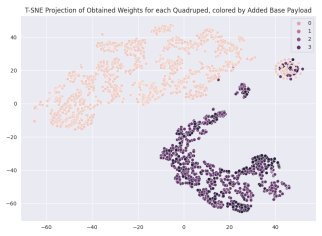
6.2 OCCAM Makes Interpretable Adaptations to the Gains

To elucidate that our method finds semantically meaningful gains, we run an additional experiment in the racing car environment where we sweep only friction coefficients across 3 different tracks and plot the average final gains chosen by OCCAM in Figure 5. As friction increases, OCCAM selects gains that cause the car to accelerate more aggressively and drive faster, while in the low friction regime, the gains tend towards slower driving (higher brake gain, lower

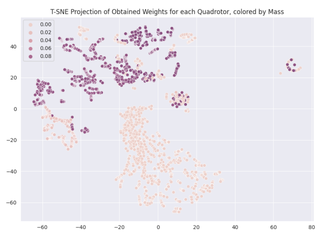
Figure 6



(a) Basis weights computed in the Racing Car Environment, projected into two dimensions and colored by the friction parameter. The weights form distinct clusters separated by different friction coefficients.



(b) Basis weights computed in the Quadruped Environment, projected into two dimensions and colored by the friction parameter. The weights form distinct clusters separated by different added base payloads.



(c) Basis weights computed in the Quadrotor Environment, projected into two dimensions and colored by the friction parameter. The weights form distinct clusters separated by mass parameters

speed in corners). Our method logically chooses a more aggressive driving profile as available traction increases, showing physically meaningful adaptation to changes in system parameters.

6.3 Is there structure to the learned weight space?

We also include preliminary experiments demonstrating that the space of weights that OCCAM adapts in has meaningful structure. For each test set in the paper, we use t-SNE to project the weights computed by OCCAM's regression procedure into two dimensions and plot the projected weights in Figures 6a, 6c, and 6b. Note that like the weight adaptation procedure, the t-SNE embedding procedure has no knowledge of the underlying system parameters. For each system, the values of the weights distinctly cluster according to the underlying system parameters.

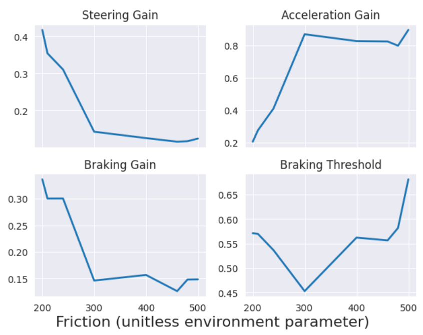
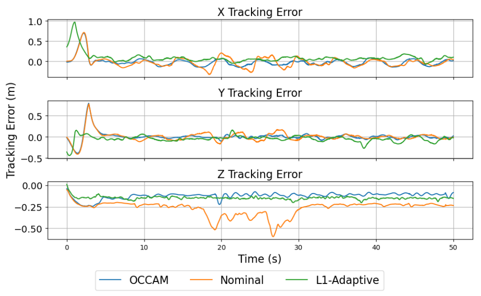


Figure 5: Adapted gains found by our framework for cars with increasing friction coefficients. For cars with higher friction coefficients, our model chooses gains that lead to faster and more aggressive driving. Both the low-end and high-end friction coefficients are out of the training distribution of the model.

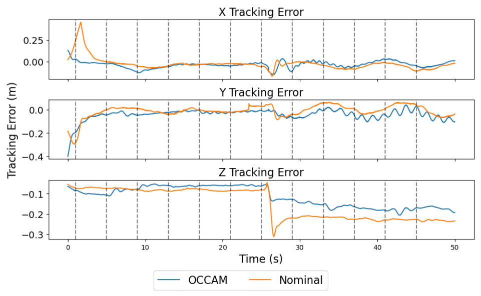
7 Additional Physical Crazyflie Experiments

We ran additional experiments on the physical Crazyflie platform in which we added a 5-gram mass from the beginning of the experiment and in the middle of the experiment. Plots of the tracking error obtained by the controller with OCCAM's optimized gains, the nominal gains, and with the $\mathcal{L}1$-Adaptive control augmentation are shown in Figures 7a and 7b. In both cases, OCCAM finds gains that result in more robust tracking in the Z-axis. We hypothesize that because our predictive model is trained on data gathered from many quadrotors with varied masses, it learns to select gains that better compensate for these variations.

An interesting result are the minor, high frequency oscillations observed in the Z-axis in Figure 7a and in the X- and Y-axes in Figure 7b towards the end of the experiment. These are most likely the result of marginally stable closed-loop attitude dynamics. One possible solution to this is augmenting the performance measures y and measurement vector z with pitch and roll angular velocities, which might encourage the predictive model and optimizer to select gains that do not result in oscillations. Another solution is to add small random force perturbations to the training simulations so that marginally stable controllers achieve worse performance metrics. We leave exploring these additions to future work.



(a) Results with a 5-gram mass added from the start.



(b) Results with a 5-gram mass added at roughly 26s

Figure 7: Positional tracking error results on physical Crazyflie quadrotor following a 3-dimensional ellipsoidal reference trajectory, with added masses.

References

[1] L. C. W. Dixon and G. P. Szego. The global optimization problem: An introduction, 1978.

[2] J. Rothfuss, D. Heyn, jinfan Chen, and A. Krause. Meta-learning reliable priors in the function space. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=H_qljL8t_A.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[4] T. Lee, M. Leok, and N. H. McClamroch. Geometric tracking control of a quadrotor uav on se(3). In *49th IEEE Conference on Decision and Control (CDC)*, pages 5420–5425, 2010. doi:10.1109/CDC.2010.5717652.

[5] Crazyflie 2.1 — Bitcraze. https://www.bitcraze.io/products/crazyflie-2-1/.

[6] S. Folk, J. Paulos, and V. Kumar. Rotorpy: A python-based multirotor simulator with aerodynamics for education and research. *arXiv preprint arXiv:2306.04485*, 2023.

[7] G. B. Margolis and P. Agrawal. Walk these ways: Tuning robot control for generalization with multiplicity of behavior. In *6th Annual Conference on Robot Learning*, 2022. URL https://openreview.net/forum?id=52c5e73SlS2.

[8] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning, Aug. 2021.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms, 2018.