Sampling-based optimal control for quadrotor trajectory tracking using learned dynamics

Mohit Gupta
M.S.E. Robotics
Johns Hopkins University
mgupta27@jh.edu

***Abstract*—In recent years, sampling-based optimal control methods have gained a lot of attention in path and trajectory tracking applications. These control methods work on the principle of generating multiple trajectories in a control step by sampling inputs (also referred to as rollouts) and computing a sub-optimal sequence of control inputs using these rollouts. This is achieved by implementing a finite time-horizon optimization, applying the optimized input for the first time step and repeating the optimization for the next time step. In this project, I aim to study and compare two such methods called Model Predictive Path Integral Control and Cross-Entropy Method for trajectory tracking applications. These methods have the edge over the classical control techniques as they directly use the system model for computing the control inputs and do not require computing their derivates. Another addition to these algorithms is using the learned system's dynamics to estimate rollouts. This makes these algorithms more robust to model uncertainties and environmental disturbances.**

I. Related Work

Since the past few years, Model Predictive Control has been applied to various robotics systems such as self-driving cars [1]–[3] and Quadcopters [4], [5]. It is also widely used in the industry [6]. The first version of MPPI was introduced in [4] with an application for autonomous aggressive driving tasks. The information-theoretic dualities between free energy and relative entropy are explored, and the MPPI control logic was derived. Later, work was done to develop MPPI for non-affine dynamics [2], [3]. The problem of generating time-optimal trajectories through multiple waypoints for quadrotors is explored in [5]. The cross-entropy method [8] was first introduced in the 1990s as a stochastic, derivative-free global optimization technique. These sampling-based methods require a high-computation time due to the intrinsic nature of population-based optimization and, thus, are not very suitable for real-time planning. One effort to make the CEM work in real-time was explored in [9]. It is evident that these planners can be optimized to work in real-time scenarios, which generates sub-optimal trajectories. The work in [7] uses a neural network to learn the system dynamics and concludes that simple feedforward networks are capable of learning and generalizing the system dynamics to good accuracy.

II. Dynamics model

In this section, we present the non-linear dynamic model of the quadcopter to be used and the associated model uncertainties and external disturbances.

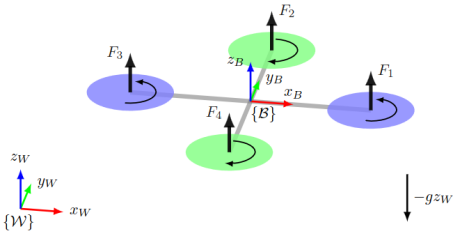

Fig. 1: Quadcopter model along with coordinate system and forces acting on the frame.

A general discrete-time stochastic non-linear system is of the form:

$$x_{t+1} = F(x_t, u_t + \epsilon_t) + w_t \tag{1}$$

where $x \in \mathbb{R}^N$ is the state, and $u \in \mathbb{R}^M$ is the control input. The term $\epsilon \in \mathcal{N}(0, \sum)$ is the disturbance with normal distribution in the control signals generated in the lower-level controllers for real-world robot systems. The term w represents the combined errors due to inaccurate model representations and purely stochastic or unobserved external disturbances, such as wind in the case of quadcopters.

A. Quadcopter model

Let the quadcopter model be as shown in Fig.1, with an inertial reference frame defined as $\mathcal{W}$ and a body-fixed frame defined as $\mathcal{B}$ with origin defined at the center of the quadcopter. The Euler angles to define the quadrotor rotations in $\mathcal{W}$ about the body-fixed axes $x_\mathcal{B}$, $y_\mathcal{B}$, and $z_\mathcal{B}$, are roll ϕ, pitch θ, and yaw ψ respectively.

The rotation from W to B with ZYX transformation is expressed as

$${}^{\mathcal{W}}\mathcal{R}_\mathcal{B} = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\phi s_\theta - c_\phi s_\psi & s_\psi s_\phi + c_\phi c_\psi s_\theta \\ c_\theta s_\psi & c_\phi c_\psi + s_\psi s_\phi s_\theta & c_\phi s_\psi s_\theta - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\phi c_\theta \end{bmatrix} \tag{2}$$

where $s_x = \sin(x)$, $c_x = \cos(x)$ for $\forall x \in \{\phi, \theta, \psi\}$. The transformation matrix from Euler angular velocities, $\dot{\Phi}$, to body frame angular velocity Ω is given by

$$T = \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & \frac{s_\phi}{c_\theta} & \frac{c_\phi}{c_\theta} \end{bmatrix} \tag{3}$$

The dynamics of the quadcopter are given by

$$
\begin{aligned}
\dot{x} &= v \\
m\dot{v} &= -mge_3 + {}^{\mathcal{W}}\mathcal{R}_{\mathcal{B}} F e_3 \\
\dot{\Phi} &= T^{-1}\Omega \\
J\dot{\Omega} &= \Gamma - \Omega \times J\Omega
\end{aligned} \tag{4}
$$

where m is the mass of the quadcopter, J is the inertia matrix expressed in $\mathcal{B}$, $x = [x, y, z]^T$ is the position vector, $v = [v_x, v_y, v_z]^T$ is the velocity vector, $\Phi = [\phi, \theta, \psi]^T$ is the Euler angle vector, $\Omega = [p, q, r]^T$ is the body angular rate vector, $e_3 = [0, 0, 1]^T \in \mathbb{R}^3$, $F \in \mathbb{R}^+$ is the body thrust input generated by the motors, $\Gamma = [\tau_x, \tau_y, \tau_z]^T \in \mathbb{R}^3$ is the torque input to the vehicle in $\mathcal{B}$, and $L \in \mathbb{R}^+$ is the arm length of the quadrotor. the state of the system is defined as $\mathbf{x} = [x, y, z, \phi, \theta, \psi, v_x, v_y, v_z, p, q, r]^T \in \mathbb{R}^{12}$. Each rotor produces a vertical force $F_i = k_F \omega_i^2$ and Moment $M_i = k_M \omega_i^2$, where ω_i is the angular velocity of i^{th} rotor. We map the control inputs F and Γ to the system inputs $u = [\omega_1, \omega_2, \omega_3, \omega_4]$ as

$$
\begin{bmatrix} F \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & k_F L & 0 & -k_F L \\ -k_F L & 0 & k_F L & 0 \\ k_M & -k_M & k_M & -k_M \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{5}
$$

We also define the actuator dynamics as

$$
\dot{u} = k_m(u_{new} - u_{prev}) \tag{6}
$$

III. Learning the Dynamics Model using Neural Network

This section covers the neural network architecture used to mimic the dynamics of the quadrotor. The increase in computational power has enabled us to make predictions in large batches and in a very short time. Neural networks, if trained properly, can very well generalize the dynamic equation using supervised learning techniques. Using the dynamic equations directly with iterative control strategies poses a disadvantage by not taking into consideration the model uncertainties and external disturbances. This can be overcome by replacing them with neural networks, which can be updated online as new data is generated from the quadrotor flights to adapt to the model changes, thus producing better control. We further discuss the data collection process, the NN model and training steps, and the results.

A. Data collection

The input to the neural network is 17-dimensional data, and the output is the 12-state derivatives. Multiple input combinations are sampled from uniform distributions within the operational range defined for the quadrotor. The state and their ranges are specified in Table I.

The sampled variables $(\sigma_1, \sigma_2, \sigma_3, \sigma_4)$ are not used as inputs to the NN directly for training. We use these sampled variables to calculate the following additional inputs to the NN as

Input	Range	Description
(x,y,z)	[-10,10]	Position
(v_x,v_y,v_z)	[-10,10]	Velocity
(ϕ, θ)	[-$\pi/4, \pi/4$]	Roll and Pitch
(ψ)	[-pi,pi]	Yaw
(r,p,y)	[-2,2]	Angular velocity
(σ_1,σ_2,σ_3,σ_4)	[0,900]	Rotor speed

TABLE I: Inputs sampling and ranges

$$
\begin{aligned}
F &= k_F(\sigma_1^2 + \sigma_2^2 + \sigma_3^2 + \sigma_4^2) \\
\tau_x &= Lk_F(-\sigma_2^2 + \sigma_4^2) \\
\tau_y &= Lk_F(-\sigma_1^2 + \sigma_3^2) \\
\tau_z &= k_M(-\sigma_1^2 + \sigma_2^2 - \sigma_3^2 + \sigma_4^2) \\
\gamma &= \sigma_1 - \sigma_2 + \sigma_3 - \sigma_4
\end{aligned} \tag{7}
$$

These inputs are passed through the quadrotor dynamics defined in Section II. The training output for the NN is the state derivatives obtained for each input.

B. NN model and training

I have used a simple, fully connected neural network with 3 hidden layers having 256, 128 and 64 neurons, respectively. Each hidden layer contains a ReLU activation function. The network is trained over multiple cycles of 10000 samples for 5000 epochs. Adam optimizer is used with a custom learning rate scheduler to achieve better accuracy. The model was built using Tensorflow2 and trained on a system with an Intel i7-12700H processor, 32 GB RAM and 8GB NVIDIA GeForce RTX 3070 Ti. The NN model achieved a Mean Squared Error test accuracy of 0.017 and a Mean Absolute Error test accuracy of 0.078.

IV. Trajectory tracking

I aim to simulate and compare different iterative control methods to track a reference trajectory generated for a quadrotor. Model Predictive Path Integral (MPPI) and Cross-Entropy Method (CEM) control methods will be used for the following application, and the results will be compared to see which method produces the minimum errors.

For a discrete-time dynamical system with states $\mathbf{x}_t \in \mathcal{X}$, control inputs $\mathbf{u}_t \in \mathcal{U}$, ϵ error in control input, w as external disturbances, and mapping $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ from current state and input to next state given by

$$
\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \tag{8}
$$

A general optimal control problem consists of finding a control policy $\pi(x)$, a map from the current state to optimal input, $\pi : \mathcal{X} \to \mathcal{U}$ such that a cost $J : \mathcal{X} \times \mathcal{U} \to \mathbb{R}^+$ is minimized and is given by

$$
\begin{aligned}
\pi(\mathbf{x}) = \underset{u}{\text{argmin}} \quad & J(\mathbf{x}, \mathbf{u}) \\
\text{subject to} \quad & \mathbf{x}_0 = \mathbf{x} \\
& \mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \\
& \mathbf{x}_k \in \mathcal{X}, \mathbf{u} \in \mathcal{U}
\end{aligned} \tag{9}
$$

Algorithm 2: MPPI

Given: $\mathbf{F}$: Transition Model;
K: Number of samples;
T: Number of timesteps;
$(\mathbf{u}_0, \mathbf{u}_1, ...\mathbf{u}_{T-1})$: Initial control sequence;
Σ, ϕ, q, λ: Control hyper-parameters;
while *task not completed* **do**
  $\mathbf{x}_0 \leftarrow$ GetStateEstimate();
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $\mathbf{x} \leftarrow \mathbf{x}_0$;
    Sample $\mathcal{E}^k = \{\epsilon_0^k, \epsilon_1^k, \dots \epsilon_{T-1}^k\}$;
    for $t \leftarrow 1$ **to** T **do**
      $\mathbf{x}_t \leftarrow \mathbf{F}(\mathbf{x}_{t-1}, \mathbf{u}_{t-1} + \epsilon_{t-1}^k)$;
      $S(\mathcal{E}^k)$ += $\mathbf{q}(\mathbf{x}_t) + \lambda \mathbf{u}_{t-1}^T \Sigma^{-1} \epsilon_{t-1}^k$;
    $S(\mathcal{E}^k)$ += $\phi(\mathbf{x}_T)$;
  $\beta \leftarrow \min_k[S(\mathcal{E}^k)]$;
  $\eta \leftarrow \sum_{k=0}^{K-1} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $w(\mathcal{E}^k) \leftarrow \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $t \leftarrow 0$ **to** $T-1$ **do**
    $\mathbf{u}_t$ += $\sum_{k=1}^{K} w(\mathcal{E}^k)\epsilon_t^k$;
  SendToActuators($\mathbf{u}_0$);
  for $t \leftarrow 1$ **to** $T-1$ **do**
    $\mathbf{u}_{t-1} \leftarrow \mathbf{u}_t$;
  $\mathbf{u}_{T-1} \leftarrow$ Intialize($\mathbf{u}_{T-1}$);

A. Model Predictive Path Integral

The path integral control method for developing optimal control algorithms based on a stochastic sampling of trajectories. The advantage of using this method is that it does not use any derivatives for the system dynamics or the cost function, thus providing robustness in estimating the system dynamics and constructing the cost function. For a transition model F, number of samples N, number of time steps(horizon) T, and an initial control sequence $(u_0, u_1, ..., u_{T-1})$, the algorithm calculates K rollouts for T time steps for the inputs and evaluates different possible trajectories by running them through the dynamics model F. The cost and weight for each of these rollouts are calculated using the provided cost function. These K weights are used to evaluate the contribution of each rollout to the final optimal input sequence, and only the initial input is applied. The rest of the sequence is used as the initial input sequence for the next iteration, and the process is continued. The algorithm for the same is presented in Alg.2.

B. Cross-Entropy Method

CEM is an effective control strategy that computes a control policy by iterative sampling and evaluating a set of candidate solutions. The algorithm samples the perturbations in the control inputs by sampling from a standard normal gaussian distribution initialized with a pre-defined variance. Multiple rollouts are generated using these perturbations and are evaluated to find a new variance and mean of inputs that minimize the given cost. This process is done iteratively to converge on a solution for the variances and means. These converged parameters are used to calculate the final input sequence for the system.

V. Simulation

The neural network is trained with Tensorflow2 on Google Colab using the local system resources. The simulation for the drone is done using MATLAB. The MATLAB Deep Learning Toolbox provides tools to import and use the pre-trained Tensorflow models. We implement 3 control methods namely: MPPI with actual drone dynamics, MPPI with learned dynamics, and CEM with learned dynamics. The simulations are performed for two tasks, the first is the waypoint navigation, and the second is tracking a circular trajectory. For all the simulations, the plant is integrated at 100 Hz (every 0.01 second), and the controller is run at 20 Hz (every 0.05 second). For all simulations, the starting location for the quadcopter is [0,0,10], and the initial control is [620.6108, 620.6108, 620.6108, 620.6108]. The desired position for the waypoint navigation task is [3,3,10]. The circular trajectory is defined with a radius of 2m and an angular speed of $pi/10$. All the algorithms are run on the same cost function, which penalizes the position tracking, velocity magnitudes, angular rotations, deviation of a control signal from the nominal hovering control (indirectly the control effort), and the change in control signals. A critical cost is imposed if the sampled trajectory goes below z=0 (below the reference ground). The initial code implementation is referred from $https://github.com/yujiyuji0610/MPPI$.

A. MPPI control with actual dynamics

1) Waypoint Navigation: The number of rollouts per step is 300, with a prediction horizon of 25. At each step, the controller takes an average of 0.16 seconds to execute. The simulation time is 10s, and the actual program execution time is 32.12s. The final tracking error is 0.1526 m. The results for the same are shown in Fig. 2.

A second simulation with the number of rollouts per step of 800 and a prediction horizon of 25 is executed. At each step, the controller takes an average of 0.39 seconds to execute. The simulation time is 10s, and the actual program execution time is 81.73s. The final tracking error is 0.1477 m.

2) Circle tracking: The number of rollouts per step is 800, with a prediction horizon of 25. At each step, the controller takes an average of 0.45 seconds to execute. The simulation time is 30s, and the actual program execution time is 262.27s. The results for the same are shown in Fig. 3.

B. MPPI control with learned dynamics

1) Waypoint Navigation: The number of rollouts per step is 800, with a prediction horizon of 25. At each step, the controller takes an average of 0.6 seconds to execute. The simulation time is 10s, and the actual program execution time is 119.68s. The results for the same are shown in Fig. 4.

2) Circle tracking: The number of rollouts per step is 800, with a prediction horizon of 25. At each step, the controller takes an average of 0.66 seconds to execute. The simulation time is 30s, and the actual program execution time is 357.94s. The results for the same are shown in Fig. 5.

Control	Sim WP	Exec WP	Sim C	Exec C	single step
MPPI1	10	81.73	30	262.27	0.39
MPPI2	10	119.68	30	375.94	0.6
CEM	10	131.18	30	362.63	0.56

TABLE II: Time comparison for Waypoint and Circle tracking task. The last column represents the time taken for each control step. MPPI1 is the MPPI control with actual dynamics, MPPI2 is the MPPI control with learned dynamics, and CEM is the CEM control with actual dynamics,

C. CEM control with learned dynamics

1) Waypoint Navigation: The number of rollouts per step is 200, with a prediction horizon of 15. The initial variance and mean for each of the 4 inputs are taken as 30 and 620.6108. At each step, the variances and means are optimized for 4 iterations. The controller takes an average of 0.56 seconds to execute per control step. The simulation time is 10s, and the actual program execution time is 131.18s. The results for the same are shown in Fig. 6.

2) Circle tracking: The number of rollouts per step is 200, with a prediction horizon of 15. The initial variance and mean for each of the 4 inputs are taken as 30 and 620.6108. At each step, the variances and means are optimized for 4 iterations. The controller takes an average of 0.56 seconds to execute. The simulation time is 30s, and the actual program execution time is 362.63s. The results for the same are shown in Fig. 7.

The computation times for all three algorithms are shown in Table II.

VI. Conclusion

In this project, the performance of MPPI and CEM control methods was analyzed using the learned quadrotor dynamics. Making predictions for a single input is computationally expensive, and thus, the code is modified to make predictions in large batches to reduce the computation times. Using these iterative control methods with the actual dynamics model generates good results in simulation with no model uncertainties. However, when testing in the real world, they can produce non-optimal results where the dynamics are affected by uncertainties. This can be overcome by using neural networks to learn these uncertainties and model errors on the fly as it collects more data during the flight and improve tracking with time. Thus using learned models produce good results when used in conjunction with iterative planners.

The MPPI algorithm requires a large number of rollouts to arrive at an optimal sequence of control inputs. Even with the learned dynamics, it is able to match the performance of MPPI with actual dynamics in terms of trajectory tracking. It, however, is computationally expensive due to long NN prediction times. It also requires a longer prediction horizon since it directly uses the costs to assign weight to the rollouts as compared to the CEM method.

The CEM algorithm was tested with numerous values of prediction horizon, rollouts, number of iterations per step, initial variance and mean for inputs. Increasing the prediction horizon and rollouts does not have a significant impact on the convergence or response time of the system but increase the computation time of the algorithm. Increasing the number of iterations allows the algorithm to work with a wide range of initialization of variance and mean but again increases the computation time significantly. The initial variance and mean for inputs directly impact the response time of the algorithm and thus making it robust to noise and disturbances. A low or high variance does not allow the algorithm to explore enough rollouts near the optimal solution, thus requiring a higher number of iterations and rollouts for complete exploration. For example, with the parameters used for the above simulations, changing the initial variance to 10 for each input does not allow the algorithm to converge, even for the waypoint navigation task. The results for a very high initial variance of 100 are shown in Fig. 8 and Fig. 9. The drone is highly responsive and is able to converge to the trajectory but has a high control signal fluctuation (high control effort), thus taking a very aggressive trajectory. Such high gains can result in the quadrotor becoming unstable in some cases with high impulsive disturbances. Thus, an optimal combination of all the parameters is required to manage the computation time while allowing convergence.

From the generated results, we can observe that MPPI converges to the trajectory much faster than CEM but has higher tracking errors as compared to CEM. We also observe that CEM generated low control efforts making the trajectories much smoother.

VII. Future Scope

The primary future goal of this project is to implement the algorithms in a low-level language such as C++ and tune the parameters to make it operate in real time. These experiments can be done in SITL simulators such as RotorS and PX4-Autopilot, which are compatible with ROS. Another interesting yet important direction of this work is to test the performance of these algorithms in environments with disturbances and also with modelling errors. Adding parallel training of the dynamics model without affecting the performance of controller execution should be able to generate much better results in uncertain environments as compared to vanilla algorithms without learned dynamics.

References

[1] Williams, Grady, et al. "Aggressive driving with model predictive path integral control." 2016 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2016.

[2] G. Williams, P. Drews, B. Goldfain, J. M. Rehg and E. A. Theodorou, "Information-Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving," in IEEE Transactions on Robotics, vol. 34, no. 6, pp. 1603-1622, Dec. 2018, doi: 10.1109/TRO.2018.2865891.

[3] G. Williams, P. Drews, B. Goldfain, J. M. Rehg and E. A. Theodorou, "Information-Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving," in IEEE Transactions on Robotics, vol. 34, no. 6, pp. 1603-1622, Dec. 2018, doi: 10.1109/TRO.2018.2865891.

[4] Mohamed, Ihab S., Guillaume Allibert, and Philippe Martinet. "Model predictive path integral control framework for partially observable navigation: A quadrotor case study." 2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV). IEEE, 2020.

[5] Romero, Angel, et al. ”Model predictive contouring control for time-optimal quadrotor flight.” IEEE Transactions on Robotics 38.6 (2022): 3340-3356.
[6] Ruchika, Neha Raghu. ”Model predictive control: History and development.” International Journal of Engineering Trends and Technology (IJETT) 4.6 (2013): 2600-2602.
[7] Bansal, Somil, et al. ”Learning quadrotor dynamics using neural network for flight control.” 2016 IEEE 55th Conference on Decision and Control (CDC). IEEE, 2016.
[8] Rubinstein, Reuven. ”The cross-entropy method for combinatorial and continuous optimization.” Methodology and computing in applied probability 1 (1999): 127-190.
[9] Pinneri, Cristina, et al. ”Sample-efficient cross-entropy method for real-time planning.” Conference on Robot Learning. PMLR, 2021.

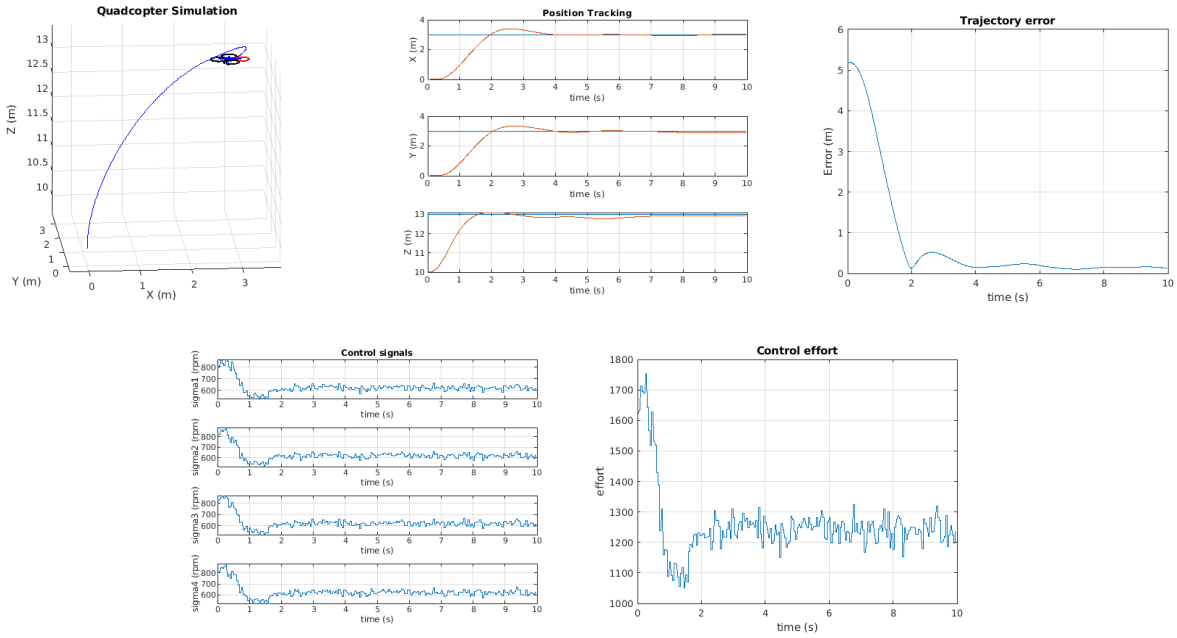

Fig. 2: Results for MPPI running with actual dynamics for waypoint navigation task.

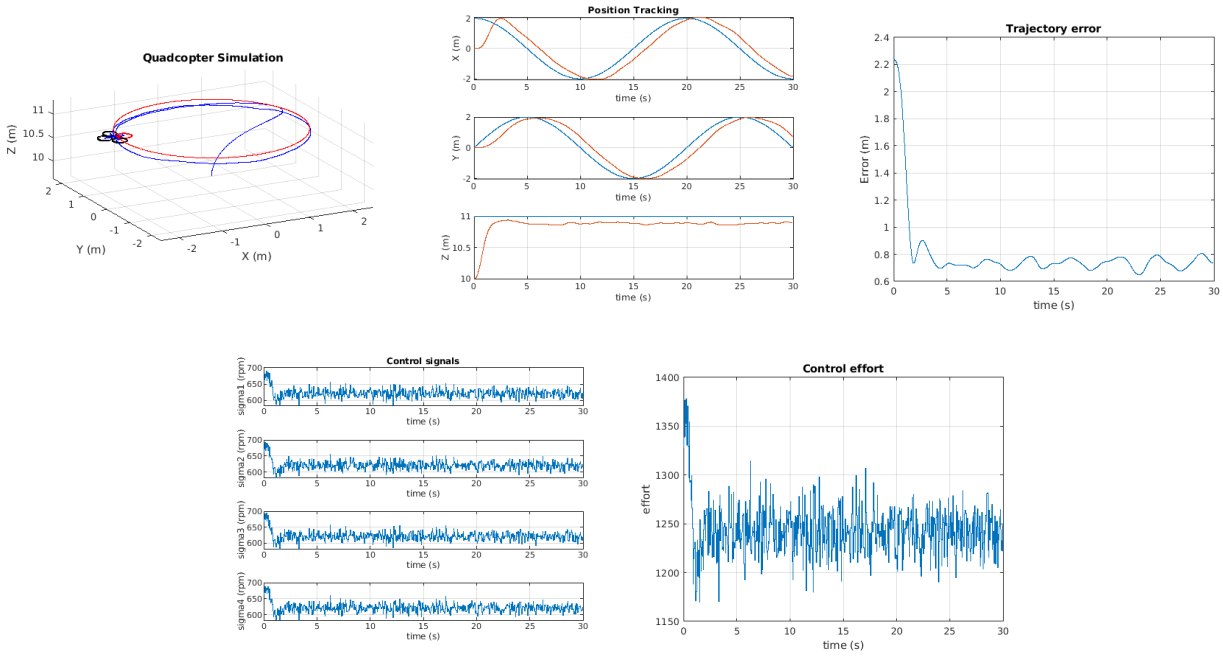

Fig. 3: Results for MPPI running with actual dynamics for circle tracking task.

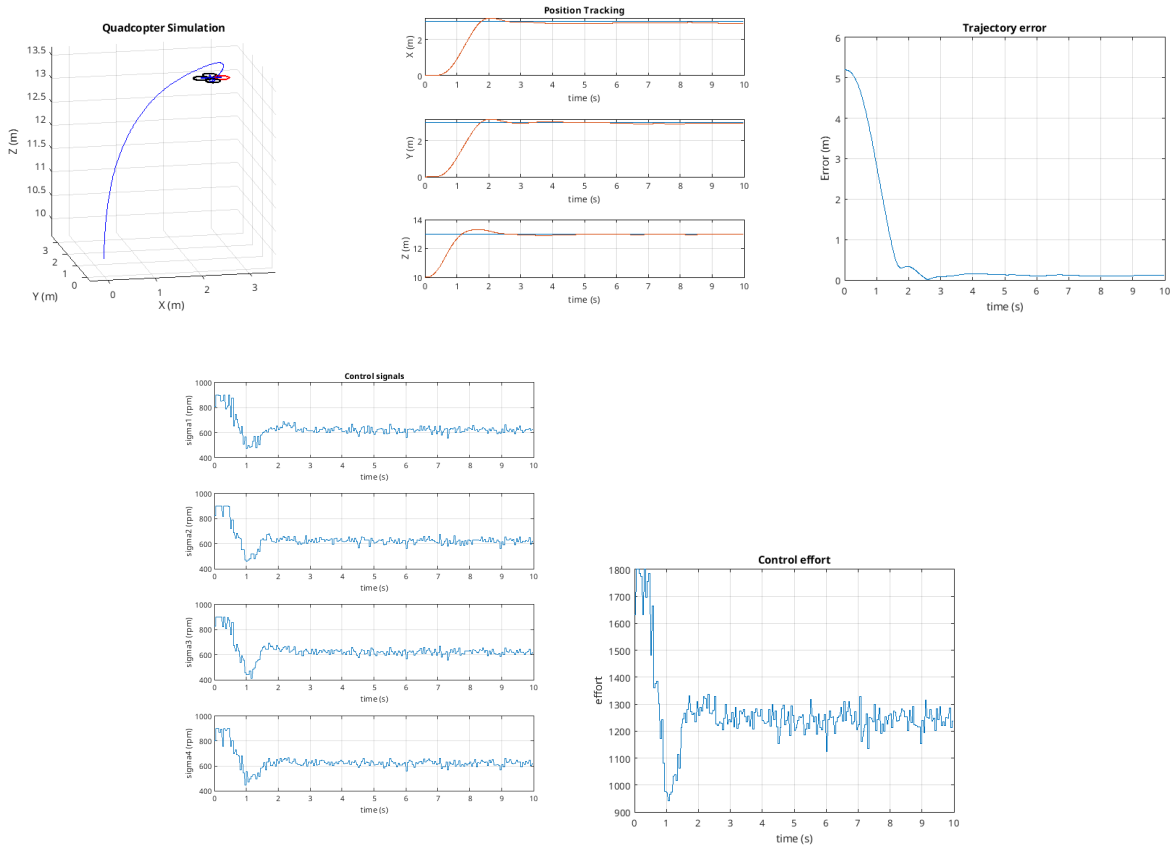

Fig. 4: Results for MPPI running with learned dynamics for waypoint navigation.

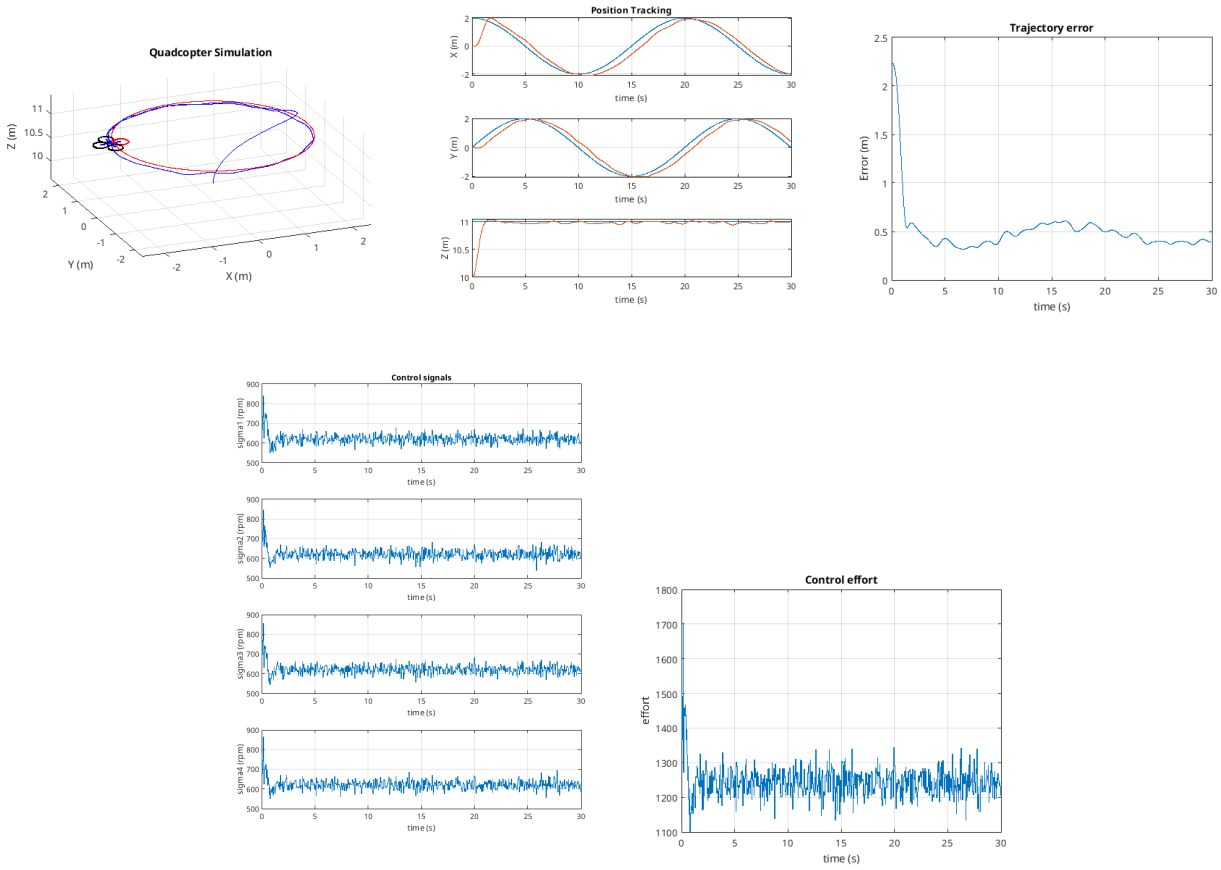

Fig. 5: Results for MPPI running with learned dynamics for circle tracking.

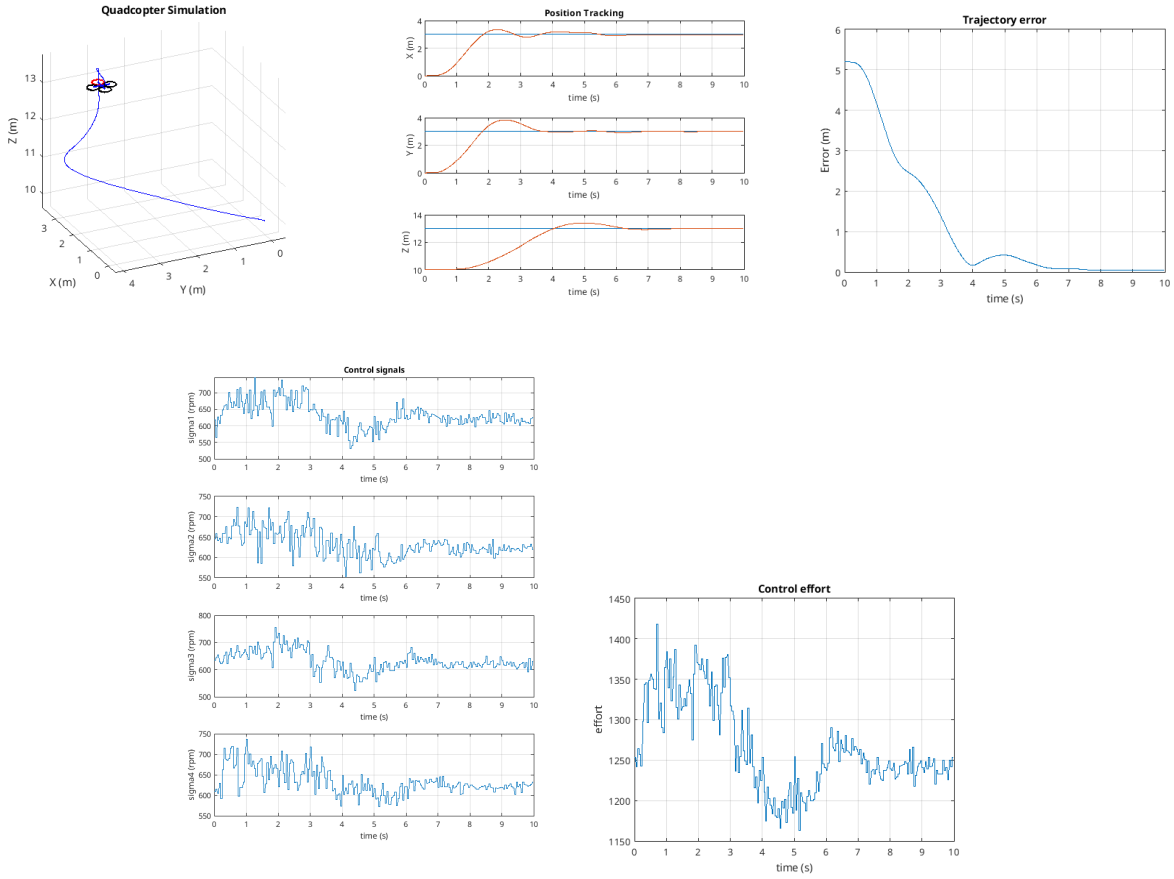

Fig. 6: Results for CEM running with learned dynamics for waypoint navigation.

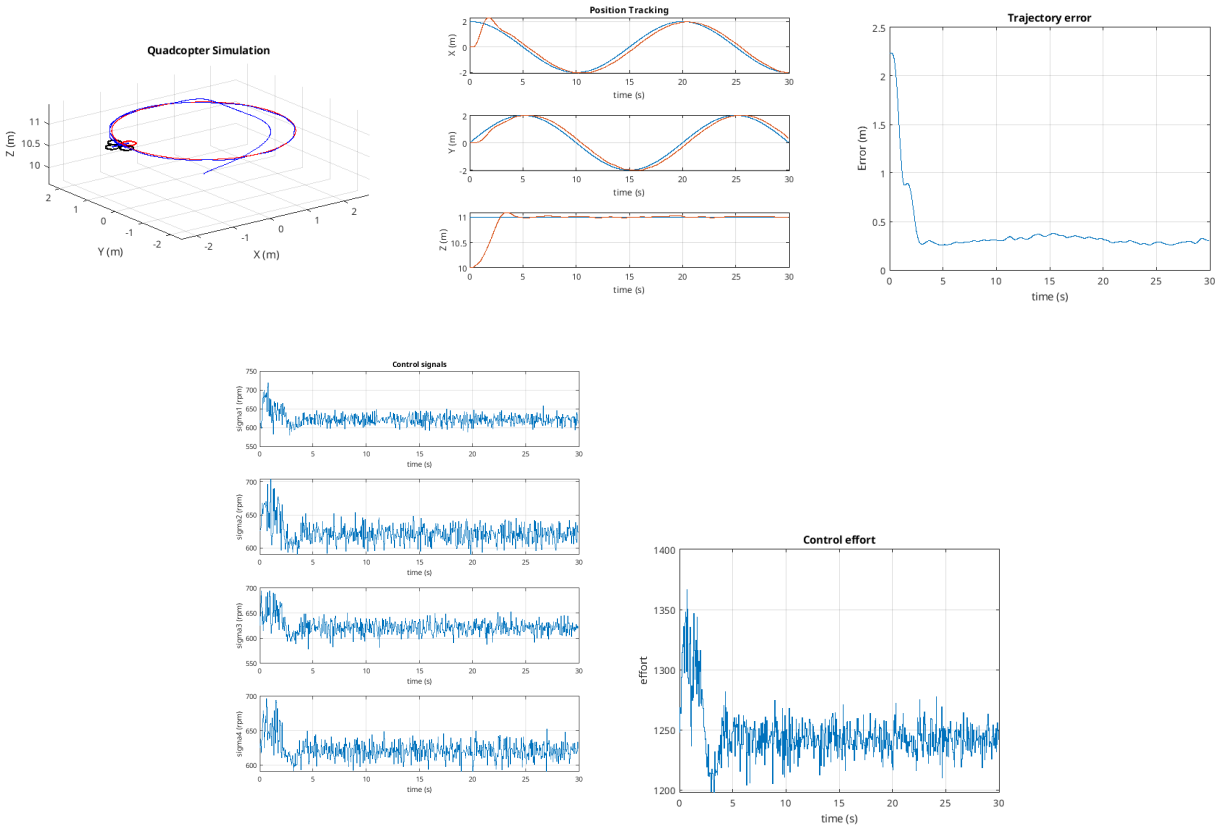

Fig. 7: Results for CEM running with learned dynamics for circle tracking.

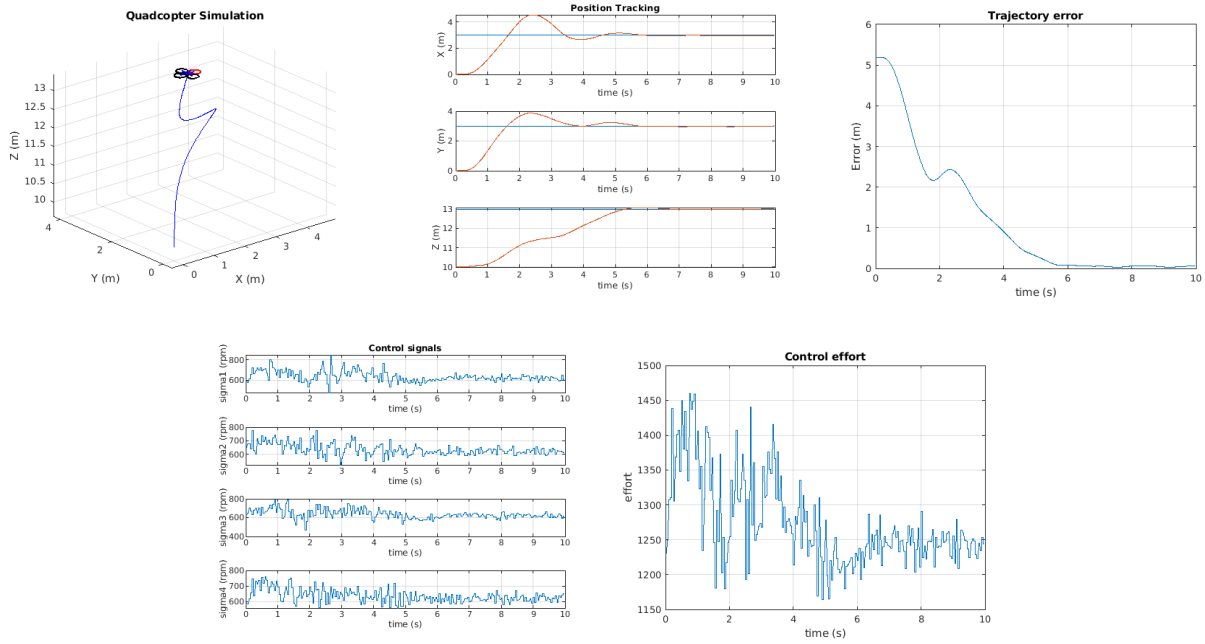

Fig. 8: Results for CEM running with learned dynamics for Waypoint navigation with high variance.

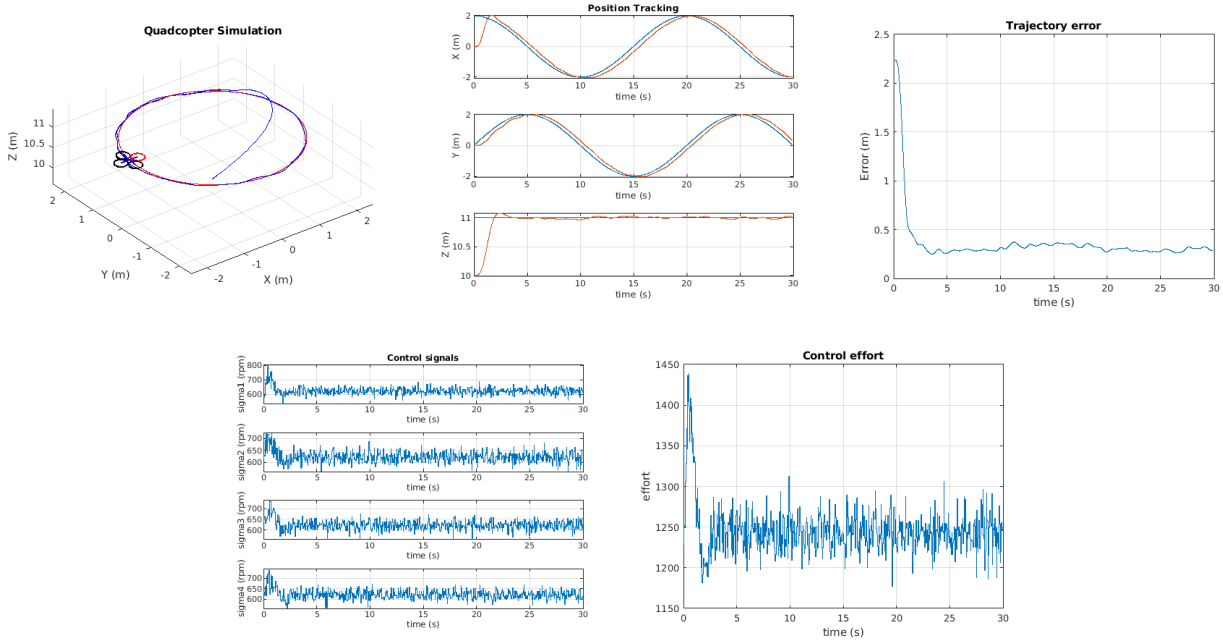

Fig. 9: Results for CEM running with learned dynamics for circle tracking with high variance.