Parameter-free Regression-based Autonomous Control of Off-the-shelf Quadrotor UAVs

Rahul Peddi and Nicola Bezzo

***Abstract*—**

I. Introduction

Unmanned Aerial Vehicles (UAVs) have become widespread for both civilian and military applications in recent years. There are several applications for which UAVs are uniquely suited over other robotic systems, such as surveillance, delivery services, and search and rescue. All of these applications require the UAV to autonomously follow trajectories to different goal or task locations. For example, a package delivery UAV may have to autonomously reach multiple locations at certain times to deliver and receive new packages.

The development of dedicated platforms for such autonomous tasks can be expensive and not necessary since hundreds of off-the-shelf UAVs are available nowadays and for low cost. These platforms are usually well designed, very stable, and ready to fly out of the box but are typically restricted to teleoperation usage.

Generating autonomous flight behavior - subject of this paper - however, can be difficult since every UAV has a different dynamical model, typical not available, and reverse engineering is often not possible. Model-based approaches that include system identification, model extraction, and control design are well know procedures to deal with this issue however they are time demanding and often not precise requiring a lot of tuning and testing ♠[1]. Even when these approaches are successful, among similar vehicles there can be model mismatch due to manufacturing error, different usage, physical changes, or aging which can change parameters thus needing more tuning or sophisticated adaptive control architectures to guarantee safe and reliable control.

On the other hand, data-driven machine learning techniques like neural networks, reinforcement learning, and regression techniques have recently emerged and have demonstrated to be effective to learn from training data. The main drawbacks of such procedures is that there is still not a clear reasoning about how these techniques work and typically large and dense training sets are required to obtain precise results.

In this work we propose a novel approach that leverages both model-based and data-driven theories to generate autonomous flight behavior for an aerial vehicle from few user teleoperation demonstrations. We focus on off-the-shelf quadrotor UAVs which have a well studied dynamical and control architecture as will be described later in the paper and are by far the most common UAVs because of the growing hobby and DIY community. Our framework however can scale and apply to any other aerial vehicle. Fig. **??** shows some examples of such quadrotors focus of this paper, all characterized by having the same shape but different motors, propellers, boom size, and electronics and hence different dynamic and control models.

Different from other supervised learning approaches, in this work we leverage the knowledge about the generalized architecture for the UAV dynamics and their controller and use regression-based techniques to extract a model for closed loop trajectory tracking. The main contribution of this work is a generalized framework that combines both model-based and data-driven theories to generate autonomous control for aerial vehicles. ♠[2] ♠[3]
♠[4] ♠[5]

II. Related Work

♠[6]

III. Problem Formulation

In this work, we are interested in developing autonomous navigation for a UAV by leveraging knowledge about its system dynamics and controller and by leveraging a set of human-piloted demonstrations. Formally, the problem we investigate in this work can be stated as:

♠[7] **Problem 1:** ***Parameter-free Autonomous Control of UAVs***: Consider a pre-trained ready to fly UAV, with known model architecture $\dot{x} = f(x, u, p)$ and control architecture $u = g(y)$ and unknown internal parameters p. Design a

Rahul Peddi and Nicola Bezzo are with the Department of Systems and Information Engineering and the Charles L. Brown Department of Electrical and Computer Engineering, University of Virginia, Charlottesville, VA 22904, USA. Email: {rp3cy, nb6be}@virginia.edu

[1]NB: cite some work on model id

[2]NB: need to reinforce this statement

[3]NB: need to add a figure here that summarizes the approach

[4]NB: we need to stress how different this work is in comparison with the rest of the literature on learning from demonstration. We need to show our contribution clearly

[5]NB: we need to stress that these quadrotors already come with well designed controllers and at the bare minimum we can teleop

[6]NB: overall outline for reference: model of quadrotor is unknown but skeleton architecture is known and the controller is u1, u2, u3 = equations. Because of how the controller is designed, we know that u2 is used to change x and u3 y. A change in u2 is equivalent to a change in the x position. The higher is this change the more motion is generated hence the area under the curve described by the commands sent to the quadrotor is related to the motion of the system and here is where the regression approach is sued. Instead of learnign the parameters that is hard we use regression to learn to fly directly by leveraging the integral and the knowledge about the system.

[7]NB: in latex there is a formal way to define problems by adding begin problem and end problem...please correct

policy to generate the sequence of inputs $\mathbf{u}$ to track a given trajectory $\mathbf{p}_r(t),\ t \in [0, T]$ over a finite user defined horizon T. Specifically the policy should guarantee that the UAV position $\mathbf{p}(t)$ is always within a certain threshold δ_d from the generated trajectory

$$||\mathbf{p}(t) - \mathbf{p}_r(t)|| \leq \delta_d,\ \forall t \in [0, T] \tag{1}$$

where $||\cdot||$ is the euclidean norm.

IV. System Models

♠[8] As hinted in the Introduction, in this work we leverage knowledge about the UAV dynamics and control architectures to extract useful information about the system to guide a regression approach for the generation of autonomous control commands for the UAV. As mentioned above, here we focus on offf-the-shelf tuned and stable quadrotors that typically are ready for teleoperation but not for autonomous flight.

A quadrotor can be modeled using a 12^{th} order state vector: ♠[9]

$$\mathbf{q} = \begin{bmatrix}\mathbf{p}_q^\intercal & \phi & \theta & \psi & v_x & v_y & v_z & \omega_x & \omega_y & \omega_z\end{bmatrix}^\intercal$$

where $\boldsymbol{p}_q = [x\ y\ z]^\intercal$ is the world frame position, v_x, v_y and v_z are the world frame velocities, ϕ, θ and ψ are the roll, pitch and yaw Euler angles and ω_x, ω_y and ω_z are the body frame angular velocities.

The dynamics of the vehicle are then described as follows:

$$\dot{\boldsymbol{p}}_q^\intercal = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}$$

$$\begin{bmatrix}\dot{v}_x \\ \dot{v}_y \\ \dot{v}_z\end{bmatrix} = \begin{bmatrix}0 \\ 0 \\ -g\end{bmatrix} + \frac{1}{m}\begin{bmatrix}\cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi \\ \cos\phi\sin\theta\sin\psi - \cos\psi\sin\phi \\ \cos\theta\cos\phi\end{bmatrix} u_1$$

$$\begin{bmatrix}\dot{\phi} \\ \dot{\theta} \\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta\end{bmatrix}\begin{bmatrix}\omega_x \\ \omega_y \\ \omega_z\end{bmatrix}$$

$$\begin{bmatrix}\dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z\end{bmatrix} = \begin{bmatrix}\frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y\end{bmatrix} + \begin{bmatrix}\frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}}\end{bmatrix}\begin{bmatrix}u_2 \\ u_3 \\ u_4\end{bmatrix}$$

[CITE]

Fig. 1 summarizes the overall control architecture for the quadrotor. The position and low-level controllers, which consist of a series of PID loops [CITE], generate the angle inputs and attitude control that follow the desired trajectory, $\mathbf{p_r}$. The attitude controller and motion dynamics generate the thrust and moment values applied by each rotor. Lastly, the quadrotor body dynamics give the position of the system, which is then fed back into the position controller. This control architecture assumes that the quadrotor is moving at a constant z level with a zero yaw constraint, which is discussed further in

[8]NB: in this section we need to show the dynamical model, and control model and architecture with teleoperation. We need to stress that everything is a gray box with known model but unknown params and that the inputs and outputs are known and used later to obtain a model for autonomous fly. We need to explain the relation between teleop and u1 u2 u3 and u4 and how increasing the joystick increase the inputs and thus the integral measures mobility in the system.

[9]NB: why q in $\mathbf{p}_q$? In the problem formulation there is not q

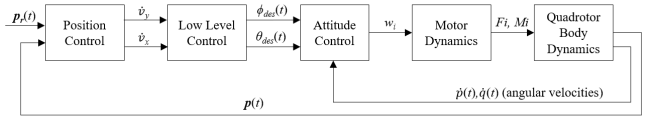


Fig. 1. UAV Control Architecture

♠[10] ♠[11] Both model parameters and control gains shown above, however, are difficult to identify and can vary a lot from UAV to UAV. ♠[12]

A. Learning by Demonstration

♠[13] In human piloted trajectories, a user is controlling u_2, u_3, and u_4 from the system dynamics, which correspond to the desired roll, pitch, and yaw angles. u_1 is the thrust provided to the system, which is not directly sent by the user ♠[14], but it is adjusted when the pilot is modifying roll, pitch, and yaw. In our case, we have a zero-yaw ♠[15] constraint for ease of discussion. In this work, we are interested in learning from commands ♠[16] sent from the human pilot to the UAV, so we assume in our training that we are able to collect all information communicated from the pilot to the UAV. During run-time, we are also able to get the world frame position $\mathbf{p}(t)$ of the UAV using a motion capture system. ♠[17]

Through observation of a single demonstrated trial, we are able to identify important commands sent to the system. An example of this is shown below:

In Fig. 2, we show a sample of a single flight demonstration. The pilot flies the UAV forward over a distance of approximately 3 meters. The command string, in this case, corresponds to pitch, where a command of 0 would imply no pitch adjustment, and 1 corresponds to the UAV's maximum allowable pitch. The minimum allowable pitch command is -1, which would send the UAV in in the negative y-direction. Along with the string of commands sent, the shaded portion of Fig.2, represents the total area under the string of commands, or the integral. Because the pitch is proportional to the linear velocity in the y-direction, we know that the command string is proportional to the

[10]NB: this figure should change to include the teleoperation control instead of the position control

[11]NB: we should include the inner par of the diagram into a gray box and leave outside the joystick commands and the output of the quadrotor.

[12]NB: here we need to mention that the low level control is typically already developed

[13]NB: this section is not good. I don't understand why it's here and not later or what's the poiint. It needs to be rewritten more carefully and more generally. What's the purpose of this section? Why is it outside the approach?

[14]NB: this is not true!!!

[15]NB: this is also not true

[16]NB: this is also not true...we are not interested in learning from user. We are levergaing user teleop because it't the only thing we have and we are interested in generating commands for autonomous flight

[17]NB: this part needs to be rewritten and generalized. We can mention later that in our case we use a MOCAP but here we just need to say that we assume that we can obtain position and velosity of the system during training as well as the commands sent to the vehicles as s result of the user teleop

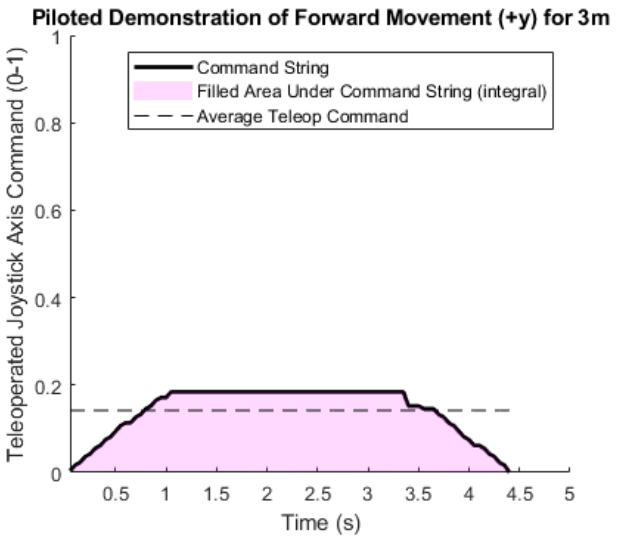


Fig. 2. A sample of a single demonstrated flight

velocity, and its integral, therefore, is proportional to the distance traveled.

Lastly, we are interested in the average command sent to the system, as indicated by the dotted line. The integral itself, while conveying information about the distance traveled, lacks precision in that the same integral can be achieved with commands strings of different lengths and heights. Making use of the average command, we are able to identify the correct height of the teleoperation commands and improve precision in the process of generating autonomous commands, which must be fixed to a certain length (time) and travel a certain distance.

V. Methodology

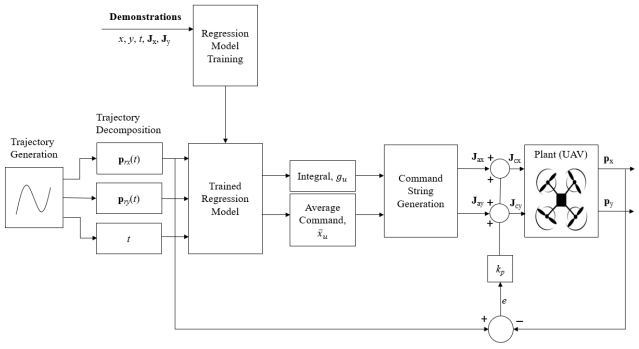


Fig. 3. Block Diagram of Proposed Approach

♠[18]

The proposed approach is outlined in Fig.3. The approach begins with multiple demonstrations that are used to build a regression model that enables identification of an integral, g_u ♠[19], and average teleoperation command, $\bar{x}_u$, given a user-set target distance, d_u, and time, t_u. With the integral and average velocity,a new command string consisting of commands for both lateral and longitudinal direction is created. At this stage, we have obtained the actions ♠[20], $\mathbf{J}_x(t)$ and $\mathbf{J}_y(t)$, that track trajectory $\mathbf{p}_r(t)$. Because these actions are created in open-loop, some error is possible. This error is reduced by closing the loop via a robust control approach to correct and adjust future inputs $\mathbf{J}_x(t+1)$ and $\mathbf{J}_y(t+1)$ to obtain closed loop input commands $\mathbf{J}_{cx}(t+1)$ and $\mathbf{J}_{cy}(t+1)$. ♠[21]

A. Trajectory Generation and Decomposition

In this section, we show how we generate trajectory based on given waypoints ♠[22]. We want our vehicle to track a smooth trajectory as much as possible with linear motion, so this part of the approach is twofold: generating a smooth trajectory, and decomposing this trajectory into linear parts ♠[23], as to be congruous with our training set, which consists of training only on linear motion, which will be discussed further in Section V-B.

From the system dynamics, we know that input commands are responsible for specific motions of the system. Roll enables lateral motion while enables longitudinal motion. Yaw enables rotations around the center of mass and thrust is mostly responsible for changes in height along the global z coordinate. For ease of discussion in this work we do not consider yaw, as we are primarily interested in planar motion, which also implies that the height is kept constant. As a result, any planar motion assuming constant z can be represented as a combination of roll and pitch commands, or commands that send the vehicle in along the global x and y coordinates ♠[24].

For smoothing purposes, we use the minimum jerk trajectory ♠[25]. Jerk is defined as the time derivative of acceleration and is often associated rapidly changing actuator forces. [CITE] The general equation for a minimum jerk trajectory is as follows:

$$\mathbf{p}^*(t) = \arg\min_{p(t)} \int_0^T \mathcal{L}(\dddot{p}, \ddot{p}, \dot{p}, p, t)dt \tag{2}$$

Solving (2), we obtain the form:

$$\mathbf{p}^*(t) = c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0 \tag{3}$$

where $c_0, \ldots, c_5$ are constants specific to the waypoints in the trajectory. ♠[26] Having obtained the smooth trajectory, we are want to decompose said trajectory into lines. This

[18]NB: the figure needs to be changed. jx(t) jy(t) should input into the plant and the output is x(t) y(t) and all goes as input to the Training. The figure is still blurry

[19]NB: what integral?

[20]NB: rewrite. At this stage we have obtain nothing because we have not presented the approach yet. From the integral we synthesis the inputs needed by the system to follow the given trajectory...

[21]NB: introduce next section... the paper reads like independent sections

[22]NB: YOU NEED TO EXPLAIN WHY

[23]NB: you need to explain why

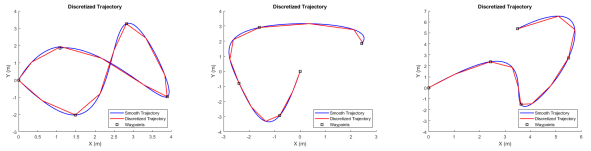
[24]NB: this section needs to go before and needs to be explain better

[25]NB: why? you need to explain better. Besides, we should use snap not jerk

[26]NB: we need to think if these equations are really necessary here. for now leave them but keep the comment

decomposition is done using temporal discretization the trajectory into n parts, each of which will have a time-step of $\delta t = \frac{T}{n}$, where T is the total amount of time allotted for the full trajectory. The discretization is done using linear interpolation within the trajectory over the interval $[\mathbf{p}^*(t_s), \mathbf{p}^*(t_{s+1})]$, where t_s is the time at the start of a segment and t_{s+1} is the end of the segment, defined as: $t_{s+1} = t_x + \delta t$ ♠[27]. ♠[28]

Several ♠[29]examples of decomposed trajectories are shown in Fig.4. All of the trajectories shown in Fig.4 start at the world frame origin $(0, 0)$.



(a) (b) (c)

Fig. 4. Discretized Trajectories

From the discretized trajectory, we have the x and y components of each leg of the trajectory, which require roll and pitch commands, as discussed previously. As a result, ♠[30] our training only needs to be performed on lateral and longitudinal motion. In the case that the UAV is symmetrical, it is possible to only train on movement in one direction, and use the same results for any motion of the system. In this work, we assume we are learning to enable autonomous behaviors on symmetrical UAVs, as expanding to the non-symmetrical case adds little complexity to the approach.

B. Regression Based Training and Evaluation

♠[31] In order to build the appropriate policy for autonomous command generation, we perform offline training on data collected over m human-piloted trials. Because the goal indicated in our problem statement is to be able to track a certain trajectory, the inputs of the training phase are $\{d_i, t_i, \mathbf{J_i}\}$, ♠[32] where $i = 1, \ldots, m$ reflects the number of demonstrations, d_i is the distance travelled in each trial, t_i is the length of the trial, and $\mathbf{J}_i$ is the sequence of user teleoperation commands. The outputs of our training phase are

- The integral of the string of teleoperation commands, $g_i = \int_0^{t_i} \mathbf{J}_i$
- The steady-state, or average, teleoperation command, $\bar{j}_i \in \mathbf{J}_i$
- The position of the system at all times, $\mathbf{p}(t)$

[27]NB: why so complicated this description...please simplify you are just doing a discretization. Also correct gramamr

[28]NB: please write the mathematical expression for the x and y component from the trajectory otherwise this section is incomplete

[29]NB: several? they are just 3...please ad the decomposition otherwise this section is useless

[30]NB: no no no...this is completely wrong. It's not a result of this. It's the opposite. We decompose the trajectory in x and y

[31]NB: what is this section about? Why do we need this section?

[32]NB: the reasoning here doesn't make sense. you write that because we need to track a trajectory, we use d and j. This is not intuitive. Why?

The steady-state command, in this case, provides information about the pilot's average in-flight velocity, and is obtained by taking the mean of all entries in $\mathbf{x_i}$. The integral, meanwhile, describes the area under the string of commands, as discussed in the previous section.

C. Offline Training Data

Our training data in this work consists of multiple demonstrations of a human-pilot flying forward at varying speeds for varying distances. In Fig. 5, we show the raw data received from the demonstrations. In this case, we have training samples where the pilot flies forward and then stops, meaning the initial and final velocities are both zero, which is why there are an extended periods where there is no forward motion at the ends of the trajectories in Fig. 4.

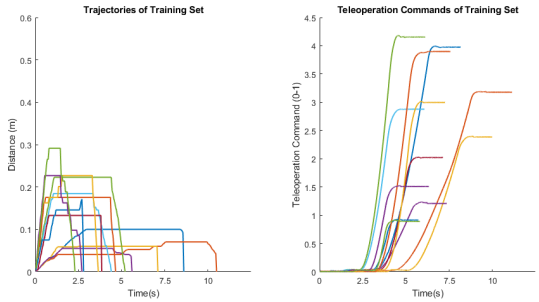


Fig. 5. Training Set, Teleoperation Commands pictured left, Trajectories pictured right

♠[33] This data, however, would only be effective if trajectories involved stopping at each waypoint. In order to further expand the information obtained, considerations were made on the training data to include trajectories for demonstrations where the velocity does not start and end at zero ♠[34].

The velocities and trajectories of the new training data is pictured in Fig. 6.

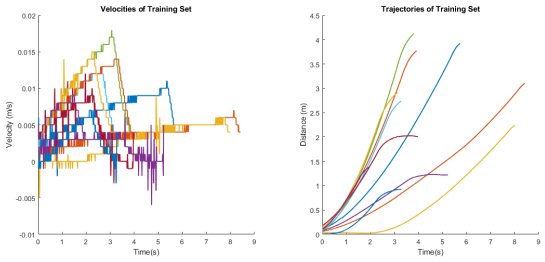


Fig. 6. Training Data with Trajectory Considerations

Having trained on not only start-to-stop trajectories, but also on segments enables more specific command generation based on where the system is in relation to the provided trajectory♠[35]. For example, if the command for the middle

[33]NB: rewrite stating that with these data we can train a model for trajectory starting and finishing with 0 speed. If the initial and final velocities are not 0...

[34]NB: this sentence states nothing. What considerations? This is a technical paper and these sentences kills the paper.

[35]NB: you are not explaing well how you are doing this

the trajectory (i.e., between two points that neither the start nor end) needs to be generated, then the training from the trajectories in Fig. 6 is used ♠[36].

♠[37]

D. Thin-Plate Spline Regression Analysis

The offline training data is applied to a thin-plate spline surface fit to describe the relationship between training inputs and integral and average velocity. The general form of a thin-plate spline equation is

$$f(x,y) = a_1 + a_x x + a_y y + \sum_{i=1}^{m} w_i U(||(x_i,y_i) - (x,y)||) \quad (4)$$

where $a_1, a_x,$ and a_y are scalar coefficients, w_i is a coefficient that corresponds to each specific trial, subject to the following condition:

$$\sum_{i=1}^{m} w_i = \sum_{i=1}^{m} w_i x_i = \sum_{i=1}^{m} w_i y_i = 0 \quad (5)$$

and the function U is of the form

$$U(r) = r^2 \log r \quad (6)$$

Given the corresponding z_i for each (x_i, y_i) pair, we are able to solve the following linear system to obtain the coefficients $w_i, \ldots, w_m$ and a_1, a_x, a_y,

$$\begin{bmatrix} K & P \\ P^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{z} \\ \mathbf{o} \end{bmatrix} \quad (7)$$

where $K_{ij} = U(||(x_i,y_i) - (x_j,y_j)||)$, $P_i* = (1, x_i, y_i)$, $\mathbf{0} \in \mathbb{R}^{3\times3}$ is a matrix of zeros, $\mathbf{o} \in \mathbb{R}^{3\times1}$ is a column vector of zeros, $\mathbf{w} \in \mathbb{R}^{m\times1}$ and $\mathbf{z} \in \mathbb{R}^{m\times1}$ are formed from w_i and z_i, respectively, and $\mathbf{a}$ is the column vector with elements a_1, a_x, a_y.

Given the general framework for performing the thin-plate spline, we develop two separate relationships for our specific application. In our work, the first regression is applied to d (distance) and t (time) as inputs to obtain the integral of the commands $g_i = f(d_i, t_i)$ while the second one has the same inputs to obtain the average teleoperation command $\bar{x}_i = h(d_i, t_i)$. With the functions we have obtained, we are able to find an estimated integral and steady-state command, g_u and $\bar{j}_u$, for any given desired distance and time, d_u and t_u, respectively,

$$g_u = f(d_u, t_u) \quad (8)$$

$$\bar{j}_u = h(d_u, t_u) \quad (9)$$

where d_u and t_u are the user-set desired distance and time, respectively. In Figs. 7 and 8, we show the surfaces that represent the two functions $g_i = f(d_i, t_i)$ and $\bar{j}_i = h(d_i, t_i)$, respectively. The points in these figures represent actual training data points, all of which lay on the surface created with the thin-plate spline.

[36]NB: no no no...this is terrible. Delete and rewrite this section from the beginning.

[37]NB: I'm stopping here for now...the paper needs major changes

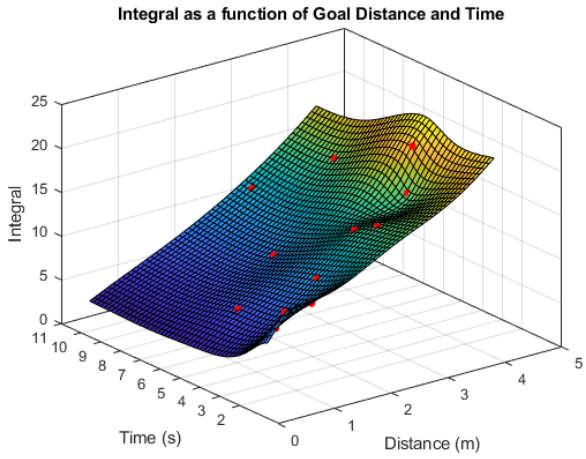


Fig. 7. Integral vs Distance and Time

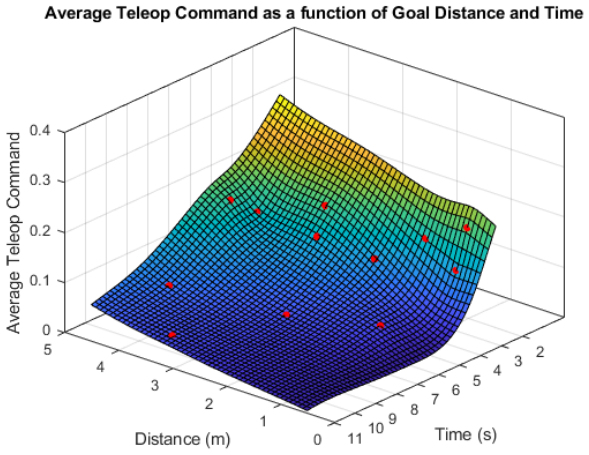


Fig. 8. Average Teleop Command vs Distance and Time

While a thin-plate spline is continuous, and a result can be obtained with any combination of distance and time as an input pair, the accuracy of the results of any pair (d_u, t_u) can suffer as the distance between evaluation points and training points increases [CITE]. In order to quantify this, we leverage the standard error of the estimate, which is a statistic used to measure the accuracy of predictions given a certain type of regression with known values:

$$\sigma_{est} \approx \frac{s}{\sqrt{m}} \quad (10)$$

where s is the sample standard deviation of all of the points in the training set and m is the number of training samples. The standard error, σ_{est}, is then used to determine the t-statistic for different test values. The t-statistic is defined as the ratio of departure of a test point from a known point to its standard error and is defined as

$$t_{\hat{\beta}} = \frac{|\hat{\beta} - \bar{\beta}|}{\sigma_{est}} \quad (11)$$

where $\hat{\beta}$ is the test point and $\bar{\beta}$ is the nearest known point.

In equation (11), if $t_{\hat{\beta}} \geq 1$, that means that the test point $\hat{\beta}$ propagates an error higher than the standard error. As a result, it is desirable to have $t_{\hat{\beta}} < 1$. Using a certain set-point for the t-statistic, the maximum allowable departure from known points is obtained:

$$\sigma_{est} t_{\hat{\beta}} = |\hat{\beta} - \bar{\beta}| \tag{12}$$

The appropriate departure is used to set the bounds for each training data point, for example:

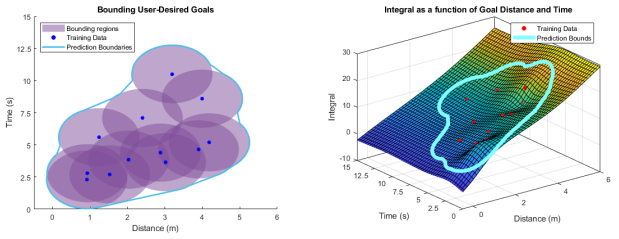
$$\begin{aligned} \hat{\beta}_{\min} &= \bar{\beta} - \sigma_{est} t_{\hat{\beta}} \\ \hat{\beta}_{\max} &= \bar{\beta} + \sigma_{est} t_{\hat{\beta}} \end{aligned} \tag{13}$$

where $\beta_{\min}$ and $\beta_{\max}$ are the lower and upper bounds for known parameter $\bar{\beta}$.

In our case, there are two parameters we are primarily concerned about for prediction; distance and time. As a result, we perform this calculation twice over, treating each of the two parameters as statistically independent, to obtain a generic interval for each point in the training set, denoted $[d_{i\,\min}, d_{i\,\max}]$ and $[t_{i\,\min}, t_{i\,\max}]$, where $i = 1, \ldots, m$. Because we are ultimately interested in using the distance and time values together as input pairs, we then obtain a maximum distance from each of the training data points as follows:

$$\delta_i = \sqrt{(\sigma_{d_{est}})^2 + (\sigma_{t_{est}})^2} \tag{14}$$

Using the maximum distance for each point Δ_i, we are able the generate intervals around each point, where the data is within the standard error of the estimate. Fig. 9(a), contains a pictorial representation of the standard error intervals around each point. In Fig. 9(b), the surface shown in Fig. 7 is modified with a conforming boundary [CITE] created by the intervals.



(a) Prediction Bounds (b) Prediction Bounds on Surface

Fig. 9. Prediction Intervals and Bounds

E. Autonomous Behaviour Generation

In order for the system to autonomously reach a user-set goal, g_u and $\bar{j}_u$ are obtained using equations (8) and (9), and the offline training samples are leveraged to generate a new string of commands. A string of commands is generated for each leg of a trajectory, as described in Section V-A.

From the training set of m samples, we select the trial that has the closest integral, g_i, to the estimated integral g_u. This is done by forming an error vector, $\mathbf{e} \in \mathbb{R}^m$, where each element is defined by

$$e_i = |g_i - g_u|, \; i = \{1, \ldots, m\} \tag{15}$$

The lowest error is then found and is paired with the appropriate pre-trained sample, $\mathbf{J}^*$,

$$\mathbf{J}^* = \mathbf{J}_i \in \mathbf{J} | e_i = \min_e(\mathbf{e}) \tag{16}$$

This optimal pre-trained sample is then adjusted to reflect the user-set time, t_u. This is done by performing vector interpolation to re-size $\mathbf{J}^*$, such that important features in the optimal command vector are not lost. These methods are often used for re-sizing complex images, and have shown effectiveness in minimizing feature loss [CITE]. Bicubic interpolation is our chosen method for re-sizing, as it performs better than nearest-neighbor and bilinear interpolation methods, while only marginally increasing computational complexity [CITE]. The general form of a bicubic interpolation equation is

$$b(x) = \sum_{i=0}^{3} a_i j^i \tag{17}$$

where j is an entry in vector $\mathbf{J}^*$ and a represents the coefficients of the function at each point. Bicubic interpolation takes the weighted sum of the four nearest neighbors of each entry in the command vector in order to identify a function for the intermediate points between each value in $\mathbf{J}^*$. After resizing, we obtain the time adjusted input vector $\mathbf{J}'$.

The next step is to adjust the input vector such that the system reaches the user-set goal d_u. This is done by leveraging the average velocity information, that is, $\bar{j}_u$. Because distance is a function of average velocity and time, the scale time-adjusted vector $\mathbf{J}'$ is scaled such that its mean is equivalent to $\bar{j}_u$. We then obtain

$$\mathbf{J}_a = \mathbf{J}' \left(\frac{\bar{j}_u}{\bar{j}'} \right) \tag{18}$$

In Fig. 10, we show visually the steps of command generation. In Fig. 10(a), we start with the original pre-trained command string that was shown above in Fig. 2. This flight travelled approximately 4.4 seconds for 3 meters. For testing purposes, our use input values are $d_u = 3.5$m and $t_u = 3.75$s. As indicated, a time adjustment is made first; this is shown in Fig. 10(b). It is important to note that the average command, indicated by the dashed line, is still the same, which is a verification of feature retention using bicubic interpolation from (17). If the vector were just stretched without using an interpolation method, we would expect slight variation in the mean, which could damage the integrity of the final step, shown in Fig. 10(c), which is obtained using (18). At this point we have obtained the autonomous command string $\mathbf{J}_a$, which is then sent to the UAV to follow the trajectory $\mathbf{p_r}(t)$.

(a) Original String (b) Time Adjustment (c) Final Commands

Fig. 10. Command Generation Process

F. Online Adaptation of Generated Commands

The commands generated in the previous section are generated and sent to the UAV in open-loop. In order to close the loop, we propose a method to control for any possible error. While executing generated command string, $\mathbf{x_a}(t)$, we constantly monitor for error between the position of the vehicle, $\mathbf{p}(t)$ and the reference position as per the generated trajectory, $\mathbf{p_r}(t)$:

$$\xi(t) = \mathbf{p}(t) - \mathbf{p_r}(t) \tag{19}$$

The error $\xi(t)$ is attributed to the most recent command $\mathbf{J}_a(t)$, and a proportional controller is used to adjust the following command $\mathbf{J}_a(t+1)$ to obtain the closed-loop command $\mathbf{J}_c(t+1)$ as follows

$$\mathbf{J}_c(t+1) = \mathbf{J}_a(t+1) + k_p(t)\xi(t) \tag{20}$$

The proportional gain, $k_p(t)$, is automatically tuned based loosely off the Ziegler-Nichols method [CITE] where the proportional gain is increased until there is stable and consistent oscillation in the output, which in our case, is the position error, $\xi(t)$. In order to set the interval of gain increase each iteration, we use the ratio of the error to desired position:

$$k_p(t+1) = k_p(t) + \frac{\xi(t)}{\mathbf{p_r}(t)} \tag{21}$$

The quantity $\frac{\xi(t)}{\mathbf{p_r}(t)}$ related the current error to the desired position within the trajectory.

VI. Experiments

VII. Conclusions

VIII. Acknowledgement

References

[1] G. O. Young,