Project in Adaptive Control and Real Time Systems

March 30, 2016

Contents

1 Introduction **3**

2 Dynamics **3**
2.1 Non-linear model 4
2.2 Attitude PD control dynamics 5
2.3 Rotor-loop dynamics 5

3 Linearization and state estimation **6**

4 Motion planning **8**

5 MPC control **9**

6 $\mathcal{L}_1$-control **10**

7 General TODOs **11**

8 Appendix **12**

1 Introduction

2 Dynamics

In this project, we consider the non-linear quadcopter equations as derived by Lukkonen et al. [1]. A brief description of the dynamics is given to define terms which will be used in the control scheme. Let

$$\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \quad \boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix}, \tag{1}$$

where $\boldsymbol{\xi}$ [m] denotes the position of the centre of mass in a global cartesian coordinate system, $\boldsymbol{\eta}$ [rad] is the euler-angles in the body coordinate system and ω_i [rad/s] is the angular speed of the rotor i. For future reference, the basis vectors in the cartesian coordinate system are written $\hat{\cdot}$, and the subindexing $\cdot_B$ refers to the vector of matrix defined in the body coordinate system.

The translation from the global- to the body coordinate system is done by the orthogonal rotation matrix

$$\mathbf{R} = \begin{bmatrix} \cos(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\sin(\phi) + \cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \tag{2}$$

such that a vector defined in the body system $\mathbf{v}_B$ can be translated to the global coordinate by the mapping

$$\mathbf{v} = \mathbf{R}^{-1}\mathbf{v}_B = \mathbf{R}^T\mathbf{v}_B. \tag{3}$$

The force generated by the rotor i is assumed to be proportional to the rotor speed squared,

$$f_i = c_2\omega_i^2 + c_1\omega_i + c_0 \approx k_i\omega_i^2 \tag{4}$$

in the positive $\hat{\mathbf{z}}_B$ direction with some constant k_i. In previous work, a non-linear regression of measured force as a function of rotor speed yielded $c_1, c_0 < 10^{-4}$. The approximation is deemed good enough, but the coefficients will be identified for our hardware.

Furthermore, we let the torque around each motor axis be written

$$\tau_{M_i} = b\omega_i^2 + I_M\dot{\omega}_i \tag{5}$$

where b is a drag constant and I_M is the rotor inertia. By virtue of symmetry and under the assumption that $k_i \approx k \in \mathbb{R}^+ \ \ \forall i$, the thrust and torque vectors in the body coordinate system can be written

$$\mathbf{T}_B = T\hat{\mathbf{z}}_B = \begin{bmatrix} 0 \\ 0 \\ k\sum_{i=1}^{4}\omega_i^2 \end{bmatrix}, \qquad \boldsymbol{\tau}_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} kl(-\omega_2^2 + \omega_4^2) \\ kl(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^{4}\tau_{M_i} \end{bmatrix} \tag{6}$$

To make the model more accurate, we introduce air resistance or drag, which increases with $\dot{\boldsymbol{\xi}}$ similarly to viscous friction. This drag matrix is defined as

$$\mathbf{D} = \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \tag{7}$$

where $D_{11} = D_{22} < D_{33}$, and the coefficients remain to be estimated. With the above definitions, the non-linear dynamics of the quadcopter can then be derived from the Newton-Euler equations as

$$\begin{cases} m\ddot{\boldsymbol{\xi}} = m\mathbf{G} + \mathbf{T}_B - \mathbf{D}\dot{\boldsymbol{\xi}} \\ \ddot{\boldsymbol{\eta}} = \mathbf{J}^{-1}(\boldsymbol{\eta})(\boldsymbol{\tau}_B - \mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}), \end{cases} \tag{8}$$

A brief description of $\mathbf{J}$ and $\mathbf{C}$ matrices can be found in **Section 8**, but the interested reader is referred to [1] for a more thorough derivation of the Newton-Lagrange equations. In the work of Lukkonen, this system was simulated in continuous time, and here we will take an alternate approach in order to implement the dynamics as a discrete time ROS node in Python.

2.1 Non-linear model

By defining the states and control signals as

$$\mathbf{x} = \begin{bmatrix} \boldsymbol{\xi} \\ \dot{\boldsymbol{\xi}} \\ \boldsymbol{\eta} \\ \dot{\boldsymbol{\eta}} \end{bmatrix} \in \mathbb{R}^{12\times 1}, \quad \text{and} \quad \mathbf{u} = \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \in \mathbb{R}^{4\times 1} \tag{9}$$

respectively, the full non-linear system can then be written

$$\begin{aligned} \dot{\mathbf{x}}(t) =& \mathbf{A}^c\mathbf{x}(t) + \mathbf{B}^c\mathbf{u}(t) + \mathbf{G}^c \\ \mathbf{y}(t) =& \mathbf{C}^c\mathbf{x}(t) \end{aligned} \tag{10}$$

with

$$\mathbf{A}^c = \begin{bmatrix} \mathbf{0} & \mathbb{I}_{3\times 3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\frac{1}{m}\mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbb{I}_{3\times 3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}}) \end{bmatrix}, \mathbf{B}^c = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \frac{1}{m}\mathbf{R}\hat{\mathbf{z}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{J}(\boldsymbol{\eta})^{-1} \end{bmatrix}, \mathbf{G}^c \begin{bmatrix} \mathbf{0} \\ \mathbf{G} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \mathbf{C}^c = \begin{bmatrix} \mathbf{0}_{7\times 2} & \mathbb{I}_{7\times 7} & \mathbf{0}_{7\times 2} \end{bmatrix} \tag{11}$$

The $\mathbf{C}^c$ matrix was chosen to reflect the available sensory information. The height z is measured by a pressure sensor, the angles $\boldsymbol{\eta}$ are retrieved from a gyroscope aboard the quadcopter and the velocities $\dot{\boldsymbol{\xi}}$ are integrated from readings of the combined sensory feedback from the accelerometer and magnetometer.

In order to simulate the dynamics, the continuous time system (10) was implemented in Simulink (see `quadcopter_model.m`, **Section 8**), and validated by comparison to the results in [1]. The discrete time system was then computed using zero-order hold at a time step h, with

the discrete state space representation

$$x(t_k + h) = \mathbf{A}^d x(t_k) + \mathbf{B}^d \mathbf{u}(t_k) + \mathbf{G}^d \tag{12}$$
$$y(t_k) = \mathbf{C}^d x(t_k) \tag{13}$$
$$\tag{14}$$

where

$$\mathbf{A}^d = e^{\mathbf{A}^c h}, \qquad \mathbf{B}^d = \int_0^h e^{\mathbf{A}^c h} ds B^c, \qquad \mathbf{G}^d = h\mathbf{G}^c, \qquad \mathbf{C}^d = \mathbf{C}^c. \tag{15}$$

As the final implementation of the control system was done in ROS based in Python and C++, a script was written do simulate the system using only the scipy and numpy modules (see simulate_system.py, **Section 8**). The result (see Figure **??**).

2.2 Attitude PD control dynamics

The system is inherently unstable, as the only truly stable position is at $\dot{\boldsymbol{\xi}} = \boldsymbol{\eta} = \dot{\boldsymbol{\eta}} = 0$. In the derivation of the control schemes, it might be interesting to develop controllers not only for the unstable non-linear systems (10), but also for a system which is stabilised with a attitude PD controller. Such a controller was derived in [1] [2], and can be summarised in the scheme

$$\begin{aligned} T &= (g + K_{D,z}(\dot{z}_{ref} - \dot{z})) + K_{P,z}(z_{ref} - z)) \frac{m}{\cos(\phi)\cos(\theta)} \\ \tau_\phi &= (K_{D,\phi}(\dot{\phi}_{ref} - \dot{\phi})) + K_{P,\phi}(\phi_{ref} - \phi)) I_{xx} \\ \tau_\theta &= (K_{D,\theta}(\dot{\theta}_{ref} - \dot{\theta})) + K_{P,\theta}(\theta_{ref} - \theta)) I_{yy} \\ \tau_\psi &= (K_{D,\psi}(\dot{\psi}_{ref} - \dot{\psi})) + K_{P,\psi}(\psi_{ref} - \psi)) I_{zz} \end{aligned} \tag{16}$$

In using this controller, setting all references to 0 results in a stable hovering system assuming the state estimation is good.

2.3 Rotor-loop dynamics

TODO: Find transfer function from motor current to rotor speed on the form

$$H_{I \to \dot{\omega}}(s) \approx \frac{b_0}{s + a_0} \tag{17}$$

The step response of the system is then

$$\frac{b_0}{s + a_0} \frac{1}{s} = \frac{b_0}{a_0} \Big(\frac{1}{s} - \frac{1}{s + a_0} \Big) \xrightarrow{\mathcal{L}_t} \frac{b_0}{a_0} (e^{-t} - e^{-a_0 t}) \tag{18}$$

where the coefficients can be found with a regression. A simple test would be to measure the response of two unit steps in succession to determine both coefficients. The motor is incredibly responsive, going from $\boldsymbol{\omega} = 0$ to $\boldsymbol{\omega} \approx 2.5 \cdot 10^4$ in < 180 ms. (see https://www.bitcraze.io/2015/02/measuring-propeller-rpm-part-3/). We have access to RPM measurements and can therefore construct a very fast PI or PID loop for each motor to keep $\boldsymbol{\omega}^2$ at a desired value. In this section, we could advance

such a controller and show that it is indeed viable and very responsive. With this result in mind, it could be an idea to investigate methods of control when using $\boldsymbol{\omega}^2$, in which case the derived dynamics are the same except for the $\mathbf{B}^c$ matrix, which the must include the mapping of $\boldsymbol{\omega}^2$ to thrust and torques,

$$\begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \mathbf{M}_\omega \boldsymbol{\omega}^2 \quad \text{where} \quad \mathbf{M}_\omega = \begin{bmatrix} k & k & k & k \\ 0 & -kl & 0 & kl \\ -kl & 0 & kl & 0 \\ -b & b & -b & b \end{bmatrix} \quad \text{and} \quad \mathbf{M}_\omega^{-1} = \begin{bmatrix} \frac{1}{4k} & 0 & -\frac{1}{2kl} & \frac{1}{4b} \\ \frac{1}{4k} & -\frac{1}{2kl} & 0 & -\frac{1}{4b} \\ \frac{1}{4k} & 0 & \frac{1}{2kl} & \frac{1}{4b} \\ \frac{1}{4k} & \frac{1}{2kl} & 0 & -\frac{1}{4b} \end{bmatrix} \tag{19}$$

as derived from equation (6). The updated continuous system, with $\mathbf{u} = \boldsymbol{\omega}^2$ is then

$$\begin{aligned} \dot{\mathbf{x}}(t) =& \mathbf{A}^c \mathbf{x}(t) + \mathbf{B}^c \mathbf{M}_\omega \mathbf{u}(t) + \mathbf{G}^c \\ \mathbf{y}(t) =& \mathbf{C}^c \mathbf{x}(t) \end{aligned} \tag{20}$$

3 Linearization and state estimation

Consider the continuous time system with input signal $\mathbf{u} = \boldsymbol{\omega}^2$, it is clear then that the non-linear part of the state-space matrices only concern the $\boldsymbol{\eta}, \dot{\boldsymbol{\eta}}$-states and the input signal. By defining the linearisation point of the euler angles around the stable point $\boldsymbol{\eta}_p = \dot{\boldsymbol{\eta}}_p = 0$, and the rotor velocities around the rotor speed required to hover $\boldsymbol{\omega}_p = \sqrt{\frac{gm}{4k}}[1,1,1,1]^T$, such that the deviation from the linearisation points become

$$\begin{aligned} \Delta\boldsymbol{\eta} =& \boldsymbol{\eta} - \boldsymbol{\eta}_p \\ \Delta\dot{\boldsymbol{\eta}} =& \dot{\boldsymbol{\eta}} - \dot{\boldsymbol{\eta}}_p \\ \Delta\boldsymbol{\omega} =& \boldsymbol{\omega} - \boldsymbol{\omega}_p \end{aligned} \tag{21}$$

Close to this point, the non-linear component of the $\mathbf{A}^c$-matrix ($\mathbf{J}^{-1}\mathbf{C}\dot{\boldsymbol{\eta}}$) can be linearised as

$$\frac{\partial \mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}}{\partial \boldsymbol{\eta}}\Big|_{\boldsymbol{\eta}_p, \dot{\boldsymbol{\eta}}_p} = \mathbf{0}_{3\times 3}, \quad \text{and} \quad \frac{\partial \mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}}{\partial \dot{\boldsymbol{\eta}}}\Big|_{\boldsymbol{\eta}_p, \dot{\boldsymbol{\eta}}_p} = \mathbf{0}_{3\times 3} \tag{22}$$

implying that the linearized continuous-time system matrix is

$$\mathbf{A}^c_{\Delta\omega} = \begin{bmatrix} \mathbf{0} & \mathbb{I}_{3\times 3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\frac{1}{m}\mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbb{I}_{3\times 3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{23}$$

Using the fact that

$$\mathbf{J}^{-1}(\mathbf{0}) = \begin{bmatrix} \frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}, \quad \mathbf{R}(\mathbf{0})\hat{\mathbf{z}}_B = \hat{\mathbf{z}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{24}$$

we conclude that the linearized continuous-time **B**-matrix can be written

$$\mathbf{B}^c_{\Delta\omega} = \frac{\partial \mathbf{B}^c(\boldsymbol{\eta})\mathbf{M}_\omega \boldsymbol{\omega}^2}{\partial \boldsymbol{\omega}}\Big|_{\boldsymbol{\eta}=\mathbf{0},\boldsymbol{\omega}=\boldsymbol{\omega}_p} = \sqrt{\frac{gm}{k}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ k & k & k & k \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{-kl}{I_{xx}} & 0 & \frac{kl}{I_{xx}} \\ \frac{-kl}{I_{yy}} & 0 & \frac{kl}{I_{yy}} & 0 \\ \frac{-b}{I_{zz}} & \frac{b}{I_{zz}} & \frac{-b}{I_{zz}} & \frac{b}{I_{zz}} \end{bmatrix} \tag{25}$$

It is then clear that using the linearized process model

$$\begin{aligned} \dot{\Delta\mathbf{x}} &= \mathbf{A}^c_{\Delta\omega}\Delta\mathbf{x} + \mathbf{B}^c_{\Delta\omega}\Delta\boldsymbol{\omega} \\ \Delta\mathbf{y} &= \mathbf{C}^c\Delta\mathbf{x} \end{aligned} \tag{26}$$

for state estimation with a time invariant Kalman filter requires the positional states, $\boldsymbol{\xi}$, to be measured in order for the system to be fully observable. This could be done by integrating measurements from the accelerometer twice, or introducing a complementary filter to decrease positional drift in stationarity.

In addition to the regular kalman filter, an extended kalman filter (EKF) was implemented in python using the system representation

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{F}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{v}) \\ \mathbf{y}_k &= \mathbf{H}(\mathbf{x}_k, \mathbf{e}) \end{aligned} \tag{27}$$

The prediction step in the EKF is then

$$\begin{aligned} \hat{\mathbf{x}}^-_k &= \mathbf{F}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_{k-1}, \mathbf{v}) \\ \mathbf{P}^-_k &= \mathbf{J}^F_{k-1}\mathbf{P}_{k-1}\mathbf{J}^F_{k-1} + \mathbf{Q} \end{aligned} \tag{28}$$

based on the system dynamics, and a corrector step,

$$\begin{aligned} \mathbf{K}_k &= \mathbf{P}^-_k(\mathbf{J}^H_k)^T(\mathbf{J}^H_k\mathbf{P}^-_k(\mathbf{J}^H_k)^T + \mathbf{R})^{-1} \\ \hat{\mathbf{x}}_k &= \hat{\mathbf{x}}^-_k + \mathbf{K}_k(\mathbf{y_k} - \mathbf{H}(\mathbf{x}^-_k, \mathbf{e})) \\ \mathbf{P}_k &= (\mathbb{I} - \mathbf{K}_k\mathbf{J}^H_k)\mathbf{P}^-_k \end{aligned} \tag{29}$$

where

$$\mathbf{J}^F_k = \frac{\partial \mathbf{F}(\mathbf{x}, \mathbf{u}_k, \mathbf{v})}{\partial \mathbf{x}}\Big|_{\mathbf{x}_k} \qquad \mathbf{J}^H_k = \frac{\partial \mathbf{H}(\mathbf{x}, \mathbf{e})}{\partial \mathbf{x}}\Big|_{\mathbf{x}_k}. \tag{30}$$

4 Motion planning

In this part of the project, we consider the polynomial generation method advanced by Roy et.al. cite XX. The general method is presented to make the Matlab code coherent, using the same nomenclature as in the code. The idea is to set up a constrained QP problem and solve it using Matlab's `quadprog` to find a minimum snap trajectory. The code will later be rewritten in Python/ROS with CVXGEN, and additional constraints will be enforced to ensure that the trajectory always stay in safe convex regions of space. These regions can conceivably be computed using IRIS, but will be assumed to be known initially.

Let the trajectory be composed of n polynomials $P_1(t), ..., P_n(t)$, where

$$P_k(t) = \sum_{i=0}^{N} p_i t^i, \qquad t \in [0, T_k] \tag{31}$$

with a maximum degree of $\deg(P_k) = N$, and a corresponding coefficient vector $\mathbf{p}_{(k)} = [p_{k,0}, ..., p_{k,N}]$. The problem is then to minimise a cost function for every polynomial spline

$$J(T_k) = \int_0^{T_k} c_0 P_k(t)^2 + c_1 P_k'(t)^2 + ... + c_N P_k^{(N)}(t)^2 dt = \mathbf{p}_{(k)}^T \mathbf{Q}_{(k)} \mathbf{p}_k \tag{32}$$

such that continuity is preserved and boundary conditions are enforced. In cite XX, the hessian matrix corresponding to a polynomial was derived by differentiating the term in the cost function containing the r^{th} derivative of $P_{(k)}(t)$ with regards to the polynomial coefficients p_i, i.e. finding

$$\mathbf{Q}_{r,(k)} = \frac{\partial^2}{\partial p_i \partial p_j} \int_0^{T_k} P_k^{(r)}(t)^2 dt \tag{33}$$

and constructing the matrix

$$\mathbf{Q}_{(k)} = \mathbf{Q}_{(k)} = \sum_{r=0}^{N} c_r \mathbf{Q}_{r,(k)} \in \mathbb{R}^{(N+1)\times(N+1)} \tag{34}$$

The complete constrained QP-formulation, including all n polynomials is then

$$\text{Minimize}\Big(\sum_{k=1}^{n} J(T_k) \Big) \qquad \text{subject to} \qquad \mathbf{A}\mathbf{p} - \mathbf{b} = 0 \tag{35}$$

where

$$\tag{36}$$

$$\sum_{k=1}^{n} J(T_k) = \begin{bmatrix} \mathbf{p}_{(1)} & \cdots & \mathbf{p}_{(n)} \end{bmatrix} \begin{bmatrix} \mathbf{Q}_{(1)} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \mathbf{Q}_{(n)} \end{bmatrix} \begin{bmatrix} \mathbf{p}_{(1)} \\ \vdots \\ \mathbf{p}_{(n)} \end{bmatrix} = \mathbf{p}^T \mathbf{Q} \mathbf{p}. \tag{37}$$

For the k^{th} polynomial, the r^{th} derivative can be written

$$P_k^{(r)}(t) = \sum_{n=r}^{N} \Big(\prod_{m=0}^{r-1} (n-m) \Big) p_{k,n} t^{n-r}. \tag{38}$$

Using this formula, boundary conditions can be enforced for each spline by finding a matrix

$$\mathbf{A}_{(k)}\mathbf{p}_{(k)} = \mathbf{b}_{(k)} \tag{39}$$

for every know derivative at the time $t = 0$ (collected in $\mathbf{A}_0$) and time $t = T_k$ (collected in $\mathbf{A}_T$). With n_c boundary conditions for the k^{th} spline, then

$$\mathbf{A}_{(k)} = \begin{bmatrix} \mathbf{A}_{0,k} \\ \mathbf{A}_{T,k} \end{bmatrix} \in \mathbb{R}^{n_c \times N+1} \quad \text{and} \quad \mathbf{b}_{(k)} = \begin{bmatrix} \mathbf{b}_{0,k} \\ \mathbf{b}_{T,k} \end{bmatrix} \in \mathbb{R}^{n_c \times 1} \tag{40}$$

For the remaining, free boundary endpoints where no fixed derivative is specified, the splines on each side of a boundary point are set equal by enforcing

$$\mathbf{A}_{T,k}\mathbf{p}_k - \mathbf{A}_{0,k+1}\mathbf{p}_{k+1} = 0. \tag{41}$$

A general method of generating the **Q**- and **A**-matrices was implemented in matlab (see `get_Q`, `get_A`), called by the `compute_splines` method which used `quadprog`'s interior point method to find a solution given a set of points, times and a cost vector. This, and the demo script `splines_2_1D_example.m` is located in the /crazy_trajectory directory and is almost in a good state. The problem can be solved using various polynomial degrees and finds connecting splines, but the continuity conditions in free points is still not working properly (see Figure 1). Clearly, the jerk squared is minimised and the the polynomial splines are continuos, but the derivatives are not. We have yet to figure out why.

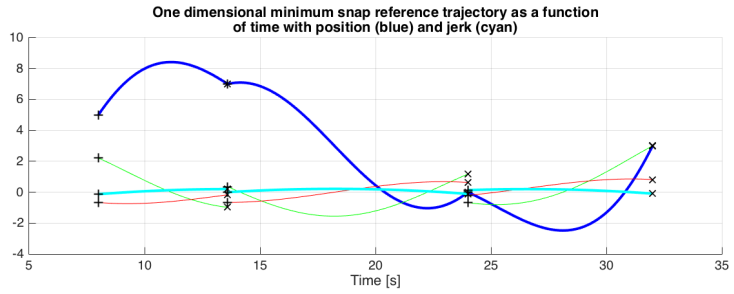

Figure 1: A reference trajectory composed of three splines with a maximum order of $N = 5$ and cost vector $c = [0, 0, 0, 1, 0, 0]$ (minimum snap), enforcing positional endpoint conditions and at unevenly spaced times, with $t = [8, 13.6, 24, 32]$.

5 MPC control

Different implementation paths

1. Include rotor control loops in the system matrix before linearising, thus giving a better model of the system for the MPC controller at the cost of increasing the dimension of $\mathbf{A}^c$ from 12×12 to 21×21 (if using PID control for the rotors). as the rotors are very responsive,

this might not be necessary, and we may get away with simply using $\boldsymbol{\omega}^2$ as input signal and assume that the real value is equal to the reference value at all times. However, this will have to be examined in simulations.

2. Decide using simulations is the stabilizing PD controller should be used in MPC or if the $\boldsymbol{\omega}^2$ control signals should be controlled directly.

In total, we could test all four combinations of with/without omega-loop dynamics and with-/without stabilizing PD.

TODO

1. ~~Create simplified linearised system ss-model for use in MPC~~ (see eg. [3]).
2. ~~Validate by comparison to the results in~~ [3].
3. ~~Set up QP-MPC controller with Simulink MPC-library~~ (see eg. [3]).
4. ~~Validate by comparison to the results in~~ [3].
5. Set up QP-MPC controller with CVXGEN m-code (see eg. [3] [4]).
6. Validate by comparison to the results in Simulink.
7. System identification.
8. Simulate system with proper parameters.
9. Compare the four different implementations based on speed and stability.

6 $\mathcal{L}_1$-control

Here we consider control of the linearized system The $\boldsymbol{\Gamma}$-projection operator for two vectors $\theta, y \in \mathbb{R}^k$ is defined as

$$\text{Proj}_{\boldsymbol{\Gamma}}(\theta, y, f) = \begin{cases} \boldsymbol{\Gamma} y - \boldsymbol{\Gamma} \dfrac{\nabla f(\theta)(\nabla f(\theta))^T}{||\nabla f(\theta)||_2} \boldsymbol{\Gamma} y f(\theta) & \text{if } \ f(\theta) > 0 \ \text{ and } \ y^T \nabla f(\theta) > 0 \\ \boldsymbol{\Gamma} y & \text{otherwise.} \end{cases} \tag{42}$$

where $\boldsymbol{\Gamma} = \mathbb{I}_{k \times k} \Gamma$ for some scalar $\Gamma > 0$ (typically $\Gamma \approx 10^5$) and $f(\theta)$ is a convex function [5]. By solving the Lyapunov equation

$$\mathbf{A}_m \mathbf{X} + \mathbf{X} \mathbf{A}_m^T + \mathbf{Q} = 0, \tag{43}$$

for $\mathbf{P} = \mathbf{P}^T$, with some arbitrary $\mathbf{Q} > 0$, the feedback controller

$$\begin{cases} u(t) = \hat{\theta}^T x(t) + k_g r(t) \\ \dot{\hat{\theta}}(t) = \text{Proj}_{\boldsymbol{\Gamma}}(\hat{\theta}^T(t), x(t) \tilde{x}^T(t) \mathbf{X} b) \end{cases} \tag{44}$$

can be constructed, where $\tilde{x} = \hat{x} - x$ is the state estimation error, k_g is a gain and $r(t)$ is the reference signal. By designing the companion system

$$\begin{cases} \dot{x}(t) = \mathbf{A}_m \hat{x}(t) + b(u(t) - \hat{\theta}^T(t)x(t)) \\ y(t) = c^T \hat{x}(t) \end{cases} \tag{45}$$

it can be shown (by Theorem 2 [6]) that the state estimation error,

$$\lim_{t \to \infty} \tilde{x} = 0. \tag{46}$$

By a corollary of the theorem, choosing

$$k_g = -\frac{1}{c^T \mathbf{A}_m^{-1} b} \Rightarrow \lim_{t \to \infty} y(t) = r \tag{47}$$

if $r \equiv$ constant.

TODO

1. ~~Define general control structure.~~
2. Create Simulink projection operator (see eg. [7])
3. Validate projection operator against benchmark Simulink models (eg. [6]).
4. Define robustness metrics (see eg. [7] [8])
5. Create script for computing the $\mathcal{L}_1$-gain (see eg. [7]).
6. Validate script against benchmark Simulink models (eg. [6]).
7. Simulate control.

7 General TODOs

1. Found that one of the copters were broken - sent to Bitcraze for repairs, expected to be done in early march.
2. Tried installing IRIS in Ubuntu but ran into issues using the PODS "make" command required to get everything up and running. TODO: contact Claes or Anders to get help in finding someone experienced with PODS.

References

[1] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, 2011.

[2] İ. Dikmen, A. Arīsoy, and H. Temeltaş, "Attitude control of a quadrotor," in *Recent Advances in Space Technologies, 2009. RAST'09. 4th International Conference on.* IEEE, 2009, pp. 722–727.

[3] P. Bouffard, "On-board model predictive control of a quadrotor helicopter: Design, implementation, and experiments," Master's thesis, EECS Department, University of California, Berkeley, Dec 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-241.html

[4] J. Mattingley and S. Boyd, "Cvxgen: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[5] E. Lavretsky, T. E. Gibson, and A. M. Annaswamy, "Projection operator in adaptive systems," 2011.

[6] C. Cao and N. Hovakimyan, "Design and analysis of a novel l1 adaptive controller, part i: Control signal and asymptotic stability," in *American Control Conference, 2006.* IEEE, 2006, pp. 3397–3402.

[7] N. Hovakimyan, "L1 tutorial." [Online]. Available: http://naira-hovakimyan.mechse.illinois.edu/l1-adaptive-control-tutorials/

[8] M. Q. Huynh, W. Zhao, and L. Xie, "L 1 adaptive control for quadcopter: Design and implementation," in *Control Automation Robotics & Vision (ICARCV), 2014 13th International Conference on.* IEEE, 2014, pp. 1496–1501.

8 Appendix