Distributed Drone Collision Avoidance in Simulink

Luke Rickard

September 26, 2021

Contents

1 Software Requirements

The list of required add-ons to run the Simulink model (in MATLAB R2021a) are as follows:

- Simulink
- DSP System Toolbox
- Optimization Toolbox
- UAV Toolbox

If you wish to run the scripted MATLAB only version, then Simulink and the DSP System toolbox are not required.

2 Problem Statement

The problem we are looking at solving involves the coordination of a fleet of unmanned aerial vehicles (refered to later as "drones" or agents). Each drone has their own individual target destination, and they are required to navigate from some defined starting positions to their target destinations without colliding with other drones. We allow for communication between drones to plan these collision free trajectories, and consider both fixed and time varying communication topologies. We implement this in MATLAB and Simulink. For the associated code see `https://github.com/lukearcus/Distributed_drone_tasks`.

The problem we are looking at solving here can be formulated as an optimisation problem, with a quadratic objective minimising the distance from the goal state. In order to formally describe this problem we will outline the objective function and the constraints, as well as the linearisation of the collision avoidance constraint. In section 4 we will then discuss how this problem can be solved in MATLAB/ Simulink.

The general form of an optimisation problem is

$$\arg\min_{z \in \mathcal{Z}} f(z), \tag{1}$$

and we will discuss the specific form of the objective, $f(z)$, and constraint set, $\mathcal{Z}$.

2.1 Objective

The overall objective function takes the form:

$$f(z) = \sum_{i=1}^{M} J_i(x_i, u_i) = \sum_{i=1}^{M} \left(\sum_{k=0}^{N-1} \left[(x_i(k) - r_i)^\top \mathbf{Q} (x_i(k) - r_i) + u_i(k)^\top \mathbf{R} u_i(k) \right] + (x_i(N) - r_i)^\top \mathbf{Q}_N (x_i(N) - r_i) \right). \tag{2}$$

Whereby we introduce a quadratic penalty for the deviation of the current state $x_i(k)$ (which is the state of agent i at time k) from the reference state r_i. As well as a penalty on the magnitude of the input (which is the acceleration here). We then sum these over the time period, and sum the value for each drone. $\mathbf{Q}_N$ is the terminal matrix to ensure this works for model predictive control, and can be found as the solution of the Riccati equation

$$\mathbf{Q}_N = \mathbf{A}^\top \mathbf{Q}_N \mathbf{A} - \mathbf{A}^\top \mathbf{Q}_N \mathbf{B} (\mathbf{B}^\top \mathbf{Q}_N \mathbf{B} + \mathbf{R})^{-1} \mathbf{B}^\top \mathbf{Q}_N \mathbf{A} + \mathbf{Q}. \tag{3}$$

For a system of the form

$$x(k+1) = \mathbf{A} x(k) + \mathbf{B} u(k), \tag{4}$$

which we will describe later.

2.2 Constraints

There are two constraints to consider in this problem. We have the constraints on the drone dynamics (for which we use a simple double integrator model with bounded acceleration), as well as the collision avoidance constraints.

$$\mathcal{Z} = \mathcal{D} \cap \mathcal{C}. \tag{5}$$

2.2.1 Dynamics

The dynamics we use are of the form:

$$\begin{aligned} p_i(k+1) &= p_i(k) + T v_i(k) \\ v_i(k+1) &= v_i(k) + T a_i(k) \\ \|a_i(k)\|_\infty &\le a_{\max} \\ p_i(k), v_i(k), a_i(k) &\in \mathbb{R}^3 \\ k &\in \{0, \dots, N\}. \end{aligned} \tag{6}$$

With acceleration being the natural choice of control input (so that $u_i(k) = a_i(k)$).

The constraint $\mathcal{D}$ is then the intersection of these constraints for each agent.

$$\begin{aligned} \mathcal{D}_i &= \{(p_i, v_i, a_i) \mid (6)\} \\ \mathcal{D} &= \mathcal{D}_1 \cap \dots \cap \mathcal{D}_M. \end{aligned}$$

2.2.2 Collision Avoidance

The second constraint to consider is the collision avoidance constraint. In order to ensure drones do not collide at any point we impose a limit on the minimum distance at any time step.

$$\mathcal{C}_i = \{p_i \mid \min_{k \in \{3,\dots,N+2\}} \|p_i(k) - p_j(k)\|_2 \geq \Delta, \forall j \in \{1,\dots,M\}\} \tag{7}$$

$$\mathcal{C} = \mathcal{C}_1 \cap \cdots \cap \mathcal{C}_M.$$

Here, Δ is the minimum allowable distance between drones.

This constraint is obviously nonlinear, and non convex. In order to make this more easily solvable we linearise around some trajectories $\bar{p}$, and recover constraints of the form

$$\begin{aligned} g_{ij}^k(p_i, p_j) &= {\eta_{ij}^k}^\top \left[(p_i^k - p_j^k) - (\bar{p}_i^k - \bar{p}_j^k)\right] - \Delta \\ \bar{h}_{ij}(p_i, p_j) &= \min_{k \in \{3,\dots,N+2\}} g_{ij}^k(p_i, p_j) \geq 0, \forall j. \end{aligned} \tag{8}$$

Where $\eta_{ij}^k = \frac{(\bar{p}_i^k - \bar{p}_j^k)}{\|\bar{p}_i^k - \bar{p}_j^k\|_2}$. These nominal trajectories $\bar{p}$ can be computed beforehand, or can simply be the most recent known planned trajectory.

3 Distributed Formulation

In order to solve this optimisation more easily, we use distributed optimisation. This involves each agent optimising locally at each iteration, then sharing results with other agents, and repeating until convergence.

One common way to achieve this is through the alternating direction method of multipliers (ADMM) [1] which solves problems of the form

$$\begin{aligned} \min F_1(x) + F_2(z) \\ \text{subject to: } x \in C_1, z \in C_2 \\ Ax = z. \end{aligned}$$

It achieves this with three main steps

1. $x(k+1) = \arg\min_{x \in C_1} F_1(x) + \lambda(k)^\top Ax + \frac{c}{2}\|Ax - z(k)\|^2$
2. $z(k+1) = \arg\min_{z \in C_2} F_2(z) - \lambda(k)^\top z + \frac{c}{2}\|Ax(k+1) - z\|^2$
3. $\lambda(k+1) = \lambda(k) + c\big(Ax(k+1) - z(k+1)\big).$

In order to place our optimisation in this form we first to need to introduce some duplicates of our optimisation variables (similar to the use of z in ADMM).

We use a copy of each agent's own trajectory, w_i, as well as $w_{i\to j}$, which is agent i's copy of agent j's trajectory. Additionally, we now define $\mathcal{N}_i$ to be all neighbours of agent i (i.e. the set of agents which agent i communicates with). We can then slightly reformulate the problem as

$$\begin{aligned}
\min_{\{p_i, w_i, v_i, u_i\}_i, \{w_{i\to j}\}_{i,j}} & \sum_{i=1}^{M} J_i(x_i, u_i) \\
\text{s.t. } (p_i, v_i, u_i) &\in \mathcal{D}_i \quad \forall i \\
\bar{h}_{ij}(w_i, w_{i\to j}) &\geq 0 \quad \forall i \text{ and } j \in \mathcal{N}_j \\
w_i = p_i; w_{i\to j} &= p_j \quad \forall i \text{ and } j \in \mathcal{N}_j.
\end{aligned}$$

This allows us to obtain a set of update steps, similar to ADMM, which can be used to plan collision free trajectories (as in [2]). There are again three steps, but now with two sets of communication steps. First, we have a *prediction* step which updates p,v and u in parallel. It should be noted that we drop the subscript i but still refer to an individual drone's trajectory, we also use $w_{j\to}$ to refer to agent j's proposed trajectory for the current drone.

$$\begin{aligned}
\arg\min_{p,v,u} & J(x,u) + \lambda^\top (p - w) + \frac{c}{2}\|p - w\|_2^2 + \sum_{j\in\mathcal{N}} \left(\lambda_{j\to}^\top (x - w_{j\to}) + \frac{c}{2}\|x - w_{j\to}\|_2^2\right) \\
\text{s.t. } & (p, v, u) \in \mathcal{D}.
\end{aligned} \tag{9}$$

Next, we communicate our updated trajectory, p, to our neighbours, $j \in \mathcal{N}$, and receive our neighbours updated trajectories $\{x_j\}_{j\in\mathcal{N}}$.

Following this, we have a *coordination* step, which updates our collision free trajectory, w, and our neighbours proposed trajectories, $\{w_{\to j}\}_{j\in\mathcal{N}}$.

$$\begin{aligned}
\arg\min_{w, \{w_{\to j}\}_{j\in\mathcal{N}}} & \lambda^\top (p - w) + \frac{c}{2}\|p - w\|_2^2 + \sum_{j\in\mathcal{N}} \left(\lambda_{\to j}^\top (x_j - w_{\to j} + \frac{c}{2}\|x_j - w_{\to j}\|_2^2\right) \\
\text{s.t. } & \bar{h}_j(w, w_{\to j}) \geq 0 \quad \forall j \in \mathcal{N}.
\end{aligned} \tag{10}$$

The penultimate step is a *mediation* step where we update the dual variables λ and $\{\lambda_{\to j}\}_{j\in\mathcal{N}}$.

$$\begin{aligned}
\lambda &\leftarrow \lambda + c(p - w) \\
\lambda_{\to j} &\leftarrow \lambda_{\to j} + c(p_j - w_{\to j}) \quad \forall j \in \mathcal{N}.
\end{aligned} \tag{11}$$

Finally, we communicate $\lambda_{\to j}$ and $w_{\to j}$ to all neighbours and receive $\lambda_{j\to}$ and $w_{j\to}$ from neighbours.

4 Implementation

In order to get the optimisations to work in MATLAB they need to be put in the form

$$\arg\min_x \frac{1}{2}x^\top Hx + f^\top x \text{ such that } \begin{cases} Ax \leq b \\ A_{\text{eq}}x = b_{\text{eq}} \\ lb \leq x \leq ub \end{cases}, \tag{12}$$

we will now briefly run through the rearrangements for the prediction and communication steps. For a more detailed discussion

Once again, we have ommitted the subscript i for simplicity.

4.1 Prediction Step

Starting from (6) and then stacking $p_i(k) \in \mathbb{R}^{n_d}$ and $v_i(k) \in \mathbb{R}^{n_d}$ into a vector $x_i(k) \in \mathbb{R}^{n_x}$. Where n_x refers to the number of states, and n_d) the number of dimensions (we use three), similarly, n_u refers to the number of inputs (here three since $n_u = n_d$). The constraints can be rewritten in the form

$$\begin{aligned} x(k+1) =& \ \mathbf{A}x(k) + \mathbf{B}u(k) \\ \mathbf{A} =& \begin{bmatrix} \mathbf{I}^3 & T \cdot \mathbf{I}^3 \\ \mathbf{0}^3 & \mathbf{I}^3 \end{bmatrix} \\ \mathbf{B} =& \begin{bmatrix} \mathbf{0}^3 \\ T \cdot \mathbf{I}^3 \end{bmatrix} \\ \|u(k)\| \leq& \ a_{\max} \\ x(0) =& x_0, \end{aligned}$$

where T is the sample time, and x_0 comprises the initial conditions. Finally, by stacking the state vectors $x(k)$ for each time step on top of one another, and then stacking this on top of a similar vector of control inputs,

$$s = \begin{bmatrix} x(0) & x(1) & \cdots & x(N) & u(0) & u(1) & \cdots & u(N) \end{bmatrix}^\top \in \mathbb{R}^{n_x(N+1)+Nn_u},$$

we can write the constraints in the same form as (12).

$$\begin{bmatrix}(\tilde{\mathbf{A}}-\mathbf{I}) & \tilde{\mathbf{B}}\end{bmatrix} s = \begin{bmatrix}-x_0\\ 0\\ 0\\ \dots\\ 0\end{bmatrix}$$

$$\begin{bmatrix}-\infty\\ \dots\\ -\infty\\ -a_{\max}\\ \dots\\ -a_{\max}\end{bmatrix} \leq s \leq \begin{bmatrix}\infty\\ \dots\\ \infty\\ a_{\max}\\ \dots\\ a_{\max}\end{bmatrix}$$

where $\tilde{\mathbf{A}} = \begin{bmatrix}\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0}\\ \mathbf{A} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{A} & \cdots & \mathbf{0} & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A} & \mathbf{0}\end{bmatrix}$ and $\tilde{\mathbf{B}}$ is defined similarly.

The objective function from (9) can also be rearranged to the necessary form, ultimately arriving at

$$\frac{1}{2}s_i^\top \begin{bmatrix}\tilde{\mathbf{Q}} & \mathbf{0}\\ \mathbf{0} & \tilde{\mathbf{R}}\end{bmatrix} s_i + \hat{q}^\top s_i.$$

Where $\tilde{\mathbf{Q}} = \Big(\begin{bmatrix}\mathbf{Q}\otimes\mathbf{I}^N & \mathbf{0}^{n_x\times n_x}\\ \mathbf{0}^{n_x\times n_x} & \mathbf{Q}_N\end{bmatrix} + \frac{|\mathcal{N}|c}{2}\mathbf{I}^{N+1}\Big)$, $\tilde{\mathbf{R}} = \mathbf{R}\otimes\mathbf{I}^{N+1}$ and $\hat{q}$ is defined as in (13). We use $\otimes$ to define the kronecker product, and $|\cdot|$ to define the set cardinality.

$$\begin{aligned}\hat{q}^\top &= \frac{1}{2}\begin{bmatrix}\big(-\tilde{q}^\top + \big(\tilde{\lambda}^\top - c\tilde{w}^\top + \sum_{j\in\mathcal{N}}(\tilde{\lambda}_{j\to}^\top - c\tilde{w}^\top)\big)\mathbf{H}\big)\\ \mathbf{0}^{Nn_u\times 1}\end{bmatrix}\\ \tilde{q} &= \begin{bmatrix}2\mathbf{Q}r & \dots & 2\mathbf{Q}r & 2\mathbf{Q}_N r\end{bmatrix}^\top \in \mathbb{R}^{n_x(N+1)}\\ \mathbf{H} &= \begin{bmatrix}\mathbf{I}^{n_d} & \mathbf{0}^{n_x-n_d}\end{bmatrix}\otimes\mathbf{I}^{N+1}.\end{aligned} \tag{13}$$

$\tilde{\lambda}$ and $\tilde{w}$ are the augmented versions of λ and w, achieved by stacking the dual variables for each time step on top of one another.

4.2 Communication Step

Once again we start by taking all the optimisation variables and stacking them to obtain a vector encompassing all the necessary variables as

$$\tilde{w} = \begin{bmatrix}w(0) & \dots & w(N) & w_{\to 1}(0) & \dots & w_{\to 1}(N) & \dots & w_{\to M}(0) & \dots & w_{\to M}(N)\end{bmatrix}^\top \in \mathbb{R}^{n_d(|\mathcal{N}|+1)(N+1)},$$

which contains the proposed trajectories for ourself as well as all our neighbours.

Following this, we can write out the reformulated optimisation problem

$$\begin{aligned} &\arg\min_{\tilde{w}} \frac{1}{2}\tilde{w}^\top \mathbf{P}\tilde{w} + q^\top \tilde{w} \\ &\text{subject to: } -\mathbf{A}\tilde{w} \leq l. \end{aligned} \tag{14}$$

In this formulation we have $q^\top = \begin{bmatrix}(-\tilde{\lambda} - c\tilde{p}) & (-\tilde{\lambda}_{\to 1} - c\tilde{p}_1) & \cdots & (-\tilde{\lambda}_{\to M} - c\tilde{p}_M)\end{bmatrix}$, and $\mathbf{P} = \frac{c}{2}\mathbf{I}^{n_d|\mathcal{N}|(N+1)}$. The inequality constraint is the transformed version of the collision avoidance constraint with

$$\mathbf{A} = (\mathbf{I}^{N+1} \otimes \eta)\hat{\mathbf{H}}$$

$$\eta = (\mathbf{I}^{|\mathcal{N}|} \otimes \mathbf{1}^{1\times n_d})\begin{bmatrix}\eta_{i1}(0) & \dots & \eta_{i1}(N) & \dots & \eta_{iM}(0) & \dots & \eta_{iM}(N)\end{bmatrix}$$

$$\hat{\mathbf{H}} = \begin{bmatrix}\mathbf{I}^{N+1} \otimes (\mathbf{1}^{|\mathcal{N}|\times 1} \otimes \mathbf{I}^{n_d}) & -\mathbf{I}^{N+1} \otimes (\delta_1 \otimes \mathbf{I}^{n_d}) & \cdots & -\mathbf{I}^{N+1} \otimes (\delta_M \otimes \mathbf{I}^{n_d})\end{bmatrix}$$

$$\delta_j = \begin{array}{c} \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \end{array} \begin{array}{c} 1 \\ \vdots \\ j \\ \vdots \\ M \end{array}$$

$$l = \begin{bmatrix} \Delta + \eta_{i1}(0)(\bar{p}(0) - \bar{p}_1(0)) - \|\bar{p}(0) - \bar{p}_1(0)\| \\ \vdots \\ \Delta + \eta_{i1}(N)(\bar{p}(N) - \bar{p}_1(N)) - \|\bar{p}(N) - \bar{p}_1(N)\| \\ \vdots \\ \Delta + \eta_{iM}(N)(\bar{p}(N) - \bar{p}_M(N)) - \|\bar{p}(N) - \bar{p}_M(N)\| \end{bmatrix}.$$

4.3 Structure

With the various steps reformulated as standard quadratic programs we can now look at how to structure the overall algorithm.

4.3.1 Path Planning

Firstly, we will discuss the actual path planning structure, before then discussing how we can interface with other systems in Simulink.

The first thing to run through is the setup, in order to run properly we need to input a number of constants, and initialise a number of variables. The desgin variables we need to choose are $\mathbf{Q}, \mathbf{R}, c, T, N, \Delta$ and the necessary inputs are $\mathbf{A}, \mathbf{B}, x_0, r, a_{\max}, M, \mathcal{N}$. With these defined we then plan an initial trajectory for the drones by simply minimising the objective $J(x, u)$ whilst following the drone dynamics, but ignoring collision avoidance. This initial trajectory is then used as an initial point for the active-set solver [3,4] for the first prediction step.

Following this setup we can simply loop through *prediction*, *communication*, *coordination*, *mediation* and a *final communication*. This should be done until some stopping criterion is reached, however in order for this to run properly in simulink we must instead use a fixed number of iterations (which we set beforehand).

Following these iterations we then extract the accerelation commands from the collision free trajectories w using $a_{\text{safe}}(k) = \frac{w(k+2)-2w(k+1)+w(k)}{T^2}$. These may not work perfectly since they do not consider drone dynamics, so it is also possible to extract the acceleration part of the vector from the prediction step, this gives feasible accelerations but does not guarantee collision avoidance.

4.3.2 Interfacing

With this setup correctly we can then call this path planning sub-system at any point to get improved trajectories. To do this we simply set x_0 to be the current state and $\mathcal{N}$ to define the current connectivity (if different), then once the path planning is complete we receive new acceleration commands for, up to, the next N time steps.

These acceleration commands will simply be a 3D acceleration vector, this can be converted to a more familiar, pitch-roll-yawrate-thrust ($\Phi - \theta - \dot{\Psi} - T$) command with the following equations (assuming a North-East-Up coordinate frame for a)

$$\begin{aligned}
\Phi &= \operatorname{atan2}(-a_x \cos(\Psi) - a_y \sin(\Psi), a_z + g) \\
\theta &= \operatorname{atan2}((-a_x \sin(\Psi) + a_y \cos(\Psi)) \cos(\Phi), a_z + g) \\
\dot{\Psi} &= 0 \\
T &= \frac{m_{\text{drone}}(a_z + g)}{\cos\Phi \cos\theta},
\end{aligned} \tag{15}$$

where atan2$(\cdot)$ is the four-quadrant inverse tangent. It is then possible to feed these commands into a MATLAB UAV guidance model.

5 Challenges

In implementing this work in to MATLAB/ Simulink we found some parts to be more challenging than others, in this section we detail any parrticular changes we faced and the solution to such challenges.

Implementing the path planning in MATLAB was relatively straightforward, largely due to previous code being available thanks to the work of A. Karapetyan [5]. The only minor issues faced at this stage involved some confusion over labelling of variables (which arose in part due to the fact we have $n_u = n_d$ here), this confusion should be cleared up in the code but it is possible that this is not entirely fixed.

Following the succesful implementation in MATLAB we then moved on to get the code to work in Simulink, this created a number of interesting challenges we had to solve.

A relatively minor first hurdle came with the difference in implementation of *quadprog* in MATLAB and Simulink. In Simulink the only available solver is the *active-set* solver which requires a feasible initial point, and does not accept sparse matrices at the input. However, once the code had been adjusted to provide a feasible initial point (generally by using the result from the previous iteration), this did manage to run. There was a severe decrease in the performance from approximately 10s runtimes to 600s which may be in part due to the fact that we no longer explot the sparsity of the problem, and that it is a generally large optimisation problem. In order to attain reasonable runtimes the prediction horizon was reduced by approximately half to 30 steps and the code placed in a system object, we then saw runtimes of approximately 100s. Reduction of the prediction horizon beyond this seemed to lead to a large deterioration in performance.

Another issue faced at this stage involved the Simulink UAV Guidance Model which was not designed to be used in a foreach subsystem and so had to be amended. Once this had been amended the initial conditions for each drone could not be changed independently (i.e. they all recorded identical start positions at the world origin), to fix this we simply added the initial conditions to the state output. In addition to this, we had been using a fixed step solver with a sample time of 0.1s and found that the UAV Guidance Model behaved very strangely under this solver, recording values far exceeding what we would expect for all states.

Finally, we faced some difficulty with the differing update rates (where we required the path planning to happen much less frequently than the simulation update step). In particular placing a rate transition block between the path planning and simulation parts lead to the path planning being a step behind (i.e. always using the state of the previous call). Initially, a triggered subsystem was used (and this is still available with the code) but it should also now work with the different sample rates following the introduction of a small delay of approximately 10^{-3}s before the rate transition.

6 Case Study

In this section, we provide details of some results using this code which are useful in demonstrating its effectiveness.

Firstly, a useful starting point is the path planning with no considerations of collision avoidance. In this case agents move directly towards their goals, typically overshooting this goal and then returning. The plot of these paths can be seen in Figure 1. The lines show the trajectories of the drones, the circles their initial positions, their goals are shown with crosses (these are not visible here since they're covered by the stars which show the drones' final positions), the red diamonds show points at which the drones were closer than

$\Delta = 0.4\,\mathrm{m}$ (i.e. where the collision avoidance has failed). In order to run the code without consideration of the collision avoidance we have prevented drones from communicating at all.

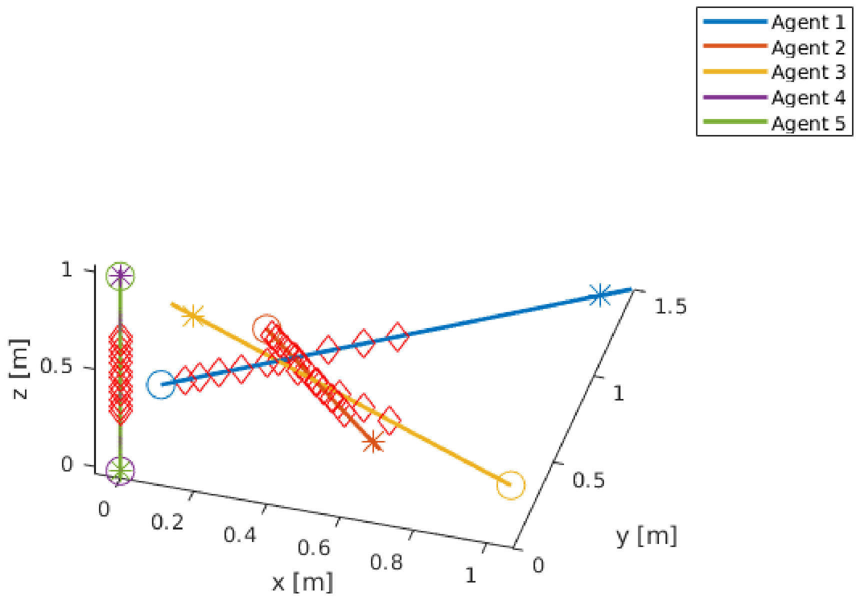


Figure 1: Path planning without collision avoidance constraint

We can clearly see from the plot in Figure 1 that drones are able to navigate to their goal positions, however they do collide many times in this scenario. If we now compare this with the paths taken when the drones are actively avoiding collisions we can see the effectiveness of our approach.

Figure 2 provides the paths taken with a fully connected communication graph. We use the exact same initial and goal states as in Figure 1 but now we see that there are no collisions at any point in the drones' trajectories, whilst drones still end up reaching their target positions.

In both Figure 1 and Figure 2 the trajectories plotted are taken from the UAV guidance model simulink block, with the acceleration commands taken from the path planning step. In Figure 3 we provide plots of the acceleration commands inputted (in orange) as well as the actual accelerations seen in the simulation (in blue).

We can see from the plot in Figure 3a that in the case of a relatively local target position the drone does a very good job at tracking the acceleration commands on all three axes. In this case the maximum deviation in acceleration was approximately $0.0245\,\mathrm{m\,s^{-2}}$. In the second plot, Figure 3b, we see that when much larger accelerations are requested there can be some more significant error

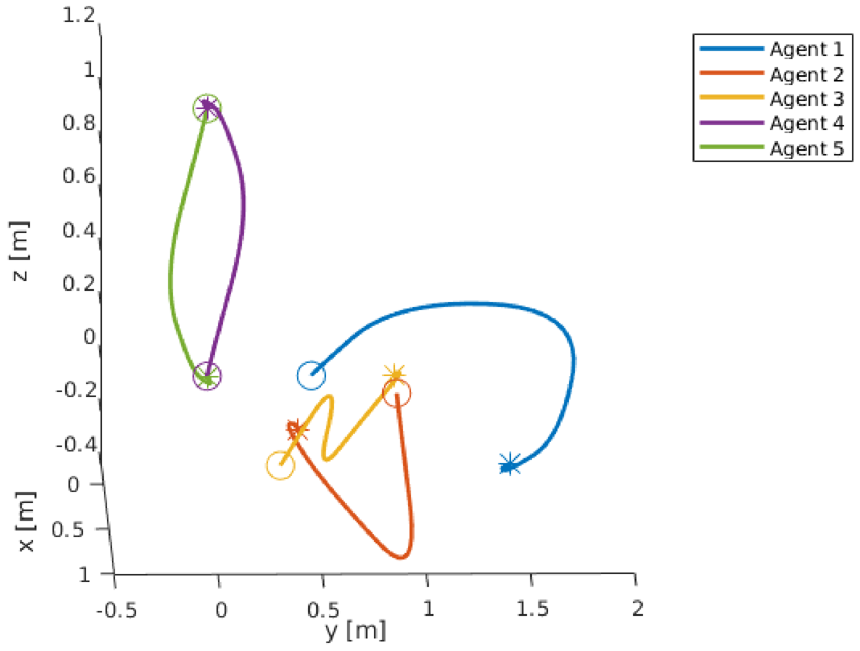
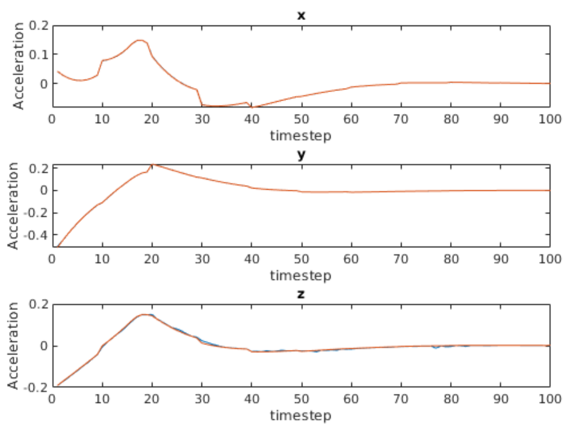
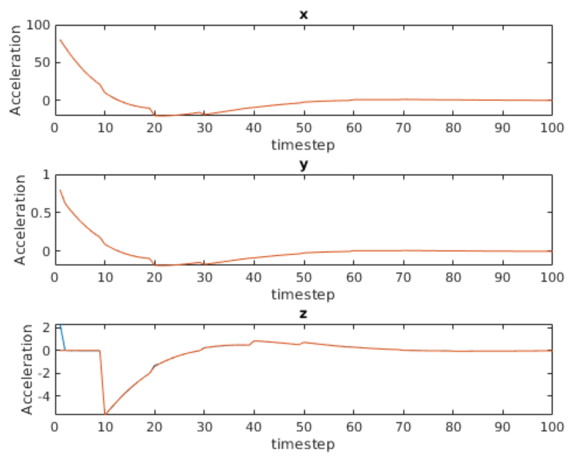


Figure 2: Drone trajectories with full communication and collision avoidance



(a) Target approximately 1 m away



(b) Target approximately 100 m away

Figure 3: Drone target and actual accelerations when navigating to differently distant targets

(particularly in the z direction, arising from the significant thrust needed to achieve the large acceleration in the x direction). Here we reach a maximum deviation of $2.3275\,\mathrm{m\,s^{-2}}$, resulting in a significant deviation in position as well. In order to ensure such large deviations are avoided it is generally best to limit acceleration and set more frequent, but less distant, waypoints.

7 Future Work

There are a number of avenues available for future work and we will briefly discuss a few options here.

7.1 Waypoints

Currently, the simulink code only uses a single reference states, a good extension would be to introduce the ability to allow for multiple reference states (or *waypoints*). We consider there to be two main options for this implementation, either new waypoints are provided after a fixed time, or new waypoints are provided only once a drone has become sationary at the current waypoint. The latter option could require some consideration in the event that a drone is entirely unable to reach a waypoint.

7.2 Dynamic Graph Connection

We have implemented the ability to change communicaiton graphs, however, this currently relies upon fixed communication graphs being provided beforehand. A more realistic scenario would allow for the graph to be updated during execution, with agents only communicating with other agents who are within some fixed communication distance when the path planning begins.

7.3 Fixed Obstacles

Another avenue for further improvement involves fixed obstacles, currently the drones can avoid each other but there is no consideration of fixed obstacles. Some possibilities to improve this could be to have fixed limits for some directions (i.e. have a lowerbound on a drone's position in the z axis to prevent collision with the ground); to have drones consider a "dummy drone" when they detect an obstacle that is placed at the obstacle with no dynamics (this dummy drone could then be forgotten when the obstacle is no longer in sight; or to use a simplistic method whereby the drone is instructed to yaw right when it sees an object in front of it (although this might cause problems since it results in two collision avoidance directives and prioritising one could lead to a failure of the other).

7.4 Gradient Descent

Finally, an entirely separate avenue could involve looking in to entirely removing the ADMM architecture and using a soft collision avoidance constraint (i.e. by putting it in the objective function) and then using projected subgradient descent to plan a new path.

References

[1] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers, 2010.

[2] Felix Rey, Zhoudan Pan, Adrian Hauswirth, and John Lygeros. *Fully Decentralized ADMM for Coordination and Collision Avoidance; Fully Decentralized ADMM for Coordination and Collision Avoidance*. 2018.

[3] MATLAB quadprog. `https://uk.mathworks.com/help/optim/ug/quadprog.html`. Accessed: 2021-09-17.

[4] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2019.

[5] A.Karapetyan GitHub. `https://github.com/akarapet/admm_collision_avoidance`. Accessed: 2021-09-17.

A Live Script

Collision avoidance simulink model

This code shows how to setup a simulink model for distributed collision avoidance. Including description of the underlying ADMM (Alternating Direction Method of Multipliers) algorithm.

Table of Contents

Model Overview

We will begin by exploring the simulink model, looking at the connectivity and layout of the various subsystems used in the simulation.

Open the Model

Firstly, clear the workspace and add functions to the current path, then open up the chosen simulink model. There are 2 slightly different models but both have the same main steps, triggered subsystem simply allows for triggering the path planning according to pulses.

```
clear
addpath("Funz");
mode = "Triggered_subsystem_mdl";
open_system(mode);
```

Inputs & outputs

The planning and simulation subsystem takes in 3 inputs: A connectivity graph (or a set of connectivity graphs for varying communication), an initial state vector (x0) and a reference/ goal state (r). In order to add more drones to the simulation, simply add additional rows to r and x0, and increase the size of the connectivity graph.

The initial and reference states should be input as matrices with each row being a 1x6 row vector containing the relevant state. This vector contains the XYZ position and their derivatives in a North-East-Up coordinate frame.

In addition to these inputs we also need to specify a few further parameters. These are the number of steps to look ahead during the path planning (N), the minimum separation between drones (delta), the maximum acceleration of a drone in any direction (note this script allows a drone to accelerate maximally in all 3 directions at once), the simulation run time, the time period between updates to the path planning and the discretisation timestep used in the path planning.

These can be setup as follows (or a premade scenario can be selected).

```
option = "custom";
if option ~="custom"
    addpath("Scenarios");
    load(option);
else
    N =30 ;%number of steps to look ahead
    delta = 0.4; %collision avoidance separation
    max_acc = 1; %max drone acceleration
    endTime = 10; %simulation run time
    MPC_period = 1; %seconds between each trajectory update

    T = 0.1; %discretisation time step

    r = [
         1.0,1.4,0.0, 0,0,0;
         0.6,0.4,0.0, 0,0,0;
         0.0,0.9,0.0, 0,0,0;
         0.0,0.0,1.0, 0,0,0;
         0.0,0.0,0.0, 0,0,0;
         ];

    x0 = [
          0.0,0.5,0.0, 0,0,0;
          0.2,0.9,0.0, 0,0,0;
          1.0,0.3,0.0, 0,0,0;
          0.0,0.0,0.0, 0,0,0;
          0.0,0.0,1.0, 0,0,0;
          ];

    M = size(x0,1); % Number of agents (derived from number of rows of x0)

    graph_form{1} = ones(M) - eye(M); %connectivity graph
    % graph_form{1} = [0 1 0 0; 1 0 0 0; 0 0 0 1; 0 0 1 0]; %define connection graphs
    % graph_form{2} = [0 0 1 0; 0 0 0 1; 1 0 0 0; 0 1 0 0];
    % graph_form{3} = [0 0 0 1; 0 0 1 0; 0 1 0 0; 1 0 0 0];

    graph_update_step = 1; %how many MPC calls before connection graph is updated

    %puts graphs into a 3D matrix to allow proper iteration
    if size(graph_form,2) == 1
        graph_form{2} = graph_form{1};
    end


    for i = 1:size(graph_form,2)
```

```
        graph(:,:,graph_update_step*(i-1)+1:i*graph_update_step) = graph_form{i};
    end
end
```

Upon completion the subsystem outputs the positions of the drones sampled at the chosen time step (T).

Model Architecture

Opening this provides some insight into the separate planning and simulation steps.

```
open_system(mode + '/Planning and simulation');
```

The path planning block takes the current states in as the initial states for the current iteration, as well as the fixed reference states and the current connectivity graph. This block then loops over a number of iterations of the ADMM algorithm (which will be explored later) and outputs accelerations for the next *N* time steps.

These accelerations are then fed into a Simulation subsystem. The resulting state is then sent back to the path planning block.

Simulation Subsystem

```
open_system(mode + '/Planning and simulation/Simulate');
```

In this subsytem we find the current acceleration command (found by taking the *k*th acceleration vector from the most recent path planning output) and then convert this from a simple 3D acceleration to pitch, roll, thrust commands for the UAV guidance model (since this provides us with 3 DoF we simply set yaw rate to 0, but do still use it when calculating commands in case of any drift).

These commands are then fed into a control bus which is fed into the UAV guidance block. The UAV states are then updated and outputted, with the yaw being fed back around.

The UAV guidance block outputs a 13D state vector which includes euler angles and thrust. For our path planning subsystem we only require positions and velocities, and so the change coord frame selects these and converts them from a North-East-Down coordinate frame to a North-East-Up coordinate frame with a global origin.

Path Planning Subsystem

```
if mode == "Triggered_subsystem_mdl"
    path = mode + '/Planning and simulation/Path Planning/For loop';
else
    path = mode + '/Planning and simulation/Path Planning';
end
open_system(path);
```

The final subsystem to explore is the path planning subsystem, in this subsystem the UAVs use the ADMM algorithm to plan a feasile trajectory w. They then calculate the acceleration at each time step as follows:

$$a(k) = \frac{w(k+2) - 2\,w(k+1) + w(k)}{T^2},$$

and output this information for the simulation step.

In this subsystem each block is a step of the ADMM algorithm, with the exception of x_bar setup which runs only on the first iteration to set up initial trajectories.

ADMM Algorithm

We now move on to explore the ADMM algorithm, with an explanation of each step of the algorithm and follow a single iteration.

Parameters

In order to run the ADMM algorithm we first need to define a few key parameters. We start with the model definition, in order to simplify the optimisation we have chosen a very simple double integrator model with updates of the form:

$$\begin{aligned} p(k+1) &= p(k) + T\,v(k) \\ v(k+1) &= v(k) + T\,a(k) \end{aligned}$$

p,v and a being position, velocity and acceleration respectively, with acceleration being the chosen control input.

We also define the number of states (6), number of inputs (3 for acceleration in each direction) and finally the number of dimensions.

Finally, we have the objective matrices, which are defined as in LQR control. These can be adjusted for differing performance depending on specific requirements.

```
A = [1 0 0 T 0 0;
    0 1 0 0 T 0;
    0 0 1 0 0 T;
    0 0 0 1 0 0;
    0 0 0 0 1 0;
    0 0 0 0 0 1];

B = [0 0 0;
    0 0 0;
    0 0 0;
    T 0 0;
    0 T 0;
    0 0 T];

m = 0.1; %Mass

%set up model structure
model.A = A;
model.B = B;
model.umax = max_acc;
model.umin = -max_acc;
```

```
nx = size(model.A,1); %state vector dimension
nu = size(model.B,2); %number of inputs
Nd = 3; %number of dimensions

%Objective matrices setup (note that we ignore velocity in the
%optimisation)
PosQ = eye(Nd)*10;
velQ = eye(nx-Nd)*0;
Objective.Q = blkdiag(PosQ,velQ);
Objective.R = eye(nu)*13;
Objective.N = N;
QN = dlqr(A,B,Objective.Q,Objective.R);

Acc_from_pos = eye((N+1)*Nd) + ...
               [zeros((N)*Nd,Nd) -2*eye(N*Nd); zeros(Nd,(N+1)*Nd)] + ...
               [zeros((N-1)*Nd,2*Nd) eye((N-1)*Nd); zeros(2*Nd,(N+1)*Nd)];
Acc_from_pos = Acc_from_pos/T^2; %finds accelerations based on only position
Bus_setup; %sets up buses for model
```

Initial Trajectory

In order to start the optimisation we need some initial trajectory to optimise around, for this we simply take an optimal trajectory which does not account for collision avoidance as follows:

```
x_init = zeros(nx*(N+1),M);
u_init = zeros(nu*N,M);
for i = 1:M
    [x_init(:,i),u_init(:,i)] = init_traj(ADMM_vec(i).rho,Objective,constraints_struct(
end
```

Init_traj which simply minimises $\sum_0^N x^T Q\, x + u^T R\, u$ whilst ensuring the planned trajectory is feasible (i.e. is possible according to the simple quadrotor models employed and does not exceed the maximum acceleration).

We then communicate these initial trajectories to all neighbours (i.e. drones that we can communicate with).

```
for i =1:M
    ADMM_vec(i).x = x_init(:,i);
    ADMM_vec(i).u = u_init(:,i);
    ADMM_vec(i).x_bar(:,i) = x_init(:,i);
    nonzero_N_j = nonzeros(constraints_struct(i).N_j);
    for j = 1:nnz(constraints_struct(i).N_j)
        ADMM_vec(i).x_bar(:,nonzero_N_j(j)) = x_init(:,nonzero_N_j(j));
    end
end
```

Following this all drones have an initial trajectory for all their neighbours and can then begin the main steps of the algorithm.

Predict

This initial step is parallelisable, and we use a for each susbsytem to make use of this. Agents try to minimise the actual cost function (in the first term). The other terms are the augmented lagrangian terms, with the first 2 encompassing the constraint $p_i = w_i$. The final component involves the agent trying to stay close to the trajectory proposed by all its neighnours. The constraints in this step ensure the trajectory follows the agent dynamics (the double integrator model).

$$\arg\min_{p_i,v_i,a_i\in \boldsymbol{S}_i} \boldsymbol{F_i}(p_i,a_i) + \lambda_i^T(p_i - w_i) + \frac{\rho}{2}\|p_i - w_i\|^2 + \sum_{j=1}^{N_i}(\lambda_{j\to i}^T(x - w_{j\to i}) + \frac{\rho}{2}\|$$

This can be recast into a form recognisable to quadprog and the update step is run as follows:

```
for i = 1:M
    admmxlength = nx*(N+1);
    admmulength = nu*(N);
    umin = -max_acc;
    umax = max_acc;
    min_input = repmat(ones(nu,1)*umin,N,1);
    max_input = repmat(ones(nu,1)*umax,N,1);

    u_bound = [inf((N+1)*nx,1);max_input];
    l_bound = [-inf((N+1)*nx,1);min_input];


    posM = blkdiag(eye(Nd),zeros(nx-Nd)); % Matrix to take only position from the state
    d = [eye(Nd),zeros(Nd,nx-Nd)]; % Matrix to take only position from the state vecto
    posMN = kron(eye(N+1),d); %returns pos across time steps

    Ax = kron(speye(N+1), -speye(nx)) + kron(sparse(diag(ones(N, 1), -1)), A);
    Bu = kron([sparse(1, N); speye(N)], B);
    Aeq = [Ax, Bu];
    b_eq = [-constraints_struct(i).x0 zeros(1, N*nx)]';
    Q = Objective.Q;
    R = Objective.R;
    P_new = blkdiag( kron(eye(N), Q+ (nnz(constraints_struct(i).N_j)+1)*ADMM_vec(i).rho
        QN+(nnz(constraints_struct(i).N_j)+1)*ADMM_vec(i).rho/2*posM, kron(eye(N), R) )

    q = prediction_linear(ADMM_vec(i).lambda, ...
        ADMM_vec(i).lambda_from_j, ...
        ADMM_vec(i).w, ...
        ADMM_vec(i).w_from_j, ...
        ADMM_vec(i).rho, ...
        constraints_struct(i).r', Q, N, nu, posMN,nonzeros(constraints_struct(i).N_j),C

    init = [ADMM_vec(i).x; ADMM_vec(i).u];
    opts = optimoptions('quadprog','Display','off','Algorithm','Active-set');
    res = quadprog(P_new,q,[],[], full(Aeq), b_eq, l_bound, u_bound,init,opts);

    ADMM_vec(i).x = res(1:admmxlength);
    ADMM_vec(i).u = res(admmxlength+1 : admmxlength + admmulength);
```

```
end
```

Communicate

Following the prediction step all agents share their proposed trajectories p_i to all their neighbours so that these can then be used to plan collision free trajectories in the coordinate step. Agents then store these as x_bar.

```
for i =1:M
    ADMM_vec(i).x_bar(:,i) = ADMM_vec(i).x;
    nonzero_N_j = nonzeros(constraints_struct(i).N_j);
    for j = 1:nnz(constraints_struct(i).N_j)
        ADMM_vec(i).x_bar(:,nonzero_N_j(j)) = ADMM_vec(nonzero_N_j(j)).x;
    end
end
```

Coordinate

Again, this step is parallelisable and we therefore use a foreach subsystem. Each agent minimises the augmented lagrangian with respect to the z variables (w_i and $w_{i\to j}$ here) and the collision avoidance constrain is enforced, we use the most recent known trajectories for all neighbours for this.

$$\text{arg} \min_{w_i,\{w_{i\to j}\in N_i\}} \lambda_i^T(p_i - w_i) + \frac{\rho}{2}\|p_i - w_i\|^2 + \sum_{j=1}^{N_i}(\lambda_{i\to j}^T(p_j - w_{i\to j}) + \frac{\rho}{2}\|p_j$$

$$\text{subject to: } \overline{h_{ij}}(w_i, w_{i\to j}) \geq 0, \quad j \in N_i,$$

The code for this is as follows:

```
for i = 1:M
    H = kron(eye(N+1),repmat(d, nnz(constraints_struct(i).N_j), 1));
    Hw = kron(eye(N+1),repmat(eye(nu), nnz(constraints_struct(i).N_j), 1));
    for j = 1:nnz(constraints_struct(i).N_j)
        v = zeros(nnz(constraints_struct(i).N_j),1);
        v(j) = 1;
        H_M = kron(eye(N+1),kron(v, -1*d));
        H_Mw = kron(eye(N+1),kron(v, -1*eye(nu)));

        H =  [H, H_M];
        Hw = [Hw,H_Mw];
    end
    rhoM = kron(speye((N+1)),ADMM_vec(i).rho/2*eye(nu)); % matrix for the quadratic obj
    Pc = kron(eye(1+nnz(constraints_struct(i).N_j)),rhoM);


    qc = coordination_linear(ADMM_vec(i).lambda,ADMM_vec(i).lambda_to_j,ADMM_vec(i).rho
        ADMM_vec(i).x_bar,posMN,constraints_struct(i).N_j,i);
    % Update matrices
    actual_N_j = nonzeros(constraints_struct(i).N_j);

    init = ADMM_vec(i).w;
    for j = 1:nnz(constraints_struct(i).N_j)
        init = [init;ADMM_vec(i).w_to_j(:,actual_N_j(j))];
    end
    [A_ineq,l_ineq] = communicate(ADMM_vec(i).x_bar,N,constraints_struct(i).N_j,H,Hw,co

    if A_ineq*init < l_ineq
        init = A_ineq\l_ineq;
    end
    opts = optimoptions('quadprog','Display','off','Algorithm','Active-set');
    resc = quadprog(full(Pc),qc,-A_ineq,-l_ineq,[],[],[],[],init,opts);
    %             resc = qp_grad(full(Pc),qc,-A_ineq,-l_ineq,[],[],[],[],init,1e-2);


    ADMM_vec(i).w = resc(1:Nd*(N+1),1);
    for j = 1:nnz(constraints_struct(i).N_j)
        ADMM_vec(i).w_to_j(:,actual_N_j(j)) = resc(j*obj.Nd*(obj.N+1)+1:(j+1)*obj.Nd*(o
    end
end
```

Mediate

Next, the dual variables (lambda) are all updated.

$$\lambda_i \leftarrow \lambda_i + \rho(x_i - w_i)$$
$$\lambda_{i\to j} \leftarrow \lambda_{i\to j} + \rho(x_j - w_{i\to j}), \quad j \in N_i.$$

```
for i =1:M
    ADMM_vec(i).lambda = ADMM_vec(i).lambda + ADMM_vec(i).rho * (posMN * ADMM_vec(i).x-
    nonzero_N_j = nonzeros(constraints_struct(i).N_j);
    for j = 1:nnz(constraints_struct(i).N_j)
        ADMM_vec(i).lambda_to_j(:,nonzero_N_j(j)) = ADMM_vec(i).lambda_to_j(:,nonzero_N
        ADMM_vec(i).rho * (posMN*ADMM_vec(i).x_bar(:,nonzero_N_j(j)) - ADMM_vec(i).w_to
    end
end
```

Final Communication

Finally, each agent communicates its proposed trajectories witoj and dual variables lambdaitoj to all its neighbours and receives these variables from its neighbours.

```
for i = 1:M
    nonzero_N_j = nonzeros(constraints_struct(i).N_j);
    for j = 1:nnz(constraints_struct(i).N_j)
        curr_comm = nonzero_N_j(j);
        ADMM_vec(i).lambda_from_j(:,nonzero_N_j(j)) = ADMM_vec(curr_comm).lambda_to_j(:
        ADMM_vec(i).w_from_j(:,nonzero_N_j(j)) = ADMM_vec(curr_comm).w_to_j(:,i);
    end
```

```
end
```

We would then return to the prediction step and continue to iterate for a fixed number of iterations.

Run Whole Simulation

We can now return to the simulink model which contains all these steps and runs them a fixed number of times for us.

```
w=warning('off','all');
tic
simout = sim(mode,'StopTime', string(endTime));
sim_time = toc
```

View results

Finally, we can observe the results to see the final paths taken by the drones. In this step we also print out the final minimum separation between any 2 drones.

The initial positions of the drones are shown with circles, their goals are shown with crosses, and any points at which drones have been closer than the minimum specified by delta are shown with a red diamond.

```
x = reshape(simout.yout{1}.Values.Data(:,:,:),M,[])';
visualise_drones_3d(r,x,T,nx,delta);
```

B Functions

```
function q = prediction_linear(lambda,lambda_from_j,w,
    w_from_j,rho,r,Q,N,nu,V,N_j,QN)


q = -(rho/2 * w' * V)' + 0.5*(lambda'*V)';

sigma_j = zeros(size(q));
for j = 1:nnz(N_j)
    sigma_j = sigma_j + 0.5 * (lambda_from_j(:,N_j(j))' *
        V)' - (rho/2 * (w_from_j(:,N_j(j)))' * V)';
end

q=q+[repmat(-Q*r, N, 1); -QN*r] + sigma_j;
q = [q;zeros(N*nu, 1)];

end
```

```
function qc = coordination_linear(lambda,lambda_to_j,rho,
    x,V,N_j,i)


qc = -0.5 * lambda - rho/2 * V * x(:,i);

N_j = nonzeros(N_j);
for j = 1:nnz(N_j)

    qc = [qc;( -0.5 * lambda_to_j(:,N_j(j)) - rho/2 * V *
        x(:,N_j(j)))];

end

end
```

```
function [A_ineq,l_ineq] = communicate(x,N,N_j,H,Hw,delta
    ,Nd,i)

% placeholders
A_ineq = zeros(N*nnz(N_j),(N+1)*Nd*(nnz(N_j)+1));
l_ineq = zeros(N*nnz(N_j),1);

x_bar = x(:,i);
nonzero_N_j = nonzeros(N_j);
for j = 1:nnz(N_j)
    x_bar = [x_bar;x(:,nonzero_N_j(j))];
```

```
end

delta_x_bar = H * x_bar;

for k = 2:N+1

    eta_M_k = zeros(nnz(N_j),Nd*nnz(N_j)); % placeholder
    delta_x_k = delta_x_bar((k-1)*Nd*nnz(N_j)+1:k*Nd*nnz(
        N_j)); % get k-th delta_x

    for m = 1:nnz(N_j)

        x_bar_norm = norm(delta_x_k((m-1)*Nd+1:m*Nd)); %
            get the 2 norm of x_bar
        eta_ij_k = delta_x_k((m-1)*Nd+1:m*Nd)' * 1/
            x_bar_norm; % formulate eta ij k
        eta_M_k(m,(m-1)*Nd+1:(m-1)*Nd+Nd) = eta_ij_k; %
            populate matrix eta_M

        l_ineq((k-2)*nnz(N_j)+m) = delta + eta_ij_k *
            delta_x_k((m-1)*Nd+1:m*Nd) - x_bar_norm; %
            fill in l_ij

    end

    A_ineq( (k-2)*nnz(N_j)+1 : (k-1)*nnz(N_j) , :) =
        eta_M_k * Hw((k-1)*nnz(N_j)*Nd+1:(k)*nnz(N_j)*Nd
        ,:);

end

% A_ineq(isnan(A_ineq)) = 0;
% l_ineq(isnan(l_ineq)) = 0;

end
```