Multi-Robot Ground Swarms Using Minimally Invasive Safety Critical Controller Solved Using Dykstra's Projection Algorithm

Hayato Kato
ECE236C - Optimization Methods for Large-Scale Systems

June 5, 2022

1 Introduction

One of the many hot topics in modern robotics research is the area of swarm robotics, where multiple units need to work together with each other to accomplish a certain task. One common task of such application is to ensure collision avoidance amongst the other robots in its proximity for safe operation. Borrmann and his colleagues solved this problem by proposing a minimally invasive safety critical control of multi-agent ground swarms [3]. This approach involved defining a control safety function and using that to obtain a centralized safety barrier certificate that ensures that no two robots would come too close to each other. The certificate is obtained by solving an optimization problem at each global time step which computes the optimal control input for each agent which minimizes the penalty for approaching too close to a neighboring robot.

2 Optimization Model

The control safety function used to formulate this control problem as a safety critical quadratic program is shown below:

$$u_{act} = \underset{u \in \mathbb{R}^m}{\arg\min} \frac{1}{2} ||u - u_{des}(x)||_2$$

$$s.t.\ L_g h_{i,j}(x) u + \alpha_{i,j} h_{i,j}(x) \geq 0, \forall i, j \in 1, ..., N, i \neq j \tag{1}$$

where

$$\dot{x}_i = u_i, x_i \in \mathbb{R}^2, u_i \in \mathbb{R}^2 \tag{2}$$

$$h_{i,j}(x) = ||x_i - x_j||^2 - D_S \tag{3}$$

$$S_{i,j} = \{x \in \mathbb{R}^{2N} | h_{i,j}(x) \geq 0\} \tag{4}$$

Here, x_i represents the ith agent's state, which consists of its position in the world frame. u_i represents the input velocity given to each agent, where u_{des} is the desired velocity originally set to accomplish the task at hand and u_{act} is the actual command that gets sent to each agent after being filtered by the "safety filter". h represents the safety function that dictates what is considered to be "safe" in this system, and the safe set S is derived directly from this safety function. α is used here to represent the derivative condition that needs to be met by the safety function to ensure that the system never leaves the safety set, i.e. is positive definite. This is a user parameter that can be adjusted to modify the behavior of the swarm which dampens the collision avoidance response of each agent, thus can be treated as a predefined constant [1].

3 Convex Analysis

The cost function is a simple Euclidean norm, which is known to be convex. By observing the terms within the constraints, it is seen that both the Lie derivative of the safety function and the safety function itself is only dependent on the current state of the system, which would be known and act as a constant. This is because this optimization program is computed for each time step, thus making the constraint merely a collection of hyperspaces which overlap to create a convex set. It is determined that a modified version of the proximal gradient method is probably ideal for solving this type of optimization program, since it consists of a differentiable function that has constraints which would be dealt with using a collection of indicator functions. Upon asking for advice from the TA, Dykstra's projection algorithm was used to simplify the projection computation onto the entire set by individually projecting the proposed control input onto each halfspace [2].

4 Implementation

The CVX implementation of the problem was tested in order to act as a comparison against the proposed technique using Dykstra's projection algorithm. Here, the implementation itself was fairly straightforward, with the problem being properly set up by defining the cost function and the multiple constraints using a for loop, as shown below:

```
cvx_begin quiet sdp
    variable u(2*Swarm.N,1);
    minimize 0.5*power(2,norm(u - u_des,2));
    subject to
       for i = 1:Swarm.N
           for j = 1:Swarm.N
               if i ~= j
                   Swarm.Lgh(i,j)*u >= -Swarm.ALPHA * Swarm.h(i,j);
               end
           end
       end
cvx_end
```

In contrast, the implementation for the Dykstra's projection algorithm was done by following the proximal gradient algorithm's updates and slightly modifying the projection step by dividing it into multiple projections onto multiple halfspaces, one at a time:

```
u = zeros(2*Swarm.N,1);
t = 1;
temp = u - t*(u-u_des);
for i = 1:Swarm.N
   for j = 1:Swarm.N
       if i ~= j
           b = Swarm.ALPHA * Swarm.h(i,j);
           a = -Swarm.Lgh(i,j);
           % Projection iff the constraint is not met
           if a*temp > b
               temp = temp + (b-a*temp)*a'/(a*a');
           end
       end
   end
end
u = temp;
```

Here, the formula for the projection onto a halfspace was borrowed directly from the class slides on page 6.11 [4]. The original Dykstra's algorithm usually requires these projections to be repeated over several iterations for it to converge towards the solution of the original problem. However, this step is omitted under the assumption that the approximate solution is good enough for this specific application, since this tentative solution still satisfies all constraints given.

5 Results

Both algorithms were implemented within MATLAB, and the simulation results are shown below:

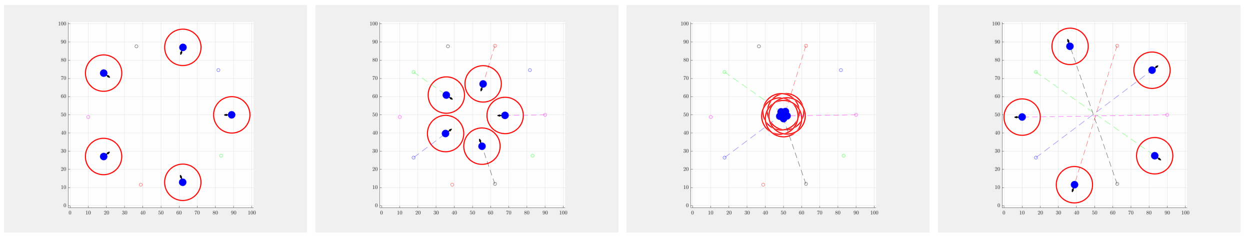

Figure 1: Desired Trajectory of Robot Swarm (Link to animated GIF: https://imgur.com/a/WEjvLcL)

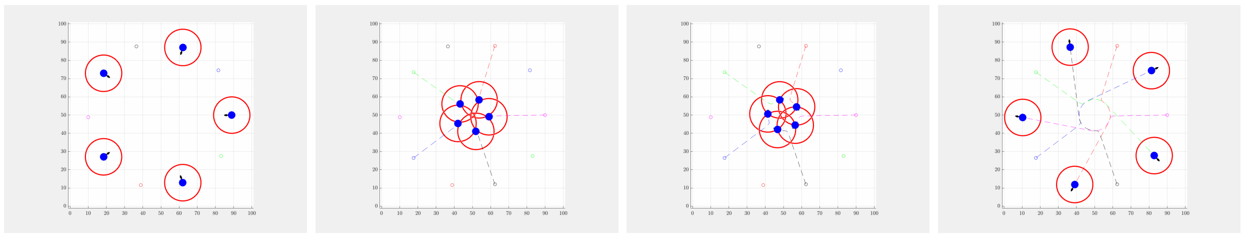

Figure 2: Trajectory Generated by Minimally Invasive Safety Critical Controller Solved Using CVX (Link to animated GIF: https://imgur.com/8w6Y2rM)

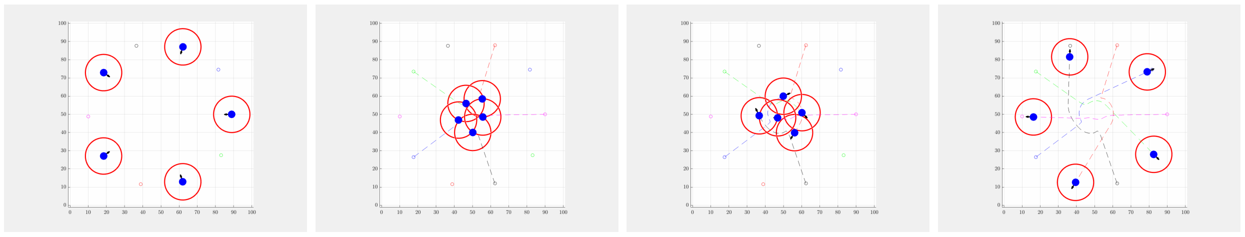

Figure 3: Trajectory Generated by Minimally Invasive Safety Critical Controller Solved Using Dykstra's Projection Algorithm (Link to animated GIF: https://imgur.com/a/qatkTPA)

The simulation consisted of 5 robots which were told to start in a ring formation and given the objective to move towards the opposite side of the ring. Since the most naive way to accomplish this task is to move straight across the field, each robot generates a "desired" path that crosses over the middle of the field and consequently intersects the other robot's paths. If this theoretical path is taken, all 5 robots would collide with each other at the middle, as depicted by the third frame of Figure 1.

Figure 2 shows the trajectory that the minimally invasive safety critical controller solved using CVX derived. The red circle around the robots indicate the bounds that they wish to keep around itself to ensure safety, thus depicts the region that the robot wants to prevent other robots from entering. As seen from the second and third frame, each robot slows down before approaching the bounds of the other robots, and proceeds to avoid collision by finding a detour path around them. This results in a synchronized clockwise rotation about the middle and lets the robots reach their goal without any collisions. This simulation was ran 5 times, and each iteration step took approximately 284.20 milliseconds to compute.

In contrast, Figure 3 shows the save minimally invasive safety critical controller, but this time solved using the proposed Dykstra's projection algorithm. Interestingly, it is observed that the generated trajectories are slightly different from the one computed by CVX, and shows that the paths are non-symmetric, yet still satisfies the constraints by ensuring that no robot breaches another robot's safety region at all times. Most importantly, this simulation was also ran 5 times, but each iteration step took approximately only 289.17 microseconds, which is an improvement of nearly 10000 times. This result shows that the proposed method is indeed practical and useful for this application, since it is able to achieve the same result 10000 times faster than the method relying on CVX.

6 Analysis and Conclusion

As seen from the two different trajectories generated by the CVX implementation and the Dykstra's projection algorithm implementation, it was interesting to see that the two algorithms generated slightly different paths compared to each other. This result is however expected due to Dykstra's algorithm working off of the assumption that the series of projections are generating a control input close enough to the actual solution of the problem. The fact that the generated control input is merely an approximated solution to the "real" solution makes it so that it is in general less optimal than the solution obtained from solving the entire optimization problem. The tradeoff for the inaccuracies is its computation speed, which in this case seems to be justified due to the Dykstra's projection algorithm implementation still achieving the goal while maintaining safety. For obvious reasons, this system can easily be scaled up to show that the CVX implementation quickly becomes impractical due to how much the complexity of the quadratic program increases as compared to a few additional projection computations for the Dykstra"s projection algorithm. This project clearly depicts the useful applications of the proximal gradient methods explored in class, albeit slightly modified to accommodate the large number of constraints imposed between each combination of robots.

7 References

1 Ames, Aaron D, and Paulo Tabuada. "Lectures on Nonlinear Dynamics and Control." 10 Mar. 2022.

2 Bauschke, Heinz H, and Adrian S Lewis. "Dykstras Algorithm with Bregman Projections: A Convergence Proof." Optimization, vol. 48, no. 4, 2000, pp. 409–427., https://doi.org/10.1080/02331930008844513.

3 Borrmann, Urs, et al. "Control Barrier Certificates for Safe Swarm Behavior." IFAC-PapersOnLine, vol. 48, no. 27, 2015, pp. 68–73., https://doi.org/10.1016/j.ifacol.2015.11.154.

4 Vandenberghe, Lieven. "7. Accelerated Proximal Gradient Methods." ECE236C (Spring 2022).

8 MATLAB Source Code

```
% main.m
% Runs the simulation
% Author - Hayato Kato

clear;
close all;
clc;

% Simulation Parameters
N = 5;
FieldWidth = 100;
FieldHeight = 100;
Distance = 10;

% Generate a ring of swarm robots trying to cross over to the other side
radius = 40;
```

```
a = 2*pi*[1:N]'/N;
x1 = [radius*cos(a)+FieldWidth/2,radius*sin(a)+FieldHeight/2];
a = a+pi*1.0;
x2 = [radius*cos(a)+FieldWidth/2,radius*sin(a)+FieldHeight/2];
swarm = Swarm(x1,x2,FieldWidth,FieldHeight,Distance);

% Define the figure used to plot the swarm trajectory
f = figure;
filename = 'swarmAnimation.gif';
frameinterval = 0.001;
time = 1;

swarm.plot();
count = 0;
totalElapsedTime = 0;
% Run the simulation until the swarm reaches its goal position
while max(vecnorm(swarm.u_des')) > 0.1
    count = count + 1;
    totalElapsedTime = totalElapsedTime + swarm.update_csf_cvx();
    disp(totalElapsedTime/count);
    swarm.plot();
    drawnow;

    % Save the plot as a gif animation
    frame = getframe(f);
    im = frame2im(frame);
    [image,cmap] = rgb2ind(im,256);
    if time == 1
        imwrite(image,cmap,filename,'gif','LoopCount',Inf,'DelayTime',frameinterval);
    else
        imwrite(image,cmap,filename,'gif','WriteMode','append','DelayTime',frameinterval);
    end
    time = time + 1;
end


% Swarm.m
% Swarm class definition, which handles all of the state updates using the
% two proposed algorithms and plotting
% Auther - Hayato Kato

classdef Swarm < handle
    properties
        fX
        fY
        distance

        N
        x
        x_0
        x_f
        u_des
        u_act
        x_history
```

```
end
properties (Constant)
    SPEED = 1;

    ALPHA = 0.2;
end

methods
    function obj = Swarm(startPos, endPos, fieldX, fieldY, dist)
        if nargin > 0
            obj.fX = fieldX;
            obj.fY = fieldY;
            obj.distance = dist;

            obj.x_0 = startPos;
            obj.x = startPos;
            obj.x_f = endPos;
            obj.N = length(startPos);
            diff = obj.x_f - obj.x;
            obj.u_des = min(Swarm.SPEED,norm(diff)) * normr(diff);
            obj.u_act = obj.u_des;
            obj.x_history = cat(3,[],obj.x);
        end
    end

    % Directly generates control input off of desired control input,
    % which results in a collision
    function elapsedTime = update_collide(Swarm)
        diff = Swarm.x_f - Swarm.x;
        Swarm.u_des = min(norm(diff),Swarm.SPEED)*normr(diff);
        Swarm.u_act = Swarm.u_des;
        Swarm.x = Swarm.x + Swarm.u_act;
        Swarm.x_history = cat(3,Swarm.x_history,Swarm.x);
        elapsedTime = 0;
    end

    % Minimally Invasive Safety Critical Control using CVX
    function elapsedTime = update_csf_cvx(Swarm)
        diff = Swarm.x_f - Swarm.x;
        Swarm.u_des = min(norm(diff),Swarm.SPEED)*normr(diff);
        u_des = reshape(Swarm.u_des',2*Swarm.N,1);
        tic;
        cvx_begin quiet
            variable u(2*Swarm.N,1);
            minimize 0.5*power(2,norm(u - u_des,2));
            subject to
               for i = 1:Swarm.N
                   for j = 1:Swarm.N
                       if i ~= j
                           Swarm.Lgh(i,j)*u >= -Swarm.ALPHA * Swarm.h(i,j);
                       end
                   end
               end
        cvx_end
    elapsedTime = toc;
    Swarm.u_act = reshape(u,2,Swarm.N)';
    Swarm.x = Swarm.x + Swarm.u_act;
    Swarm.x_history = cat(3,Swarm.x_history,Swarm.x);
end

% Minimally Invasive Safety Critical Control using Dykstra's
% Projection Algorithm
function elapsedTime = update_csf_Dykstra(Swarm)
    diff = Swarm.x_f - Swarm.x;
    Swarm.u_des = min(norm(diff),Swarm.SPEED)*normr(diff);
    u_des = reshape(Swarm.u_des',2*Swarm.N,1);
    tic;
    u = zeros(2*Swarm.N,1);
    t = 1;
    temp = u - t*(u-u_des);
    for i = 1:Swarm.N
       for j = 1:Swarm.N
           if i ~= j
               b = Swarm.ALPHA * Swarm.h(i,j);
               a = -Swarm.Lgh(i,j);
               % Projection iff the constraint is not met
               if a*temp > b
                   temp = temp + (b-a*temp)*a'/(a*a');
               end
           end
       end
    end
    u = temp;
    elapsedTime = toc;
    Swarm.u_act = reshape(u,2,Swarm.N)';
    Swarm.x = Swarm.x + Swarm.u_act;
    Swarm.x_history = cat(3,Swarm.x_history,Swarm.x);
end

function result = h(Swarm,i,j)
    temp = Swarm.x(i,:)-Swarm.x(j,:);
    result = temp*temp'-Swarm.distance^2;
end

function result = Lgh(Swarm,i,j)
    Lgh = zeros(1,2*Swarm.N);
    Lgh(2*i-1:2*i) =  2*(Swarm.x(i,:)-Swarm.x(j,:));
    Lgh(2*j-1:2*j) = -2*(Swarm.x(i,:)-Swarm.x(j,:));
    result = Lgh;
end

function plot(Swarm)
    clf;
    FieldWidth = Swarm.fX;
    FieldHeight = Swarm.fY;

    hold on;
    swarmColor = ['r','g','b','k','m'];
```

```
            for i=1:Swarm.N
                arrow = 2*Swarm.u_act(i,:);
                plot([Swarm.x_0(i,1),Swarm.x_f(i,1)], ...
                     [Swarm.x_0(i,2),Swarm.x_f(i,2)], ...
                     'o','Color',swarmColor(i));
                quiver(Swarm.x(i,1),Swarm.x(i,2),arrow(1),arrow(2),2, ...
                    'filled','MaxHeadSize',5,'Color','k','LineWidth',2);

                viscircles(Swarm.x(i,:),Swarm.distance);

                plot(reshape(Swarm.x_history(i,1,:),[],1), ...
                     reshape(Swarm.x_history(i,2,:),[],1), ...
                     '--','Color',swarmColor(i));
            end

            scatter(Swarm.x(:,1),Swarm.x(:,2),200,'b','filled');

            axis equal;
            xlim([-1,FieldWidth+1]);
            ylim([-1,FieldHeight+1]);
            grid on;
        end
    end
end
```