ENPM667

Project - I

REPORT

On

Planning and Control of Ensembles of Robots with Non-holonomic Constraints

Aashrita Chemakura (119398135) achemaku@umd.edu
Saketh Narayan Banagiri (118548814) sbngr@umd.edu

University of Maryland, College Park

CONTENTS

LIST OF FIGURES

ABSTRACT

Utilizing swarm robots is one of the latest strategies for creating intelligent decision-making systems. Due to their desirable collective behaviour interacting with the environment and other robots in addressing various problems based on inputs, these multi-robot systems are emerging as more efficient systems in the fields of artificial swarm intelligence.

In this study, we focus on the creation of distributed formation control laws, which enable the control of individual mobile ground robots in a formation to a desired distribution with a minimum understanding of the overall state.

The robot swarm is managed by individual team members, requiring only a basic understanding of the ensemble state. Furthermore, the robot swarm is controlled regardless of its size, making it immune to failures in any one of its members. The article goes into detail about ensemble motion planning and avoiding robot-to-robot collisions. To enable the desired distribution of a specific number of robots, a decentralized control law that is safe from robot-to-robot collisions has been derived.

Python is used to program the differential drive robot's algorithms and output.

INTRODUCTION

As the development of pervasive embedded computing, sensing, and wireless communication enables the application of multiagent systems to hard tasks such as environmental monitoring, surveillance and reconnaissance for security and defence, and support for first responders in a search and rescue operation, effective tactics for controlling large teams of robots in complicated situations are becoming increasingly significant. Such circumstances call for the use of control algorithms that enable robots to adapt to various settings and carry out difficult jobs without colliding. Additionally, robot controllers must be strong enough to accommodate robot failures or adjustments to the team size.

There are various methods for managing large robot teams such as formation of a geometric rigid virtual structure, control of these structures using formation graphs, or using standard techniques like input-output linearization, leader-follower architectures, etc.

The control of the position and orientation of a formation of mobile robots as well as the shape adaptation to the surroundings are the main topics of this article. Planning the team's structure and trajectory and creating efficient coordination mechanisms to divide the team into subgroups and combine two subgroups are two related issues that are also taken into consideration. We see these issues and their solutions as components that can help a robot team explore an environment while adjusting to the limitations imposed by environmental impediments.

In the context of our work, we place a strong emphasis on team robot control and decouple the difficulties of taking the estimation problem into account. Furthermore, we think that some degree of centralization is necessary to control a big group of robots.

BACKGROUND

Consider the difficulty of controlling a large number of robots, say 100 planar robots that must move as a team from one part of space to another. The most basic approach uses reference trajectories and control laws to keep each robot on the intended path. While this is certainly feasible, it is computationally hard. As the number of robots grows, a certain amount of abstraction becomes desirable. The motion generation/control problem should be handled in a lower-dimensional space that encapsulates the group's behavior and the nature of the activity. One approach could be to require the robots to adhere to one or more rigid virtual structures.

[1] tries to create a formal abstraction for a team of robots that may be used to govern the team's position, orientation, and shape. The abstraction is based on the formulation of a map from the robots' configuration space Q to a lower dimensional manifold A, independent of the robots' number and ordering. We need that the manifold have a product structure $A = G \times S$ where G is a Lie group that captures the ensemble's reliance consider kinematic robots in the plane (see [11] for a treatment on the selected world coordinate frame and S is a shape manifold that describes the team intrinsically.
Furthermore, we require that the shape variables $s \in S$ and the group variables $g \in G$ be controlled separately, so that the user can simply command the independent variables. The user, for example, can change the shape of the formation without changing the group trajectory, and vice versa.

The two key benefits of this abstract representation are (a) that it lends itself to planning in a lower dimensional space and (b) that its dimension is independent of the number of robots in the team.

The state space of the N-robot system is constructed by creating N copies of Q_i, the state space of the i^{th} robot:

$$Q = Q_1 \times Q_2 \times . . . \times Q_N.$$

A smooth, differentiable map defines the abstract space, M, whose dimension is lower and independent of the dimension of Q.

$$\Phi: Q \rightarrow M, \Phi(q) = x, \tag{1}$$

Where Φ is a mapping between higher-dimensional state $q \in Q$ and the lower-dimensional abstract state $x \in M$.

In this, we consider kinematic robots in a plane where q represents the collection of robot positions. Thus, q is given by:

$$q = [q_1, . . ., q_i, . . ., q_N]^T$$

where $qi \in R^2$

In addition, M is considered to have a product structure of the form

$$M = G \times S;\ x = (g, s),\ \Phi = (\Phi_g, \Phi_s)$$

Characterizing the distribution of robots around the mean position is used to model the shape. The group's centroid is given by:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} q_i$$

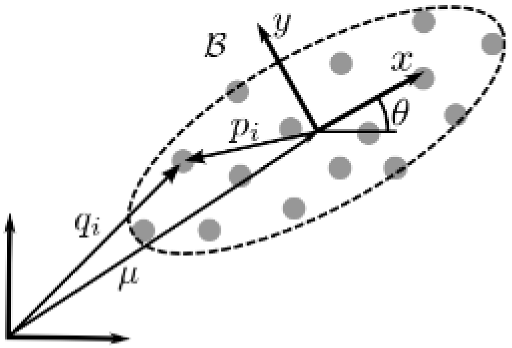


Figure 1: *The frame B fixed to the group of robots moves with respect to the inertial frame. [2]*

As shown in Fig. 1, we may build a local frame, B, whose origin is at the centroid, by requiring the orientation to be such that the coordinates of the robots in this frame, $p_i = [x_i, y_i]$, satisfy

$$\sum_{i=1}^{N} x_i y_i = 0$$

The inertia tensor (assuming uniform unit mass) or a matrix of second moments can be used to approximate the distribution of robots in this local frame:

$$\tau = \sum_{i=1}^{N} p_i {p_i}^T = \begin{bmatrix} \tau_{11} & 0 \\ 0 & \tau_{22} \end{bmatrix}$$

We define two shape variables proportional to the diagonal elements:

$$s_1 = \kappa\tau_{11},\ s_2 = \kappa\tau_{22},$$

where $\kappa \neq 0$. Choosing $\kappa = \frac{1}{N-1}$ gives the shape variables a geometric interpretation. They are the semi-major and semi-minor axes of a concentration ellipse for a group of robots whose coordinates in the plane satisfy a normal distribution.

The abstract description of the team of robots, x, is given by the position and orientation of the team, g, and the shape s. We take g to be the position and orientation of $\mathcal{B}$:

$$\begin{bmatrix} cos\theta & -sin\theta & \mu 1 \\ sin\theta & cos\theta & \mu 2 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\mu = (\mu_1, \mu_2)$ are the components of the centroid in the inertial frame and the shape $s = (s_1, s_2)$. The map ϕ defined in this way can be easily shown to be a submersion.

The abstract space, M, is naturally decomposed into a shape space, S, and a Lie group, G. Since ϕ is a submersion, it follows that there is a unique $\dot{x}$ for every $\dot{q}$ but not the other way around.

The optimal velocity $u*$ at any point q $\in Q$ for a desired $\dot{x}$ at the corresponding point $x = \phi(q) \in M$ for the system can be found by considering the time derivative of the transformation described by (1),

$$d\emptyset\dot{q} = x \tag{2}$$

From (1), the definitions of (μ, θ, s1, s2), and algebraic simplification, the transformation dϕ becomes

$$d\phi = \kappa \begin{bmatrix} \frac{1}{\kappa N} I_2 & \dots & \frac{1}{\kappa N} I_2 \\ \frac{q_1-\mu}{s_1-s_2} H_3 & \dots & \frac{q_N-\mu}{s_1-s_2} H_3 \\ (q_1-\mu)^T H_1 & \dots & (q_N-\mu)^T H_1 \\ (q_1-\mu)^T H_2 & \dots & (q_N-\mu)^T H_2 \end{bmatrix} \tag{3}$$

Here H_1, H_2, and H_3 are defined by $H_1 = I_2 + R^2E_2$, $H_2 = I_2 - R^2E_2$, $H_3 = R^2E_1$, where I_2 is the 2×2 identity matrix and

$$E_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \qquad E_2 = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

The minimum energy solution has been obtained in [1] as follows using Moore-Penrose Inverse:

$$u* = d\emptyset^T \, (d\emptyset d\emptyset^T)^{-1}\dot{x} \tag{4}$$

Further algebraic simplification of (4) using (3) results in the control law for each individual agent, $u_i = \dot{q}_i$

$$u^* = \mu + \frac{s_1 - s_2}{s_1+s_2} H_3 \, (q_i - \mu)\dot{\theta} + \frac{1}{4s_1} H_1 \, (q_i - \mu)\dot{s_1} + \frac{1}{4s_2} H_2 \, (q_i - \mu)\dot{s_2} \tag{5}$$

In the next section, we will pursue a slightly different formulation by writing these equations in the moving frame Ɓ for the formulation of minimum-norm control inputs, such as (5).

PROBLEM FORMULATION

A. *Dynamic in a moving frame*

At any point $x = (g, s) \in M$ in the abstract space, the derivative can be written as:

$$\dot{x} = \begin{bmatrix} \dot{g} \\ \dot{s} \end{bmatrix} = \begin{bmatrix} g & 0 \\ 0 & I_2 \end{bmatrix} \begin{bmatrix} \xi \\ \sigma \end{bmatrix} \tag{6}$$

$\dot{x} = (\dot{g}, \dot{s})$ is the time derivative of the abstract space in the inertial frame while $\zeta = (\xi, \sigma)$ is the time derivative in the moving frame B, and

$$\Gamma = \begin{bmatrix} g & 0 \\ 0 & I_2 \end{bmatrix}$$

is a non-singular 5×5 transformation matrix. If v_i is the robot velocity in the frame $\mathcal{B}$ so that $u_i = Rv_i$,

$$\kappa \begin{bmatrix} \frac{I_2}{\kappa N} & \cdots & \frac{I_2}{\kappa N} \\ \frac{1}{s_1 - s_2} {p_1}^T E_1 & \cdots & \frac{1}{s_1 - s_2} {p_N}^T E_1 \\ {p_1}^T (I_2 + E_2) & \cdots & {p_N}^T (I_2 + E_2) \\ {p_1}^T (I_2 - E_2) & \cdots & {p_N}^T (I_2 - E_2) \end{bmatrix} \begin{bmatrix} \nu_1 \\ \cdot \\ \cdot \\ \cdot \\ \nu_N \end{bmatrix} = \begin{bmatrix} \xi \\ \sigma \end{bmatrix}$$

Thus, the minimum-energy solution (5) can be written as:

$$v_i^* = \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} + \frac{s_1 - s_2}{s_1 + s_2} E_1 p_i \xi_3 \tag{7}$$

with the simplification:

$$v_i{}^* = \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} + \frac{s_1 - s_2}{s_1 + s_2} E_1 p_i \xi_3 + \frac{1}{4 s_1} (I_2 + E_2) p_i \sigma_1 + \frac{1}{4 s_2} (I_2 - E_2) p_i \sigma_2 \tag{8}$$

The control law defined by (8) does not consider inter-agent collisions or the spatial size of individual robots.

B. Collision Avoidance

The separation distance between the reference points on robots *i* and *j* is:

$$\delta_{ij} = \|p_i - p_j\|$$

To avoid collisions between robots, we define a safe separation distance between two robots:

$$\epsilon = 2\rho + \epsilon_s \tag{9}$$

Where ρ is the radius of each robot and ϵ_s is a specified safety region

We define the neighbourhood $\mathcal{N}_i$ as the set of all robots sensed by or communicating with robot *i* such that *i* is able to gain knowledge of its neighbours' positions and velocities, $\{p_j, v_j\}$, $\forall\ j \in \mathcal{N}_i$. To ensure that the robots do not collide, we require that

$$(p_i - p_j) \cdot (v_i - v_j) \geq 0 \tag{10}$$

for all j ∈ $\mathcal{N}_i$ such that $\delta_{ij} \leq \epsilon$

C. Asymptotic convergence to a desired abstract state

The easiest way to guarantee convergence, in absence of collision to a time-invariant abstract state x^{des} is to require the error $\tilde{x} = (x^{des} - x)$ to converge exponentially to zero:

$$\dot{x} = K\tilde{x}$$

or equivalently,

$$\zeta = \Gamma^{-1}K\tilde{x} \tag{11}$$

where K is any positive-definite matrix, and use (7, 8) to obtain robot velocities that guarantee globally asymptotic convergence to any abstract state.
In the next chapter we look at a control law that guarantees convergence to an abstract state satisfying certain conditions, while guaranteeing safety (i.e., there are no inter-agent collisions).

CONTROL WITH COLLISION AVOIDANCE

A. Monotonic convergence

To accommodate the safety constraints in (10), we replace the exponential convergence with a slightly different notion of convergence. For this, we find the solution closest to the minimum energy solution satisfying the safety constraints instead of a rigid solution as in (7).

We relax the requirement of exponential convergence to an abstract state and replace it with a slightly different notion of convergence in order to accommodate the safety constraints in (10). Specifically, instead of insisting on the minimum-energy solution, (7), we find the solution closest to the minimum energy solution satisfying the safety constraints.

First, we require that the error in the abstract state decrease monotonically:

$$\tilde{x}^T K \dot{x} \geq 0 \tag{12}$$

Substituting (2) into (12) gives:

$$\tilde{x}^T K \Gamma \begin{bmatrix} I_2 & \cdots & I_2 \\ \frac{1}{s_1 - s_2} {p_1}^T E_1 & \cdots & \frac{1}{s_1 - s_2} {p_N}^T E_1 \\ {p_1}^T (I_2 + E_2) & \cdots & {p_N}^T (I_2 + E_2) \\ {p_1}^T (I_2 - E_2) & \cdots & {p_N}^T (I_2 - E_2) \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \\ \cdots \\ \nu_N \end{bmatrix} \geq 0 \tag{13}$$

A sufficient coefficient to satisfy the above monotonic convergence condition in (13) is that each robot select inputs that satisfy:

$$\tilde{x}^T K \Gamma \begin{bmatrix} I_2 \\ \frac{1}{s_1 - s_2} {p_i}^T E_1 \\ {p_i}^T (I_2 + E_2) \\ {p_i}^T (I_2 - E_2) \end{bmatrix} \nu_i \geq 0 \tag{14}$$

If all robots choose controls satisfying (14), the error in the abstract state will decrease monotonically. It is essential to show that the minimum-energy control law (8) satisfies this inequality.

Proposition 1. The minimum-energy control law (8) with ζ given by (11) satisfies the monotonic convergence condition (14).

Proof: Let define g_i and m_i be such that

$$m_i = \left[I_2, \frac{s_1 - s_2}{s_1 + s_2} E_1 p_i, \frac{1}{4s_1}(I_2 + E_2)p_i, \frac{1}{4s_2}(I_2 - E_2)pi\right]$$

$$g_i = \left[I_2, \frac{1}{s_1 - s_2} p_i{}^T E_1, p_i{}^T(I_2 + E_2), p_i{}^T(I_2 - E_2)\right]^T$$

Substituting (8) and (11) into the left hand side of (14) gives the quadratic form:

$$\tilde{x}^T K\Gamma \,[g_i \quad m_i]\, \Gamma^{-1} K\tilde{x}$$

The 5×5 matrix $[g_i, m_i]$, although asymmetric, can be shown to be positive semi-definite with the two non-zero eigenvalues to be given by:

$$\lambda_1 = 1 + \frac{\|p_i\|^2}{s_1 + s_2} \qquad \text{and} \qquad \lambda_2 = 1 + \frac{p_{i,x}^2}{s_1} + \frac{p_{i,y}^2}{s_2}$$

Since K is chosen to be positive definite the inequality (14) is satisfied.

B. <u>A safe minimum-energy control law</u>

As discussed earlier, we need to derive a decentralized control law that selects a control input as close as possible to the minimum energy controls while satisfying the monotonic convergence inequality and the safety constraints.

Proposition 2. Equation (15) is a decentralized control law that selects a unique control input that has the smallest energy instantaneously while satisfying the monotonic convergence inequality and the safety constraints.

$$v_i = \underset{\hat{v}_i \in U}{arg\ min} \|v_i{}^* - \hat{v}_i\|^2, \qquad s.t.$$

Proof:

The safety guarantees and monotonic convergence condition are provided by the constraints in (10, 14). The function being minimized is the difference between the minimum-energy input and the maximum-energy input. Because the inequality constraints in vi are linear and the function to be minimized is a positive-definite, quadratic function of vi, (15) is a convex, quadratic problem with a unique solution. Furthermore, it is a decentralized control law because each robot relies solely on its own state and knowledge of the inaccuracy in the abstract state.

Convergence properties of (15)

We introduce the Lyapunov function to investigate the global convergence properties

$$V(q) = \frac{1}{2}\tilde{x}^T\tilde{x}$$

Since the solution of (15) must satisfy the inequality (14), we know that $\tilde{x}^T K\dot{x} \geq 0$. If K is chosen to be diagonal with positive entries, this condition also implies $\tilde{x}^T\dot{x} \geq 0$. In other words,

$$V(q) = -\tilde{x}^T\dot{x} \leq 0$$

We can deduce from [1] that q is bounded if x is bounded and *V(q)* is defined as $V(q) \to \infty \; as \; \|q\| \to \infty$. *V(q)* is also globally uniformly asymptotically stable. As a result, we can deduce from LaSalle's invariance principle that the abstract state will converge to the largest invariant set given by $\tilde{x}^T\dot{x} = 0$. We can deduce from (2) that $\dot{x} = 0$ only when $\upsilon = 0$. Thus, the invariant set is defined as the set of conditions that cause the system of inequalities given by (10, 14) to have a solution of $\upsilon = 0$.

Proposition 3. For any desired change in the abstract state $\tilde{x}$, subject to the condition $\tilde{x}_4 \geq 0, \tilde{x}_5 \geq 0$, (i.e., a condition where the size of the shape of the formation is not decreasing), there is a non-zero solution to the inequalities (10, 14).

Proof: Consider the solution given by the minimum energy control law (8). In component form

$$v_i^* = \begin{bmatrix} \xi_1 + \dfrac{s_1 - s_2}{s_1 + s_2} y_i \xi_3 + \dfrac{x_i}{4s_i}\sigma_1 \\ \xi_2 + \dfrac{s_1 - s_2}{s_1 + s_2} x_i \xi_3 + \dfrac{y_i}{4s_2}\sigma_2 \end{bmatrix}$$

It is easy to see that this satisfies the collision constraints (10) for every pair of robots (i, j):

$$\left[(x_i - x_j)\ (y_i - y_j)\right] \begin{bmatrix} v_{i,x}^* - v_{j,x}^* \\ v_{i,y}^* - v_{j,y}^* \end{bmatrix} \geq 0$$

Remark 1

It is clear from the above proof that there are no guarantees when the shape in the abstract state is shrinking in area. If $\tilde{x}_4 < 0, \tilde{x}_5 < 0$, there may not be a non-zero velocity vector that satisfies the inequalities (10, 14). It is only in this condition that the system will reach an equilibrium away from the desired abstract state.

In the next chapter we discuss an energy metric for motion planning of a deformable ellipse. Such a metric permits the computation of optimal motion plans in complex environments.

MOTION PLANNING IN THE ABSTRACT SPACE

The abstract representation of the robot team enables motion planning that only requires consideration of an abstract state space of fixed size rather than one that scales with the number of robots. We will look at the problem of constructing reference paths in the abstract space in this part.
To begin, we define a metric on M. A Riemannian metric can be defined on the abstract space as a bi-linear form produced by an inner product.
Given two twists $\{\xi_1, \xi_2\}$ we can define the inner product using [3] as follows:

$$< \xi_1, \xi_2 >= \xi_1^T W \xi_2$$

where W is a definite positive matrix Left translation yields the inner product of two velocities or tangent vectors $\dot{g}_{1,}\ \dot{g}_2$ at any arbitrary element:

$$< \dot{g}_{1,} \dot{g}_2 >_g =< g^{-1}\dot{g}_{1,} g^{-1}\dot{g}_{2,} >_e$$

where $g^{-1}\dot{g}_{i,}$ are tangent vectors at the identity element e (the 33 homogeneous transformation). A left-invariant Riemannian metric is defined in this way. Following [4,] we may utilize a rigid body's inertia tensor and kinetic energy to define $W_g$in the body-fixed coordinate system B.

$$W_g = \begin{bmatrix} mI_2 & 0 \\ 0 & \tau_{11} + \tau_{22} \end{bmatrix}$$

The method described above was for a rigid form. Because $M = G \times S$ is a product space, we treat the form space separately. To simulate the "cost" of changing the shape, we assume a constant metric $W_s = \alpha I_2$ As a result, the rate of change of the abstract shape in B given ζ by has the following norm, which is well-defined everywhere on M.

$$||\zeta|| = \frac{1}{2}\zeta^T \begin{bmatrix} W_g & 0 \\ 0 & W_s \end{bmatrix} \zeta \qquad (16)$$

Realistically, the potential energy involved with deforming the shape must also be modelled. To begin developing an abstract model for potential energy storage, consider the expansion or contraction as a reversible, adiabatic process in which no energy is lost. Internal energy is increased during compression, which can then be recovered during expansion. It is commonly understood that in such processes, the pressure p and volume v are connected by the specific heat γ ratio by the equation:

$$pv^\gamma = constant$$

Thus, the effort done to shift the volume from $v_1\ to\ v_2$, resulting in an increase in internal energy, is given by:

$$\Delta V = k\left(\frac{1}{v_2^{\gamma-1}} - \frac{1}{v_1^{\gamma-1}}\right)$$

where k is a constant, We can use the area of the ellipse (with a unit depth) in the plane instead of volume, which we know to be $\pi\sqrt{s_1 s_2}$. A reference shape $s^0(N)$is defined as a circular form for N robots with zero potential energy. The radius of a zero-energy circular shape, $r_0(N)$, must logically increase with N. In this study, we assume that, $r_0(N) = \frac{N_\epsilon}{2}$. The potential energy associated with any shape $s \in\ S$ can be calculated as follows:

$$V(s) = \beta\left(\frac{1}{(s_1 s_2)^{\frac{\gamma-1}{2}}} - \frac{1}{(r_0^2(N))^{\gamma-1}}\right) \tag{17}$$

where β is a constant. Thus, the total energy associated with any motion in any arrangement can be calculated as follows:

$$E(g,s,\zeta) = \frac{1}{2}\zeta^T\begin{bmatrix} W_g & 0 \\ 0 & W_s \end{bmatrix}\zeta + V(s) \quad ` \tag{18}$$

Equation (18) provides a logical technique to calculating the cost of configuration changes using kinetic and potential energy with two constants α and β. It also enables us to describe trajectory generation and motion planning problems as geodesic search problems.

The space can be divided and the trajectories can be evaluated for:

$$\mu \in \mathbb{R}^2 \text{and} (\theta, s_1, s_2) \in \mathbb{R}^2 \times SO(2)$$

because of the product nature of the metric independently in open environments (18). Instead, we propose designing motion plans by discretizing the abstract space with obstacles and admissible abstract states as constraints. There are numerous discrete optimal planning algorithms that allow the use of the energy measure provided by (18) to find the shortest energy path through open or crowded situations [5].

SPLITTING AND MERGING OF GROUPS OF ROBOTS

When a group of robots is pushed to squeeze through confined places, the only feasible geometries have small areas and huge values of internal energy *V(s)*, as seen in Figure 1. (17). As a result, we design a threshold such that when *V(s)* > V_{max}, the group divides into two subgroups, each with half the number of robots and identical shapes to the original. This allows the group to reduce the number of limitations while remaining in the same configuration to reset its energy to a lower level. A supervisory agent, on the other hand, can decide when to divide the robots into subgroups. Sect. IV's formulation can be applied to many subgroups using the new abstraction manifold $M = M_1 \times M_2, where\ M_i = G_i \times S_i$. The only additional thing needed is a procedure for each robot to decide which subgroup it belongs to. Clearly, each subgroup's motion plan, $x_i^{des}(t)$, can be constructed, and the desired trajectory may be broadcast to the group. x_i received a response from the abstract state. With knowledge of the unique subgroup *i* to which it belongs, each robot may compute its own commands. We offer two alternative event-triggered strategies for decentralized and distributed robot team splitting and merging.

A. *Market-based auctioning method*

Our first approach is based on the market-based auctioning strategy presented in [19], which guarantees polynomial time convergence but necessitates communication between the robots. It enables a team of robots to separate into subgroups by creating an auction based on the desired abstract subgroup states $\{x_1^{des}, \ldots., x_k^{des}\}$,and the maximum number of agents allowed in each subgroup $\{n_{1,} \ldots., n_k\}$ where *k* is the desired subgroup count. The auction results in all agents being assigned to a subgroup and being dispersed in accordance with the maximum number of agents in each group. The algorithm requires $N = \sum_{i=1}^{k} n_i$. to ensure accuracy.

B. *Stochastic policy for splitting*

As the number of robots increases, it is advantageous to utilize a mean-field model to simulate robot distribution and design stochastic switching rules that provide the appropriate ensemble features [20]. The ensemble properties of the group of robots employing stochastic switching rules converge to the required qualities as $N \to \infty$. In other words, if each robot chooses one subgroup over another using a probability distribution, the ensemble attributes of the group can be derived from this probability distribution. In practice, a split between *k* subgroups with $\{n_{1,} \ldots., n_k\}$ robots can be approximated by each robot selecting group I with probability $p_i = \frac{n_i}{N}$.

C. *Merging of groups*

In the proposed system, employing the controller, combining disparate groups is straightforward (15). The redefinition of $\tilde{x}$ to account for the desired merged abstract state yields a single group while taking inter-agent collision avoidance into consideration.

RESULTS

A. Expected sample results

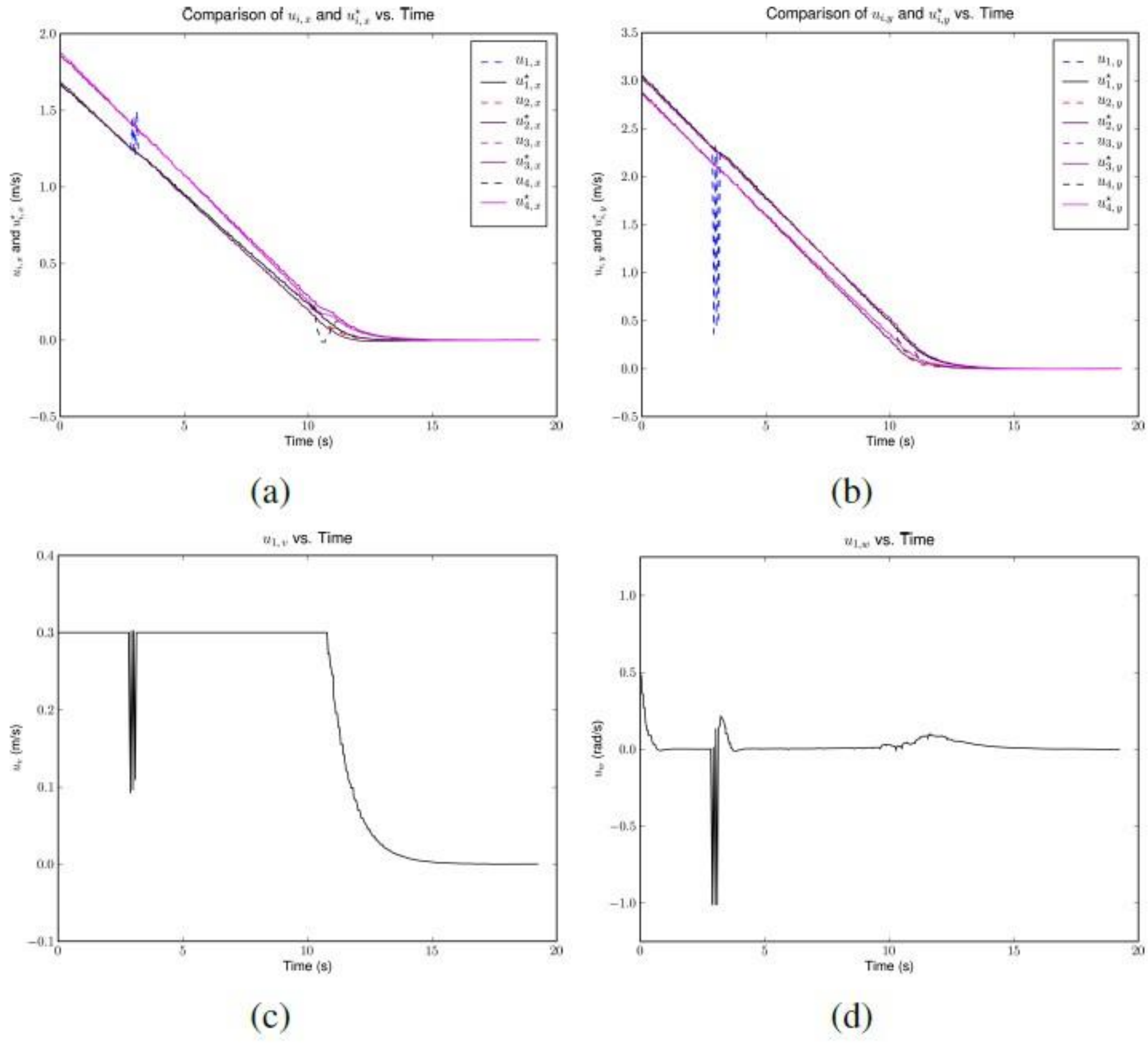


Figure 2: The optimal control, u_i^* , and u_i from (15) defined in the robot's local frame (Figures. 2(a)–2(b)) resulting from (15). In general, the optimal control law is the solution to (15). However, during inter-agent interactions the resulting control varies from the optimal solution. The linear and angular velocities resulting from feedback linearization (Figures. 2(c)–2(d)). The above plots are for a system of 4 robots. [2]

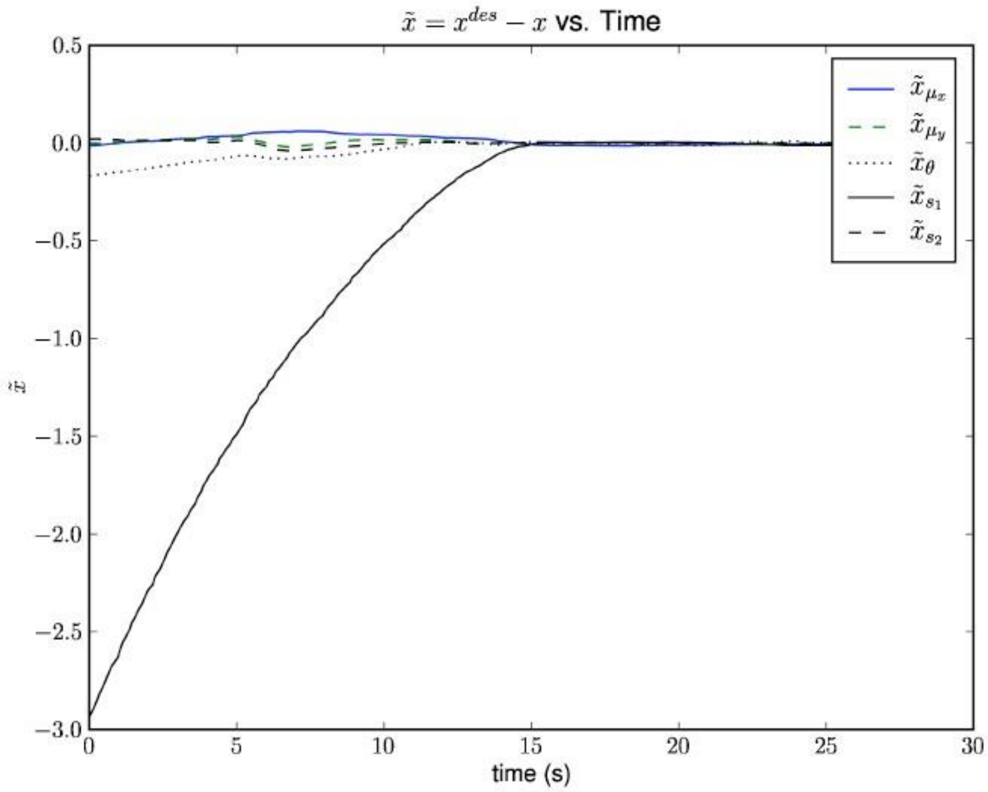
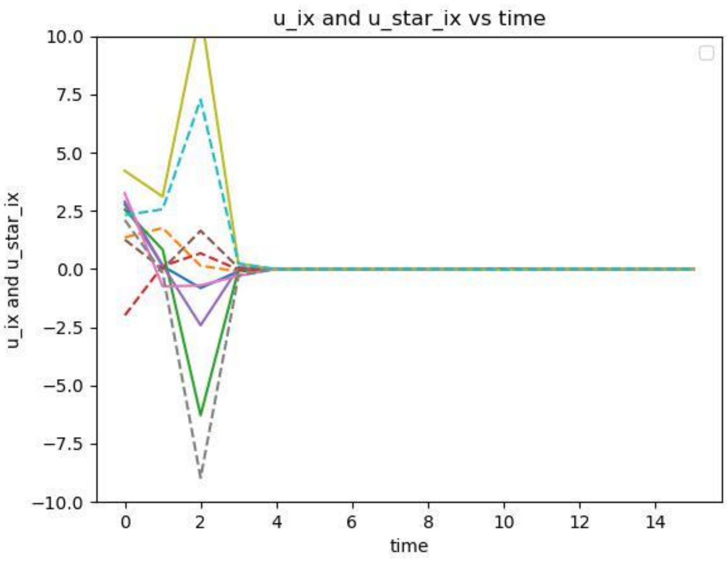


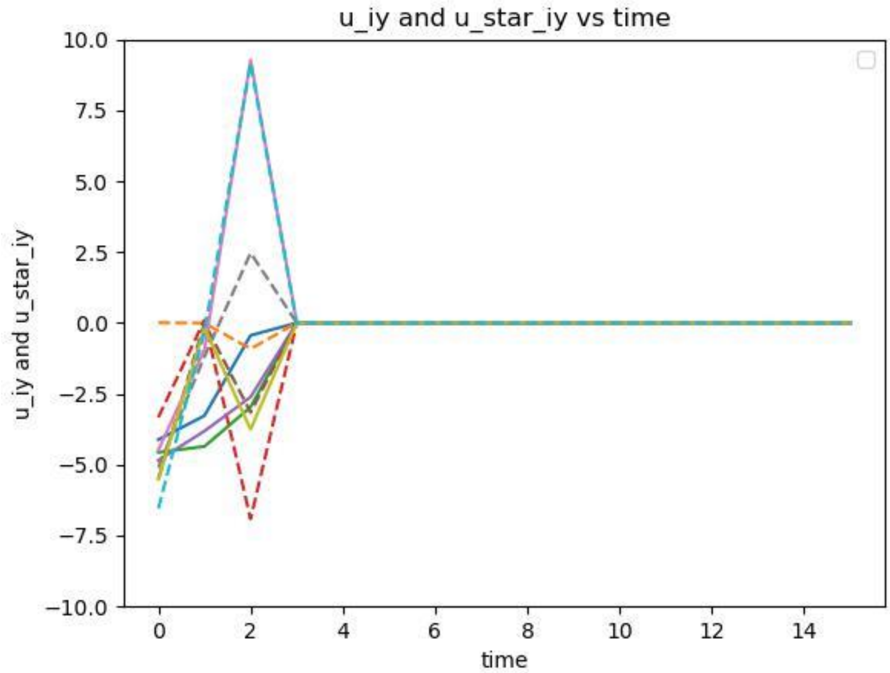
Figure 3: The convergence of the team of seven robots in experimentation to x^{des} or desired state. The trial represents a merging scenario where the robots were distributed in distinct groups separated by several meters. This result is for a system of 7 robots. [2]

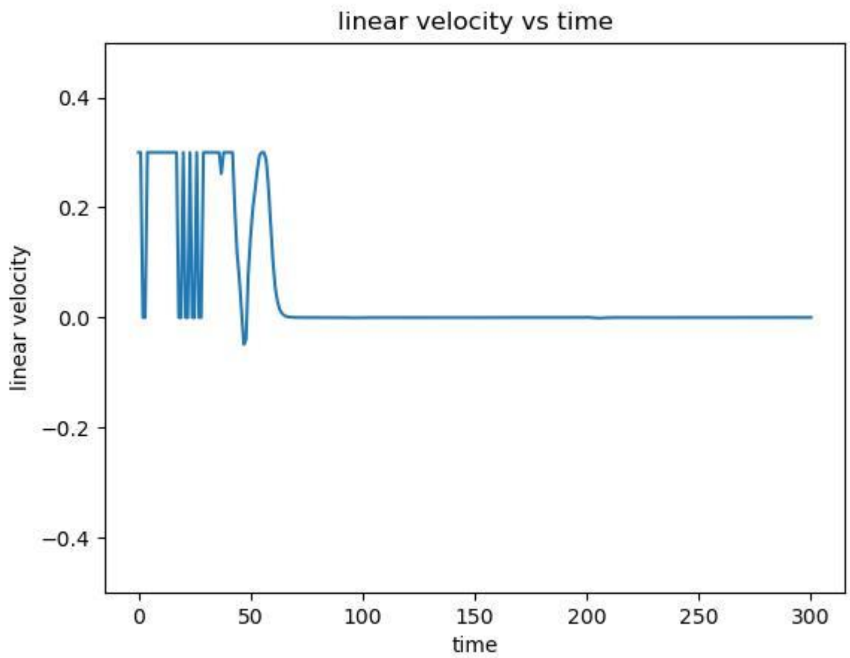
B. Obtained results

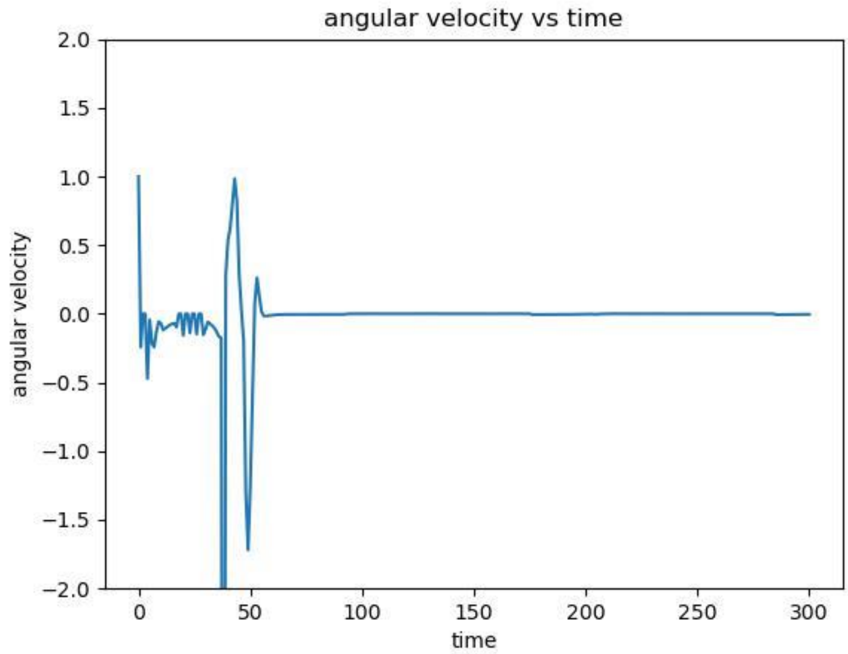
The python code was evaluated for a system of 5 robots by taking desired parameters into consideration. The following are the results obtained.



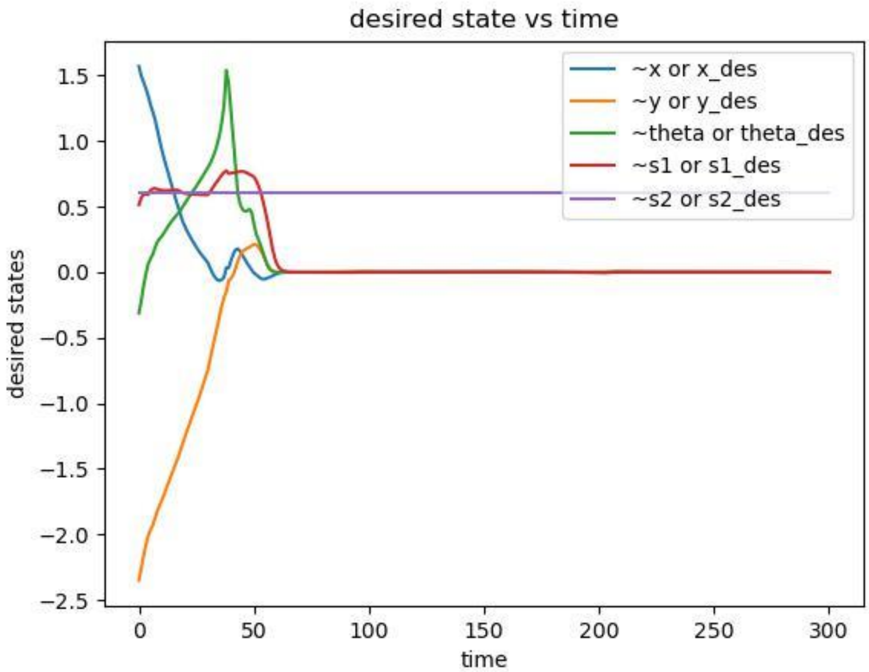
4 (a)


u_iy and u_star_iy vs time
u_iy and u_star_iy
time
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
2
4
6
8
10
12
14
4 (b)


linear velocity vs time
linear velocity
time
0.4
0.2
0.0
−0.2
−0.4
0
50
100
150
200
250
300
4 (c)



4 (d)



4(e)

Figure 4: 4(a)-4(b) *Obtained optimal control,* u_i^**, and* u_i *from (15) defined in the robot's local frame.*

4(c)-4(d) *The linear and angular velocities resulting from feedback linearization.*

4(e) *Convergence to desired state*

CHALLENGES FACED

Although we were able to obtain all the required parameters and derive all the desired graphs, we were not able to achieve the desired state for the problem for every configuration of the swarms. The reason for this being that we have considered the swarm bots to be initialized at random positions and the algorithm we designed wasn't that robust to achieve the required state in every case. Nonetheless, the results obtained in the study serve as a good estimation for the results obtained in the original study.

CONCLUSION AND FUTURE SCOPE

The challenge of organizing and managing the location, orientation, and shape of a team of robots was addressed. In contrast to the majority of earlier research, we model the actual shape of the robot and take into account controllers that are assured to prevent robot collisions. We also provide a metric for the design of the ensemble's deformable shapes and trajectories as well as for the creation of efficient coordination techniques for dividing the team into subgroups and merging subgroups. We see these issues and their solutions as the building blocks that can let a robot team explore an environment while adjusting to the limitations imposed by environmental impediments. The efficiency of the control algorithm when used with nonholonomic robots is demonstrated by simulation.

CODE

The code is divided into three sub parts. They are as follows.

A. Base.py:

```
import numpy as np
import math
from math import *


iters = 15000 #iterations

bots_count = 5 #number of bots

pos_mean = 8.5 #mean

sd = 1 #standard deviation

del_t = 0.01 #time difference

# gain matrices
KU = np.array([[2, 0], [0, 2]])
KS1 = 2
KS2 = 2
KT = 2

# formation variables
E1 = np.array([[0.0, 1.0], [1.0, 0.0]])
E2 = np.array([[1.0, 0.0], [0.0, -1.0]])


# robot specifications
rad = 0.15
axle_len = 0.1
safe_dist = 0.1
max_lin_vel = 0.3
max_ang_vel = 1

# separation distance
seperation = 2*(rad + axle_len) + safe_dist

# calculate concentration ellipse
prob = 0.99
concentrated_ellipse = -2*math.log(1-prob)

# desired state space
# desired centroid
u_des = np.array([[10], [6]], dtype=np.float32)

# desired orientation
theta_des = 0.0
```

```python
# desired semi major axis of the ellipse
s1_des = 0.8

# desired semi minor axis of the ellipse
s2_des = 0.6

# desired state space
des_abstract_state = [[u_des, theta_des, s1_des, s2_des]]

# Gain matrix K
gain_matrix = np.vstack((np.array([1, 0, 0, 0, 0]),
                    np.array([0, 1, 0, 0, 0]),
                    np.array([0, 0, 0.8, 0, 0]),
                    np.array([0, 0, 0, 0.8, 0]),
                    np.array([0, 0, 0, 0, 0.8])))

I = np.eye(2)


class abs_space:

   def __init__(self):

       self.u_curr = np.zeros(shape=[2, 1], dtype=np.float32)
       self.theta_curr = 0.0
       self.s1_curr = 0.0
       self.s2_curr = 0.0

   def formation_variables(self, theta):

       R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta),
np.cos(theta)]])
       H1 = np.eye(2) + np.matmul(np.matmul(R, R), E2)
       H2 = np.eye(2) - np.matmul(np.matmul(R, R), E2)
       H3 = np.matmul(np.matmul(R, R), E1)

       return R, H1, H2, H3


   def get_centroid(self, swarm_bots=None):

       if swarm_bots is None:
           return self.u_curr
       bot_centroids = 0.0
       for bot in swarm_bots:
           bot_centroids = bot_centroids + bot.q.transpose()


       return bot_centroids / len(swarm_bots)



   def parameters(self, swarm_bots):
       abstract_space_x = 0.0
       abstract_space_y = 0.0
       abstract_space_s1 = 0.0
```

```
        abstract_space_s2 = 0.0

        if swarm_bots is None:
            return self.theta_curr, self.s1_curr, self.s2_curr

        self.u_curr = self.get_centroid(swarm_bots)
        for bot in swarm_bots:
            bot_pos = bot.q.transpose()
            abstract_space_y = abstract_space_y +
np.matmul(np.matmul((bot_pos - self.u_curr).transpose(), E1),
                                 (bot.q.transpose() -
self.u_curr)).item()
            abstract_space_x =  abstract_space_x +
np.matmul(np.matmul((bot_pos - self.u_curr).transpose(), E2),
                                 (bot.q.transpose() -
self.u_curr)).item()
            self.theta_curr = np.arctan2(abstract_space_y,
abstract_space_x) / 2.0
            _, H1, H2, _ = self.formation_variables(self.theta_curr)

            abstract_space_s1 += np.matmul(np.matmul((bot_pos -
self.u_curr).transpose(), H1),
                                (bot.q.transpose() -
self.u_curr)).item()
            abstract_space_s2 += np.matmul(np.matmul((bot_pos -
self.u_curr).transpose(), H2),
                                (bot.q.transpose() -
self.u_curr)).item()


            self.s1_curr = abstract_space_s1 / (2 * (len(swarm_bots) -
1))
            self.s2_curr = abstract_space_s2 / (2 * (len(swarm_bots)  -
1))
            return self.theta_curr, self.s1_curr, self.s2_curr


class swarmbot:
   def __init__(self):

       self.q = np.zeros(shape=[1, 2], dtype=np.float32)
       self.vel = np.zeros(shape=[1, 2], dtype=np.float32)
       self.vel_star = np.zeros(shape=[1, 2], dtype=np.float32)
       self.theta = 0.0
       self.lin_vel = 0.0
       self.ang_vel = 0.0

   def move_bot(self, vel_inertial_frame_cvxopt,
vel_inertial_frame_optimal): #updates tbot position and orientation

       transform_matrix = np.vstack(([np.cos(self.theta),
np.sin(self.theta)],
                                [-np.sin(self.theta)/axle_len,
np.cos(self.theta)/axle_len]))

       vel_moving_frame = np.matmul(transform_matrix,
vel_inertial_frame_cvxopt)
```

```
        (self.linear_vel, self.angular_vel)  =
(min(vel_moving_frame[0].item(), max_lin_vel),
min(vel_moving_frame[1].item(), max_ang_vel))


        linear_disp = del_t * self.linear_vel *
np.array([[np.cos(self.theta), np.sin(self.theta)]])

        angular_disp = del_t * self.angular_vel

        self.q=np.add(self.q, linear_disp)
        self.vel=vel_inertial_frame_cvxopt.transpose()
        self.vel_star=vel_inertial_frame_optimal
        self.theta=self.theta + angular_disp
```

B. *utils.py:*

```
import base
from cvxopt import solvers, matrix
import numpy as np
from base import swarmbot
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import os
import imageio


def check_for_separation_dist(x, y, bot_positions):

    for set_pos in bot_positions:
        dist = ((set_pos[0] - x) ** 2 + (set_pos[1] - y) ** 2) ** 0.5
        if dist < base.seperation:
            return False
    return True


def initialize_swarm(num_of_bots = base.bots_count):

    # container to store the bot positions

    bots = []
    for i in range(num_of_bots):
        bot = swarmbot()
        bots.append(bot)
    bot_positions = []

    for i in range(len(bots)):
        pos_x, pos_y = np.random.normal(base.pos_mean, base.sd, size=(1,
2))[0]

        if check_for_separation_dist(pos_x, pos_y, bot_positions):
            bot_positions.append([pos_x, pos_y])
        else:
```

```
            dist_pass = False
            while not dist_pass:
                pos_x, pos_y = np.random.normal(base.pos_mean, base.sd,
size=(1, 2))[0]
                dist_pass = check_for_separation_dist(pos_x, pos_y,
bot_positions)

            bot_positions.append([pos_x, pos_y])

    for bot_, pos in zip(bots, bot_positions):
        bot_.q = np.array([[pos[0], pos[1]]])

    return bots


def goal_reached(current_state, desired_state):


    curr_centroid_x, curr_centroid_y, curr_theta, curr_s1, curr_s2 =
current_state[0][0].item(), current_state[0][1].item(),
current_state[1].item(), current_state[2], current_state[3]

    des_centroid_x, des_centroid_y, des_theta, des_s1, des_s2 =
desired_state[0][0].item(), desired_state[0][1].item(),
desired_state[1], desired_state[2],desired_state[3]

    # the logic is if the difference is non-ZERO in any case it means
desired state is not reached yet.We round it off to two difits.
    if round(curr_centroid_x - des_centroid_x, ndigits=2):
        return False
    if round(curr_centroid_y - des_centroid_y, ndigits=2):
        return False
    if round(curr_theta - des_theta, ndigits=2):
        return False
    if round(curr_s1 - des_s1, ndigits=2):
        return False
    if round(curr_s2 - des_s2, ndigits=2):
        return False

    return True


def convex_optimizer(C, d, A, b):

    # in order to satisfy the inputs C and d of convex optimization
equation
    C = np.sqrt(2) * C
    d = np.sqrt(2) * d

    # convert to cvxopt matrix
    C = matrix(C, C.shape, 'd')
    d = matrix(d, d.shape, 'd')

    A = matrix(A, A.shape, 'd')
    b = matrix(b, b.shape, 'd')
```

```
    P = C.T * C


    q = -d.T * C
    solvers.options['show_progress'] = False
    solution = solvers.qp(P, q.T, A, b)

    if 'x' in solution.keys():
        return solution['x']
    else:
        return np.array([[0], [0]])


def check_collision_avoidance(i, bots, R, u_curr,
convergence_condition, convergence_constraint):

    # calculate bot position in moving frame
    first_bot_pos = np.matmul(R.transpose(),
np.subtract(bots[i].q.transpose(), u_curr))

    # iterate through all the bots
    for j in range(len(bots)):
        if j == i:
            continue
        else:
            second_bot_pos = np.matmul(R.transpose(),
np.subtract(bots[j].q.transpose(), u_curr))

            position_diff = np.subtract(first_bot_pos, second_bot_pos)

            delta = np.linalg.norm(position_diff, 2)

            # collision avoidance condition
            if delta <= base.seperation:

                first_bot_vel = np.matmul(R.transpose(),
bots[i].vel.transpose())

                second_bot_vel = np.matmul(R.transpose(),
bots[j].vel.transpose())

                velo_diff = np.subtract(first_bot_vel, second_bot_vel)

                collision_avoidance_condition = np.matmul(position_diff,
velo_diff.transpose())

                convergence_condition =
np.vstack((convergence_condition, -collision_avoidance_condition))
                convergence_constraint =
np.vstack((convergence_constraint, np.array([[0.0], [0.0]])))

    return convergence_condition, convergence_constraint

def draw_ellipse(centroid, s1, s2, orientation):
```

```
    return Ellipse(xy=(centroid[0], centroid[1]),
width=base.concentrated_ellipse * s1, height=base.concentrated_ellipse
* s2, angle=orientation * 180/np.pi, edgecolor='b', fill=False)


def plot_swarm(bots, current_state, goal_state, count):
    centroid = current_state[0]
    orientation = current_state[1]
    s1 = current_state[2]
    s2 = current_state[3]
    centroid_g = goal_state[0]
    orientation_g = goal_state[1]
    s1_g = goal_state[2]
    s2_g = goal_state[3]

    results = "./results"
    if not os.path.exists(results):
        os.mkdir(results)
    plt.figure()
    ax = plt.gca()
    ellipse1 = draw_ellipse(centroid, s1, s2, orientation)
    ellipse2 = draw_ellipse(centroid_g, s1_g, s2_g, orientation_g)
    ax.add_patch(ellipse1)
    ax.add_patch(ellipse2)
    plt.xlim((-20, 20))
    plt.ylim((-20, 20))

    for bot in bots:
        bot_pos = bot.q.T
        circle = plt.Circle((bot_pos[0], bot_pos[1]),
(base.axle_len+base.rad), color='r', fill=False)
        ax.add_patch(circle)

        x, y = bot_pos[0], bot_pos[1]
        length = (base.axle_len+base.rad)
        orientation = bot.theta
        endy = y + length * np.sin(orientation)
        endx = x + length * np.cos(orientation)

        plt.plot([x, endx], [y, endy])

    # if draw_lines is not None:
    #     plt.plot(draw_lines[0][0], draw_lines[1][0], color='r',
markersize=3)
    #     plt.plot(draw_lines[0][1], draw_lines[1][1], color='r',
markersize=3)

    file_path = os.path.join(results, str(count) + ".png")
    plt.savefig(file_path)
    plt.show()




def results(state_pts, vel_cap, vel_star, last_lin_vel, last_ang_vel):

    state_tld_pts_x = [state_tld[0].item() for state_tld in state_pts]
```

```
    state_tld_pts_y = [state_tld[1].item() for state_tld in state_pts]
    state_tld_pts_theta = [state_tld[2].item() for state_tld in
state_pts]
    state_tld_pts_s1 = [state_tld[3].item() for state_tld in state_pts]
    state_tld_pts_s2 = [state_tld[4].item() for state_tld in state_pts]

    vel_star_all_x = []
    vel_star_all_y = []
    for i in range(len(vel_star[0])):
        vel_star_all_x.append([])
        vel_star_all_y.append([])
    for i in range(len(vel_star)):
        vel_all_bots = vel_star[i]
        for j in range(len(vel_all_bots)):
            vel_star_all_x[j].append(vel_all_bots[j][0])
            vel_star_all_y[j].append(vel_all_bots[j][1])

    vel_cap_all_x = []
    vel_cap_all_y = []
    for i in range(len(vel_cap[0])):
        vel_cap_all_x.append([])
        vel_cap_all_y.append([])
    for i in range(len(vel_cap)):
        vel_all_bots = vel_cap[i]
        for j in range(len(vel_all_bots)):
            vel_cap_all_x[j].append(vel_all_bots[j][0])
            vel_cap_all_y[j].append(vel_all_bots[j][1])

    plt.figure()
    plt.ylim((-10, 10))
    for i in range(len(vel_star_all_x)):
        plt.plot(vel_star_all_x[i], '-')
        plt.plot(vel_cap_all_x[i], '--')
    plt.legend(loc='upper right')
    plt.title("u_ix and u_star_ix vs time")
    plt.ylabel("u_ix and u_star_ix")
    plt.xlabel("time")
    plt.savefig('results/optimal_computed_velocity_x.jpg')

    plt.figure()


    plt.ylim((-10, 10))
    for i in range(len(vel_star_all_y)):
        plt.plot(vel_star_all_y[i], '-')
        plt.plot(vel_cap_all_y[i], '--')
    plt.legend(loc='upper right')
    plt.title("u_iy and u_star_iy vs time")
    plt.ylabel("u_iy and u_star_iy")
    plt.xlabel("time")
    plt.savefig('results/optimal_computed_velocity_y.jpg')

    plt.figure()
    plt.ylim((-0.5, 0.5))
    plt.plot(last_lin_vel)
    plt.title("linear velocity vs time")
    plt.xlabel("time")
```

```
    plt.ylabel("linear velocity")
    plt.savefig('results/linear_velocity.jpg')

    plt.figure()
    plt.ylim((-2, 2))
    plt.plot(last_ang_vel)
    plt.title("angular velocity vs time")
    plt.xlabel("time")
    plt.ylabel("angular velocity")
    plt.savefig('results/angular_velocity.jpg')

    plt.figure()
    plt.title("desired state vs time")
    plt.xlabel("time")
    plt.ylabel("desired states")
    plt.plot(state_tld_pts_x, label='~x or x_des')
    plt.plot(state_tld_pts_y, label='~y or y_des')
    plt.plot(state_tld_pts_theta, label='~theta or theta_des')
    plt.plot(state_tld_pts_s1, label='~s1 or s1_des')
    plt.plot(state_tld_pts_s2, label='~s2 or s2_des')
    plt.legend(loc='upper right')
    plt.savefig('results/state_tilde_plot.jpg')

    plt.show()

def simulate():
    png_dir = './results'
    images = []
    for file_name in sorted(os.listdir(png_dir)):
        if file_name.endswith('.png'):
            file = os.path.join(png_dir, file_name)
            images.append(imageio.imread(file))

    for _ in range(100):
        images.append(imageio.imread(file))

    imageio.mimsave('./results/2d_simulation.gif', images)
```

C. main.py:

```
import base
from base import abs_space
import utils
import numpy as np



def main_algo(num_bots=None, num_iterations=None):
    if num_bots is None:
        num_bots = base.bots_count
    if num_iterations is None:
        num_iterations = base.iters

    abstract_space = abs_space()
    curr_centroid = abstract_space.get_centroid(None)
```

```
    theta_curr, s1_curr, s2_curr = abstract_space.parameters(None)

    bots = utils.initialize_swarm(num_bots)

    counter = 0
    vel_cap_pts = []
    vel_star_pts = []
    bot_last_vel_pts_lin = []
    bot_last_vel_pts_ang = []
    state_tilde_plot_pts = []

    desired_state_index = 0

    while True:

        curr_centroid = abstract_space.get_centroid(bots)
        theta_curr, s1_curr, s2_curr = abstract_space.parameters(bots)
        current_state = [curr_centroid, theta_curr, s1_curr, s2_curr]

        desired_state = base.des_abstract_state[desired_state_index]

        if counter % 1000 == 0:
            utils.plot_swarm(bots, current_state,   desired_state,
str(desired_state_index) +"_" + str(counter))

        if utils.goal_reached(current_state, desired_state) \
                or counter > num_iterations:
            if len(base.des_abstract_state) > 1 and desired_state_index
< len(base.des_abstract_state) - 1:
                desired_state_index += 1
                counter = 0
            else:
                break


        state_tilde = np.vstack((np.subtract(desired_state[0],
current_state[0]),
                                   desired_state[1] - current_state[1],
                                   desired_state[2] - current_state[2],
                                   desired_state[3] -
current_state[3]))


        control_vector_fields = [np.matmul(base.KU,
np.vstack((state_tilde[0], state_tilde[1]))),
base.KT*state_tilde[2].item(), base.KS1*state_tilde[3].item(),
base.KS2*state_tilde[4].item()]
        centroid_derivative, theta_derivative, s1_derivative,
s2_derivative = control_vector_fields[0],\

  control_vector_fields[1],\

  control_vector_fields[2],\

  control_vector_fields[3]
        R, H1, H2, H3 = abstract_space.formation_variables(theta_curr)
```

```
       Lie_grp = np.vstack(((np.hstack((R, curr_centroid))),
np.array([0, 0, 1])))

       Gamma = np.vstack((np.hstack((Lie_grp, np.zeros(shape=(3, 2)))),
                         np.hstack((np.zeros(shape=(2, 3)), base.I))))

       vel_cap_all_bots = []
       vel_star_all_bots = []

       for i in range(len(bots)):

           bot = bots[i]

           vel_star = np.add(np.add(centroid_derivative, (
                                    (s1_curr - s2_curr) * np.matmul(H3,
np.subtract(bot.q.transpose(), curr_centroid)) * theta_derivative /
(s1_curr + s2_curr))),
                                    np.add((np.matmul(H1,
np.subtract(bot.q.transpose(), curr_centroid)) * s1_derivative / 4 *
s1_curr),
                                    (np.matmul(H2,
np.subtract(bot.q.transpose(), curr_centroid)) * s2_derivative / 4 *
s2_curr)))

           bot_pos_moving_frame = np.matmul(R.transpose(),
np.subtract(bot.q.transpose(), curr_centroid))

           # calculate differential of surjective submersion
           diff_phi = np.vstack((base.I,
                             (1/s1_curr-
s2_curr)*np.matmul(bot_pos_moving_frame.transpose(), base.E1),
                             np.matmul(bot_pos_moving_frame.transpose()
, np.add(base.I, base.E2)),
                             np.matmul(bot_pos_moving_frame.transpose()
, np.subtract(base.I, base.E2))))

           # monotonic convergence constraint
           convergence_condition =
np.matmul(np.matmul(np.matmul(state_tilde.transpose(),
                                                                  base.g
ain_matrix), Gamma), diff_phi)

           # flip the sign to maintain the inequality given at (14) of
[1] for convex optimization
           convergence_condition = -convergence_condition
           convergence_constraint = np.array([[0.0]])


           # check for collision avoidance with every team member and
impose additional constraint if neccessary
           convergence_condition, convergence_constraint =
utils.check_collision_avoidance(i, bots, R, curr_centroid,

                    convergence_condition,
```

```
                    convergence_constraint)

            try:
                vel_convex_opt = utils.convex_optimizer(base.I,
np.matmul(R.transpose(), vel_star),
                                                        convergence_cond
ition, convergence_constraint)
            except Exception as e:
                break

            if counter == 0 or counter % 1000 == 0:
                vel_cap_all_bots.append(vel_convex_opt)
                vel_star_all_bots.append(vel_star)

            vel_inertial_frame = np.matmul(R, vel_convex_opt)

            bots[i].move_bot(vel_inertial_frame_cvxopt=vel_inertial_fram
e,
                                 vel_inertial_frame_optimal=vel_star.tr
anspose())

        # i = 0
        if counter % 50 == 0:
            state_tilde_plot_pts.append(state_tilde)
            bot_last_vel_pts_lin.append(bots[0].linear_vel)
            bot_last_vel_pts_ang.append(bots[0].angular_vel)
        if counter % 1000 == 0:
            vel_cap_pts.append(vel_cap_all_bots)
            vel_star_pts.append(vel_star_all_bots)

        counter = counter + 1

   utils.results(state_tilde_plot_pts, vel_cap_pts, vel_star_pts,
bot_last_vel_pts_lin, bot_last_vel_pts_ang)

   utils.simulate()

main_algo()
```

REFERENCES

1. https://sites.bu.edu/hyness/files/2014/05/icra06_swarms_3D.pdf
2. Michael, N. and Kumar, V. (2008). Controlling shapes of ensembles of robots of finite size with nonholonomic constraints. Robotics: Science and Systems, Zurich, Switzerland.
3. R. Murray, Z. Li, and S. Sastry, A Mathematical Introduction to Robotic Manipulation. Florida: CRC Press, 1993.
4. M. Zefran, V. Kumar, and C. B. Croke, “On the generation of smooth three-dimensional rigid body motions,” IEEE Transactions on Robotics and Automation, vol. 14, no. 4, pp. 576–589, Aug. 1998.
5. S. M. LaValle, Planning Algorithms. Cambridge, U.K.: Cambridge University Press, 2006.