Decentralised Algorithms for Area Coverage

By

SAM YOUNG

CDT for Communications Engineering
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of MASTERS BY RESEARCH in the Faculty of Engineering.

AUGUST 2018

Abstract

Decentralised area coverage is a fundamental problem in swarm robotics, in which robots seek to disperse evenly across an area. Area coverage is the basis for many swarm applications from environmental monitoring and network deployment to cancer treatment using nanobots. Yet most implementations chose an area coverage strategy without a clear understanding of the tradeoffs they entail in terms of optimal performance, robot design, and environmental constraints.

In this paper we derive analytic lower bounds for optimal area coverage when the number of robots is large. We compare four popular algorithms for area coverage: Langevin Dynamics, Run and Tumble, Electrostatic Repulsion and Local Repulsion. We derive analytic approximations for the distribution of robots and performance for the first three, and suggest likely values for Local Repulsion. We run extensive simulations of each algorithm and compare the results to the lower bounds, showing how close each algorithm gets to optimal performance.

Dedication and acknowledgements

To Pete, who believed in this research.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

TABLE OF CONTENTS

List of Tables

Table **Page**

List of Figures

CHAPTER 1

INTRODUCTION

As robots become cheaper and more efficient, there are an increasing number of situations where it makes sense to deploy them in large swarms. For diverse applications from underwater ocean monitoring with autonomous buoys, to delivering cancer drugs within human tissue using nanobots, we need to find a way of covering a wide areas in a decentralised way [1] [12] [3] [4] [16] [21] [26]. Key requirements for such systems are longevity and low cost per unit. To deploy tens or hundreds of thousands of robots, we need to build them cheaply and service them occasionally or never [20].

Inroads are being made into producing cheap robots capable of collaborating in large numbers. One such example is the Kilobot, a low cost robot for studying collective behaviours. [28] [29] Kilobots can control their motion - although not precisely, interact with robots within communication range, and can sense their environment. Swarms that operate in larger numbers, for example nanobots in biomedical applications may instead be only capable of random motion and minimal or no interactions with the environment or other robots [11]. Swarms that work in smaller numbers may benefit from more advanced robot hardware with precise control, longer range interactions a sensing, and more computational power [14].

Constraints in terms of robot numbers and capabilities are essential when considering optimal area coverage. To design affordable swarms capable of covering wide areas (relative to the size of each robot), we need to develop area coverage algorithms that require minimal communication and low power. Where possible we would like to make use energy from the surrounding environment to propel the agents, such as the wind or ocean currents, or diffusion gradients for nanoscale systems. This allows each agent to be as simple and low power as possible. However, it requires control algorithms that can cope with (or indeed require) significant levels of noise in motion.

In this paper we will assess the area coverage capabilities of a swarm of n robots. Specifically

we will tackle the following three questions: (a) With optimal control, what will the furthest distance from any point in the area to the nearest robot be? (b) How close to optimal performance can we get? (c) How long will it take to reach this level of coverage? We consider these problems in the context of area coverage algorithms with different levels of communication and motion noise [13] [19] [21].

Broadly speaking, we can divide the problem into two cases, one in which the robots act independently, and another in which they interact. In section 2, we analyse algorithms for independently acting robots. For such systems, optimal coverage is achieved when robots are scattered uniformly at random over the area. To achieve this, robots must have a random element in their control, either by design or due to noise.

For fast deployment and convergence, the robots' motion needs to have high autocorrelation. Robots should travel in straight lines, only changing direction when they reach the edge of the area, or occasionally tumbling to ensure that overall coverage remains even. Hence for independently acting robots, how fast and effective coverage is largely depends on the inertia of the robot relative to the perturbations due to noise.

For robots that are capable of interaction, the problem becomes more interesting. There are many effective algorithms that allow improvements based on avoiding other robots that are nearby. In section 3, we find mathematical bounds on the improvements offered by interaction, and consider two simple algorithms- electrostatic repulsion and local repulsion. Despite their simplicity, these algorithms give close to optimal coverage with fast convergence. For n robots in an area of radius R, we find that a communication range proportional to $R/\sqrt{n}$ is necessary to reach optimal coverage, and in practise, usually sufficient too.

Using simulations, we model the effective coverage in situations with communication distances close to $R/\sqrt{n}$, and in section 4, we consider how these results might apply with different levels of noise and in different applications. A table of convergence speeds from the simulations and analytic results for coverage performance is given at the bottom. Finally, we suggest which algorithms are most suited to the use cases given in the leading paragraph.

1.1 Stochastic Processes

To get a complete picture of how non-deterministic agents travel around a continuous space, in this paper we will model the trajectory of each agent i as a stochastic process, $\mathbf{X}_i(t)$:

$$\mathbf{x}_i(t) \sim \mathbf{X}_i(t) \tag{1.1}$$

A stochastic process $\mathbf{X}_i(t)$ is a collection of random variables $\{\mathbf{X}(t) : t \in T\}$, where T is an ordered set with state space Ω and an associated σ-algebra of Borel sets. In this paper, we will stick to a real timeline $T = \mathbb{R}^+$and a two-dimensional state space $\Omega = \mathbb{R}^2$. Note that the following overview draws heavily from [23]- a recent graduate textbook on stochastic processes.

A stochastic process is strictly stationary if its distribution $\mathbf{X}(t)$ does not change over time, i.e if for any $t_1,...,t_k \in T$, $A_1,...,A_k \subset \Omega$ and $s \in T$:

$$\mathbb{P}\big(\mathbf{X}(t_1+s) \in A_1,\ \dots\ , \mathbf{X}(t_k+s) \in A_k\big) \ = \ \mathbb{P}\big(\mathbf{X}(t_1) \in A_1,\ \dots\ , \mathbf{X}(t_k) \in A_k\big) \tag{1.2}$$

A stochastic process is weakly stationary if the first moment is constant and the covariance depends only on $t-s$:

$$\mathbb{E}\big(\mathbf{X}(t)\big) = \boldsymbol{\mu} \tag{1.3a}$$

$$\mathbb{E}\big((\mathbf{X}(t)-\boldsymbol{\mu})(\mathbf{X}(s)-\boldsymbol{\mu})\big) = C(t-s) \tag{1.3b}$$

where $\boldsymbol{\mu}$ is the mean of $\mathbf{X}(t)$ and $C(t-s)$ is the covariance. The covariance function $C : T \to \mathbb{R}$ is also referred to as the autocorrleation. It expresses how correlated the motion of $\mathbf{X}(t)$ is over time. A smooth process that changes direction slowly will have high positive autocorrelation for a significant amount of time, whereas a very wiggly process that changes direction rapidly will have an autocorrelation that quickly tends to zero.

The space of stochastic processes is very large and not all are appropriate for modelling the behaviour of robots, or particles in a real space. In particular, we want to guarantee that the trajectories of the robots are continuous and sufficiently smooth. For simplicity, in certain circumstances it will also expedient to make a Markov assumption, i.e. that the process retains no memory of the past, and the future trajectory of a robot is dependent only on its present state. More formally, we can say that a stochastic process is a Markov process iff:

$$\mathbb{P}\big(\mathbf{X}(t) \in \Gamma \mid \mathscr{F}_s^{\mathbf{X}}\big) \ = \ \mathbb{P}\big(\mathbf{X}(t) \in \Gamma \mid \mathbf{X}(s)\big) \tag{1.4}$$

for all Borel sets $\Gamma \subset \Omega$, and $s,t \in T$ with $s \leq t$, where $\mathscr{F}_s^{\mathbf{X}}$ is the filtration of $\mathbf{X}(t)$ at time s [23] (see definition 2.3). We will also need to ensure that the conditional probability density exists:

$$\mathbb{P}\big(\mathbf{X}(t+h) \in \Gamma \mid \mathbf{X}(t) = \mathbf{x}\big) = \int_\Gamma p(\mathbf{y}, t+h|\mathbf{x}, t)dt \tag{1.5}$$

This guarantees that the transition function $P(\Gamma, t|\mathbf{x}, s) := \mathbb{P}(\mathbf{X}(t) \in \Gamma \mid \mathscr{F}_s^{\mathbf{X}})$ exists and is measurable.

For larger robots, which have mass, these constraints alone will not be sufficient. Hence we will also want certain constraints on the smoothness of the path of the robot. In this case we will model the paths of robots as diffusion processes. Again from [23] (definition 2.6), a diffusion process is a Markov process with transition function $P(\Gamma, t|\mathbf{x}, t)$ satisfying the following constraints:

1. Continuity

$$\forall \mathbf{x} \in \Omega\ \forall \epsilon > 0\ \forall s < t \quad \int_{|\mathbf{x}-\mathbf{y}|>\epsilon} P(d\mathbf{y}, t|\mathbf{x}, s) = o(t-s) \tag{1.6}$$

2. Definition of drift coefficient

$$\exists \mathbf{b}(\mathbf{x},s)\ \forall \mathbf{x}\in\Omega\ \forall \epsilon>0\ \forall s<t \quad \int_{|\mathbf{x}-\mathbf{y}|\le\epsilon}(\mathbf{y}-\mathbf{x})P(d\mathbf{y},t|\mathbf{x},s)=\mathbf{b}(\mathbf{x},s)+o(t-s) \tag{1.7}$$

3. Definition of diffusion coefficient

$$\exists \mathbf{\Sigma}(\mathbf{x},s)\ \forall \mathbf{x}\in\Omega\ \forall \epsilon>0\ \forall s<t \quad \int_{|\mathbf{x}-\mathbf{y}|\le\epsilon}(\mathbf{y}-\mathbf{x})(\mathbf{y}-\mathbf{x})^T P(d\mathbf{y},t|\mathbf{x},s)=\mathbf{\Sigma}(\mathbf{x},s)+o(t-s) \tag{1.8}$$

For more details on multidimensional stochastic processes, see [30], chapter 2.

The dynamics of diffusion processes can be understood by looking at the infinitesimal generator $\mathscr{L}$ of the Markov group that governs transitions P_t of the Markov process, and its adjoint $\mathscr{L}^*$. By manipulating these, it is possible to derive the backward and forward Kolmogorov equations. In particular, the forward Kolmogorov or Fokker-Planck equation is a PDE (Partial Differential Equation) that determines the time-evolution of the probability density of a diffusion process.

For a multidimensional diffusion process $\mathbf{X}(t)$ with a transition function $P(\Gamma,t|\mathbf{x},t)$ with smooth density, drift coefficient $\mathbf{b}(\mathbf{x},s)$, and diffusion coefficient $\mathbf{\Sigma}(\mathbf{x},s)$, the Fokker-Planck equation is:

$$\frac{\partial p}{\partial t}=\nabla_y\cdot\Big(-\mathbf{b}(t,\mathbf{y})p+\frac{1}{2}\nabla_y\cdot(\mathbf{\Sigma}\big(t,\mathbf{y})p)\Big) \tag{1.9}$$

with boundary condition $p(\mathbf{y},s|\mathbf{x},s)=\delta(\mathbf{x}-\mathbf{y})$. For more details, and a derivation, see [23], chapters 2 and 3.

The simplest example of a diffusion process is Brownian motion in a bounded domain with reflective boundary conditions. Brownian motion is defined as a Wiener process $\mathbf{W}(t)$. $\mathbf{W}(t)$ is a Markov process with initial condition $\mathbf{W}(0)=\boldsymbol{\delta}(0)$, and a Gaussian transition function $\mathbf{W}(t+s)-\mathbf{W}(t)\sim\mathscr{N}(\mathbf{0},s)$. We can think of browning motion as the limit of a the sum of a sequence of scaled normal random variables. More precisely, we can say that if $\boldsymbol{\xi}(k)\sim\mathscr{N}(\mathbf{0},1)$ for $k\in\mathbb{N}$ are iid random variables,

$$\frac{1}{\sqrt{n}}\sum_{1\le k\le\lfloor nt\rfloor}\boldsymbol{\xi}(k)+\frac{nt-\lfloor nt\rfloor}{\sqrt{n}}\boldsymbol{\xi}(\lfloor nt\rfloor+1) \tag{1.10}$$

converges in distribution to $\mathbf{W}(t)$ as $n\to\infty$ [23] p12.

We can also describe the time-evolution of diffusion processes using SDEs (Stochastic Differential Equations). SDEs take the form:

$$d\mathbf{X}(t)=\mathbf{b}(t,\mathbf{X}(t))dt+\boldsymbol{\sigma}(t,\mathbf{X}(t))d\mathbf{W}(t) \tag{1.11}$$

The first term $\mathbf{b}(t,\mathbf{X}(t))$ is a standard multidimensional Lesbegue integral, however the second term $\boldsymbol{\sigma}(t,\mathbf{X}(t))d\mathbf{W}(t)$ has yet to be defined. It can be shown that if f is square-integrable, then there exists a stochastic process $\mathbf{I}(t)$ such that:

$$\sum_{k=0}^{K-1}f(t_k)\big(\mathbf{W}(t_{k+1})-\mathbf{W}(t_k)\big)\to\mathbf{I}(t)\quad\text{as}\quad K\to\infty\quad\text{in probability} \tag{1.12}$$

[23] (p52-60). We say that $\mathbf{I}(t)$ is the It'o integral of f:

$$\mathbf{I}(t)=\int_0^t f(s,\mathbf{X}(t))d\mathbf{W}(s) \tag{1.13}$$

1.2 Literature Review

The project began with a thorough review of area coverage and swarming network deployment applications, and the mathematical techniques used to model them. From the beginning it became clear that there are essentially three arenas where people hope to deploy swarms of robots:

1. Macroscopic scenarios, where robots are deployed to cover a very wide area of land or sea.

2. Microscopic scenarios, where tiny robots or active particles are deployed in very large numbers in small area.

3. Mid-scale scenarios, where robots are deployed on an approximately human scale, such as in a building, or around a university campus.

Broadly speaking, it became apparent quite early on that the techniques appropriate for scenario 3. are often quite different to those that can be applied in 1. or 2.

Our literature review focused on papers about the deployment of a large number of agents, i.e. how they would disperse from a small area to cover a larger area. We included papers from maths, physics, engineering and even biology journals that have something to say about algorithms for dispersion, or area coverage.

Being based in the Communications engineering department, the first applications to garner our interest were wide scale networking problems. In particular we were interested in how to deploy a series of drones or balloons carrying network equipment across a wide area. Such technology has already been deployed by Google [9], and other large technology companies and network operators aim to deploy similar systems in the near future.

Another major application for macroscopic swarm deployment is environment monitoring [20], in which large sensor networks are deployed to keep track of changes or processes occurring in the natural environment. Wireless sensor networks have a wide array of applications such as detecting the presence of forest fires [34], monitoring the underwater acoustics to detect animals and submarines or drones [1] , and monitoring animal populations in remote areas [17].

Literature on area coverage and deployment for macroscopic systems has largely focused on flocking strategies for intelligent robots with point to point communication, sensing capabilities and accurate motors. An early algorithmic description of flocking is given in Reynolds' paper on rendering flocks of birds in computer graphics [27]. Reynolds' gives the following three rules for flocking in a followed by each agent in a proximity network:

1. Flock centring - agents attempt to stay near to their fellow flocking agents.

2. Collision avoidance - agents repulse each other locally to avoid crashing.

3. Velocity matching - agents aim to keep pace with nearby agents.

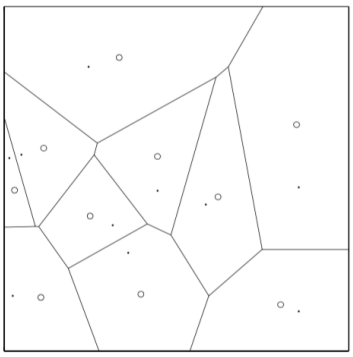


Figure 1.1: A voronoi tessellation

Reynolds' rules describe a proximity network, in which each agent makes decisions based on the behaviour of its neighbours. The criterion for neighbourhood is usually defined by distance metric. For dispersion algorithms, we simply take a ball of radius r, centred at the location $\mathbf{x}_i(t)$ of agent i at time t. By moving towards the centre of the flock, and attempting to avoid their neighbours, a flock of approximately even density is created.

These rules are expressed mathematically in Olfati-Saber's first algorithm [19] in which agents achieve these goals by reaching a consensus on their velocity, according dynamics of the form:

$$\dot{\mathbf{v}}_i(t) = \sum_{j \in N_i} \phi(\mathbf{x}_j(t) - \mathbf{x}_i(t)) + \sum_{j \in N_i} (\mathbf{v}(t)_j - \mathbf{v}_i(t)) \tag{1.14}$$

where $\mathbf{v}_i(t)$ is the velocity of agent i at time t, and N_i are the neighbours of agent i, defined as a the agents within a distance d of $\mathbf{x}_i(t)$, the location of agent i at time t, and ϕ is a smooth scalar function. Olfati-Saber applies the techniques from complex systems to show that agents reach a consensus on the flock velocity. He then applies a stability analysis to show that the motion converges to an equilibrium where the agents are arranged in a lattice-like configuration.

Another algorithm appropriate for robots capable of communicating their location and velocity accurately over short distances is voronoi descent. The voronoi descent method is a discrete algorithm in which at each timestep t, robots determine the boundary of their voronoi cell based on the location of their neighbours, and accelerate towards the barycentre. This method leads to an approximately even coverage of the space. At each timestep, a voronoi partition is created by drawing a line bisecting the vector between each pair of neighbouring robots. The barycentre of each voronoi cell V is defined as the centre of mass $\int_{\mathbf{y} \in V} \mathbf{y}$. Figure 1.1 shows a voronoi tessellation of points, with the barycentres marked as circles. Robots accelerate towards the barycentre of their voronoi cell, until the next timestep. In [7], precise mathematical descriptions of a family of voronoi descent algorithms are given, and it is proven that the voronoi descent method converges in linear time.

All of the algorithms we had considered up to this point had been deterministic. But we wanted to look at algorithms that took account of the random effects that are present, both on a macroscopic scale, and on a microscopic scale. In particular, at a very small scale, we wanted to consider how semi-controlled particles might behave.

In many ways the base case to compare how effective dispersion algorithms are at a microscopic level is diffusion. Diffusion can be described formally as Brownian motion, a limit of a series of random walks as the step size tends to zero. A more thorough definition is given in 1.1. However simple diffusion is painstakingly slow, and has the rather unfortunate side effect of having non-smooth trajectories with infinite acceleration. For anything larger than a small particle, diffusion is usually not the best model. In chapter 2 we take a closer look at the Langevin dynamics, which models a massive particle suspended in a fluid, and also Run and Tumble, which models a particle moving in straight lines but bouncing of neighbours and/or obstacles.

The simplest type of interaction between particles is concentration-dependent diffusion, where the density of particles affects the rate of diffusion. Concentration dependent diffusion can be expressed as a differential equation:

$$\frac{\partial C}{\partial t} = \frac{\partial}{\partial x}\Big(D(C)\frac{\partial C}{\partial x}\Big) \tag{1.15}$$

where x is the location of a particle, t is time, C is concentration and $D(C)$ is the diffusion coefficient at concentration C [6]. By solving this equation, we can find the dynamics of the particles, and their distribution $C(x)$ as $t \to \infty$.

Depending on $D(c)$, the dimensionality of the space, and the boundary conditions, various different solutions to 1.15 have been given. For a smooth function D, Philip [24] gives a numerical solution based on the Runge-Kutta method. In [22], Pattle derives a diffusion from an instantaneous point source in a circularly symmetric space, where $D = D_0 C^n$, for some n. Philip also gives a series solution for a concentration-dependent diffusion from a circularly symmetric membrane in two and three dimensions [25].

Of particular interest in the literature were studies that considered active particles, or semi-controlled particles. For a detailed survey see: [2]. One such algorithm is the Vicsek model, in which similarly to Run and Tumble, particles travel at a constant speed, but align their direction of travel over time [31]. At each timestep, the position of each particle is updated according to the difference equation:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t)\Delta t \tag{1.16}$$

where the velocity $\mathbf{v}_i(t)$ of each agent has constant magnitude and an angle $\theta_j(t+1)$ given by:

$$\theta(t+1) = \langle \theta_j(t) \rangle_{j \in N_i} + \xi(t) \tag{1.17}$$

where N_i are neighbours of i, and $\xi(t) \sim Unif(-\eta/2, \eta/2)$. Vicsek demonstrates numerically that when η increases to around 2.5, a phase transition occurs, where the correlation of the particles' velocity $v_a = \frac{1}{N v_0} |\sum_i \mathbf{v}_i|$ collapses to almost zero.

Understanding behaviour in terms of averages is an important step when modelling non-deterministic agents in large numbers. Hamann and Wörn want to produce a general purpose framework for predicting and analysing the behaviour of swarms. Their approach uses statistical physics [10], in particular the Langevin dynamics and Fokker-Planck equations. The Langvein equation for a single robot is given by:

$$\dot{\mathbf{x}}(t) = -\mathbf{a}(\mathbf{x}(t), t) + b(\mathbf{x}(t), t)\boldsymbol{\xi}(t) \tag{1.18}$$

Where $\mathbf{a}(\mathbf{x}, t)$ is a function that describes the deterministic motion based on information provided by the environment at location $\mathbf{x}$ and time t, $b(\mathbf{x}, t)$ is a deterministic function that gives the magnitude of random fluctuations experienced at point $\mathbf{x}$ at time t, and $\boldsymbol{\xi}(t)$ is white Gaussian noise caused by random perturbations.

The Fokker-Planck equations for this model, derived under the Stratonovich convention are given by:

$$\frac{\partial \rho(\mathbf{x}, t)}{\partial t} = -\nabla\big(\mathbf{a}(\mathbf{x}, t)\rho(\mathbf{x}, t)\big) + \frac{1}{2}D\nabla^2\big(b^2(\mathbf{x}, t)\rho(\mathbf{x}, t)\big) \tag{1.19}$$

where $D = \mathbb{E}[\boldsymbol{\xi}(t)^2]$ is the mean square of the random perturbations before they are scaled by $b(\mathbf{x}, t)$. In the simple case of external influence, we can define $\mathbf{a}(\mathbf{x}, t)$ as a function of a potential field $\mathbf{q}$ that influences the robot's movements: $\mathbf{a}(\mathbf{x}, t) = -\nabla\mathbf{q}(\mathbf{x}, t)$. This potential field also be used to define pheromone-like behaviour where robots can implicitly influence one another by interacting with the potential field.

In an illustrative application, robots move towards a beacon while avoiding collisions and staying close enough to one another to maintain communication links. Robots continue moving forward unless one of the following happens:

1. Two robots get close to one another and have to take avoiding action
2. Robots start to receive pings from too few neighbours and attempt to realign
3. Robots change direction randomly and attempt to realign.

Hamann and Wörn derive the drift coefficient $\mathbf{a}$ and set the diffusion coefficient D for Fokker-Plank equation for this set of controller rules, and derive a taxis behaviour, where it can be shown that robots move in a convoy towards the beacon over time.

In [3], Berman et. al. give a description of how to control a swarm of robots using broadcasting, based on smoothed-particle hydrodynamics. The application is pollination of Rabitege blueberries. The positions of the robots are given by the following Ito integral:

$$\mathbf{X}_i(t)dt = \mathbf{X}_i(t) + \mathbf{V}(\mathbf{X}_i(t))dt + \sqrt{2D}\,d\mathbf{W}(t) \tag{1.20}$$

The species concentration is described using an advection-diffusion reaction, which results in a PDE for the Fokker-Planck equation. The solution to this equation is approximated using

Smoothed particle hydrodynamics, where the influence of each robot is assumed to tend towards zero outside its immediate neighbourhood.

More recently, there have been significant advances on the topic of area coverage. For instance in [5], Chupeau et. al. consider optimal strategies for a single random agent searching a compact, geometric lattice with n nodes. For large enough n, they find optimal search strategies to visit all nodes, in terms of n, and the average passage time taken to reach an arbitrary node, starting from an arbitrary point in the lattice. This applies not just to random walks, but also Levy flights and persistent random walks, where the agent continues in one direction for a random number of steps before switching direction. It is, however, a discrete result, and has not been extended to a walker on a plane.

In [8], collaborative approaches to area coverage are explored. Agents move in a correlated, continuous space random walk characterised by a step size and a turning angle. The turning angle is given by a Cauchy distribution parametrised by ρ. This distribution can be tuned to give a ballistic motion ($\rho = 0$), where the turning angle becomes a Dirac delta function centred at zero, or a completely uncorrelated random walk ($\rho = 1$), where the turning angle becomes a uniform distribution on $(-\pi, \pi)$, or anything in between these extremes. Agents remember the point at which they last encountered another agent, and move in a correlated random walk away from that point with the trajectory fixed for an amount of time T.

Giuggioli at. al. partition the area into a grid of discrete cells, and define the coverage time as the amount of time taken for at least one agent to visit each cell once. Firstly they look at the impact of interaction between n agents, finding a relationship between $\eta = n \cdot \pi(d/2)^2/A$, the ratio between the area of repulsion multiplied by n, and the confining area A. Secondly they consider the impact of correlation of motion at each timestep on coverage time for a single agent. They find an optimal value of $\zeta = -\lambda/(A \cdot \ln(\rho))$, the ratio between the average distance a correlated walker moves without turning, and the size of the confining area A, which minimises the coverage time. Both of these results are discovered empirically through simulations.

1.3 Area Coverage in a Disc

To complement these useful results quantifying the time taken to perform an exhaustive search, in this paper we wanted to consider how long it takes to reach a stable distribution of agents throughout an area, or at least a stable distance from an arbitrary point to the nearest agent. This is particularly useful for tasks such as deploying communications networks and persistent sensor networks, where the main prerogative is providing a consistent level of coverage. It is less useful for area sweeping tasks such as foraging or search and rescue. We remain interested in coverage time, but in regard to how quickly a consistent level of coverage is reached.

Suppose we have a circular region $\Omega = D(\mathbf{0}, R)$ and let $\mathbf{y} \in \Omega$. We start with robots $i = 1, ..., n$ at locations $\mathbf{x}_i(t_0)$ sampled uniformly at random from a smaller disc of radius R_0. Using

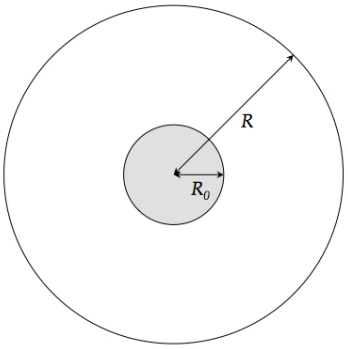


Figure 1.2: Starting distribution of the robots

decentralised control, we seek to disperse the robots in such a way as to minimise the maximum distance from a target to the nearest robot. To measure this, we define the coverage distance C:

$$C(t) := \max_{\mathbf{y}\in\Omega} \min_{1\leq i\leq n} ||\mathbf{y} - \mathbf{x}_i(t)|| \tag{1.21}$$

Given the trajectories of each robot over time $\mathbf{x}(t)$, we can calculate $C(t)$. For each algorithm, we want to know (a) What is the limit of $C(t)$ as $t \to \infty$? (b) How quickly does $C(t)$ converge to this limit? Algorithms that perform well converge quickly to a low coverage distance. Note that (b) is a similar metric to 'coverage time' from [8], but for the coverage distance metric C, rather than the time for at least one agent to visit each cell in a discretised area.

In all of the use cases outlined above, there is some element of noise that affects the motion of the robots. Hence we model the location of each robot as a stochastic process $\mathbf{x}_i(t) \sim \mathbf{X}_i(t)$. A stochastic process is a collection of random variables indexed by time $\{\mathbf{X}(t)\}_{t\geq t_0}$, with the same state space. In this case, $\Omega = D(\mathbf{0}, R)$, the disc of radius R, centred at the origin. We use lower case letters $\mathbf{x}(t), \mathbf{v}(t) \in \mathbb{R}^2$ to refer to values of the locations or velocities of robots, and upper case letters $\mathbf{X}(t), \mathbf{V}(t)$ to refer to stochastic processes that model them.

There are two types of coverage algorithm: those in which the robots interact with each another, and those in which they make move independently. If they move independently, then the $\mathbf{X}_i(t)$ are iid (independent and identically distributed). Hence the probability distribution of the location of any robot $\mathbf{x}_i(t)$ is given by the same stochastic process $\mathbf{x}_i(t) \sim \mathbf{X}(t)$. If the robots interact, we need a separate process to describe the motion of each robot $\mathbf{X}_i(t)$, which is depends on $\mathbf{X}_j(t)$ for all $j \neq i$.

In useful coverage algorithms, the locations of the robots on the disc stabilises over time to a particular probability distribution $p(\mathbf{x}, t) \to p(\mathbf{x})$ as $t \to \infty$. Such a distribution is referred to as the stationary distribution. If $\mathbf{X}(t)$ is a Markov process (its past is independent, given its present state), and ergodic (the long term time average of its values is equal to the expected value across the domain Ω) then has a stationary distribution [23]. Of particular interest are algorithms that converge to the uniform distribution, since these provide the best coverage of the disc for independently acting robots.

CHAPTER 2

DISTRIBUTION OF INDEPENDENTLY ACTING ROBOTS

In many of the scenarios in which area coverage algorithms are deployed, robots are exposed to significant noise from the environment. Furthermore, for algorithms in which the robots act independently, randomness is necessary to reach a uniform stationary distribution. If every robot followed the same path, then they would all end up in the same place. To reflect the effect of noise on the motion of the robots, we model the position of each robot $\mathbf{x}_i(t)$ using a single stochastic process $\mathbf{x}_i(t) \sim \mathbf{X}(t)$ with domain Ω.

Suppose that a robot travels in a straight line, but is buffeted by Gaussian perturbations and experiences linear drag. Then its motion can be modelled by a massive particle that moves subject to the Langevin dynamics:

$$\mathbf{V}(t) = \frac{d\mathbf{X}(t)}{dt} \tag{2.1a}$$

$$m\frac{d\mathbf{V}(t)}{dt} = -\gamma\mathbf{V}(t) + \sigma\boldsymbol{\xi}(t) \tag{2.1b}$$

where m is the mass, γ is the drag coefficient, σ^2 is the variance of the perturbations, and $\boldsymbol{\xi}(t) \sim \mathcal{N}(\mathbf{0}, 1)$ is white Gaussian noise. Note that (2.1) are stochastic differential equations (SDEs), since they apply to stochastic processes rather than deterministic functions. This makes the analysis much harder, so we focus on the limit cases. For a detailed introduction to SDEs see [23].

Since we aim to distribute the robots evenly across the target area, there are two main questions to ask about the robots trajectory $\mathbf{X}(t)$. Firstly, does it converge to a uniform distribution? Secondly, how long does it take to reach its stationary distribution? The main determining factor for both of these questions is how ballistic the motion of the robots is, or to put it slightly differently: the autocorrelation of $\mathbf{V}(t)$.

Also in [23], it is shown that $\mathbf{V}(t)$ is an Ornstein-Uhlenbeck process, with a zero-mean Gaussian stationary distribution as $t \to \infty$. Hence we can write the autocorrelation of $\mathbf{V}(t)$ as a

value depending only on t:

$$C_{\mathbf{V}}(t) = \mathbb{E}[(\mathbf{V}(t) \cdot \mathbf{V}(0)] \tag{2.2}$$

In free space, the Kubo relation states that once a stationary distribution for velocity has been reached, the diffusion coefficient D for $\mathbf{X}(t)$ is the integral of $C_{\mathbf{V}}(t)$. Or to put it in terms of the mean squared displacement:

$$||\mathbb{E}[(\mathbf{X}(t) - \mathbf{X}(0))^2]|| \to 2\int_0^\infty C_{\mathbf{V}}(u)du \cdot t \quad \text{as} \quad t \to \infty \tag{2.3}$$

We can relate this to the coefficients of the Langevin equation using the Stokes-Einstein equation, which states that $D = \frac{m}{\gamma}\mathbb{E}[||\mathbf{V}(t)^2||]$ in the stationary, or equilibrium distribution for velocity in two dimensions. Hence we can see here that the autocorrelation of the velocity of the Langevin motion is inversely proportional to the drag coefficient γ.

To illustrate how this affects the dynamics of the Langevin particle, we can consider two limit cases, one in which $m \ll \gamma$, resulting in very low autocorrelation of velocity, and another in which $m \gg \gamma$ resulting in a very high autocorrelation of velocity. In a discrete simulation, as conducted in chapter 4, suppose a robot has velocity $\mathbf{v}(t_0) = (r_0, \theta_0)$ at time t_0, and velocity $\mathbf{v}(t_1) = (r_1, \theta_1)$ at time t_1. In the high friction limit where $C_{\mathbf{V}}$is almost zero, θ_1 is given by a uniform distribution on $[0, 2\pi)$, i.e. at each timestep the robot starts travelling in a new direction with probability 1. In the low friction limit where $C_{\mathbf{V}}$ is high, θ_1 is given by a Dirac distribution centred around θ_0, i.e. the robot essentially continues in the same direction it was travelling in before, unless it bounces off an obstacle.

2.1 High friction limit - Diffusion

If drag is strong compared to the mass of the particles $m \ll \gamma$, then equation (2.1) becomes the following SDE:

$$d\mathbf{X}(t) = \frac{\sigma}{\gamma}\boldsymbol{\xi}(t)dt \tag{2.4}$$

Equation (2.4) is the SDE for a Weiner process. The dynamics of the Wiener process are dictated by the forward Kolmogorov equation, or Fokker-Planck equation. The Fokker-Planck equation is a partial differential equation in the PDF, $p(\mathbf{x}, t)$ of $\mathbf{X}(t)$ [23]:

$$\frac{\partial p}{\partial t} = \frac{\sigma}{2\gamma}\nabla^2 p \tag{2.5}$$

Since $\mathbf{X}(t)$ is an ergodic Markov process, we know that there is a unique stationary distribution $p(\mathbf{x})$, such that $p(\mathbf{x}, t) \to p(\mathbf{x})$ as $t \to \infty$. As this distribution does not evolve over time, we can find it by setting $p_t = 0$, with the appropriate boundary conditions.

For a reflective boundary, we use the Neumann boundary conditions, requiring that the partial derivative in the direction of the circular boundary is zero. Writing $\mathbf{x} = (r, \theta)$, and expressing the Laplacian in polar coordinates, we have the following initial value problem:

$$0 = \frac{\sigma}{2\gamma}\left(p_{rr} + \frac{1}{r}p_r + \frac{1}{r^2}p_{\theta\theta}\right) \tag{2.6a}$$

$$p_r(R) = 0 \tag{2.6b}$$

However by the circular symmetry of the problem, we can see that any stationary distribution must have the same value $p(r) = p(r, \theta)$ for any θ. Hence the angular derivative p_θ is zero everywhere. Dividing (2.6a) through by a constant, and setting $p_{\theta\theta} = 0$, we have: $p_{rr} = -p_r/r$ an ordinary differential equation with solution: $p_r = c/r$ for some constant c.

Invoking the boundary condition (2.6b), we have that $c = rp_r = Rp_r(R) = 0$. Now since $p_r = p_\theta = 0$, we have $\nabla p = 0$ everywhere, so $p(\mathbf{x})$ is constant on the entire domain. Hence the stationary distribution for Brownian motion on a disc is uniform:

$$p(\mathbf{x}) = \frac{1}{\pi R^2} \tag{2.7}$$

Hence in the high friction limit when the velocity of robots has almost zero autocorrelation, the Langevin dynamics is an effective coverage algorithm, albeit a slow one. Hence for applications such as cancer treatment using nanobots, Langevin dynamics is strong solution. But what happens for robots with a high autocorrelation of velocity, in the low friction limit?

2.2 Low friction limit - Billiards

In the low friction limit of the Langevin dynamics, noise and drag are very weak. This causes the robots to travel in straight lines until the hit the edge of the arena, and then bounce off. We can see this by looking at the Langevin equations (2.1) that when $m \gg \gamma$ and $m \gg \sigma$ the acceleration of each robot is close to zero:

$$\begin{aligned}\frac{d\mathbf{V}(t)}{dt} &= -\frac{\gamma}{m}\mathbf{V}(t) + \frac{\sigma}{m}\boldsymbol{\xi}(t) \\ &\approx 0\end{aligned}$$

This makes the trajectory of an robot similar to a mathematical billiard that starts at an arbitrary point in the domain. Suppose that the billiard first hits the edge of the circle at angle ψ. Then each subsequent collision will also be at angle ψ.

Tracing a robot's trajectory for 50 collisions (see figure 2.2), we can see that it does not sample from the circle evenly. In fact there is a circular region in the middle that it never reaches. The edge of this reason is referred to as the caustic circle. We can see that its trajectory is most dense around the caustic circle, indicating that the billiard spends more time there than anywhere else in the domain.

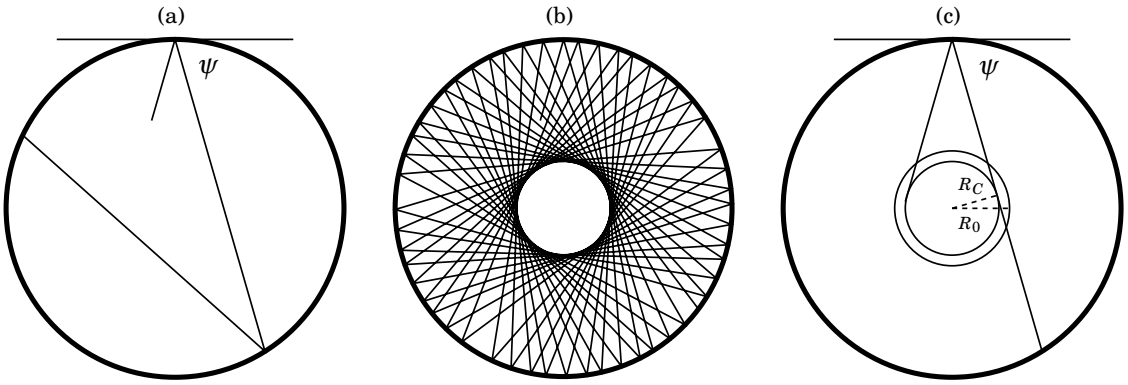


Figure 2.1: (a) The trajectory of a billiard incident at angle ψ. (b) The trace of a billiard as it bounces off the edge of the arena (c) Finding the radius of the caustic circle

From figure 2.1(a), we can see that the angle of incidence for the billiard remains the same, at all times. From figure 2.1(c) we can see that the caustic circle for any robot that begins its trajectory from within $D(\mathbf{0},R_0)$ will have radius $R_C < R_0$. Since the robots spend significantly more time on the caustic circle anywhere else, the stationary distribution for mathematical billiards is weighted slightly towards the edge of the circle due to the void in the centre, and hence is non-uniform.

This effect can be seen in the plots for the Langevin dynamics simulations (figure 4.2, top left plot). In cases with low drag coefficient $\gamma = 0.05, 0.1$ the radial displacement from the origin $r(t)$ converges to a level higher than $\frac{2}{3}R$, the average distance from the centre of a disc for points $(r,\theta) \in D(\mathbf{0},R)$.

2.3 Run and Tumble

There are other examples of algorithms where the robots act independently, and are driven by noise. Specifically, we might want to consider processes with sudden changes of motion, or jumps. Mathematically, this makes the velocity of the robots discontinuous through time, and hence it cannot be described as using a diffusion process [23] (p30). Such processes can be described using Lévy flights, a broader category of Markov processes that allow for jumps. To give an idea of how a Markov process with jumps might differ from the Langevin dynamics, we shall briefly consider a one example.

In Run and Tumble, robots 'run' in the same direction for a period of time given by a Poisson random variable $t \sim exp(\lambda)$, then 'tumble', picking a new direction uniformly at random $\theta \sim \mathcal{U}([0,2\pi))$. These two steps are repeated indefinitely. Run and Tumble is also of interest to microbiologists since it approximates the motion of e.coli bacteria.

The only parameters in the Run and Tumble algorithm that we can tweak are the tumble rate λ and the speed s of the robots. Note that since $t \sim exp(\lambda)$, $\mathbb{E}[t] = 1/\lambda$ so the expected distance

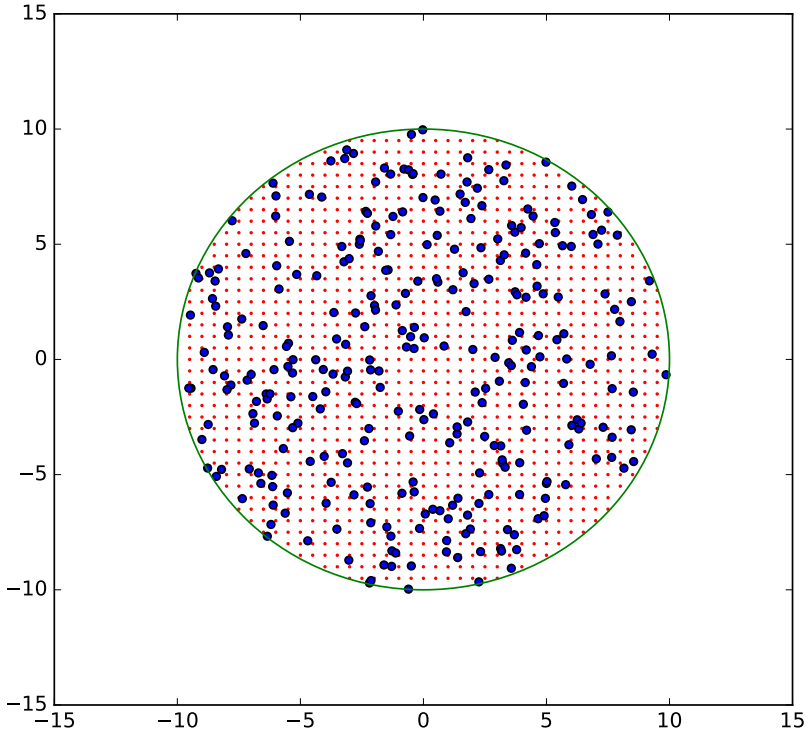


Figure 2.2: Simulation of robots covering a disc shaped area using Run and Tumble. Final robot positions are shown in blue. Distances to sample points (red) were recorded at each timestep.

travelled in a single run is s/λ. As mentioned in the previous section, we can think of inertia as the propensity of robots to move on a constant heading. Clearly if λ is small, robots will tumble rarely and their motion will be highly correlated. Conversely if λ is large, robots will continually change direction, and their motion will be highly fractal.

To put this another way, despite being defined very differently, Run and Tumble has the same limit cases as the Langevin dynamics. As $\lambda \to 0$, the particles do not tumble at all and behave like billiards. As $\lambda \to \infty$, the particles tumble constantly, and their motion becomes Brownian. Hence the stationary distribution for Run and Tumble should be uniform as $\lambda \to 0$, and non-uniform as $\lambda \to \infty$. A complete analysis of the dynamics of Run and Tumble robots can be found in [18], including the probability distributions over time. The boundary effects and diffusion coefficient are also derived.

The coverage distance for Run and Tumble in the stationary distribution is similar to that of the Langevin dynamics. This reflects the fact that with finite inertia, the distribution is approximately uniform. At the lowest tumble rate, robots have a high angle of incidence $\psi \approx \pi/2$ and bounce into the centre and out again in synch. We can also see that for higher tumble

rates, Run and Tumble takes longer to converge. This is due to the lower autocorrelation of $\mathbf{V}(t)$, resulting in a low diffusion coefficient D.

CHAPTER 3

DISTRIBUTION OF INTERACTING ROBOTS

By adding interaction between the robots, we can improve the coverage in two ways: firstly we can provide better coverage of the same area with the same number of robots; secondly, we can provide coverage faster. The main type of interaction that is useful for dispersion and area coverage is repulsion. In this section we examine the theoretical benefits of repulsion, a model for electrostatic repulsion, and the how noise affects the outcome. Simulations indicate that simple local repulsion offers close to optimal performance.

3.1 Coverage improvements for repulsive robots

Clearly if robots are able to repulse one another, the coverage distance ought to improve. But by how much? By applying some results from probability theory, we can find lower bounds for interacting and non-interacting robots. When the arena is a circle, the problem of finding the minimum coverage distance for n robots bears striking similarity to the disc covering problem. Suppose we place a disc of radius r centered at the base of each robot. What is the smallest such r for which the smaller discs cover the entire domain: (a) If the robots are placed at optimal positions? (b) If robots are placed at positions sampled uniformly at random?

The disc covering problem is a well known recreational maths problem. For a few small values of n, the ratio r^2/R^2 has been calculated, but there is no general solution [32]. There is, however a limit result for large n which gives us the ratio r^2/R^2. Let $k(r)$ be the smallest number of disks of radius r needed to cover a disk $\Omega = D(\mathbf{0},R)$. The limit of the ratio of the area of Ω to the area of the discs is given by:

$$\lim_{r\to 0^+} \frac{r^2 k(r)}{R^2} = \frac{2\pi}{3\sqrt{3}} \tag{3.1}$$

This result is established in [15] by tiling the larger disc with hexagons, then calculating the overlapping area. The overlapping area is ratio of the area of a disc of radius r to the area of a perfect hexagon with sides of length r.

Suppose that we wanted to find the configuration of robots that minimises the coverage distance C. As suggested above, when n is large, the best strategy is to tile the arena with regular hexagons of side length r, and place one robot at the centre of each hexagon. But what is the best value of r that we can achieve? Applying the result above, we get:

$$\frac{r^2 n}{R^2} = \frac{2\pi}{3\sqrt{3}} \tag{3.2}$$

Setting $h = \sqrt{\frac{2\pi}{3\sqrt{3}}}$ and solving for r, we have:

$$C > r = \frac{hR}{\sqrt{n}} \tag{3.3}$$

This is the lower bound for interacting robots.

Now lets look at the lower bound for non-interacting robots. As described in section 2, the best stationary distribution that can be achieved for independent robots is to sample uniformly at random from the domain Ω. Imagine that the domain were split into discrete cells of the same size, so that each robot is equally likely to be in any cell. Given n robots, what is the maximum number of cells that we could split the domain into so that there is at least one robot in each cell? Does this problem seem familiar? It is in fact the inverse of the Coupon Collector problem.

The Coupon Collector problem is usually stated thus: suppose I am collecting coupons (or football stickers) of different types (players) that I buy from a shop where the odds of buying each type of coupon (football player) are the same each time I get a new one. How many should I expect to buy before I have a complete set? To put it another way, suppose that there are k types, and $\mathbb{E}(N_i)$ is the number of coupons I have to buy to get the i-th different type of coupon, given that I already have $i-1$. What is the value of $\mathbb{E}(N)$, the expected number of coupons I need to buy to get all k different coupons?

The probability of getting a new coupon is geometric, with $\mathbb{P}(n_i) = (k-(i-1))/k$. Hence $\mathbb{E}(N_i) = k/(k-(i-1))$. By the linearity of expectation, we have:

$$\begin{aligned}\mathbb{E}(N) &= \mathbb{E}(N_1) \ + \ \mathbb{E}(N_2) \ + \ \ldots \ + \ \mathbb{E}(N_k) \\ &= \frac{k}{k} \ + \ \frac{k}{k-1} \ + \ \ldots \ + \ \frac{k}{1} \\ &= k \cdot H_k\end{aligned}$$

where H_k is the k-th harmonic number. In the limit as $k \to \infty$ we have that:

$$k \cdot H_k \ \to \ k \log k + \gamma k + \tfrac{1}{2} + O(\tfrac{1}{k}) \tag{3.4}$$

Where $\gamma \approx 0.5772$ is the Euler-Mascheroni constant. So for large k, we can make the following approximation:

$$\mathbb{E}(N) \approx k \log k + \gamma k + \tfrac{1}{2} \tag{3.5}$$

For our problem, however, we don't want to know how many robots n we need to have one in each of k cells. For a given number of robots n, we want to know how many cells k we can expect to occupy. To calculate this, we need to solve (3.5) for k. The inverse of (3.5) is given by:

$$k(n) = \frac{n - \frac{1}{2}}{W(e^{\gamma}(n - \frac{1}{2}))} \tag{3.6}$$

where $W()$ is the Lambert-W function, which is defined on the complex plane as the inverse of $f(z) = ze^z$, and calculated using a series approximation.

So given that we have n robots, we can split the arena into $k(n)$ cells, with at least one robot in each cell. As we established earlier, for large k, the best way of achieving this is to tile the arena hexagonally with side length r. In such a configuration, the ratio of area covered by cells of radius r to the total area was given by (3.1):

$$\frac{r^2 k(n)}{R^2} = \frac{2\pi}{3\sqrt{3}} \tag{3.7}$$

Note that when the number of cells k is large, given a particular point $\mathbf{y}$ we can redraw the cells in such a way that $\mathbf{y}$ is at the centre of a cell, so each point is at most r away from the nearest robot. Setting $h = \sqrt{\frac{2\pi}{3\sqrt{3}}}$. and solving for r, we find the lower bound for independently acting robots:

$$C > r = \frac{hR}{\sqrt{k(n)}} \tag{3.8}$$

Since $k(n) < n$ for large n, we can see that the lower bound for non-interacting robots (3.8) is higher than the general lower bound for robots that may interact with one another (3.3). Hence in order to get the best results we need to add some interactions!

For optimal coverage, useful interactions are likely to involve repulsion a of some kind, effective at a distance that must exceed lower bound for interaction (3.3). For instance, in the case of our optimal hexagonal lattice for large n, which has a side length of $C_{\min} = hR/\sqrt{n}$, to communicate with a neighbour, a robot would have to interact at a distance of:

$$d = 2 \cdot \frac{\sqrt{3}}{2} \cdot C_{\min} = \sqrt{3} h \cdot \frac{R}{\sqrt{n}} \tag{3.9}$$

In the next two section we consider two types of repulsion: electrostatic and local.

3.2 Electrostatic repulsion

The electrostatic repulsion algorithm is from a classic swarming paper by Howard et al. which describes how to distribute robotic sensor networks inside buildings [13]. Each robot is modelled as a charged particle that repels all the others according to Coulomb's law. Robots act deterministically, but start in random locations on a smaller disc in the centre of our target area. In order to prevent them from spreading out towards the walls, there is a boundary force that repels them from the wall.

The robots are modelled as particles with mass m and charge q that move according to Newton's laws of motion, subject to the forces of drag and electrostatic force:

$$\ddot{\mathbf{x}}_i = \frac{\mathbf{F}_c + \mathbf{F}_d + \mathbf{F}_b}{m} \tag{3.10}$$

Electrostatic force $\mathbf{F}_e$ is governed by Coulomb's law, i.e. force is proportional to the inverse square of the distance between particles $\mathbf{r}_{ij} := ||\mathbf{x}_i - \mathbf{x}_j||$. Drag $\mathbf{F}_d$ is linear, with viscosity v. The boundary force $\mathbf{F}_b$ is the force acting on a particle due to a charged ring of radius R,

$$\mathbf{F}_d = -\gamma \dot{\mathbf{x}}_i \tag{3.11a}$$

$$\mathbf{F}_c = -k_e \sum_{j \neq i} \frac{q^2}{||\mathbf{r}_{ij}||^2} \hat{\mathbf{r}}_{ij} \tag{3.11b}$$

$$\mathbf{F}_b = f(||\mathbf{x}_i||, R)\ q\ \hat{\mathbf{x}}_i \tag{3.11c}$$

where $f(r,s)$ is defined below.

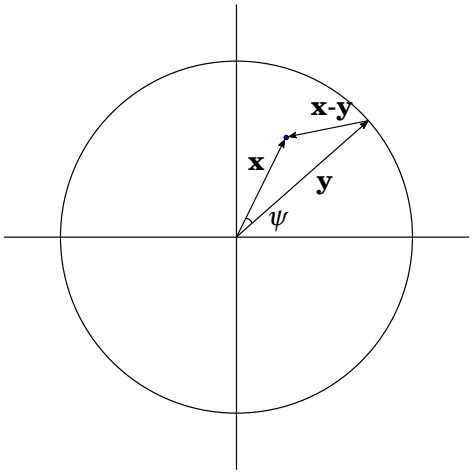
In the continuum limit, the particles are modelled as a charge distribution with density $\rho(\mathbf{x})$. In order to find the stationary distribution of the particles we look for the equilibrium distribution where the electric flux at each point is zero. Drag is zero when the distribution is static, so does not affect the equilibrium distribution. Hence to achieve electrostatic equilibrium, we require $(\mathbf{F}_c(\mathbf{x}) + \mathbf{F}_b(\mathbf{x})) \cdot \hat{\mathbf{n}} = 0$ for all points $\mathbf{x}$ in the disc, and any unit vector $\hat{\mathbf{n}}$.

Due to the radial symmetry of the disc and the boundary force $\mathbf{F}_b$, can tell that the flux going in all directions other than radially outwards is zero. Hence we just need to find a charge distribution $\rho(\mathbf{x})$ that satisfies:

$$\mathbf{F}_c(\mathbf{x}) \cdot \hat{\mathbf{x}} = -\mathbf{F}_b(\mathbf{x}) \cdot \hat{\mathbf{x}} \tag{3.12}$$

Consider the forces acting on a charged particle at position $\mathbf{x} = (r, \theta)$, due to a charged ring of radius $||\mathbf{y}|| = s$. Integrating around the ring, we can work out the net force $f(r,s)$ acting on $\mathbf{x}$ due to the ring at s:

$$\begin{aligned} f(r,s) &= \int_{||\mathbf{y}||=1} \frac{\rho(\mathbf{y})(\mathbf{x}-\mathbf{y})}{||\mathbf{x}-\mathbf{y}||^3} \cdot \hat{\mathbf{x}}\ d\mathbf{y} \\ &= \rho(s) \int_0^{2\pi} \frac{r - s\cos\psi}{(r^2 + s^2 + 2rs\cos\psi)^{3/2}}\ d\psi \\ &= \frac{2\rho(s)}{r} \operatorname{sign}(r-s) \Big(\frac{K}{r-s} + \frac{E}{r+s} \Big) \end{aligned}$$



$\mathbf{x} = (r\cos\theta, r\sin\theta)$
$\mathbf{y} = (s\cos\phi, s\sin\phi)$
$\psi = \theta - \phi$

Figure 3.1: Finding the force at a point **x** from a charged ring of radius s

where K and E are the complete elliptic integrals of the first and second kind, applied to the argument $\frac{-4rs}{(r-s)^2}$. This integral can be derived either by hand or using a symbolic integrator. Clearly bounding force F_b is given by the force acting from a charged ring at radius R, however we can think of the repulsive force due to the charge distribution ρ as an integral of charged rings at radius s:

$$F_b(r) = f(R,r)$$
$$F_c(r) = \int_0^1 \rho(s) f(r,s) ds$$

We can approximate this integral as a sum:

$$F_c(r_i) \approx \sum_{j=0}^{J} \rho(s_j) f(r_i, s_j) \tag{3.13}$$

Or in vector notation: $\mathbf{c} = F\boldsymbol{\rho}$, where $[F_{ij}] = f(r_i, s_j)$. We can also write the boundary force as **b**, where $[b_i] = f(R, r_i)$. Hence in order to estimate the equilibrium distribution, we just need to invert the F matrix:

$$\boldsymbol{\rho} = F^{-1}\mathbf{c} = -F^{-1}\mathbf{b} \tag{3.14}$$

We computed the inverse of the F matrix using MATLAB, and calculated $\boldsymbol{\rho}$. The resulting distribution is shown in figure 3.2(a).

That the distribution given in 3.2(a) is not uniform came as somewhat of a surprise to the author. We had initially thought that for a large number of particles electrostatic, or indeed in the continuum limit, there would be a uniform distribution. However, on computing the distribution and running the simulations we found that the stationary distribution has a will in the centre, where the number of particles is low.

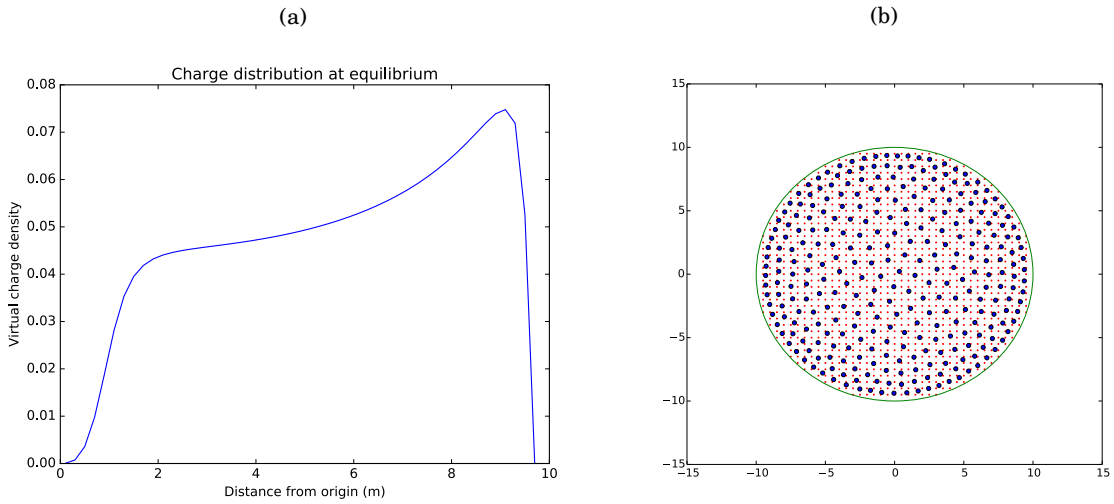


Figure 3.2: (a) Circularly symmetric charge distribution with density ρ at distance r from origin. (b) Unusual non-uniform distribution of repulsive robots following Coulomb's law, at the end of a simulation.

On reflection, however this makes more sense than it would first appear. Particles in the centre are surrounded on all sides, so cast an outward force in all directions that influences all other particles. Particles close to the rim however, cast an inward force that only has influence within the nearside of the ring, and in fact contribute marginally to the outward force on the other side. Hence the overall outward forces are greater than the overall inward forces, unless the confining force of the ring is large.

3.3 Local Repulsion

Electrostatic repulsion has a few drawbacks. As we have just seen, it only approximates the uniform distribution, and also requires very fast acceleration for nearby robots. This can be problematic in a robotics context where robots have a top speed. However most importantly, as stated, it is not truly decentralised. Each robot needs to interact with every other robot in order to compute its acceleration vector. To address some of these shortcomings, we wanted to consider a simpler algorithm.

The local repulsion algorithm is defined by robots accelerating away from one another at a constant rate a whenever they come within a distance d. We can model this as:

$$\ddot{\mathbf{x}}_i = \frac{\mathbf{F}_c + \mathbf{F}_d + \mathbf{F}_b}{m} \tag{3.15}$$

where $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ as above, and:

$$\mathbf{F}_c = \sum_{\{j:\mathbf{r}_{ij}<d\}} -a\hat{\mathbf{r}}_{ij} \tag{3.16a}$$

$$\mathbf{F}_d = -\gamma\dot{\mathbf{x}} \tag{3.16b}$$

$$\mathbf{F}_b = \begin{cases} -a\hat{\mathbf{x}}_i, & \text{when } (R - ||\mathbf{x}_{ij}||) < d \\ \mathbf{0}, & \text{otherwise} \end{cases} \tag{3.16c}$$

The main advantage of local repulsion over electrostatic repulsion is that we lose the non-linear effects that cause very high acceleration for nearby robots, and low acceleration for robots that are slightly further away. If we think about this in light of robots attempting to avoid each other, it means we can start accelerating away from our neighbours as soon as we spot them, vacating their sphere of influence. This, in turn, means we can reduce the communication requirements by limiting all interactions to robots within each others sphere of influence.

But what distance of repulsion d yields the best coverage distance? Based on the bound for the minimal coverage distance (3.3), we can calculate sensible values for d. We have seen from [15] that for large n, the optimal disc tiling is hexagonal. In a hexagonal tiling with side length 1, the distance between centres of neighbouring hexagons is $\sqrt{3}h$. Hence from equation (3.9), we need to set the repulsion distance to $\sqrt{3}h \cdot R/\sqrt{n}$ to ensure that robots repel one another when they enter their sphere of influence, but do not repel one another in equilibrium state.

Normalising by the size of the arena and number of particles, set the the parameter $\eta = d/(R/\sqrt{n})$. We ran simulations with 200 robots at varying normalised distance of repulsion η. We ran 40 simulations for double the number of timesteps required for the robots to reach a stationary distribution (tested empirically, see chapter 4), calculating the mean coverage distance $\langle C(t)\rangle$ for t in the second half of each sample.

Plotting the results in the range $\eta \in (1,4)$, we find a minimum coverage distance at $\sqrt{3}h = 1.9046...$ as expected. More unexpected is the nonlinear response of coverage distance $C(t)$ to the tuning of the normalised repulsion distance η. This is a result of the geometry of the hexagonal lattices of robots are reached for different values of η. For $\eta < 1.9$, robots move freely, repulsing each other (fig. 3.3(a)), until the point $\eta \approx 1.9$, where they reach a stable hexagonal lattice with one robot at each lattice point (fig. 3.3(b)). Increasing η above 1.9, increases $C(t)$ as robots start to pair up, up to the value of $\eta \approx 2.6$ where a lattice is formed with two robots at each lattice point (fig. 3.3(c)). As η increases again symmetry is broken until $\eta \approx 3.2$, where robots form a lattice with three robots at each lattice point. This remains stable for longer, with the occasional group of four, until symmetry is again broken at $\eta \approx 3.7$, and so on.

Note that the mathematical lower bounds for coverage in interacting robots are only attainable for large n, and are not achievable for finite values of n. This is a result of inefficiencies in coverage close to the boundary, where robots have to bunch to cover edge points. Increasing the number of robots produces results that get closer the lower bound, as number of edge robots is small

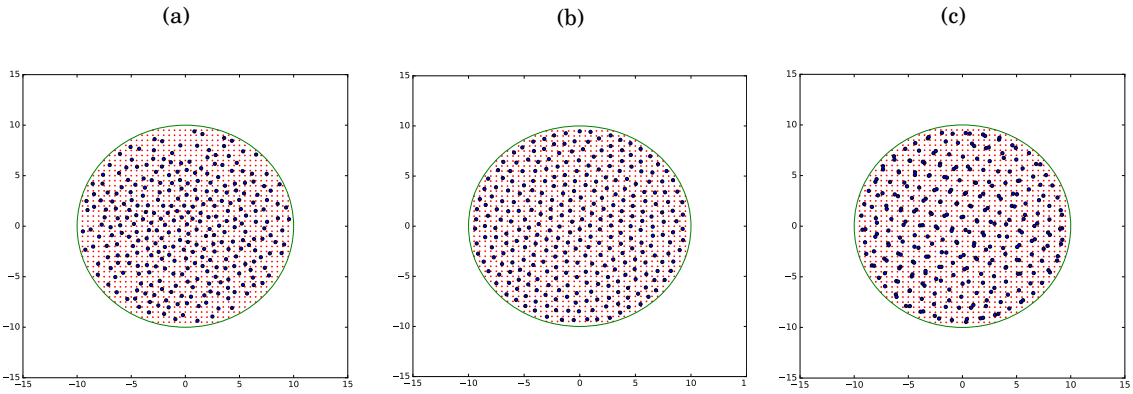


Figure 3.3: Three simulations of robots covering a disc using the local repulsion algorithm with different radii of repulsion. Final configuration of robots is shown in blue (a) $\eta < \sqrt{3}h$ (b) $\eta = \sqrt{3}h$ (c) $\eta > \sqrt{3}h$

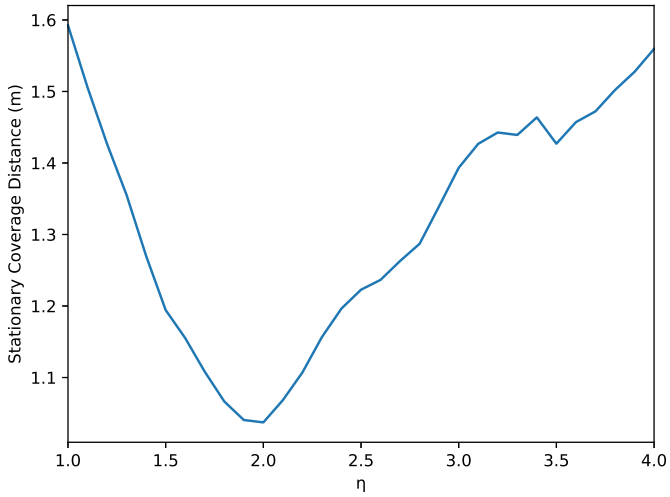


Figure 3.4: Mean Coverage distances $\langle C(t) \rangle$ at different values of normalised repulsion distance η once a stationary distribution has been reached in the linear repulsion algorithm

compared to n. Unfortunately the non-linear nature of the local repulsion function (3.16c) makes it hard to analyse mathematically. As far as the author are aware, proving (or indeed disproving) that for large n, this algorithm converges to an equilibrium where robots are distributed on a hexagonal lattice with an optimal coverage distance given by (3.3) at $\eta = \sqrt{3}h$ is an unsolved problem in non-linear dynamics.

CHAPTER 4

SIMULATION RESULTS

We performed a Monte Carlo simulation to estimate how long it takes the robots to reach the stationary distribution, and what the coverage is like in the stationary distribution. The easiest way to observe this is to measure the coverage distance $C(t)$ over time over a significant number of samples. The source code for the simulations is open source and written in python 2. It can be cloned from [33].

As explained in chapters 2 and 3, the two factors that affect the distribution and speed of convergence are the autocorrelation (or inertia) of the robots, and the type and magnitude of interactions between the robots.

4.1 Simulation methodology

Simulations were carried out in Python, the code for which is published in the open source project 'particles' `https://github.com/samyoung17/particles` [33]. The project consists of a particle simulator **particlesim.py**, a series of algorithm scripts, and the **coverage.py** script which executes a series of 20 simulations of each algorithm specified in the **coverageconfig.py** file. Instructions for how to build and run the project can be found in **README.md**. Selected python files are also included in the appendix.

During each simulation, the particles are started in a random configuration within the starting circle $D(\mathbf{0}, r)$, and moved according to the rules specified by the relevant algorithm script. Movements and boundary effects are specified using polymorphic functions, with the simulations of each algorithm running on a different thread to reduce the overall running time of the simulation.

The trajectories of the robots for the most recent simulation is saved in the 'data' folder of the root directory and the output of each simulation is saved in the 'results' folder of the root

directory, in the form of a .csv file containing the coverage distances for each algorithm and a plot containing the same data. An animation of the most recent simulation of each algorithm can be viewed by executing the **particlesim.py** script with the path to the data folder for that algorithm and an integer speed multiplier.

The project includes scripts for the Metropolis-Hastings algorithm, the Langevin dynamics, Run and Tumble, Voronoi Descent, Electrostatic repulsion, Linear repulsion, and the **electrostaticlangevin.py** script, which simulates both electrostatic repulsion and linear repulsion with noise depending on the α coefficient which specifies the exponent of the power law for repulsion. It permits both hard boundaries that the particles bounce off, and soft boundaries that repulse the particles according to a power law. It is also possible to specify different shapes of boundary in the **coverageconfig.py** file.

As would be expected when performing calculations on a Turing machine, the continuous trajectory of each particle must be approximated as a sequence of vectors $(\mathbf{x}_i)_t$ of floating point numbers. During the simulations, these vectors are saved in memory-mapped files. Since they are very large, only the coverage distances for each simulation are saved for comparison. The timestep resolution for the simulations is 0.25s, which offers a reasonable compromise between smoothness and runtime performance.

4.2 Langevin Dynamics Simulation

The main factor driving how quickly independent robots cover the disc is the autocorrelation of the Langevin path. As discussed in chapter 2, the γ parameter dictates the autocorrelation of the Langevin path, with $\gamma = 1$ resulting in ballistic motion tracing the caustic paths within the circle, and $\gamma = 0$ resulting in Brownian motion. For the first set of simulations, we wanted to discover the effect of different values of γ on the coverage time of the robots. In order to have a sensible benchmark, we wanted to set the equilibrium velocity to a constant value.

In the Langevin dynamics, the velocity of the particle $\mathbf{v}(t) \sim \mathbf{V}(t)$ is an Ornstein-Uhlenbeck process. The stationary distribution of an Ornstein-Uhlenbeck process is Gaussian, with mean 0 and standard deviation D/α, where D is the diffusion coefficient of of the noise term $\boldsymbol{\xi}(t)$, and α is the coefficient of $\mathbf{V}(t)$ [23] (p39).

$$\frac{d\mathbf{V}(t)}{dt} = -\frac{\gamma}{m}\mathbf{V}(t) + \frac{\sigma}{m}\boldsymbol{\xi}(t) \tag{4.1}$$

Setting $m = 1$, we have $\alpha = \gamma$ and $D = \sigma^2$. Hence the stationary distribution of the Langevin dynamics in this case is given by $\mathbf{V}_{eq}(t) \sim \mathcal{N}(\mathbf{0}, \sigma^2/\gamma)$. In such a distribution, the mean squared speed $\langle s^2 \rangle$ is given by:

$$\langle s^2 \rangle = \mathbb{E}\left[\left|\left|\mathbf{V}_{eq}(t)\right|\right|^2\right] = \frac{\pi}{2} \cdot \frac{\sigma^2}{\gamma} \tag{4.2}$$

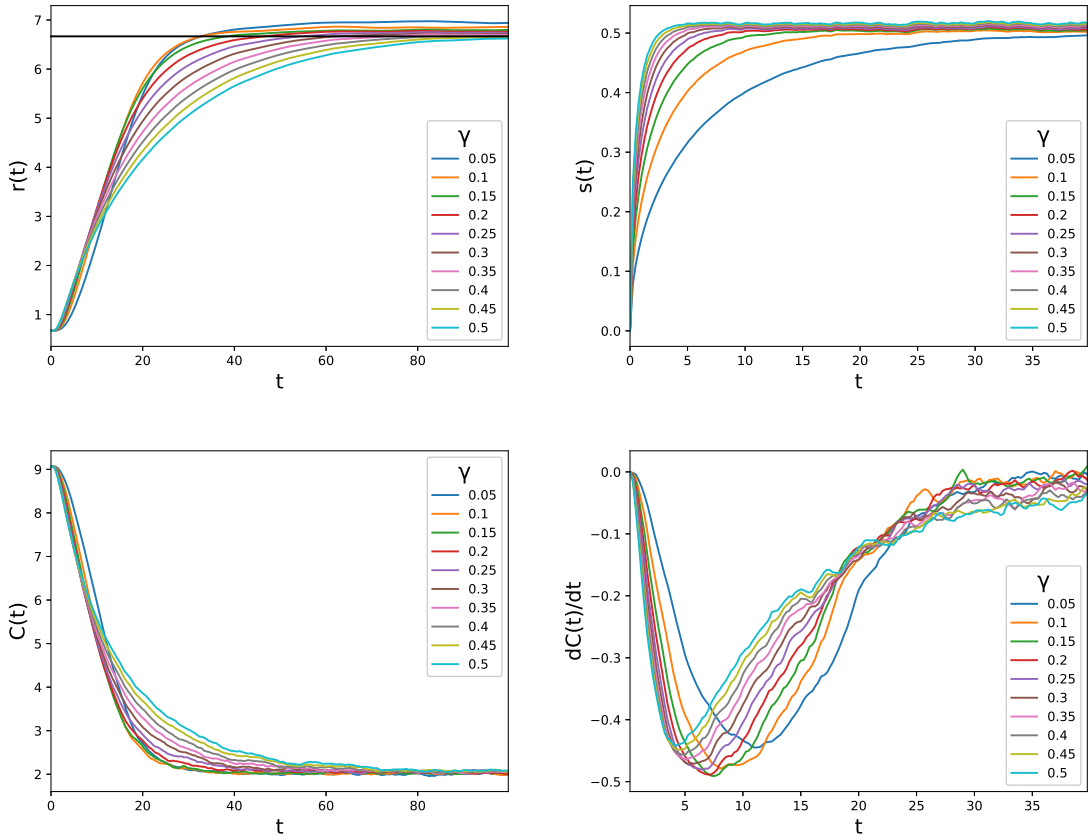


Figure 4.1: Plots showing distance from the centre $r(t)$, speed of robots $s(t)$, coverage distance $C(t)$ and the change in coverage distance $dC(t)/dt$ through time for the Langevin dynamics simulation with different drag coefficients γ.

where the factor of $\pi/2$ is introduced due to the area element. Hence, to ensure a consistent speed s at equilibrium between different simulations, we need to set $\sigma^2 = 2\langle s^2 \rangle \gamma / \pi$.

Setting $\langle s^2 \rangle = 0.25$, we measured different summary statistics for 200 simulations with 200 robots each in simulation. Figure 4.2 shows several key statistics, including mean distance from the origin $r(t) := \langle ||\mathbf{x}(t)|| \rangle$, mean speed $s(t) := \langle ||\mathbf{v}(t)|| \rangle$, mean coverage distance $C(t)$, and the change in mean coverage distance $dC(t)/dt$ across simulations. We can see that overall, regimes with high autocorrelation (and hence low drag) result faster convergence to the equilibrium coverage distance $C(t)$ for that regime.

The dynamics depicted in these charts show important features of robots moving according to the Langevin dynamics, bounded within a disc. Most strikingly, the cyan curve for coverage distance $C(t)$ with the highest drag coefficient $\gamma = 0.5$ (and therefore least autocorrelation of motion) traverses all the other curves as a function of time. Looking at the speed chart (top right), we can see that this is because it reaches equilibrium speed fastest, giving it a quick initial expansion. Looking at the radial distance chart (top left), we can see that despite speeding up

quickly, robots do not manage to move away from the origin as quickly under a high-drag regime.

The Stokes-Einstein equation states that when the velocity of a Langevin particle of unit mass on the plane reaches a stationary distribution, the diffusion constant for $\mathbf{X}(t)$ is $D = \langle s^2 \rangle / \gamma$, or to put it in terms of the mean squared displacement:

$$||\mathbb{E}[(\mathbf{X}(t) - \mathbf{X}(0))^2]|| \to 2\frac{\langle s^2 \rangle}{\gamma} t \quad \text{as} \quad t \to \infty \tag{4.3}$$

Furthermore, the Kubo relation states that the mean squared displacement of the Langevin particle is equal to the time integral of the autocorrelation of the velocity. We can apply these results to understand why our curve for $\gamma = 0.5$ intercepts the curves of all the other values at different times.

Equilibrium of speed is reached fastest for $\gamma = 0.5$. This is because we defined $\sigma^2 = 2\langle s^2 \rangle \gamma / \pi$ in this simulation, so at the beginning when $\mathbf{V}(t)$ is close to zero, the robots accelerate much faster than they are slowed down by drag. This causes $C(t)$ to drop at the beginning of the simulation for low values of γ. However, once the robots reach equilibrium velocity, but are still sufficiently far away from the boundary, their mean squared displacement from the origin will be around one tenth of robots in the $\gamma = 0.05$ case across the same time interval, due to the Stokes-Einstein equation. This effect can be seen in the top left plot for $r(t) := \langle ||\mathbf{x}(t)|| \rangle$, and its corresponding impact on the coverage distance plot $C(t)$, where the cyan curve for $\gamma = 0.5$ is the slowest to converge overall.

As the Stokes-Einstein equation only applies in free space, similarities start to break down as robots approach the edge of the arena, where boundary effects come into play. We can also see from radial distance $r(t)$ plot the effect of the caustics discussed in section 2.2. A uniform distribution of robots should yield an average radial distance of $r(t) = \frac{2}{3}R = 6\frac{2}{3}$, the black line on the chart. In fact we can see that $r(t)$ is higher than $\frac{2}{3}R$ when γ is low. This is caused by the fact that in low drag regimes, we approach the distribution of a robot travelling caustic, which has higher density at the edge of the circle than in the centre.

We can measure the efficiency of the Langevin dynamics algorithm for robots covering a disc with different levels of autocorrelation of motion by measuring the amount of time it takes for $C(t)$ to approach the lower bound given by (3.8) in the simulations. As the stationary distribution of robots is not uniform,and the mathematical lower bounds may be attainable for finite n, we may not be able to meet the lower bound. Hence we measure the amount of time it takes to get within ϵ:

$$CT_{\text{indep}}(\epsilon) := \min\left\{t | C(t) < \tfrac{hR}{\sqrt{k(n)}} + \epsilon\right\} \tag{4.4}$$

this quantity is called the coverage time.

Based on the same set of simulations, we can plot the coverage time for $\epsilon \in \{2^{-1}, ..., 2^{-5}\}$ at different values of γ. We can see that coverage times are low when autocorrelation of velocity is high. As predicted by the Stokes-Einstein equation, the coverage time increases approximately

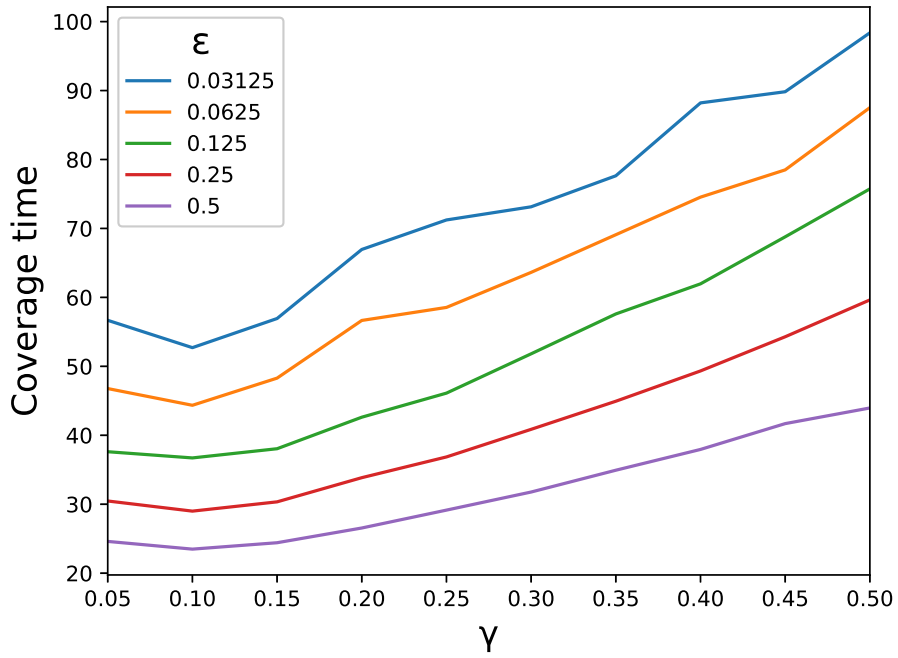


Figure 4.2: Average coverage time $\langle CT_{\text{indep}}(\epsilon)\rangle$ of independent langevin agents for different values of γ and ϵ

linearly with γ. The curves flatten off when γ is close to zero, due to the fact that the time taken to reach the equilibrium speed $\langle s\rangle$ is comparable to the overall coverage time. This is visible in the $s(t)$ and $dC(t)/dt$ plots of figure 4.2, where the blue and orange curves for $\gamma = 0.05, 0.1$ lag behind the others. Equilibrium speed is not reached fast enough for the robots with $\gamma = 0.05$ to benefit from diffusing faster at equilibrium than their competitors with $\gamma = 0.15$.

4.3 Local Repulsion Simulation

We would like to set up a toy local linear repulsion system, where we can investigate the impact of interactions on the coverage distance reached, and the amount of time it takes robots to reach it. Ideally we would like to find some parameter ϕ that describes the strength of interactions relative to noise. i.e. setting $\phi = 1$, should yield a simulation in which all motion is driven by the forces of local repulsion and the bounding ring, and setting $\phi = 0$ should yield a simulation such that all motion is driven by the random perturbations.

However, this poses a problem. When robots are clustered in the centre, all forces point outwards, so the net force robots exert on each other is strong, and will lead to significant acceleration. Conversely, once robots reach the hexagonal lattice (which we hypothesis to be their equilibrium state), the sum of forces that neighbouring robots exert on each other, plus

the confining force of the ring will close to zero. This means most motion will be a result of random perturbations. Given the constraints of this issue, what is the fairest way to set up the parameters in order to yield comparable simulations?

To be specific, the parameters we have to select are the strength of interactions α, the repulsion distance d, the mass m, the drag coefficient γ, and the strength of perturbations σ. Evidently there are many choices of such parameters, and each will yield different behaviour over time. So lets begin with some easy choices. As before, we can normalise the mass to $m = 1$ and scale the forces instead σ and α instead without limiting ourselves. Following the discussion in chapter 3, we can set the repulsion distance to $d = \sqrt{3}h \cdot R/\sqrt{n}$, so that robots repel each other except when they form a hexagonal lattice that covers the disc.

As to the strength of the interactions, it seems that as suggested above, when we properly calibrate d, it will not be constant through time. The fairest way I could think to do this, was to imagine a situation in which robots were distributed uniformly at random around the disc, and ask what the sum ϕ of the magnitudes of all forces due to interaction acting on a single robot was.

$$\begin{aligned}\phi &= \alpha \cdot n \frac{\pi d^2}{\pi R^2} \\ &= \alpha \cdot n \frac{\pi \cdot (\sqrt{3}hR/\sqrt{n})^2}{\pi R^2} \\ &= \alpha \cdot 3h^2\end{aligned}$$

Hence to reach a situation where the sum of magnitudes of forces adds up to ϕ, we need to set $\alpha = \phi/3h^2$.

Now, in the Langevin dynamics case with interaction, we wanted to set σ^2 in order that the mean squared speed was $\langle s^2 \rangle = 0.25$. We calculated that in order to achieve such an equilibrium speed, we needed to set $\sigma^2 = 2\gamma\langle s^2 \rangle/\pi$. When $m = 1$, this is equivalent to a force due to perturbations of strength $\sigma = \sqrt{2\gamma\langle s^2 \rangle/\pi}$. Now imagine that our choice of $\alpha = \phi/3h^2$, led to an average force due to interactions of ϕ. Then to balance the system with a mean squared speed of $\langle s^2 \rangle$ we should set:

$$\alpha = \frac{\phi}{3h^2}\sqrt{\gamma\frac{2\langle s^2 \rangle}{\pi}} \tag{4.6a}$$

$$\sigma = (1-\phi)\sqrt{\gamma\frac{2\langle s^2 \rangle}{\pi}} \tag{4.6b}$$

Of course this is wishful thinking, since as stated before, this choice of α merely balances the sum of magnitudes of forces in a situation in which robots are distributed uniformly at random on the disc! In general robots are not distributed uniformly at random, either at the start or in equilibrium state, and repulsive forces do not all act in the same direction, so the sum of magnitudes does not equal the magnitude of the sum. Nonetheless, empirically, I have found that

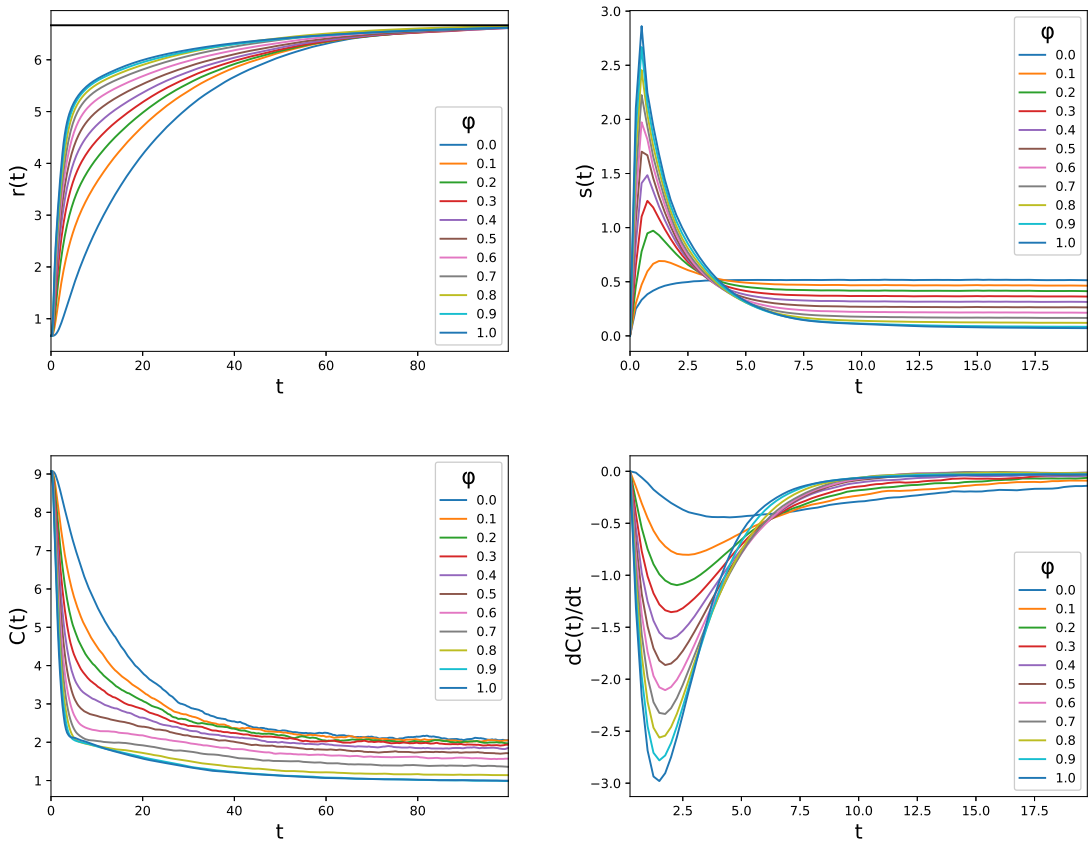


Figure 4.3: Plots showing distance from the centre $r(t)$, speed of robots $s(t)$, coverage distance $C(t)$ and the change in coverage distance $dC(t)/dt$ through time for the Langevin dynamics with local linear repulsion of strength paramatrised by ϕ.

this parametrisation leads to sensible simulations with a number of interesting features, so I present the results below.

In figure 4.3, we can see the dynamics of the coverage distance for the linear repulsion algorithm with noise, with parameters set to $m = 1$, $\gamma = 0.5$, $d = \sqrt{3}h \cdot R/\sqrt{n}$, and α, σ set as in (4.6), for values of $\phi \in [0,1]$. We can see that when $\phi = 0$, we get the same curve as in the Langevin dynamics simulation for $\gamma = 0.5$. As we increase ϕ we get lower overall coverage distances $C(t)$, and a faster downward trend in coverage distance through time.

We can see from the top right chart that this due to higher speeds $s(t)$ in the first five seconds, driven by a strong repulsive forces at the beginning of the simulation, where all agents are bunched together in the central starting area $D(\mathbf{0}, R_0)$. Interestingly, there is an inflection point in terms of speed around $t = 3$, where agents are sufficiently far apart that the force due to random perturbations $\boldsymbol{\xi}(t)$ starts to become stronger than the force due to repulsion. For values of $\phi > 0$, this inflection point marks a phase transition from the initial explosion to the steady state where an equilibrium speed will be reached, based on the average magnitude of the perturbations

σ.

We can see in the top right chart that the speed reaches $s(t) \approx (1-\phi)\langle s \rangle$ in steady state. This is due to the fact that we tuned d such that once the robots reach a hexagonal lattice, net force acting on each robot will be close to $\mathbf{0}$. Though there is some acceleration attributable to oscillations, since drag is relatively high in these simulations, this effect is minimal, and mostly limited to very low noise scenarios where $\phi = 1$. Therefore it comes at no surprise that as we discounted σ by $(1-\phi)$ from the value it would have needed to be to reach an equilibrium speed of $\langle s \rangle$, the equilibrium speed would be close to $(1-\phi)\langle s \rangle$.

We can see the effects of this faster initial speeds on $C(t)$ in the coverage dynamics chart $dC(t)/dt$ in the bottom right of figure 4.3. This resembles a family of upside-down gamma distributions, of a similar scale, but with larger shape parameters for ϕ close to 0, yielding a higher spike and shorter tail. Interestingly, we can see from the top right chart for $r(t)$ that all values of ϕ approach the average radial distance at around the same time, indicating that convergence to the stationary distribution of $r(t)$ takes a similar length of time overall, though from the chart for $C(t)$ we can see that the overall levels of coverage are better i.e. lower for values of ϕ close to 1, and therefore $C(t)$ drops faster in order to reach a lower base level over the same time.

As for the Langevin dynamics, we can see this effect by plotting the amount of time it takes for $C(t)$ to come within ϵ of the analytic lower bound for interactive agents, given in (3.3). We define the coverage time for interactive agents at ϵ from the lower bound as:

$$CT_{\text{inter}}(\epsilon) := \min\left\{t | C(t) < \tfrac{hR}{\sqrt{n}} + \epsilon\right\} \tag{4.7}$$

Again, due to boundary effects, and the fact that (3.3) applies only for large n, we can't hit it exactly, however in figure 4.3, we plot coverage times $CT_{\text{inter}}(\epsilon)$ for $\epsilon \in \{2^{-1}, ..., 2^{-5}\}$. Nonetheless, we come within $\epsilon = 2^{-5}$ for the simulations with high values of ϕ, indicating that we are very near to the optimal solution. Though we haven't been able to derive the stationary distribution of robots analytically, I would conjecture that the local linear repulsion with $d = \sqrt{3}h \cdot R/\sqrt{n}$ yields a hexagonal configuration, which turns out to be optimal for large n.

4.4 Conclusions

These results demonstrate that it is possible to achieve near-optimal coverage of wide areas with relatively simple algorithms that involve maintaining a constant bearing and avoiding nearby robots. Crucially, when the number of agents is large, a communication radius proportional to $R/\sqrt{n}$ is sufficient in most cases, where R is the radius of the region and n is the number of robots. Even when the motion of robots are dispersed using diffusion or have limited independent power, they should be capable of dispersing effectively given enough time. There are sound theoretical

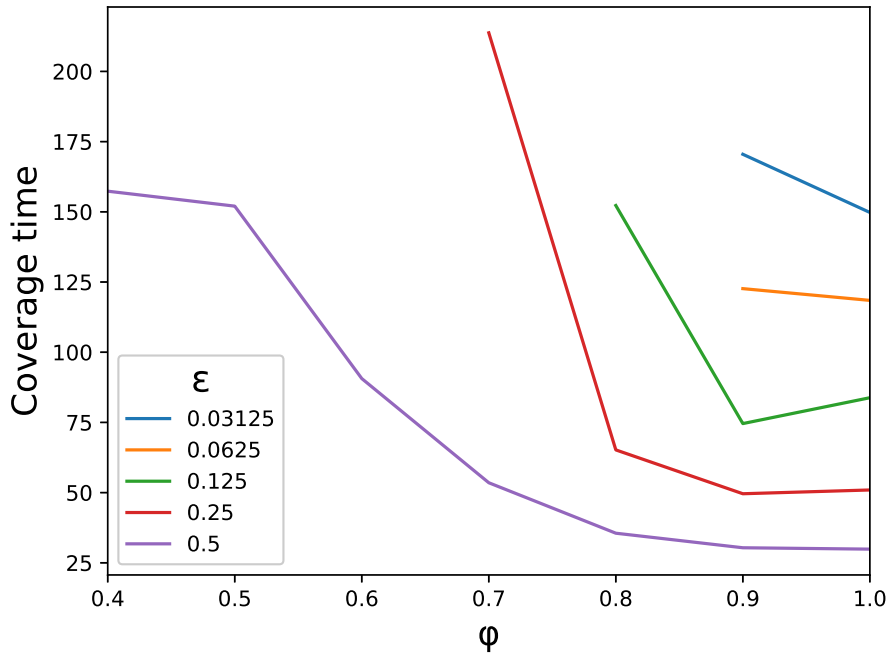


Figure 4.4: Average coverage time $\langle CT_{\text{inter}}(\epsilon)\rangle$ for repulsive Langevin robots at different values of ϕ and ϵ

reasons to believe that it is possible to cover wide areas with swarms of cheap robots with limited power and communications capabilities.

It seems likely that complex techniques and far ranging communication abilities are not necessary to reach a uniform distribution of robots or active particles across a space. It should be enough to simply avoid neighbours subject to some motion noise, whether that noise is due to natural fluctuations or injected artificially. As we can see from chapter 2, the speed at which coverage is largely determined by the autocorrelation $C(t)$ of the stochastic process $\mathbf{X}(t)$ that governs the motion of each robot or active particle.

There are, however many things yet to be discovered about area coverage. Even on this comparatively simple problem of area coverage within a bounded area, it was difficult to rigorously prove limit results about the stationary distribution of simple algorithms. In practise, most papers in this field either make drastic simplifications (i.e. low noise, or independence assumptions) about the motion of the robots, or rely entirely on simulations to benchmark algorithms against one another. In this work we have deduced some rudimentary mathematical bounds to assess the objective performance of four algorithms against the theoretical optimum.

An obvious candidate for further work would be an attempt to find the Fokker-Planck equation for the Linear Repulsion algorithm with motion noise, described at the end of chapter 3. This algorithm is very simple, however simulations show that it gives close to optimal performance.

We suspect that despite its apparent simplicity, the non-smooth nature of the mask of the sphere of influence of each robot makes it hard to describe mathematically. That is to say that the fact when robots move from a distance of d to $d+\epsilon$ away from one another, their acceleration drops immediately from to zero, makes it hard to apply standard techniques from dynamical systems, such as Lyapanov stability analysis.

Another direction to improve the analysis of coverage algorithms would be to remove the boundary altogether and replace it with some kind of flock centring. This could be achieved either by some kind of potential, anchoring robots towards the centre, or by accelerating towards the approximate barycentre of the swarm, similar to [19]. Further work in this area would make the analysis more widely applicable to scenarios where there is no natural boundary, such as aerial network deployment, or injection on nanoparticles carrying drugs that attack cancer cells within a tumour [26].

Neither of these problems are easy- in fact we have attempted both and not made much headway. It is our belief that heavier techniques from dynamical systems are probably necessary, and potentially even novel techniques. However we believe that the benefit of having a good analytic model for such systems makes such problems worth looking at. When the number of particles becomes large, simulations quickly become impractical.

Furthermore, it is my belief that rigorous analysis of simple swarming algorithms such as Reynolds flocking is the first step to understanding swarming algorithms more generally. If we are going to deploy swarms of robots in public places or within the human body, it will not be enough to have an approximate idea of what will happen, based on some simulations. Governments and regulators will require some kind of appropriate guarantees that the robots will behave as intended. A full stochastic model would give the most complete picture of what a swarm will do over time, enabling limit results, estimations of failure rates, and probabilities of exceptional events.

APPENDIX A - SELECTED SIMULATION CODE

langevin.py - Physics for langevin dynamics algorithm

```
import numpy as np
import particlesim
import hardboundary
import matplotlib.pyplot as plt


"""
Parameters:
        m:                      The mass (inertia) of the particles
        gamma:          The friction coefficient
        s:                      The average speed of the particles (not including electrostat
"""
def moveParticles(particles, t, boundary, params):
        m, gamma, s = params['m'], params['gamma'], params['s']
        T = 2 * m * pow(s,2) / np.pi
        var = 2 * gamma * T * t
        cov = [[var, 0], [0, var]]
        mean = (0, 0)
        b = np.random.multivariate_normal(mean, cov, len(particles))
        for i, particle in enumerate(particles):
                x0, v0 = particle.x, particle.v
                dv = - (gamma/m)*v0*t + (1/m)*b[i]
```

```
                v = v0 + dv
                x = x0 + v0 * t
                x,v = boundary.bounceIfHits(x0, v0, x, v)
                particle.x, particle.v = x, v

def main():
        params = {
                'm': 0.1,
                'gamma': 0.02,
                's': 0.35
        }
        n, iterations = 300, 1000
        folder = 'data/langevin n={} iter={}'.format(n, iterations)
        boundary = hardboundary.Circle(10.0)
        data = particlesim.simulate(iterations, n, moveParticles, folder, boundary, params
        # averageSpeed(data)
        particlesim.motionAnimation(data, 100, boundary)

def averageTemp(data, m):
        e = np.apply_along_axis(lambda v: 0.5 * m * pow(np.linalg.norm(v), 2), 2, data.v)
        ebar = e.mean(axis=1)
        plt.plot(ebar)
        plt.show()

def averageSpeed(data):
        e = np.apply_along_axis(np.linalg.norm, 2, data.v)
        ebar = e.mean(axis=1)
        plt.plot(ebar)
        plt.show()

if __name__=='__main__':
        main()
\end{lstlisting}

\textbf{runtumble.py} - Physics for the run and tumble algorithm
\begin{lstlisting}[language=Python]
import numpy as np
import particlesim
```

```
import hardboundary


def newDirection(angle, rng):
        theta = angle + np.random.uniform(rng[0], rng[1])
        x = np.sin(theta)
        y = np.cos(theta)
        return np.array((x,y))

def moveParticles(particles, t, boundary, params):
        rate, s = params['rate'], params['s']
        tumbleProb = rate * t
        for i, particle in enumerate(particles):
                x0, v0 = particle.x, particle.v
                x = x0 + v0 * t
                if np.linalg.norm(v0) == 0 or np.random.uniform(0,1) < tumbleProb:
                        v = s * newDirection(np.arctan2(v0[0], v0[1]),[-np.pi, np.pi])
                else:
                        v = v0
                x, v = boundary.bounceIfHits(x0, v0, x, v)
                particle.x, particle.v = x, v

def main():
        n, iterations = 300, 1000
        params = {'rate': 0.01, 's': 0.5}
        folder = 'data/run tumble n={} iter={}'.format(n, iterations)
        boundary = hardboundary.Circle(10.0)
        data = particlesim.simulate(iterations, n, moveParticles, folder, boundary, params)
        particlesim.motionAnimation(data, 100, boundary)

if __name__=='__main__':
        main()
```

electrostaticlangevin.py - Physics for the electrostatic langevin algorithm and linear repulsion with noise

```
import numpy as np
import particlesim
import electrostaticboundary
import repulsiveboundary
import linalgutil


def findNearbyParticleIndices(particles, distances, rNeighbour):
        return filter(lambda j: distances[j] > 0 and distances[j] < rNeighbour, range(len(

def electrostaticForce(r, q, alpha):
        # Bound the distance between two particles below by EPSILON to bodge distretisatio
        r = max(r, electrostaticboundary.EPSILON)
        return q**2 * pow(r, alpha)

"""
Parameters:
        m:                          The mass (inertia) of the particles
        gamma:                  The friction coefficient
        s:                          The average speed of the particles (not including electr
        rNeighbour:         Distance cut off of the electrostatic field
        qTotal:                  Total amount of charge
        alpha:                  Exponent of the electrostatic field
                                                -2 in 3D Coulomb's law
                                                -1 in 2D Coulomb's law
                                                 0 for linear repulsion
"""
def moveParticles(particles, t, boundary, params):
        m, gamma, s, rNeighbour, qTotal, qRing, alpha \
                = params['m'], params['gamma'], params['s'], params['rNeighbour'], \
                  params['qTotal'], params['qRing'], params['alpha']
        T = 2 * m * pow(s, 2) / np.pi
        var = 2 * gamma * T * t
        cov = [[var, 0], [0, var]]
        mean = (0, 0)
        b = np.random.multivariate_normal(mean, cov, len(particles))
```

```
        xx = tuple(map(lambda p: p.x, particles))
        D = linalgutil.distanceMatrix(xx, xx)
        q = qTotal / len(particles)
        for i, particle in enumerate(particles):
                jj = findNearbyParticleIndices(particles, D[i], rNeighbour)
                F = sum(map(lambda j: (particles[i].x - particles[j].x)/D[i,j] * electrostaticF
                x0, v0 = particle.x, particle.v
                if (boundary.rMax - np.linalg.norm(x0)) < rNeighbour/2.0:
                        Fb = boundary.force(x0, q) * qRing
                else:
                        Fb = 0.0
                dv = - (gamma / m)*v0*t + (Fb + F)/m*t + (1/m)*b[i]
                v = v0 + dv
                x = x0 + v0 * t
                x,v = boundary.bounceIfHits(x0, v0, x, v)
                particle.x, particle.v = x, v

def main():
        n, iterations = 300, 3000
        folder = 'data/electrostatic langevin n={} iter={}'.format(n, iterations)
        boundary = repulsiveboundary.Circle(10.0)
        params = {'m': 0.1, 'gamma': 0.05, 's': 0.02, 'rNeighbour': 0.9, 'qTotal': 30.0, 'qRing
        data = particlesim.simulate(iterations, n, moveParticles, folder, boundary, params)
        particlesim.motionAnimation(data, 20, boundary)

if __name__=='__main__':
        main()
```

particlesim.py - Simulator and animator for all algorithms

```
import numpy as np
import sys
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import linalgutil as la
import datamodel

R_0 = 1
FURTHEST_TARGET = 10
TARGET_SEPERATION = 0.5
TIMESTEP = 0.25

class Particle(object):
        def __init__(self, x, v):
                self.x = x
                self.v = v
                self.F = np.zeros((1,2))
                self.Fd = np.zeros((1,2))
                self.neighbours = 0

class Target(object):
        def __init__(self, y):
                self.y = y

def randomPointOnDisc(r):
        costheta = np.random.uniform(-1,1)
        u = np.random.uniform(0,1)
        theta = np.arccos(costheta) * np.random.choice((-1,1))
        s = r * np.power(u, 1/2.0)
        return la.polarToCart((s, theta))

def initParticles(n, r0):
        particles = []
        for i in range(n):
                x = np.array(randomPointOnDisc(r0))
                v = np.array((0,0))
                particle = Particle(x, v)
```

```
                particles.append(particle)
        return particles

def initTargets(d, rMax, boundary):
        furtherestTarget = rMax * 10.0
        targets = []
        for x1 in np.arange(-furtherestTarget, furtherestTarget, d):
                for x2 in np.arange(-furtherestTarget, furtherestTarget, d):
                        x = np.array((x1, x2))
                        if boundary.contains(x):
                                targets.append(Target(x))
        return targets

def recordData(particles, targets, data, i):
        data.x[i] = tuple(map(lambda p: p.x, particles))
        data.v[i] = tuple(map(lambda p: p.v, particles))
        data.F[i] = tuple(map(lambda p: p.F, particles))
        data.Fd[i] = tuple(map(lambda p: p.Fd, particles))
        data.t[i] = i * TIMESTEP
        data.y[i] = tuple(map(lambda tgt: tgt.y, targets))

def logIteration(i, iterations):
        perc = (i+1) * 100.0 / iterations
        sys.stdout.write("\rCalculating... %.2f%%" % perc)
        sys.stdout.flush()

def simulate(iterations, n, moveFn, folder, boundary, params={}):
        particles = initParticles(n, R_0)
        targets = initTargets(TARGET_SEPERATION, FURTHEST_TARGET, boundary)
        data = datamodel.Data(folder, 'w+', iterations, n, len(targets), boundary)
        recordData(particles, targets, data, 0)
        for i in range(1, iterations):
                moveFn(particles, TIMESTEP, boundary, params)
                recordData(particles, targets, data, i)
                logIteration(i, iterations)
        return data

def draw(i, scat, data):
```

```
        points = data.x[i]
        scat.set_offsets(points)
        return scat,

def motionAnimation(data, speedMultiplier, boundary):
        fig = plt.figure()
        axes = plt.gca()
        padding = 1.5
        axes.set_xlim([-FURTHEST_TARGET * padding, FURTHEST_TARGET * padding])
        axes.set_ylim([-FURTHEST_TARGET * padding, FURTHEST_TARGET * padding])
        axes.scatter(data.y[0,:,0], data.y[0,:,1], color='r', s=1)
        boundary.plot(axes)
        scat = axes.scatter(data.x[0,:,0], data.x[0,:,1])
        interval = TIMESTEP * 1000 / speedMultiplier
        ani = animation.FuncAnimation(fig, draw, interval=interval, frames = range(data.it
        plt.show()

def averageSpeed(data):
        s = np.linalg.norm(data.v, axis=2)
        sbar = s.mean(axis=1)
        plt.plot(data.t, sbar)
        plt.show()

def averageRadialDisplacement(data):
        r = np.linalg.norm(data.x, axis=2)
        rbar = r.mean(axis=1)
        plt.plot(data.t, rbar)
        plt.show()

def main(filePath, speedMultiplier):
        data = datamodel.Data(filePath, 'r')
        motionAnimation(data, speedMultiplier, data.boundary)

if __name__ == '__main__':
        if not len(sys.argv) == 3:
                raise ValueError('Arguments should be <data file path>, <speed multiplier>
        python, fPath, speedMultiplier = sys.argv
        main(fPath, int(speedMultiplier))
```

coverage.py - Main class for running trials for coverage

```
import numpy as np
import matplotlib.pyplot as plt
from multiprocessing import Pool
import signal
import pandas as pd
import scipy.special
import datetime
import pprint
import os
import sys

import particlesim
import datamodel
import coverageconfig

EULER_GAMMA = 0.577215664901532

def maxNumberOfCouponsApprox(n):
        return n / np.real(scipy.special.lambertw(n))

def maxNumberOfCoupons(n):
        return (n - 0.5) / np.real(scipy.special.lambertw(np.exp(EULER_GAMMA) * (n - 0.5)))

H = np.sqrt(2 * np.pi / (3 * np.sqrt(3)))
LOWER_BOUND = coverageconfig.R_MAX * H / np.sqrt(coverageconfig.N)
INDEPENDENT_LB = coverageconfig.R_MAX * H / np.sqrt(maxNumberOfCoupons(coverageconfig.N))

D_OPTIMAL_AGENTS = LOWER_BOUND * np.sqrt(2)

TIMEOUT = 99999999999999999

def init_worker():
        signal.signal(signal.SIGINT, signal.SIG_IGN)

def distanceToNearestTarget(y, particles):
        return np.min(list(map(lambda x: np.linalg.norm(x - y), particles)))
```

```
def supMinDistance(particles, targets):
        distances = list(map(lambda y: distanceToNearestTarget(y, particles), targets))
        return np.max(distances)

def supMinDistanceOverTime(data):
        d = np.zeros((data.iterations))
        for i in range(data.iterations):
                particlesim.logIteration(i, data.iterations)
                d[i] = supMinDistance(data.x[i], data.y[i])
        return d

def meanDistanceTravelled(data):
        s = np.linalg.norm(data.v, axis = 2)
        sbar = np.average(s, axis = 1)
        dx = sbar * particlesim.TIMESTEP
        return np.cumsum(dx)

def loadDataFromFile(algorithmProperties):
        data = datamodel.Data(algorithmProperties['filePath'], 'r')
        dataSet = {
                'name': algorithmProperties['name'],
                'data': data
        }
        return dataSet

def runSimulations(algorithmProps):
        params = algorithmProps['params'] if 'params' in algorithmProps else {}
        data = particlesim.simulate(coverageconfig.ITERATIONS, coverageconfig.N,
                                                                algorithmProps['moveFn'],
                                                                algorithmProps['boundary']
        dataSet = {
                'name': algorithmProps['name'],
                'data': data
        }
        return dataSet

def calculateCoverage(dataSet):
        covarageDistance = supMinDistanceOverTime(dataSet['data'])
```

```
        distanceTravelled = meanDistanceTravelled(dataSet['data'])
        return (dataSet['name'], (distanceTravelled, covarageDistance))

def drawGraph(df, names, filename):
        plots = []
        for name in names:
                plot, = plt.plot(df['time'], df[name + '.coverage'], label=name)
                plots.append(plot)
        plt.axhline(y=LOWER_BOUND, color='k')
        plt.axhline(y=INDEPENDENT_LB, color='k')
        plt.legend()
        plt.xlabel('Time (s)')
        plt.ylabel('Coverage distance (m)')
        plt.gca().set_ylim(bottom=0)
        plt.title('Maximum distance to the nearest agent')
        plt.savefig(filename)
        plt.show()

def drawGraphFromCsv(folder, cfg):
        df = pd.read_csv(folder + '/mean_coverage_distance.csv')
        names = map(lambda i: i['name'], cfg)
        outfilename = folder + '/test.png'
        drawGraph(df, names, outfilename)

def createDataFrame(t, distanceAndCoverage):
        df = pd.DataFrame()
        names = distanceAndCoverage.keys()
        df['time'] = t
        for name in names:
                distance, coverage = distanceAndCoverage[name]
                distanceColName = name + '.distance'
                coverageColName = name + '.coverage'
                df[distanceColName] = distance
                df[coverageColName] = coverage
        return df

def saveResults(folder, config, meanDistanceAndCoverage):
        names = list(map(lambda d: d['name'], config))
```

```
        t = np.arange(0, coverageconfig.ITERATIONS * particlesim.TIMESTEP, particlesim.TIM
        df = createDataFrame(t, meanDistanceAndCoverage)
        df.to_csv(folder + '/mean_coverage_distance.csv')
        out = open(folder + '/config.txt', 'w')
        out.write('ITERATIONS={} N={}\n'
                        .format(coverageconfig.ITERATIONS, coverageconfig.N))
        out.write(pprint.pformat(config))
        out.close()
        drawGraph(df, names, folder + '/mean_coverage_distance.png')

def saveTrialResults(folder, distanceAndCoverage, trialNumber):
        t = np.arange(0, coverageconfig.ITERATIONS * particlesim.TIMESTEP, particlesim.TIM
        df = createDataFrame(t, distanceAndCoverage)
        df.to_csv(folder + '/trial' + str(trialNumber+1) + '.csv')

def multipleTrials(config, numberOfTrials):
        names = list(map(lambda d: d['name'], config))
        pool = Pool(len(config))
        coverageData = np.ndarray((numberOfTrials, len(config), coverageconfig.ITERATIONS)
        distanceData = np.ndarray((numberOfTrials, len(config), coverageconfig.ITERATIONS)
        folder = 'results/coverage_comparison_' + str(datetime.datetime.now())[:19]
        os.mkdir(folder)
        for i in range(numberOfTrials):
                print('Trial ' + str(i + 1) + '/' + str(numberOfTrials))
                print('Running simulations...')
                dataSets = pool.map(runSimulations, config)
                print('\nCalculating coverage distances...')
                distanceAndCoverage = dict(pool.map(calculateCoverage, dataSets))
                saveTrialResults(folder, distanceAndCoverage, i)
                for j, name in enumerate(names):
                        distanceData[i,j] = distanceAndCoverage[name][0]
                        coverageData[i,j] = distanceAndCoverage[name][1]
                print('\n')
        meanCoverageData = np.mean(coverageData, axis=0)
        meanDistanceData = np.mean(distanceData, axis=0)
        meanDistanceAndCoverage = dict([(name, (meanDistanceData[j], meanCoverageData[j]))
        saveResults(folder, config, meanDistanceAndCoverage)
```

```
if __name__=='__main__':
        if not len(sys.argv) == 3:
                raise ValueError('Arguments should be <config>, <number_of_trials>')
        python, configName, numberOfTrials = sys.argv
        if not os.path.isdir('results'):
                os.mkdir('results')
        if not os.path.isdir('data'):
                os.mkdir('data')
        multipleTrials(coverageconfig.CONFIG[configName], int(numberOfTrials))
```

BIBLIOGRAPHY

[1] I. F. AKYILDIZ, D. POMPILI, AND T. MELODIA, *Underwater acoustic sensor networks: research challenges*, Ad hoc networks, 3 (2005), pp. 257–279.

[2] C. BECHINGER, R. DI LEONARDO, H. LÖWEN, C. REICHHARDT, G. VOLPE, AND G. VOLPE, *Active particles in complex and crowded environments*, Reviews of Modern Physics, 88 (2016), p. 045006.

[3] S. BERMAN, V. KUMAR, AND R. NAGPAL, *Design of control policies for spatially inhomogeneous robot swarms with application to commercial pollination*, in Robotics and Automation (ICRA), 2011 IEEE International Conference on, IEEE, 2011, pp. 378–385.

[4] M. BRAMBILLA, E. FERRANTE, M. BIRATTARI, AND M. DORIGO, *Swarm robotics: a review from the swarm engineering perspective*, Swarm Intelligence, 7 (2013), pp. 1–41.

[5] M. CHUPEAU, O. BÉNICHOU, AND R. VOITURIEZ, *Cover times of random searches*, Nature Physics, 11 (2015), p. 844.

[6] J. CRANK ET AL., *The mathematics of diffusion*, Oxford university press, 1979.

[7] Q. DU, V. FABER, AND M. GUNZBURGER, *Centroidal voronoi tessellations: Applications and algorithms*, SIAM review, 41 (1999), pp. 637–676.

[8] L. GIUGGIOLI, I. ARYE, A. H. ROBLES, AND G. A. KAMINKA, *From ants to birds: A novel bio-inspired approach to online area coverage*, in Distributed Autonomous Robotic Systems, Springer, 2018, pp. 31–43.

[9] GOOGLE, *Technology - loon*. `https://loon.co/technology/`, 2018.

[10] H. HAMANN AND H. WÖRN, *A framework of space–time continuous models for algorithm design in swarm robotics*, Swarm Intelligence, 2 (2008), pp. 209–239.

[11] S. HAUERT AND S. N. BHATIA, *Mechanisms of cooperation in cancer nanomedicine: towards systems nanotechnology*, Trends in biotechnology, 32 (2014), pp. 448–455.

[12] J. HEIDEMANN, M. STOJANOVIC, AND M. ZORZI, *Underwater sensor networks: applications, advances and challenges*, Phil. Trans. R. Soc. A, 370 (2012), pp. 158–175.

[13] A. HOWARD, M. J. MATARIC, AND G. S. SUKHATME, *Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem*, Distributed autonomous robotic systems, 5 (2002), pp. 299–308.

[14] S. JONES, M. STUDLEY, S. HAUERT, AND A. F. T. WINFIELD, *A two teraflop swarm*, Frontiers in Robotics and AI, 5 (2018), p. 11.

[15] R. KERSHNER, *The number of circles covering a set*, American Journal of mathematics, 61 (1939), pp. 665–671.

[16] E. MACK, *Meet google's project loon: Balloon-powered net access*, CNET. Retrieved, 15 (2013).

[17] A. MAINWARING, D. CULLER, J. POLASTRE, R. SZEWCZYK, AND J. ANDERSON, *Wireless sensor networks for habitat monitoring*, in Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, Acm, 2002, pp. 88–97.

[18] K. MARTENS, L. ANGELANI, R. DI LEONARDO, AND L. BOCQUET, *Probability distributions for the run-and-tumble bacterial dynamics: An analogy to the lorentz model*, The European Physical Journal E: Soft Matter and Biological Physics, 35 (2012), pp. 1–6.

[19] R. OLFATI-SABER, *Flocking for multi-agent dynamic systems: Algorithms and theory*, IEEE Transactions on automatic control, 51 (2006), pp. 401–420.

[20] L. M. OLIVEIRA AND J. J. RODRIGUES, *Wireless sensor networks: A survey on environmental monitoring.*, JCM, 6 (2011), pp. 143–151.

[21] J. OYEKAN, H. HU, AND D. GU, *Exploiting bacteria swarms for pollution mapping*, in Robotics and Biomimetics (ROBIO), 2009 IEEE International Conference on, IEEE, 2009, pp. 39–44.

[22] R. PATTLE, *Diffusion from an instantaneous point source with a concentration-dependent coefficient*, The Quarterly Journal of Mechanics and Applied Mathematics, 12 (1959), pp. 407–409.

[23] G. A. PAVLIOTIS, *Stochastic processes and applications*, Springer, 2016.

[24] J. PHILIP, *Numerical solution of equations of the diffusion type with diffusivity concentration-dependent*, Transactions of the faraday society, 51 (1955), pp. 885–892.

[25] ——, *Absorption and infiltration in two-and three-dimensional systems*, in Water in the Unsaturated Zone, Proceedings of the Wageningen Symposium, vol. 1, 1966, pp. 503–516.

[26] Y. REN, S. HAUERT, J. H. LO, AND S. N. BHATIA, *Identification and characterization of receptor-specific peptides for sirna delivery*, ACS nano, 6 (2012), pp. 8620–8631.

[27] C. W. REYNOLDS, *Flocks, herds and schools: A distributed behavioral model*, in ACM SIGGRAPH computer graphics, vol. 21, ACM, 1987, pp. 25–34.

[28] M. RUBENSTEIN, C. AHLER, AND R. NAGPAL, *Kilobot: A low cost scalable robot system for collective behaviors*, in Robotics and Automation (ICRA), 2012 IEEE International Conference on, IEEE, 2012, pp. 3293–3298.

[29] M. RUBENSTEIN, A. CORNEJO, AND R. NAGPAL, *Programmable self-assembly in a thousand-robot swarm*, Science, 345 (2014), pp. 795–799.

[30] K. SCHULTEN AND I. KOSZTIN, *Lectures in theoretical biophysics*, University of Illinois, 117 (2000).

[31] T. VICSEK, A. CZIRÓK, E. BEN-JACOB, I. COHEN, AND O. SHOCHET, *Novel type of phase transition in a system of self-driven particles*, Phys. Rev. Lett., 75 (1995), pp. 1226–1229.

[32] E. W. WEISSTEIN, *Disk covering problem from mathworld– a wolfram web resource*. Last visited on 15/1/2018.

[33] S. J. YOUNG, *Particles*. `https://github.com/samyoung17/particles`, 2018.

[34] L. YU, N. WANG, AND X. MENG, *Real-time forest fire detection with wireless sensor networks*, in Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on, vol. 2, IEEE, 2005, pp. 1214–1217.