Distributed and Decentralized Algorithms for Functional Programmable Matter

by

John Klingner

B.A., Cornell College, 2012

M.S., University of Colorado Boulder, 2015

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science

2018
This thesis entitled:
Distributed and Decentralized Algorithms for Functional Programmable Matter
written by John Klingner
has been approved for the Department of Computer Science

______________________________
Prof. Nikolaus Correll

______________________________
Prof. Nisar Ahmed

______________________________
Prof. Robert MacCurdy

______________________________
Prof. Julien Bourgeois

______________________________
Prof. Bradley Hayes

Date _________________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Programmable matter is made up of large quantities of particles that can sense, actuate, communicate, and compute. Motivated to imbue these materials with functionality, this thesis presents algorithmic and hardware developments to meet the unique challenges presented by large-scale robot collectives. The quantity of robots involved necessitates algorithms and processes which scale – in terms of required communication, computation, and memory – sub-linearly to the number of robots, if scaling at all can not be avoided. Included are methods for communication, movement, synchronization, and localization. To encourage application to a variety of hardware platforms, the theoretical underpinnings of these contributions are made as abstract as possible. These methods are tested experimentally with real hardware, using the Droplet swarm robotics platform I have developed. I also present abstractions which relate global performance properties of a functional object composed of programmable matter to local properties of the hardware platform from which the object is composed. This thesis is further supported by example implementations of functional objects on the Droplets: a TV remote control, a pong game, and a keyboard with mouse.
Dedication

To my Mom.

Thank you, so much, for helping me grow in to the person I am.

Thank you, for everything.

I will always love you.
Acknowledgements

I owe a lot of gratitude to my wife, Lorena Klingner: for her years of support as I conducted my research, and especially over the last couple of months as she picked up extra weight to help me focus on finishing my dissertation.

Thanks to my Dad, for helping raise me, and for all of his love and support through years, especially during some rough times for our family.

Thanks to my advisor, Professor Nikolaus Correll, for six years of guidance and mentorship for me and my research.

Thanks to all of my colleagues in the lab, particularly my collaborators: Yang Li and Dr. Anshul Kanakia, and Dr. Dana Hughes, who generously gave of his time to share his skills and experience with me – on many topics.

And, thanks to Jeremy Jones and many others for their friendship and helping provide some escape.
# Contents

Chapter

1 Introduction 1

2 Droplet Hardware 5

2.1 Communication 5

2.2 Synchronization 6

2.3 Range Sensing 9

2.3.1 Algorithm 12

2.4 Motion 15

3 Localization 18

3.1 Method 21

3.1.1 Covariance Intersection 21

3.1.2 Covariance Union 23

3.1.3 Combining CI and CU 26

3.2 Experimental Validation 27

3.2.1 Comparison 29

4 Local Requirements for Global Functionality 32

4.1 Method Outline 33

4.2 Constructive Solid Geometries 33

4.3 CSG Description to Code 33
List of Figures

Figure

1.1 A large-scale robot collective. ................................................................. 4

2.1 Two Droplets, shown with a quarter for scale. The right Droplet is shown on its side and with the top shell removed. ................................................................. 6

2.2 Each of the nine images above shows the experimental setup used to produce each plotted datum, which is the average of ten trials. The figures are labeled with the robot density (in \( \text{node m}^{-2} \)) and the distance between transmitting and receiving nodes (in cm). For densities greater than 162 \( \text{node m}^{-2} \), the number of available robots necessitated reducing the distance. .................................................. 7

2.3 Collection of figures from my range-sensing paper [34]. .................................................. 10

2.4 This figure demonstrates the variables identified by our range sensing model. In this text, ‘bearing’ indicates what is here denoted \( \theta \), ‘heading’ is \( \phi \), and ‘range’ is \( |\Delta x_{i,j}^{k}| \). .................................................. 11

2.5 Simple 1DoF model of stick-slip motion principle. Horizontal motion of the mass causes the platform to scoot right and left. The vertical motion of the mass affects the magnitude of the friction experienced by the platform as it scoots. Thus, the platform experiences net motion over a full rotation, in a direction determined by the rotation’s direction [65]. .................................................. 15

2.6 Arrangement of vibration motors and legs. .................................................. 16

2.7 Each graph contains two different experiments, with paths by two different (solid, dashed) autocalibrated Droplets. .................................................. 17

2.8 A Droplet programmed to walk straight, rotate 90°, and repeat; it should form a square. 17
b Droplet programmed to walk in three different directions without rotating; it should form a triangle. These attempts have been aligned with a common initial position and orientation.

3.1 Shows distance, in communication hops, from four different root nodes in a fixed configuration. In all cases the root node is red; zero hops away from itself.

3.2 The two blue ellipses show the estimates being fused. The red ellipse shows the fused estimate.

   a Covariance Intersection.
   b Covariance Union. Dashed line shows CI covariance for comparison.

3.3 Line shows mean position error over six runs collected from three different centrally-located robots. Colored region shows mean estimated position error. All runs collected with Droplets in the configuration shown.

   a Experimental Setup. Units: mm.

3.4 Line shows median position error of last received position from all robots, collected from three different centrally-located robots over three different runs. Colored region shows upper and lower quartiles. All runs collected with Droplets in the configuration shown.

   a Experimental Setup. Units: mm. Cyan circles: landmarks.

3.5 Each line shows a single Droplet’s position error, measured over a single run. Solid lines show CI+CU; dashed lines show the simulated just CI. Configuration as in Figure 3.4a.

3.6 Thick lines show the mean position error over three different runs sampling three different Droplets. The dashed lines show each individual experiment incorporated in the mean.

4.1 Example CSG code and resulting model, produced with openSCAD.

4.2 "Illustration of CSG tree" by Zottie.

5.1 The Droplet Display.
5.2 The Droplet Remote Control in action. Left images, from top to bottom: different regions and associated roles; Droplets, placed but unpowered; Droplets, after Localization-based role determination is completed.

5.3 The Droplet Mouse & Keyboard

5.4 Droplet Pong. Top left image shows a visualization of the Droplets’ positions, with red intensity periodically proportional to \( x \) position and green intensity periodically proportional to \( y \) position (period of 175 mm). Remaining images shows the progression of a ball through the pong board; Droplets light up white with an intensity proportional to how much the ball overlaps their position. The bottom two images are overlayed with a representation of the ball, which has a 120 mm diameter. The affect of localization errors is visible in the scattering of illuminated Droplets not part of the ball.

6.1 Left: an image from [99] shows Droplets gathering to cooperatively extinguish a simulated “fire”, represented by the red region. Bottom-Right: an example of Acetic Acid (\( \text{C}_2\text{H}_4\text{O}_2 \)) from [100]. Top-Right: An image from [12] showing a camouflage pattern of vertical stripes.

B.1 Top view of Droplet Circuit Board (3D Rendering)

B.2 Bottom view of Droplet Circuit Board (3D Rendering)

B.3 Droplet Shell Parts. Both interactive in the Adobe® Acrobat® PDF Reader, or other PDF Readers supporting embedded 3D. Click to activate.

a Top Shell

b Bottom Shell

B.4 Schematic for the Droplet Swarm Robotics Platform
Chapter 1

Introduction

The idea of programmable matter conceived in [1] and [2] is that of matter which can be reprogrammed to have a different shape, or different physical properties. Such matter would have a wealth of useful applications for a wide variety of areas. Current efforts towards this goal are developing mm-scale robots with capability for computing (based on the M3 mote [3]), locomotion, and communication [4]. The desired shape could then be formed out of large collections of such robots: hundreds if not thousands or tens of thousands. A wealth of research [4]–[8] exists on algorithms for swarm assembly; on the problem of the robot collective forming itself into a desired shape. My work is motivated by the goal of making these objects functional: while the shape of a mug is largely what makes it useful, a TV remote control is made useful by the algorithms, sensing, and actuation which make up its functionality. In [8], the Kilobots form the shape of a wrench, but what would it require to make the wrench sense applied torque?

More broadly, the algorithmic developments I present in this thesis are designed for any large-scale homogeneous collective of robots; it is not required that they be able to assemble the shape of the object themselves. Imagine manufacturing a functional keyboard by mixing a slurry of robots in a substrate, then stamping it in to the desired physical shape. Imagine a magazine, its cover a little thicker than usual. Instead of an image, the cover is an interactive display. Though these may seem far-fetched, I present distributed algorithms which make them possible, requiring only investment in miniaturizing the required hardware.

Though foreign to current manufacturing methods, nature itself is a testament to this cellular approach as a viable engineering paradigm. In the longer term, mastery of such an approach could allow the
possibility of mimicking complex biological systems such as skin, muscles, or bones [9]. A variety of functions ranging from locomotion [10], shape change [11], reaction/diffusion models for generating Turing patterns [12], or complex classification and optimization operations for texture [13] or gesture recognition [14] have been demonstrated to run on collections of identical computing elements in a fully distributed manner.

Regardless of whether the robots formed themselves in to the object’s shape (as in programmable matter) or not, the large quantity of robots involved and limited computational resources require computational, communication, and storage complexity invariant to the size of the collection\footnote{Ideally. In practice, we can design for this invariance but it is difficult to test for it; emergent properties which scale at $O(\sqrt{n})$, $O(\log n)$, or slower can be difficult to detect with the size of hardware swarms available.}. The spatial computing paradigm [15], forms part of the algorithmic basis for my approach. The functional object is treated as a continuous computational medium, approximated by the discrete collection of robots in it.

Work in spatial computing goes back to the paper by Abelson, Allen, Coore, et al. [16] which introduced the term amorphous computing to describe computing on systems of many interconnected computing nodes. Beal and Bachrach [17] present the programming language Proto as a general-purpose language for spatial computing. The authors have followed this with an in-depth analysis of Proto [18] and, later, a further abstraction of spatial programming as a discrete field calculus [19]. This abstraction is used to develop a handful of low-level algorithmic building blocks (which accomplish, for example, information propagation), and prove that these building blocks have certain desired properties (such as stability) for a spatial computer [15], [20]. More recently, this work has led to the development of Protelis [21]–[24], a Java-based implementation of the Computational Field Calculus first presented using Proto. Current development seems to be focused on a scala-based implementation of these ideas, called scafi [25], [26].

This body of work has focused on the development of Computational Field Calculus and programming languages which implement it. They aim to solve the global-to-local problem by presenting a language for describing the desired global behavior in a way that can be used to generate the needed individual code. This is an exciting project, but still needs more development before a meaningful program in Protelis or scafi can be reliably compiled for low-level embedded systems.
paintable computer presented in [27] represents an early realization of a spatial computer. This work is notable for the unique nature of its abstraction: the display works through the virtual self-assembly of process fragments (“pfrags”). The author focuses on relatively-advanced graphics rendering and text display [28]. The Cubelets [29] are a commercially-available system of heterogeneous cubes which can be connected together to create functional objects. Different Cubelets possess different sensors and actuators. Though each Cubelet has its own microcontroller, only specific cubes expose the ability to write code to run on each Cubelet, making the system effectively centralized. It is still possible that the Cubelets would make an excellent platform for testing the manufacturing paradigm I present, but that would require greater control over each cube’s software. Also notable are the Bitblox presented in [30]. Bitblox are small circuit boards (~ 1 cm³), and are heterogeneous: batteries, microcontrollers, and various sensors and actuators each existing on their own individual Bitblox. The authors present a pipeline for production of a desired functional object, from modeling of the shape to automated assembly. Other work in this space aims to 3D print functional objects in a more ‘traditional’ sense: additive manufacturing of conductive materials and other materials with certain specific properties [31], [32]. These works all provide ways to combine the production of an object’s shape with that of its functionality. However, they are not cellular, and suffer from the inherently sequential nature of additive manufacturing.

My low-level hardware and algorithmic contributions for functionality include movement calibration [33] and range-sensing [34], as well as implementations for communication and synchronization. These are all discussed in detail in Chapter 2. An important primitive for any spatially computing robot is to know where in space it is. I have developed a fault-tolerant distributed algorithm for cooperative localization by a large group of robots [35], discussed in detail in Chapter 3. Chapter 4 has theoretical analyses of the relationship between local properties of individual robots and global performance limitations of the functional object, and Chapter 5 has examples of functional objects I have made using the Droplet swarm robotics platform.
Figure 1.1: A large-scale robot collective.
Chapter 2

Droplet Hardware

My hardware results all use the Droplet Swarm Robotics Platform. The development of the “Droplets” was started by Dustin Reishus and Nicholas Farrow in 2011. I have led their development since 2013. The Droplets are completely open source, with code and manufacturing information available online. Each Droplet – pictured in Figure 2.1 – is roughly cylindrical, with a radius of 2.2 cm and a height of 2.4 cm. They use an Atmel xMega128A3U microcontroller with a 32 MHz clock, 8 kB of RAM, and 64 kB of SRAM. The Droplets receive power via their legs through a floor with conductive, alternating strips with 5 V of potential between them. A large capacitor (1.5 F) is used to keep the Droplet powered during temporary separations from the powered floor. Appendix B includes a reproduction of the Droplets’ circuitry and shape. Each Droplet has six sets of IR emitters, sensors, and receivers used for inter-robot communication and range sensing.

2.1 Communication

Each of the six sets of IR hardware on a Droplet is connected to a separate UART. Outgoing messages are modulated with a 38 kHz carrier wave. The IR receiver handles conversion of the modulated signal back to a sequence of bits for the xMega’s UART. The frequency of the carrier wave limits the Droplet’s data-transmission rate to 3200 bit/s. Communication occurs in packets, each containing up to 40 B of data. All packets include an additional 7 B of header information. This information includes: the sending Droplet’s ID (2 B), a crc-code for error checking (2 B), message length (1 B), and, optionally, a target Droplet’s ID (2 B).

1 http://github.com/correlllab/cu-droplet
As the Droplets all communicate in a single shared medium, a CSMA-based approach is used to avoid a Droplet starting a message while another is communicating.

In discussing the performance of a spatial computer, the most-important aspect of communication is how quickly information can travel through the network (ie, through the computational medium). Two different scenarios present themselves: when information needs to travel from one (or a small number) of cells to the rest of the computer, and when information needs to travel from all of the cells to the rest of the computer. The data in Figure 2.2 represent an analysis of the effect network density has on communication speed for the former scenario. Beal and Bachrach [17] describe the latter scenario as ‘gossip’.

2.2 Synchronization

Algorithmic development in synchronization began with the need to synchronize computer networks. The challenges and constraints of such systems, however, are quite different from those which face wireless sensor networks, and spatial computers. An early method for wireless sensor networks is RBS [36]. This method works by synchronizing all agents based on time-of-receipt for a single broadcasted message. If all agents are in range to receive that message, the algorithm works well, leaning on the reasonable assumption that message-processing delays will be similar for all agents. This method does not synchronize robots across multiple hops (ie, if there are too many agents to reach all of them with a single message).
Figure 2.2: Each of the nine images above shows the experimental setup used to produce each plotted datum, which is the average of ten trials. The figures are labeled with the robot density (in \(\frac{\text{node}}{m^2}\)) and the distance between transmitting and receiving nodes (in cm). For densities greater than 162 \(\frac{\text{node}}{m^2}\), the number of available robots necessitated reducing the distance.
The TPSN algorithm presented in [37] offers a solution to the multi-hop problem. It starts with some root agent, and constructs a spanning tree (ideally, a minimal one) from that agent. Synchronization error increases at worst linearly with distance from the root agent, but is overall less than with RBS. This is achieved by adding a time stamp to each message, right before it is sent. The FTSP algorithm in [38] works with similar constraints, but to avoid the problems/overhead of producing a spanning tree, each agent keeps track of how many hops away from the root it is, and collects synchronization information from nearby agents which are closer to the root. FTSP also makes use of a leader election algorithm to choose the root. These changes make the algorithm more robust to node failure and other changes in network topology.

In a 1990 paper, Mirollo and Strogatz [39] discuss various examples of synchronization found in nature, citing examples like synchronized firefly flashes, and pacemaker cells in the heart. All of these examples can be modeled as “pulse-coupled oscillators”. This concept is fairly simple: each agent has some value, $X$, which increases until reaching some threshold, which causes it to “fire”, and reset. The function for $X$ over time could be as simple as:

$$X(t) = \begin{cases} X(t-1) + 1 & X(t-1) < T \\ 0 & X(t-1) \geq T \end{cases}$$

for some threshold $T$, but some methods use more complicated monotonically-increasing functions. This periodic “firing” gives us an oscillation. The coupling comes from — whatever it means for an agent to “fire” – other agents sense that firing event, and increase $X$ as a result, an increase which may cause them to “fire” and reset $X$ immediately. As Mirollo and Strogatz demonstrate, such a system of pulse-coupled oscillators will synchronize – provided that all agents can sense all firing events. The literature often refers to this as “Firefly Synchronization”. It was later proven by Lucarelli, Wang, et al. [40] that firefly synchronization will converge even when not all agents can sense all other agents’ events – ie, in a more realistic multi-hop scenario. In 2005, Werner-Allen, Tewari, Patel, et al. provide a version of firefly synchronization that addresses some of the issues which arise in real hardware implementations. These issues include delays between when a message is sent (agent fires) and received, and limitations in an
agent’s ability to sense several simultaneous firing events. The key ideas needed to solve these issues are: instead of increasing their counter immediately when an event is detected, the agents wait until the next counter reset, and reset it to some value larger than 0, based on the appropriate cumulative increase from all events detected over the previous period. Additionally, there is a small window after each firing event during which agents ignore any firing events they detect. This is the method employed by the Droplets.

2.3 Range Sensing

The measurement of relative position and orientation is a critical operation in multi-robot systems and swarm robotics aggregation [42], dispersion [43], pattern recognition [44], flocking [45], and navigation [46], [47]. There exist multiple systems based on infrared [48], ultrasound/radio [49] and ultra-wideband radio [50] for medium-scale multi-robot systems, but precise measurements on centimeter-scale robots such as Alice [51], Jasmine [52] or Kilobot [53] remains a challenge due to the size of emitter, receiver, and signal processing hardware. For clarity: “range” is the distance between the measured and measuring robots, “bearing” is the rotation of a vector pointing from the measuring robot to the measured robot, and “heading” is the orientation of the measured robot from the measuring robot’s perspective. Note that range and bearing together represent the position of the measured robot in polar coordinates, in a coordinate frame centered on the measuring robot. Existing methods for small-scale robots therefore either focus on range-only sensors [53], or implement only crude bearing measurements [51], [52].

Robots moving on the plane of a floor or tabletop have three degrees of freedom: two describing position and one describing orientation. Thus, for one measurement to fully describe the relative position between measured and measuring robot, it must also have three degrees of freedom. While range information can be used to derive bearing, and bearing information can be used to derive range and heading [34], [54], [55], these approaches rely on multiple measurements, communicated between multiple robots. This has two important drawbacks: the additional complexity and time costs of communication, and the introduction of correlated error in shared measurements. Treating this correlation appropriately adds complexity to any localization method built on these measurements. Localization methods are discussed in Section 3.

The algorithm my co-authors and I presented in [34] takes a separate IR-intensity measurement for
Figure 2.3: Collection of figures from my range-sensing paper [34].

(a) Distance Model
(b) Emitter Model
(c) Sensor Model

Apparatus used to collect data.
Figure 2.4: This figure demonstrates the variables identified by our range sensing model. In this text, ‘bearing’ indicates what is here denoted $\theta$, ‘heading’ is $\phi$, and ‘range’ is $|\Delta x_{i,j}^k|$. each sensor-emitter pair, and leverages this additional information to calculate range, bearing, and heading as part of a single range measurement. This model is presented with platform-specific equations and values abstracted from the algorithm itself; previous work in this area has tended to be hardware specific.

Collecting and using the full set of intensity measurements does have a cost, both in how long a range measurement takes, and in algorithmic complexity. But the benefit is that a robot is able to get enough information from a measurement to calculate the complete relative pose of the measured robot, without additional communication and without introducing correlated error. Another benefit is that accounting for signal attenuation from the transmitting robot’s orientation allows for more accurate measurement of range, particularly when the two robots are quite close to one another [34]; prior work has treated the transmitting robot as a point light source. This model assumes that all robots are identical, that each robot has a well-defined forward direction, and that it exists in a plane. It further assumes that one robot’s state relative to another robot can be fully described by its two-dimensional position ($[x, y] \in \mathbb{R}^2$) and orientation ($\phi$). Each sensor and emitter is assumed to be directional and outward facing. Figure 2.3 shows a sampling of figures from this paper.

An arbitrary robot platform has $N$ emitters and $M$ sensors. A full set of measurements between each sensor-emitter pair thus results in $N \times M$ pairwise intensity measurements. The positions of these hardware components relative to the center of the platform are given by vectors $\vec{TX}_{1,N}$ and $\vec{RX}_{1,M}$ respectively. I identify three values that might effect measured intensity: the distance between the sensor and emitter ($r_{i,j}$), attenuated by the incident angle on the sensor ($\alpha_{i,j}$) and by the emission angle from the emitter ($\beta_{j,i}$).
Figure 2.4 provides a demonstration of these values for the Droplet swarm robotics platform, on which the sensors and emitters are co-located. I refer to the full set of $N \times M$ intensity measurements as a “brightness matrix”, denoted $\Lambda \in \mathbb{Z}^{N \times M}$. An individual element of the brightness matrix is denoted $\lambda_{I,J}$; this is the intensity measured by sensor $J$ from emitter $I$. With $F$ as the model describing how the measured intensity is effected by the three factors identified above, I can write:

$$\lambda_{I,J} = F(|r_{i,j}|, \alpha_{i,j}, \beta_{i,j})$$  \hspace{1cm} (2.1)

### 2.3.1 Algorithm

This algorithm has as input a full “brightness matrix” $\Lambda$. First, the bearing and heading are calculated by using the intensities in $\Lambda$ as weights for a weighted average of a collection of unit vectors. Intuitively, if one value in $\Lambda$ is much brighter than the others, the associated sensor and emitter are likely to be facing one another. If two values in $\Lambda$ are roughly equal, the sensed robot is likely to be roughly halfway between the field of view of the two sensors. This is the intuitive basis for my weighted-average approach. More precisely, with $\Theta$ denoting the collection of vectors for bearing and $\Phi$ denoting the collection of vectors for heading,

$$\Theta = \begin{bmatrix} \hat{R}_X_1 & \hat{R}_X_2 & \ldots & \hat{R}_X_M \\ \hat{R}_X_1 & \hat{R}_X_2 & \ldots & \hat{R}_X_M \\ \vdots & \vdots & \ddots & \vdots \\ \hat{R}_X_1 & \hat{R}_X_2 & \ldots & \hat{R}_X_M \end{bmatrix}$$

$$\Phi = \begin{bmatrix} \delta \angle (\hat{R}_X_1, -\hat{T}_X_1) & \delta \angle (\hat{R}_X_2, -\hat{T}_X_1) & \ldots & \delta \angle (\hat{R}_X_M, -\hat{T}_X_1) \\ \delta \angle (\hat{R}_X_1, -\hat{T}_X_2) & \delta \angle (\hat{R}_X_2, -\hat{T}_X_2) & \ldots & \delta \angle (\hat{R}_X_M, -\hat{T}_X_2) \\ \vdots & \vdots & \ddots & \vdots \\ \delta \angle (\hat{R}_X_1, -\hat{T}_X_N) & \delta \angle (\hat{R}_X_2, -\hat{T}_X_N) & \ldots & \delta \angle (\hat{R}_X_M, -\hat{T}_X_N) \end{bmatrix}$$

With $\delta \angle (\mathbf{a}, \mathbf{b})$ the angular difference between two unit vectors, $\delta \angle (\mathbf{a}, \mathbf{b})$ is a vector of unit length pointing in that direction:

$$\delta \angle (\mathbf{a}, \mathbf{b}) = \arctan \mathbf{b} - \arctan \mathbf{a}$$

$$\delta \angle (\mathbf{a}, \mathbf{b}) = \begin{bmatrix} \cos (\delta \angle (\mathbf{a}, \mathbf{b})) \\ \sin (\delta \angle (\mathbf{a}, \mathbf{b})) \end{bmatrix}$$

In an effort to reduce the complexity of the notation, the definition of $\Theta$ above assumes that all $\hat{R}_X_j$ vectors are of the same length. If they are not, the lengths should be normalized. While $\Phi$ may seem complex,
the formalism presented here for each element is only a general-purpose way to answer this question: “If emitter TX$_i$ on robot B and sensor RX$_j$ on robot A are exactly facing one another, what is the orientation of robot B in the coordinate frame of robot A?” With $\Lambda$, $\Theta$, and $\Phi$ in hand, the bearing and heading can be calculated by simply taking the weighted average:

$$\text{bearing} = \theta = \arg \left( \sum_{i,j} \Lambda \odot \Theta \right)$$

$$\text{heading} = \phi = \arg \left( \sum_{i,j} \Lambda \odot \Phi \right)$$

(2.2)

where $\odot$ is used to denote the element-wise product, and the $\arg$ function is used to indicate the angle of the unit vector. Conceptually, $\Theta$ and $\Phi$ are really matrices of angles, and $\theta$ and $\phi$ both average angles, weighted by $\Lambda$. The representation of $\Theta$ and $\Phi$ as a matrix of unit vectors pointing in the direction of the relevant angle, and the $\arg$ function before the result, are only needed due to the problems with taking the mean of circular quantities.

Now that the bearing and heading have been calculated, I need to calculate the range. Returning to the attenuation model in Equation 2.1, I make the simplifying assumption that the attenuation from $|r_{i,j}|$, $\alpha_{i,j}$, and $\beta_{i,j}$ are independent. That is:

$$\lambda_{i,j} = F(|r_{i,j}|, \alpha_{i,j}, \beta_{i,j}) = D(|r_{i,j}|) S(\alpha_{i,j}) E(\beta_{i,j})$$

(2.3)

Where $D$ describes the reduction in measured intensity with distance, and $S$ and $E$ describe attenuation due to the angle of incidence/emission. These functions are sometimes provided by the manufacturer, or can be determined experimentally. Though it doubtless simplifies some effects, classical optics tells us that simply taking the cosine of the angle may be reasonable [56]. The independence assumption that lets us get from Equation 2.1 to Equation 2.3 is certainly not true in all cases. For the Droplets, at separation distances less than around 10 mm, some relative orientations cause the Droplet’s shell to occlude sensors from emitters. However, the domain of values for which the assumption breaks down is small enough not to present an issue to the Droplets, and our experiments in [34] generally validate this simplifying assumption. By inverting Equation 2.3 I get:

$$|r_{i,j}| = D^{-1} \left( \frac{\lambda_{i,j}}{S(\alpha_{i,j}) E(\beta_{i,j})} \right)$$

(2.4)

If I know $\alpha_{i,j}$, $\beta_{i,j}$, and have a model for each of $S$, $E$, and $D$, then the calculation of the distance between...
sensor \( i \) and emitter \( j \) is simply a matter of plugging in the numbers. Unfortunately, \( \alpha_{i,j} \) and \( \beta_{i,j} \) are not independent of \( |\Delta x| \), the range. With \( r_{i,j} \) the vector pointing from sensor \( i \) on the sensing robot to emitter \( j \) on the transmitting robot, this relationship is:

\[
r_{i,j} = \left( \Delta x + \begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix} \hat{T}_X_j \right) - \hat{R}_X_i \tag{2.5}
\]

\[
\alpha_{i,j} = \delta \angle (\hat{R}_X_i, r_{i,j}) \tag{2.6}
\]

\[
\beta_{i,j} = \delta \angle \left( \begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix} \hat{T}_X_j, -r_{i,j} \right) \tag{2.7}
\]

Recalling that \( \Delta x = \text{range}\begin{bmatrix} \cos(\text{bearing}) \\ \sin(\text{bearing}) \end{bmatrix} \), the relationship between \( \alpha \), \( \beta \), and the range is clear. Thus, I can not factor out the effects of \( S(\alpha) \) and \( E(\beta) \) directly. Instead, it is done in two parts. First: I calculate an approximate \( \alpha_{i,j} \) and \( \beta_{i,j} \) using the simplifying assumption that \( r_{i,j} \) is parallel to \( \Delta x \), which is true if \( |\Delta x| \gg |\hat{R}X|. \) Under this assumption:

\[
\alpha_{i,j} = \theta - \text{arg}(\hat{R}_X_i)
\]

\[
\beta_{i,j} = \theta - \text{arg}(\hat{T}_X_j) - \phi - \pi
\]

both of which can be calculated using the already-determined bearing (\( \theta \)) and heading (\( \phi \)). I choose the largest element of \( \Lambda \) (the sensor-emitter pair with the most intense IR measurement), and use this value to calculate an approximate \( r_{i,j} \) using Equation \( 2.4 \), and I can plug this value in to Equation \( 2.5 \) to calculate the approximate range. Second: I use this approximate range to recalculate a much better approximation of \( \alpha_{i,j} \) and \( \beta_{i,j} \) using Equations \( 2.6 \) and \( 2.7 \), and again apply Equation \( 2.4 \) to obtain a value for \( r_{i,j} \), and Equation \( 2.5 \) to calculate the range. This time, however, I use these steps to calculate a range value for every pairwise intensity measurement \( \lambda_{i,j} \). Finally, I used \( \Lambda \) as weights for a weighted average of the calculated range; this provides the resulting range. In practice, the directionality of our sensors and emitters means only a subsection of \( \Lambda \) will be non-zero. The final calculated range is a weighted sum of only the non-zero ranges, weighted by the original \( \lambda_{i,j} \). This final weighting is because large \( \lambda \) values are less affected by noise.
Figure 2.5: Simple 1DoF model of stick-slip motion principle. Horizontal motion of the mass causes the platform to scoot right and left. The vertical motion of the mass affects the magnitude of the friction experienced by the platform as it scoots. Thus, the platform experiences net motion over a full rotation, in a direction determined by the rotation's direction [65].

2.4 Motion

Classically, miniature robot platforms such as r-one [57], Jasmine [52], and Alice [51] have required geared motors. Another class of approaches are variants of the “stick-slip” actuator introduced in [58] and shown to be particularly attractive for high precision movements [59]–[63] and force control [64].

Whereas stick-slip actuators traditionally require changing their length, e.g., using piezo ceramics, [58], [62], a similar effect can be achieved with a vibration motor — a spinning weight with axis of rotation offset from center of mass — an effect well-known from the “Bristle-bot” series of toys. The basic motion dynamics of these toys is illustrated in Figure 2.5. The simplicity and availability of this actuator has led to its use on low-cost miniature robot platforms, such as the Kilobots [53]. The Kilobots’ drive train is based on the design presented in [66], which approximates the dynamics of a differential-drive wheel platform. Achieving fully-holonomic motion on the plane requires at least three vibration motors. Vartholomeos and Papadopoulos [65] simulate such a drive train, but this implementation requires that all three motors rotate (a) at the same speed, and (b) with nearly the same phase. Their follow-up real-hardware implementations [66] have fallen back to a two-motor approach due to the complexity of satisfying these constraints.

The work I published in [33] avoids the requirement for phase and rotation-locked motors by only actuating one motor at a time. In contrast to previous implementations, the Droplet platform has its motors opposite its legs. This can be seen in Figure 2.6. Thus, the effect demonstrated in Figure 2.5 causes the robot...
to pivot about the leg opposite the actuated motor, at least over small time scales. I refer to one of these short pivoting motions as a “step”. The Droplet has three motors. Therefore, there are six possible steps a Droplet might take: pivoting clockwise or anticlockwise, around each of the three legs. Holonomic motion is achieved by chaining together an appropriate sequence of steps. Moving towards $l_2$, for example, requires an alternating sequence of clockwise steps induced by $m_3$ and anticlockwise steps induced by $m_1$.

A drawback to this method is that each Droplet needs to be calibrated such that all steps are of the same magnitude. The work I published in [33] addresses this with an auto-calibration routine using an overhead camera. Since uncalibrated robots move in circles, this routine fits a circle to the robot’s path of motion and uses Nelder-Mead Optimization to maximize the fitted circle’s radius. Figures 2.7 show paths walked by two different robots after being calibrated using the automatic routine.

Figure 2.6: Arrangement of vibration motors and legs.
(a) Droplet programmed to walk straight, rotate $90^\circ$, and repeat; it should form a square.

(b) Droplet programmed to walk in three different directions without rotating; it should form a triangle. These attempts have been aligned with a common initial position and orientation.

Figure 2.7: Each graph contains two different experiments, with paths by two different (solid, dashed) autocalibrated Droplets.
Chapter 3

Localization

Localization refers to the ability for each robot in a swarm to determine its position in a coordinate system shared by the whole swarm. This is an important primitive for robot swarms [67]. There are a number of ways that localization has been achieved: hard-coding, environment-sensing, gradient-based methods, and by combining local measurements. In situations where the robots in a swarm are static, a trivial solution is to hard-code each robot’s position. This requires either IDs and $O(N)$ storage for $N$ robots, which scales poorly, or requires different code to be programmed to each robot, which greatly complicates deployment.

Another approach allowing for more haphazard deployment is for the robots in a swarm to sense their position from their environment. Wang, Colas, Liu, et al. [68] use robots moving across a known two-dimensional bar-code pattern with a pair of simple light sensor. By comparing the signals from the light sensors to the known pattern and wheel odometry, the robots are able to determine their position. A similar approach suited for static robots might be to project a changing pattern of light such that there were a bijective mapping between positions and light patterns.

There are also gradient-based methods which don’t require any specialized localization-sensing hardware to establish a coordinate system. These methods are built on counting the number of communication hops a robot is from other specific, fixed robots. A key challenge is that these approaches rely to some degree or another on a unit-disk model of communication: the number of communication hops used as a crude distance metric. Our observations indicate that real-world communication — at least with directional IR hardware – is not disc shaped. Particularly in a dense arrangement of nodes, reflections and multi-path
effects \cite{69} render the assumption that physical distance can be mapped to hop count unfeasible. Though existing hop-count algorithms address this challenge to varying degrees \cite{70}, \cite{71}, they still rely on the unit disk model being ‘close enough’.

In a physical system, this is not always the case, leading to inconsistent hop counts that make deployment of robust coordinate systems difficult. Figure 3.1 shows an example of this; a collection of hop counts for a typical Droplet configuration. The kilobots \cite{8} and Butera’s paintable computing nodes \cite{28} use hop-count gradient-based localization for their swarm-assembly algorithm. Both of these devices use omnidirectional IR hardware reflecting off the floor, as opposed to the outward-facing directional IR hardware in the Droplets \cite{34}.

The last class of localization methods I will discuss are those which combine individual robot’s range measurements. This is referred to in the literature as cooperative localization, and was first posed in \cite{72}. Cooperative localization methods have been demonstrated on a small group of robots using Maximum-Likelihood estimation \cite{73}, and in simulation with Maximum A Posteriori estimation \cite{74}. The problem has also been approached stochastically: Prorok, Bahr, and Martinoli \cite{75} use a particle-filter method which was first applied to this problem by Fox, Burgard, Kruppa, et al. \cite{76}.

Roumeliotis and Bekey \cite{77} were the first to apply Kalman Filtering to the cooperative Localization problem. The key issue with a direct Kalman Filter approach is that, in order for robot A to update its position with a measurement from robot B, robot A needs to include any correlation between errors in its position estimate with errors in robot B’s position estimate. This would require that every robot store and update a cross-covariance matrix for every other robot from which it might get an update. Later work has
mitigated this issue by subdividing the collective into smaller groups [78], or by ignoring the problematic noise terms [79].

Covariance Intersection (CI), first presented in [80], provides a means by which to fuse estimates with approximate cross-covariance matrices. This method was first applied to the cooperative localization problem in [81]. More recent work by Carrillo-Arce, Nerurkar, Gordillo, et al. [82] builds on this further: where prior methods would have each robot communicate an observation directly (requiring the observed robot to know information about the observing robot’s position), this method has each robot combine their observation with their own current position estimate to produce a “relative pose observation”; this is what the robot communicates. The observed robot can then fuse this relative pose observation with its current position estimate using CI. This approach is particularly well-suited to multirobot localization in large-scale swarms, because each robot only needs to store and update its own state, and only one communication is needed per measurement.

Though CI provides a good method by which to fuse measurements with unknown cross-correlation, an important drawback is that the resulting fused covariance is, by necessity, pessimistic about the amount of correlation present. For some applications, some error terms may be completely independent, and CI alone is unable to take advantage of this independence. The split covariance intersection method [83], [84] addresses these problems by updating a Kalman filter with the independent information and fusing that result with the pure CI estimate. In [85], the authors balance between the costs and benefits of the Kalman Filter and CI approaches by only communicating a measurement when the expected information gain is above some threshold. These explicit measurements are fused with the position estimate using a Kalman Filter. The lack of such communication is treated as an implicit measurement and fused with the current state estimate using CI.

Another drawback to CI is that it can only provide consistent updates when the observed position is consistent with the previous position estimate. When a group of robots may have faulty sensors, or when observations otherwise could have error larger than the measurement model expects, it can lead to inconsistent updates. Uhlmann [86] first presented a solution to this problem: if an observation is determined to be inconsistent with the current estimate and it is unknown whether the current position
or observed position is correct, Covariance Union provides an updated covariance consistent with both estimates [87].

My contribution to this area is in a large-scale (~ 80 robots) hardware implementation of the CI algorithm presented in [82], augmented with the Covariance Union [86] to maintain consistency in the presence of intermittent unmodeled measurement error [35]. I use a thresholded Mahalanobis distance, as suggested in [86], to assess an observation’s consistency. Most of the existing work in cooperative localization has been on groups of ten or fewer robots; the simulated results for 30 robots in [88] stand out for their scale. The most similar existing work is [89], in which the authors augment their cooperative localization algorithm to maintain consistency when faced with faulty sensor readings. They use the Kullback-Leibler divergence to test for consistency. Their hardware implementation uses three Turtlebots.

3.1 Method

3.1.1 Covariance Intersection

Suppose a team of $N$ robots. With $B_i^j$ indicating the $i$-th robot, that robot’s position at some time $t$ is denoted

$$x_i^j = \begin{bmatrix} x_i^j \\ y_i^j \\ \theta_i^j \end{bmatrix} \quad (3.1)$$

with $x_i^j, y_i^j$ indicating the robot’s position in the plane, and $\theta_i^j$ indicating the robot’s orientation. At each time $t$, each robot is tracking an estimate of its position, $\hat{x}_i^j$. This estimate is assumed to have some error with corresponding covariance $P_i^j$; this covariance is tracked as well. A measurement contains information about the relative pose of the measured robot. A measurement of robot $B_i^j$ by robot $B_i^j$ at some time $t$ is denoted $z_{i}^{i,j}$. The model for such a measurement is:

$$z_{i}^{i,j} = h_{i}^{i,j}(x_i^j, x_i^j) + n_{i}^{i,j} \quad (3.2)$$
with \( h_{i,j} \) indicating the true relative pose between the two robots at time \( t \):

\[
h_{i,j}(x_i, x_j) = (\Gamma_{x_i})^T (x_j - x_i)
\]

\[
\Gamma_{x_i} = \begin{bmatrix}
\cos \theta_i & -\sin \theta_i & 0 \\
\sin \theta_i & \cos \theta_i & 0 \\
0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (3.3)

and \( n_{i,j} \) is the zero-mean Gaussian measurement noise with covariance \( R_{x_i} \).

When \( B_i^j \) obtains a relative-pose measurement of \( B_j^i \) — \( \hat{z}_{i,j}^j \) — it uses that measurement and its own estimated position, \( \hat{x}_i^j \), to compute an estimated position for \( B_j^i \):

\[
\hat{x}_j^i = \hat{x}_i^j + \Gamma_{x_i} \hat{z}_{i,j}^j
\]  \hspace{1cm} (3.4)

This calculation comes from “undoing” Equation (3.2), using \( \Gamma \) as defined in Equation (3.3). Propagating the measurement error present in \( z_{i,j}^j \) with the error in \( B_i^j \)'s estimate of its own position gives the following equation for the error in Equation (3.4):

\[
\tilde{x}_j^i = \tilde{H}_{i,j} \tilde{x}_i^j - \Gamma_{x_i} \tilde{z}_{i,j}^j
\]

\[
\tilde{H}_{i,j} = \begin{bmatrix}
1 & 0 & y_j^i - y_i^j \\
0 & 1 & x_j^i - x_i^j \\
0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (3.5)

From this, the covariance matrix describing the error in the estimated position can be written as

\[
P_{x_i}^{j,i} = \tilde{H}_{i,j} P_{x_i}^i (\tilde{H}_{i,j})^T + \Gamma_{x_i} R_{x_i} (\Gamma_{x_i})^T
\]  \hspace{1cm} (3.6)

After computing \( \tilde{x}_j^i \) and \( P_{x_i}^{j,i} \), \( B_i^j \) communicates these values to \( B_j^i \).

Next, \( B_j^i \) needs to fuse this new information with its own position estimate, \( \hat{x}_j^i \) and \( P_{x_i}^i \). Note that I have no assumption that \( \hat{x}_j^i \) and \( \hat{x}_j^{i*} \) are uncorrelated estimates: even if they were, after \( B_j^i \) has updated its position by fusing the estimate from \( B_i^j \), they will be correlated for all future updates. It is of course possible to track this correlation as another \( 3 \times 3 \) matrix but doing so for all possible \( i, j \) pairs requires \( O(N) \) storage overall, and \( O(N) \) communications per measurement \[82\], as all other robots need to be informed of the updated covariances.

To mitigate these issues, Covariance Intersection is used, which provides a way to estimate the
amount of mutual information in the two estimates. With CI, the covariance update step is:

\[ P_{j+1}^t = \left( \omega \left( P_j^t \right)^{-1} + (1 - \omega) \left( P_j^* \right)^{-1} \right)^{-1} \]  

with \( \omega \in [0,1] \) chosen to minimize \( \text{Tr}(P_{j+1}^t) \). The updated estimate is then:

\[ \hat{x}_{j+1}^t = P_{j+1}^t \left( \omega \left( P_j^t \right)^{-1} \hat{x}_j^t + (1 - \omega) \left( P_j^* \right)^{-1} \hat{x}_j^* \right) \]  

As mentioned, the optimal \( \omega \) minimizes the trace of the fused matrix. Though finding the best \( \omega \) is a relatively simple optimization problem, it still introduces additional computational complexity for a resource-constrained system. As such, I use the method described in [90] for calculating an approximation of the optimal \( \omega \), denoted \( \tilde{\omega} \). This is:

\[ \tilde{\omega} = \frac{\left| \left( P_j^t \right)^{-1} + \left( P_j^* \right)^{-1} \right| - \left| \left( P_j^* \right)^{-1} \right| + \left| \left( P_j^t \right)^{-1} \right|}{2 \left| \left( P_j^t \right)^{-1} + \left( P_j^* \right)^{-1} \right|} \]  

where \( |P| \) is used to indicate the determinant of \( P \). Figure 3.2a shows the results for an example pair of observations. The notation for Equations (3.7) to (3.9) is much cleaner if the information matrix is used instead of the covariance. That is, with \( I_j^t = \left( P_j^t \right)^{-1} \):

\[ I_{j+1}^t = \omega I_j^t + (1 - \omega) I_j^* \]

\[ \hat{x}_{j+1}^t = P_{j+1}^t \left( \omega I_j^t \hat{x}_j^t + (1 - \omega) I_j^* \hat{x}_j^* \right) \]

\[ \tilde{\omega} = \frac{\left| I_j^t + I_j^* \right| - \left| I_j^* \right| + \left| I_j^t \right|}{2 \left| I_j^t + I_j^* \right|} \]  

3.1.2 Covariance Union

The method described so far covers the typical case: \( B_i^j \) gets a measurement and uses Equations (3.4) and (3.6) to calculate \( \hat{x}_j^* \) and \( P_j^* \). \( B_i^j \) then communicates these values to \( B_j^i \), which uses Equations (3.8), (3.7) and (3.9) to update its state. A problem with this approach is that the Covariance Intersection method assumes that the two estimates it is fusing are consistent. Though a reasonable assumption for some applications, I (and others [89]) have found that our hardware is not always so well behaved.
Figure 3.2: The two blue ellipses show the estimates being fused. The red ellipse shows the fused estimate.
Covariance Union (CU) methods are one way to fuse two inconsistent measurements \cite{86}, \cite{87}. Less formally, where CI fuses two covariances by covering the region present in both, CU fuses two covariances by covering all regions between both. This is shown in Figure 3.2b. A closed-form application of CU to fuse two estimates was originally presented in \cite{91}. The reproduction here is based on that presentation, and that in \cite{87}. Let
\[
U_1 = P_j^t + \left(\hat{x}_j^{t+1} - \hat{x}_j^t\right)\left(\hat{x}_j^{t+1} - \hat{x}_j^t\right)^T
\]
(3.11)
\[
U_2 = P_j^t + \left(\hat{x}_j^{t+1} - \hat{x}_j^t\right)\left(\hat{x}_j^{t+1} - \hat{x}_j^t\right)^T
\]

Note that a fused position estimate is one of the inputs to CU. $U_1$ includes the covariance of the position estimate communicated by $B_j^t$, and the difference between this position estimate and the fused estimate. $U_2$ similarly includes the difference between $B_j^t$'s previous position estimate and the new fused estimate, along with the covariance of the previous position estimate. Then compute $S$ such that
\[
U_2 = SS^T
\]
(3.12)

That is, $S$ is the upper-triangular Cholesky Decomposition of $U_2$. Next, compute the matrix
\[
R = (S^{-1})^T U_1 S^{-1}
\]
(3.13)
And calculate the eigendecomposition of $R$. That is, with eigenvalues $\lambda_{1-3}$, find $D$ and matrix of eigenvectors $V$ such that:
\[
R = VDV^{-1}
\]
(3.14)
Using these computed matrices, I can finally calculate the CU-fused covariance for the new estimated position $\hat{x}_j^{t+1}$:
\[
P_j^{t+1} = U = S^T V \begin{bmatrix}
\max(\lambda_1, 1) & 0 & 0 \\
0 & \max(\lambda_2, 1) & 0 \\
0 & 0 & \max(\lambda_3, 1)
\end{bmatrix} V^T S
\]
(3.15)
3.1.3 Combining CI and CU

I have said that a CU-fused covariance must be used when the two estimates are inconsistent, but have not yet formally discussed what that means. An estimate \((\hat{x}_j^*, P_j^*)\) is inconsistent with another estimate, \((\hat{x}_j^i, P_j^i)\) when the distance between the two estimates is greater than some threshold, for some distance metric. The choice of distance metric and threshold is inherently application specific \[86\], requiring tuning to distinguish between a faulty measurement and a low-probability observation. The metric I use is from \[86\], and is incorrectly described as the Mahalanobis distance:

\[
d = \left(\hat{x}_j^i - \hat{x}_j^*\right) \left(P_j^i + P_j^*\right)^{-1} \left(\hat{x}_j^i - \hat{x}_j^*\right) \tag{3.16}
\]

This distance metric is clearly related to the squared Mahalanobis distance; a Mahalanobis Distance of summed covariances.

If \(d\) is less than some threshold \(d_{\text{CU}}\) the two estimates are fused using CI. If \(d\) is greater than a second, larger threshold \(d_{\text{TOSS}}\), then the observation is ignored. Intermediate values are fused using CU. The full method is summarized in Algorithm 1. Typically, a CI update reduces the uncertainty of an estimate, or keeps it constant. A CU update will increase the uncertainty of the estimate.

Algorithm 1 Pseudocode description of B\(^i\(_j\) updating its position based on a measurement from B\(^i\(_j\).

\[
\begin{align*}
\text{function UPDATE}(\hat{x}_j^i, P_j^i, \hat{x}_j^*, P_j^*) \\
& \quad \hat{x}_{t+1}^i, P_{t+1}^i \leftarrow \text{CI}(\hat{x}_j^i, P_j^i, \hat{x}_j^*, P_j^*) \\
& \quad d \leftarrow \text{DISTANCE}(\hat{x}_j^i, P_j^i, \hat{x}_j^*, P_j^*) \\
& \quad \text{if } d > d_{\text{TOSS}} \text{ then} \\
& \quad \quad \hat{x}_{t+1}^i, P_{t+1}^i \leftarrow \hat{x}_j^i, P_j^i \\
& \quad \text{else if } d > d_{\text{CU}} \text{ then} \\
& \quad \quad P_{t+1}^i \leftarrow \text{CU}(\hat{x}_{t+1}^i, \hat{x}_j^i, P_j^i, \hat{x}_j^*, P_j^*) \\
& \quad \quad \text{end if} \\
& \quad \text{end function}
\end{align*}
\]

Now, the method described above provides a way for a robot, B\(^i\(_j\), to update its estimated position based on an update communicated to it by another robot, B\(^i\(_j\). In order to calculate that estimate, B\(^i\(_j\) needed to already have a position estimate of its own, in addition to the measurement of B\(^i\(_j\). If B\(^i\(_j\) gets a position estimate from B\(^i\(_j\) and B\(^i\(_j\) does not yet have any estimate for its position, it simply initializes its position

\[\text{The squared Mahalanobis distance between an observation, } \mathbf{x} \text{ and a multinormal distribution with mean } \mu \text{ and covariance } \Sigma \text{ is defined as } (\mathbf{x} - \mu)\Sigma^{-1}(\mathbf{x} - \mu) \tag{92}.\]
estimate to be the estimate from $B_i^j$.

If our team of $N$ robots all start with uninitialized positions, then the method as described has no way to produce an update. Thus, a small number of robots $\ell$ ($1 \leq \ell \ll N$), all start in pre-determined hard-coded positions with very small covariances. These $\ell$ robots serve as landmarks for the rest of the swarm. This could also be accomplished by combining this approach with any other localization data to initialize the system.

If a robot moves, its position estimate can be updated as described in [82].

Due to numerical-stability issues arising from floating point arithmetic, the compression of our covariance matrix, and/or the circular nature of the orientation, we found it necessary to check that any given covariance update was positive definite, before using it to update the robot state. Violations of this check were rare.

### 3.2 Experimental Validation

With the data shown here, the distance thresholds used for consistency checking are $d_{CU} = 1.0$ and $d_{TOSS} = 4.0$. These values were chosen through qualitative experiment and by consulting the cumulative $\chi^2$-distribution. In all cases, each Droplet is performing a range-sensing broadcast every 15 s to 20 s. Any nearby Droplets which receive that message may complete a range-sensing measurement and thus calculate and send a position update message to the measured Droplet. Typically, a given broadcast will yield between 0 and 3 update messages. There are four landmark Droplets ($\ell = 4$), which are located in the four corners; these can be seen in Figure 3.4a. The “position error” shown is the distance between the Droplets’ true position and estimated position. Orientation error is not represented due to the difficulty in collecting ground truth for orientation, but as the relative positions are quite dependent on an accurate orientation, this data still demonstrates overall accuracy. The centrally-located robots sampled for most of the data shown are furthest from the landmark Droplets, and thus tend to have the largest errors.

The “estimated position error”, as used in Figures 3.3 and 3.4, is the square root of the largest eigenvalue of the position estimate covariance matrix. This value is added and subtracted from the true error in the position estimate to show the Droplet’s estimated error in the position estimate.
Figure 3.3: Line shows mean position error over six runs collected from three different centrally-located robots. Colored region shows mean estimated position error. All runs collected with Droplets in the configuration shown.

Figure 3.4: Line shows median position error of last received position from all robots, collected from three different centrally-located robots over three different runs. Colored region shows upper and lower quartiles. All runs collected with Droplets in the configuration shown.
Figure 3.5: Each line shows a single Droplet’s position error, measured over a single run. Solid lines show CI+CU; dashed lines show the simulated just CI. Configuration as in Figure 3.4a.

### 3.2.1 Comparison

By logging all observations of the sampled robots for the data shown in Figure 3.4, we are able to simulate the performance of an unmodified CI algorithm. The CI update steps are performed off line, in order for each new observation. These results for three separate trials can be seen in Figure 3.5. The two plots for each run only deviate when the original data performed a CU update step while the simulation performs a CI update step. Overall, the performance of the two methods is nearly identical; indeed, the simulated CI algorithm seems to perform slightly better.

This is because the ~ 40 updates represented in Figure 3.5 don’t exhibit the rare errors of unmodeled magnitude our algorithm is adapted to. Importantly, the situation simulated here is one where all Droplets are running the CI+CU algorithm, and only the sampled Droplet is running an unmodified CI+CU. With 81 Droplets making observations, the likelihood of a rare faulty measurement is much larger.

Thus, we performed an additional experiment in which all robots are running an unmodified CI for localization. These results can be seen in Figure 3.6. With none of the Droplets running CI+CU, the performance of the sampled robots’ localization is much more erratic.

I have presented the largest-scale hardware validation of CI-based localization yet seen. The method

---

2 Updates with a Mahalanobis distance greater than 4 \( (d > d_{TOSS}) \) were still ignored.
Figure 3.6: Thick lines show the mean position error over three different runs sampling three different Droplets. The dashed lines show each individual experiment incorporated in the mean. Configuration similar to Figure 3.4a.
is robust to occasionally-faulty sensor readings, and thus performs more stably than existing scalable Co-
operative Localization techniques. Compared to existing swarm-robotic localization methods, this is a
continuation of the work by Pires et al. [88] in bringing Cooperative Localization to Swarms: the CI+CU
method fully models for measurement error and uncertainty.
Chapter 4

Local Requirements for Global Functionality

First, I want to acknowledge a perhaps-obvious set of hardware requirements: in order for a functional object composed of many homogeneous nodes to react to some stimulus, the individual nodes must have some ability to react to that stimulus. Similarly, in order for a functional object to perform some sort of actuation, each individual node must have some ability to actuate thus. At their most trivial, these requirements manifest as the light sensor needed to record pixel values in the Droplet Display, and the visible-light LEDs needed to be a part of that display. However, some examples are less straightforward. The Droplet Keyboard uses a simple electret microphone: not to record audio, but as a button. Taps on the Droplet’s shell produce brief, high-amplitude spikes in the signal from the microphone, which I interpret as a button press. Other inputs may require local communication: in a more-flexible substrate, for example, a button press would shift the robot’s position slightly, in a way that could be detected using localization. The drawback to this approach would be a relatively high cost in communication and customization. This approach would have a relatively high cost in both communication and computation.

It would be beyond the scope of this work — if not impossible — to attempt to enumerate all possible inputs and outputs of functional objects, along with some mapping of the desired inputs and outputs to individual hardware. In the less trivial cases, the problem of producing hardware capable of detecting a certain set of global inputs or affecting a certain set of global actuations is left to the system designer. This chapter first introduces some of the techniques I employ for creating functional objects, then concludes with a discussion of relationships between hardware/low-level performance, and global object performance.
4.1 Method Outline

The method — summarized by Algorithm 2 — begins with an understanding of an object’s shape, and the necessary functionality of the object, or parts of the object. Constructive Solid Geometries (CSG) provides a formal way to describe the region or regions which make up an object, and any necessary associated functionality. This is described further in Section 4.2. The implementation of UpdateLocalization is described in Chapter 3. The implementation of UpdateRoles is described in Section 4.4.

Algorithm 2 Pseudocode description of the main loop for robot $i$, $B^i$.

\[
\text{loop} \\
\hat{x}_i^t, P_i^t \leftarrow \text{UpdateLocalization}(\hat{x}_{i-1}^t, P_{i-1}^t) \\
R^t \leftarrow \text{UpdateRoles}(\hat{x}_i^t, P_i^t) \\
\text{for } r_j \in R^t \text{ do} \\
\quad \triangleright \text{Whatever role } r_j \text{ is supposed to do.} \\
\text{end for} \\
\text{end loop}
\]

4.2 Constructive Solid Geometries

CSG is a technique for modeling solids. It consists of geometric primitives (spheres, cuboids, cylinders), transformations of those primitives (translation, rotation, color), and combinations of those transformed primitives (union, difference, intersection). Figure 4.1 includes a simple CSG model of a remote control, and the accompanying source code. One property that makes a CSG representation of solids desirable is that it is easy to determine what solid(s) a given point is within. A key component of an implementation is the function which maps from a position (the robot’s location) to the role or set of roles associated with that position. The code for this function can be produced directly from the CSG specification of the shape. The idea of using CSG for encoding shapes used in distributed robotics comes from [93].

4.3 CSG Description to Code

CSG descriptions of objects consist of a few primitive solids, and then transformations (ie, scaling, translation, rotation) and/or combinations (ie, union, intersection) of these primitives. These can be visualized as a tree, seen for example in Figure 4.2
remoteHeight = 10;
remoteLength = 100;
remoteWidth = 60;
buttonSize = 20;
buttonDepth = 5;

module button(x, y, col, type){
  role(type){
    role("button"){
      color(col){
        translate([x,y,buttonDepth/2]){cylinder(h=buttonDepth,r=buttonSize/2,center=true);
      };
    };
  }

  role("body"){
    color("DimGray"){
      cube([remoteWidth,remoteLength,remoteHeight], center);
    }
  }

  role("emitter"){
    color("Black"){
      translate([0,remoteLength/2,0]){cube([remoteWidth, buttonDepth, remoteHeight], center);
    }
  }
}

button(remoteWidth/4, 0, "Red", "chanUp");
button(remoteWidth/4, -remoteWidth/2, "Green", "chanDown");
button(-remoteWidth/4, 0, "Blue", "volUp");
button(-remoteWidth/4, -remoteWidth/2, "Yellow", "volDown");
button(0, (1/2)*remoteWidth, "White", "power");

Figure 4.1: Example CSG code and resulting model, produced with openSCAD.
Figure 4.2: "Illustration of CSG tree" by Zottie.
OpenSCAD has primitives for a sphere, cube, cylinder, and an arbitrary polyhedron. Given a sphere, cube, or cylinder of unit size, aligned with the axes and centered at the origin, it is trivial to write code to determine if a given point is inside that shape. An arbitrary sequence of transformations of that primitive shape can then be composed together in to a single transformation matrix, call it $T_{CSG}$. Provided $T$ is invertible, we can combine the transformation matrix for sampling from a multivariate Gaussian (described in Appendix A.2) with $T_{CSG}^{-1}$, and then simply test whether the transformed point is in the appropriate unit solid. When the CSG includes the union, intersection, or difference from multiple shapes, these combinations become boolean checks on different composed transformations.

More concretely, the code can contain the simple functions: `isInSphere()`, `isInCube()`, `isInCylinder()`. Each distinct region which needs to be tested for role differentiation can be converted to a set of transformation matrices ($T_{CSG}$) which convert one of these primitive solids to a set of regions needed to determine if a robot has a specific role. Each transformation in this set can be inverted, giving $T_{CSG}^{-1}$, a transformation which converts a relevant region to a primitive solid. Each robot only needs to store the composed tree of inverse transformation matrices for each role. To determine if a certain position in space has a role, all of the inverse transformation matrices for that role are applied to that point, and primitive-solid-measurement tested, giving a set of booleans. If a role has only a single region, then a position has that role if the value is True, indicating that after transformation that position is within the appropriate primitive solid. Some CSG specifications involving the union or intersection of different solids will require an additional step of logical ands and ors of the set of booleans to determine role assignment. As sampling from the multivariate-normal distribution (which represents the robot’s current position estimate) requires a transformation matrix $T_{\mathcal{N}}$, and as that transformation matrix is constant for all points sampled for the Monte-Carlo integration described in Section 4.4, the computation can be optimized by composing the $T_{\mathcal{N}}$ transformation with the $T_{CSG}$ transformation up-front.

4.4 Role Mapping

This section discusses methods for mapping from a position estimate to the set of roles which correspond to that position. Naively, this might be done by simply returning all roles associated with the mean
position estimate provided by localization. However, this ignores the fact that our position estimate comes with uncertainty.

Simply using the first estimated position from localization performs poorly, since the early estimates tend to have large error and fluctuate dramatically. But how long should the robot wait? How precise of an estimate is precise enough? These questions don’t have a fixed answer, since they are affected by the number of different roles and size of the nearby role-mapping regions. A more principled approach allows one to choose a generic threshold appropriate for all functional objects.

Recalling that our position estimate $\hat{x}_i^t, P_i^t$ represents a multivariate Gaussian distribution for the robot’s position and orientation, we can integrate the PDF of that distribution over each role-mapped region to calculate the probability that the robot has the associated role, given the estimated position. With $r_i$ a specific role, $R_i$ the associated region, $\hat{x}_i^t, P_i^t$ the position estimate of robot $B_i$ and $R_i$ the set of robot $B_i$’s roles, this is:

$$
\text{Pr}(r_i \in R_i \bigg| x_i^t \sim N(\hat{x}_i^t, P_i^t)) = \int_{R_i} \int \frac{1}{\sqrt{\text{Det}(2\pi P_i^t)}} \exp\left(-\frac{1}{2} (x - \hat{x}_i^t)^\top (P_i^t)^{-1} (x - \hat{x}_i^t)\right) dx
$$

Using $f_{N(\hat{x}_i^t, P_i^t)}(x)$ as shorthand for the probability density function, this is:

$$
\text{Pr}(r_i \in R_i \bigg| x_i^t \sim N(\hat{x}_i^t, P_i^t)) = \int_{R_i} \int f_{N(\hat{x}_i^t, P_i^t)}(x) dx \quad (4.1)
$$

As the regions only care about the robot’s position and not its orientation, we are integrating over the domain of all possible orientations. As the probability of the robot having one of all possible orientations is 1, we can ignore the orientation components of the position estimate and covariance, and write:

$$
\text{Pr}(r_i \in R_i \bigg| x_i^t \sim N(\hat{x}_i^t, P_i^t)) = \int_{R_i} \int \int f_{N(\hat{x}_i^t, P_i^t)}(x) dx dy dz \quad (4.2)
$$

Once all of the probabilities have been calculated, $B_i^t$ is determined to have all roles $r_i$ for which the probability $\text{Pr}(r_i \in R_i)$ is greater than some fixed threshold; I used 0.6 for all of the functional objects included in this work.

To actually calculate this integral on a low-cost platform, I use Monte-Carlo integration. I sample $M$ points from the $N(\hat{x}_i^t, P_i^t)$ distribution and tally up the proportion of those $M$ points which end up
Algorithm 3 Pseudocode for calculating role likelihoods based on position estimate.

1: InverseTfms $\leftarrow \{\text{body}: \begin{bmatrix} 1/60 & 0 & 0 \\ 0 & 1/100 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{emitr}: \begin{bmatrix} 1/60 & 0 & 0 \\ 0 & 1/5 -19 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{pwr}: \begin{bmatrix} 1/60 & 0 & -3 \\ 0 & 1/50 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{chnUp}: \begin{bmatrix} 1/60 & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{chnDn}: \begin{bmatrix} 1/60 & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{volUp}: \begin{bmatrix} 1/10 & 0 & -3 \\ 0 & 1/10 & -8 \\ 0 & 0 & 1 \end{bmatrix}, \text{volDn}: \begin{bmatrix} 1/10 & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}\}$

2: function $\text{InSquare}\left((\text{InverseTfms}[r_i] \circ \text{EstimateTfm})\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}\right)$ then $\text{Increment}(\text{RoleFrequencies}[\text{role}])$ end if

3: function $\text{InCircle}\left((\text{InverseTfms}[r_i] \circ \text{EstimateTfm})\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}\right)$ then $\text{Increment}(\text{RoleFrequencies}[\text{role}])$ end if

4: function $\text{UpdateRoles}(\hat{x}, P)$
5: EstimateTfm $\leftarrow \text{GetMultinormalTransform}(\hat{x}, P)$ $\triangleright$ Equation (A.4)
6: RoleFrequencies $\leftarrow 0$ $\triangleright$ Map, initialized to 0 for all roles.
7: for $N$ Monte-Carlo Samples do
8: $\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$ i.i.d. $\mathcal{N}(0, 1)$ $\triangleright$ Two points sampled from standard normal.
9: for all $r_i \in \{\text{body, emitr}\}$ do
10: if $\text{InSquare}\left((\text{InverseTfms}[r_i] \circ \text{EstimateTfm})\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}\right)$ then $\text{Increment}(\text{RoleFrequencies}[\text{role}])$ end if
11: end for
12: for all $r_i \in \{\text{pwr, chnUp, chnDn, volUp, volDn}\}$ do
13: if $\text{InCircle}\left((\text{InverseTfms}[r_i] \circ \text{EstimateTfm})\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}\right)$ then $\text{Increment}(\text{RoleFrequencies}[\text{role}])$ end if
14: end for
15: end for
16: return $\left\{ r_i \in R \right\| \frac{\text{RoleFrequencies}[r_i]}{N} \geq 0.6 \}$
17: end function
in each region. This proportion is an estimate of the value of the integral Equation (4.2), with accuracy increasing as $M$ increases. The threshold represents how confident a robot must be about being in a given region $\mathcal{R}_i$ before assigning itself the associated role $r_i$. Appendix A.2 describes a method for sampling from a multivariate normal distribution in a computationally-limited environment. In summary, this method involves applying a transformation matrix, $T^\mathcal{N}$, to a vector of points each sampled from the standard normal distribution.

Algorithm 3 provides the implementation of UpdateRoles specified by the CSG description; several variables referenced there are defined in Figure 4.1. The justification for this implementation is provided in Section 4.4. Note that the composition of the inverse role transformations with the estimate transformation which occurs on lines 16 and 21 only needs to be calculated once, up front. Thus, the per-sample-point calculation is reduced to single matrix multiplication (in addition to the random number generation and calls to InSquare or InCircle).

4.5 Localization and Role

Consider the relationship between the quality of a robot’s localization and the accuracy of role assignment for the completed functional object. Flawless localization would result in every robot always having the correct role for its position. Without flawless localization, it is possible that a robot’s estimated position $\hat{x}$ would be sufficiently distant from its true position $x$ that it is assigned the wrong role. The greater the error in $\hat{x}$, ie, the greater $\|\hat{x} - x\|$ tends to be – the more likely it is that a robot will be assigned the incorrect role. Other factors that affect this likelihood are the distribution of robots throughout the object, and the complexity of the location-to-role-mapping specification; a pong game with only a small handful of roles is less complex than a remote control with five buttons, and far less complex than a full computer keyboard.

This section presents a method by which to calculate this role-assignment-error likelihood, given a fixed CSG specification of the location-to-role mapping and a localization system. With it, system designers can tune their localization to the needed accuracy – or simplify their location-to-role specification – as needed to achieve the desired levels of role-assignment accuracy.
4.5.1 A Localization System

I assume that each robot’s localization system provides \((\hat{x}, \hat{P})\) estimates; pairs of an estimated position and the estimated error in that position. If a localization provides only \(\hat{x}\), a single fixed covariance matrix \(\hat{P}\) could be calculated empirically based on the error \((\hat{x} - \hat{x}) = \hat{x}\) across sampled data, and the following calculations performed using \(\hat{P}\) as the \(\hat{P}\) for each point.

The first step is to produce a characterization of the performance of the localization system, in the form of a model which generates \((\hat{x}, \hat{P})\) pairs. These pairs should simulate what the localization system would produce once it has reached a steady state, under the assumption that \(\hat{x} - x\) is independent of \(x\) — that is, that the error in the robot’s position estimate is independent of the position itself.

The normal-inverse-Wishart (NIW) distribution is one such model, discussed in detail in Appendix A.1. Given a data set of empirically connected \((\hat{x}, \hat{P})\) pairs collected once the localization system is in a steady state, one can find the NIW distribution parameters which maximize the summed log-likelihood of all points in the data set:

\[
\arg\max_{\mu_0, \Lambda, \Psi} \sum^N \log \left( \mathcal{N}\left( \begin{pmatrix} \hat{x} \\ \hat{P} \end{pmatrix} \middle| \mu_0, \frac{1}{\Lambda} \hat{P} \right) \mathcal{W}^{-1}\left( \begin{pmatrix} \hat{P} \end{pmatrix} \middle| \Psi, \nu \right) \right) \tag{4.3}
\]

So parameterized, the random variates from this NIW distribution can be used to simulate \((\hat{x}, \hat{P})\) pairs coming from the localization system. If a system designer wanted to reduce the frequency of role-mapping errors, one option would be to reduce the determinant of the shape matrix \(\Psi\). Other models for the localization system could be similarly parameterized, and would have similar parameters to target for improvement.

For the Droplet swarm robotics platform described in Chapter 2, the NIW distribution parameters that characterize its localization system are:

\[
\mu_0 = \begin{bmatrix} -39.365 \\ -16.272 \\ 0.003 \end{bmatrix} \quad \lambda = 0.877 \quad \Psi = \begin{bmatrix} 65933 & 6941.6 & -16.25 \\ 6941.6 & 41196 & -18.81 \\ -16.25 & -18.81 & 6.90 \end{bmatrix} \quad \nu = 9 \tag{4.4}
\]
4.5.2 Calculating Error Likelihood

The second step is to use our localization system model to generate simulated robot configurations. Since \( \hat{x}^i = (\tilde{x}^i + \bar{x}^i) \) this amounts to generating \( N \) \((\tilde{x}^i, \hat{P}^i)\) pairs – one for each robot \( i \) – and adding the robot’s true position \( x^i \) to \( \tilde{x}^i \) to get a complete simulated position estimate \((\hat{x}^i, \hat{P}^i)\). The set of true positions \( x^i \) could be randomly generated (non-overlapping) robot positions, or any desired fixed configuration.

Finally, given a characterized-localization system model, a set of roles and regions which map to them, and a robot \((B_i)^t\)’s true position, we can write an equation for the expected value of the roles assigned to a robot; integrating the product of the NIW PDF and the set of roles for that estimate over its domain:

\[
\int \int f_{NIW}(\mu_0, \lambda, \Psi, \nu)(\tilde{x}^i, P) \ast \left( \hat{R}^i \bigg| x^i + \tilde{x}^i, P^i \right) d\tilde{x}^i dP
\]

(4.5)

where \( \left( \hat{R}^i \bigg| x^i + \tilde{x}^i, P^i \right) \) is the set of expected roles for robot \( i \) \((B_i)^t\), given a specific estimate from the NIW distribution. This is a vector of length equal to the number of roles, with each value the probability that the robot’s true position should be mapped to that role, based on the estimate. Using Equation (4.1) and recalling that \( R_i \) represents the region(s) which maps to \( r_i \), the \( i \)th role, this can be written out:

\[
\left( \hat{R}^i \bigg| x^i + \tilde{x}^i, P^i \right) = \begin{bmatrix}
\int_{R_1} f_N(x^i + \tilde{x}^i, P^i)(x) dx \\
\int_{R_2} f_N(x^i + \tilde{x}^i, P^i)(x) dx \\
\vdots \\
\int_{R_M} f_N(x^i + \tilde{x}^i, P^i)(x) dx
\end{bmatrix}
\]

(4.6)

for \( M \) different roles. With \( R \) a vector of roles based on the true position of \( B_i^t \) – 1 for roles \( B_i^t \) should have and 0 for roles \( B_i^t \)’s should not – the final quality of the role-mapping result for a single robot might be calculated as the distance between \( R \) and \( \hat{R} \):

\[
|\hat{R} - R|
\]

(4.7)

If I assume a certain initial configuration of robots – a set of true positions \( x_i^t \) for each \( B_i^t \) in a set of \( N \) robots, then the overall quality of the role-mapping is simply the average of the quality for each individual robot:

\[
\frac{1}{N} \sum_{i \in B_i} \left( \int \int f_{NIW}(\mu_{0, \lambda}, \Psi, \nu)(\tilde{x}^i, P) \ast \left( \hat{R}^i \bigg| x^i + \tilde{x}^i, P^i \right) d\tilde{x}^i dP \right)
\]

(4.8)
Rather than a result dependent on a specific robot configuration, recalling that the spatial computing model treats space as a continuous computational medium approximated by a discrete set of individual robots, it might well make sense to replace the sum in Equation (4.7) with an integration over the domain of the object (that is, the region which fully encompasses all possible robot locations for an object), denoted $O$:

$$\int_{O} \cdots \left( \int_{O} f_{NIW(\mu_{0}, \Psi, \nu)} \left( \hat{x}^{i}, P \right) \star \left( \bar{R}^{i} - R^{i} \big| x^{i} + \hat{x}^{i}, P^{i} \right) d\bar{x}^{i} dP \right) d\hat{x}^{i}$$

(4.9)

Further analysis and numerical integration results are needed in this area.

### 4.6 Communication Rate and Latency

This section concerns the relationship between the performance of a robot’s communication system and the effect of those individual performance characteristics on the maximum update rate of the functional object. Now, the properties an individual-robot designer controls are (for example): communication medium (IR, Bluetooth, Radio), physical placement of communication hardware, number of communication channels, supported communication rate, and medium-access control strategies. These choices affect the range and throughput of each robot’s communication.

The metric I identify to encapsulate these factors affecting communication is speed: how quickly can information propagate through the functional object? I use $V_{Comm}$ to denote this communication speed, which has units of $\frac{m}{s}$. Given $V_{Comm}$ and an object specification, I can begin to discuss relationships between $V_{Comm}$ and some object performance characteristics. For some role $r_{i}$, it is often useful to consider $T_{r_{i}}$, the maximum amount of time it takes information to propagate across all of the region(s) associated with $r_{i}$. This time can be calculated as a product of $V_{Comm}$ and the greatest distance between any two points in the region(s) associated with $r_{i}$.

Consider a case where all robots assigned to some role need to perform some task concurrently. In a game like pong, for example, you might want all robots which are part of that display to update together. First, this would require that the robots be synchronized, as is described in Section 2.2. With synchronization, the robots can all be programmed to update with some period $T_{p}$; at every time divisible by $T_{p}$. Naively, one might think it adequate to ensure that $T_{p} > T_{r_{i}}$, but that condition is neither necessary
nor sufficient. To explain: if a robot (asynchronously) receives information indicating that it (and all other robots with $r_i$) should update its display, that might occur immediately prior to a would-be update, whereas other robots would not get that information until as much as $T_{r_i}$ later. The solution to this, then, is for all messages to be time-stamped with the origin time for the relevant information: $t_o$. The soonest any robot receiving a message can safely update its display, then, is $t_o + T_{r_i}$. All robots should update at the first time $T$ divisible by $T_p$ and greater than $t_o + T_{r_i}$. This is how the latency, or delay between input and response, has as upper-bound related to $V_{Comm}$, though the update period $T_{pp}$ can be much smaller than that.
Chapter 5

Examples

5.1 Display

The simplest of the examples, this object is simply a display. There is only one region covering the entire display, making localization unnecessary. The only sensing or actuation required are the ability to sense and emit light. Synchronization keeps the robots together.

Algorithm 4 Pseudocode description of the full display algorithm.

\[
\begin{align*}
\text{Wait for bright light.} & \quad \triangleright \text{Bright light signals start of video to be recorded.} \\
i & \leftarrow 0 \\
\text{repeat} & \\
\quad \text{Frame}_i & \leftarrow \text{ReadColorSensors()} \\
\quad i & \leftarrow i + 1 \\
\quad \text{Delay}\left(\frac{1}{\text{framerate}}\right) & \\
\text{until Video end} & \\
\text{loop} & \\
\quad \text{for Frame}_i \in \text{Frames} & \\
\quad \quad \text{SetLED(Frame}_i\text{)} & \\
\quad \quad \text{Delay}\left(\frac{1}{\text{framerate}}\right) & \\
\quad \text{end for} & \\
\text{end loop} & \\
\end{align*}
\]

5.2 Remote Control

While still a simple example, this object makes greater use of the Droplet’s communication, and requires more hardware. The Droplets’ IR hardware is used to control the television. RGB LEDs are used to help the user differentiate between buttons, and the Droplet’s microphone is used to detect button presses: taps on the Droplet’s shell. Localization is used for role differentiation. If a Droplet with one of the “button”
Figure 5.1: The Droplet Display
Figure 5.2: The Droplet Remote Control in action. Left images, from top to bottom: different regions and associated roles; Droplets, placed but unpowered; Droplets, after Localization-based role determination is completed.
roles detects a button press, it broadcasts a message of this event. Any “body” Droplets, upon receiving such a message, repeat it. This method for spreading information throughout a system is described in the spatial computing literature \cite{24} as “gossip”. When an “emitter” Droplet receives a message, either directly from a “button” Droplet or second hand, it sends the appropriate IR sequence to the TV.

5.3 Mouse & Keyboard

This example object has a lot of similarity with the remote control, though there are many more buttons. Instead of an “emitter” role, one Droplet in this object has the “wired” role; the Droplet physically connected to the computer, which communicates any key presses and mouse-position updates to the computer. This object also serves as an example of the limitations imposed by communication: one key can only be pressed in succession as rapidly as it takes information to propagate from that key to the “wired” Droplet. The “mouse” role is associated with a large region, in which only a single Droplet is placed. This Droplet performs range-sensing broadcasts more frequently than the others, to allow for a faster update rate.
5.4 Pong

In some ways, the Pong-game object is similar to the Display. Most of the Droplets all have the same role; I call them “display” Droplets. These are all responsible for tracking and displaying the current position of the ball. The ball has a certain position, size, and velocity vector. At each time step, every Droplet which has heard about the ball propagates the ball forward using its velocity vector. Any Droplets close enough to the ball send out a message about the ball’s current status. This represents a compromise between too many messages (which suffers from congestion) and fewer messages (which increases the risk of “dropping” the ball).

In order to make the pong board dynamically responsive to changes in its shape, each robot must keep track of whether or not it is on an outer boundary. This is done by storing the positions of nearby Droplets, and looking at the angular distance between each nearby Droplet: angular distances greater than some threshold ($\frac{\pi}{3}$) result in the Droplet determining it is part of the boundary.

Specifically, the triad made up of $B_i^t$ and two nearby Droplets ($B_j^t$, $B_k^t$) makes two lines: $B_i^t \rightarrow B_j^t$ and $B_i^t \rightarrow B_k^t$. If there are no other known robots in the radial region between these vectors, and the angular distance between the vectors is greater than some threshold, then the two edges $B_i^t \rightarrow B_j^t$ and $B_i^t \rightarrow B_k^t$ are considered outer boundaries by $B_i^t$. These outer boundaries are only computed and stored locally based on the positions of nearby robots; they are never communicated explicitly.

Only the Droplets which make up an outer-boundary edge know about it, and thus they are the only Droplets which can accurately simulate a bounce off of that edge. As only nearby Droplets can form an outer edge, such a bounce can only occur when the ball is very close to the Droplets which know about it. When the ball bounces, at first only the Droplets which make up the bounced-off edge have accurate ball models. Once those Droplets send out their next message about the ball, any Droplet receiving that message learns of the updated ball and begins to propagate it forward through time. Implicit to this method is the assumption that the accuracy of a Droplet’s ball model is proportional to that Droplet’s distance from the ball. Thus, any ball messages from a Droplet which thinks itself closer to the ball than the receiving Droplet will replace the receiving Droplet’s ball parameters with the new ones. Messages from Droplets
Figure 5.4: Droplet Pong. Top left image shows a visualization of the Droplets’ positions, with red intensity periodically proportional to $x$ position and green intensity periodically proportional to $y$ position (period of 175 mm). Remaining images shows the progression of a ball through the pong board; Droplets light up white with an intensity proportional to how much the ball overlaps their position. The bottom two images are overlayed with a representation of the ball, which has a 120 mm diameter. The affect of localization errors is visible in the scattering of illuminated Droplets not part of the ball.
more distance are ignored.

The result of the systems described above is that it doesn't take much time before all of the Droplets are updating some version of the ball. However, many of those Droplets are tracking an outdated version of the ball, because one or more bounces have occurred but not yet reached them. However, since a bounce can only occur near a Droplet, Droplets with outdated versions of the ball aren't going to light up again. The ball will just keep moving away from them until they hear otherwise. The closer a Droplet is to the ball, the more likely that Droplet is to need to light up and/or manage a bounce of its own. But, that proximity also greatly increases the likelihood that the Droplet will hear a message with up-to-date information about the ball. One limitation imposed by this method is that the ball speed can not be any greater than the communication speed; otherwise information about the ball can not keep up. The progression of a ball through the pong board is visible in Figure 5.4.

The other roles are the goal, paddles, and controls. There is one control Droplet on each corner of the board. The controls serve as buttons which move that sides' paddle closer to them. The paddles are moving regions on each side of the board. They light up to indicate that they are paddles, and otherwise behave nearly the same as the “display” Droplets. The paddle region is always a subset of the goal region. The “goal” Droplets are nearly the same as “display” Droplets as well, except that if the ball crosses an exterior boundary for a goal Droplet, it triggers the game to end with one of the players victor.
Chapter 6

Conclusion

A variety of diverse research efforts work towards the dream of matter which can be programmed to take any shape. These efforts include the programmable matter project [1], [2], [6], [94] and swarm assembly [8], [95], [96].

For much of the swarm assembly and programmable matter research presented thus far, the robots’ work is done once they have finished forming the target shape. They wait, inert, until programmed in to the next shape. Motivated to leverage the collective ability of these robots to make the shaped object have not just shape, but functionality, I have made a variety of hardware and algorithmic contributions for distributed robotics. Specifically:

• I have co-developed a range-sensing system for small and low-cost robots which provides range, bearing, and relative orientation to the sensed robot all as part of the measurement [34].

• I have developed a holonomic motion system for low-cost robots, with an automatic calibration method to improve movement accuracy [33].

• I have adapted the Covariance Intersection method for Cooperative Localization [82] to account for the rare unmodeled faulty sensor readings which occur all-too-frequently in large collectives of real robots, creating a scalable and robust localization method [35].

Other research efforts work to solve the global-to-local problem. Inspired by the impressive feats of insects like bees and termites, how do we design and program discrete collections of individual robots to
achieve some collective task? These efforts include Amorphous/Spatial computing [16], [19], [26], [97], as well as Butera’s paintable computers [27], [28], and globally-acted devices like in [98].

Given how minimal an individual bee or termite is in terms of available sensing and computational power, how minimal can these individual robots be made in turn? This question has been given far less attention. Economic constraints and large quantities of robots desired for experiments in this area naturally lead to “fairly” minimal hardware, but it would be difficult to prove (for a given robot swarm and given collective task) that the swarm could not complete that task if it were any less capable.

In Chapter 4 my thesis also includes otherwise-unpublished work discussing some relationships I have identified between global performance and local hardware capability. These relationships come short of what would be necessary to prove minimality, yet still provide a framework to reason about the performance bounds imposed by distributed computation for functionality. Further, Chapter 4 presents some of the lessons I have learned in creating the variety of distributed and decentralized functional objects, all of which can be found in Chapter 5.

In addition to these contributions, I have also been the lead developer of the Droplet Swarm Robotics platform for nearly six years. Among other work, this has resulted in my collaborating with a variety of other researchers, helping them to implement their distributed algorithms on a real hardware platform. Specifically, these hardware implementations have been:

• For probabilistic task-allocation in the face of uncertainty [99].

• For distributed camouflage [12], [101].

• For chemistry education by using the Droplets (as atoms) to simulate chemical bond formation in to molecules [100].

Figure 6.1 shows images from these works.

Of course, there is much more which could be done. My individual published contributions all conclude with their own discussion of how that work might be further improved upon, or a discussion of (what I believe to be) natural directions for future inquiry. Further, the unpublished work I present in

\[\text{http://github.com/correlllab/cu-droplet}\]
Figure 6.1: Left: an image from [99] shows Droplets gathering to cooperatively extinguish a simulated “fire”, represented by the red region. Bottom-Right: an example of Acetic Acid (C$_2$H$_4$O$_2$) from [100]. Top-Right: An image from [12] showing a camouflage pattern of vertical stripes.
Chapter 4 is ripe for further experimentation and study. As new, lower-cost hardware becomes available, it is natural for new swarm robotics platforms, and new versions of existing platforms to be released. This will present new opportunities for research groups to compare the performance of distributed and decentralized algorithms on multiple swarm platforms, and lead in turn to richer understanding of the relationships between local and global performance.

I believe the time is ripe for the development of a full pipeline, using CSG [93] and a field-calculus-based language like Protelis [21] to compile a global description of an object and its functionality into individual real-hardware code for making that object, functionality and all. The contributions I have presented here – along with parallel related developments such as those cited – have the laid the groundwork to make such development possible.


[75] A. Prorok, A. Bahr, and A. Martinoli, “Low-cost collaborative localization for large-scale multi-
robot systems”, in Robotics and Automation (ICRA), 2012 IEEE International Conference on,


[78] A. Martinelli, “Improving the precision on multi robot localization by using a series of filters hier-
archically distributed”, in Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ Interna-

filter”, in Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, IEEE,

pp. 2369–2373.

[81] P. O. Arambel, C. Rago, and R. K. Mehra, “Covariance intersection algorithm for distributed space-
vol. 6, 2001, pp. 4398–4403.

cooperative localization using covariance intersection”, in Intelligent Robots and Systems (IROS),


Appendix A

Statistics

A.1 Normal-Inverse-Wishart Distribution

The normal-inverse-Wishart distribution is a compound probability distribution which first generates a covariance matrix $\Sigma$ from an inverse-Wishart distribution with scale matrix $\Psi$ and $\nu$ degrees of freedom:

$$\Sigma \sim \mathcal{W}^{-1}(\Sigma | \Psi, \nu)$$

and then generates a position error from a multivariate normal distribution:

$$\mu \sim \mathcal{N}(\mu | \mu_0, \frac{1}{\lambda} \Sigma)$$

with the generated covariance matrix $\Sigma$. In this case, $(\mu, \Sigma)$ is distributed according to a normal-inverse-wishart distribution (NIW) with parameters $\mu_0, \lambda, \Psi, \nu$:

$$(\mu, \Sigma) \sim \text{NIW}(\mu_0, \lambda, \Psi, \nu)$$

The probability-density function for the normal-inverse-Wishart distribution is the product of the multivariate normal and inverse Wishart probability-density functions:

$$f(\mu, \Sigma, \mu_0, \lambda, \Psi, \nu) = \mathcal{N}(\mu | \mu_0, \frac{1}{\lambda} \Sigma) \mathcal{W}^{-1}(\Sigma | \Psi, \nu)$$

A.2 Sampling from Multivariate Normal

In order to sample from an $N$-dimensional multivariate distribution, one first needs $N$ points independently sampled from the standard normal distribution. Using the Box-Muller [102] transform, we
can convert \( N \) points sampled from the standard uniform distribution to the needed standard-normal distributed points. Specifically, with \( u_1, u_2 \overset{\text{i.i.d.}}{\sim} U(0, 1) \), calculate \( n_1, n_2 \) according to:

\[
\begin{align*}
    r &= \sqrt{-2 \ln u_1} \quad \theta = 2 \pi u_2 \\
n_1 &= r \cos \theta \quad n_2 = r \sin \theta
\end{align*}
\]

and \( n_1, n_2 \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1) \). As any linear combination of normally distributed variables is also normally distributed, a linear transform exists to transform variables sampled from one multivariate normal to another.

Using Equation (A.1), we can produce \( n_1, n_2, n_3 \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1) \) which is equivalent to:

\[
\begin{bmatrix}
    n_1 \\
n_2 \\
n_3
\end{bmatrix} \sim \mathcal{N}
\begin{pmatrix}
    0 & 0 & 0 \\
    0 & 0 & 1 \\
    0 & 0 & 1
\end{pmatrix}
\]

(A.2)

To sample points from the \( \mathcal{N}(\hat{x}_i^t, P_i^t) \) distribution, then, we need to compute the transformation matrix \( L \) such that \( LL^\top = P_i^t \), and then

\[
\begin{bmatrix}
    \hat{x}_i^t + L
\end{bmatrix}
\begin{bmatrix}
    n_1 \\
n_2 \\
n_3
\end{bmatrix}
\]

will have the appropriate distribution. I use the cholesky decomposition of \( P_i^t \) to calculate \( L \), but other factorizations could also be used [103]. Using homogenous coordinates, this can be written:

\[
\begin{bmatrix}
    \hat{x}_i^t \\
    \hat{y}_i^t \\
    \hat{z}_i^t \\
    1
\end{bmatrix} =
\begin{bmatrix}
    L_{3 \times 3} & \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} \\
0 & 0 & 0 & 1
\end{bmatrix}
\sim \mathcal{N}(\hat{x}_i^t, P_i^t)
\]

(A.4)

This construction as a transformation matrix is helpful when we use the sampled point to determine the robot’s role. I also note that the inverse of a homogenous transformation matrix is not a typical matrix inverse, rather:

\[
T_{4 \times 4} =
\begin{bmatrix}
    R_{3 \times 3} & d_{3 \times 1} \\
    0_{1 \times 3} & 1
\end{bmatrix}
\quad \Rightarrow \quad
T^{-1} =
\begin{bmatrix}
    R^{-1} & -R^{-1} d \\
    0 & 1
\end{bmatrix}
\]

(A.5)
Appendix B

Droplet Swarm Robotics Platform
Figure B.1: Top view of Droplet Circuit Board (3D Rendering)
Figure B.2: Bottom view of Droplet Circuit Board (3D Rendering)
Figure B.3: Droplet Shell Parts. Both interactive in the Adobe® Acrobat® PDF Reader, or other PDF Readers supporting embedded 3D. Click to activate.
Figure B.4: Schematic for the Droplet Swarm Robotics Platform.