École Nationale Supérieure des Techniques Avancées
2011-2012

Industrial Project

Applying Parallel Distributed Computing to Swarm Robotics

Lieutenant DELAUNAY Benoît

Class 2012, IASE

16 March 2012

Remerciements / Acknowledgments

Je remercie en premier lieu Monsieur Luc Jaulin, professeur à l'ENSTA-Bretagne, pour m'avoir permis de travailler sur ce projet qui me tenait particulièrement à cœur, ainsi que pour l'ensemble de ses remarques et suggestions.

Je tiens également à remercier mes camarades de la promotion 2012, et notamment tous ceux qui auront partagé mon quotidien au cours de ces trois années passées à l'ENSTA-Bretagne.

Je tiens à remercier tout particulièrement Monsieur Olivier Debant, camarade de promotion et également ami, qui m'aura accompagné durant la plupart des projets et autres travaux j'ai pu mener durant ces trois ans à l'ENSTA-Bretagne.

I would like to thank Mr. Luc Jaulin, professor at ENSTA-Bretagne, for having allowed me to work on that project, that did really interest me, as well as for his remarks and suggestions.

I would also like to thank my classmates, in particular those who shared my daily life for these three years at ENSTA-Bretagne.

I would like to thank especially Mr. Olivier Debant, classmate and friend, who worked with me for most of the projects and other works for these three years at ENSTA-Bretagne.

Contents

Remerciements / **Acknowledgments** **1**

Introduction **3**

1 Investigations **4**
1.1 Starting point: an example of centralized swarm . . . 4
1.2 Towards decentralization . . . 5
1.3 Fork and Join . . . 5
1.4 Distributed state machines . . . 6
1.4.1 Leader election . . . 6
1.4.2 State machine replication . . . 8
1.5 Limits of centralized algorithm distribution . . . 8
1.6 Potential energy and Forces . . . 9

2 Implementation **12**
2.1 Means and Objectives . . . 12
2.2 Network communications . . . 12
2.3 Consensus protocols . . . 13
2.4 *Fork-Join* . . . 13
2.5 Swarm simulation . . . 14
2.5.1 Overview . . . 14
2.5.2 Parameters and Objectives . . . 15
2.5.3 Choice of a potential function . . . 16
2.5.4 Robots' state equation . . . 18
2.5.5 Global Swarm Management . . . 19
2.6 Perspectives . . . 19
2.6.1 Software structure . . . 19
2.6.2 Realism . . . 20
2.6.3 Swarm management . . . 20
2.6.4 Distributed intelligence? . . . 20

Conclusion **21**

Introduction

Swarm robotics is an approach that emerged few years ago. It consists of a large set of robots, that are individually very basic and can only perform trivial tasks, but that are also able to communicate, and thus, with suited behavioural algorithms, that may cooperate in order to achieve a common goal. The principle of swarm robotics is the same than that of animal societies: alone, an ant or even a human being is meaningless, unable to achieve something more complex than finding food or ensuring its own safety but, when they form societies with fellows, they become able to perform greater deeds.

Swarm management algorithms are surely the most challenging point in swarm robotics. Indeed, whether a swarm will be able to do what is expected from it, or not, relies utterly on them. Conventional approaches are based on a centralized structure: each ounce of intelligence of the system is held by a particular element, a server to whom each robot is connected. This server retrieves all the data about the robots, especially the measurements of their sensors, then computes them in order to determine what should be done at the moment by each robot to achieve the objectives, and finally sends each individual the corresponding instructions (*e.g.* to move in a certain direction, to reach a certain area...). This kind of centralized approach is quite easy to implement, but lacks robustness. Indeed, shall the server experience a problem, the swarm is literally decapitated. In addition, it requires a large network bandwidth, at least at the level of the server. For large swarms, this bandwidth as well as the computing capabilities that the server must have in order to manage the entire swarm well, may be hard to obtain with the current technology and therefore be very expensive.

Hence, two axes of progress appear: on one hand, the robustness of the system and, on the other hand, the computation and communication capabilities. In this project, solutions based on parallel distributed computing have been investigated, and some of them have been eventually implemented. They rely all on the same idea. Nowadays, on-board electronics offer more and more possibilities, especially in terms of computing capabilities. Therefore, distributing the workload between all the robots appears as an interesting and feasible solution. It would have several virtues, in particular to take advantage of the individual computing capabilities of the robots, as well as to make the swarm less subject to individual failures, mostly network malfunctions.

1 Investigations

1.1 Starting point: an example of centralized swarm

Last year, an industrial project entitled *Buggies Game* [1] dealt with cooperation problems in centralized swarm robotics. The swarm, that consisted in several little autonomous robots, the so-called *buggies*, was entrusted with surrounding a human player on a soccer field. Each robot was equipped with a *smartphone*, that was meant to perform some measurements, mostly positioning using GPS, as well as to make the robots able to communicate with other devices through *Wi-Fi*. The motors of the *buggies* were, as for them, commanded by an *Arduino* card that was also connected to the same *Wi-Fi* network than the smartphones.

This swarm was ruled by an identified machine, a laptop, that retrieved all the data measured by the smartphones, then computed them in order to determine how each *buggy* should move, and lastly sent each *buggy* its instructions. The intelligence embedded on the *buggies* was quite limited: knowing its position and its orientation, each *buggy* should only be able to move in a certain direction or to a certain position.

Moving on to the swarm's behavioural algorithm, it relied on linear programming [2]. On one hand, there was a quantity to minimize: the distances between the *buggies* and the player [3]. On the other hand, several constraints had to be observed: every *buggy* shall remain within the limits of the field, two *buggies* cannot get too close to one another, and any *buggy* cannot be told to move to a certain place unless it can reach it before the player. To make the problem fully linear [4], the Manhattan distance [5] has been used instead of the conventional Euclidean distance [6], and circles that were used for determining where a *buggy* could arrive before the player [7] have been approximated by polygons. The problem posed by the absolute value introduced by the Manhattan distance has eventually been gotten around by subdividing the original problem into subproblems in which additional constraints made known the sign of what was within the absolute value expressions. Finally, a set of linear problems was obtained. They were all solved and the best solution [8] was selected. It gave the position each *buggy* should reach, and that the laptop sent to each robot of the swarm.

The following figure (1.1) is meant to give an overview of the swarm behaviour. The position of the player (in red) assumed to be known, the *buggies* (in blue) try to surround him/her. The problem consists actually in finding the lowest positive real ρ such that the *buggies* may arrange themselves in a regular way around the circle of center the player with radius ρ, before the player may escape from this area. In *Buggies Game*'s approach, the target positions of the *buggies* are calculated in a local polar basis, whose reference point is the player. To make the problem linear, the polar angle θ of each target position is initially fixed. However, to ensure that the *buggies* will circle the player as much as possible, several configurations (*i.e.* values of θ for each *buggy*) are considered.

---

[1] *Jeu des Buggys*, by DEBANT Olivier and DELAUNAY Benoît, ENSI 2 students at ENSTA-Bretagne, 2011

[2] To put it simply, an optimization (*i.e.* minimization or maximization) of a linear function subjected to linear constraints (*i.e.* inequalities)

[3] These distances were computed in a certain way in order to obtain a single function to be minimized

[4] And avoid the resolution of an $\mathcal{NP}$-hard problem...

[5] The L_1 distance between two points $A(x_A, y_A)$ and $B(x_B, y_B)$ is defined by $d_1(A,B) = |x_B - x_A| + |y_B - y_A|$

[6] The L_2 distance between two points $A(x_A, y_A)$ and $B(x_B, y_B)$ is defined by $d_2(A,B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

[7] If we consider a *buggy* B, a player P and some point M of the field, and if we set down v_B the velocity of the *buggy* and v_P that of the player, then the *buggy* may reach the point M before the player if and only if $\frac{\|\overrightarrow{BM}\|}{v_B} < \frac{\|\overrightarrow{PM}\|}{v_P}$; it may be shown that there is a point $I(x_I, y_I)$ and a positive real R such as this relation is equivalent to $(x_M - x_I)^2 + (y_M - y_I)^2 - R^2 < 0$, which defines a disk of center I with radius R

[8] That is the one that makes the *buggies* advance the closest possible from the player, assuming he/she does not move

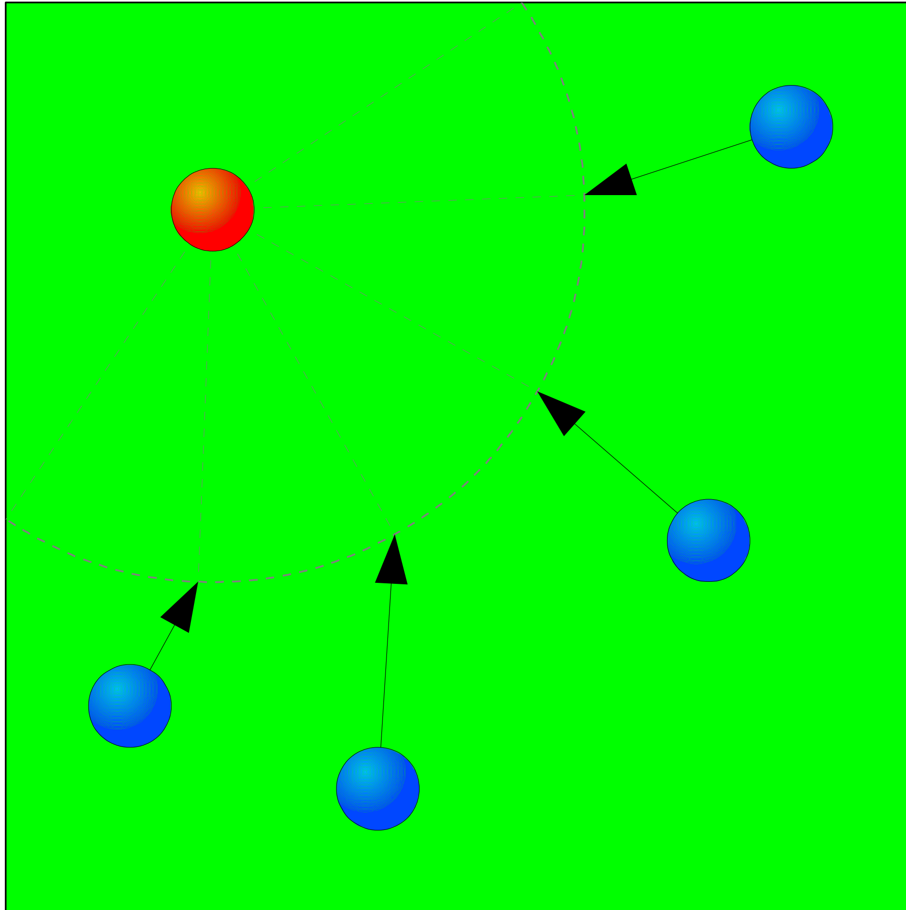

Figure 1.1: Overview of *Buggies Game*'s surrounding algorithm

1.2 Towards decentralization

In this project, the initial idea was to decentralize *Buggies Game* behavioural algorithm. As explained in the introduction, two axes of evolution had to be explored: the distribution of the computation, and the swarm's robustness against individual failures.

1.3 Fork and Join

Buggies Game behavioural algorithm presents a clear parallelism: the most significant part of the computation consists in solving independent subproblems. In other words, the solving of a subproblem does not rely at all on that of another. It may therefore be imagined that, in the swarm, different robots could perform the computations concomitantly. Even more than that, it might also be imagined that a single *buggy* with a multi-core processor [9] could process several solving at once. Hence, the idea would be to introduce an abstract layer between the set of data to be processed and the hardware that may be used to perform the computation. Basically, there would be on one hand a set of linear problems to be solved, and on the other hand a pool of processor cores, the additional layer making the link between them. In this way, assuming the network latent period negligible in comparison to the computation time of a single subproblem, the total duration of the global problem solving may become significantly swifter. Figure 1.2 shows how the *fork-join* approach might be used to enhance *Buggies Game*'s behavioural algorithm without changing it that much.

[9] Smartphones' processors are most of the time multi-core processor - *i.e.* they are meant to execute tasks in parallel

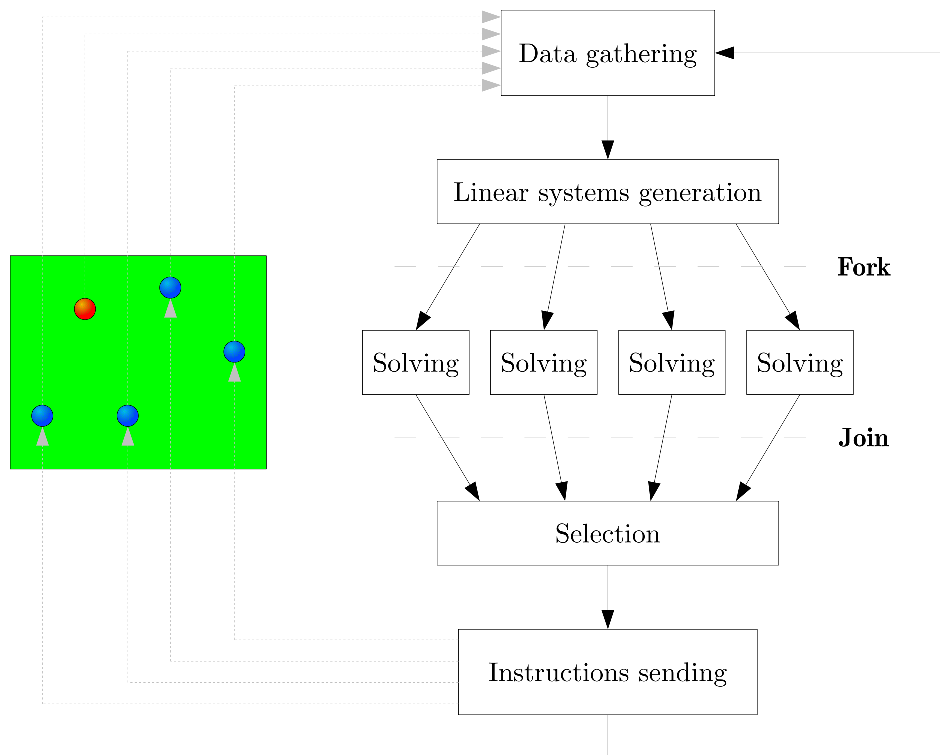

Figure 1.2: Possible *Fork-join* integration in *Buggies Game* behavioural algorithm

1.4 Distributed state machines

The *Fork and Join* approach makes it possible to take advantage of the swarm's parallel computing capabilities. However, this solution does not bring anything in terms of robustness. Indeed, if any device involved in the computation crashes along the way, the system cannot recover. In addition, provided there is a way to choose one, an identified leader is still required to gather all the data needed for the computation, prepare the linear systems to be solved, distribute them, and eventually compute the final results and transmit them. In a way, this leader would play the role of the laptop of the centralized approach. The objective for the swarm is therefore to be able to choose a leader, to distribute the computation, and to recover if and when any robot involved in it fails.

1.4.1 Leader election

Leader election in a network is an issue that has been studied for a few decades. Some distributed algorithm have been proposed to solve it, the most famous being probably the *Paxos* and *Chandra-Toueg* consensus algorithms. I actually focused on Leslie Lamport's [10] *Paxos* algorithm for a few reasons. Firstly, it is well-suited to the network formed by the robots: a network of unreliable processors [11] communicating through asynchronous messages [12] [13] [14]. Secondly, it was widely studied in the scientific literature, at least much more than *Chandra-Toueg* algorithm. Thirdly, it admits some extensions that enhance some aspects of the algorithm. Among them, the *Byzantine Paxos*, that makes the system robust against *byzantine* failures [15], has especially caught my attention. Indeed, as a future engineer of the French Defence Procurement Agency, I cannot utterly elide this security issue even though it is not really the purpose of this project.

[10] Leslie Lamport (born in 1941) is an American computer scientist best known for his work in distributed systems
[11] That are processors that may fail during a computation for whatever reason, delivering no result
[12] Asynchronous means that no acknowledgement is delivered at the reception of a message
[13] UDP (asynchronous protocol) has been preferred to TCP (synchronous protocol) because it allows multicast diffusion
[14] Some consensus algorithms work only with synchronous communications
[15] *Byzantine* failures are intentional failures meant to sabotage the system, *e.g.* a malicious intruder that would have been accepted in the network as a normal and trusted element, and that would actually deliberately issue mistaken results

The *Paxos* protocol distinguishes different kinds of "logical" actors [16], which are usually [17] :

- The client, that solicits a consensus in order to access a distributed resource, and waits for a response
- The proposer(s), contacted by the client, that advocate(s) the client's request to the acceptors
- The acceptors, that choose the value that will be eventually retained
- The learners, that will perform the action requested by the client (*e.g.* to create or modify a shared resource) and that will eventually report it to it

The fail-safety of this protocol relies on the multiplicity of the actors. One of its prerequisite is that any message sent to a particular actor must be sent to every actor of the same kind, which may be simply achieved by using multicast diffusion [18]. In this way, if and when an isolated actor crashes, its team-mates continue to perform their tasks and send their results, as planned. The receiver of these messages will consider only the results of a majority: for instance, if only three out of a pool of five acceptors answer to a proposer, this proposer will not consider any problem occurred in the process, no matter what the results of the two remaining acceptors, even if they are missing. The principle is similar in the case *byzantine* failures are also considered. Moreover, in the *Paxos* approach, once a value has been accepted, *i.e.* when the consensus has been found, it does not change any more. Figure 1.3 illustrates how the consensus is found between the proposer(s) and the acceptors.

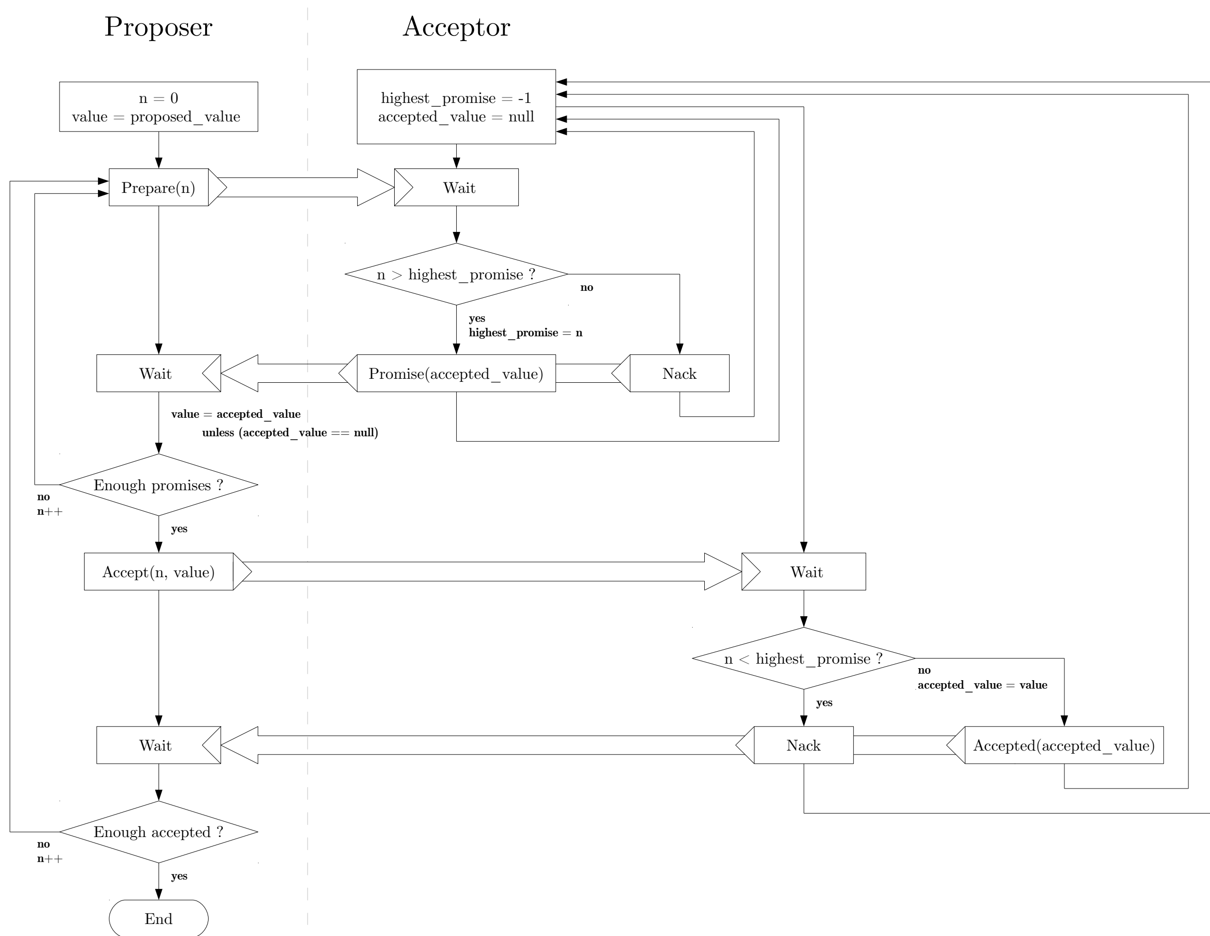

Figure 1.3: Consensus process in the *Paxos* approach

[16] A "physical" actor may play several "logical" roles, which is moreover generally the case
[17] *Paxos* variants may introduce other actors
[18] This is why UDP has been preferred to TCP

1.4.2 State machine replication

One application of such consensus algorithms is state machine replication, which consists in executing a same state machine, jointly, on different computers. Basically, a state machine is a succession of states and transitions, which may require some entries as well as generate some outputs. Thus, in order to make any replication self-consistent, the different computers must agree on the entries, which may be performed by using the *Paxos* algorithm, for instance. Provided they receive exactly the same entries in the same order, identical state machines will arrive at the same state having generated the same outputs, even if they are executed independently on different computers. In addition, such state machine replication benefits from the fail-safety of consensus algorithms, which implies that isolated failures have no serious impact on the system unless they concern a majority of the performers.

State machine replication may also offer some extra features, for instance the possibility for a computer to join the pool of performers or to quit it. Joining may be achieved through a "state transfer" functionality, which allows an outsider to get, at a given time, all the parameters about the execution of the distributed state machine. The performers being independent in their work between consensus phases, "state transfer" is typically implemented using checkpoints, that correspond to coordination steps. When a performer joins or quits the pool, every actor involved in the replication shall know about it in order to update its notion of "majority".

1.5 Limits of centralized algorithm distribution

Well, as explained in the two previous sections, *Buggies Game* behavioural algorithm may be adapted in order to achieve two objectives: using the number of robots to achieve both a high degree of parallelism in the computation and some robustness against individual failures.

An interesting point in the combination between *fork-join* and state machine replication is that it requires no major modification in the behavioural algorithm: most of the improvements are only visible at the implementation level. However, it does not solve all the issues. In particular, the problem posed by the gathering of some data about all the robots, which is needed to work out the linear problems to be solved, still remains. It has even become more problematic since the devices used to perform this, typically smartphones, have not the computing capabilities of laptops, which implies that they may not process as many requests as standard computers in the same amount of time. As a result, the size of the swarm may be limited not only because of the network bandwidth, but also because of the data processors. Actually, this problem may be partially solved by lowering the responsiveness of the swarm, which would give more time for processing the requests and performing the computation. However, this leeway is rather limited since the replication of the state machines used for the computation slows already the system [19].

In any case, robustness has a cost in terms of velocity, even though its impact may be minified by the parallelization of the computation. Assuming the system well parametrized [20] and not subjected to too many failures, this is nevertheless a viable solution.

[19] Depending on the network load, consensus protocols may take some time

[20] Regarding the size of the swarm, the size of the pools of actors in the consensus protocol, the timeouts...

1.6 Potential energy and Forces

Well, the major sticking point of *Buggies Game*'s approach is that knowing everything about the entire swarm is needed prior to determining the best suited behaviour for the singlest robot. Actually, if the behaviour of that singlest robot could be calculated with only a partial knowledge of the swarm, for instance the robots in its close neighbourhood, then the swarm could become very large without increasing too much the need for network bandwidth and the workload of the robots. In practice, it may be imagined that each robot knows the relative position of its neighbours [21]. This may be achieved, with an *ad-hoc Wi-Fi* network, by flooding one's neighbours with one's GPS coordinates. Without GPS, a solution involving UWB [22] beacons might also be considered.

Physics may be a source of inspiration. In particle physics, the motion of a particle is governed by the forces to which it is subjected, these forces being exerted by every particle of the space. By doing some approximations, it is possible to model these forces, or the potential energy they are deriving from [23]. For instance, the interaction between a pair of neutral atoms or molecules may be approached by the Lennard-Jones potential. Figure 1.4 gives the form of this potential, which depends solely on the distance r between the atoms or molecules. A potential curve is easily interpretable: anything subjected to a potential energy tries to minimize it. In the case of the Lennard-Jones potential, it means that two particles will always try to be and remain at a precise distance from each other, this distance corresponding to the value of r for which the value of the potential is minimal.

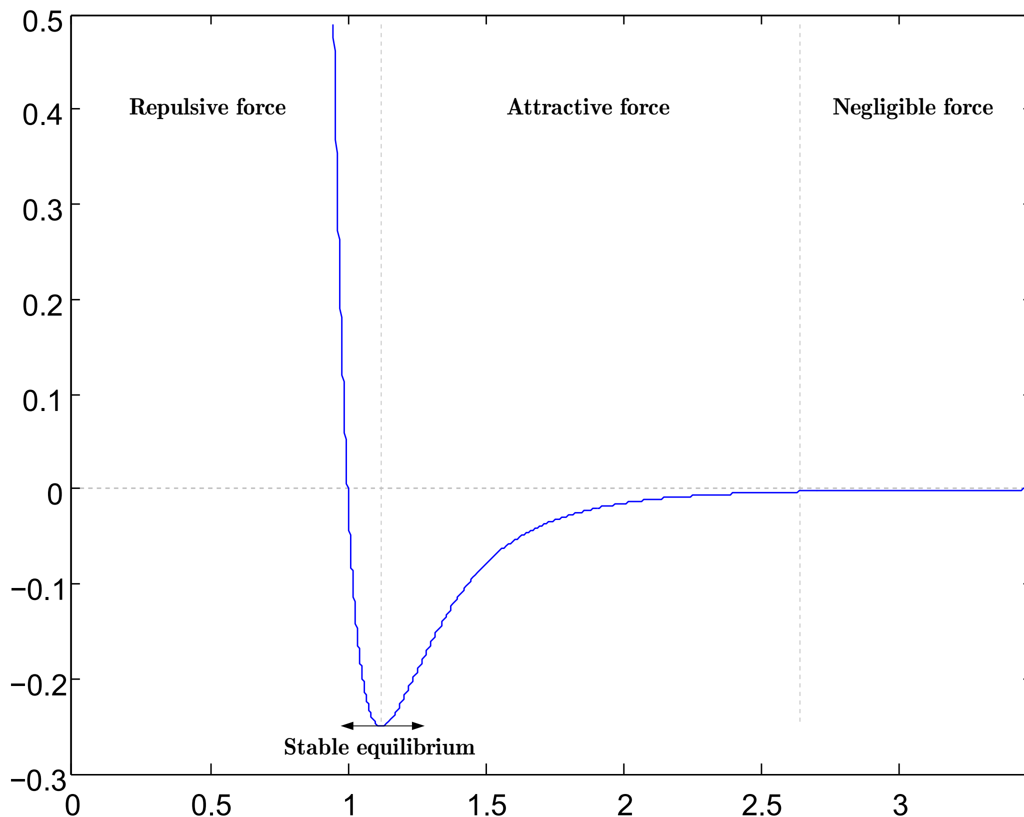

Figure 1.4: Lennard-Jones potential: $V_{LJ}(r) = 4 \cdot E_0 \cdot \left(\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6} \right)$
(above, $4 \cdot E_0 = 1$ and $r_0 = 1$)

[21] That is their distance from it as well as the direction in which they are

[22] Ultra-Wide Band, a radio technology that can be used for short-range high-bandwidth communications, one of its application being positioning systems

[23] A force $\overrightarrow{F}$ is said to derive from a potential V if and only if $\overrightarrow{F} = -\overrightarrow{\nabla} V$

The Lennard-Jones potential might be used for our problem. In particular, the strong repulsion force exerted between particles that are too close from each other is a simple way to avoid collisions between robots. However, the force deriving from this potential is not attractive enough beyond a certain distance. As a result, if a robot moves too far away from the swarm, because of an inertia effect or whatever reason, it may become utterly disconnected from the other robots, unable to return to the swarm again. Such a scenario would be made easier by the fact that a robot has a limited visibility of its neighbourhood: whatever the potential function may be, any robot that is too far away does not exert any force at all. Thus, it might be considered to choose a potential that will guarantee a strong attractive force for high values of r. In this way, any robot that would move too far away from its neighbours would be immediately called back before becoming out of range. This would ensure the cohesion of the swarm. Figure 1.5 shows an example of such a potential.

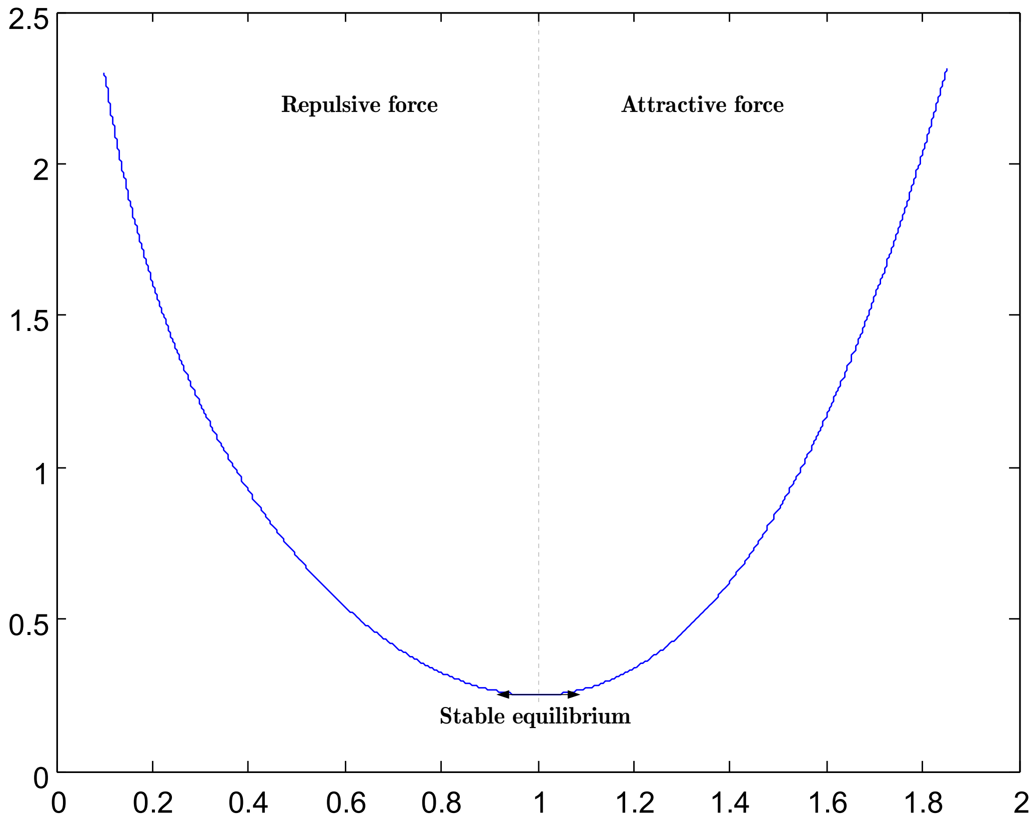

Figure 1.5: A potential that might be used for our problem: $V(r) = \frac{r^4}{4} - \ln(r)$

With this approach, robots that are within range of each other exert a reciprocal attractive or repulsive force, so that two close neighbours will eventually remain at an arbitrary distance, depending on the way the potential is calculated. Figure 1.6 illustrates the way forces are exerted between robots.

A definite asset of this method is the natural distribution and robustness of the behavioural algorithm: each robot decides of its behaviour on its own, so should a robot fail, there is almost no impact on the swarm, and even no risk of collision provided each robot detects its neighbours trough a radar. However, *fork-join* and state machine replication have not become useless. Indeed, *fork-join* is still an interesting mean to take advantage of the multi-core processors that equip present electronic boards and smartphones. State machine replication may also be used for controlling the swarm, at a high level, for instance by introducing an external potential that would make the swarm move to a certain location, or make it take a certain shape. All this will be discussed more precisely in the next part.

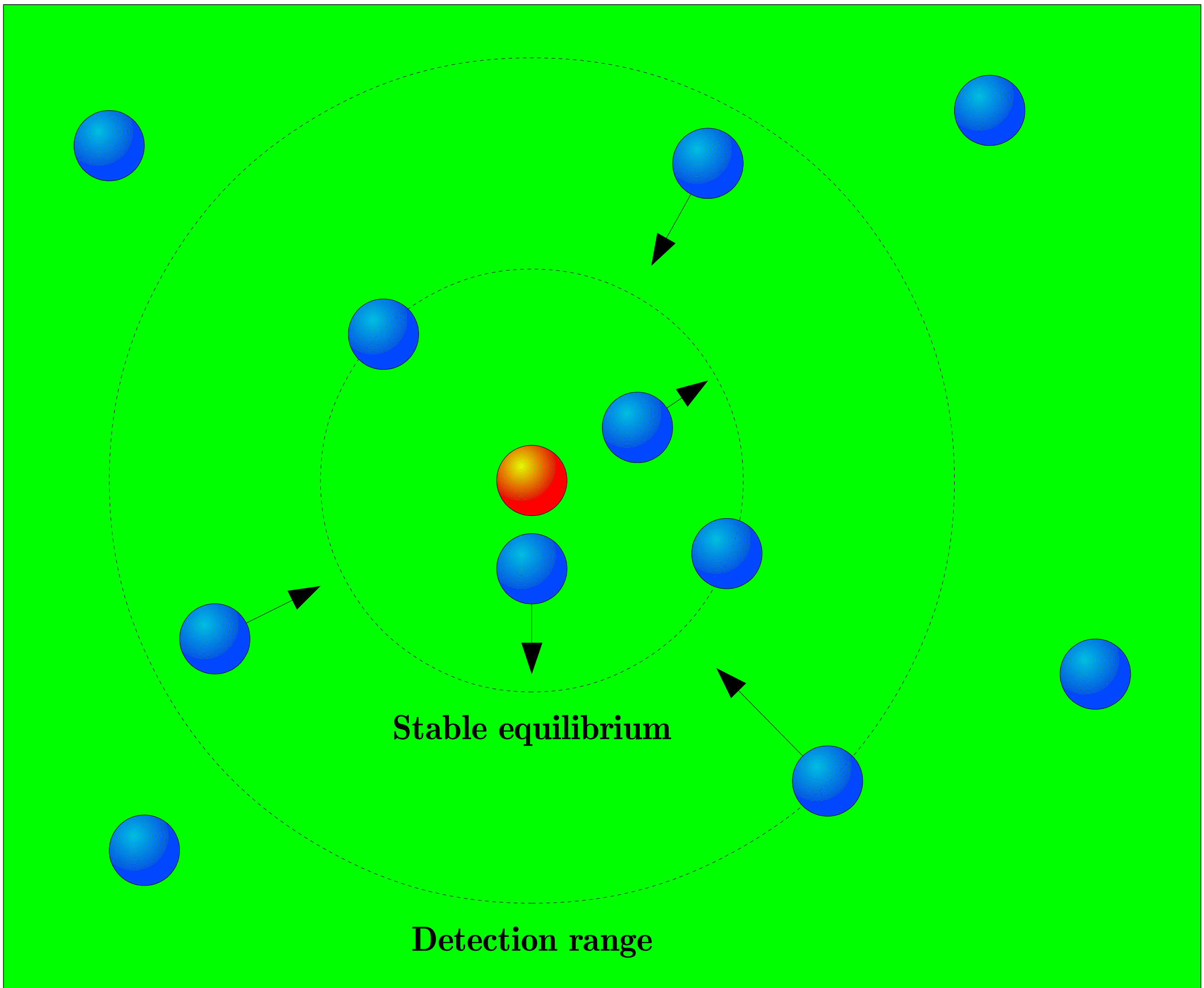

Figure 1.6: Overview of the behavioural algorithm using potentials
(above, only the forces exerted by the red robot on the blue ones have been represented)

2 Implementation

2.1 Means and Objectives

The previous part of this study dealt with theoretical concepts that might be used to achieve our goal: managing a swarm of robots in an utterly decentralized way. This is a part of this project. The other part consists in implementing these approaches, at least partially, in order to determine whether they are viable, and to what extent their results may match our expectations.

The purpose of this project being mostly related to algorithmics, the implementation has been performed on a simulated swarm of robots. A more advanced implementation involving real robots could be considered, using the communication and command routines worked out as part of *Buggies Game* project. However, since developing on smartphones is nowadays much more time-consuming than developing on traditional computers [1], this extension will not be studied here.

The implementation will therefore be carried out using the *Java* programming language. In this way, it may be ported on *Android* smartphones without any modification but, if any, regarding the graphical interface.

This implementation part has been performed in parallel with the theoretical investigation part. As a result, it comprises several parts that are not clearly connected with each other. Generally, every point presented below corresponds to a point tackled in the investigations part.

2.2 Network communications

In swarm robotics, communication is naturally one of the first issues that has to be considered. In *Buggies Game* project, communications between the robots and the server rely on a synchronous network protocol, TCP. However, this solution is not really suited to this project. Indeed, when it comes to consensus protocols [2] or even simply to joining a swarm whose IP addresses (*i.e.* those of the robots constituting it) are not precisely known [3], multicast is needed. Some experimentations involving UDP have therefore been conducted [4], in order to obtain very basics UDP server and UDP client able to exchange messages. Unicasting and broadcasting transmissions have been tested, with success.

As pointed out in *Buggies Game* report, UDP is not currently supported by the *WiFly* library, used for communicating with an *Arduino* card through *Wi-Fi*. Several solutions may be considered to get around this issue: wait for the *WiFly* library supporting UDP, develop a driver to use UDP with a *WiFly* module, use an other *Wi-Fi* module for *Arduino* (if any), or do not use *Arduino* cards at all [5].

On this basis, an embryo of *Kademlia* servant has been implemented. *Kademlia* is a distributed hash table, initially designed for decentralized peer-to-peer computer networks. Within the scope of this project, I thought it might be a good idea to use it to store and retrieve pieces of information in an utterly distributed way all over the network [6]. Moreover, the induced complexity is logarithmic, which means that just one additional iteration is needed for storing or retrieving something when the size of the network doubles. This shall be seen as a tool for creating a unique shared and distributed memory for the entire swarm.

[1] This is at least what was noticed during *Buggies Game* project

[2] That require that any message sent to an actor has to be sent at the same time to the entire pool of actors

[3] Typically, only the IP address of the LAN dedicated to the swarm will be known

[4] Sources may be found in the package *network.example*

[5] Since *Arduino* cards are only used for generating PWM signals (to command the motors), they may be avoided provided this functionality is performed by an other device, for instance an *Android* smartphone equipped with a IOIO board

[6] I already simulated a *Kademlia* network as part of an other project, in 2008

2.3 Consensus protocols

Even on a single machine, accessing safely to a shared resource requires synchronization. This rule shall therefore be naturally applied to distributed systems as well. Consensus protocols are a mean that may be used for implementing distributed synchronisation mechanisms. For instance, *Google* has developed a distributed lock mechanism, *Chubby*, that relies on the *Paxos* algorithm.

Distributed synchronisation mechanisms may also be used for coordinating the execution of a distributed algorithm. State machine replication is an example of application.

As I explained it in the previous part, I chose to focus on the *Paxos* algorithm. From various descriptions I read about it, I undertook to implement and integrate it within my simulator. Actually, I was unsure about my correct comprehension of the protocol. In particular, I did not see very well how the system worked when several proposers advocated different values, especially when some of them failed and rejoined the protocol during the process. This is why I used *Rational Rhapsody Developer for Java* to carry out a first implementation, before considering integrating it into the simulation. Indeed, even though I never had any intention to use it for code generation purposes [7], *Rhapsody* is a well-suited tool for implementing algorithms using state machines. In particular, its animated sequence diagrams give a clear view of the system, and especially of the exchanges of messages between actors. This was precisely what I was looking for.

In this simulation, whose operating principle is given by figure 1.3 (see page 7), there is a pool of five proposers dealing with a pool of five acceptors [8]. At the beginning of the process, each proposer is given a different number, which is its initial proposal. When the process is launched, the different proposers try concurrently to make their value accepted by the acceptors [9]. As soon as the proposal of a proposer has been accepted, both the number of the proposer and its final proposal are printed on the console, and the proposer stops its execution. The consensus ends when every proposer has made a proposal that has been accepted. To give spice to the process, I also made every proposer fail between the reception of enough *promises* and the sending of *accept* requests with a certain probability (20 %). In any case, after many replicas, using the information printed on the console, I noticed that the five proposers always agreed on the same value.

2.4 *Fork-Join*

As it will be presented a bit later, the simulated swarm of robots has been managed using the potentials method, that was presented in the previous part of this report. In this approach, each robots performs its own computations for determining its own behaviour. Hence, the computation is distributed but, strictly speaking, not shared between different robots. However, as it has already been explained, this does not make the *fork-join* method useless, since the computation may still be parallelized between the different cores of mutli-core processors used in smartphones or whatever electronic board equipping the robots.

Since 2009, two particular packages may be found on the fringes of the *Java 6* standard library: *jsr166y*, that contains a fine-grained parallel computation framework using *fork-join* processing, and *extra166y*, that proposes an array-like structure supporting parallel operations called *ParallelArray*.

[7] In this case, because the integration of the generated code within my hand-written project would have raise too many issues, that is I would probably have to rework few parts of the generated code; in addition, I am not really fond of default code generation templates, especially when I have to rework a code generated using any of them

[8] These numbers may be changed

[9] This value may change during the process; it is besides the principle of the *Paxos* algorithm when different proposers do not initially agree on a same value

Parallel computation is performed using a pool of threads, called a *ForkJoinPool* (package *jsr166y*). To put it simply, a *ForkJoinPool* is a set of threads, called *workers*, that are used for performing any operation required on a *ParallelArray*. When it is created, any *ParallelArray* is associated with a *ForkJoinPool*. Different *ParallelArray* may rely on the same pool of *workers*. This is usually the case. Indeed, the purpose of a *ForkJoinPool* is to provide a set of *workers* offering the best parallelism for performing computations, on *ParallelArray* or whatever structure that uses such a pool. The number of *workers* is automatically adjusted depending on the context, that is initially the number of available cores, but may increase if a worker is blocked, because of a synchronization mechanism for instance, and decrease when there are too many threads running regarding the number of available cores. This adjustment is internal to the pool: at the level of a *ParallelArray*, the number of *workers* does not matter at all. This is what makes *ParallelArray* especially interesting: provided they may be performed element by element independently, parallel operations may be considered on a entire set of data without having to think precisely about it at the implementation time.

Unfortunately, *jsr166y ForkJoinPool* experiences an awkward issue with recent versions of *Java 6* and *7*: the size of the pool still increases, no thread being destroyed at any time as it should be. Originally, both packages *jsr166y* and *extra166y* were planned to be integrated in the *Java 7* standard library. Actually, only *jsr166y* has been and, surprisingly, the version contained in the new standard library works fine. So I solved the issue by reworking the source code of every *extra166y*'s class, in order to make them use the fine *Java 7*'s *ForkJoinPool* instead of *jsr166y*'s bugged one.

For the simulation, an additional kind of *ParallelArray* has been implemented: *ParallelVector*, which allows parallel operations on vectors of real numbers. Within the scope of the simulation, the size of these vectors is generally limited to two, so using a complex structure for trying to take advantage of parallelism is unlikely to offer better computation performances, than just using a mere *for each* loop on a conventional structure. This should therefore be seen more as an experimentation. However, *ParallelArray* used for performing the computations needed for moving the robots, during each elementary time interval, are very likely to do a much more interesting job, especially when the swarm is large. A great quality of *ForkJoinPool* and *ParallelArray* is that they allow to perform computations with the highest possible degree of parallelism with the most suited number of threads [10].

2.5 Swarm simulation

2.5.1 Overview

The simulation is aimed at representing the behaviour of a swarm of robots managed using a certain process, in this case a distributed algorithm relying on potentials, as it was presented in the first part of this report.

The simulation software is quite simple. It simulates a set of robots, that are points that may move in an unbounded two-dimension-virtual-space. Every robot is registered within the virtual environment, that makes the link between all the robots, especially regarding the neighbours detection. So, for an elemental time interval [11], depending on what is detected in its neighbourhood [12], a robot will move (or not at all) in a certain direction at a certain velocity during the elemental time interval. Once a robot has moved, the view is updated. Figure 2.1 gives a snapshot example taken during a simulation.

[10] Multi-threading for achieving parallelism rely on two basic and antagonist principles: too few threads do not make it possible to give work to all the available cores, while too many threads slow the system down

[11] That is a fixed parameter of the simulation

[12] That is inside a circle of center the position of the robot, with a given and fixed radius corresponding to its visibility

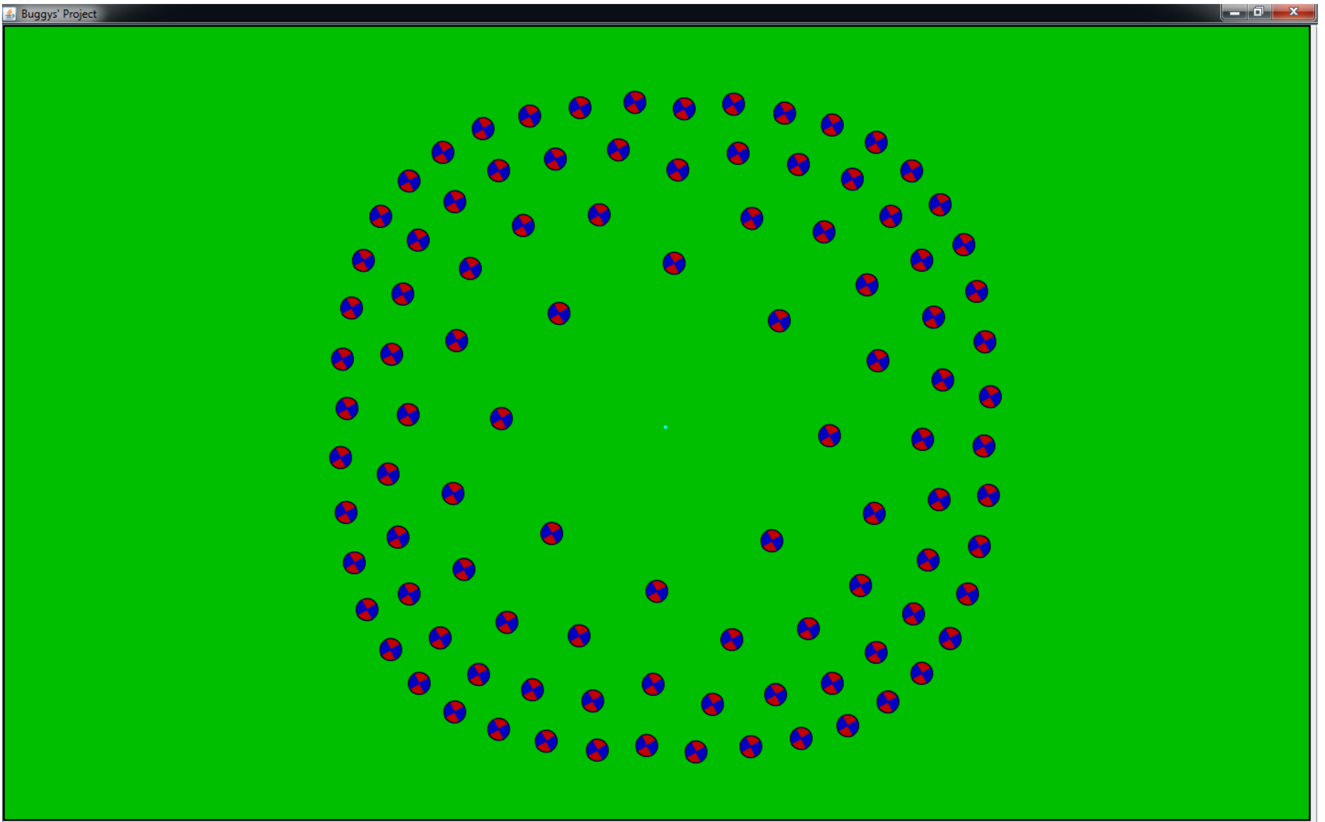

Figure 2.1: Overview of the simulation output
(above, the swarm encircles the point in cyan, at the center of the screen)

2.5.2 Parameters and Objectives

The simulation has several parameters that influence the swarm's behaviour:

- The way potentials are calculated
- The way robots determine their move from their potential
- The visibility range of the robots
- The existence of an external potential
- The swarm's size

Different configurations have been considered. At the beginning, one hundred robots [13] are located randomly on the field. In every case, the purpose was to obtain eventually a coherent swarm, that is a unique swarm, occupying a given shape, approximately constant over time. Within the swarm, robots may move, or not, this does not really matter. Experimentations have also been performed to control the swarm in a global way, for moving it or modify its shape.

[13] Other replicas have also been performed with two hundreds robots, but the animation was significantly slowed down

2.5.3 Choice of a potential function

Figure 1.5 (see page 10) shows the kind of potential functions that has been considered. It depends only on the distance between the robots, r, and has the following characteristics:

- $V(r) \underset{r \to 0}{\longrightarrow} +\infty$
- V decreases until it reaches a global minimum, in $a \in \mathbb{R}_+^*$
- V increases on $[a; +\infty[$
- $V(r) \underset{r \to +\infty}{\longrightarrow} +\infty$

As explained in the theoretical part, choosing such a potential has the following consequences, regarding the interaction force $\overrightarrow{F}$ exerted between two robots:

- $\overrightarrow{F}$ is the more repulsive, the lower r is
- $\overrightarrow{F}$ is the weaker, the closer r stands from a
- $\overrightarrow{F}$ is the more attractive, the higher r is

Actually, the robots having a limited perception of their neighbours, V becomes constant and therefore $\overrightarrow{F}$ becomes null, when r becomes greater than a certain distance. In the simulation, this limit is a fixed parameter, b, comprised between 0 and $+\infty$. To make the algorithm work properly, b shall obviously be greater than a ; $2 \cdot a$ should be a minimum.

Several functions have been tested. Among them, two are proposed in the simulation. Switching from one to the other may be performed by pressing on the keys "A"/"Q" and "Z"/"S" [14]. Setting down r the reduced distance between two robots ($r > 0$), which is equal to the real distance divided by the target distance between robots, these functions are the following:

- $V_1(r) = \frac{r^4}{4} - \ln(r)$, so $\overrightarrow{F_1(r)} = \left(\frac{1}{r} - r^3\right) \cdot \overrightarrow{e_r}$
- $V_2(r) = \frac{r^8}{8} - \ln(r)$, so $\overrightarrow{F_2(r)} = \left(\frac{1}{r} - r^7\right) \cdot \overrightarrow{e_r}$

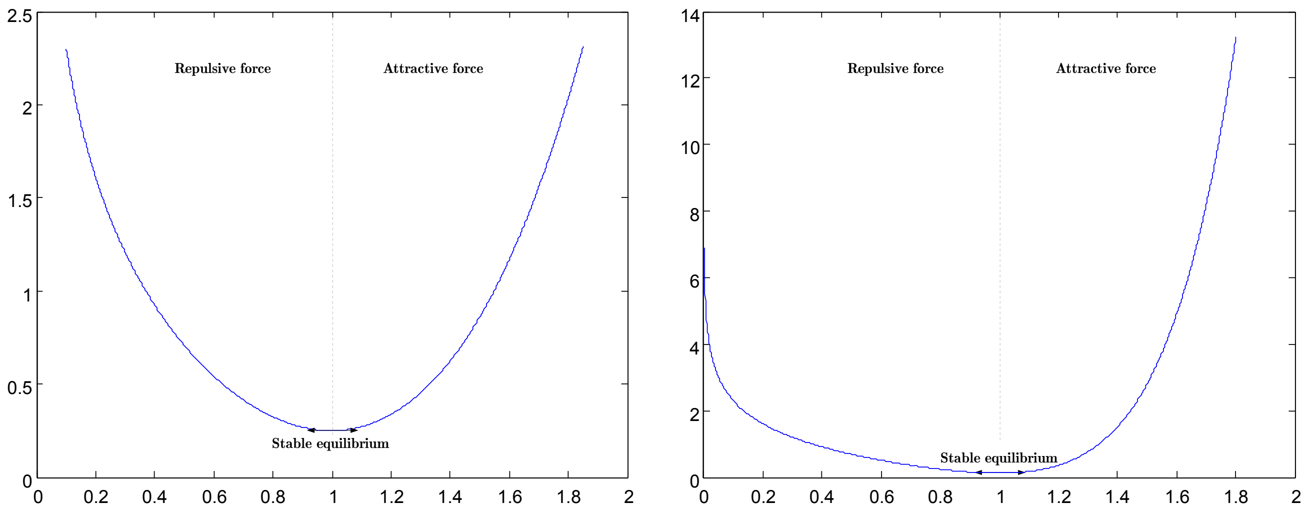

Figure 2.2: Available potentials (V_1 on the left, V_2 on the right)

[14] Projection on X et Y axes are independent for experimentation purposes

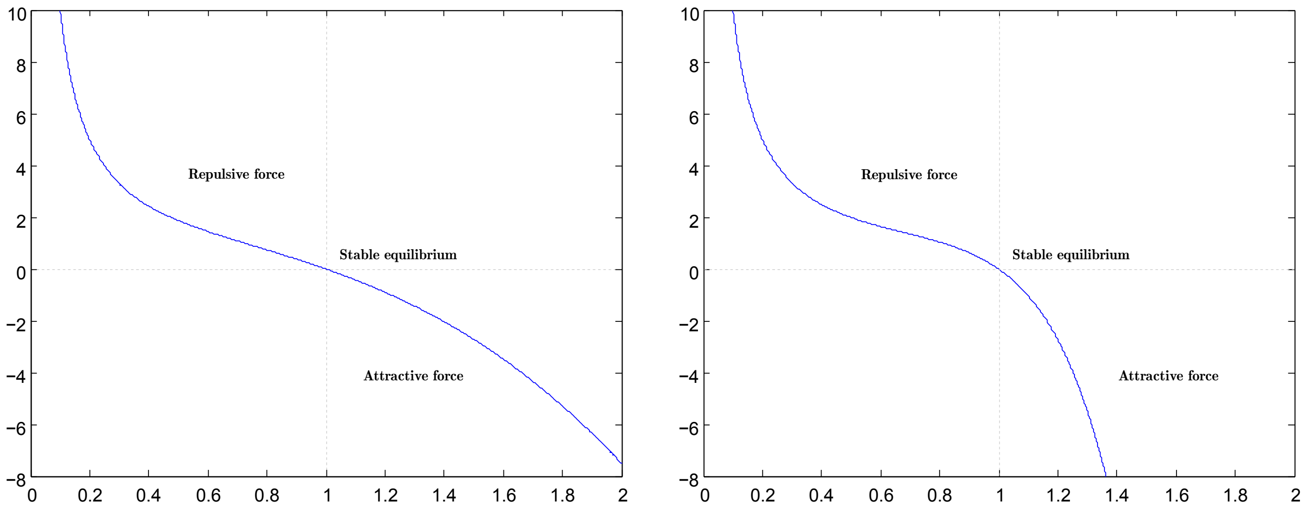

Figure 2.3: Available forces ($\overline{F_1}$ on the left, $\overline{F_2}$ on the right)

As illustrated by figure 2.2, the difference between these potentials is the attractive term, which is much stronger for the second potential, when r is great enough. In any swarm gathering simulation [15], two main phases may be observed: a transient state, and a steady-state (or a nearly steady-state).

The choice of a potential has a significant effect on the steady-state. In particular, regarding the two functions considered here, both potential functions result in a swarm with a circular shape. However, as shown by Figure 2.4, there is a difference in the robot distribution within the swarm. In the first case, it is rather uniform throughout the swarm area, while in the second case, robots desert the center of the zone to make up outlying concentric circles. This may be explained quite simply. Indeed, the attractive force (see figure 2.3) when $r > 1$ is greater for V_2 than for V_1 so, since the repulsive force is almost the same when $r < 1$, distant robots are subjected to more significant attractive interactions, while close robots are subjected to the same repulsive interactions. Hence, outlying robots attract more those located somewhere around the center of the swarm, without being able to move that much themselves because of their close neighbours. As a result, the robots position themselves closer from the swarm's periphery.

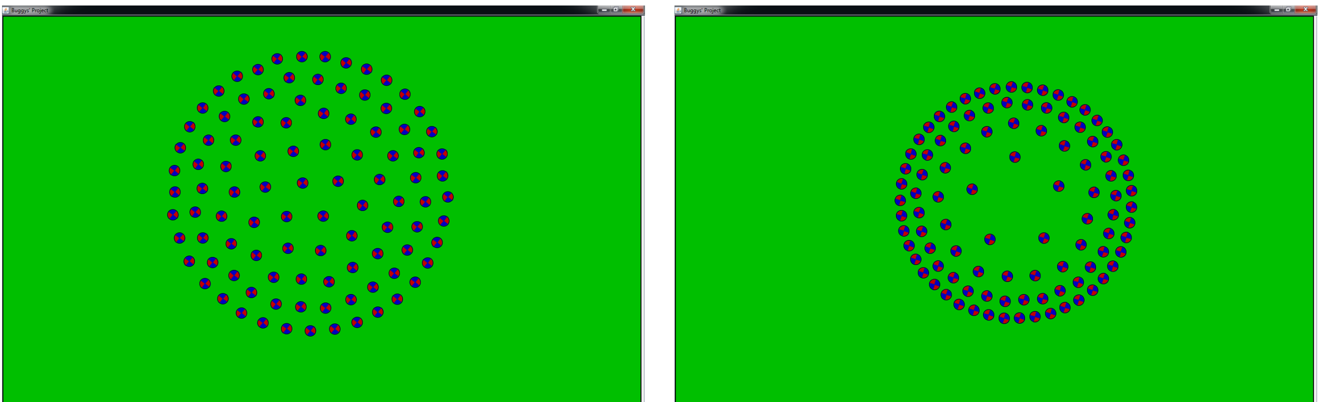

Figure 2.4: Steady-state according to the chosen potential (V_1 on the left, V_2 on the right)

[15]That is a basic simulation: robots are initially located randomly on the field and, due to the interactions they exert between each other, they move and eventually form a structured swarm (at least when the simulation successes)

2.5.4 Robots' state equation

In particle physics, the particles move in accordance with Newton's second law of motion: setting down $\overrightarrow{F}$ the force acting on the particle and $\overrightarrow{a}$ the particle's acceleration, $\overrightarrow{F} = m \cdot \overrightarrow{a}$.

Following this approach, it may be considered here that $\overrightarrow{F}$ corresponds to the interaction exerted between the robots, plus possibly an external force deriving from an external potential. Doing this way, the system hugely oscillates, without converging towards an equilibrium. This may be easily explained by the fact that, since $\overrightarrow{F}$ is a conservative force [16], the mechanical energy of the system is constant. So, given that the mechanical energy is equal to the potential energy plus the kinetic energy, and that the system tries to minimize its potential energy, as a result, the kinetic energy of the system will raise. Hence, the robots will move.

In order to avoid these annoying oscillations, a (non-conservative) friction force may be added to $\overrightarrow{F}$. With a friction strong enough, the robots do not oscillate at all any more. In addition, since the swarm's mechanical energy may only decrease over time, the swarm is guaranteed to reach an equilibrium. However, this process may be quite slow. Moreover, since the robots have an inertia, because of the mass m intervening in Newton's second law of motion, it may be observed that, with a not so strong enough friction force, even if the swarm do not oscillate at all, robots may get too close from each other due to inertia effects [17]. A compromise has therefore to be found regarding this friction's intensity: a weaker friction may make collisions possible, while a stronger friction slows down the swarm's evolution.

Well, even though using Newton's second law of motion for simulating the swarm dynamic gives interesting results, it is not that suited to determine the behaviour of robots. Indeed, robots are not commanded by acceleration, but rather by velocity, and have a bound velocity, which cannot be ensured by this method. So, it would be interesting to calculate the robots' motion with a greater control over the velocities.

Thus, a proportional command of the velocity of the robots according to their potential gradient, with saturation when it would be higher than their maximal speed, may be considered. In this approach, there is no inertia any more, and robots never collide with each other, unless the simulation is poorly parametrized, since they cannot approach too much from a neighbour without being repelled immediately. The absence of inertia and friction makes the swarm's reactivity extremely good, maybe too much. As a result, the transient state lasts a quite short period, and the steady-state comes quickly, but may oscillate in a perceptible yet not embarrassing way. With this new process, the notion of mechanical energy has no meaning any more, and there is therefore no theoretical guarantee that the system will reach a stable equilibrium. Actually, oscillation phenomena seem to occur rather with V_2 than V_1, rather when the robots' detection radius is limited, and rather when the swarm is dense. This situation is finally quite the same as it was when Newton's second law was used without frictions strong enough. In both cases, the order of magnitude of the simulation parameters shall be consistent, especially the ability of the robots to move (velocity and reactivity [18]), that shall remain reasonable in comparison with the desired dimensions of the swarm.

[16] Because $\overrightarrow{F}$ derives from a potential energy

[17] The inertia of a robot makes it able to move up against a strong repulsive force for a short time, that may be enough to cause a collision with another robot

[18] The inertia and the frictions for the approach by acceleration (Newton's second law of motion), the maximal velocity and the proportional command constant for the approach by velocity

2.5.5 Global Swarm Management

On the fringes of swarm cohesion simulations, experimentations have been led to control the swarm in a global way, especially regarding its position and its shape. This has been performed by introducing an external potential, to whom every robot is subjected.

Let us consider a position $M_0(x_0, y_0)$ we want to be the central point of the swarm. If we take an external potential of the form [19] $V_{ext}(r) = \frac{r^4}{4}$, where $r = d(M, M_0) = \sqrt{(x - x_0)^2 + (y - y_0)^2}$, then every robot located in a point $M(x, y)$ will be subjected to a force $\overrightarrow{F_{ext}}(M) = -r^3 \cdot \overrightarrow{e_r} = -r^3 \cdot \frac{\overrightarrow{M_0M}}{\|\overrightarrow{M_0M}\|}$, that will attract the robots towards M_0. So, if the location of M_0 is changed, for instance by moving M_0 according to the currently pressed arrow keys, the entire swarm will be displaced. This feature works well in the simulation software. Binding the swarm to adopt a certain non-circular shape has also been experimented by using non-spherical potentials, but the results were not that convincing. Future work might consider this point.

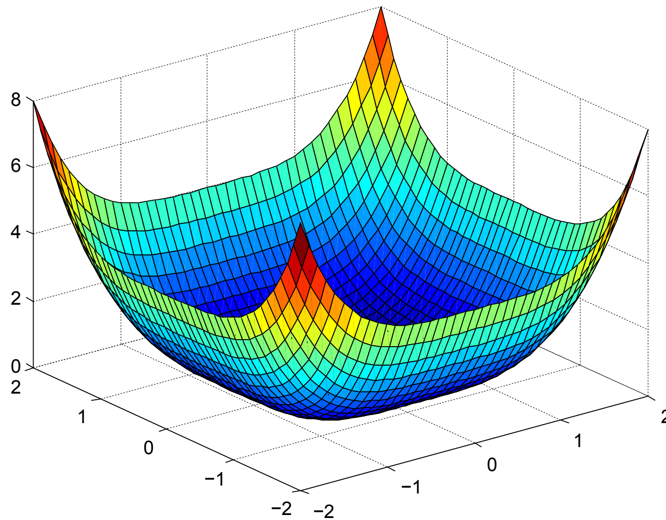

Figure 2.5: An example of external potential

2.6 Perspectives

2.6.1 Software structure

Since it has been developed throughout the project, following all the different approaches that have been considered, the source code comprises two different parts that are disconnected from each other:

- An embryo of an implementation related to the beginning of the project (decentralization of *Buggies Game* behavioural algorithm), launched *via main.MainV1*, and that uses the packages *graphics* (graphical interface), *network* (socket-related utilities), *communication* (Kademlia servant), and *partitioning* (polygon mesh of the field meant to avoid collisions between robots)
- The implementation of the potentials method, launched *via main.MainV2*, and that uses mainly the packages *graphics* (graphical interface), *entry* (keyboard inputs), *sensors* (neighbours localization), *simulation* (virtual objects and environment), *utils* (utility classes), and *extra166z* (copy of the official package *extra166y*, modified to use JDK 7's *ForkJoinPool* instead that of *jsr166y*)

Future work might include a restructuring of the code. If closures and *ParallelArray* are finally added to *Java 8* standard library [20], a significant cleaning of the code could also be performed.

[19] See figure 2.5 for a representation

[20] Both were originally planned to be introduced in *Java 7*

2.6.2 Realism

The way robots are modelled is extremely simple. In practise, even without speaking of acceleration and velocity, robots cannot change completely their course easily, especially when it requires a right-angled bend or a turn-back [21]. Future work might therefore consider more realistic state equations for the robots.

In addition, some errors could also be considered at the sensors level, especially the neighbours detection. This might deal with inaccuracy as well as non-detection phenomena, for instance when the sensor fails.

2.6.3 Swarm management

As shown by various figures in this report, a stabilized swarm may only take a circular shape. It is true that playing with the inter-robots potential makes it possible to change the internal structure of the swarm. However, its global shape remains circular. In some cases, it might be useful to give an other shape to the swarm. For instance, if the robots have to surround a player, like in *Buggies Game*, then a curved line seems more appropriate.

Two axes of refinement may be considered, at different levels:

- Micromanagement: the way robots interact with each other; in particular, making the potential dependent not only on the distance between robots, but also on the angle between them [22]
- Macromanagement: the way the swarm is globally managed; a process relying on an external potential has been successfully tested for basic purposes, and may be refined, or other approaches may be followed

2.6.4 Distributed intelligence?

Originally, the purpose of this project was to reproduce *Buggies Game*'s swarm intelligence. Well, at least regarding the potentials method, this is a point that has not been really handled. Yet it might not be that difficult, this problem seems quite tricky to solve, since it would require the swarms' objectives to be put under the form of a more or less complex external potential.

In addition, to ensure that the entire swarm pursues the same goal [23], and therefore uses the same external potential, a solution relying on a consensus protocol might be considered. And if the objective may change over time, a replicated state machine might deal with periodically determining the corresponding potential function, and spread it to every robot. This would make a link between the two distributed approaches presented in this project.

[21] Unless the robot has the ability to move forwards and backwards without any difference, which is the case for the *buggies* used in *Buggies Game*

[22] This notion of "angle" is voluntarily vague and needs to be clarified

[23] And to prevent the swarm for having a globally inconsistent behaviour

Conclusion

This industrial project was meant to develop a parallel and distributed behavioural algorithm that may be used for managing a swarm of robots. To reach this objective, a first approach was led from an existing project, *Buggies Game*, which proposes such a swarm management but performed in a centralized way. To distribute this swarm management and make it more robust against individual failures, the idea was simple: to use all the robots composing the swarm in order to parallelize and replicate the computation, which would make it more effective and more robust. However, this method kept one of the drawbacks of the centralized approach: the gathering of some data concerning all the robots. Hence, beyond a certain threshold, the number of robots would not have made the process more effective or more robust, but would have rather been likely to overload it.

While this first approach was running out of steam, a second has been considered. Inspired by particle physics, the principle was to manage the swarm of robots in the same way atoms permanently manage themselves, all around us. From this point forward, the simulation has been focused around a simple problem: being initially located in random places, the robots have to gather and form a structured swarm. With judiciously chosen parameters, the results are rather convincing. An arbitrary control has even been performed on the swarm, making it move as wished.

These two approaches are very different, especially regarding the way intelligence is implemented. Swarm intelligence is quite easy to design in the first case, since the behavioural algorithm has a global view of the system. The robots are therefore mere pawns, mere limbs of a common body. In the second case, however, the situation is utterly different, since each robot decides on its own of its own actions according to its close neighbourhood. The robots are not limbs of a same body any more, but individuals setting up a society. Intelligence is harder to design. The purpose is not to create a mere collective behaviour any more, but a unique individual behaviour that, in interaction with itself (*i.e.* with other robots), becomes a consistent collective behaviour.