Spacial Debris Collector Robot

An AI-based and autonomous debris collector satellite simulation

David Fernàndez López
Grau d'Enginyeria Informàtica
Treball Final de Grau
David Isern Alarcón
June 12, 2013

Outside intelligences, exploring the Solar System with true impartiality, would be quite likely to enter the Sun in their records thus: Star X, spectral class G0, 4 planets plus debris.

Isaac Asimov

Document built with LaTeX

Abstract

Space Debris is a growing threat to both manned and unmanned missions in space. The growing number of out-of-order satellites, used rocket boosters and other debris makes the calculation of safe orbital paths harder and harder, until a day in which the density of debris will render orbital space unusable. To avoid this situation a debris removal system is required. In this paper the core functions of an artificial intelligence (AI) for controlling a debris collector robot are designed and implemented. Using the robot operating system (ROS) as the base of this work a multi-agent system is built with abilities for task planning using a genetic algorithm, self-location through the use of particle filters and path-planning and collision avoidance through the use of a modified elastic force algorithm. This MAS contains a simulated world as well, to test its performance.

Contents

Chapter 1

Introduction

Since humanity started sending unmanned probes to space thousands of satellites have been put in orbit. Every year the number of satellites put in orbit grew, and that, together with the fact that usually satellites are put in their orbit and left there without any further thought means that space around earth has become widely populated.
Currently there are more than 19.000 pieces of debris larger than 5 cm tracked, with more than 300.000 pieces smaller than 1 cm, all of them below 20.000 km of altitude [1].

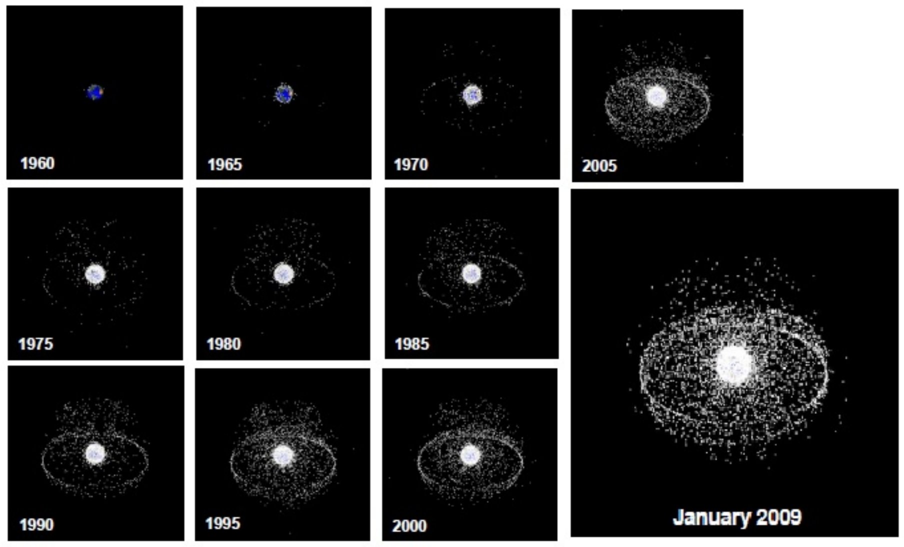


Figure 1.1: Tracked space object population growth until January 2009 [1].

This Space Debris is formed by thousands of different objects, from out-of-order satellites and rocket boosters to flecks of paint and debris created by previous collisions, but all of them share the same potential for trouble. This debris presents a huge danger for both manned and unmanned craft, and is making the calculations of safe orbital paths and exploration missions harder and harder to the point where they may be almost impossible.
The main reason why space debris must be taken care of is what is known as the Kessler syndrome [4], a scenario in which the density of space debris is high enough to create a collision cascade, each collision generating more debris and thus increasing the likelihood of new collisions, those generating new debris in turn. If this scenario was reached a single collision could create a cascading effect that would render every object in that orbital range damaged and useless.
Several solutions to this problem have been proposed, like the MDA Space Infrastructure Servicing vehicle [2], which includes the capability to move satellites to a “graveyard orbit” or the use of an electrodynamic tether [3] attached to a satellite.

1.1 Proposed Solution

The solution proposed in this paper consists in an autonomous spacecraft, able to pick up satellites and drag them to a designed "safe point" for later disposal, either by shipping them back to Earth or by recycling them in a suitable space facility.
This paper will focus on the Artificial Intelligence (AI) which will be the brain of the aforementioned robot.

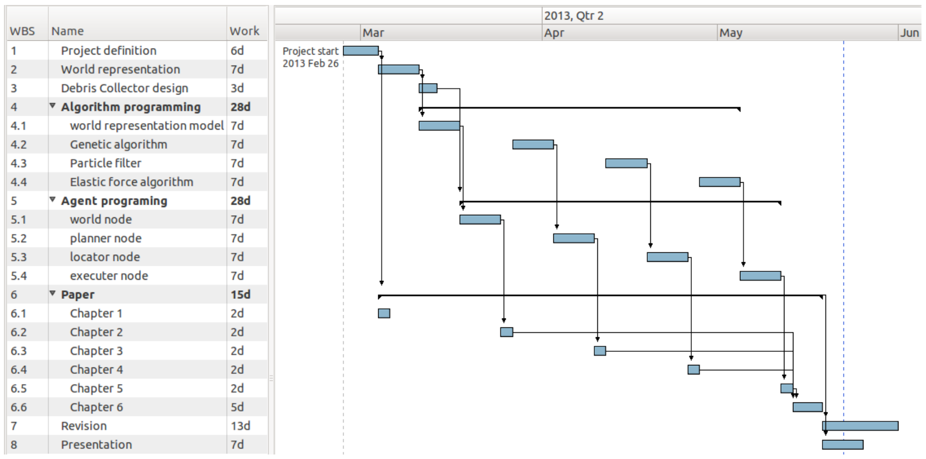
1.2 Goals

This project aims to design and program the AI that will control the Debris Collector .
It is not in this project's scope to provide blueprints, nor schematics, nor even hardware configurations beyond the requirements for the project to work.

- Definition of the world model.
  - Define the boundaries of the world model.
  - Define the main abilities the Debris Collector will have.
- Program the world model.
- Identify and define scenarios.
  - Locate the robot in space.
  - Locate the objects around the robot in the same model.
  - Calculate the best action plan to capture all objects.
- Program each defined scenario.
- Model the calculated plans as task sequences.

1.3 Work plan

The project has been split in the following tasks:



WBS	Name	Work
1	Project definition	6d
2	World representation	7d
3	Debris Collector design	3d
4	▾ **Algorithm programming**	**28d**
4.1	world representation model	7d
4.2	Genetic algorithm	7d
4.3	Particle filter	7d
4.4	Elastic force algorithm	7d
5	▾ **Agent programing**	**28d**
5.1	world node	7d
5.2	planner node	7d
5.3	locator node	7d
5.4	executer node	7d
6	▾ **Paper**	**15d**
6.1	Chapter 1	2d
6.2	Chapter 2	2d
6.3	Chapter 3	2d
6.4	Chapter 4	2d
6.5	Chapter 5	2d
6.6	Chapter 6	5d
7	Revision	13d
8	Presentation	7d

Figure 1.2: The work plan for the project.

There are eight tasks:

1 Project Definition Defining the project to be presented, its scope and goals.

2 World representation Designing and defining how will the world be represented for the Debris Collector .

3 Debris Collector design Designing how will the Debris Collector be implemented.

4 Algorithm programming Programming the various algorithms required by the Debris Collector and testing them.

4.1 World representation model Programming the model defined in task 2.

4.2 Genetic algorithm Programming and testing the genetic algorithm required.

4.3 Particle filter Programming and testing the particle filter algorithm required.

4.4 Elastic force algorithm Programming and testing the elastic force algorithm for collision avoidance.

5 Agent programming Programming the intelligent agents required by the Debris Collector .

5.1 world node Programming the world node intelligent agent.

5.2 planner node Programming the planner node intelligent agent.

5.3 locator node Programming the locator node intelligent agent.

5.4 executer node Programming the executer node intelligent agent.

6 Paper Writing the paper.

- **6.1 Chapter 1** Writing the "Introduction" chapter.
- **6.2 Chapter 2** Writing the "The World" chapter.
- **6.3 Chapter 3** Writing the "The task planner node" chapter.
- **6.4 Chapter 4** Writing the "The locator node" chapter.
- **6.5 Chapter 5** Writing the "The task executer node" chapter.
- **6.6 Chapter 6** Writing the "Conclusions" chapter.

7 Revision Revision of the paper.

8 Presentation Writing and recording the presentation of the project.

1.4 Deliverables

The resulting program will be delivered as a zip file containing all the required code and files to compile and execute the Debris Collector . The zip contains the following files:

- debris_collector - This folder contains the ROS package with the source code ready to be compiled and executed.
- test_log - This folder contains the log of a simple test exeution of the code.

1.5 Structure of the document

The following chapter gives a more detailed explanation on how the Debris Collector's AI works and its architecture, as well as how it understands the world.

1.5.1 Chapter 2: The World

This chapter explains how the world surrounding the Debris Collector will be represented, discusses the chosen architecture and explains the world node, the node responsible for the simulation of the world.

1.5.2 Chapter 3: The task planner node

This chapter explains how will the Debris Collector plan its work, discusses the use of genetic algorithms and explains the planner node, the node responsible for planning the tasks.

1.5.3 Chapter 4: The locator node

This chapter explains how the Debris Collector will localize itself, discusses the use of particle filters and explains the locator node, the node responsible for localizing the Debris Collector in the world.

1.5.4 Chapter 5: The task executer node

This chapter explains how the Debris Collector will carry out the tasks planned before, discusses the use of elastic force algorithms to avoid collisions, and explains the executer node, the node responsible for carrying out the planned tasks.

1.5.5 Chapter 6: Conclusions

This chapter explains the complete agent network built through this paper, explains the tests performed, and suggests further improvements that could be made on the Debris Collector .

Chapter 2

The World

This chapter discusses the Debris Collector in greater detail. First I will define how it understands its surrounding world, to proceed then to examine its mind, the AI's software architecture that will be implemented in following chapters.

2.1 Modeling the world

To be able to think about the world we need to understand the world, and something similar goes on with robots. For them to be able to extract conclusions and make decisions about the surrounding world they need a suitable model to represent it.
For the Debris Collector I will build a geocentric model, meaning that every object (including itself) will be expressed in terms of their relative position to the Earth (in fact to a fixed point in Earth, namely it's mass center). This model can be extrapolated to whichever planetary body the Debris Collector should orbit.
To represent every object's position a spherical coordinate system will be used. For this coordinate system to work we need to define some important concepts first:

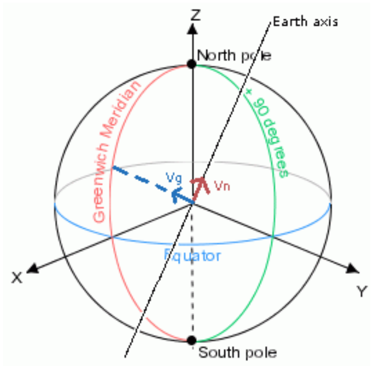


Figure 2.1: V_N (red) and V_G (blue) in relation with the world sphere.

- The Earth's center of mass, called c.
- The polar vector, a unitary vector withs its origin in c pointing towards the north pole, following the Earths rotation axis direction, called V_N .
- The Greenwich vector, a unitary vector with its origin in c pointing towards the Greenwich meridian and perpendicular to V_N , called V_G . There are two points in Earth which satisfy this conditions, it is not relevant which one is picked as long as the system is coherent.

It is important to take notice that this model does not consider Earth as a moving object, but rather as a stationary one. That model is good enough for the Debris Collector , however, as it will be orbiting Earth and not traveling through the solar system.

Once those concepts have been specified I can define every single object's position orbiting Earth as the triad (r, θ, φ), where:

r is the distance form the object center of mass to c in meters.

θ ,also called the polar angle, is the angle between V_N and the projection of the distance vector over the plane containing V_N and perpendicular to V_G .

φ , also called the azimuth angle, is the angle between V_G and the projection of the distance vector over the plane containing V_G and perpendicular to V_N .

2.1.1 Targets and Obstacles

Besides modeling the world and the Debris Collector's position in it I need to model every other object moving around it. Every single object around the Debris Collector will be classified as either a Target or an Obstacle. Targets are those objects the Debris Collector has to pick up, and thus it's position and trajectories must be tracked. Obstacles, on the other hand, are those objects that are not targets, and must be avoided by the Debris Collector .
Both objects are very similar in terms of position modeling but, besides its (r, θ, φ) triad we need to know how are they moving, as most objects in space are not stationary. To model the trajectory we will use another triad (v, α, β), where:

v is the radial speed, it defines how r changes through time in meters per second.

α is the polar angular speed, it defines how θ changes through time in degrees per second.

β is the azimuthal angular speed, it defines how φ changes through time in degrees per second.

Thus the orbit's equation is:

$$(r_1, \theta_1, \varphi_1) = (r_0 + vt, \theta_0 + \alpha t, \varphi_0 + \beta t) \tag{2.1}$$

Where $(r_0, \theta_0, \varphi_0)$ is the object's original position, t is the elapsed time, (v, α, β) is the orbital speed as defined above and $(r_1, \theta_1, \varphi_1)$ is the objects position after the elapsed time.

2.1.2 Debris Collector's movement

Physics in orbit, thought using the same principles that physics on Earth, have some important differences. Usually movement on Earth is obtained from friction, like walking or cars do. Instead, in space, movement is obtained from applying Newton's second law of movement, "*the acceleration of a body is inversely proportional to its mass, parallel and proportional to the net force acting on it and in the same direction that this force*" [5].

Usually this translate in the use of thrusters, devices that generate a force on the body they are attached to, usually by the expulsion of matter (rockets use the expulsion of a high-speed fluid exhaust [6], while an ion thruster uses accelerated ions [7], though the base principle remains the same).
For the purposes of this paper it matters not which kind of thruster will be used by the Debris Collector as they operate in a similar fashion as long as movement is concerned. During the rest of this paper a thruster, or more than one thruster acting simultaneously, will be modeled as a net force applied on the Debris Collector during a certain amount of time, thus generating an acceleration of the Debris Collector in the direction of the force.
Besides how the movement is generated there are other forces that must be considered, as they have an effect on the Debris Collector . Earth's gravity, though having a lower pull because of the Debris Collector's altitude, is still an important force that should be considered as it will pull the Debris Collector towards the Earth, degrading its orbit. Other forces, such as the gravity well, or the push of solar winds should be considered as well, as they can modify the Debris Collector's orbit and trajectory.

2.2 The Debris Collector's Software

To develop the Debris Collector's AI I need to decide both the operating platform on which this software will run and the goals this software must accomplish. In the previous chapter I talked about its main goal, collecting space debris and dragging it to a predefined point. In the following sections, however, I will define how this general goal should be accomplished and how am I going to make it possible.

2.2.1 Debris Collector's behavior

The Debris Collector should be fully autonomous, needing no human operators to work. In order to provide the Debris Collector of those capabilities I need to design its behavior by defining a finite-state machine [8] which will guide it.

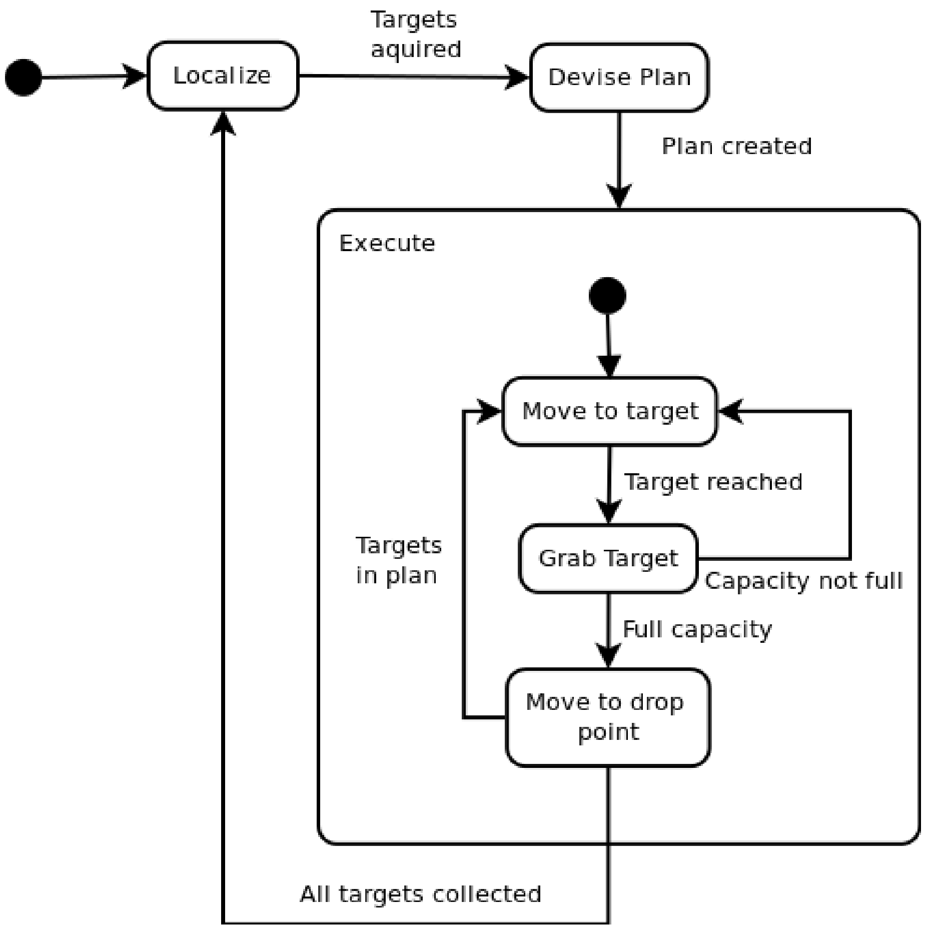


Figure 2.2: The UML state diagram defining the finite-state machine for the Debris Collector .

As shown the Debris Collector will start in its "Localize" state, which is also its general state when nothing else has to be done. In this state the Debris Collector will be localizing itself and everything around it within the world. When the Debris Collector finds a Target it will then move to the "Devise Plan" state, where it will try to devise the best path to catch and retrieve all Targets. Once a plan is devised the Debris Collector will move to the "Execute" superstate, in which most of the heavy work (at least physically) will be done. Once in the "Execute" superstate the Debris Collector will start by moving to the first substate "Move to target" to move to a Target, once there it will move to the "Grab Target" substate and catch it. If it can carry no more Targets, or there are no more Targets to carry the Debris Collector will then move to the "Move to drop point" substate and drag the Targets to the predefined drop point. Otherwise it will, once again, move to the "Move to Target" substate and start over. Once it has released its cargo in the drop point, if there are more Targets in the plan it will move to the "Move to Target" and start over otherwise the robot will move to the "Localize" state until more targets are found. As can be easily noted there is no ending state, as the Debris Collector will be always working unless a forceful shutdown must be done.

2.2.2 Robot Operating System

The Robot Operating System (ROS) is an open source project to provide "libraries and tools to help software developers create robot applications" [9], released under a BSD license. It provides an abstraction layer that allows the programmer to focus on the actual AI involved instead of hardware driver management, message passing and such.
The ROS architecture is designed with a distributed approach, with a series of Nodes interconnected. A Node is a small C++ program doing a specific task. Each Node can then *publish* or *subscribe* to a Topic (in fact to any number of Topics), sending information to it or receiving the information contained in the Topic respectively.
Additionally a Node can offer *Services*, which are functions that can be called by any other Node.
Both Topics and Services will be used in this project. The fact that ROS handles all the complexity of sending and receiving messages makes it a powerful tool, as the work can focus on the actual intelligence programmed without losing time and effort on the underlying architecture.

2.2.3 The Multi-Agent System approach

A multi-agent system [10] consists in a group of intelligent agents working together towards a common goal. Each intelligent agent [11] is an autonomous program able to interact with and take decisions about its environment. By making those agents able to interact with each other and communicate between them a much more complex and powerful behavior emerges. Thanks to the fact that the work is divided between multiple agents, each of them fine-tuned to solve a specific simple problem, complex goals can be easily achieved.
If we take a look at ROS's architecture it's trivial to see that this is the preferred approach to work within it, creating each agent as a ROS Node and communicating them through Topics or Services.
Each node is autonomous, as it works by itself, yet they need to communicate to send their results to the other nodes, i.e. the executer node needs to know where the Debris Collector is to decide where to go, information that is computed by the locator node.

2.2.4 Debris Collector's node map

As said in previous sections the Debris Collector will use a multi-agent system architecture, developed using ROS. For this we need to decide which tasks will be done by which agents and how they interact with each other.

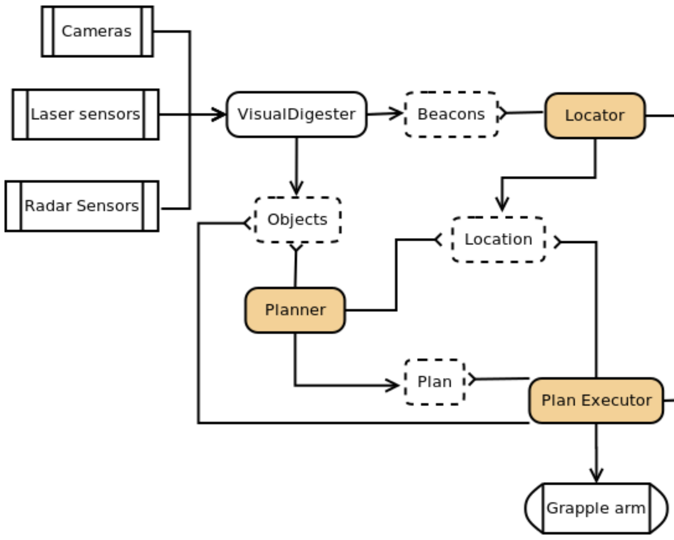


Figure 2.3: The agent map shows how the different agents interact between them. Round-cornered rectangles represent the Nodes, round-cornered rectangles with dotted borders represent the Topics. The Nodes discussed and implemented in this paper are highlighted.

As can be seen there are five Nodes, each of them an intelligent agent with a single purpose. Those Nodes communicate to Topics and read from them. Some Nodes also read or write from sensors or to actuators respectively.
The five Nodes are:

VisualDigester: This Node will not be implemented, yet its existence is key for the other Nodes. This Node receives the data from different sensors, either cameras, laser range sensors, radars or other sensors and transforms it in positions according to the Debris Collector's world model. On one hand the different objects around the Debris Collector have to be located in the world and put into the "Objects" Topic, on the other hand the information from the different *localizer beacons* have to be localized relatively to the Debris Collector and put into the "Beacons" Topic.

Localizer: This Node reads the *localizer beacons* relative positions from the "Beacons" Topic and uses those to deduct the Debris Collector's actual position inside the world model we are using, putting this information into the "Location" Topic.

Planner: This Node reads the information from the "Objects" and "Location" Topics and uses it to decide which will be the best route to pick all targets, publishing this route in the "Plan" Topic.

Plan Executor: This Node reads the information from the “Plan” Topic and executes the plan using the arms and other actuators. It reads from the “Location” Topic to know where he is and from the “Objects” Topic to know its surroundings. It published the required movements to make in the “Move” Topic.

Pilot: This Node will not be implemented, yet its existence is key for the other Nodes. This Node recieves the required movement through the “Move” Topic and calculates which thrusters should be used and for how long.

2.3 World node documentation

The World Node will be the node responsible for modeling the world, maintaining an absolute position of each and every object through time, checking for collisions and giving the object’s positions in the geocentric model defined before. It will also be responsible of simulating those Nodes not implemented in this paper.

2.3.1 Parametrization

The simulated world, and the objects it contains, can by parametrized by modifying the following parameters:

targets and obstacles targets and obstacles are two vectors that contain a series of SpaceObjects (TargetSO and ObstacleSO respectively) representing the objects present in the simulated world.

SENSOR_RANGE specifies the maximum sensor range of the simulated robot. Objects that are further away than the sensor range will not be “detected” (published to the Objects Topic).

GRAB_DISTANCE specifies the maximum distance allowed between the robot and the “Target” to be grabbed.

2.3.2 Topics

This node reads information from the following topic:

Move The topic in which every movement the Debris Collector has to make is published by the executer node.

It also publishes information in the following topics:

Objects Every space object surrounding the Debris Collector is published in this topic, as a Spherical coordinates position and angular speeds.

Beacon The three closer beacons to the Debris Collector are published here, as its position and distance to it.

2.3.3 Services

The World Node provides the “Grab” service, which receives the id of a “Target” and tries to grab it. This service returns a 0 if it has been successful and a 1 otherwise.

Chapter 3

The task planner node

In this chapter the “Planner” node which will be responsible for devising the best way to pick up the targets will be built.

3.1 Problem definition

In any given moment the Debris Collector will have a set of targets around it. The goal of this intelligent agent is to find a way to meet each target while keeping fuel consumption to the minimum possible.

3.1.1 The Traveling Salesman Problem

This problem is quite similar to the “Traveling Salesman” problem [12]. In this problem a traveling salesman has to visit a number of cities. The goal is to calculate the cheapest route between cities which visits every city exactly once. If “cities” are substituted by “Targets”, the salesman by the Debris Collector and “smaller cost” by “less fuel consumption” the problems look almost the same, even if the “Targets” are moving and have to be brought to a predefined drop point.

3.1.2 Genetic Algorithms

Though the “Traveling Salesman” problem may be easily solved with an $A*$ algorithm the time needed by the algorithm increases exponentially with the number of “cities”. To avoid this situation a number of algorithms have been developed, one of them being the Genetic Algorithm [13]. The genetic algorithm imitates the evolution of a population towards the “best fit” for its environment. The genetic algorithm is as follows:

- First a “Solution Population” is created, for example, in the “Traveling Salesman” problem every individual is a possible order to visit the cities.
- Then a “Fitness” is calculated for each individual using an heuristic, for example, in the “Traveling Salesman” problem the heuristic is the total cost of the proposed route.
- After that a new “Solution Population” is created by randomly picking up individual pairs according to their “Fitness” (smaller costs will reproduce more often) an combining those two solutions to create a new one.
- On some randomly chosen individuals a mutation is applied, for example, in the “Traveling Salesman” problem two cities could be swapped in the order.
- When convergence is found, the algorithm stops. Convergence can be defined either as “the best cost hasn’t changed for a number of iterations” or as “difference between population individuals is smaller than a certain threshold”. If convergence has not yet been found the algorithm returns to step 2.

3.2 Solution

To solve this problem a genetic algorithm will be used. In order to explain more clearly the proposed solution an iterative description will be used. First the easiest problem (unmoving “Targets”) will be solved and then the additional complications of the problem will be added layer after layer.

3.2.1 Stationary satellites

The easiest scenario is when the “Targets” that have to be picked up by the Debris Collector are all “stationary” (floating over a specific earth coordinate, and thus immobile in the defined reference frame). In this scenario the problem to solve is identical to the classic “Traveling Salesman” problem. When building a genetic algorithm the population, heuristic, reproduction algorithm, mutation algorithm and convergence have to be defined.

Population: The population for this scenario is formed by a set of individuals, each one of them a possible order in which to pick up the “Targets”. In those individuals there could be no duplicated values, as each “Target” can only be picked up once. Each order-“Target” pair is called a chromosome. In the built algorithm the size od the population is equal to $n * (n - 1)$, with n being the number of “Targets” to pick up.

Heuristic: In this scenario the simplest heuristic is to calculate the sum of distances between consecutive chromosomes. The fitness of an individual is the inverse of its cost, normalized.

Reproduction: As no duplications can be found in any individual it is not possible to just mix both answers. To solve the situation a greedy approach has been used. When two individuals are chosen to reproduce the cost of moving from the current position to the first chromosome is compared, and the chromosome with smaller cost is taken. For every next chromosome the cost to the following chromosome in its parent is compared and the one with smaller cost is added to the child. This algorithm is repeated until a new population with the same size than the initial population is created.

Mutation: When a mutation is applied on an individual the pickup order is modified by swapping two random “Targets” in the order.

Convergence: Convergence is found when either the total lower cost minus the total lower cost of the previous iteration is lower than a certain threshold for a number of consecutive iterations or a predefined number of iterations have occurred.

3.2.2 Moving satellites

If “Targets” are moving then the complexity increases, as it’s not enough to calculate the best order, but when to pick each “Target” is as important because the cost can change greatly as the distance increases or decreases with time.

Population: To build the population for this scenario the pickup times for each target should be added. Now a chromosome is the triplet formed by a “Target”, its order and its pick up time. Obviously pickup times have to form a strictly increasing sequence.

Heuristic: To adapt the cost calculation the required speed is used to reach the next chromosome at the desired time, as higher speeds mean higher accelerations and thus higher fuel costs. The fitness function needs no modifications.

Convergence: Convergence is not modified.

Reproduction and Mutation are the most heavily modified parts in this scenario. Though the general algorithm for reproduction is not modified there are several possibilities to consider as time is concerned.
Two approaches have been tested:

Brute force: Time is randomly selected when the population is generated and, during reproduction the time increment for the chosen next chromosome is picked. Time is also mutated, selecting a random time-chromosome in the individual and changing its value randomly.

Minimum distance: Time is considered a consequence of order, and it is always chosen as the time in which the distance between the current point and the next “Target” is minimum. To find the minimum distance a slightly modified gradient descent algorithm is used.

To compare the performance of both algorithms they have been implemented and some simulations where performed with the same parameters, ordering 5, 10 and 15 “Targets” with 100 simulations per configuration. Medium total time and iterations required by the algorithm, medium cost, solution dispersion (how many different solutions have the algorithm returned) and medium total time the solution plan will take to execute have been compared to decide which one is the best solution.

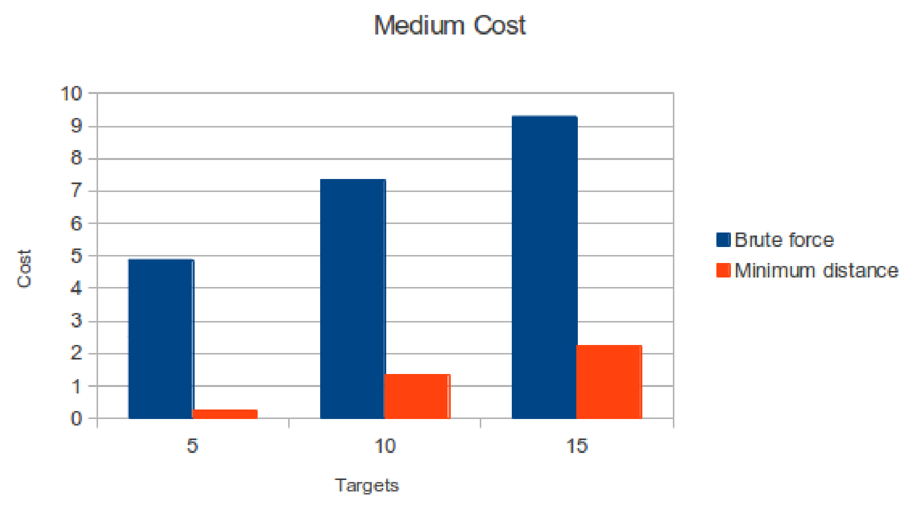


Figure 3.1: Medium cost

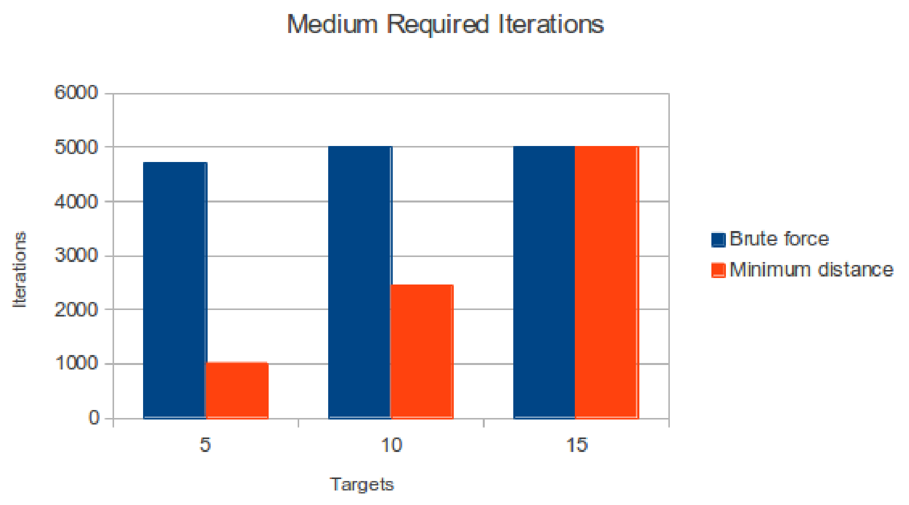


Figure 3.2: Medium iterations

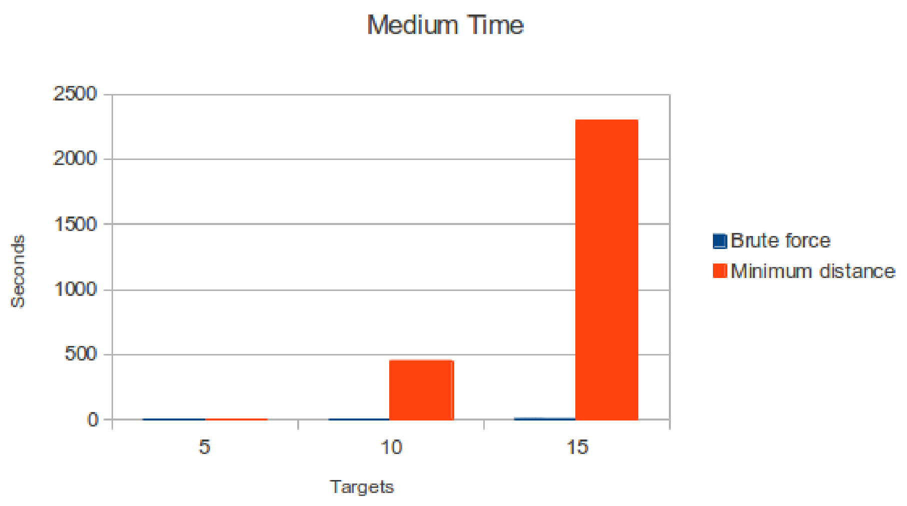


Figure 3.3: Medium time

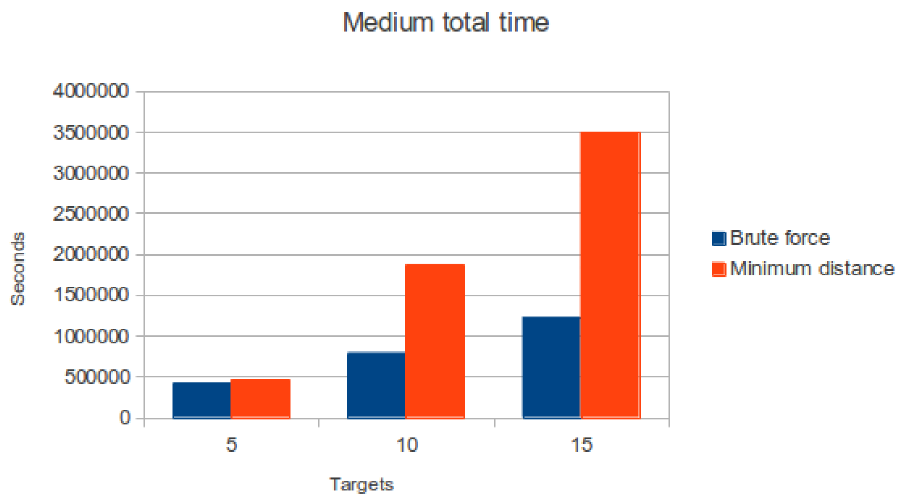


Figure 3.4: Medium execution time

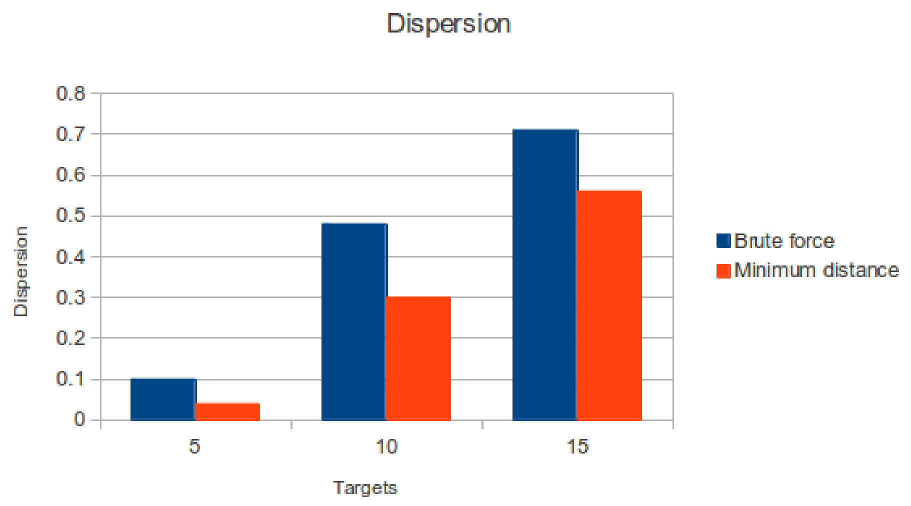


Figure 3.5: Result dispersion

As can be seen in Figures 3.1 to 3.5 the Brute force algorithm is much faster and the resulting plan takes less time to complete, but is much more dispersed and has higher costs, which are direct consequence of being a shorter plan. On the other hand the minimum distance algorithm is far slower to execute, but takes less iterations and is more cohesive. Though the results from the minimum distance algorithm are far better in regards of cost and dispersion its to slow to converge, rendering it unusable.
To try to get both algorithm's strong points a mixed solution is required. The mixed solution consists in using the minimum distance algorithm to generate the initial population. Then, during algorithm execution, the brute force approach to reproduction is used, while time mutations are introduced in the mutation process. Time mutations consist in calculating the minimum distance time for every chromosome in the individual. The results of the same test over this algorithm are better than any of its parents, as the following figures show.

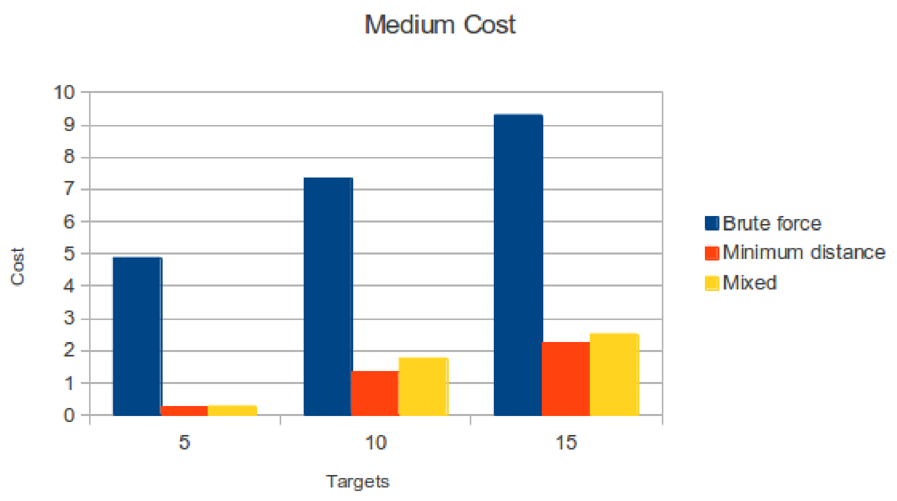


Figure 3.6: Medium cost

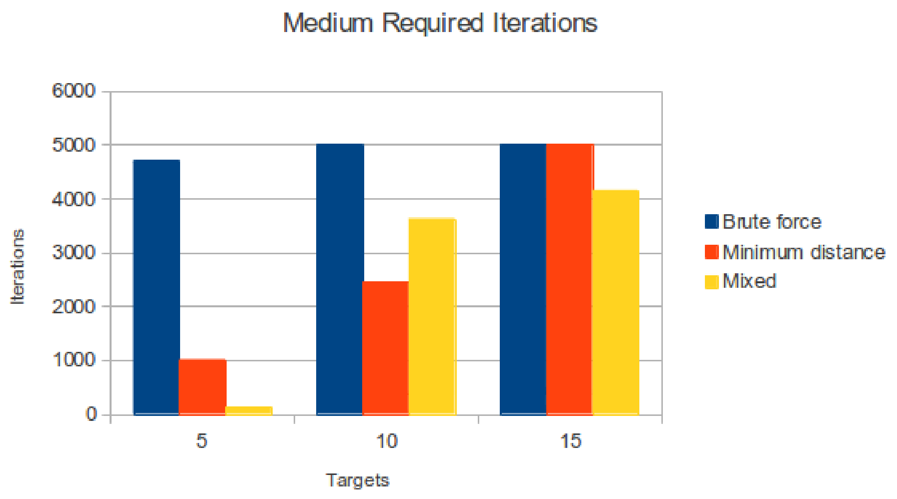


Figure 3.7: Medium iterations

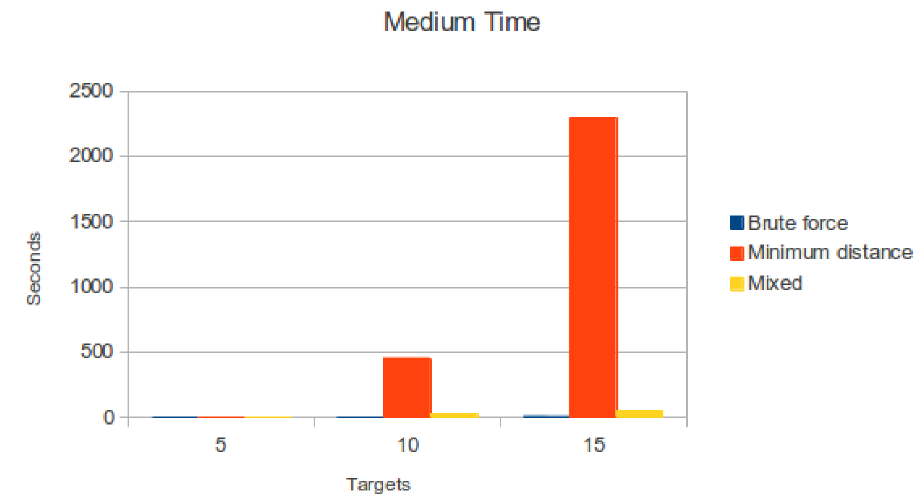


Figure 3.8: Medium time

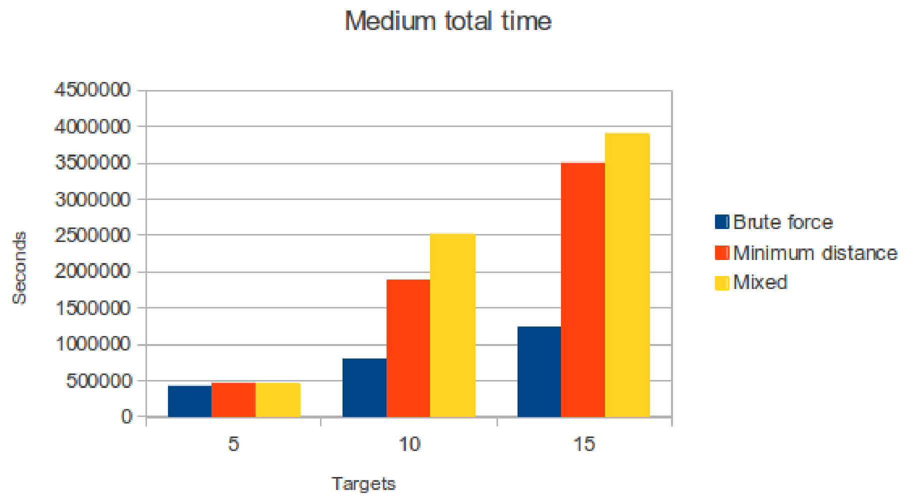


Figure 3.9: Medium execution time

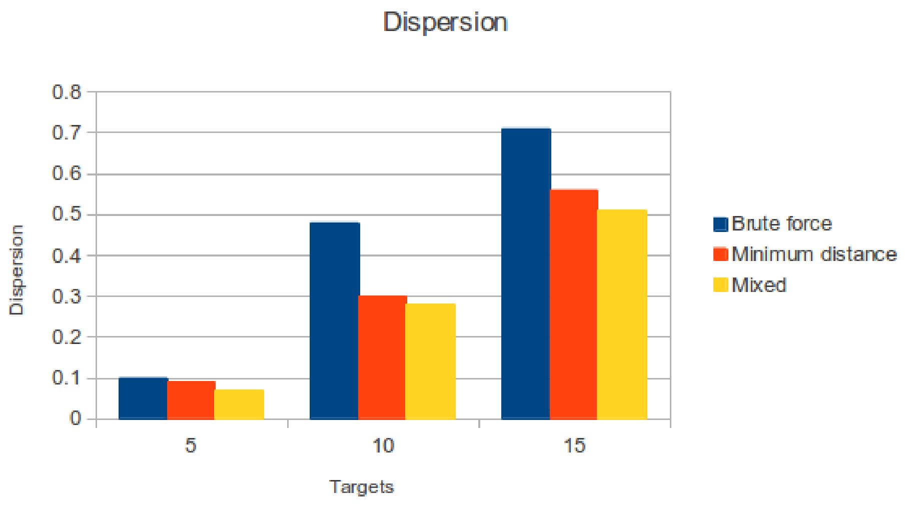


Figure 3.10: Result dispersion

As shown the mixed algorithm is much faster than the minimum distance algorithm and not much slower than the brute force algorithm, more cohesive that either of them, and it usually takes less iterations than any of them. With the mixed algorithm the benefits of each algorithm can be obtained without the associated drawbacks, and thus, this is the selected algorithm.
To implement the mixed algorithm, besides the aforementioned modifications the following operations need to be reimplemented:

Initial population: The initial population is calculated by generating a number of individuals with random pickup order and the applying the minimum distance algorithm to each one of them.

Reproduction: Reproduction remains quite similar, but the time difference between the current satellite and the "Target" is passed to the new individual as well.

Mutation: There are two different mutation algorithms. On one hand a regular mutation is performed exactly the same than in the previous algorithm, by swapping two satellites in the order (time is unmodified). On the other hand when a "Time mutation" occurs time is recalculated for the whole individual using the minimum distance algorithm.

3.2.3 Moving Targets and drop point

To complete the algorithm the drop point have to be introduced. The Debris Collector can only take a number of "Targets" with it before needing to move to the drop point to release them. As the number of "Targets" that can be captured is depending on its hardware, but is always the same value the only required modifications to the algorithm is to limit the number of "Targets" to be picked up in a single run. This has the added benefit of reducing memory consumption.

3.3 Planner node documentation

The Planner node will be responsible for executing the genetic algorithm whenever it is required. This node uses both the model and the genetic libraries.

3.3.1 Parametrization

This node's behavior can be tweaked by modifying the following parameters.

MAX_ITERATIONS Maximum number of iterations allowed in the genetic algorithm. After that many iterations the best result so far will be returned.

CONSECUTIVE_BESTS The number of consecutive convergent iterations to consider the algorithm to have converged.

MAX_DIFERENTIAL The goodness of each iteration in relation to the previous iteration is calculated as $\frac{|best_{i-1}-best_i|}{max(best_{i-1},besti)}$, if this number is smaller than the MAX_DIFERENTIAL parameter those two iterations are convergent.

MUTATION_PROBABILITY The probability (in percentage) of a order mutation occurring.

MIGRATION_PROBABILITY The probability (in percentage) of a migration occurring.

TIME_MUTATION_PROBABILITY The probability (in percentage) of a time mutation (recalculating all pickup times using the gradient descend) occurring.

3.3.2 Topics

This node reads information from the following topics:

Objects The Topic in which every surrounding object position is published by the world node.

Location The Topic in which the robot position is published by the locator node.

It also publishes information in the following topic:

Plan The Topic in which the devised plan is published as an ordered series of targets and pickup times.

3.3.3 Services

This node offers the “NewTaskList” service, which receives no parameters and returns no parameters. This service requests the plan devising algorithm to be executed.

Chapter 4

The locator node

In this chapter the “Locator” node, which will be responsible of finding the current position of the robot at any time, is built.

4.1 Problem definition

A common problem in every mobile robot is how to find out where it is in a reliable way. Measurement systems have always a certain margin of error, and motors and other movement providers may have a certain error too. At the same time if the initial position is unknown it is usually quite expensive to compute the position by using triangulation or other trigonometry algorithms.

4.2 Particle filters

One of the most powerful tools to localize a robot is a particle filter [14]. A particle filter is a simple yet powerful algorithm which takes the uncertainty of measurements and movements into account to provide an accurate guess of the current position.
To understand how the particle filters work there are a few definitions that should be explained first.

- **Particle** A particle is a possible position and orientation of the robot.
- **Localizer beacons** For this algorithm to work a set of immobile, easy to locate points have to be defined. As they do not move the robot can measure its distance to them as a reference frame.
- **Resampling (with replacement)** The process of resampling with replacement consists in selecting elements from a group at random in accordance to its probability of being picked up until a new group with the same cardinality has been created.

The algorithm works as follows:

1 Create a number of particles with random positions.

2 Calculate the actual distance between each particle and the localizer beacons.

3 Assign a weight to each particle depending on the error between the calculated distances and the real distance between the object and the beacons.

4 Generate a new particle population by resampling the previous population.

5 Apply the same movement to the robot and to every particle.

Return to 2

As can be seen the algorithm is quite simple. Usually the movement of the robot and particles is defined as a rotation and a movement speed. As each particle (and the actual robot) have an orientation the movement is assumed to be in the same direction as the orientation.

4.3 Simplified Scattered Particle Filter

The Simplified Scattered Particle Filter (SSPF) used makes some minor changes to the original algorithm. One of the biggest drawbacks of the Particle Filter is the fact that particles are never modified, just updated (in step 5 the same movement is applied to both the particles and the robot). Sometimes this leads to a point where the particles that have survived have a wrong orientation, and they move away from the robot. As no new particles are added if this situation appears the algorithm has no way to readapt itself, and the error in the localization increases with no solution.
To avoid this problem the scattering method has been introduced. In step 4, every time a new particle is copied, a small possibility of scattering is introduced. If the particle is scattered it's location is randomly changed (by a small amount). If the particle has a bigger measurement error than in the previous iteration (which means that it has a different orientation than the orientation of the actual robot) its orientation is changed at random. In both situations (either position or orientation scattering) is possible to end up with a worst particle, but the possibility of introducing a new, more correct, particle also appears, making the algorithm able to adapt itself and solve the increasing error situation described before.
The algorithm used in the robot has been simplified as well, by ignoring the possible error in the measurements and movements. Though this makes the algorithm less reliable, it is easier to implement and test.

4.4 Locator node documentation

This node computes the position of the robot using the SSPF. It uses both the model and particle libraries.

4.4.1 Parametrization

This node's behavior can be tweaked by modifying the following parameters.

MAX_ERROR The max error allowed. This error is calculates as the mean error of each particle, which is calculated as the sum of the individual errors between the particle to beacon distance and its expected distance. Once the error is smaller or equal to this parameter the node will start publishing the computed location.

4.4.2 Topics

This node reads information from the following topics:

Move The topic in which every movement the Debris Collector has to make is published by the executer node.

Beacon The topic in which each visible beacon and the current distance to it is published by the world node.

It also publishes information in the following topic:

Location The current position of the Debris Collector is published in this topic.

4.4.3 Services

This node offers and uses no services.

Chapter 5

The task executer node

In this chapter the “Executer” node will be built.

5.1 Problem definition

Once the Debris Collector knows where it is, and what it has to do it need to do the actual work, translating the devised plan into a series of simple tasks. This node will also be responsible for avoiding any “Obstacles” the Debris Collector encounters.

5.2 Decision Making

To transform the devised plan into a series of tasks a finite state machine (fsm) will be implement. This fsm will choose the action to undertake at every situation.

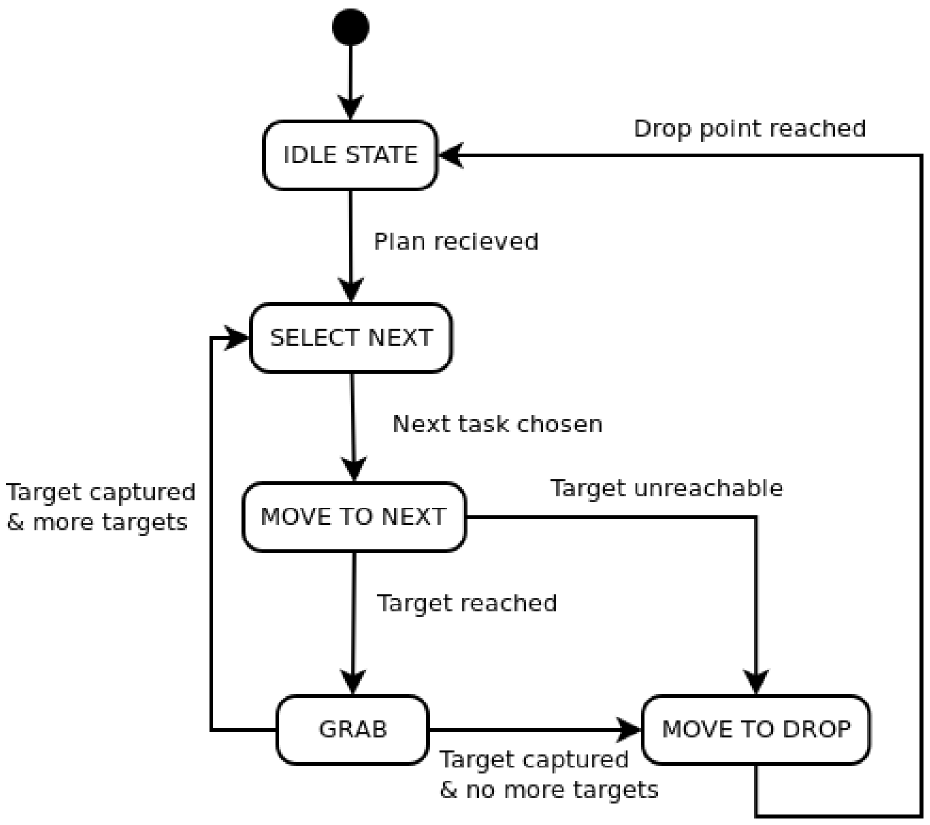


Figure 5.1: The UML state diagram defining the finite-state machine for the executer node.

As shown on the image the robot will start at the “IDLE_STATE” state. In this state the Executer node will request a plan to the Planner node until it receives a new plan. Once this happens the Executer will move to the “SELECT_NEXT” state. In this state the next element in the plan (if there are elements in the plan) will be taken as the task at hand and then the Debris Collector will proceed to the “MOVE_TO_NEXT” state. In this state the Debris Collector will calculate the movement to be done to get to the calculated pickup point. Once the Debris Collector gets to the pickup point the Debris Collector will proceed to the “GRAB” state, in which the robot will then try to grab the “Target”. Once the “Target” has been grabbed the Debris Collector will proceed to the “SELECT_NEXT” state again if there are more tasks in the plan. If, when in the “GRAB” state, there are no more tasks to be done, or when in the “MOVE_TO_NEXT” state the computed movement can’t be done the Debris Collector proceeds to the “MOVE_TO_DROP” state. In this state the robot moves to the predefined drop point to release its payload. Once this is done the Debris Collector returns to the “IDLE_STATE” again.

5.3 The perpendicular elastic force algorithm

Collision avoidance is usually a difficult task. One of the multiple algorithms for collision avoidance is the elastic force algorithm (or elastic bands as it is sometimes called) [15]. In this algorithm a force is applied to the robot that moves it to its intended target. At the same time a virtual repulsion field around every obstacle is created. If the robot enters this field a repulsion force is applied to it, forcing it away from the obstacle. This algorithm is devised for a two dimension situation, but it can be modified for a three dimensional scenario.
The Perpendicular Elastic Force algorithm (PEF) is a very similar algorithm that extends the elastic force algorithm to three dimensions. In the PEF an elastic force between the robot and its intended target is calculated as well, called $\vec{V_{robot}}$, and a spherical repulsion field is calculated around every target as in the original algorithm. The repulsive force applied by each object is calculated as:

$$\begin{aligned} \vec{F_r^+} &= (\vec{D'} - \vec{D}) \times \vec{V} \\ \vec{F_r^-} &= \vec{V} \times (\vec{D'} - \vec{D}) \end{aligned} \tag{5.1}$$

Where $\vec{D}$ is the vector from the obstacle to the robot, $\vec{D'}$ is a vector in the same direction that $\vec{D}$ with norm equal to the security distance between the robot and an obstacle and $\vec{V}$ is the vector representing the obstacle speed.
Then the less disruptive force, meaning the one which implies a smaller change in the original elastic force applied to the robot is chosen, so:

$$\vec{F_r} = \begin{cases} \vec{F_r^+} & \text{if } \frac{\vec{F_r^+} \cdot \vec{V_{robot}}}{|\vec{F_r^+}||\vec{V_{robot}}|} > \frac{\vec{F_r^-} \cdot \vec{V_{robot}}}{|\vec{F_r^+}||\vec{V_{robot}}|} \\ \vec{F_r^-} & \text{otherwise} \end{cases} \tag{5.2}$$

This repulsive force is always perpendicular to the plane containing the obstacles movement vector and the distance vector between the robot and the the obstacle, thus ensuring the robot will either dive under the movement curve of the obstacle or over it. As the less disruptive one is always chosen this implies that the resulting path will be a smooth one.
Then the resulting movement is:

$$\vec{M} = \vec{V_{robot}} + \sum_{i=0}^{N} \begin{cases} \vec{F_r} & \text{if } |\vec{D}| < Distance_s \\ (0,0,0) & \text{otherwise} \end{cases} \tag{5.3}$$

With $Distance_s$ being the the security distance between the robot and an obstacle.

5.4 Node documentation

This node decides what should the robot do at any time. It uses both the model and the elastic libraries.

5.4.1 Parametrization

This node's behavior can be tweaked by modifying the following parameters.

DROP_SPEED The speed used when moving to the drop point.

REACH The distance to the "Target" in which the "GRAB" state should be entered.

SAFETY_DISTANCE The minimum allowed distance to an obstacle.

MAX_SPEED The maximum speed the Debris Collector is allowed to get.

5.4.2 Topics

This node reads information from the following topics:

Objects The Topic in which every surrounding object position is published by the world node.

Plan The Topic in which the plan is published by the planner node.

Location The Topic in which the robot position is published by the locator node.

It also publishes information in the following topic:

Move The calculated move, expressed as a quaternion rotation and the current speed that should be applied in the orientation direction (after the rotation), is published in this topic.

5.4.3 Services

This node uses the following services:

NewTaskList A service provided by the planner node to request a new plan.

Grab A service provided by the world node to grab a “Target”.

Chapter 6

Conclusions

6.1 The Debris Collector Core

During this paper the Debris Collector core (DCc) components has been designed and built. The Dcc is a multi-agent system which controls an unmanned spacecraft to dispose of spacial debris. The DCc is fully autonomous, being able to identify the debris pieces that should be collected around him (called "Targets"), devising a plan to capture as many as possible while keeping fuel consumption to the minimum and capturing and moving them to a predefined drop point while avoiding collisions with other objects in its neighborhood (called "Obstacles").
The DCc makes intensive use of several AI algorithms, such as genetic algorithms to devise a plan, particle filters to identify its current position in the world and the elastic force algorithm to avoid nearby obstacles. It has been programed using the Robot Operating System (ROS), which provides a framework for robot intelligences using multi-agent systems.
This paper presents only the core agents of the total agent network for the DCc as well as a simulator agent which creates a virtual world around the DCc and supplies the information that other nodes or actuators should supply.

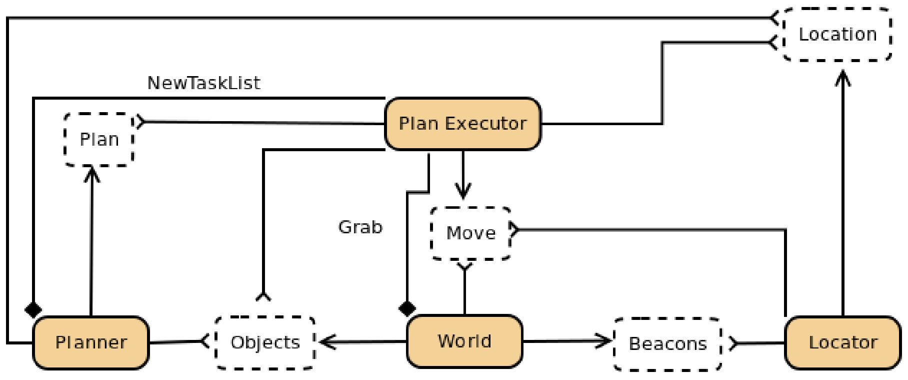


Figure 6.1: The agent map shows how the different agents interact between them. Round-cornered rectangles represent the Nodes, round-cornered rectangles with dotted borders represent the Topics. The lines ending in a diamond represent service calls.

As shown in figure 6.1 there are four agents in the DCc:

World The world node is the agent responsible for the simulated world. It contains a representation of the current state of the model, methods to update it and cmomunication topics to publish the objects information and apply the DCc movements.

Locator The locator node is the agent responsible for identifying the current

DCc position in the world, and communicates this information to the other nodes through the Location Topic.

Planner The planner node is the agent responsible for devising a plan to capture as many “Targets” as possible in a single run in the cheapest possible way (cheapest meaning less fuel-consuming) whenever it is requested through the “NewTaskList” service. It communicates the devised plan through the Plan Topic.

Plan executor The plan executor node is the agent responsible for interpreting the plan and transforming it into a series of simple movements for the DCc to make. It is also the agent responsible for avoiding collisions with surrounding Obstacles.

Those four simple intelligent agents create a multi-agent system able to autonomously drive a spacecraft to collect space debris and get it to a safe far away from the commonly used orbits.

6.2 Tests

The Debris Collector has been developed and tested in the following environment:

Type of computer: Virtual machine (using VirtualBox).

Operative System: Fedora 18 (Spherical Cow).

CPU: 4 CPU’s, each at 3.07GHz.

RAM: 16384 Mb.

Disk Space: 20 Gb.

ROS version: groovy.

There have been a number of tests through the projects life, summarized here. Each test shows which nodes should be used for that test and what checks where performed.

World Node Test Start the world node alone.

- Check: All objects are updated correctly through time.
- Check: Object information is correctly published when an object is in sensor range.
- Check: Targets are removed when Grab service is called and Targets are in grapple reach.

Locator Node Test Start the locator node together with the world node.

- Check: The locator's internal beacon list is updated correctly through time.
- Check: The algorithm runs in under a second to avoid clock incoherences.
- Check: The algorithm has a generally decreasing error, meaning that, even if sometimes the error may increase the error-time function is usually decreasing.

Planner Node Test Start the planner node together with the world node and the locator.

- Check: the planner's internal object list is correctly updated through time.
- Check: The DCc position is correctly updated through time.
- Check: The algorithm is run when required.
- Check: The algorithm returns a viable plan (strictly increasing pickup times, all selected Targets in sensor range).

Executor Node Test Start all four nodes (the executor node requires information from all other nodes to work).

- Check: the executor's internal object list is correctly updated through time.
- Check: The DCc position is correctly updated through time.
- Check: The executor state is coherent with the current situation through time.
- Check: The elastic force algorithm runs in under a second to avoid clock incoherences.
- Check: The elastic force algorithm returns the correct rotation quaternions.
- Check: The finite-state machine behaves as specified.

DCc Test Start all four nodes.

- Check: The DCc behaves as expected.
- Check: The DCc reports no collisions.
- Check: The DCc captures all Targets.
- Check: The DCc reaches the predefined drop point.

6.3 Further improvements

Besides implementing the remaining nodes (VisualDigester and Pilot), and constructing the actual robot, there are some improvements that could be made in the code.

6.3.1 Planner improvements

There are some improvements that could be made to the genetic algorithm. Using three parents on the reproduction step or improving the gradient descent algorithm should lead to faster and more cohesive results for this algorithm.

6.3.2 Locator improvements

As explained in the corresponding chapter the particle filter algorithm used in the locator node has been stripped of uncertainty, which should be implemented back. Besides the uncertainty addition the scattering method should be revisited, and tested thoroughly to make sure that its application usually improves the resulting particles, which is not guaranteed right now. Additionally the locator should be able to determine the current orientation.

6.3.3 Executer improvements

The executer node should be tested over highly crowded areas, which slow its performance and could potentially lead to unavoidable collisions. The GRAB state should also include the necessary algorithms for repositioning, orbit coupling and actually grabbing the target.

6.3.4 Other uses

The presented nodes are the core of the Debris Collector , but they could easily be adapted for a number of other robots. The presented core creates a robot able to navigate a three-dimensional environment to move to a set of points while avoiding obstacles. With little effort it could be adapted for an oceanic or airborne environment.

Bibliography

[1] National Aeronautics and Space Administration (NASA). “The Threat of Orbital Debris and Protecting NASA Space Assets from Satellite Collisions”. <http://tinyurl.com/lrdzpdb >. Retrieved 08 April 2013.

[2] de Selding, P. “MDA Designing In-orbit Servicing Spacecraft”, Space News, 3 March 2010. Retrieved 15 July 2011.

[3] Multiple authors. “Space Debris Removal”[on-line], Star-tech-inc.com. <http://tinyurl.com/6eqm2u5 >Retrieved 08 April 2013.

[4] Kessler,D.J., & Cour-Palais, B.G. “Collision Frequency of Artificial Satellites: The Creation of a Debris Belt”, Journal of Geophysical Research, Vol. 83, No. A6, pp. 2637-2646, June 1, 1978.

[5] Newton, I., Bernoulli, D., MacLaurin, C., & Euler, L. (1833). “Philosophiae naturalis principia mathematica (Vol. 1)”. excudit G. Brookman; impensis TT et J. Tegg, Londini.

[6] Sutton, G. P., & Biblarz, O. (2011). “Rocket propulsion elements.” Wiley.

[7] Beattie, J. R. (1989). U.S. “Patent No. 4,838,021”. Washington, DC: U.S. Patent and Trademark Office.

[8] Black, Paul E (12 May 2008). “Finite State Machine”. Dictionary of Algorithms and Data Structures (U.S. National Institute of Standards and Technology).

[9] Multiple authors. “ROS Homepage”[on-line] <http://tinyurl.com/y9l92rg >. Retrieved 08 April 2013.

[10] Ferber, J. (1999). “Multi-agent systems: an introduction to distributed artificial intelligence (Vol. 1)”. Reading: Addison-Wesley.

[11] Wooldridge, M., & Jennings, N. R. (1995). “Intelligent agents: Theory and practice”. Knowledge engineering review, 10(2), 115-152.

[12] Applegate, D. L., Bixby, R. M., Chvtal, V. & Cook, W. J. (2006). “The Traveling Salesman Problem”

[13] Goldberg, D. E. (1989). “Genetic algorithms in search, optimization, and machine learning.”

[14] Thrun, S. (2002, August). “Particle filters in robotics.” In Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence (pp. 511-518). Morgan Kaufmann Publishers Inc..

[15] Quinlan, S. & Khatib, O. (1993, May). “Elastic bands: Connecting path planning and control.” In Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on (pp. 802-807). IEEE.

Appendix A

Project Documentation

This appendix cover the documentation for all libraries and classes used in the code.

A.1 Model Library

The model library (model.h) is a generic library containing classes and methods to represent the model, thus ensuring its cohesion in every Node of the resulting system.

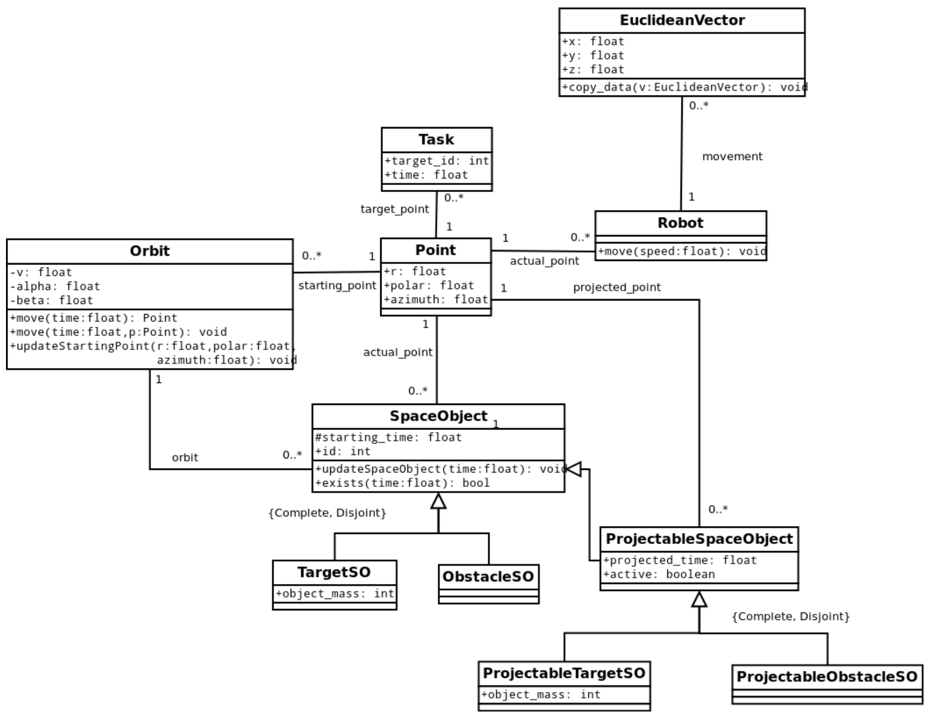


Figure A.1: The UML diagram of the classes found on this library.

A.1.1 Classes

The model library contains several classes used to represent the model:

Point The Point class contains a representation of the spherical coordinates of a given point. It also contains some methods, such as an appropriate constructor and both equals and differs operator overloads.
Its attributes are:

- **r** radial distance in meters, stored as a float.
- **polar** polar angle in degrees, stored as a float.

- **azimuth** azimuth angle in degrees, stored as a float.

Its methods are:

- **Point(float r, float polar, float azimuth)** The constructor method for this class stores the given values in the corresponding fields. It is also responsible of ensuring that both polar and azimuth angles fall in the [0, 360) range.

Orbit The Orbit class contains a representation of an orbit as a starting point and its trajectory.
Its attributes are:

- **starting_point** the orbit starting point as a Point object.
- **v** radial speed in meters per second, stored as a float.
- **alpha** polar speed in degrees per second, stored as a float.
- **beta** azimuthal speed in degrees per second, stored as a float.

Its methods are:

- **Orbit(float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed)** The constructor method for this class stores the given values in the corresponding fields, creating the required starting_point with the data provided.
- **Point move(float* time)** This method takes the elapsed time since this orbit started as a float argument and returns the current point as a Point object.
- **void move(float* time, Point* p)** Updates parameter p to be the current point after the elapsed time.
- **void updateStartingPoint(float r, float polar, float azimuth)** Updates the starting point to the point defined in the parameters.

SpaceObject The SpaceObject class combines the previous classes to provide a model for every single object in our simulation.
Its attributes are:

- **id** the object unique identifier, stored as an int. It should be a unique identifier for each SpaceObject, but it is not ensured in this library.
- **current_point** the current point in space where this object is as a Point object.
- **orbit** this object's orbit as an Orbit object.
- **starting_time** the moment in the general time frame when this object was created in seconds, stored as a float.

Its methods are:

- **SpaceObject(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed, float time)** The constructor method for this class stores the given values in the corresponding fields, creating the orbit and current_point with the supplied data.
- **void updateSpaceObject(float* time)** This method takes the elapsed time since the simulation started as a float argument and updates the object's position by setting its current_position attribute to the result of the orbit's move function, using as a parameter the time parameter given minus the starting_time for this object.
- **bool exist(float time)** This method takes the elapsed time since the simulation started as a float argument and test if the object should exist at that time (if its starting_time is smaller than the parameter).

TargetSO The TargetSO class inherits from SpaceObject to represent a target object.
Its non-inherited attributes are:

- **object_mass** the mass of the object, as an int value.

Its non-inherited methods are:

- **TargetSO(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed, float time, int mass)** The constructor method for this class creates a TargetSO and stores each value in its corresponding field, creating the required Point and Orbit objects with the provided data.

ObstacleSO The ObstacleSO class inherits from SpaceObject to represent an Obstacle object.
This class has no non-inherited attributes. Its non-inherited methods are:

- **ObstacleSO(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed, float time)** The constructor method for this class creates an ObstacleSO and stores each value in its corresponding field, creating the required Point and Orbit objects with the provided data.

ProjectableSpaceObject The ProjectableSpaceObject class inherits from the SpaceObject class to create a projectable object, able to predict its position at any given time.
Its non-inherited attributes are:

- **projected_point** The point where this object will be after an amount of time, stored as a private Point pointer.
- **projected_time** The corresponding time to the projected point, stored as a float pointer.

- **active** This value is used in the executer and the planner nodes to know if this object is still in sensor reach.

Its non-inherited methods are:

- **ProjectableSpaceObject(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed)** The constructor for this class creates a ProjectableSpaceObject with the supplied parameters.
- **Point* project(float time)** This method calculates where will the object be after time seconds have passed and returns the reference to projected_point. If the same method is called with the same time value the point is not recalculated as it is stored in projected_point.

ProjectableTargetSO The ProjectableTargetSO class inherits from ProjectableSpaceObject to represent a target object.
Its non-inherited attributes are:

- **object_mass** the mass of the object, as an int value.

Its non-inherited methods are:

- **ProjectableTargetSO(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed, int mass)** The constructor for this class creates a ProjectableTargetSO with the supplied parameters.

ProjectableObstacleSO The ProjectableObstacleSO class inherits from ProjectableSpaceObject to represent a target object.
Its non-inherited methods are:

- **ProjectableObstacleSO(int new_id, float r, float polar, float azimuth, float speed, float polar_speed, float azimuthal_speed)** The constructor for this class creates a ProjectableObstacleSO with the supplied parameters.

Task The Task class represents a task the robot has to do, storing the pickup point, the time when this point has to be reached and the id of the target.
Its attributes are:

- **target_id** The id of the "Target" object to be picked up, stored as an int.
- **time** The time when the object has to be picked up, stored as a float.
- **target_point** The point where the pickup will be made, stored as a Point object.

Its methods are:

- **Task(int target, float dest_time, const std::vector<ProjectableSpaceObject*>&ob** The constructor of this class receives a list of known objects, finds the object with the desired id (target parameter) and calculates the pickup point (at dest_time).

EuclideanVector The EuclideanVector vector represents a cartesian vector in euclidean space. Its attributes are:

- **x** The x coordinate, stored as a float.
- **y** The y coordinate, stored as a float.
- **z** The z coordinate, stored as a float.

Its methods are:

- **EuclideanVector()** The empty constructor creates an empty vector (values $(0, 0, 0)$).
- **EuclideanVector(float x0, float y0, float z0)** This constructor initializes an EuclideanVector with the desired coordinates.
- **void copy_data(EuclideanVector* v)** This method copies the EuclideanVector coordinates of v in this EuclideanVector.

Robot The Robot class represents the Debris Collector . It stores its position and orientation and contains methods to move it. Its attributes are:

- **current_point** The current point where the Debris Collector is, stored as a Point object.
- **movement** The orientation the Debris Collector has, stored as an EuclideanVector object.

Its methods are:

- **Robot()** The empty constructor initializes this class with an empty Point and EuclideanVector.
- **Robot(float r, float polar, float azimuth, float vx, float vy, float vz)** This constructor initializes the class with the given values.
- **void move(float speed)** This method moves the object along a vector with norm speed and in the direction of the movement attribute.

A.1.2 Methods

The model library contains some general methods that should be used when interacting with the model.

float distance(Point* a, Point* b) This method calculates the euclidean distance in meters between two Points and returns it as a float.

bool collision(Point* a, Point* b) This method checks if both Points are equal.

float to_deg(float angle) This method is used to make sure that all angles have a value between 0 and 360. If the parameter is outside this range it is projected into it and returned as a float.

float norm(EuclideanVector* v) This method calculates the norm of the vector v and returns it as a float.

void quaternion_to_rot(float x, float y, float z, float w, std::vector<std::vector<float>>& rotation) This method transforms a quaternion received in parameters x, y, z and w into a rotation matrix, stored in the rotation parameter.

void apply_rotation(EuclideanVector* v1, std::vector<std::vector<float>>& rotation) This method applies the rotation matrix rotation to the vector v1.

A.2 Genetics Library

The genetics library (genetics.h) is a generic library containing the required methods to apply the genetic algorithm explained in chapter 3. It uses some classes and methods found in the model library.

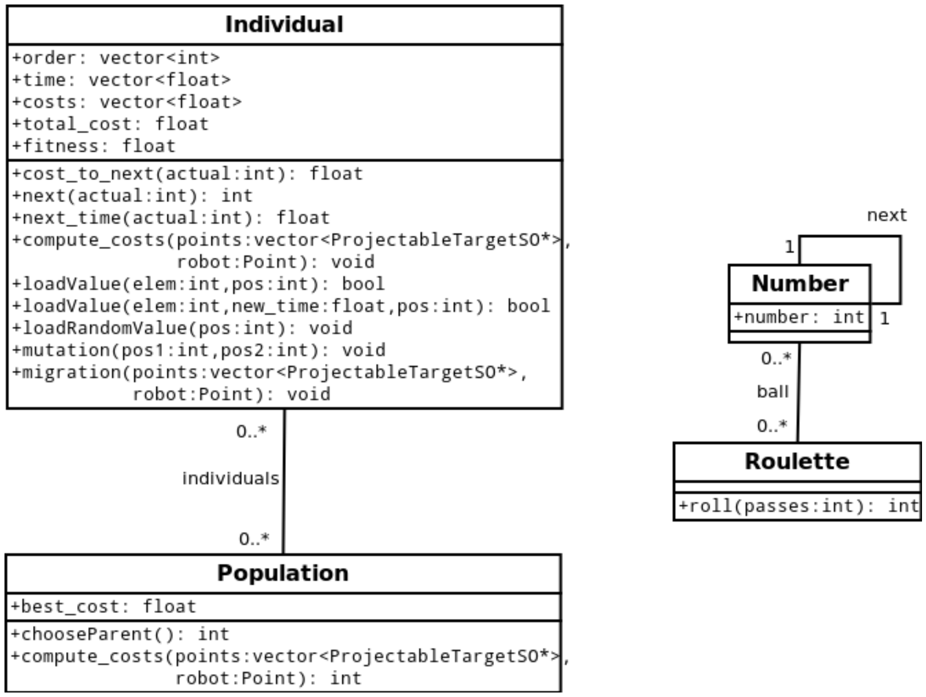

Individual
+order: vector<int>
+time: vector<float>
+costs: vector<float>
+total_cost: float
+fitness: float
+cost_to_next(actual:int): float
+next(actual:int): int
+next_time(actual:int): float
+compute_costs(points:vector<ProjectableTargetSO*>, robot:Point): void
+loadValue(elem:int,pos:int): bool
+loadValue(elem:int,new_time:float,pos:int): bool
+loadRandomValue(pos:int): void
+mutation(pos1:int,pos2:int): void
+migration(points:vector<ProjectableTargetSO*>, robot:Point): void
0..*
individuals
0..*
Population
+best_cost: float
+chooseParent(): int
+compute_costs(points:vector<ProjectableTargetSO*>, robot:Point): int
next
1
Number
+number: int
1
0..*
ball
0..*
Roulette
+roll(passes:int): int

Figure A.2: The UML diagram of the classes found on this library.

This library uses some constants defined in the “genetics.cpp” file as well.

MIN_TIME_STEP The minimum time step between two pickup times in seconds.

MIN_GRAD If the gradient of the distance function is smaller than this value the gradient descent is considered to have converged.

MAX_CARRIED_OBJ The maximum number of objects the Debris Collector can carry at the same time.

A.2.1 Classes

The genetics library contains several classes used to apply the genetic algorithm:

Roulette The Roulette class is a circular list containing numbers. It is used to speed up the generation of random, unique numbers.
Its attributes are:

- **ball** A pointer to a Number object. Each Number object is a struct containing an int called number, and a pointer to the next Number in the list, called next.

Its methods are:

- **Roulette(int max)** The constructor method for this class creates a circular list with all numbers between 0 and max minus one (inclusive).
- **int roll(int passes)** This methods moves the ball pointer "passes" times, returns the corresponding number and removes it from the circular list.

Individual The Individual class represents a possible solution for the genetic algorithm. It also contains several of the required methods.
Its attributes are:

- **order** Represents the order in which "Targets" should be picked up. It is stored as an int vector.
- **time** Represents the times in which "Targets" should be picked up. It is stored as a float vector, each position holding the time in which the corresponding "Target" in the order vector should be picked up.
- **cost** This float vector stores the cost of going from the previous chromosome to the corresponding "Target".
- **total_cost** The cost of the complete path (the sum of all costs in the cost vector), stored as float pointer.
- **fitness** The fitness of the corresponding individual.

Its methods are:

- **Individual(int size)** This constructor creates an empty Individual instance, with its vectors resized to size.
- **Individual(int size, const vector<ProjectableTargetSO*>&points, Point* robot)** This constructor creates an individual with order filled up and time calculated minimizing distance.
- **float cost_to_next(int current)** Return the cost from the current chromosome to its next chromosome in the order.
- **int next(int current)** Returns the next chromosome in the order.
- **float next_time(int current)** Returns the time increment from the current chromosome pick up time to the pick up time of the next chromosome in the order.
- **void compute_costs(const vector<ProjectableTargetSO*>&points, Point* robot)** Calculates the costs of this individual.
- **bool loadValue(int elem, int pos)** Loads a new value in position pos in the order vector if it is not yet present, otherwise returns false.
- **bool loadValue(int elem, float new_time, int pos)** Loads a new value in position pos in the order vector and its corresponding time if it is not yet present, otherwise returns false.
- **void loadRandomValue(int pos)** Loads a random "Target" in the order in position pos.
- **void mutation(int pos1, int pos2)** Swaps the "Targets" at position pos1 and pos2 in the order vector.
- **void migration(const vector<ProjectableTargetSO*>&points, Point* robot)** Loads this individual with a random order and times calculated with the minimum distance method.

Population The Population class contains all Individual solutions, as well as some methods required by the genetic algorithm.
Its attributes are:

- **best_cost** The best cost found in the population.
- **individuals** A vector containing all Individuals in this Population.

Its methods are:

- **Population(int size, const vector<ProjectableTargetSO*>&points, Point* robot** This constructor creates a Population of $size*(size-1)$ Individuals with its order vectors loaded using the Individual constructor with the same parameters.
- **Population(int size)** This constructor creates a Population of $size*(size-1)$ empty Individuals using the Individual constructor with the same parameters.

- **int chooseParent()** This method returns the position in the individuals vector of an Individual chosen randomly according to its fitness.
- **int compute_costs(const vector<ProjectableTargetSO*>&points, Point* robot** This method calculates the costs of all Individual instances.

A.2.2 Methods

The genetics library contains some general methods that are required for the genetic algorithm to work.

float random_float(float a, float b) This method returns a random float value between a and b.

float loadTime(Point* a, ProjectableTargetSO* b, float t) This method runs the gradient descent algorithm with origin point a, function to minimize b and minimum time t.

A.3 Particle Library

The particle library (particle.h) is a generic library containing classes required for the particle filter algorithm. It uses some classes and methods found in the model library.

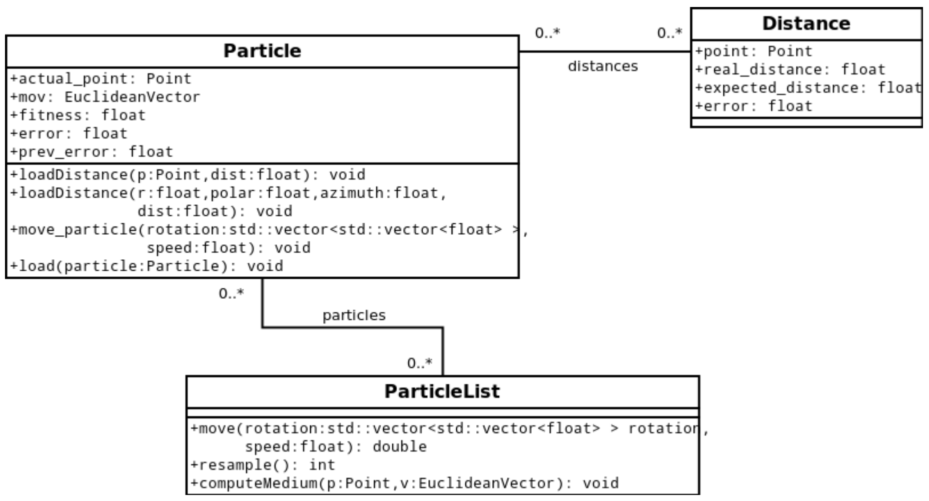

Particle
+actual_point: Point
+mov: EuclideanVector
+fitness: float
+error: float
+prev_error: float
+loadDistance(p:Point,dist:float): void
+loadDistance(r:float,polar:float,azimuth:float, dist:float): void
+move_particle(rotation:std::vector<std::vector<float> >, speed:float): void
+load(particle:Particle): void
0..*
distances
0..*
Distance
+point: Point
+real_distance: float
+expected_distance: float
+error: float
0..*
particles
0..*
ParticleList
+move(rotation:std::vector<std::vector<float> > rotation, speed:float): double
+resample(): int
+computeMedium(p:Point,v:EuclideanVector): void

Figure A.3: The UML diagram of the classes found on this library.

This library uses some constants defined in the "particle.cpp" file as well.

LIST_SIZE Number of particles created in the algorithm.

SCATTER_PROB Probability of scatter occurring (in percentage).

The parameters MAX_ERROR (ME), MIN_PROBABILITY (MP), BIG_ERROR (BE), BE_PROBABILITY (BEP), SMALL_ERROR (SE) and SE_PROBABILITY (SEP) define a probability function depending on the error. The equation has the following formula:

$$Probability(x) = \begin{cases} MP & \text{if } x > ME \\ \frac{BEP-MP}{BE-ME}(x-ME)+MP & \text{if } ME >= x > BE \\ \frac{SEP-BEP}{SE-BE}(x-BE)+BEP & \text{if } BE >= x > SE \\ \frac{1-SEP}{0-SE}(x-SE)+SEP & \text{if } SE >= x \end{cases}$$

A.3.1 Classes

The particle library contains several classes used to apply the particle filter algorithm:

Distance The Distance struct contains a representation of every distance between a given particle and one of the localizer beacons.
Its attributes are:

- **point** The localizer beacon point, stored as a Point object.
- **real_distance** The distance between the particle and the localizer beacon, stored as a float.
- **expected_distance** The distance between the robot and the localizer beacon, stored as a float.
- **error** The error between distances, stored as a float.

Particle The Particle class contains all the information required in a particle on the algorithm. It contains several methods for the particle filter.
Its attributes are:

- **distances** A private vector of Distance objects containing the Distance objects for all beacons.
- **current_point** The point where this particle is located, stored as a Point object.
- **mov** The orientation this particle has, stored as an EuclideanVector object.
- **fitness** The fitness (probability of being resampled) of this particle, stored as a float.
- **error** The sum of the errors in the distances, stored as a float.
- **prev_error** The error this particle had in the previous iteration, stored as a float.

Its methods are:

- **Particle()** The empty constructor instantiates a Particle object with its parameters empty.
- **Particle(float r, float polar, float azimuth, float vx, float vy, float vz)** This constructor initializes a Particle object with the desired values of position and orientation.
- **void loadDistance(Point* p, float dist)** Loads a new Distance object, with the beacon point and expected distance as specified in the parameters.
- **void loadDistance(float r, float polar, float azimuth, float dist)** Loads a new Distance object, with the beacon point and expected distance as specified in the parameters.
- **void move_particle(float speed, std::vector<std::vector<float>>rotation)** Moves the particle. It works exactly as the "move(float speed)" in the Robot class. After the movement is applied computes the real distance of all Distance objects in this Particle.
- **void load(Particle* particle)** Copies the values of the particle passed as a parameter, applying scattering if required.

ParticleList The ParticleList contains a list of particles, and the methods to manipulate it. Its attributes are:

- **particles** The list of particles, stored as a vector of Particle objects.

Its methods are:

- **ParticleList(bool random)** This constructor initializes the ParticleList object. If random is false it creates an list of empty Particles, otherwise it creates a list of particles which are situated en three concentric spheres at radial distances 100, 150 and 200 km.
- **double move(float speed, std::vector<std::vector<float>>rotation)** Applies a movement to all particles using the move method of every Particle. After that it normalizes the fitness of each Particle. It returns the mean error of the particle list.
- **int resample()** Returns an index to a randomly selected Particle according to its fitness.
- **void computeMedium(Point* p, EuclideanVector* v)** Calculates the mean point and orientation of the particle list.

A.4 Elastic Library

The elastic library (elastic.h) contains some methods needed for the Perpendicular Elastic Force algorithm. It uses some classes and methods from the model library.
This library uses some constants defined in the "elastic.cpp" file as well.

MAX_DIST The repulsive field radius in meters.

A.4.1 Methods

The provided methods are:

void cross_product(EuclideanVector* v1, EuclideanVector* v2, EuclideanVector* result1, EuclideanV This method calculates the cross product (vector product) of v1 and v2 and stores the results in result1 and result2. It calculates both vector products, $v1 \times v2$ and $v2 \times v1$.

float scalar_product(EuclideanVector* v1, EuclideanVector* v2) This method calculates the scalar product of vector v1 and v2 and returns it as a float.

int choose(EuclideanVector* f0, EuclideanVector* f1, EuclideanVector* v0) This method chooses the least disruptive vector between f0 and f1 if applied to v0. It returns 0 if f0 is the least disruptive one and 1 otherwise.

void closeness(EuclideanVector* D) It calculates the closeness vector, defined as $\vec{D'} - \vec{D}$, where $\vec{D}$ is the bector from the obstacle to the robot, and $\vec{D'}$ is the vector with the same direction and norm equal to MAX_DIST.

void orbit_speed(ProjectableSpaceObject* so, EuclideanVector* v) This vector computes the vector in cartesian coordinates of the orbit passed as parameter so. The vector is calculates by calculating the cartesian vector between the current point and the point where this object will be one second from now. This vector is stored in the parameter v.

void distance_vector(Point* p1, Point* p2, EuclideanVector* v) This method calculates the cartesian vector from p1 to p2 and stores it in the v parameter.

void quaternion(EuclideanVector* v1, EuclideanVector* v2, float* x, float* y, float* z, float* w) This method calculates the quaternion representing the rotation from v1 to v2, sorting its values in parameters x, y, z and w.

A.5 Messages

ROS enforces the use of messages (simple classes generated from a .msg file) for data publication to a Topic. This section covers all messages used.

A.5.1 SpaceObjectMsg

The SpaceObjectMsg is the class published to the "Objects" Topic. Its fields are:

uint8 id The object's unique identifier, stored as an unsigned 8-bit int.

float32 r The Object's radial distance, stored as a 32-bit float.

float32 polar The Object's polar angle, stored as a 32-bit float.

float32 azimuth The Object's azimuth angle, stored as a 32-bit float.

float32 v The Object's radial speed, stored as a 32-bit float.

float32 alpha The Object's polar speed, stored as a 32-bit float.

float32 beta The Object's azimuthal speed, stored as a 32-bit float.

uint8 mass The Object's mass, stored as an unsigned 8-bit int.

A.5.2 PlanTaskMsg

The PlanTaskMsg is the class published to the "Plan" Topic. Its fields are:

uint8 task_number The sequence number of this task, stored as an unsigned 8-bit int.

uint8 target_id The id of the TargetSO to pick up, stored as an unsigned 8-bit int.

float32 destination_time The time at which the TargetSO should be picked up, stored as a 32-bit float.

uint8 total_tasks Total number of tasks in this plan, stored as an unsigned 8-bit int.

A.5.3 MovementMsg

The MovementMsg is the class published to the "Move" Topic. Its fields are:

float32 x The x component of the quaternion, stored as a 32-bit float.

float32 y The y component of the quaternion, stored as a 32-bit float.

float32 z The z component of the quaternion, stored as a 32-bit float.

float32 w The w component of the quaternion, stored as a 32-bit float.

float32 speed The speed of the applied movement, stored as a 32-bit float.

A.5.4 LocationMsg

The LocationMsg is the class published to the "Location" Topic. Its fields are:

float32 r The Debris Collector's radial distance, stored as a 32-bit float.

float32 polar The Debris Collector's polar angle, stored as a 32-bit float.

float32 azimuth The Debris Collector's azimuth angle, stored as a 32-bit float.

float32 heading_x The Debris Collector's orientation x coordinate, stored as a 32-bit float.

float32 heading_y The Debris Collector's orientation y coordinate, stored as a 32-bit float.

float32 heading_z The Debris Collector's orientation z coordinate, stored as a 32-bit float.

A.5.5 BeaconMsg

The BeaconMsg is the class published to the "Beacon" Topic. Its fields are:

float32 beacon_r The beacon's radial distance, stored as a 32-bit float.

float32 beacon_polar The beacon's polar angle, stored as a 32-bit float.

float32 beacon_azimuth The beacon's azimuth angle, stored as a 32-bit float.

float32 distance The distance between the Debris Collector and this beacon, stored as a 32-bit float.

A.6 Services

Just as message classes are created for Topics, a similar system is used to define services, in this case in simple .srv files.
There are two services used in the Debris Collector , the Grab service and the NewTaskList. The second one uses and empty parameter, defined in the ROS architecture. On the other hand the Grab service uses its own service file.
Its request parameter is "uint8 id", the id of the Target to be picked up, stored as an unsigned 8-bit int.
Its response parameter is "int8 result", which returns a 0 if the object has been picked up or a 1 otherwise, stored as an 8-bit int.

Appendix B

User manual

The Debris Collector core delivered with this paper is ready for compilation and execution. As it is not a complete robot the installation feature of ROS has not been yet configured, the package is still in development state.

B.1 Preparing the environment

To execute the Debris Collector a ROS environment has to be installed. There a a number of installation manuals for different operating systems in http://www.ros.org/wiki/ROS/Installation .
Once ROS is installed in the computer a catkin workspace has to be created. To do this execute the following script:

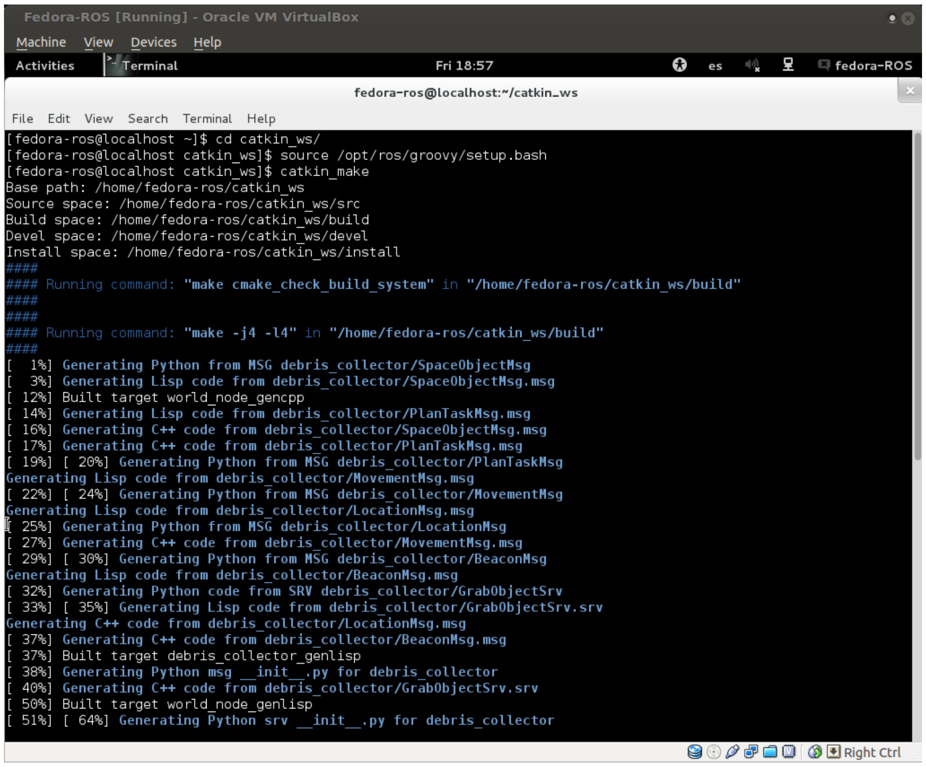
```
    $ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Then copy the extract the zip files in catkin_ws/src. Once the debris_collector folder and its subfolders have been copied the environment is ready.

B.2 Building the Debris Collector

To build the Debris Collector the following script should be executed (in the same terminal than the previous script):

```
    $ cd ..
$ catkin_make
```


Fedora-ROS [Running] - Oracle VM VirtualBox
Machine View Devices Help
Activities Terminal Fri 18:57 es fedora-ROS
fedora-ros@localhost:~/catkin_ws
File Edit View Search Terminal Help
Right Ctrl

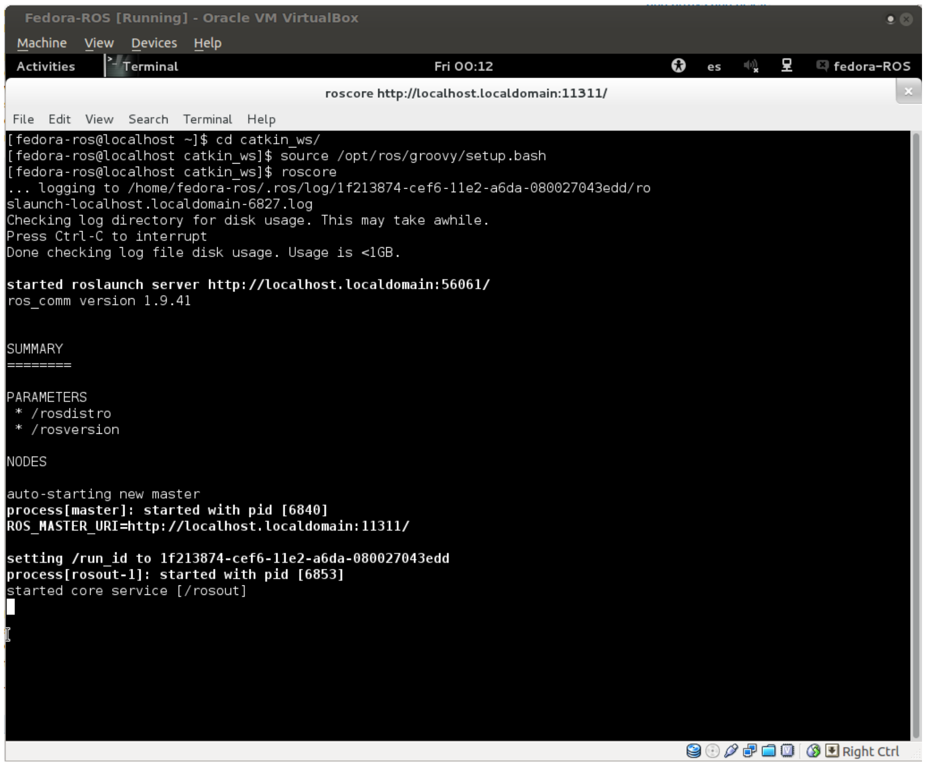
```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ catkin_make
Base path: /home/fedora-ros/catkin_ws
Source space: /home/fedora-ros/catkin_ws/src
Build space: /home/fedora-ros/catkin_ws/build
Devel space: /home/fedora-ros/catkin_ws/devel
Install space: /home/fedora-ros/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/fedora-ros/catkin_ws/build"
####
####
#### Running command: "make -j4 -l4" in "/home/fedora-ros/catkin_ws/build"
####
[  1%] Generating Python from MSG debris_collector/SpaceObjectMsg
[  3%] Generating Lisp code from debris_collector/SpaceObjectMsg.msg
[ 12%] Built target world_node_gencpp
[ 14%] Generating Lisp code from debris_collector/PlanTaskMsg.msg
[ 16%] Generating C++ code from debris_collector/SpaceObjectMsg.msg
[ 17%] Generating C++ code from debris_collector/PlanTaskMsg.msg
[ 19%] [ 20%] Generating Python from MSG debris_collector/PlanTaskMsg
Generating Lisp code from debris_collector/MovementMsg.msg
[ 22%] [ 24%] Generating Python from MSG debris_collector/MovementMsg
Generating Lisp code from debris_collector/LocationMsg.msg
[ 25%] Generating Python from MSG debris_collector/LocationMsg
[ 27%] Generating C++ code from debris_collector/MovementMsg.msg
[ 29%] [ 30%] Generating Python from MSG debris_collector/BeaconMsg
Generating Lisp code from debris_collector/BeaconMsg.msg
[ 32%] Generating Python code from SRV debris_collector/GrabObjectSrv
[ 33%] [ 35%] Generating Lisp code from debris_collector/GrabObjectSrv.srv
Generating C++ code from debris_collector/LocationMsg.msg
[ 37%] Generating C++ code from debris_collector/BeaconMsg.msg
[ 37%] Built target debris_collector_genlisp
[ 38%] Generating Python msg __init__.py for debris_collector
[ 40%] Generating C++ code from debris_collector/GrabObjectSrv.srv
[ 50%] Built target world_node_genlisp
[ 51%] [ 64%] Generating Python srv __init__.py for debris_collector
```

Figure B.1: The Debris Collector being built.

B.3 Executing the Debris Collector

To execute the robot first an instance of roscore is needed. To launch it, in the same terminal than before, run:

```
$ roscore
```


Fedora-ROS [Running] - Oracle VM VirtualBox
Machine View Devices Help
Activities Terminal Fri 00:12 es fedora-ROS
roscore http://localhost.localdomain:11311/
File Edit View Search Terminal Help
Right Ctrl

```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ roscore
... logging to /home/fedora-ros/.ros/log/1f213874-cef6-11e2-a6da-080027043edd/ro
slaunch-localhost.localdomain-6827.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost.localdomain:56061/
ros_comm version 1.9.41


SUMMARY
========

PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
process[master]: started with pid [6840]
ROS_MASTER_URI=http://localhost.localdomain:11311/

setting /run_id to 1f213874-cef6-11e2-a6da-080027043edd
process[rosout-1]: started with pid [6853]
started core service [/rosout]
```

Figure B.2: The roscore once initialized.

After the roscore is running every node should be launched **in a different terminal**. In all terminals before executing anything else the setup files should be sourced. Assuming ROS is installed in opt the following script should be executed:

```
$ cd ~/catkin_ws
$ source /opt/ros/groovy/setup.bash
$ source ./devel/setup.bash
```

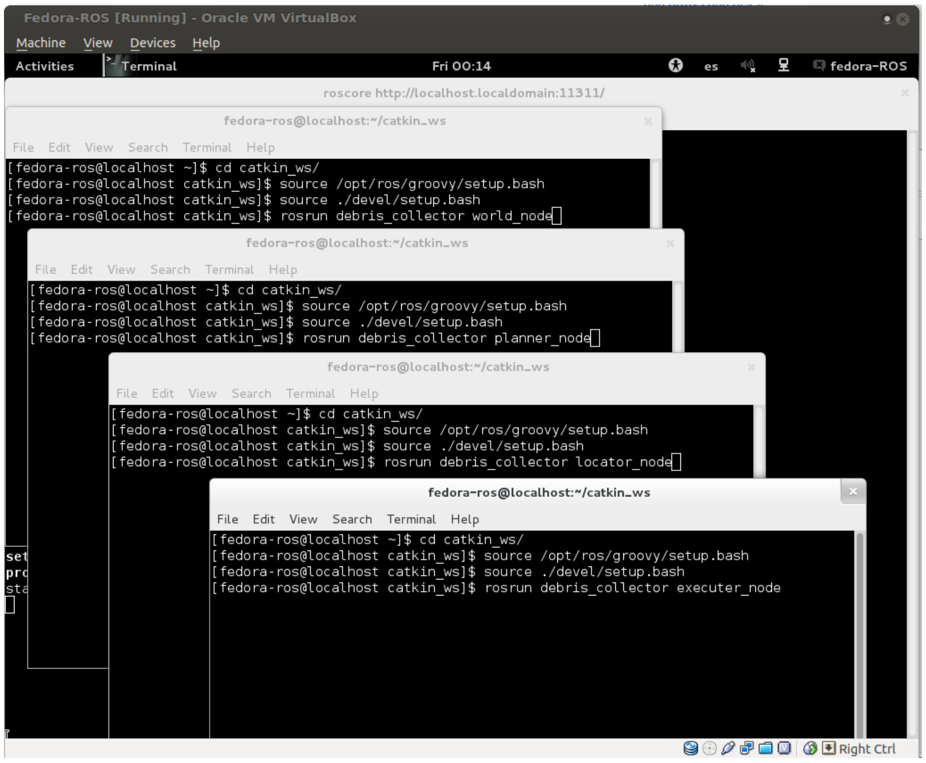
This allows ROS to be able to find the packages and nodes. Then, in a different terminal each run:

```
$ rosrun debris_collector executer_node
```

```
$ rosrun debris_collector planner_node
$ rosrun debris_collector locator_node
```

```
$ rosrun debris_collector world_node
```

Those scripts may be run in any order, but it is strongly recommended that the world node is last one to be launched. The other nodes will wait for the world node to be active before doing any work.


Fedora-ROS [Running] - Oracle VM VirtualBox
Machine View Devices Help
Activities Terminal Fri 00:14 es fedora-ROS
roscore http://localhost.localdomain:11311/
fedora-ros@localhost:~/catkin_ws
File Edit View Search Terminal Help
Right Ctrl

```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ source ./devel/setup.bash
[fedora-ros@localhost catkin_ws]$ rosrun debris_collector world_node
```

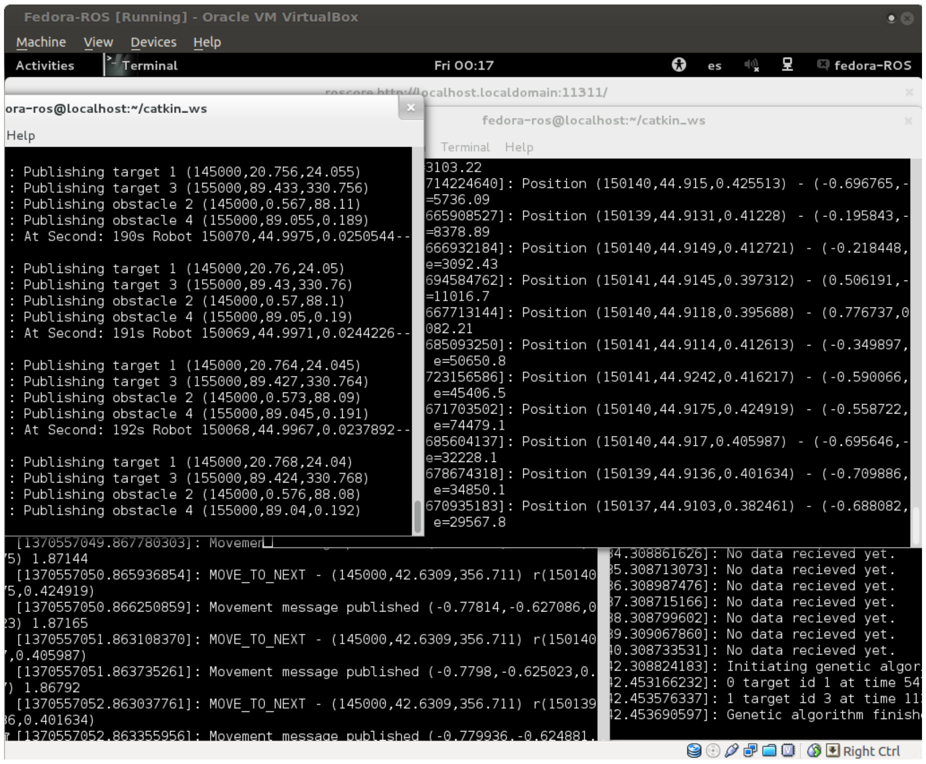
```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ source ./devel/setup.bash
[fedora-ros@localhost catkin_ws]$ rosrun debris_collector planner_node
```

```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ source ./devel/setup.bash
[fedora-ros@localhost catkin_ws]$ rosrun debris_collector locator_node
```

```
[fedora-ros@localhost ~]$ cd catkin_ws/
[fedora-ros@localhost catkin_ws]$ source /opt/ros/groovy/setup.bash
[fedora-ros@localhost catkin_ws]$ source ./devel/setup.bash
[fedora-ros@localhost catkin_ws]$ rosrun debris_collector executer_node
```

Figure B.3: The four nodes ready to be executed.

If everything has been done right there should be five open terminals, one of them with the roscore and the other four with one of the nodes each.
Once the Nodes have been launched they will start printing their log information on their corresponding terminal.


Fedora-ROS [Running] - Oracle VM VirtualBox
Machine
View
Devices
Help
Activities
Terminal
Fri 00:17
es
fedora-ROS
roscore http://localhost.localdomain:11311/
ora-ros@localhost:~/catkin_ws
fedora-ros@localhost:~/catkin_ws
Terminal
Help
Right Ctrl

```
: Publishing target 1 (145000,20.756,24.055)
: Publishing target 3 (155000,89.433,330.756)
: Publishing obstacle 2 (145000,0.567,88.11)
: Publishing obstacle 4 (155000,89.055,0.189)
: At Second: 190s Robot 150070,44.9975,0.0250544--

: Publishing target 1 (145000,20.76,24.05)
: Publishing target 3 (155000,89.43,330.76)
: Publishing obstacle 2 (145000,0.57,88.1)
: Publishing obstacle 4 (155000,89.05,0.19)
: At Second: 191s Robot 150069,44.9971,0.0244226--

: Publishing target 1 (145000,20.764,24.045)
: Publishing target 3 (155000,89.427,330.764)
: Publishing obstacle 2 (145000,0.573,88.09)
: Publishing obstacle 4 (155000,89.045,0.191)
: At Second: 192s Robot 150068,44.9967,0.0237892--

: Publishing target 1 (145000,20.768,24.04)
: Publishing target 3 (155000,89.424,330.768)
: Publishing obstacle 2 (145000,0.576,88.08)
: Publishing obstacle 4 (155000,89.04,0.192)
```

```
3103.22
714224640]: Position (150140,44.915,0.425513) - (-0.696765,-
=5736.09
665908527]: Position (150139,44.9131,0.41228) - (-0.195843,-
=8378.89
666932184]: Position (150140,44.9149,0.412721) - (-0.218448,
e=3092.43
694584762]: Position (150141,44.9145,0.397312) - (0.506191,-
=11016.7
667713144]: Position (150140,44.9118,0.395688) - (0.776737,0
082.21
685093250]: Position (150141,44.9114,0.412613) - (-0.349897,
 e=50650.8
723156586]: Position (150141,44.9242,0.416217) - (-0.590066,
 e=45406.5
671703502]: Position (150140,44.9175,0.424919) - (-0.558722,
 e=74479.1
685604137]: Position (150140,44.917,0.405987) - (-0.695646,-
e=32228.1
678674318]: Position (150139,44.9136,0.401634) - (-0.709886,
 e=34850.1
670935183]: Position (150137,44.9103,0.382461) - (-0.688082,
 e=29567.8
```

```
 [1370557049.867780303]: Movemer
75) 1.87144
 [1370557050.865936854]: MOVE_TO_NEXT - (145000,42.6309,356.711) r(150140
75,0.424919)
 [1370557050.866250859]: Movement message published (-0.77814,-0.627086,0
23) 1.87165
 [1370557051.863108370]: MOVE_TO_NEXT - (145000,42.6309,356.711) r(150140
7,0.405987)
 [1370557051.863735261]: Movement message published (-0.7798,-0.625023,0.
7) 1.86792
 [1370557052.863037761]: MOVE_TO_NEXT - (145000,42.6309,356.711) r(150139
36,0.401634)
 [1370557052.863355956]: Movement message published (-0.779936,-0.624881,
```

```
34.308861626]: No data recieved yet.
35.308713073]: No data recieved yet.
36.308987476]: No data recieved yet.
37.308715166]: No data recieved yet.
38.308799602]: No data recieved yet.
39.309067860]: No data recieved yet.
40.308733531]: No data recieved yet.
42.308824183]: Initiating genetic algor
42.453166232]: 0 target id 1 at time 54
42.453576337]: 1 target id 3 at time 11
42.453690597]: Genetic algorithm finish
```

Figure B.4: The four nodes running.