POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Energy consumption-based path planning for legged rovers

Tesi di Laurea Magistrale in
Space Engineering - Ingegneria Spaziale

Author: **Marco Elia**

Student ID: 940537
Advisor: Prof. Mauro Massari
Academic Year: 2020-21

Abstract

In recent years, legged rovers have become more and more relevant in the space applications. Whether the deployed rovers, are still wheeled ones, their limitations make clear, that rovers capable to operate on uneven and harsh terrains, are needed. For this reason, more and more researches are performed to address all the aspects of complex legged robots.
Nowadays, one of the main problems of legged robots is their high power consumption, that limits their autonomy. For this reason, multiple researches have been conducted to optimize the energy consumption, evaluating different gaits, leg trajectories, actuators and more.
This work aims to expand this branch of research on legged rovers, trying to optimize the path of the rover on the ground, to reduce the energy consumption needed to reach a target position. In particular, it aims to determine a cost function, that allows path planning algorithms to evaluate paths, not based on geometric distances and accessibility maps, but on the energy required to follow the path.
To test and validate the cost function, some path planning algorithms deriving from the RRT* are developed, to fit all the cost function and legged robot needs. Furthermore, a benchmark robot model, with mechanical parts and controllers, is built on Simulink, to test the quality of the paths generated.
The results shows, that the algorithms are able to find a path and optimize the energy consumption, yet several improvements are possible, and suggested.

Keywords: path planning, legged robot, RRT, RRT*, RRCT*.

Abstract in lingua italiana

Negli ultimi anni, i rover a zampe sono diventati sempre più rilevanti nel settore spaziale. Se i rover utilizzati sul campo, sono ancora solo robot a ruote, le loro limitazioni rendono chiaro, che robot capaci di operare su terreni sconnessi, sono necessari. Per questo motivo, sempre più ricerche vengono svolte per valutare i molteplici aspetti dei robot a zampe.
Al giorno d'oggi, uno dei maggiori problemi per i robot a zampe, è l'elevato consumo di energia, che ne limita l'autonomia. Per questo, molte ricerche sono state svolte per ottimizzare il consumo di energia, valutando differenti andature, traiettorie delle zampe, attuatori e molto altro.
Questo lavoro mira ad espandere questo ramo della ricerca sui robot a zampe, provando ad ottimizzare il percorso di un robot sul terreno, per ridurre il consumo di energia necessario per raggiungere la destinazione. In particolare, vuole determinare una funzione di costo, che consenta agli algoritmi di pianificazione del percorso di valutare i percorsi, non basandosi su distanze geometriche e mappe di accessibilità, ma sull'energia necessaria per seguire un percorso.
Per testare e validare la funzione di costo, sono stati sviluppati degli algoritmi di pianificazione del percorso derivanti dal RRT, per adattarsi alle necessità della funzione di costo e dei robot a zampe. Inoltre, è stato creato un modello di un robot di prova su Simulink, simulando parti meccaniche e anche i controllori, per testare la qualità del percorso generato.
I risultati mostrano, che l'algoritmo è capace di ottimizzare il consumo di energia, ma parecchi miglioramenti sono possibili e suggeriti.

Parole chiave: pianificazione del percorso, robot a zampe,RRT, RRT*, RRCT*.

Contents

1 | Introduction

"The time will come when man will know even what is going on in the other planets and perhaps be able to visit them." HENRY FORD, Theosophist Magazine, February 1930. Even before being able to leave the Earth, humanity has always been curious about its surroundings, trying to observe worlds far away, and using imagination where the sight could not reach. Eventually, we broke the barrier above the clouds, getting to space and moving closer and closer to other celestial bodies, first observing from afar, and then, even landing on them. Up to now we have "flown by, orbited, smacked into, radar examined, and rocketed onto, as well as bounced upon, rolled over, shoveled, drilled into, baked and even blasted. Still to come: 'stepped on.' " BUZZ ALDRIN.

1.1. Background

The rovers sent in space up to now, are wheeled rovers, those are relatively simple, but very limited. Wheels are made for roads and even terrain, and exploration is not performed on roads. Thus rovers capable to traverse rough terrain are being developed in the shape of legged rovers, that can lift their leg over rocks, cracks, obstacles; and compress a much smaller terrain area under their feet, increasing theoretical efficiency and reducing the impact on the terrain[1]. Legged robots have been object of studies since the beginning of robotic, but due to their high complexity, the wheeled ones have always been preferred in real applications. With the development of computing hardware, control theories, algorithms; and the increasing requirement for applications in high difficulty terrain, legged robots are becoming an interesting option[1]. In this environment multiple aspect related to legged locomotion have been researched, one of them being the path planning.

1.2. State of the art

Multiples studies have been conducted about robots locomotion, trying to address different aspects: leg motion planning, gait planning, path planning; first evaluating feasibility,

then optimizing energy consumption and other parameters.
Legged robots have an high number of degrees of freedom, multiple state sequences will be suitable to get a feasible motion between initial and final states, being extremely power thirsty, legged robots need to use remaining degrees of freedom to reduce power consumption. Studies to estimate the cheapest single leg trajectory have been performed in [2], while postures efficiencies have been compared in [3]. (Note: in section 1.1, legged robots have been described as "more theoretically efficient"; while, theoretically, legged locomotion is more efficient than wheeled locomotion, since wheels spend energy to deform long stripes of terrain, and legged ones just deform small areas; in real applications there are not motor as efficient as muscles and so, in practice, legged robot are more energy demanding [1]).
Motion cost can be greatly affected by the terrain, various studies can estimate the best gait and leg parameters from terrain stereo observations like [4]. Those methods work on terrain that have been studied and classified, but often the terrain being explored can be unknown and maybe different from everything tested before. In these situations algorithms that allow the robot to estimate the gait motion with optimal consumption, while walking on the terrain [5, 6] can be useful.
About the path planning, several studies have been conducted to obtain a feasible path, generally minimizing the distance: starting from a 2D heightmap [7] estimate a coarse path using an A* method, refining the trajectory and defining all other leg parameters by running an RRT-based footstep planner, while [8] uses a D* Lite planner. In [9] laser scanner and stereo vision systems have been used to autonomously drive a big robot in a very complex terrain as a forest.

Both on Earth and in space wheeled robots are still more developed and used, not only they have less degrees of freedom, reducing the need to optimize multiple motion parameters, but on-mission experience allows research to be much deeper. Several path planning algorithms for wheeled robots exist, used in domestic robots, industrial robots, cars etc...
Of particular interest in this work are the wheeled space rovers, that can provide an on-field expertise that cannot be matched by theoretical research and controlled experiments on legged robots.
Curiosity and Mars 2020 require an high degrees of autonomy for long-distance autonomous traverse, to cope with this requirement local path planning must find a feasible path to move in a direction and ensure the safety of this path. To do so, an inverse-kinematics problem with iterative nonlinear optimization under geometric constraint must be performed for multiple candidate paths. This computation is unfeasible for the limited

performance of the rover computer, thus [10, 11] proposed ACE (Approximate Clearance Evaluation). ACE does not consider the entire kinematic of the rover, but just upper and lower boundaries, obtaining feasibility boxes that can be used to ensure, in a conservative way, that the robot limits are not exceeded, obtaining much faster, yet reliable and safe, computations.
To be more accurate, ENav (Enhanced autoNav), generates a list of paths, shaped like arcs of circumference, it preliminary sort them to rank them in order of expected feasibility, and then runs ACE to ensure their feasibility. Often, ENav preliminary estimation feeds to ACE paths that it will evaluate as unfeasible, thus forcing ENav to rank again the paths and call ACE until a solution is found. Despite being cheaper than full kinematic ACE is still computationally expensive, therefore [12] proposed two alternatives to quickly and more reliably rank the paths, one using Sobel operators and convolution, the other using machine learning to predict areas that will be deemed untraversable by ACE. Other studies like [13] have added onto a feasibility evaluation, also a terrain slippage evaluation, to better estimate the traversability on different terrains.

1.3. Scope of the thesis

As seen before, several studies have been performed for optimizing energy consumption of leg and gait motion, and many addressed the path planning accounting for path feasibility, but there is a lack of studies concerning path planning of legged robots estimating and minimizing energy consumption. In particular, path planning for legged and wheeled robots, is performed through a graph technique, where some robot states are connected to each other. This theory is flexible and can be applied to any robot, if can be provided an adequate way to estimate the cost to pass from a state to another. This thesis objective is to develop a cost function, that can estimate this cost, based on the energy consumption of a legged robot. In this way, existing literature on wheeled robots path planning, could be extended to legged robots. Furthermore, hybrid legged-wheeled robots will be able to use the same path planning algorithm in wheel and leg mode, by just changing the cost function.

1.4. Structure of the thesis

After seeing the introduction of the thesis in this chapter, in chapter 2 basic concepts on path planning and legged robots are presented, concluding the "established knowledge" introduction. The main work will start in chapter 3 where the cost function is discussed, and in chapter 4 some variants of path planning algorithm are presented. To validate the

study, a simulation environment has been set up in chapter 5 and the result for a case study rover are presented in chapter 6. The work results and future work are summarized in chapter 7.

2 | Fundamentals

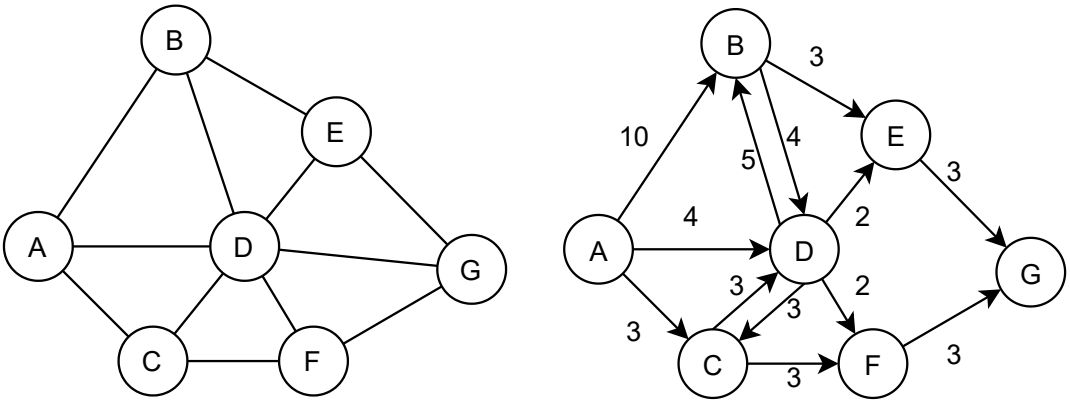
To ensure for any reader to have some foundations before discussing the research performed, and to make clear the terminology used in the following chapters, here are presented some fundamentals of path planning and legged robots. Note that all the topics are introduced just for what is strictly necessary for this thesis discussion, this is not meant to be an extensive introduction.

2.1. Path planning fundamentals

Path planning is often performed though graphs, therefore graphs are introduced.

2.1.1. Graphs

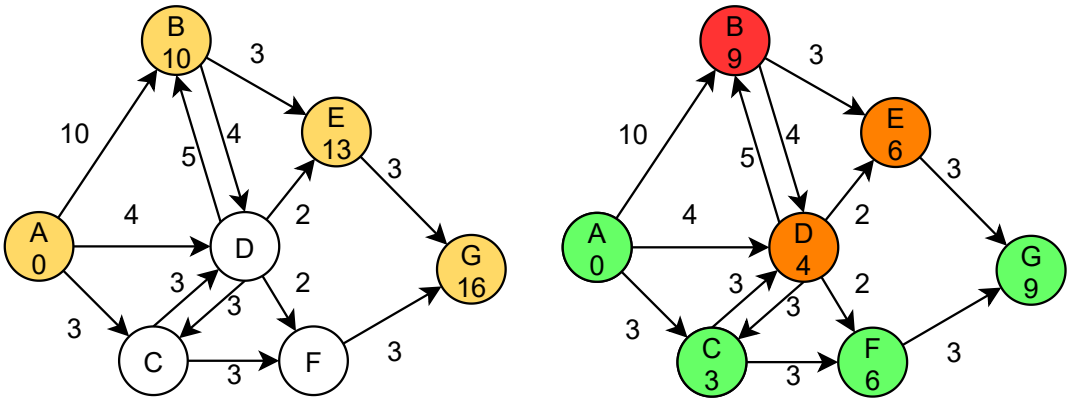
Graphs are very useful structures, used in (but not limited to) path planning. Are composed of a set of points (vertex or nodes) connected each other through edges fig. 2.1a. Those edges express a relation between the nodes, for path planning purpose this relation is usually a "path" between two nodes. Considering edges as paths, can be easily seen that they can be directional and weighted fig. 2.1b, where the direction indicates in which direction the motion is allowed and the weight the cost to move between the nodes along the edge.



(a) Graph example. (b) Directed and weighted graph example.

Figure 2.1: Graph examples.

Is clear that, in the examples, is possible to move between any vertex in the graph, passing through other vertices if necessary. The final cost to move between the vertices, is the sum of the edges weights. Assuming to define an initial point A and a target point G, for each vertex a cost to reach from the initial point can be defined, and can be computed as the cost to reach the previous point from the initial point plus the edge cost between this vertex and the previous fig. 2.2a. Is easy to note, that vertices can be reached from different vertices and paths with different cost. In these cases, the vertex retains the cheapest cost and is considered reached with the cheapest path fig. 2.2b.



(a) Path example. (b) Cheapest path example.

Figure 2.2: Paths examples.

A graph where each vertex has just one parent, is called a **tree** fig. 2.3 and is in general much cheaper to access.

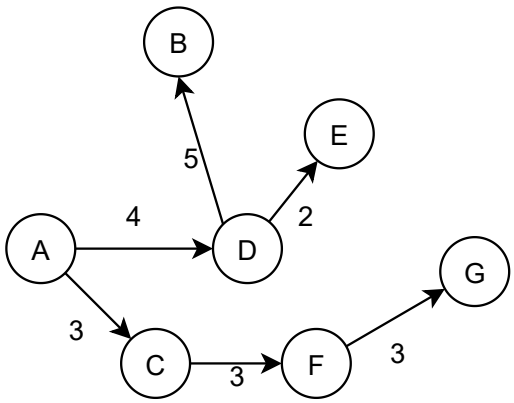


Figure 2.3: Tree example.

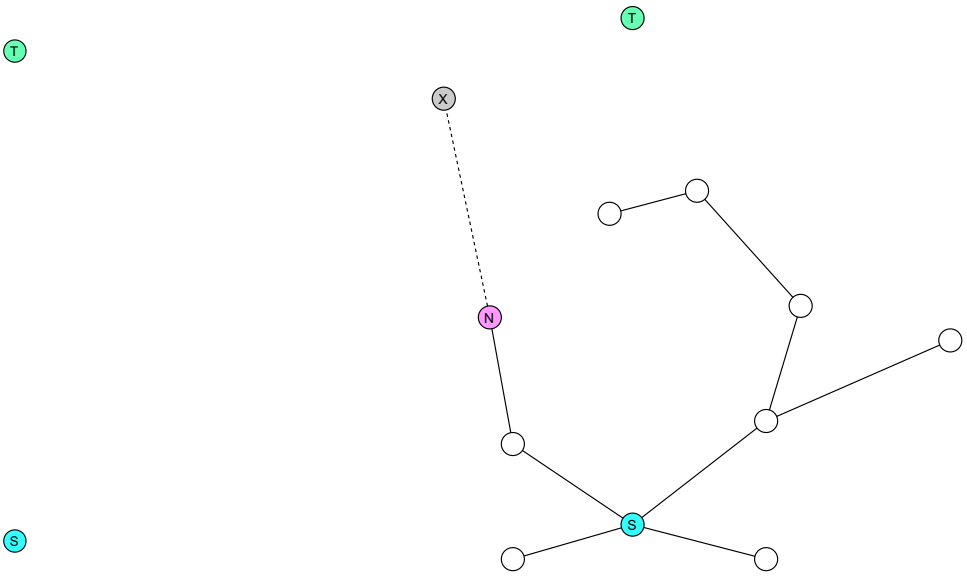
Exist different techniques to generate and search in graphs, but they will not be discussed here. In path planning, graphs are generally generated over a terrain splitting the terrain in cells and placing vertices systematically in cells (Search-based algorithms), or by placing them randomly (Sampling-based algorithms)[14]. For the scope of this thesis, just the sampling-based algorithms and in particular the RRT family will be introduced.

Note that in path planning, vertices can be any type of state, not necessarily points in a 2D or 3D space, therefore, while here they will be introduced assuming states as (x,y) positions, in general they are not limited to it.

2.1.2. RRT

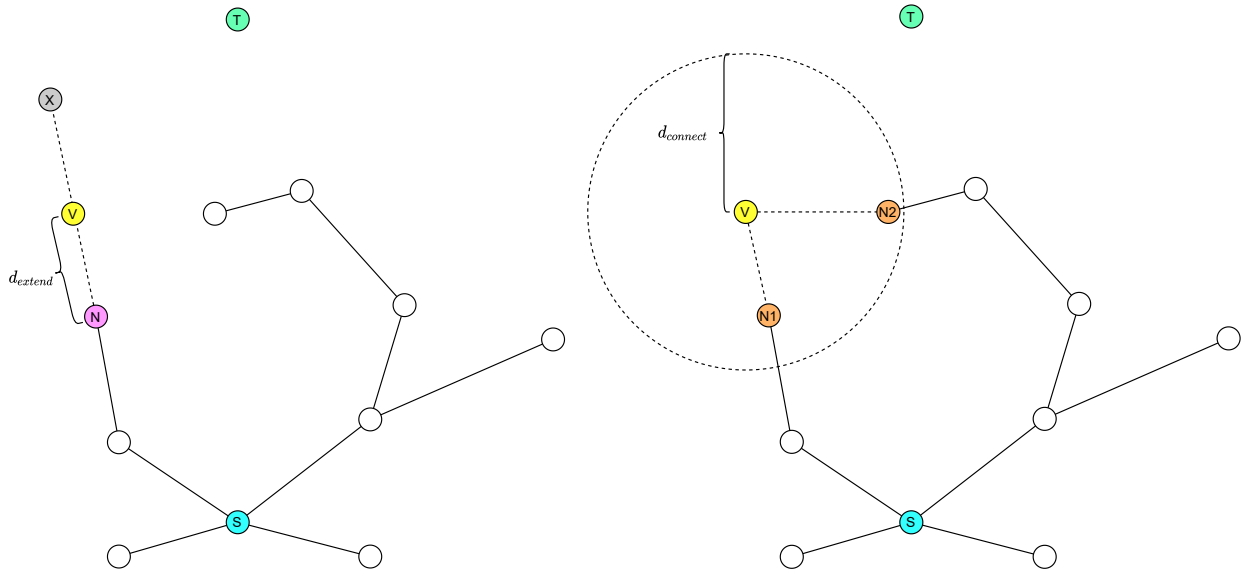
RRT (Rapidly-exploring Random Trees) is a sampling-based algorithm, where the nodes positions are determined randomly.
It starts defining a starting point S and a target point T (or a target area) (fig. 2.4a), the objective is to build a path from S to T. To do so, at any generic iteration, the algorithm works in this way:

1. A random point X in the search space is selected (fig. 2.4b).
2. The closest node N to X is identified and N is extended to X. In other words, a new vertex V is placed on the line that connect N to X, at a distance from N that can be called **Extension Distance** d_{extend} (fig. 2.4c). (If the XN distance is smaller than d_{extend}, the new vertex V is placed in X.)



(a) Starting problem. (b) Find closest point.

Figure 2.4: RRT algorithm pt1.



(c) Add new vertex. (d) Find neighbour vertices and evaluate connection edges.

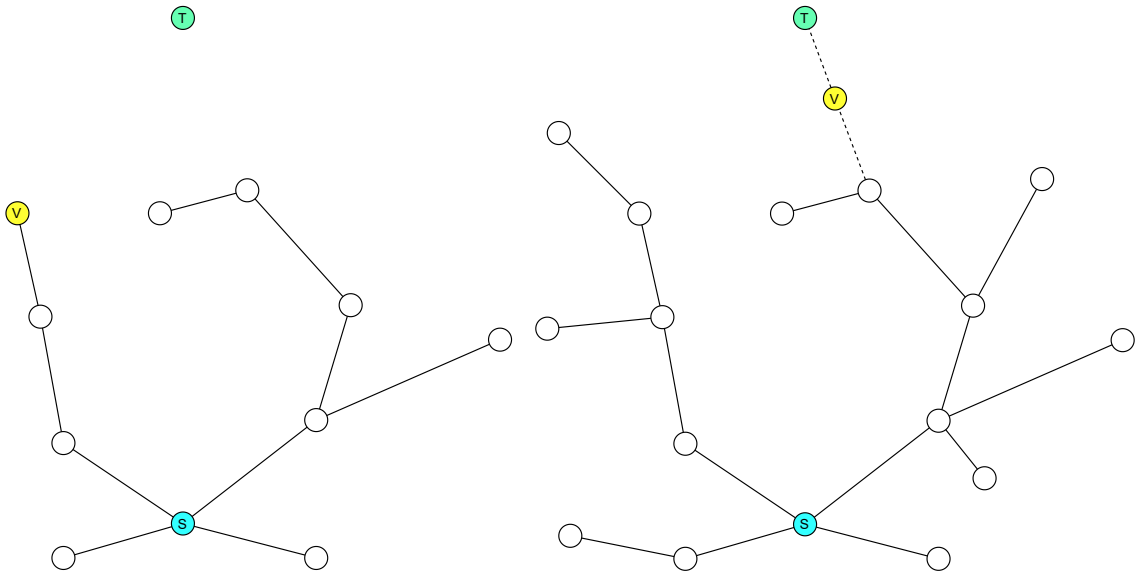
Figure 2.4: RRT algorithm pt2.

3. V is then connected as child to the cheapest neighbouring vertex. To do so all the vertex Nx, that have a distance from V smaller than a **Connection Distance** $d_{connect}$, are selected (fig. 2.4d), and all Nx are connected to V with edges. A cost function is used to compute the weight and the feasibility of each edge, then the cost

to reach V from S can be computed for each Nx. The cheapest one is maintained, while all the others are pruned away (fig. 2.4e).

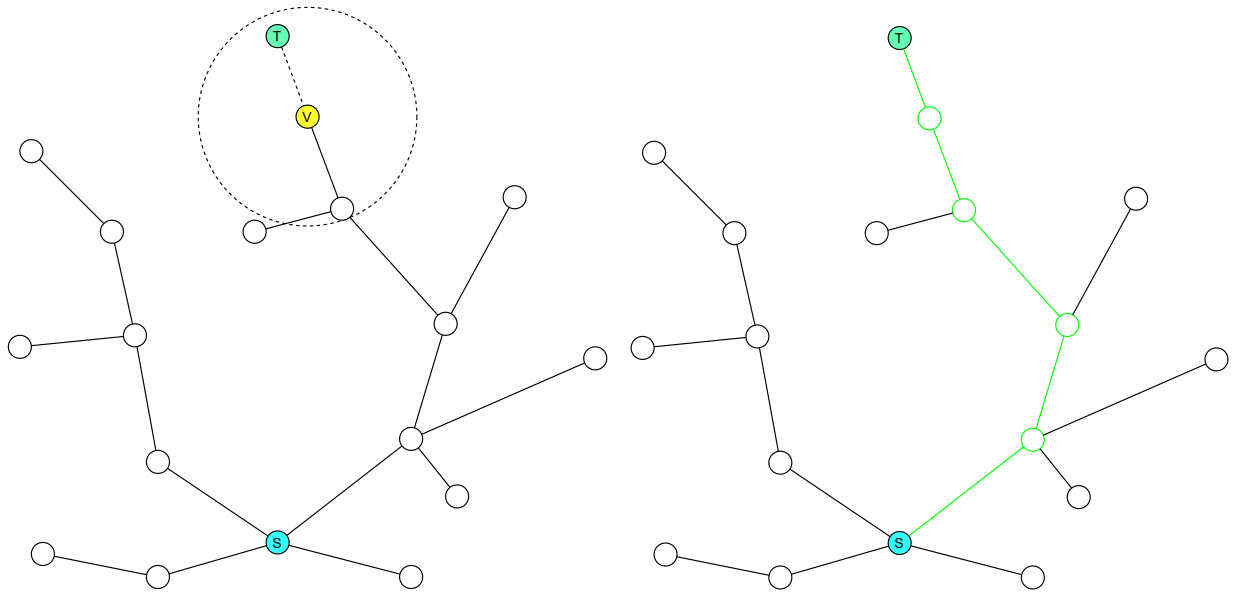
4. If the VT distance is smaller than $d_{connect}$, an attempt to connect T with V is performed (fig. 2.4g), if is successful, a feasible path is obtained (fig. 2.4h).

This procedure is iterated until a vertex can directly be connected to the target. Sometimes the random point X can be chosen as positioned in the target T (fig. 2.4f), to force the planner to explore in the direction of the target, instead of in a random direction. The ratio of iteration where X is selected as T over the total number of iterations, is called **Exploration Bias** [15, 14, 16].



(e) Connect new vertex to the tree. (f) X selected as target point.

Figure 2.4: RRT algorithm pt3.



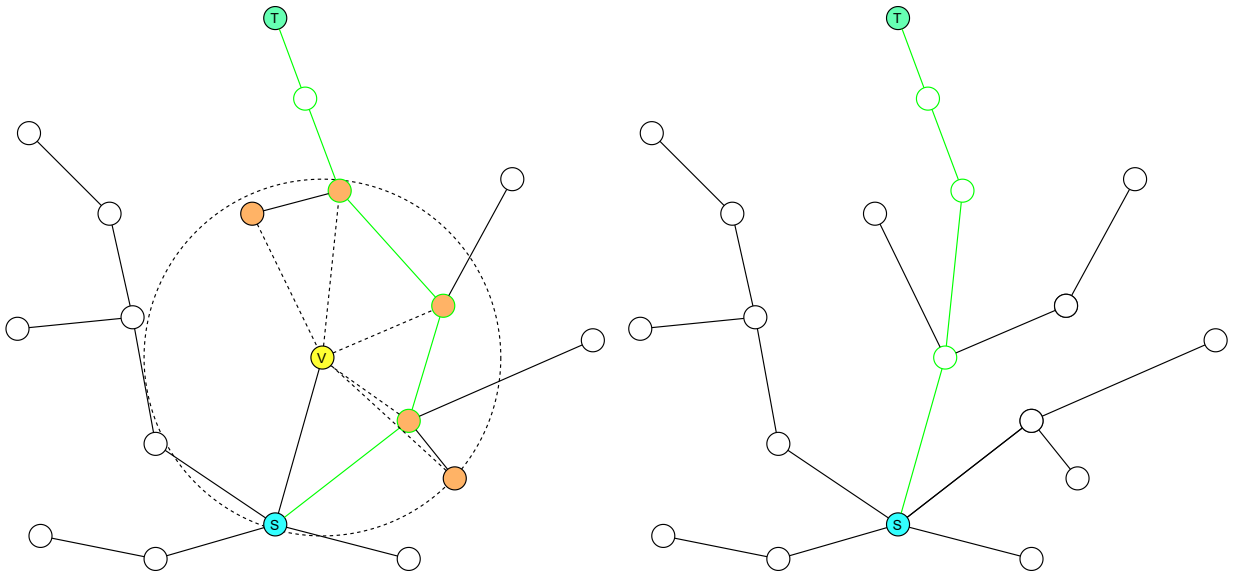
(g) Connect to target. (h) Path found.

Figure 2.4: RRT algorithm pt4.

As can be seen in fig. 2.4h a path can be found, but is not optimal. In particular, in this example, if we assume that there are no obstacles and that the cost function is the geometric distance, the optimal solution would be a straight like from S to T, while the path found takes a longer route. The algorithm can keep running after a solution is found, possibly finding a better solution, but is very inefficient in finding an optimal solution, therefore more advanced algorithms are required.

2.1.3. RRT*

RRT* is an evolution of the RRT where a rerouting step is added.
The iterations are the same as for RRT, but a step is appended at the end. After the new vertex has been added to the tree (fig. 2.5a), all vertices at distance smaller than $d_{connect}$ from the vertex, are considered as possible children of the vertex; evaluating the edges as in step 3 of the RRT method, but in this case moving from V to the neighbouring vertices, the cost to reach them from the start passing from V can be computed, if this cost is cheaper than the previous one, V become the new parent and consequently the path is rerouted (fig. 2.5b). All variation happening in this area, must be propagated to the descendant vertices.



(a) Find neighbour vertices and evaluate connection edges.

(b) Tree and path updated.

Figure 2.5: RRT* algorithm, rerouting step.

As can be clearly noted in fig. 2.5b, the RRT* can optimize the path and can be shown that RRT* guarantees asymptotic optimality [17]. While RRT* can guarantee cheaper and more reasonable paths, the numbers of calls to the cost function is highly increased, so RRT* has generally slower iterations respect to RRT [17, 14, 16].

2.1.4. Dubins-curve based RRT

As shown until now, the base RRT is used for a point moving on a plane, and connects two positions through a segment. For some robots this can be a possible way of generating a feasible path, this is the case if the robots are able to turn on spot, but some robots, as car-like robots, cannot do that. They have a limit minimum radius of curvature, they can approach while steering. Those mobile robots usually have less action variables then degrees of freedom. This kind of robots is called non-holonomic, or underactuated[18].
To account for this behaviour, some modified RRT algorithms have been developed.
First of all, the explored state space is the Dubin state space, accounting for (x,y) position and (ϑ) orientation. Any state $\underline{\mathbf{q}} = [x, y, \vartheta]$ can be connected to another state trough a curve, imposing a minimum curvature radius to the curve; can be demonstrated, that the geometrically shortest path connecting the two points, is composed just by straight lines and circumference arcs, having radius equal to the minimum curvature radius [18]. The

resulting connection curve, is called a Dubin curve (fig. 2.6).

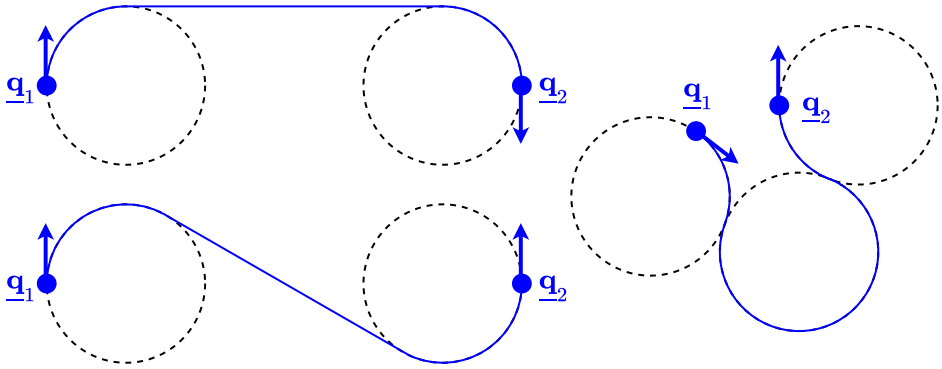


Figure 2.6: Examples of three doubin curves.

These Dubin curves can be used to connect two states, and so two vertices, in the RRT algorithm. Or alternatively can be used to connect a Dubin state $\underline{\mathbf{q}} = [x, y, \vartheta]$ to a position $\underline{\mathbf{p}} = [x, y]$, generally exiting from the $\underline{\mathbf{q}}$ with an arc of circumference having the minimum curvature radius, and entering $\underline{\mathbf{p}}$ with a straight line, then assigning to $\underline{\mathbf{p}}$ an orientation based on the arrival direction of the straight line, thus making it a Dubin state [18].

Must be noted that this algorithm adds a degree of freedom to explore ϑ, and performs all the curves with the minimum possible curvature radius.

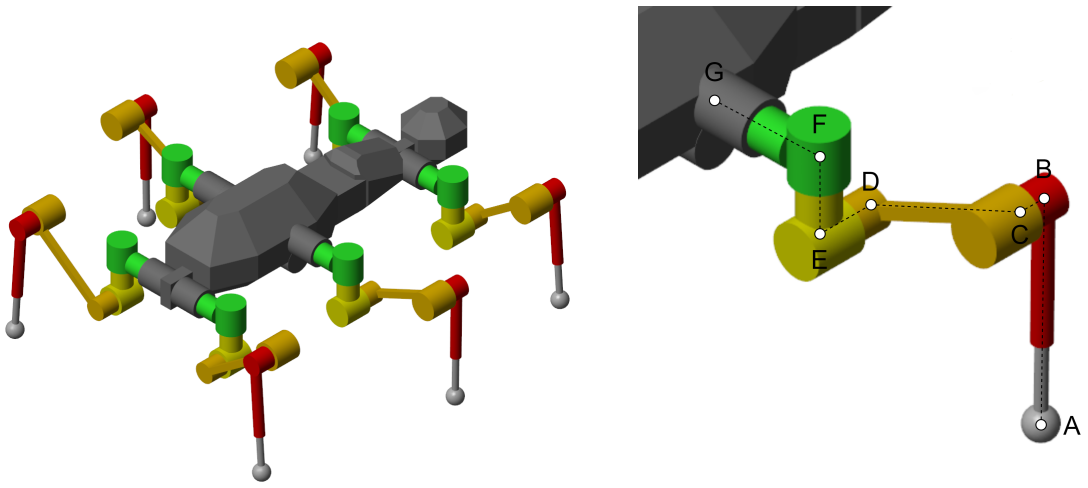
2.2. Legged robots fundamentals

The most common way in nature to move on ground is legged locomotion. As several species moves using legs, as many architectures are present in nature, and consequently in robotic.

2.2.1. Legs

For the vehicle to move in a kinematically correct way on an arbitrary trajectory on uneven ground, each foot must have, independently from the leg configuration chosen, a minimum of three degrees of freedom [1]. Three is the minimum, but often in nature more are present. For example, humans have one fundamental degree of freedom in the knee and two in the hip, while an additional one is in the ankle and more in foot fingers. These additional degrees of freedom helps to optimize the contact to the ground. In the rover used as benchmark, based on SpaceClimber [19], there are five joints, as can be seen in fig. 2.7b. The tree fundamental joints are between B and C (Distal), D and E

(Basal), E and F (Thorax), while an additional joint is added between F and G (lean) and a prismatic passive joint (a spring) is between A and B.



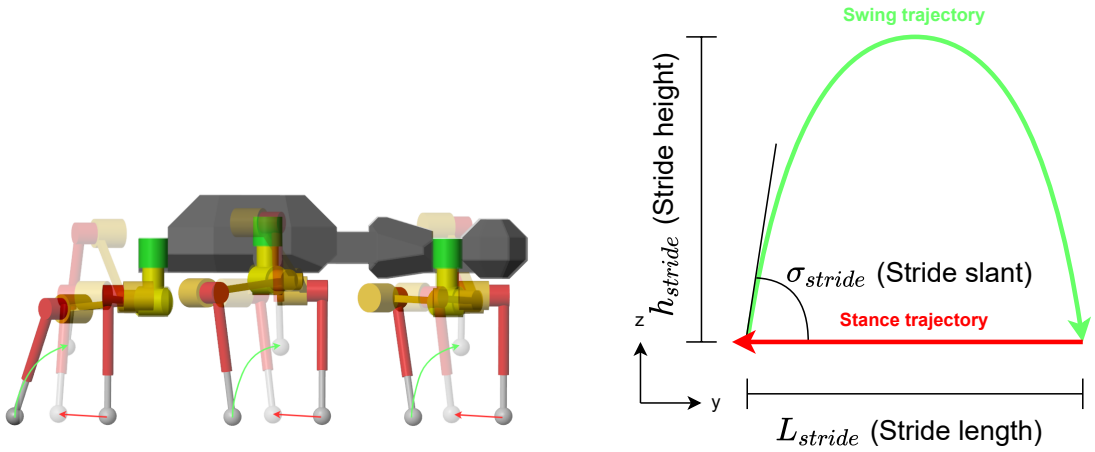
(a) Robot isometric view from behind. (b) Robot leg detail.

Figure 2.7: Rover scheme: each color represent a rigid body, therefore at every color discontinuity there is a joint.

Leg motions is generally split in two phases. The first one is the **Stance Phase** or **Support Phase**, where the leg is in contact with the ground, sustaining the weight of the body and standing still, while moving the body forward. Conceptually, after the leg has reached it maximum extension in stance phase, the **Swing Phase** or **Return Phase** is triggered, where the leg is lifted and moved forward, to then start a new stance phase (fig. 2.8a).
For each phase can be defined a duration: **Swing Time** (t_{swing}) and **Stance Time** (t_{stance}), while the time for the leg to perform a swing plus stance cycle is called **Cycle Time** (t_{cycle}). The **Duty Factor** β can be defined:

$$\beta = \frac{t_{stance}}{t_{cycle}} = \frac{t_{stance}}{t_{stance} + t_{swing}} \tag{2.1}$$

Defining a reference frame fixed on robot body, where y is parallel to the motion direction, z parallel and opposite to the gravity vector and x is to form a right-hand orthogonal frame, a side view of the foot trajectory in this frame can be defined as in fig. 2.8b. During the swing phase, the trajectory can have any feasible shape: triangle, rectangle, parabola, n-grade polynomial etc..., while the stance trajectory will be determined by the desired body trajectory, in the simple case of a forward motion, the foot will move backward in a straight line relatively to the body frame. Some parameters can thus be defined: **Stride**



(a) Side view of robot motion during tripod gait, showing legs in swing and stance phase.

(b) Foot trajectory side view.

Figure 2.8: Legs and foot motion.

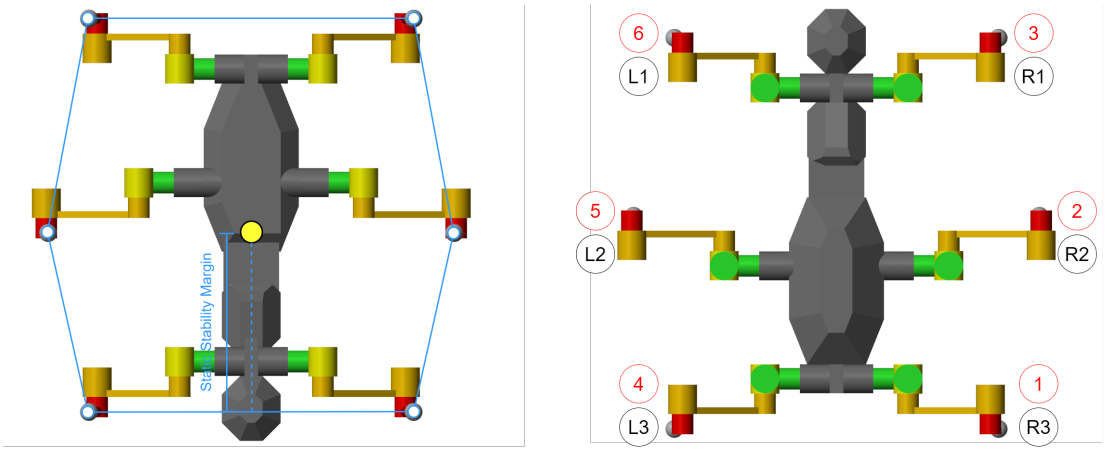
Length L_{stride}, distance covered by the leg during swing phase; **Stride Height** h_{stride}, height of the foot in the highest point of swing phase respect to a plane fitted to the ground; **Stride Slant** (σ_{stride}), angle between the horizontal line and the tangent to the swing trajectory at the end or the beginning of the contact.[1]
The velocity of the robot will be the opposite of the velocity of the feet in stance phase an thus:

$$v = \frac{L_{stride}}{t_{stance}} \tag{2.2}$$

2.2.2. Legs and gaits

Legged robots can have any number of legs higher than zero, the most relevant categories are 3: biped, quadruped and hexapod (or any robot with more legs). The classification is based on the stability, the capability of the robot to not overturn. The static stability is determined by the position of the vertical projection of the center of gravity, onto the vertical projection of the polygon being convex hull of the contact points with the ground (**Support Polygon**) (fig. 2.9a). If the said center of gravity, is inside the support polygon and the robot is still, it is guaranteed to not overturn. The smallest distance between the projected center and the projected border is often used as a parameter to quantify the static stability, and is called **Static Stability Margin** S_m (fig. 2.9a). The static stability does not ensure the dynamic stability, since the first does not account for inertia forces, that are present during motion. On the other hand, lack of static stability do not implies the lack of dynamic stability, in fact, biped do not generate any polygon and

thus the center of gravity will always be outside the boundary, yet humans are able to walk and stand still; a mix of inertia forces and control actions grants the balance. If biped robots never ensure static stability, quadrupeds ensure it when still, but generally not while moving, while robots with six or more legs can potentially ensure it in both conditions. While adding legs improves the static stability and reduce the complexity of the balance control, it also reduce the speed and increase the overall complexity. [1]



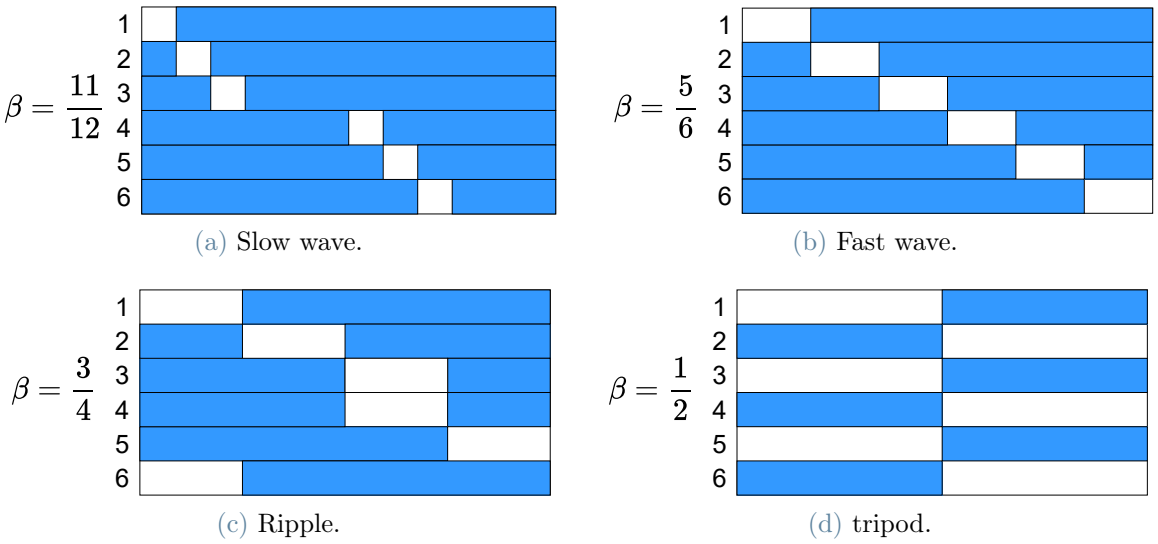
(a) Robot bottom view, with support polygon, in still condition.

(b) Legs nomenclature: in black labels, in red pure indices.

Figure 2.9: Support polygon and leg nomenclature.

Presence of multiple legs, implies the need of a coordination between them. The pattern that determines the alternation between swing and stance phases in the legs, is called **Gait**. If the duty cycle is the same for all legs, the gait is a **Regular Gait**. The **Leg Phase** is a measure of the delay, with which each leg contact the ground respect to a reference leg (to not be confused with the use done until now of the word *"phase"*).
From now on, is useful to focus on the sole example of the hexapod, and we can define a notation for the legs as in fig. 2.9b.
For hexapods exist several gaits: free gaits, precision footing, equal phase, wave, etc... The scope of this thesis is to work with a robot using regular and periodic gaits, and a very basic case. Gait patterns can be easily generated by assuming a cycle 12 units long, and a swing phase long 1, 2, 3 and 6 units. In this way, applying the proper phasing, 4 gaits are obtained, respectively **Slow Wave**, **Fast Wave**, **Ripple** and **Tripod** (fig. 2.10). Those are the only ones considered, because are the ones evaluated in the benchmark of this work.

To generate the motion, a controller will generate swing-stance pulses like in fig. 2.10 (see section 5.5.6), that will be delivered to the legs, the leg will determine a foot trajectory



(a) Slow wave. (b) Fast wave.

(c) Ripple. (d) tripod.

Figure 2.10: Gait diagrams: on x axis an adimensional time of 12 units, on the y the legs referred as fig. 2.9b: Where colored, the leg is in stance phase, where white, the leg is in swing phase.

like in fig. 2.8b (see section 5.5.1), thus generating a coordinated motion.
Needless to say, that in any of this gaits, if a lateral foot motion is added to the legs, the robot can steer and follow a path.

2.2.3. Posture parameters

A legged robot has still others degrees of freedom to be defined, in particular the ones regarding the posture. While in fig. 2.8b the foot trajectory in a body-attached frame has been defined, it is not obvious where this frame is placed, is clear that the same trajectory can be followed by the foot, while the robot can have different legs apertures or body heights. To define this parameters, we can define a reference frame of the foot trajectory in fig. 2.8b placed along y in the median point between the foot most anterior position and the rearmost one, along z to be placed on the ground, and along x to to belong to the foot trajectory plane.
For example, for this benchmark robot, the foot trajectory reference frame, can be placed (for each leg) referred to the E point (fig. 2.7b). Since the posture parameters are meant to be defined for the whole robot and not for a single leg, an E' point is constructed. In fact, as can be noted in fig. 2.11, the four front legs and the two rear ones have motors mounted in the opposite way, and middle legs G points (fig. 2.7b,2.11) are placed higher than the others. Therefore E' will be shifted by the length due to the actuators (BC + DE) and the height due to different joints height Δz_{MJ} (fig. 2.11). From E' are defined

3 parameters: x_{offset}, y_{offset} and z_{offset}, determining the displacement between the foot trajectory reference frame and E' along x,y and z, in the local horizon frame (fig. 2.11). Must be clarified that the frames referred to the left legs have an inverted x axis, being thus left-handed frames, but maintaining the same x,y and z offsets.

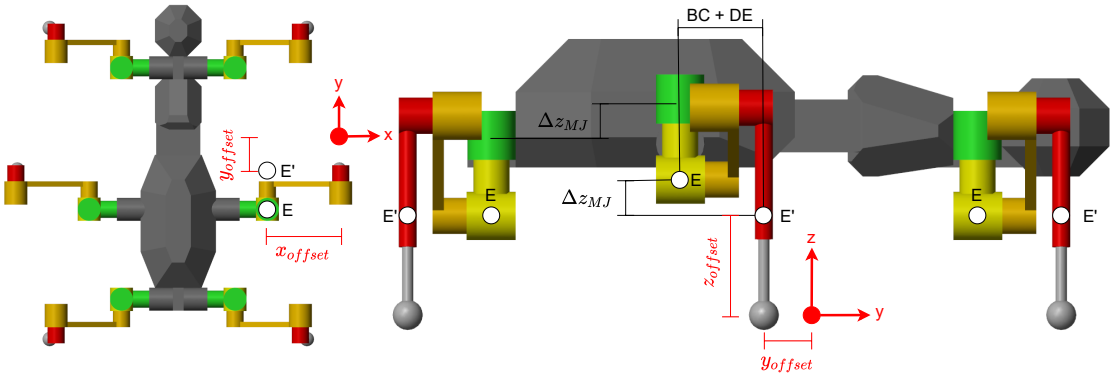


Figure 2.11: Posture parameters: Indicates the positions of E and E' nodes. In black parameters due to structure geometry, in red the foot trajectory reference frame of leg R2 and parameters referring to its position.

x_{offset} determines the leg aperture, increasing it increases the lateral stability, but reduce the maximum stride length and height allowable. y_{offset} determines the center of gravity position on the support polygon, adjusting this value with changing longitudinal slope, can be beneficial for the stability. z_{offset} determines the body height, higher value allows to pass higher obstacles, but increases consumption and reduce maximum stride length and leg aperture allowable. For the benchmark robot z_{offset} is properly the **E Height** h_E, the height of the lowest E point respect to the ground, while the **Body Height** is determined by the E height. In fact the additional degree of freedom between F and G (fig. 2.7b) is used to maintain the legs aligned to the vertical, therefore modifying the relation, but without adding other parameters.
This construction is specific for this rover, but can be extended to any legged robot by defining a **Body Height**, a **Legs Aperture**, a **Center Of Gravity Alignment**, and defining relations that can grant the values to be fulfilled based on the specific kinematic of the robot.

2.2.4. Reflexes

Determining foot trajectories based on predefined shapes and centralized pulses, so not evaluating each footstep position based on terrain map, brings some limitations. Regular-gait-driven leg motion is simple and effective on regular and flat terrain, but on rough

terrain with slopes, obstacles and holes, the robot is subject to deviations, blocking and overturning. To improve the reliability, some reflexes are generally added. Here some examples:

- If the rover is moving downhill or the foot is falling in a hole, the time to reach the ground after swing phase will be higher, therefore, the swing motion should be extended, to ensure ground contact and the rise of other legs should be prevented, to avoid unstable conditions.
- On the contrary, if is moving uphill, the time to reach the ground will be shorter than expected, so the motion could be stopped at contact.
- If an obstacle is hit during swing phase, the rover should be able to adjust the foot trajectory to avoid it.
- If the gait and trajectory controllers demand a motion that does not satisfy the mechanical constraints, like elongating too much the leg or moving it to pass through a part of the robot, the motion shall be modified or temporarily halted.

What must be clear, is that to actually work in a real environment, the full motion will be affected by much more complex controllers than the simple gait pulses described before.

3 | Cost function

The scope of this thesis is to enable a path planning, minimizing energy consumption for a legged rover. In particular to develop a cost function, that can map terrain data and a trajectory into an energy consumption along the trajectory. Furthermore, it has also to select the robot parameters described in section 2.2 to ensure path feasibility and minimum consumption.
The cost function could be applied to any graph edge, but here the path planning techniques considered are of the RRT family. Because they allow a variety of directions, that, since the consumption and feasibility are strongly affected by the direction at which the terrain is traversed, allows better optimization and higher success rate.

3.1. General concept

The fundamental idea is simple, build a robot model able to simulate the robot motion and compute the energy consumption (chapter 5), test this model on different terrains and build a matrix of results so that for any combination of terrain and robot parameters, can be interpolated the consumption. When evaluating an edge cost, a rectangular oriented window slides along the edge (section 3.3), similarly to ACE (see section 1.2), it maps the terrain in few synthetic parameters (section 3.4), and use them to access the matrix (section 3.5). The cost of each feasible parameter combination is evaluated and then the parameters combination with the lowest consumption is selected, thus obtaining parameters and estimated consumption.
Clearly an approach of this type is quite brute force, and is very computationally expensive, thus the problem must be formulated to reduce the matrix dimensions.

3.2. Parameters definition

The matrix will be evaluated with as many dimensions as parameters to evaluate, including terrain parameters and robot parameters, as said before, the number of parameters

must be reduced to the strictly necessary.

For the benchmark robot the parameters to be determined are: x_{offset}, y_{offset}, z_{offset}, σ_{stride}, h_{stride}, L_{stride}, t_{cycle}, t_{swing}, t_{stance}, β and other gaits parameters. All the gait parameters can be condensed into a gait type, that contains all coordination parameters, for example, for the benchmark robot have been used four gaits as described in section 2.2.2. t_{cycle}, t_{swing}, t_{stance} are at this point related to each other though β and eq. (2.1), so any of them can be selected as a single parameter. Using eq. (2.2) the times can be replaced with the **Forward Velocity** of the robot ($v_{forward}$). σ_{stride} adds more complexity while not actually improving performances, thus has been set to maximum feasible value, and removed from the free parameters. z_{offset} can be substituted by h_E, y_{offset} is computed by the controller based on slope and gait (section 5.9), and x_{offset} has been set to a fixed value. In this way the robot parameters that can be tuned during the path planning are 5: h_{stride}, h_E, L_{stride}, gait, $v_{forward}$.

The terrain complexity must be reduced to few synthetic parameters, mainly the slopes, obstacles heights and rocks coverage. The one selected are:

1. **Longitudinal Slope** $\alpha_|$: while climbing a slope, the potential gravitational energy increase and for sure the consumption increase.
2. **Transversal Slope** $\alpha\_$: that causes a deviation in the trajectory and so an higher consumption from the control action, or even an unfeasibility due to mechanical constraints.
3. **Max Obstacles Height** $h_{obstacle,max}$: that force the robot to increase the body height and the stride height, to avoid hitting obstacles.
4. the **Missing Contact Coverage Factor** ρ_{MCC}: percentage of the terrain having local slope unable to grant the adherence of the leg, causing its slippage and loss of stability.

Assuming that obstacles do not affect the consumption, since they are avoided, so their presence is not perceived, the max obstacle height can be used to directly determine the stride height and E height. This does not entirely hold, since the feet can hit obstacles and graze them, but is still considered reasonable. Instead, the percentage of terrain with high local slope can be used to determine the gait, ensuring enough feet to have stable foothold, shown exhaustively in section 5.10.

Therefore the parameters independently affecting the consumption are 7:

1. longitudinal slope
2. transversal slope
3. stride height
4. E height
5. stride length
6. gait
7. forward velocity

The results of the wide evaluation will be stored in the form of a **Cost Of Transport Matrix**, where the **Cost Of Transport** is defined as the energy consumption per space traversed:

$$COT = \frac{E}{L} \tag{3.1}$$

where E is the energy consumption and L the length of the trajectory covered by the robot. The **Cost Of Transport Matrix** will thus be $COT(\alpha_{|}, \alpha_{\_}, h_{stride}, h_E, L_{stride}, \text{gait}, v_{forward})$, meaning that accessed with a combination of these 7 parameters, returns a COT (Infinite value if combination not feasible).

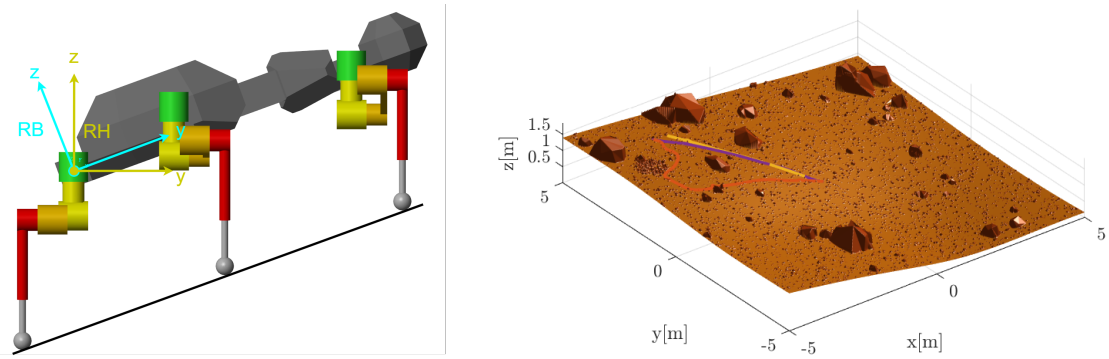
This selection is reasonable for any legged robot, but if other parameters are present, they can be added to the matrix with similar criteria, increasing computational cost, but maintaining validity.

3.3. Sliding window

The sliding window is conceptually a rectangle that crops the terrain, obtaining the portion of terrain, oriented in robot local horizon frame, where the robot will stand. To make unambiguous this procedure, robot frames and terrain must be defined.

The main reference frames used in most of this document, are the frames RB and RH (fig. 3.1a), both are fixed to the robot body, and placed at the middle point between the two back actuators attached to the body along their cylindrical symmetry axis. RB (body frame) has y oriented as the vector connecting point G of leg R3 to point G leg R1

(fig. 2.7b,2.9b), x oriented as the vector connecting point G of leg L3 to point G of leg R3 and z is the cross product between x and y. RH (local horizon frame) has z oriented as opposed to gravity, x is the cross product between y of RB and z of RH, y is the cross product between z and x.



(a) RB and RH reference frames

(b) Tearrain example.

Figure 3.1: Frames and terrain.

The terrain is modeled as a heightmap $Z_{terrain}(x, y)$ (fig. 3.1b), a rectangular matrix containing for each x,y combination inside its ranges, a z value elevation of the ground (x,y in terrain frame). A **Limit Contact Slope** can be defined as the maximum slope at which the foothold is stable, this value depends on the foot-terrain contact type. From $Z_{terrain}(x, y)$, the local terrain slope in each point can be computed. All the terrain points with slope higher than the said value, are highlighted into a boolean **Missing Contact Matrix** $\Gamma_{terrain}(x, y)$, useful for later access. (more details on terrain data and generations in section 5.6).

The sliding window will not have a continuous motion, but will be called at $L_{sampling}$ (**Trajectory Sampling Length**) intervals along the trajectory, thus building a discrete series of terrain evaluations.
The sliding window is defined starting from RH, that define the position and orientation of the robot. The window must be large enough to include all possible terrain traversed by the robot, from the actual sampling point to the next one. The window will have a **Window Length**, **Window Width** to determine it size and a **Window Back Offset** to determine its position along y_{RH}, while along x_{RH} is positioned to be symmetric (fig. 3.2). The window is actually composed by a series of bands (fig. 3.2), A-band must include every position in the plane that can be reached by any foot, E-band every position that the basal actuators in D and E can reach (fig. 2.7b), and T-band comprise the gap between the two E-bands. A and E bands can intersect each other (fig. 3.2). The reasoning is that

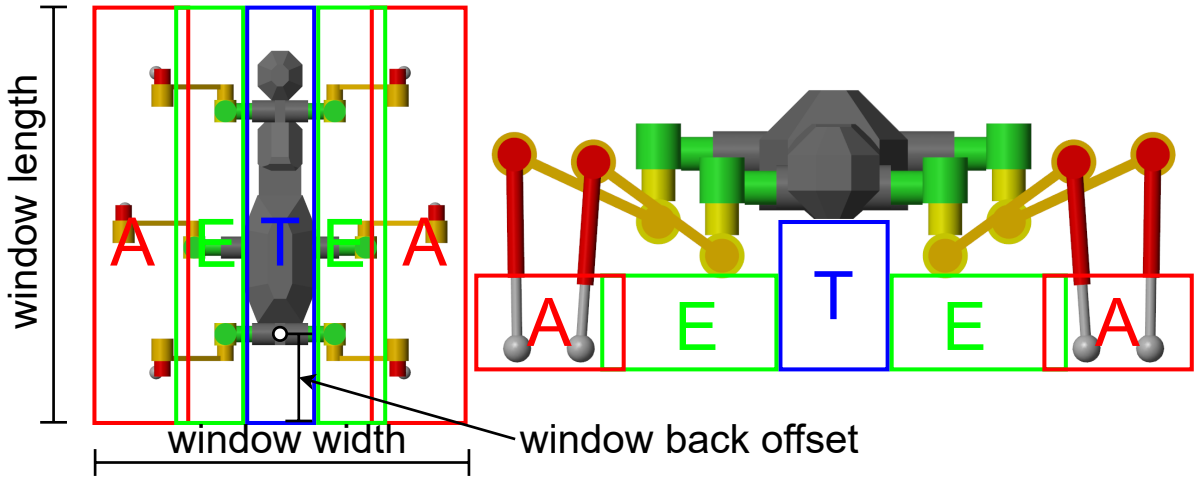


Figure 3.2: Sliding widow parameters and subdivision in bands. Specific data and computations in section 5.3.

the terrain characteristics in A-band will affect foothold and foot behavior, the ones in E-band (just obstacles heights) can hit the lower actuator, while the ones in F-band can hit the rover torso. Must be noted that if the robot is steering, the feet will move along x, increasing the width of the A-bands. To have a simple code, the function select one of just two A-band widths, depending if the path is a straight line or a curve one. The union of these bands determines the sliding window. For window parameters computations for benchmark rover see section 5.3.

The cropping process generates a new terrain (x,y)-points matrix, relative to the sliding window, positions and rotates it as for RH frame, and through linear and nearest interpolation obtains respectively a corresponding window-relative heightmap $Z_{window}(x, y)$ and missing contact matrix $\Gamma_{window}(x, y)$. If the window goes outside the terrain boundaries, its considered as an unfeasible position.

3.4. From terrain to synthetic parameters

The portion of terrain obtained in the previous section, must be synthesized in few parameters as described in section 3.2. To do so, the points of the $Z_{window}(x, y)$ can be grouped as the bands described in fig. 3.2, obtaining points sets that can be called: A-set, E-set and T-set.
The first step is to fit a plane to the terrain, since this is meant to represents the slopes perceived by the rover, the plane is fitted to the A-set. In fact, if the rover were to walk on a loading ramp, for example, A-set will have a positive slope while T-set a zero

slope, but the robot will be unaffected by gap between the ramps, which presence is totally irrelevant. Thus the fitting plane obtained including both the sets, will be more computationally expensive and less accurate in estimating the slopes felt by the body.

A plane in 3D space can be expressed as

$$z(x, y) = ax + by + c \tag{3.2}$$

Therefore, for a set of n points with x,y,z coordinates, an over-constrained system can be set up in the form:

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \tag{3.3}$$

Solving it with a least-square method, results in the plane coefficients a, b, c. Putting eq. (3.2) into implicit form

$$\alpha x + \beta y + \gamma z + \delta = -ax - by + z - c = 0 \tag{3.4}$$

a vector normal to the plane $\underline{\mathbf{V}}$ and its versor $\underline{\mathbf{v}}$ are then

$$\underline{\mathbf{V}} = [\alpha, \beta, \gamma] = [-a, -b, 1] \tag{3.5}$$

$$\underline{\mathbf{v}} = [v_x, v_y, v_z] = \frac{\underline{\mathbf{V}}}{||\underline{\mathbf{V}}||} \tag{3.6}$$

Since the coefficient in z of $\underline{\mathbf{V}}$ is always 1, and the plane cannot be vertical, the vector is always exiting the ground. From the component of this versor, the slopes can be obtained: Defining the **Longitudinal Slope** $\alpha_|$ and **Transversal Slope** $\alpha_\_$ relatively to the RH frame as in fig. 3.3, they can be computed using the relations:

$$\alpha_| = -\arcsin(v_y) \tag{3.7}$$

$$\alpha_\_ = -\arcsin(v_x) \tag{3.8}$$

Obtained the slopes, the next objective are the obstacle heights. To compute these, a new plane can be generated, having same longitudinal slope of the fitted plane, but zero transversal slope. This can be done using eq. (3.2) with b and c of the fitted plane and

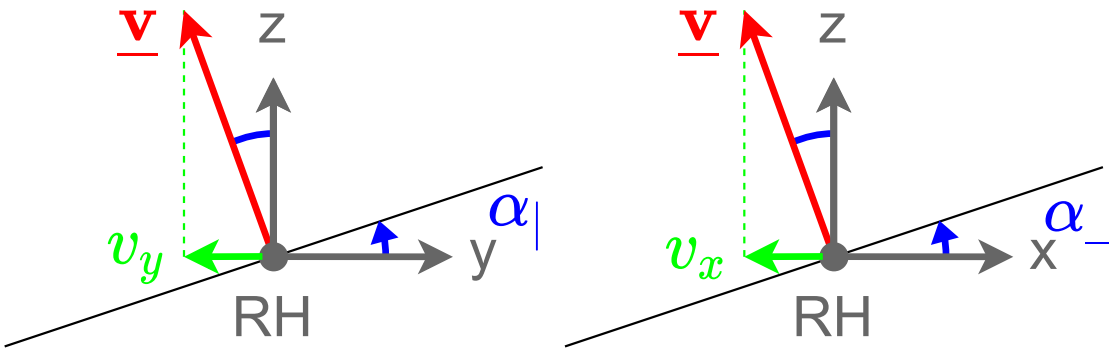


Figure 3.3: Scheme of the terrain slopes and their relations with the plane versor.

$a = 0$, obtaining a $Z_{plane}(x, y)$. An heightmap of the obstalces can be created:

$$H_{obstacles}(x, y) = Z_{window}(x, y) - Z_{plane}(x, y) \tag{3.9}$$

$H_{obstacles}(x, y)$ points can be grouped according to A, E and T bands and the maximum obstacle height can be computed for each band obtaining: a **Maximum Obstacles Height** for each band ($h_{obstacle,A,max}$, $h_{obstacle,E,max}$, $h_{obstacle,T,max}$).
Finally the $\Gamma_{window}(x, y)$ can be reduced to points in the A-band $\Gamma_{A-band}(x, y)$ and a **Missing Contact Coverage Factor** ρ_{MCC} can be computed as the number of points where the $\Gamma_{A-band}(x, y)$ is true over the total number of points in $\Gamma_{A-band}(x, y)$. ρ_{MCC} indicates the likelihood of a robot foot to step on a steep slope point.
Note: ρ_{MCC} is computed including both A-bands, but in this way if the rover passes with a band over a rough terrain full of small rocks and cracks, while the other on a plain terrain, it programs a gait for a terrain moderately rough. This cause the robot to label very rocky terrain as untraversable, but instead of avoiding it entirely, it just avoid it with one train of legs, passing on with the other. A more reasonable procedure for future works, would be to compute ρ_{MCC} for each A-band and retain the worse value.

Therefore the terrain is synthesized in: $\alpha_|$, $\alpha\_$, $h_{obstacle,A,max}$, $h_{obstacle,E,max}$, $h_{obstacle,T,max}$ and ρ_{MCC}.

3.5. From terrain parameters to robot parameters and consumption

To access the COT matrix the required parameters are: $\alpha_|$, $\alpha_\_$, h_{stride}, h_E, L_{stride}, gait, $v_{forward}$; while the computed data are: $\alpha_|$, $\alpha_\_$, $h_{obstacle,A,max}$, $h_{obstacle,E,max}$, $h_{obstacle,T,max}$, ρ_{MCC} and the **Curvature Radius** of the trajectory R_C is assigned. Must be remembered that COT is a matrix evaluated through simulation, and thus for each dimension will be evaluated for a list of values of a parameter. To avoid a seven-dimensional interpolation, just the two slopes are kept as real values doing a 2D interpolation, while the other 5 parameters, since they can be selected by the function, can be chosen as coincident with evaluated values, making the 7D space not continuous, but easier to explore.
To select those parameters, first must be discarded all unfeasible cases, to do so, a series of checks are performed:

1. The COT matrix already by itself is made of finite and infinite or NaN (not a number) values, the non-finite values are considered unfeasible.
2. For each parameter a feasibilty check is performed (since the exact computations are dependent on the kinematic of the specific robot they are discussed in detail in the later chapters):
   (a) The h_{stride} must allow the feet to overcome any obstacle, thus just values higher than the $h_{obstacle,A,max}$ plus a margin are maintained (section 5.3).
   (b) The h_E must allow the basal actuator and the body to not hit any obstacle, thus just the values that allow a certain clearance between the lower actuator an the $h_{obstacle,E,max}$ and between the body and the $h_{obstacle,T,max}$ are maintained (section 5.2.2 and section 5.3).
   (c) The gait is considered feasible if is able to keep under a certain value, the time percentage where just two feet are in stable contact to the ground. Each gait has a limit ρ_{MCC} after which cannot grant the value to be respected (those values are computed in section 5.10) and gaits not suited for the terrain ρ_{MCC} are discarded. Furthermore tripod gait is allowed just on flat terrain, therefore if the terrain slopes exceed the arbitrary value of 5°, tripod gait is discarded.
   (d) h_{stride} not only must allow the feet to overcome obstacles, but also to overcome the terrain slope, thus to have the foot trajectory highest point, higher than the starting and final position of the foot plus a margin. The combinations of h_{stride} and L_{stride}, that cannot grant this for the terrain $\alpha_|$ and $\alpha_\_$ are

discarded(section 5.2.1).

(e) The combination of L_{stride}, h_{stride}, $v_{forward}$ and gait, must avoid quick leg motion causing high torques, oscillations and tripping of the rover (section 5.2.3).

(f) The combinations of h_E and L_{stride} allowing a minimum radius of curvature higher than R_C are discarded (section 5.2.4).

3. Selecting a combination of robots parameters, the COT matrix is reduced to the form $COT(\alpha_|, \alpha\_)$, thus allowing a 2D interpolation using $(\alpha_|, \alpha\_)$ of the terrain. Being it a grid of points in the $\alpha_| - \alpha\_$ plane, the terrain point can be interpolated only if all the four matrix points delimiting the tile where the terrain $(\alpha_|, \alpha\_)$ point is contained, are feasible. If that is the case, is reasonable to consider all the points in the tile to be feasible. Thus all the parameters not able to grant the four tile vertices for the interpolation are discarded.

After the unfeasible cases have been discarded, the parameters are selected based on different criteria, in succession and reducing the cases to evaluate: (for "feasible" is meant: "having at least one full feasible combination of parameters")

1. Since has been deemed not reasonable to have a robot keeping all the time the slowest and safest gait (slow wave), when the terrain allows it, is selected a fast gait. In particular, the feasible gait with lowest duty factor β (so the fastest), is selected.

2. Higher h_{stride} intuitively increase the consumption, requiring a longer foot path and thus higher speeds and accelerations, therefore is selected the lowest feasible h_{stride}.

3. As for h_{stride} also h_E intuitively increase the consumption, plus it limits other parameters due to leg kinematic, therefore is selected the lowest feasible h_E.

4. For all the remaining combinations, the COT is interpolated and the combination of L_{stride} and $v_{forward}$ with the lowest COT is selected. The interpolation process is, in general, bilinear, but since the COT function along $\alpha_|$ has a parabolic behaviour, to improve the accuracy, if an adjacent tile along $\alpha_|$ is feasible, the interpolation is modified. First are performed two parabolic interpolations along $\alpha_|$, than a linear one along $\alpha\_$.

At the end of this process, the robot parameters h_{stride}, h_E, L_{stride}, gait, $v_{forward}$ are defined and the COT has been estimated.

3.6. From trajectory to consumption

The computations performed until now are on a single point, in general an edge is composed by a trajectory of an arbitrary shape. To estimate the cost of a trajectory a possibility is to sample it. Defining a **Sampling Length** $L_{sampling}$, a point can be placed along the trajectory every arc of curve of length $L_{sampling}$. Each point will have an (x,y) position, an orientation and a curvature radius R_C. The process showed in section 3.3, 3.4, 3.5 can be performed for each point, and multiplying the COT by the distance to the next sampled point, can be computed the estimated power consumption to travel that arc of curve. The sum of all the arc cost, results in the total edge costs, while the concatenation of all the robot parameters results in the parameters profile along the trajectory.

Must be noted that the impact of R_C to the consumption has been considered limited and thus does not enters the computations, except in limiting the parameters, changing A-bands width and in modifying the way to compute the length to the next point, since it is computed along the trajectory and not as shortest distance between the points.

3.7. Cost function restrictions

This way of building the cost function cause some limitations.
Is clear that this algorithm can work only if the robot is able to determine its position and to follow a trajectory relatively tightly. If the robot controller is not able to compensate perturbations and uncertainties, the robot will be driven outside of the path scanned by the sliding window, reaching areas not evaluated. For the purpose of the path planning of a robot of this type, a path planner based on the propagation of uncertainty is necessary.
Furthermore, for how is simulated and evaluated the terrain, with a simple hightmap, all the terrain analysis must be considered on rigid terrain and with rocks perfectly fixed on the terrain. Is likely possible to extend the algorithm including a relation for different types of terrains, like sandy terrains or with rocks that can slip when hit or stepped on by the robot, but this work has limited the study to the rigid terrain case.
The terrain generated and the terrain observed by the robot are, in this work, identical, in reality it would not be the case. The resolution would be much lower for further away terrain and terrain behind rocks and hills would be unknown. Nevertheless, in a real environment, big objects, like hills and rocks, positions and sizes, can be identified even in a low resolution; and since small rocks that affect the ρ_{MCC} are used just to estimate a synthetic value, exact position and shape might not be necessary and a direct

ρ_{MCC} estimation could be performed from images. While ways to roughly estimate the terrain characteristic where is not observable, are needed also for any other path planning algorithm.
With all those premises, the algorithm can be expanded, and can be considered relevant also for real applications, not limited to the situations studied in this thesis.

4 | Path planning algorithms

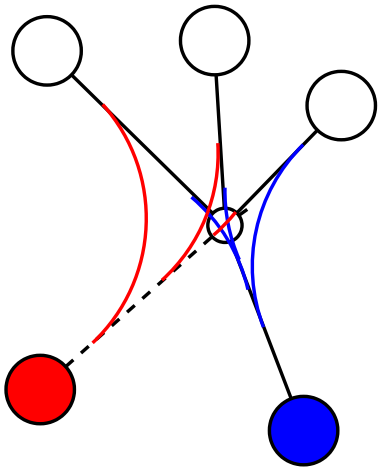
As seen in section 2.1, RRT* can be used for path planning, and, computing the edge cost using the cost function defined in chapter 3, the scope of the thesis is reached. In practice, some modifications to the RRT* must be performed, to allow proper use of the cost function and to fulfill other needs.
In particular, the main issue regards the curvature radius. In fact, even if being a legged robot, with degrees of freedom that would allow the robot to steer on spot, is not actually the case. For how is programmed the gait pattern generator (section 5.5), the gaits are generated with some fixed parameters and aided by some reflexes. When the robot is steering, the legs will have a transversal motion, that will bring them far from the body or close to it. Obviously, there are some limits to this motion, in general, the leg cannot be stretched infinitely, and cannot penetrate the robot body. Therefore, some action must be performed, when one of this limits is approached. The more effective idea, would be to start a swing phase to move the leg away from the limits and continue the motion, still ensuring enough foothold for stability. The controller used, instead is a very simple one (it was not the scope of the thesis, thus limited resources have been allocated to it), and when a leg reach its limit, the whole stance phase motion is stopped, preventing the reach of unfeasible or unstable positions. Must be noted, that the robot consumes even while being still, so a consumption with a zero travel distance will invevitably increase the COT. As said before, the motion restarts when the legs at limit are brought far from the limit with a swing phase, but since the leg coordination of the controller keep following the gait pattern, might be necessary to wait for the cycle to reach the leg that is at the limit. This cause the robot to stop for some time, causing a huge increase of COT. Therefore, this condition must be avoided, and since sharpest the turn, the bigger the legs lateral motion, and higher the chances of reaching the limits, the robot have imposed a minimum curvature radius. This radius is affected by the $v_{forward}$, smaller speed implies smaller radius, therefore, to perform sharp turns, it has to slow down increasing the consumption (section 5.2.4). This shows that, not only there are some limits to the minimum curvature radius, but also that the sharper the turn, the higher the COT. Therefore the algorithm must be able to put limits for turns and to explore different curvature radii, to use big radii

when possible, reducing the consumption, and to use small radii when needed, increasing manoeuvrability. Also, despite for different reasons, is a more natural motion to steer with high curvatures at high speeds.
This curvature considerations, also affect the path optimization process. When a feasible path is found, often it contains not needed curves, due to the sampling nature of the RRT. RRT*, increasing the samples, can make the path smoother, but might be desirable to perform this process quicker than with pure random sampling.
Furthermore, this planner is meant to work on a short range, so if the final destination is 100 m ahead, and the planning is for 5 m, is pointless to put a single target in the planning, but might be better to plan for a direction. So, if a particular point is unfeasible or expensive, the robot can still get closer to the final destination.

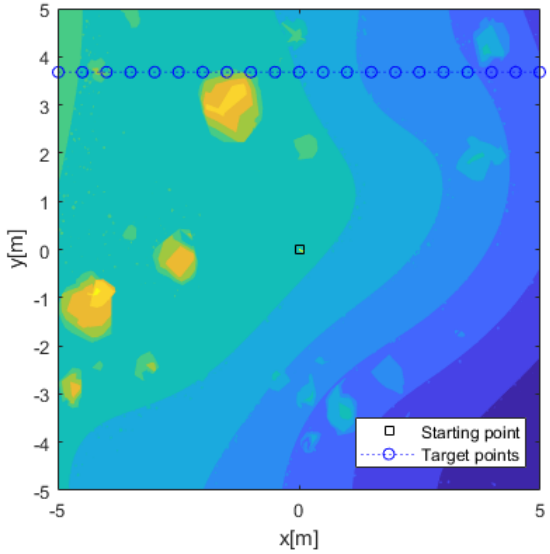
4.1. RRT*

The simplest idea to add the curvature constraint, would be to just replace each edge-edge corner with a curve. This would introduce some problems, in particular the connection between vertex will be dependant on a third vertex. This is relevant especially in the rerouting; where to check if the connection to a vertex is better, would normally be done just evaluating the vertex; but in this case would need also the children (fig. 4.1a). For this reason, the first algorithm, directly neglects the curvature radius and works as the base RRT*, but a feasibility condition on the limit angle between edges is added. If the new generated edge cannot fulfill this condition, it is immediately discarded, before running the expensive cost function.
This process runs the cost function just on straight lines, thus is not able to estimate cost due to steering, and turn parameters. To at least obtain a trajectory followable by the robot and with proper parameters, a post processing of the path must be performed.

The algorithm will connect an initial Dubin state $\underline{\mathbf{i}} = [x, y, \vartheta]$(section 2.1.4), to some target positions $\underline{\mathbf{t}} = [x, y]$. As already said, if the succession of states to reach the target must be defined, the target is not necessarily a precise state. It could be a position or a generic direction. In these algorithms, the target has been assumed as a list of target points $\underline{\mathbf{t}}$. While the algorithms accepts any list of points, they have been selected as points onto a line, positioned in the desired target direction (fig. 4.1b). Must be remembered, that the states are referred to the RH frame of the robot, thus the target line cannot be at the edge of the terrain, because while RH will be in the target, must be granted that the robot has feasible terrain to step on (fig. 4.1b).



(a) Curvature connections between three child nodes, changing the parent node.



(b) Starting point and target points.

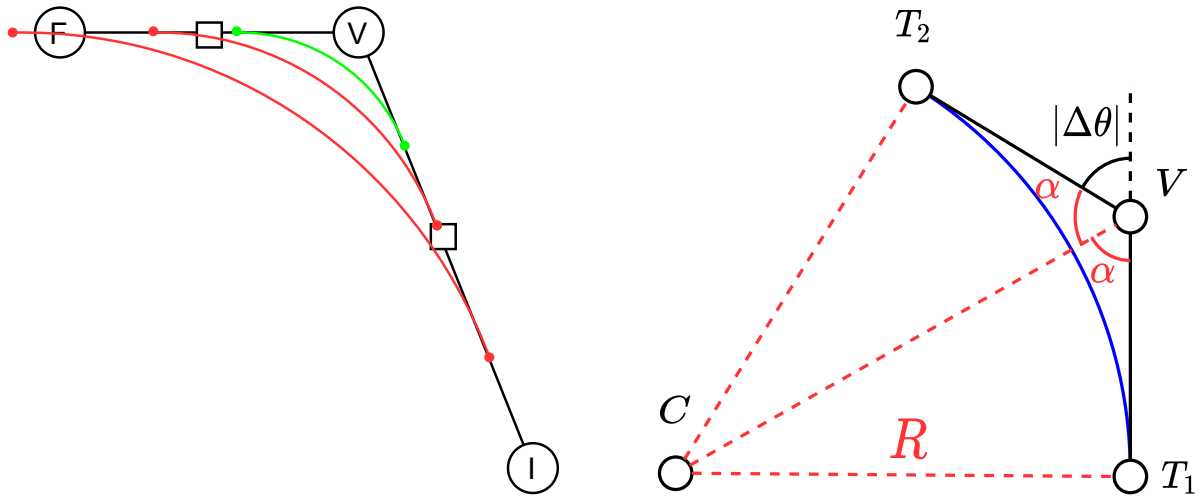
Figure 4.1: Curvature and target nodes examples.

4.1.1. Feasibility function

The feasibility function, defined tree points I,V,F, the IV and VF edges and a minimum allowed curvature radius $R_{c,min}$; checks if between IV and VF, a curve with curvature radius bigger than $R_{c,min}$, can connect the middle points of the two segments (fig. 4.2a). Since the curves are built in this algorithm by cutting corners, each segment will be affected in its first half by the curve to get to the precedent segment, and in its second, by the curve to reach the next segment. Therefore, just half segment length is available for the curve.

Furthermore, since the curve used is an arc of circumference tangent to the two segments, the tangent points distances to the corner point (VT_1,VT_2), are equal (fig. 4.2b). So, just the minimum of the two segments half length is used and called L_{half}.
Trying to define a **maximum absolute turn angle** between segments $|\Delta\vartheta|_{max}$, can be constructed fig. 4.2b. In the limit case $|\Delta\vartheta| = |\Delta\vartheta|_{max}$, $R = R_{c,min}$ and $VT_1 = VT_2 = L_{half}$. Therefore, can be obtained $|\Delta\vartheta|_{max}$ using equations:

$$\alpha_{min} = \arctan\left(\frac{R_{c,min}}{L_{half}}\right) \tag{4.1}$$



(a) Allowed curvatures (green), not allowed curvatures (red).

(b) Curve construction.

Figure 4.2: Curvature feasibility between edges.

$$|\Delta\vartheta|_{max} = \pi - 2\alpha_{min} \tag{4.2}$$

Obtained $|\Delta\vartheta|_{max}$, if it is bigger than $|\Delta\vartheta|$ between the edges, the curve is feasible, otherwise is not.

4.1.2. Algorithm

In Algorithm 4.1 can be seen the high level code.
For brevity, in the pseudo-code, the symbols € refer to the cost to reach a vertex from the parent (edge cost), while $ to the cost to reach it from the start.

As said in section 4.1.1, the feasibility function is able to evaluate the feasibility of the transition between two edges, therefore when adding a first edge, another segment must exist. For this purpose in (Algorithm 4.1 line 1) "Define first vertex", a first edge is generated. It will simply connect the starting position to a next position at small distance d_{start} along the initial direction. The new point will be the real initial position for the tree. This is just a trick to start the code, without having a node that must be treated differently.
Then, the proper RRT* begins, entering in a while cycle (line 2), and so doing each instruction at each RRT* iteration.
At line 3 "Generate new point", a new point is generated in three possible ways. Two use a random point as for normal RRT*(section 2.1), in one case selecting it as a random

Algorithm 4.1 RRT*

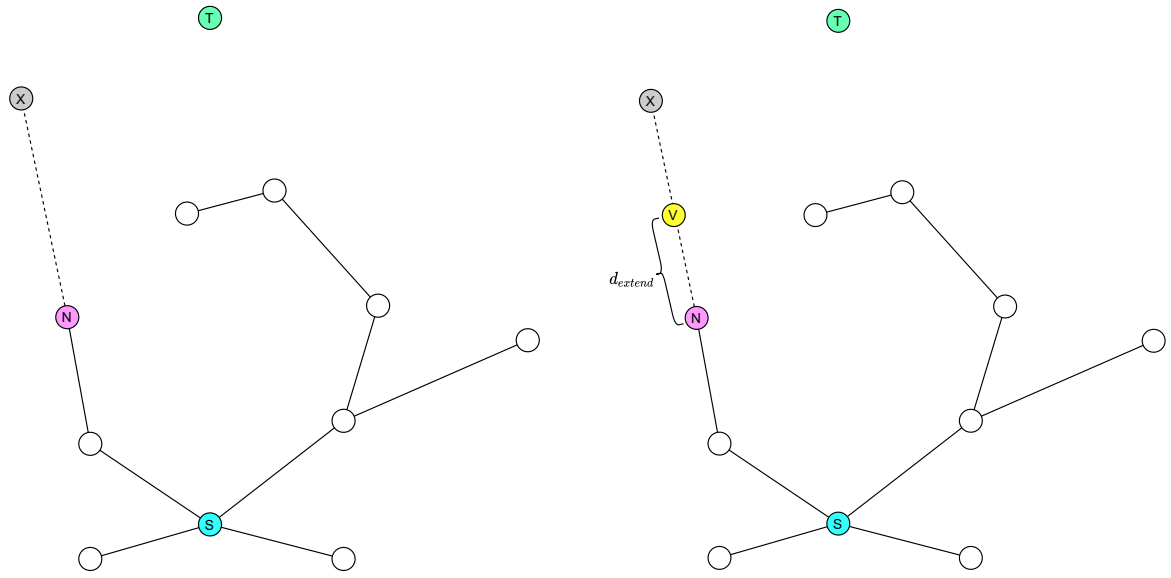
```
1: Define first vertex
2: while Searching do
3:    v = Generate new vertex
4:    Evaluate possible connections of v to the tree
5:    if Any connection possible then
6:       Connect v to the tree
7:       Try to reroute some paths through v
8:       Try to connect v to target nodes
9:    end if
10:   if Any target node $ changed then
11:      Update path
12:   end if
13: end while
14: Retrieve path
15: Post process trajectory
```

point in the plane, in the other selecting it as one of the targets points. The third way directly generates a point in an exact position (Algorithm 4.2).

The **Optimization Bias** σ_{opt} determines the probability of each iteration to be used for the optimization process, while **Exploration Bias** σ_{exp} determines the probability of the exploration iteration to use a target point instead of a random point. Since the optimization process is enabled after a first path is found, it will described later in section 4.1.3.
If the iteration is selected as an exploration one, can randomly use a target point or a random point. In the first case, $\underline{\mathbf{x}}$ is selected to be the target point. If the target points are a list, it is selected using a weighted probability, to make more likely a selection of a nearer, and so, likely cheaper to reach, target. In the second case, a random point on the terrain is selected.
Obtained the random point $\underline{\mathbf{x}}$, the closest neighbouring point $\underline{\mathbf{n}}_{closest}$ of the tree can be identified (fig. 4.3a) and a the new point $\underline{\mathbf{v}}$ is placed on the line linking the two points, at distance d_{extend} (fig. 4.3b). If d_{extend} is bigger than the distance between the two points, $\underline{\mathbf{v}}$ is set equal to $\underline{\mathbf{x}}$.

Algorithm 4.2 Generate new vertex

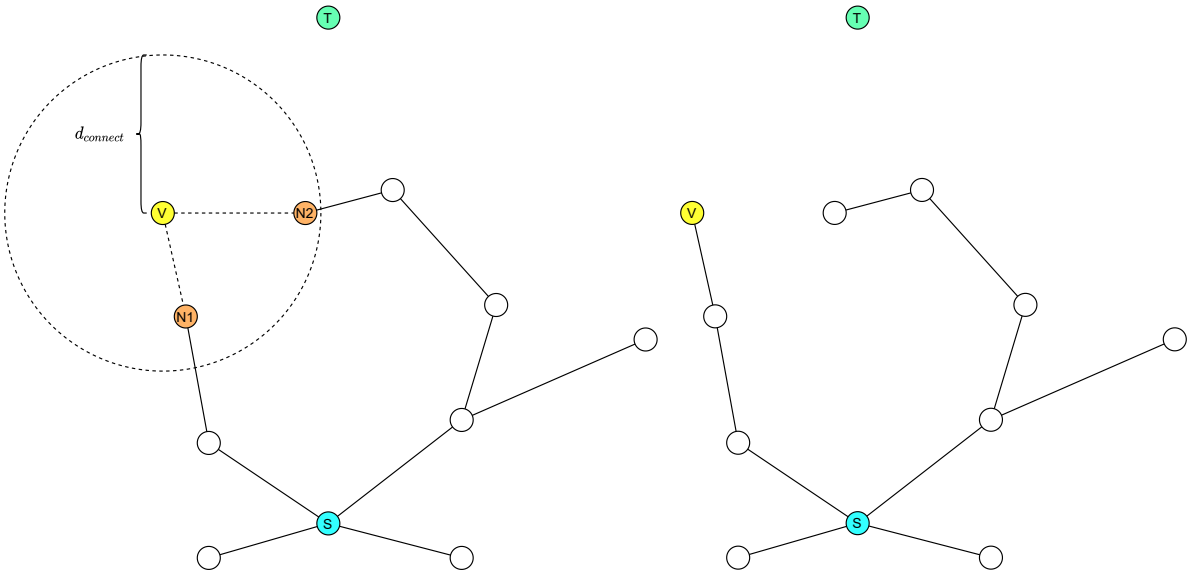
if Optimization iteration **then**
  $\underline{\mathbf{n}}$ = Optimization process
else
  if Target point as random point **then**
    $\underline{\mathbf{x}}$ = Select random point as target
  else
    $\underline{\mathbf{x}}$ = Random point in range
  end if
  $\underline{\mathbf{n}}_{closest}$ = Get closest point of the tree to $\underline{\mathbf{v}}$ that can establish a feasible connection
  if No feasible point **then**
    Discard $\underline{\mathbf{x}}$ and start a new iteration
  end if
  $\underline{\mathbf{d}}_{closest} = \underline{\mathbf{x}} - \underline{\mathbf{n}}_{closest}$
  if $d_{closest} < d_{extend}$ **then**
    $\underline{\mathbf{v}} = \underline{\mathbf{x}}$
  else
    $\widehat{\underline{\mathbf{d}}}_{closest} = \dfrac{\underline{\mathbf{d}}_{closest}}{||\underline{\mathbf{d}}_{closest}||}$
    $\underline{\mathbf{v}} = d_{extend} \cdot \widehat{\underline{\mathbf{d}}}_{closest} + \underline{\mathbf{n}}_{closest}$
  end if
end if



(a) Find closest point. (b) Add new vertex.

Figure 4.3: RRT*: extension.

Next step, in algorithm 4.1 line 4 "Evaluate possible connections of $\underline{\mathbf{v}}$ to the tree" , obtains all the point around $\underline{\mathbf{v}}$ in a radius of $d_{connect}$ (fig. 4.4a), and evaluates the connection to $\underline{\mathbf{v}}$. For each neighbouring point, the feasibility function (section 4.1.1) is run, and if successful, the cost function is run(chapter 3), computing the €-costs and $-costs.

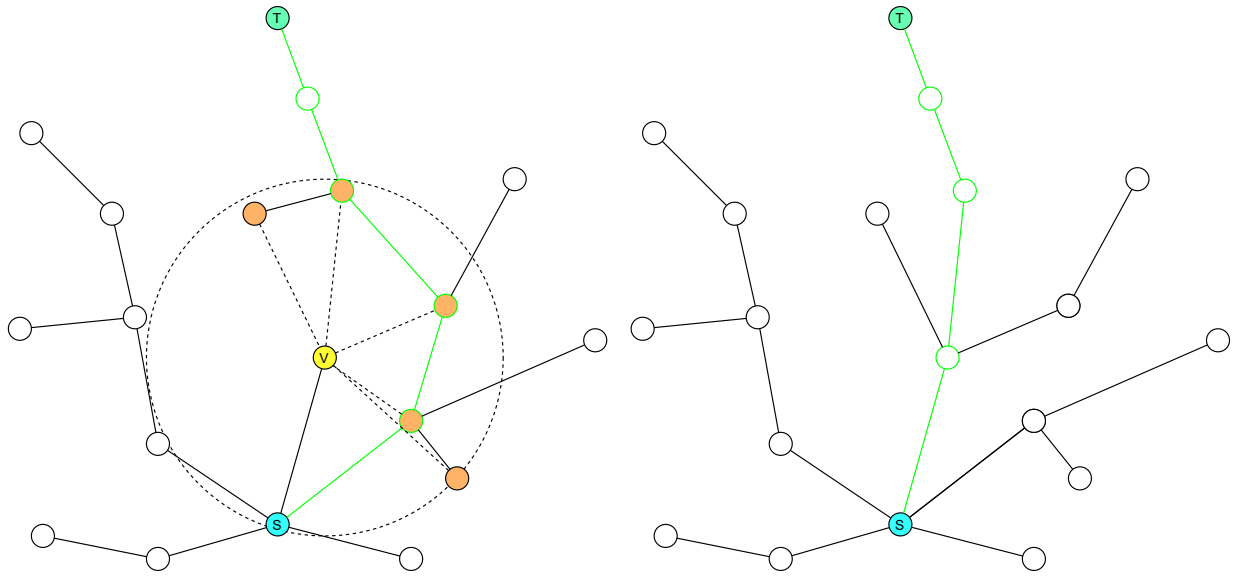


(a) Find neighbour vertices and evaluate connection edges. (b) Connect new vertex to the tree.

Figure 4.4: RRT*: connection.

If any connection is feasible, in line 5 the new vertex $\underline{\mathbf{v}}$ is maintained, and is connected to the tree (line 6) (fig. 4.4b), through the neighboring point granting the cheapest path to the start.

Then, the rerouting of existing points is attempted (line 7). The selected parent node is removed from the $\underline{\mathbf{n}}_{close}$ list. Feasibility and costs are recomputed but this time from $\underline{\mathbf{v}}$ to $\underline{\mathbf{n}}_{close}$ (fig. 4.5a). If any of the $-cost through $\underline{\mathbf{v}}$ is cheaper than the current $-cost for a $\underline{\mathbf{n}}_{close}$, its parent is modified to $\underline{\mathbf{v}}$ (fig. 4.5b). Then all the children must have their $-cost updated.

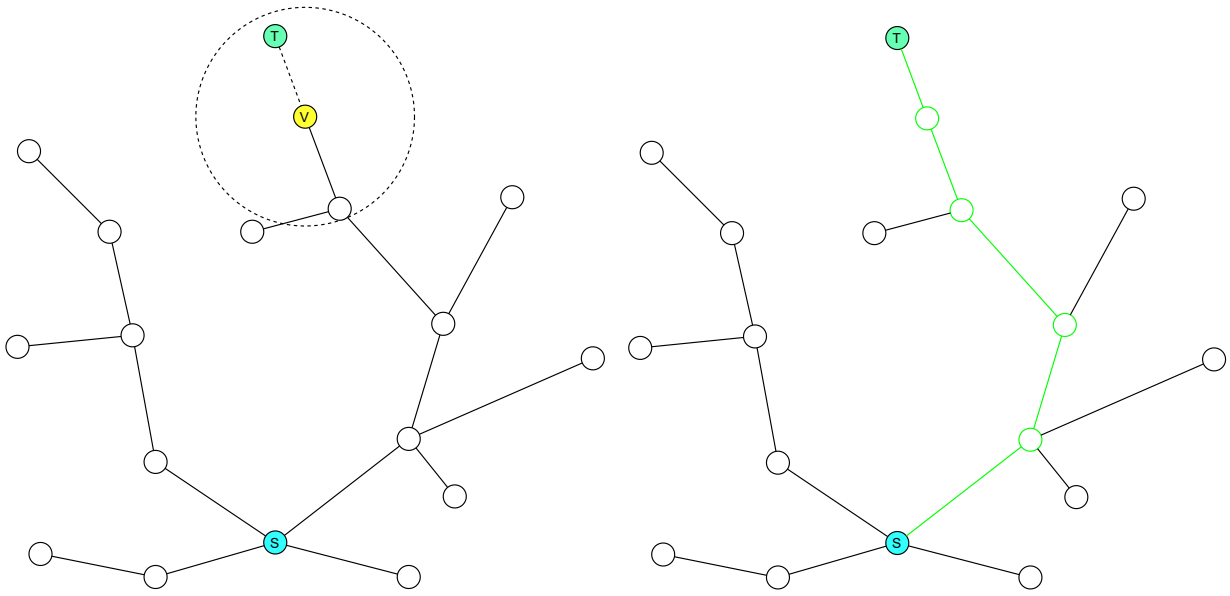


(a) Find neighbour vertices and evaluate connection edges.

(b) Tree and path updated.

Figure 4.5: RRT*: reroute.

Must be noted that, as seen in fig. 4.1a, the new connection might generate an edge direction, that makes unfeasible some children. If this is the case, and the path through $\underline{\mathbf{v}}$ is cheaper, then the vertex to update is split in two. The nodes unfeasible with the new vertex $\underline{\mathbf{v}}$ are left connected to the old node $\underline{\mathbf{n}}_{close}$, while the others became children of a new vertex, that will be in the same position as $\underline{\mathbf{n}}_{close}$ and will have as parent $\underline{\mathbf{v}}$.
A connection (or rerouting) is attempted also to the target nodes (line 8). As for the tree nodes, the neighbouring target points $\underline{\mathbf{t}}_{close}$ are identified (fig. 4.5c), feasibility and costs from $\underline{\mathbf{v}}$ to $\underline{\mathbf{t}}_{close}$ are evaluated, and if a path is cheaper, the nodes are updated.



(c) Connect to target. (d) Path found.

Figure 4.5: RRT*: connect to target.

After all the nodes affected by **v** have been updated, if the cheapest $-cost between the target nodes has changed, the path is updated to the cheapest (fig. 4.5b) (line 9). If is the first feasible one, $-cost of targets (defined as initial value +infinite), changes and a first path is created.
The iterations (line 2) are terminated after the optimization process has finished (see section 4.1.3) or the execution time has exceeded the limit.

At this point, the data relative to the path are retrieved (line 13). Among them the (x,y)positions of the path vertices. Must be remembered that this algorithm does not store the computed robot parameters. In fact, this data are post processed (see section 4.1.4) to obtain full data trajectory, that will be the final output of the algorithm.

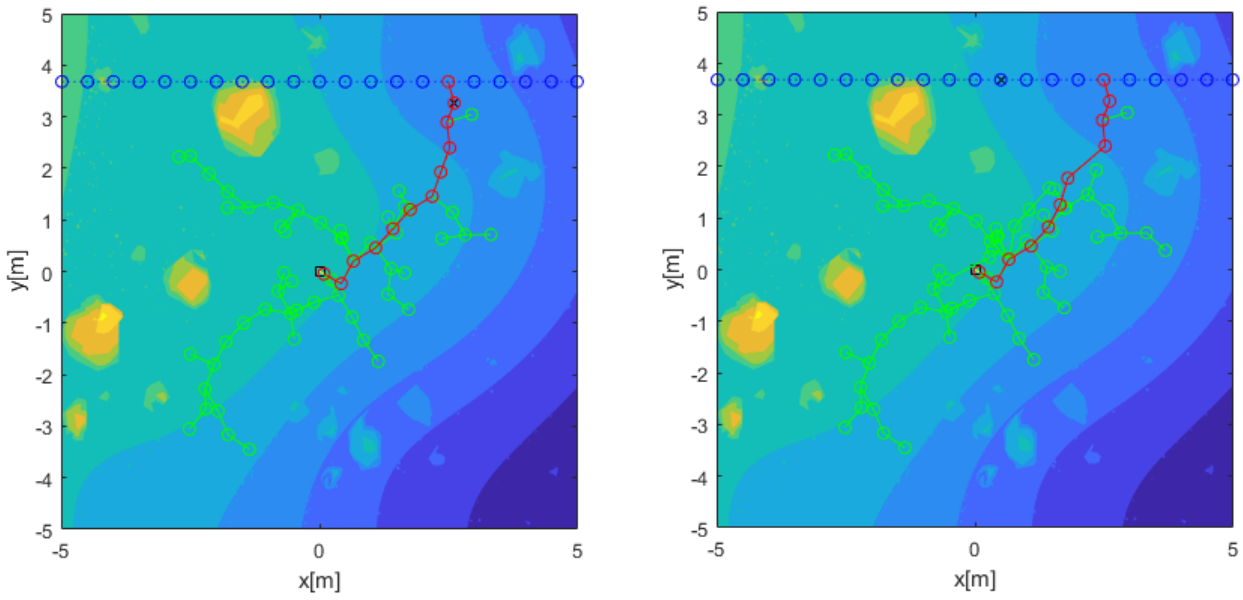
4.1.3. Optimization process

Al discussed in section 2.1, RRT* is an algorithm that can optimize the path continuing the iterations after a path is found. But just keeping sampling is nor granting an efficient optimization, nor giving any stop criteria. Therefore, a simple method to try to optimize the path has been developed.
The idea at the base is to sample in proximity if the path, so to possibly generate a shortcut, removing unnecessary bights; and sliding the path toward a free-from-obstacles

region, like sliding the path left or right until an expensive or unfeasible region is found.

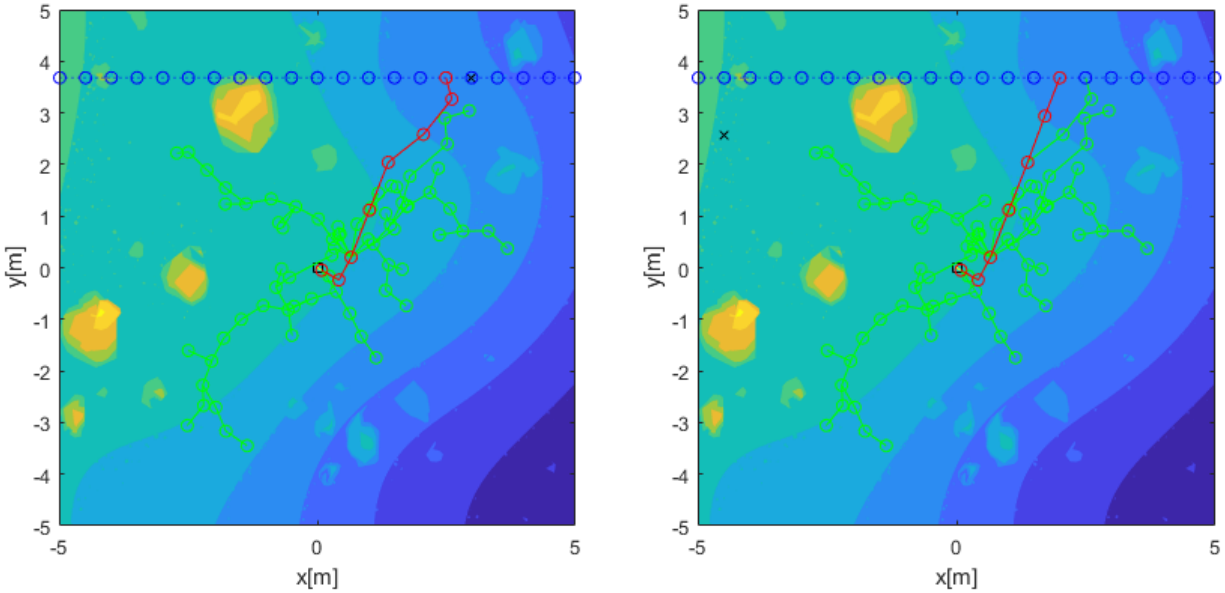
In detail, when a path is updated (Algorithm 4.1 line 10), two sets of points are generated as the offset, distanced of d_{extend}, to the left and to the right of the path points. When an iteration is selected to be for the optimization process (Algorithm 4.2 line 1), the vertex $\underline{\mathbf{v}}$ is selected as one of the offset points and the $d_{connect}$ is increased (doubled) to help the new point to not just connect to the point from which has originated. The algorithm starts from the points of the left offset, from the one created by the first path point, and proceeds toward the last. Each time a cheaper path is found, the offset are redefined and restarted from the first offset point (Algorithm 4.1 line 10).
When the algorithm evaluates the last point of the left offset set of points, founding no better path, it concludes that is pointless to move toward left, and disables the use of the left offset set, starting using the right one. As for the left one, once the right has reached the end, the algorithm is considered terminated.

Must be noted that the optimization process runs parallel to the exploration one. A "dice is rolled" each iteration and it determines if the new vertex must be generated as for optimization rules or for exploration rules. For example, if $\sigma_{opt} = 0.5$, the algorithm will alternately create an optimization point and an exploration point. If the σ_{opt} is 0, then the optimization process is not performed and the algorithm is stopped at the first path found.
In fig. 4.5 is shown an example of the optimization process.



(a) First path found.

(b) First improvement.



(c) Second improvement.

(d) Final improvement.

Figure 4.5: RRT*: optimization example. In green the tree, in red the best path found.

The initial position is in the center pointing toward south-east. Can be seen that the code start trying pushing the path toward left, making the path the straighter possible. The result is much straighter respect to the initial path, with just one fundamental turn in the path. The result shown, is at the end of the left phase. After is no more possible to push it further left, it start pushing right; but since in this case would cause a longer path, the algorithm quickly discard the option.

4.1.4. Trajectory post processing

After the path planning is done, the output must be refined, to add turns to the trajectory and to re-estimate the cost. The turns are added with the objective of being with the highest possible curvature radius, so to reduce the cost (see chapter 4 introduction).
First the corners are cut. For each corner (vertex) of the path, two points are generated on the adjacent edges at distance from the corner that is half of the shortest edge (following the reasoning in section 4.1.1), removing the vertex at the corner.
Then, the new generated points couples are connected with an arc of circumference passing through the points and tangent in them. The resulting trajectory is sampled each $L_{sampling}$ obtaining an array of (x, y, ϑ) points.
For each element of this list the cost function is run (chapter 3), obtaining a new estimation of the cost accounting for turns and a list of robot parameters in each point of the

trajectory.
An example of post processing output can be seen in fig. 4.6, where is shown a detail from the post processing of the path in fig. 4.5

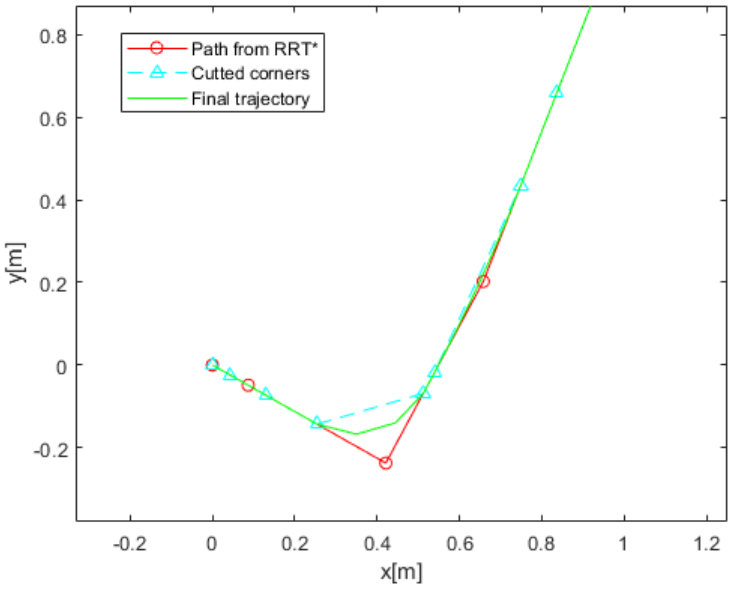


Figure 4.6: RRT*: post processing example (detail).

4.2. RRCT*

The RRT* shown in section 4.1, is able to find a solution, but with several limitations. It does not actually evaluates turns, thus even if the vertex are built to not generate curvature radius under a certain value, the turns themselves are not actually accounted for. The cost estimated by the post processed path is significantly different, than the one obtained from the RRT* optimization. That means that, while running the path planning, the optimization function will constantly have a big estimation error, reducing the capability to select the optimal path. Furthermore, the feasibility granted by the feasibility function is limited to the geometry of the path, but does not account for the terrain. Since not actually evaluated, the curves generated by the RRT* are not only much more expensive, but might be even unfeasible for the terrain. The robot is generally able to overcome the feasibility problems, but that cause a massive consumption, some unsafe motion and, anyway, does not grant the safety margin needed in space applications.

An algorithm able to evaluate the curves must be used, but the algorithm shown in section 2.1.4, has some limitations too. In particular:

1. It is not a RRT* algorithm, so it cannot asymptotically optimize the path.
2. It adds a third degree of freedom of the sampling state (ϑ), increasing the complexity.
3. It cannot explore the curvature radius state, it is in fact limited to the minimum feasible radius, that is exactly what is less desirable (chapter 4 introduction). And adding a degree of freedom on the curvature would mean adding other 2 degrees of freedom on each point or edge.
4. It is generally used with a pseudo distance $d = \sqrt{\Delta x^2 + \Delta y^2 + \Delta \vartheta^2}$ [18], that is nor a geometric distance, nor comparable to a consumption estimation. There is a concern about the way it generates the points and compute the edges. Each point is connected to another through an independent edge, evaluating several curves and straight lines even for similar paths. This is not a problem using the said pseudo-distance, but might be a very relevant problem for using a much expensive cost function, as done in this work. A way to bundle and recycle computed edges might be helpful.

In an attempt to tackle this problems, an alternative algorithm called RRCT* has been drafted.

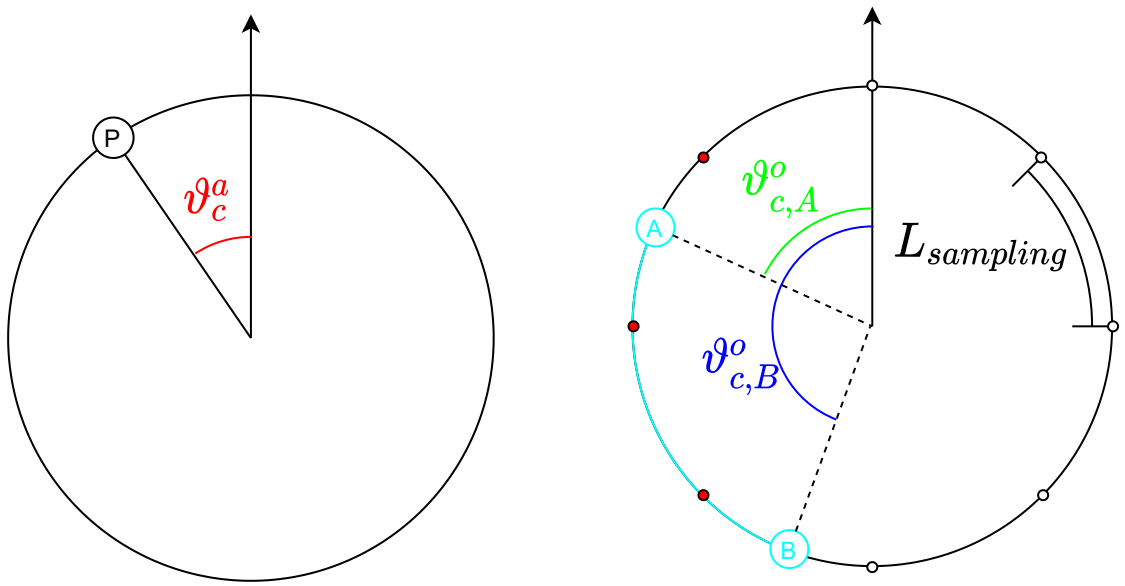
4.2.1. Base idea

The main idea of RRCT* consists in splitting the tree used to generate the path, from the actual paths of the robot. A tree is built similarly to the RRT* concept, but instead of generating points, it generates circles. Those circles works like roundabouts, and the paths are computed as passages between the roundabouts. In this way, the turns are defined and evaluated, plus, passing on the same roundabout, the cost to perform a curve is already available, recycling the turn cost.
A first tree is used just to build the set of roundabouts, while a second tree is the path tree as for the RRT*. In this case, the sampling adds elements to the circles tree, and the path tree is developed, with some constraint, onto the first one.

4.2.2. Circles definition

For the purpose of the RRCT*, the Circle object must be defined.
A circle is characterised by a position expressed as **center position** $\underline{\mathbf{c}} = [x, y]$ and by a **circle radius** R. Furthermore, since they are used to express a motion, they are oriented, so they can be skirted only clockwise or counterclockwise.
Each point belonging to the circumference of the circle, can be expressed by a single pa-

rameter ϑ_c^a, representing the central angle, oriented counterclockwise between a reference axis (in this work the vertical one), and the point on the circumference (fig. 4.7a). Another angle can be defined as ϑ_c^o, that is as ϑ_c^a, but its orientation is the same of the circle orientation.
The circle is sampled with points along its circumference at each distance $L_{sampling}$ (fig. 4.7b). These points defined with ϑ_c^o angle, can be evaluated with the cost function (chapter 3) and can store the resulting robot parameters, COTs and cost to reach the next point on the circle. When the an edge between two vertices A and B must be created on the circle, computing ϑ_c^o of A an B, the sampled point on the circles affecting the path can be easily identified, and the cost of the edge can be evaluated knowing the COT of each sampling point (fig. 4.7b). If the values of a sampling point, are already known, the cost function is not called, an interpolation of the cost on the arc is enough. In this way, the cost function is called only if a point is requested for a computation and maximum one time for each point. Generating new connections on the same circle, is in this way very cheap.



(a) Circle angle.

(b) Circle sampling points.

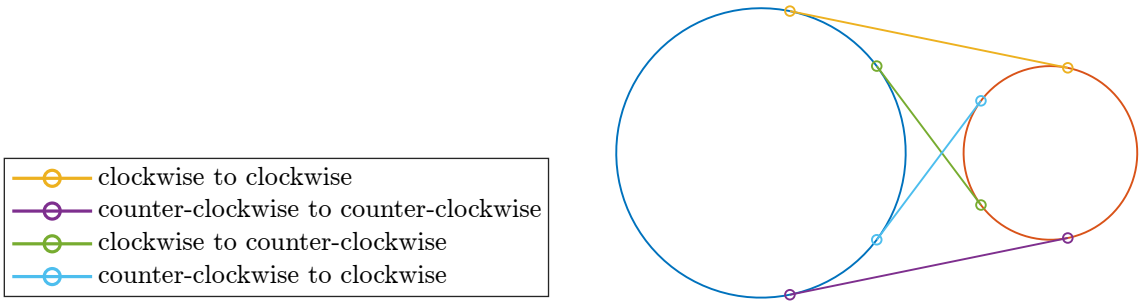
Figure 4.7: RRCT*: circle.

4.2.3. Tangent connections between oriented circles

The best way to connect two geometrical circles, is through common tangents, so to have a smooth transition between circumference arcs and straight lines.
Two circles have up to four tangents. Depending on their radius and reciprocal distance,

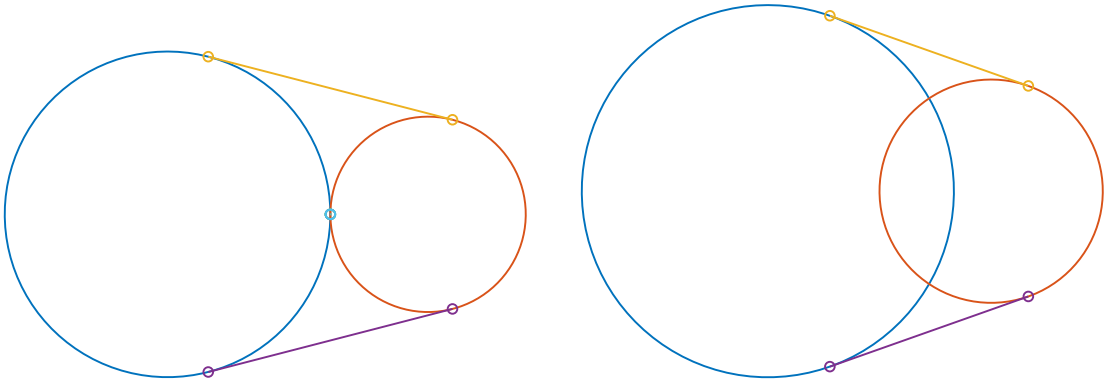
five configurations are possible. Defining the radius of the bigger as R_{big}, the one of the smaller one as R_{small} and the distance between the centers D (fig. 4.9), their summation and difference are $\Sigma R = R_{big} + R_{small}$ and $\Delta R = R_{big} - R_{small}$.

- $D > \Sigma R$, the two circles are external to each other (fig. 4.8a), 4 tangents exist, of which two internal and two external.
- $D = \Sigma R$, externally tangent circles (fig. 4.8b), the internal tangents degenerate in a point.
- $\Delta R < D < \Sigma R$ secant circles (fig. 4.8c), and the internal tangents are unfeasible.
- $\Delta R = D$ internally tangent circles (fig. 4.8d), also the external tangents degenerate in a point.
- $\Delta R < D$ one circle is internal to the other (fig. 4.8e), no feasible tangents.



(a) External circles.

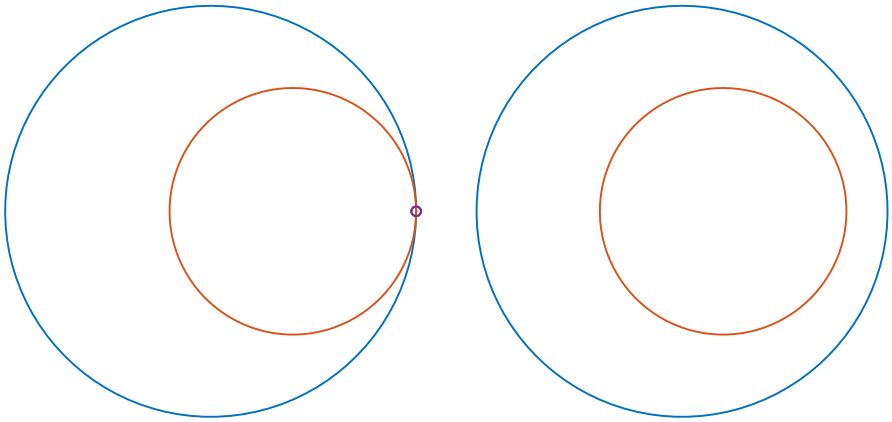
Figure 4.8: Tangents between circles pt1.



(b) Externally tangent circles.

(c) Secant circles.

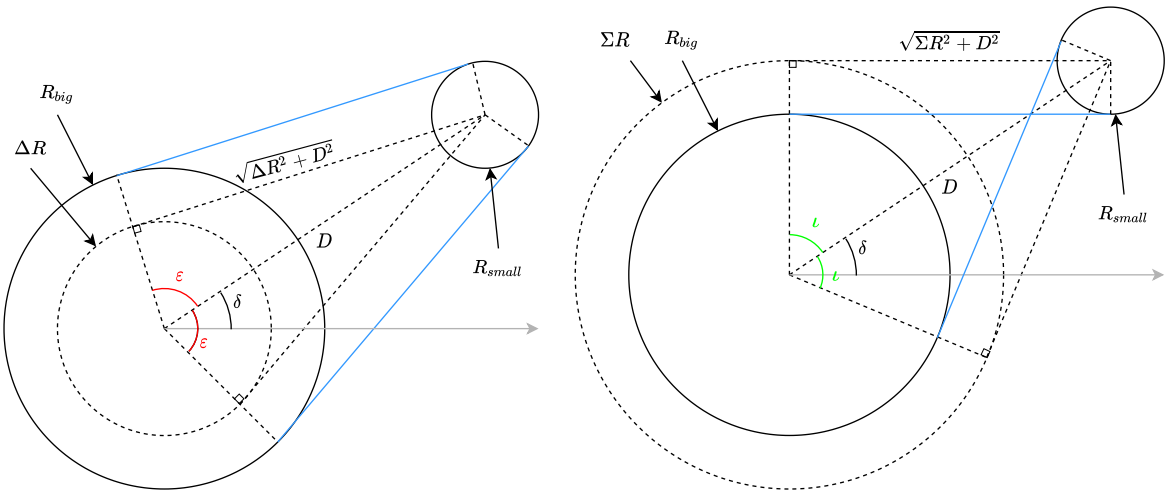
Figure 4.8: Tangents between circles pt2.



(d) Internally tangent circles. (e) Internal circles.

Figure 4.8: Tangents between circles pt3.

The tangent points between two circles can be easily computed in the form of ϑ_c^a (section 4.2.2).
ϑ_c^a is very easy to compute drawing two supports circles, concentric to the bigger circle and having radii ΔR and ΣR (fig. 4.9). The tangents to this circles passing to the small circle center, have tangent points with the same ϑ_c^a as the tangents to the two circles. Furthermore, for the external tangents the angle will be the same on both circles, while for the internal ones a π must be added (fig. 4.9).



(a) External tangents construction. (b) Internal tangents construction.

Figure 4.9: Tangents construction.

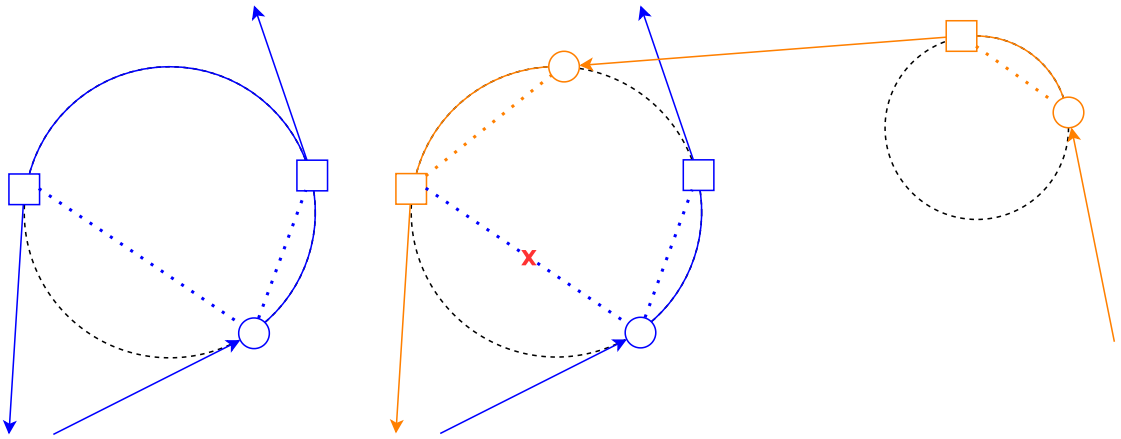
If the circles are considered oriented (section 4.2.2), can exist maximum one tangent segment that can crate a transfer from the starting circle to the arrival one (fig. 4.8). If

the tangent for the transfer is one of the unfeasible ones, then the transfer is unfeasible. If the tangents for the transfer degenerates in a point, then, in that point, the ϑ_c^a can still be computed in the same way, and the resulting zero length segment will be the transfer point between the two circles.

4.2.4. Vertices connection

Seen as two circles can be connected through tangents, to actually generate a path, must be connected the vertices on the circles.

The vertices of the path tree can belongs to three categories: **Entry points**, **Exit points** and **Target points** (fig. 4.10). The entry and exit points must belong to a circle. Entry points are connected to exit points of the same circle through a curved edge. Exit points are connected to Entry point of different circles, through a straight edge. Target points are as in section 4.1 and, exit points are connected to them through a straight edge.

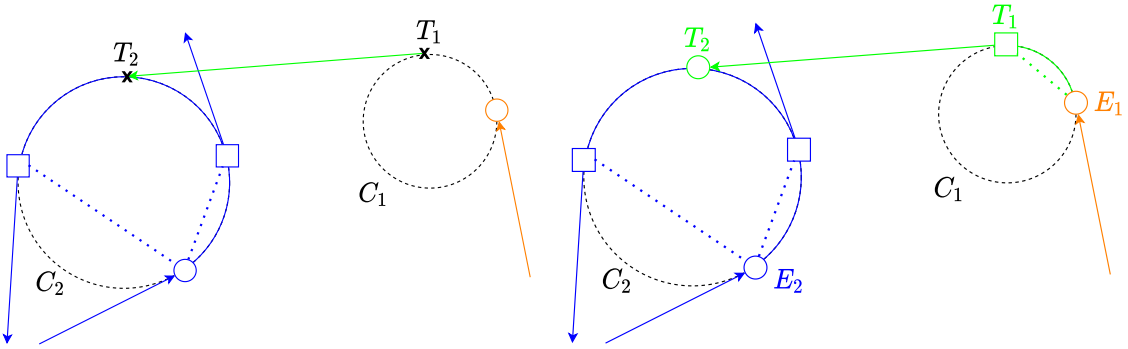


(a) Path example. (b) Adding a new circles and rerouting.

Figure 4.10: RRCT*: example. Round: entry point, square: exit point, blue: original path, orange: rerouted path, dotted lines: tree relation edges.

In this way each circle might be populated by several vertices, but each time a tangent connection between two oriented circles is attempted, since exist only one possible solution, only the connection between two vertices must be evaluated (section 4.2.3). Plus, edge parts on the circles might be partially in common with other connections (es fig. 4.10), and thus there is no need to reevaluate them (section 4.2.2).

Given two oriented circles C, to connect from C_1 to C_2, they can be connected with maximum one tangent segment (or degenerate point). This segment and the tangent points can be computed as seen in section 4.2.3, obtaining the ϑ_c^a (fig. 4.11a). Then the first precedent entry point on C_1 can be identified and called E_1, same on C_2 obtaining E_2 (fig. 4.11b). An exit point will be generated in the tangent point T_1 on C_1 and an entry point will be generated in the tangent point T_2 on C_2 (fig. 4.11b). A curved edge is evaluated between E_1 and T_1 (as in section 4.2.2), and a straight edge between T_1 and T_2 (as in RRT* section 4.1) (fig. 4.11b).



(a) Tangent segment and points identified.

(b) Exit and entry points of interest identified and path between circles generated.

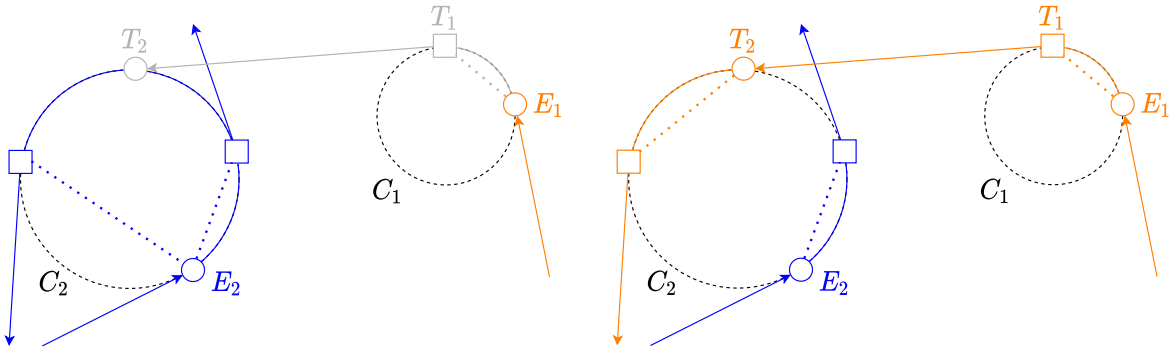
Figure 4.11: Circles connection: point definition.

If T_2 is the only entry point of C_2 the path is immediately generated, otherwise the curved edge between E_2 and T_2 is computed too, and the \$-cost to reach T_2 passing through E_1 and E_2 are compared. If the first is higher than the second, the path is not improving, thus is deactivated (fig. 4.12a). If the second is the higher one, than the new path is activated and all the children exit nodes of E_2 after T_2 and before E_2 are passed to T_2 (new path) (fig. 4.12b).

Then is estimated the cost between T_2 and the next entry points on C_2, if any is cheaper to rach from T_2, all its children are passed to T_2 and the relative path deactivated (fig. 4.13b). The first entry point to not be rerouted blocks the procedure (fig. 4.13a).

All the descendant vertices of updated vertices are updated, and if a deactivated branch is updated, it performs an evaluation like this one in C_2, checks if the deactivated entry point is now a cheaper way. If it is not, keeps the branch deactivated, if it is, reactivate the branch and updates the children fig. 4.14.

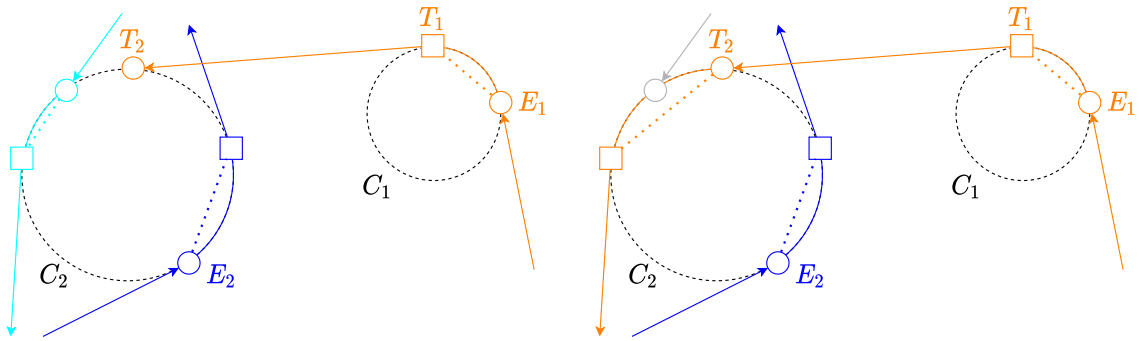
When the connection must be performed between a circle C_1 and a target node $t_2 = [x, y]$, the procedure it the same, except that the connection is tangent to C_1 and passing through



(a) The new path is less efficient, deactivated.

(b) The new path is more efficient, activated and children updated.

Figure 4.12: Circles connection: $-costs comparison with previous entry point.



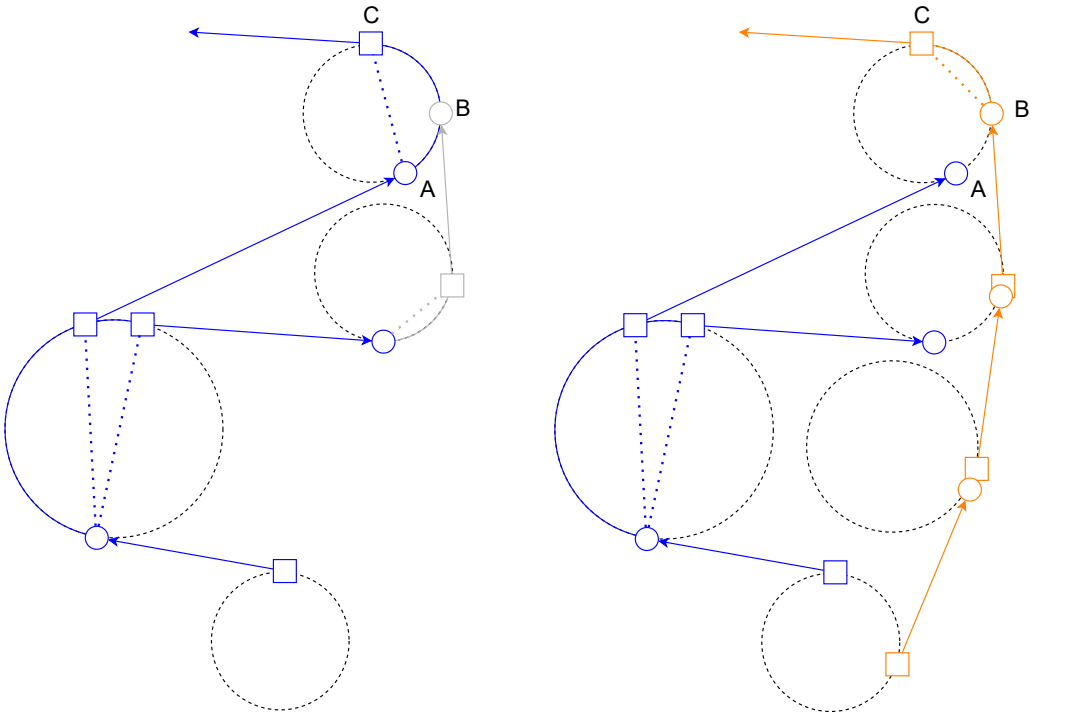
(a) The new path is less efficient.

(b) The new path is more efficient, old deactivated and children updated.

Figure 4.13: Circles connection: $-costs comparison with next entry points.

t_2 (see fig. 4.9a concept) and the others paths to compare are connected directly into t_2 and not to a circle.

Is clear that a counterclockwise circle can be approached to steer left, while a clockwise circle to steer right. Nevertheless is possible to steer also in the opposite direction by performing a loop. It is very unlikely that a loop in a path is going to reduce consumption, therefore the loops must be avoided. The rule to avoid them is simply to forbid the connection of an entry point to an exit point, if the center angle between them is higher than π.



(a) The path to reach C through B is less efficient than through A, thus is inactive.

(b) Path change before B, make the path through B more efficient, thus is reactivated.

Figure 4.14: Circles connection: path reactivation.

4.2.5. Algorithm

Conceptually the algorithm (Algoritm 4.3) is similar to the RRT* (Algorithm 4.1), but with building blocks seen in section 4.2.2, 4.2.3, 4.2.4. The algorithm will explore also the curvature radius dimension, therefore is defined a sampling space for R_c bounded by an $R_{c,min}$ and an $R_{c,max}$. $R_{c,min}$ is the generally a limiting value as described before, while $R_{c,max}$ is determined by usefulness. Is true that the sharper the turn the more expensive it is, but a turn is meant to change the direction of the robot, thus too big radii of curvature add complexity while not generating better paths. Therefore $R_{c,min}$ and $R_{c,max}$ are input parameters for the algorithm.

Since it operates with circles, but start from the initial state $\underline{\mathbf{i}} = [x, y, \vartheta]$, the initial position $\underline{\mathbf{i}}$ must be added to some circle to start the algorithm. In Algorithm 4.3 line 1, 6 starting oriented circles are defined as tangent to the oriented point $\underline{\mathbf{i}}$ and with radii $R_{c,min}$, mean($R_{c,min}$,$R_{c,max}$) and $R_{c,max}$. 3 on the left of the point oriented clockwise, 3 on the right oriented counterclockwise (fig. 4.16a).
Then, at each iteration new circles are generated (line 3). More in detail (Algorithm 4.4)

Algorithm 4.3 RRCT*

```
1: Define initial circles
2: while Searching do
3:    C = Generate new circles
4:    Try to connect C to the tree
5:    if Connection success then
6:       Try to reroute some paths through C
7:       Try to connect C to target nodes
8:    end if
9:    if Any target node $ changed then
10:      Update path
11:   end if
12: end while
13: Retrieve path
```

as for RRT* if the iteration is an optimization one, the generation is optimization based (see section 4.2.6), if is an exploration one it generates $\underline{\mathbf{x}}$ identically to RRT* and generate a random curvature radius R. The distance with all the other circles is measured. This can be done in three ways, by computing the center-center distance, by computing the circumference-circumference distance or by computing the distance along the tangent segment. The result of this work have been computed using the circumference-circumference distance.

Obtained the distances, the closest circle can be obtained and from its center is extended the new circle center $\underline{\mathbf{c}}$. Two new circles are generated in the new center position $\underline{\mathbf{c}}$ having radius R and opposite orientation. Created the new circles, both independently follow a procedure like in section 4.1 (Algorithm 4.3), so they are connected to circles in range, first from neighbouring circles to the new ones, then the opposite and finally to targets points, updating the path if a new minimum is found. The differences with RRT* are in the connection procedures described in section 4.2.2, 4.2.3, 4.2.4. Furthermore, the circles are actually just a support for the vertices. So, while the vertices itself can have just a parent, and so a tree shape, the circles will appear to have multiple parents and to be parent and child of the same circle.

To avoid a too high complexity is not shown, but if no circle can create a curved edge to start, a path to reach the new circles, probably the circle used to generate the new circles, is "hard to leave" (maybe there is an obstacle on the circumference), therefore to avoid it to stuck the code, with the same $\underline{\mathbf{x}}$ other circles are generated from the second nearest circle (and so on).

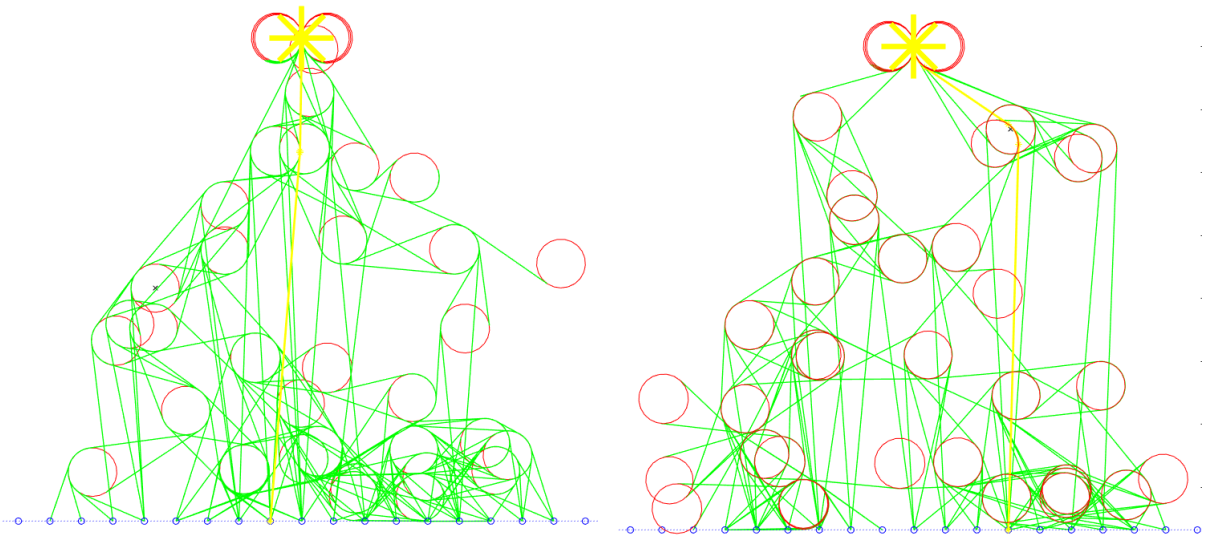
After the end of the iterations, the path is retrieved. This time, since RRCT* stores all

Algorithm 4.4 Generate new circles

```
1: if Optimization iteration then
2:    C = Optimization process
3: else
4:    if Target point as random point then
5:       x = Select random point as target
6:    else
7:       x = Random point in range
8:    end if
9:    R = Random radius in range
10:   N_closest = Get closest circle of the tree to x
11:   d_closest = Get x-N_closest distance
12:   if d_closest < d_extend then
13:      c = x
14:   else
15:      c = Extend new center from N_closest center
16:   end if
17:   C = Circles with center in c with radius R
18: end if
```

the parameters computed and generate fully feasible paths, no post processing is needed.

The algorithm shape on some terrains is prone to look like a particular tree fig. 4.15.



(a) (b)

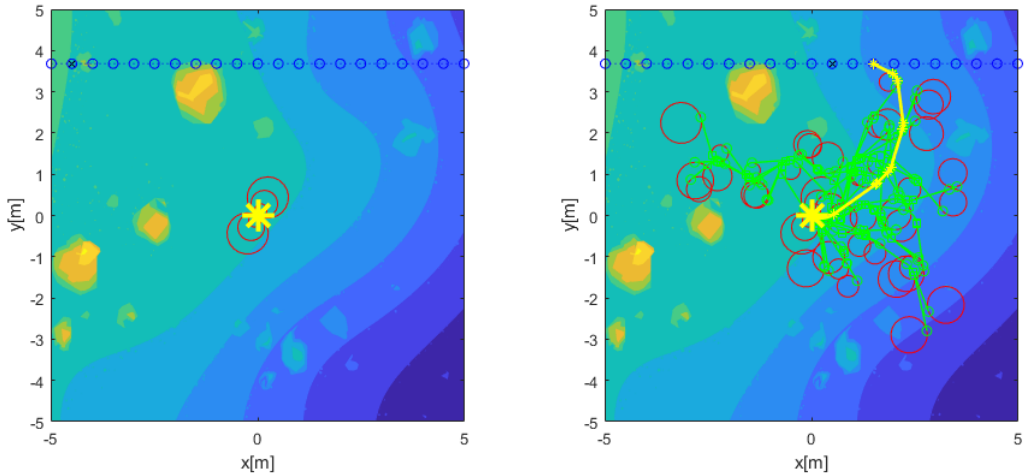
Figure 4.15: RRCT*: on a flat plane.

Therefore, this variant of the Rapidly-exploring Random Trees is called **Rapidly-exploring**

Random Christmas Trees.

4.2.6. Optimization process

The idea is similar to the optimization process of section 4.1.3, with some modifications. This optimization works in two ways: by adding circles or by connecting circles.
First it tries to connect differently path circles: all circles traversed in the optimal path are identified (both them and they opposite twin) and, starting from the first one, is attempted a direct connection to the nephew. In other words, for node ordered 1st, 2nd and 3rd, a direct connection from 1st to 3rd is attempted. If it generates a new path and it is cheaper, the path is updated. This is performed for all circles until the second last, that has no nephews, then the optimization type changes.
The second optimization type works similarly to section 4.1.3 one, but perform a simple oversampling. A path trajectory is computed as the union of all the Dubin states $\underline{\mathbf{q}} = [x, y, \vartheta]$, belonging to the straight edges of the path. For each of this trajectory points is computed the number of circles at distance closer than d_{extend} on the left side and on the right side. If the number is lower than a input value n, a circle is added in the region. This is performed for each point of the path trajectory, until all of them satisfy the condition. This impose the creation of circles around a straight line, opening to alternative paths that moves around expensive terrain.
After both the optimizations have finished, the iteration are terminated.

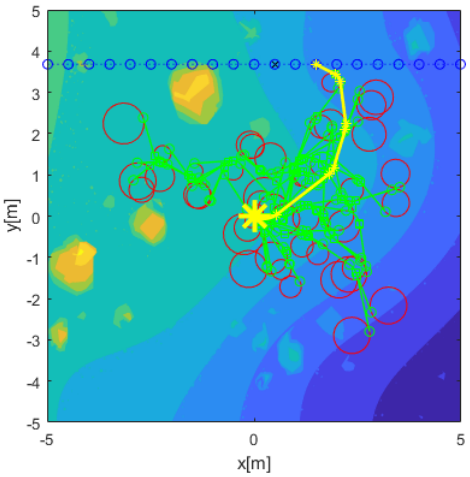


(a) Starting circles generated. (b) First path found.

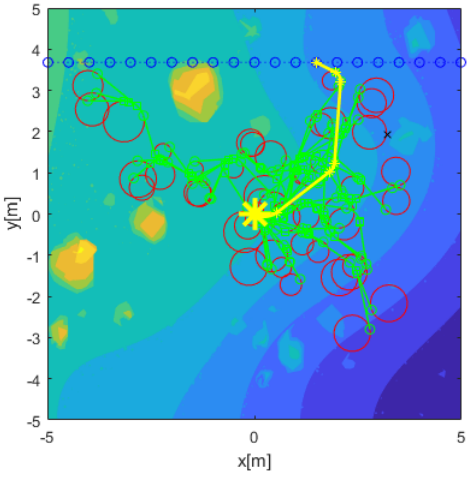
Figure 4.16: RRCT*: search.

In fig. 4.17 can be observed an example of optimization process. Starting from the first path found in fig. 4.16b it first tries connecting circles (fig. 4.17a, 4.17b, 4.17c), making

the path a lot more straight.

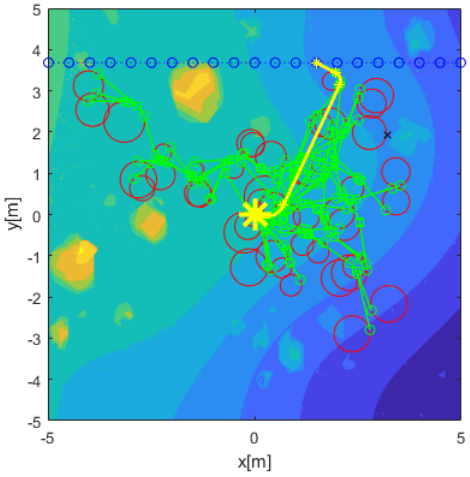


(a) Optimize reconnecting circles.

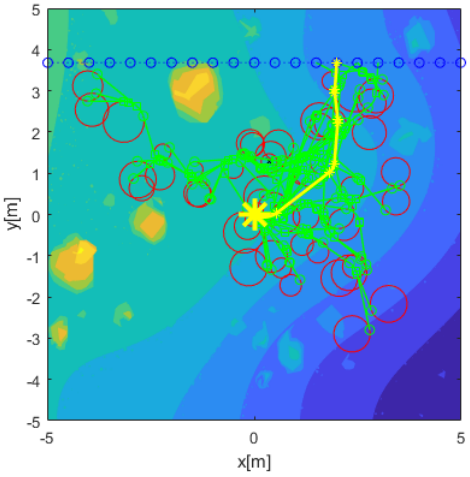


(b) Optimize reconnecting circles.

Figure 4.17: RRCT*: Optimization pt1.



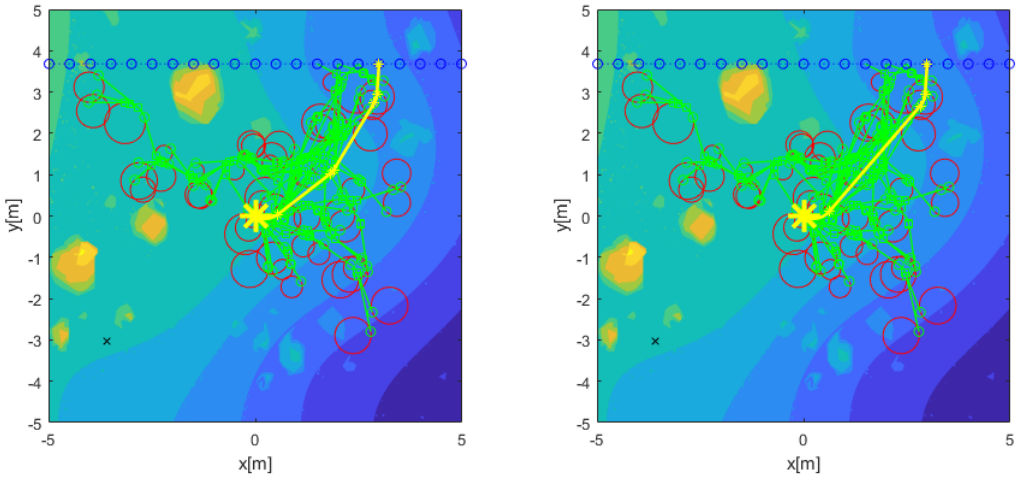
(c) Optimize reconnecting circles.



(d) Optimize adding circles.

Figure 4.17: RRCT*: Optimization pt2.

Then it tries to add circles nearby the path to search for different ways. And in fig. 4.17d and fig. 4.19a an alternative path is found, as expected since the terminal part of the path in fig. 4.17c is intuitively not efficient.
Every time a new path is found, the optimization path restart again, obtaining the final path in fig. 4.19b.



(e) Optimize adding circles. (f) Optimize reconnecting circles.

Figure 4.17: RRCT*: Optimization pt3.

4.3. RRHT*

In section 4.1 and section 4.2 have been shown the RRT* algorithm and its evolution in RRCT* to include the steering evaluations. The RRCT* still presents some limitations. In particular, for it to evaluate big radii of curvature, long straight edges must be present. In fact, big radii of curvature with small straight edges, would mean that the circles does intersect, and if it happens, the tree start taking the shape of a palm tree fig. 4.18.

This happens, because, starting from the initial circles, each new circle couple, being secant to an oriented circle, will have just on possible connection. And adding the clause that forbid loop connections, it will be oriented to steer the path as for circle direction. This is an extreme problem, since it cause the path to steer when there is no need, like when the target is in front of the robot on a flat ground in fig. 4.18.
This limitation, while with easy or medium terrain complexity is good, since reduce the complexity of the path and its cost; is a problem on hard terrain complexity, since it forbid complex paths with several small trajectory adjustments, or at least is slower at evaluating them.
To overcome this limitation, a variant of the RRCT*, the RRHT* has been quickly drafted for a preliminary evaluation.

The RRHT* algorithm is identical to the RRCT* except in the generation of the circles. The idea is, working in the conditions that cause the palm behaviour, to add a tangent

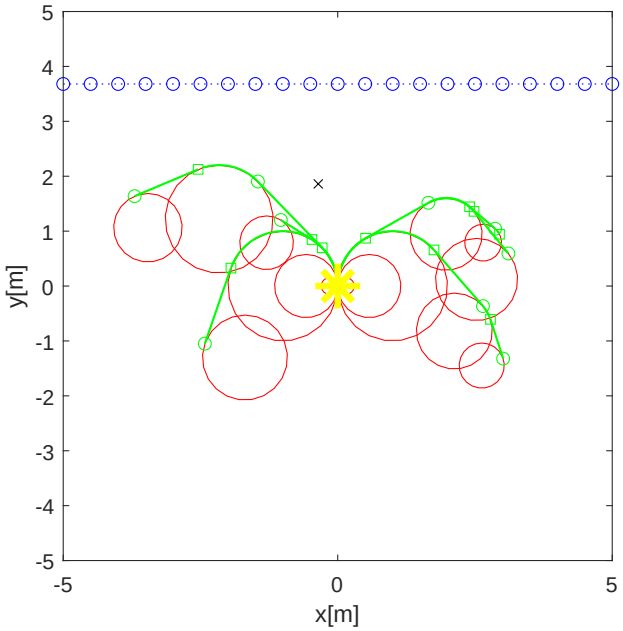


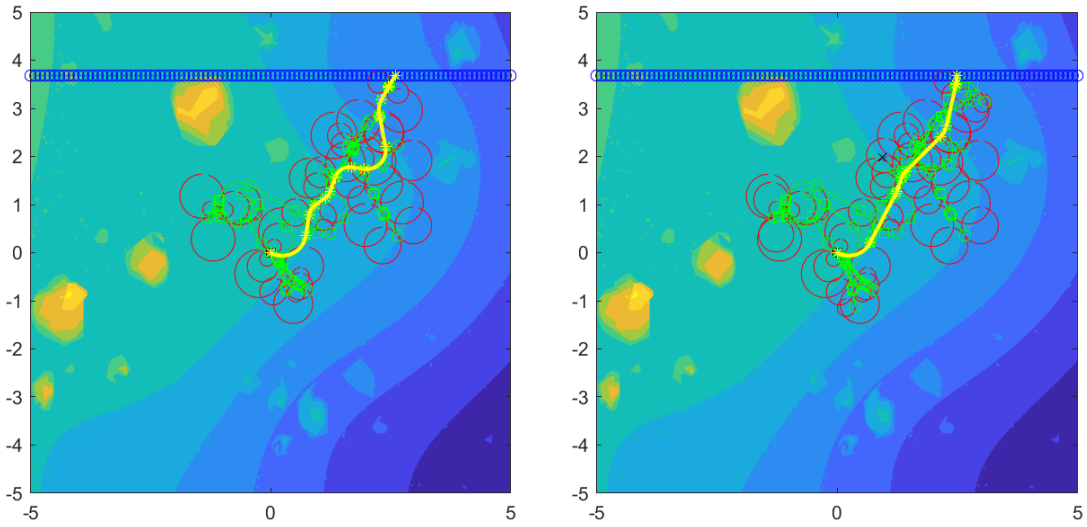
Figure 4.18: Palm behaviour.

circle, in the opposite direction, that counteract the will of the algorithm to steer in a single direction.

Practically, it is performed by adding a modification in Algorithm 4.4 after line 17. After the concentric circles $\underline{\mathbf{C}}$ have been extended from the tree, the two can be connected with tangents to the circle that originated them. Since they are secant, only one circle will have a feasible connection, this circle will be maintained while the other will be discarded. A new circle can be generated from the remaining $\underline{\mathbf{C}}$, having same radius, opposite orientation and being placed to be tangent to $\underline{\mathbf{C}}$ in the tangent point found before. The two circles are connected to each other through a degenerate segment, so to allow the passage from one to the other, and the algorithm can continue as for RRCT*. Note that the tangent connection evaluation, is just to build the new circles, is not creating any connection to the path tree.
Some path result can be seen in fig. 4.19.

This algorithm, probably, has the behaviour of a Dubin curve based RRT, since it can be seen as generating sampling points that can be connected by Dubin curves. It adds a rerouting, making it an RRT*, possibility to explore different curvature radii, and recycling of the cost of the curves to reach a point.

Therefore should be possible to directly use re-adapted Dubin curve based RRT, as algorithm for path planning.



(a) First path. (b) After some optimizations.

Figure 4.19: RRHT*: example.

For the RRHT*, the algorithm shape, looks like an holly three fig. 4.20. Hence its name **Rapidly-exploring Random Holly Trees**.

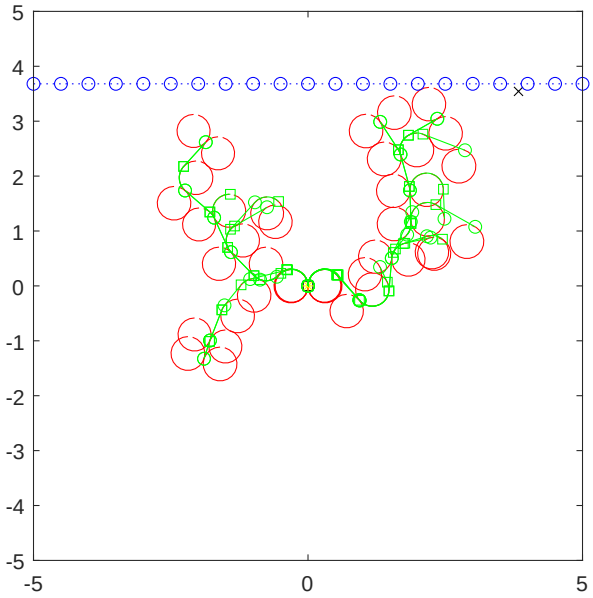


Figure 4.20: RRHT* on a flat plane.

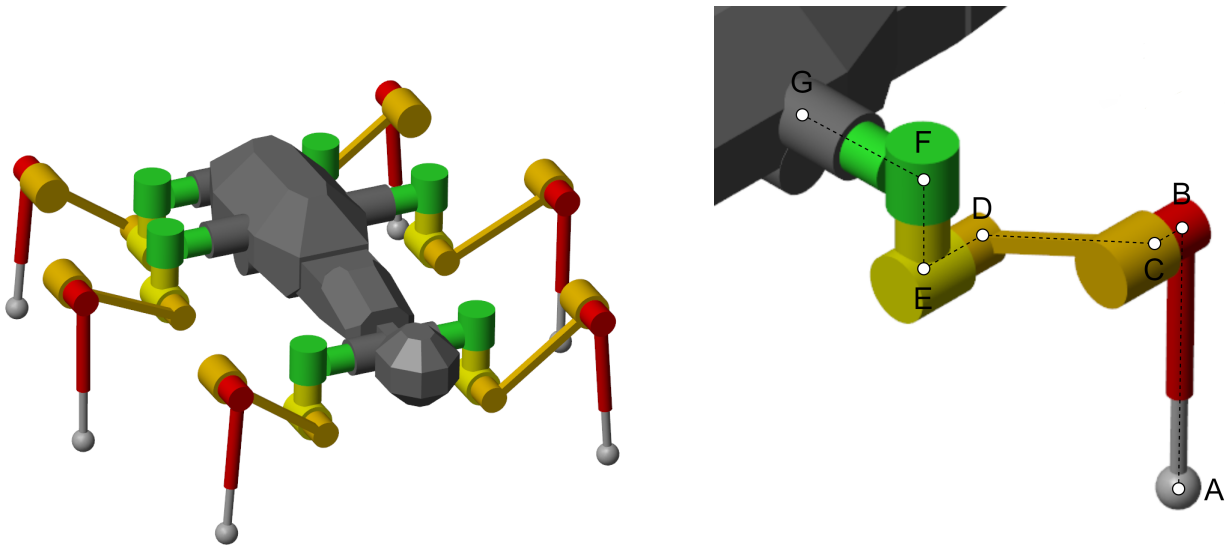
5 | Simulation framework

After the thesis has been tackled from the theoretical point of view, in a general way; the discussion must be reduced to a particular case to perform simulations, experiments and validate the procedure.
The model built in this chapter, will be used both to estimate the performance of the particular robot, and to validate the estimations performed by the path planner.
Most of the discussion in this chapter will be valid for a particular robot and simulation instrument, since is meant to clarify the instruments used to validate the path planner. That does not mean, that the path planner works only for this robot and models.

5.1. Robot mechanical model

The robot mechanical model is strongly based on the SpaceClimber rover [19, 20]. The idea is to have a robot with performance similar to a rover meant for space, so to have a more fitting analysis. This does not imply, that other legged robots cannot use the path planning discussed, but the thesis is focused on space rovers.
Note: the data taken from the rover are approximate, because some were missing. Since the objective is just to have a reasonable rover, they are not a problem. This work is not meant to be a reference for SpaceClimber.
The rover is an hexapod robot with insect-like posture (fig. 5.1a), the configuration is as the most common for these robots, with the exception of a fourth joint (lean joint) in G (fig. 5.1b).

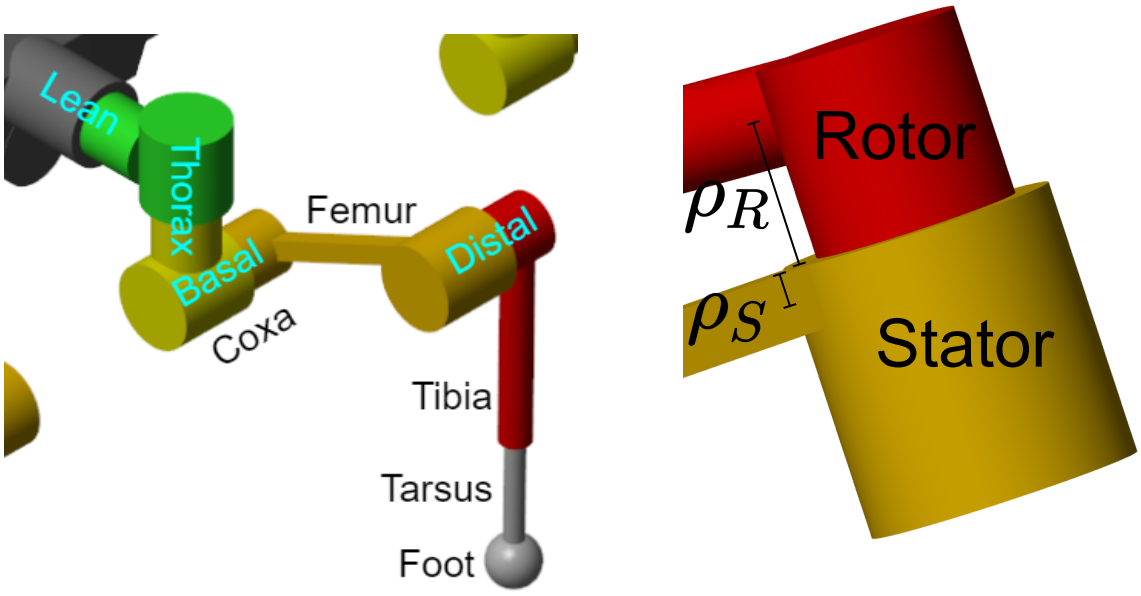
The robot would originally have a body joint and a prismatic joint between tarsus and tibia (fig. 5.2a), both have been deactivated making the body and the tibia+tarsus a single rigid body.
The foot is shaped like a sphere, tibia and tarsus like cylinders, the femur like a prism, and the actuators are modeled as two cylinders with an adjacent faces, that can rotate relatively, around their axis (fig. 5.2b).
In the model the solids intersects. In particular: tarsus ends are positioned at the foot center and inside the tibia, tibia ones are at the distal rotor center and encapsulate the



(a) Rover isometric view. (b) Leg detail.

Figure 5.1: Rover geometry.

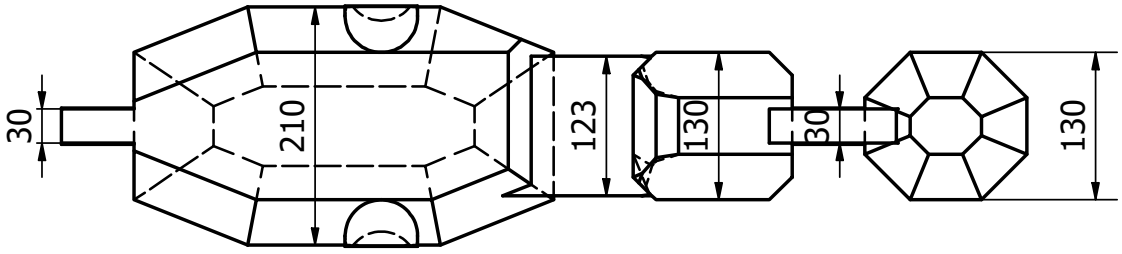
tarsus, while the femur ends are in distal stator and basal rotor fig. 5.2a. Furthermore, each actuator is connected side-way to a bone end or another actuator end. The distance at which this component is positioned from the joint face fig. 5.2b, is defined as ρ, S for the stator and R for the rotor.
A list of the measures of the body can bee seen in fig. 5.3 and table 5.1, while in table 5.2 are reported the masses.



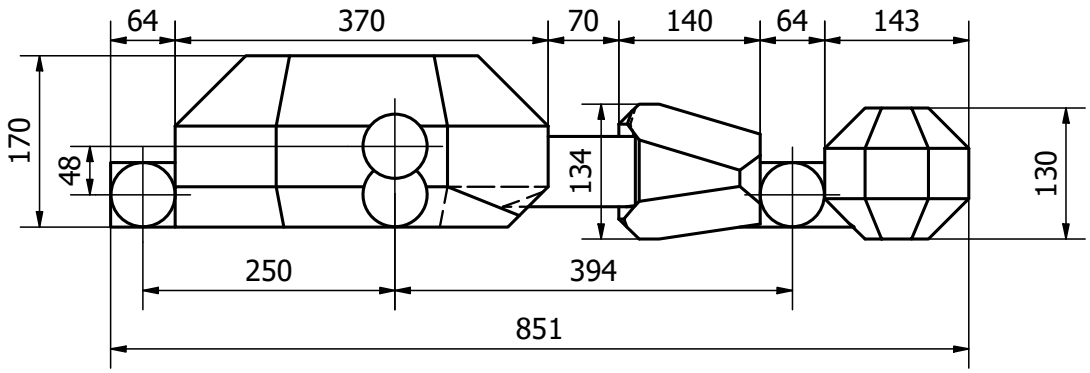
(a) Leg nomenclature: bones in black, joints in light blue.

(b) Joint actuator nomenclature.

Figure 5.2: Legs nomenclature.



(a) top view.



(b) Side view.

Figure 5.3: Rover body measures in mm.

Measure	Value [mm]
Foot radius	20
Tarsus length	142
Tarsus radius	15
Longer tarsus*	10
Tibia length	174
Tibia radius	24
Tarsus-Tibia penetration	52
Femur length	225
Femur width	15
Femur thickness	15
Actuator stator length	65
Actuator stator radius	64
Actuator rotor length	45
Actuator rotor radius	48
$\rho_{\mathbf{R}}$ **distal**	21
$\rho_{\mathbf{S}}$ **distal**	7.5
$\rho_{\mathbf{R}}$ **basal**	37.5
$\rho_{\mathbf{S}}$ **basal**	36
$\rho_{\mathbf{R}}$ **thorax**	(tangent on face)
$\rho_{\mathbf{S}}$ **thorax**	22
$\rho_{\mathbf{R}}$ **lean**	(tangent on face)
AB	264
BC	40.5
CD	225
DE	73.5
EF	99
FG	142

*The middle legs have a longer tarsus, to partially compensate the higher lean joint.

Table 5.1: Rover lengths parameters. See section 5.1 fig. 5.1b, 5.2a, 5.2b. [19, 20, 21]

Part	Value [g]
Body	8 000
Stator	378
Rotor	147
Femur	80
Tibia	133
Tarsus	107
Foot	80
Totsl	23 000

Table 5.2: Rover mass parameters.[19, 20, 21]

5.2. Robot mechanical boundaries

As discussed in section 3.5, the robots parameters cannot assume any value, but must belong to feasible combinations, respecting some constraints due to robot mechanics.

5.2.1. Stride height

To make sure the foot leaves and touches the ground without sliding, the top point of the swing trajectory must be the highest point (fig. 5.4). Therefore, assuming the terrain to have a constant slope:

$$h_{stride} \geq \frac{L_{stride}}{2} \tan |\alpha_|| + \delta_{SH} \tag{5.1}$$

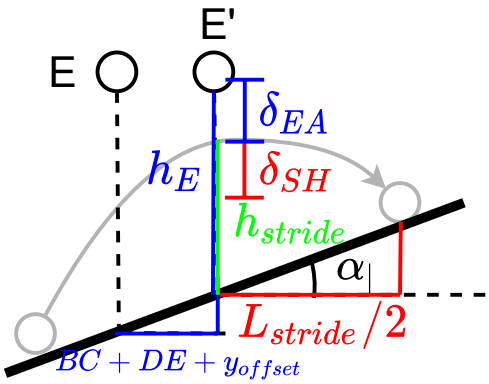


Figure 5.4: Foot trajectory on slope, reference frame solidal in E. In red quantity referred to stride height, in blue referred to E height.

Where δ_{SH} is the margin of how much the top point must be higher than any contact point.
In this work $\delta_{SH} = 1cm$.

5.2.2. E height

The foot is considered not allowed to go above the basal joint (point E), because that would cause a too quick motion of the leg and the distal actuator, to follow the foot trajectory. The E height h_E must grant this condition both while the foot is on the ground with a lateral slope (fig. 5.5a), and when it is swinging especially at the top point (fig. 5.4). Therefore, defining $x_{foot,middle}$ as the x position in RH of the foot of the middle leg:

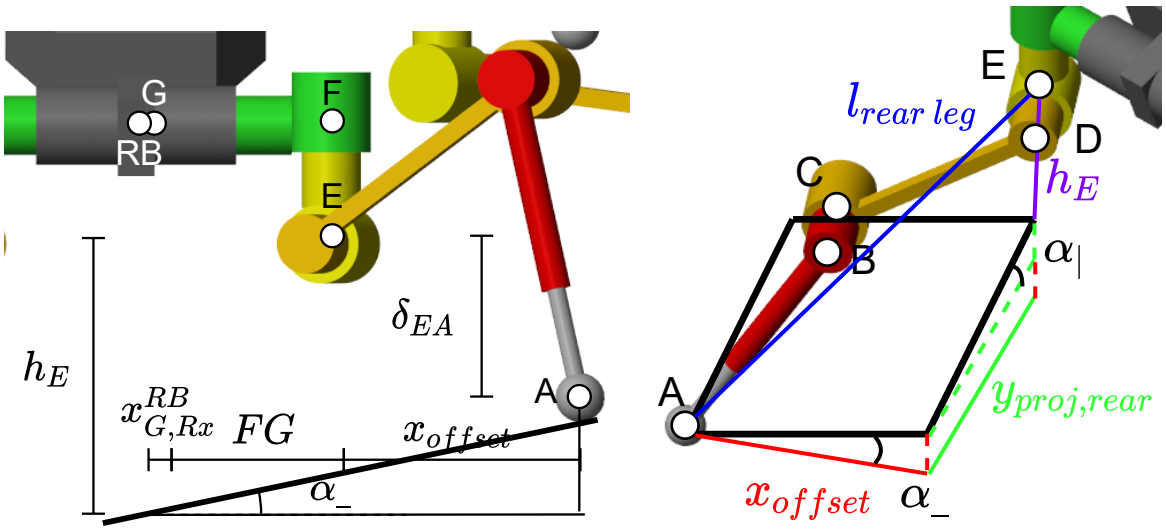
$$x_{foot,middle} = x_{offset} + FG + x^{RB}_{G,R2} \tag{5.2}$$

$$h_E \geq x_{foot,middle} \tan |\alpha_\_| + \delta_{EA} \tag{5.3}$$

$$h_E \geq (+(BC + DE) + y_{offset}) \tan \alpha_| + h_{stride} + \delta_{EA} \tag{5.4}$$

$$h_E \geq (-(BC + DE) + y_{offset}) \tan \alpha_| + h_{stride} + \delta_{EA} \tag{5.5}$$

Where $x^{RB}_{G,R2}$ is the x position of the G point of the R2 leg in RB and δ_{EA} is the margin between A and E heights to be satisfied.
In this work $\delta_{EA} = 1cm$.



(a) Lateral slope. (b) Extended rear leg.

Figure 5.5: E height boundaries.

It also must be ensured that h_E is low enough to guarantee the legs to be able to touch the ground. Therefore defining $l_{middle\,leg}$, $l_{rear\,leg}$ as approximate lengths of the middle and rear legs; $y_{proj,middle}$, $y_{proj,rear}$ as the distance along y between A and E; and $x_{foot,rear}$ as the x position of the G point of the R2 leg in RB (fig. 5.5b).

$$l_{middle\,leg} = AB + CD + \text{longer tarsus} \tag{5.6}$$

$$l_{rear\,leg} = AB + CD \tag{5.7}$$

$$y_{proj,middle} = \left(\frac{L_{stride}}{2} + BC + DE\right) + y_{offset} \tag{5.8}$$

$$y_{proj,rear} = -\left(\frac{L_{stride}}{2} + BC + DE\right) + y_{offset} \tag{5.9}$$

$$x_{foot,rear} = x_{offset} + FG + x^{RB}_{G,R3} \tag{5.10}$$

$$h_E \leq \sqrt{l^2_{middle\,leg} - x^2_{offset} - y^2_{proj,middle}} - x_{offset} \tan|\alpha\_| + y_{proj,middle} \tan\alpha_| \tag{5.11}$$

$$h_E \leq \sqrt{l^2_{rear\,leg} - x^2_{offset} - y^2_{proj,rear}} - x_{offset} \tan|\alpha\_| + y_{proj,rear} \tan\alpha_| \tag{5.12}$$

5.2.3. Forward velocity

When the rover is lifting a leg, it generates a torque on its body. If the lift speed is significantly high, the torque might be high enough to cause oscillations in the rover, even detaching some foot from ground. This happens at high forward velocities, with low t_{swing} (high β), small L_{stride} and high h_{stride}; generating frequent and quick leg rise and fall. Therefore, being the other parameters defined by the environment, the velocity must be limited:

$$v_{forward} \leq \left(\frac{1}{\beta} - 1\right) \frac{L_{stride}}{h_{stride}} \nu \tag{5.13}$$

This derives remembering

$$t_{swing} = (1-\beta) t_{cycle} \tag{5.14}$$

$$t_{stance} = \beta t_{cycle} = \frac{L_{stride}}{v_{forward}} \tag{5.15}$$

and assuming a limiting parameter

$$\nu = \frac{h_{stride}}{t_{swing}} = \frac{h_{stride}}{(1-\beta) t_{cycle}} = \frac{h_{stride}}{(1-\beta)\dfrac{L_{stride}}{\beta v_{forward}}} \tag{5.16}$$

Must be noted that in this work has been used a wrong formula (eq. (5.13) with a plus in place of the minus and with $\nu = 0.0167$). While this is a problem, because introduce unacceptable parameters in the COT matrix analysis; it is counteracted by the simulations to compute the COT matrix, that fails, creating NaN COT elements, that will be discarded by the cost function, not affecting the results.

5.2.4. Curvature radius

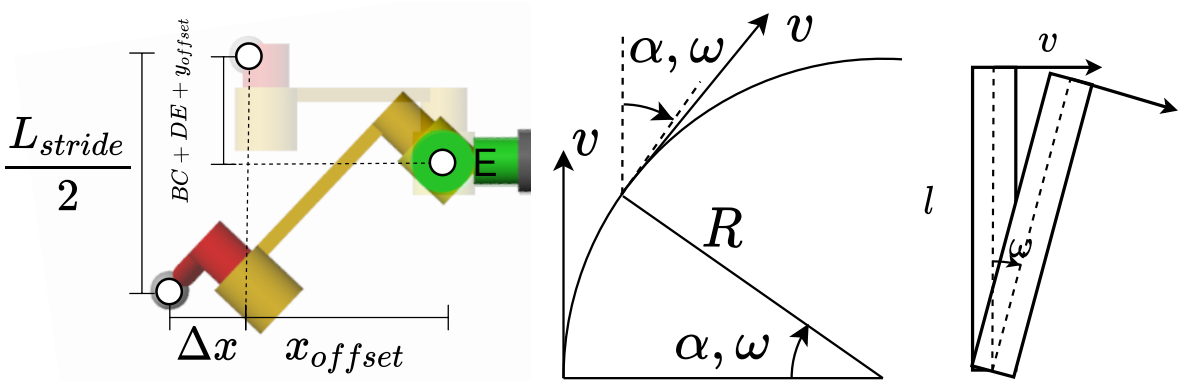
While steering, the objective is to not reach the mechanical limit of the legs in extension, and to not hit the body in contraction. The rotation, for how the controller is built, happens around the z_{RH} axis, therefore the legs experiencing the biggest lateral motion, are the front ones. The most limiting condition is the extension of the leg, while passing

from a linear trajectory to a curve. In fact, in this condition the leg is placed in about the middle of its operative range along x, and so has just half of the max lateral motion available (fig. 5.6a and similarly to fig. 5.5b). The limits of the allowed lateral distance can be approximated with:

$$l_{front\,leg} = AB + CD \tag{5.17}$$

$$y_{front\,rear} = -\frac{L_{stride}}{2} + BC + DE + y_{offset} \tag{5.18}$$

$$\Delta x_{max} = \sqrt{l_{front\,leg}^2 - h_E^2 - y_{front\,rear}^2} - x_{offset} \tag{5.19}$$



(a) Steering top view. (b) Cicular motion. (c) Beam rotation.

Figure 5.6: Steering boundaries.

Then, to compute the minimum radius: the forward velocity is defined and on a circular motion (fig. 5.6b)

$$v_{forward} = \frac{L_{stride}}{t_{stance}} = R_c \omega_{yaw} \tag{5.20}$$

assuming the body as a bar (fig. 5.6c), the lateral velocity at the front legs joints will be about

$$v_{lateral\,front} = y_{R1}^{RB} \omega_{yaw} \tag{5.21}$$

and to obtain it, the foot must have a lateral velocity of

$$v_{lateral\,front} \approx \frac{\Delta x}{t_{stance}} \tag{5.22}$$

obtaining

$$R_c \geq R_{min} = y_{R1}^{RB} \frac{L_{stride}}{\Delta x_{max}} \tag{5.23}$$

This value is margined by 20% in the path planning algorithms, to make sure the limit is avoided.

The slope are not accounted for, because, if the slope is low is negligible, if is not, the robot rotation will put it into an high transversal slope condition, making the motion already unfeasible since there are very strict limits on it.

5.3. Robot window bands definition

As described in section 3.3 the sliding window is made of different bands. For the benchmark rover they are defined as follows:

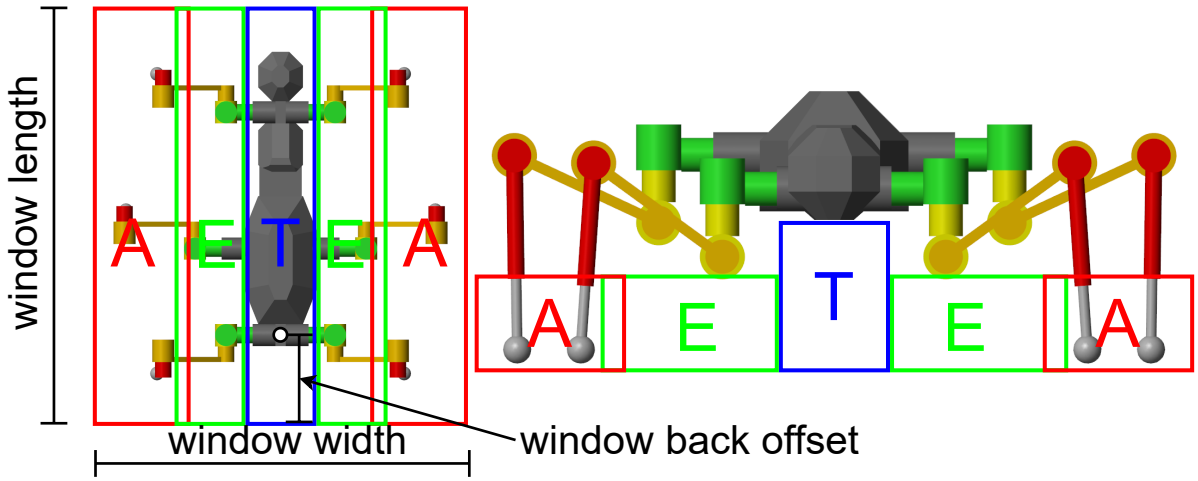


Figure 5.7: Sliding widow parameters and subdivision in bands.

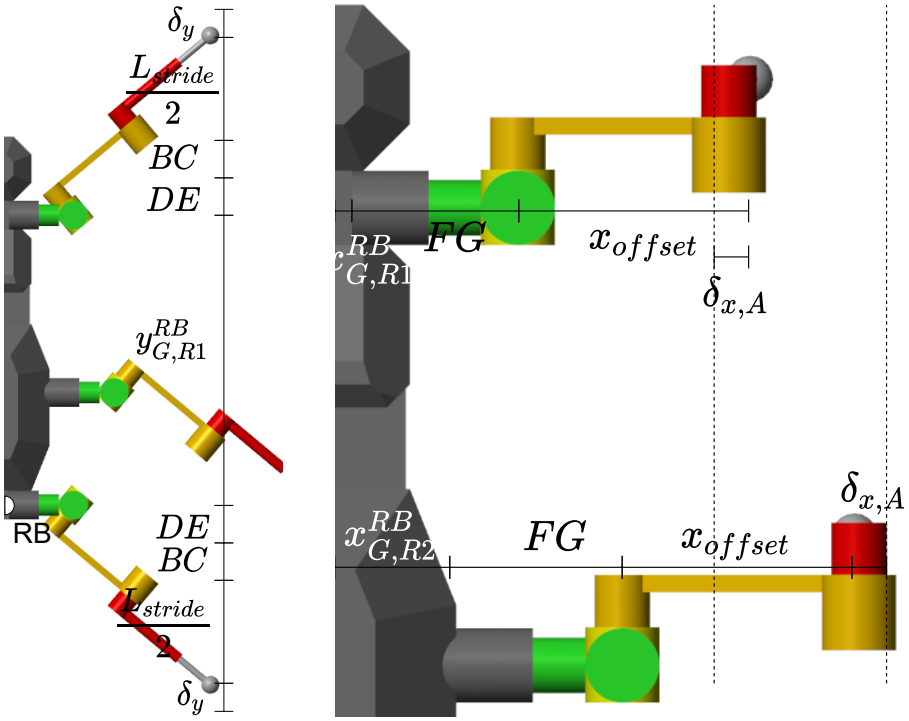
Their limits along y are the same for all the bands and are determined by the furthest distance a foot can get forward and backwards plus a margin $\delta_y = 5cm$ (Must be remembered the reference frame is RH).(fig. 5.8a)

$$y_{min} = -\left(DE + BC + \frac{L_{stride}}{2} + \delta_y\right) \tag{5.24}$$

$$y_{max} = y^{RB}_{G,R1} + DE + BC + \frac{L_{stride}}{2} + \delta_y \tag{5.25}$$

The width of an E-band is determined by the position the basal joint can reach (fig. 5.9b). The internal boundary is determined by the front and rear legs basal joints, while the external limit by the middle legs ones(fig. 5.9b). Defining l_{edge} as the longest distance from E to any point of the basal joint actuator (fig. 5.9a):

$$l_{edge} = \max\left(\sqrt{R^2_{actuator,big} + (L_{actuator,big} - s_E)^2}, \sqrt{R^2_{actuator,small} + (DE + w_{DE}/2)^2}\right) \tag{5.26}$$



(a) Bands length. (b) A-band.

Figure 5.8: Bands definition.

$$x_{E,min} = x^{RB}_{G,R1} + FG - l_{edge} - \delta_{x,E} \tag{5.27}$$

$$x_{E,max} = x^{RB}_{G,R2} + FG + l_{edge} + \delta_{x,E} \tag{5.28}$$

with margin $\delta_{x,E} = 1cm$.

Similarly, the width of an A-band is determined by the positions the feet can reach (fig. 5.8b).
In straight path:

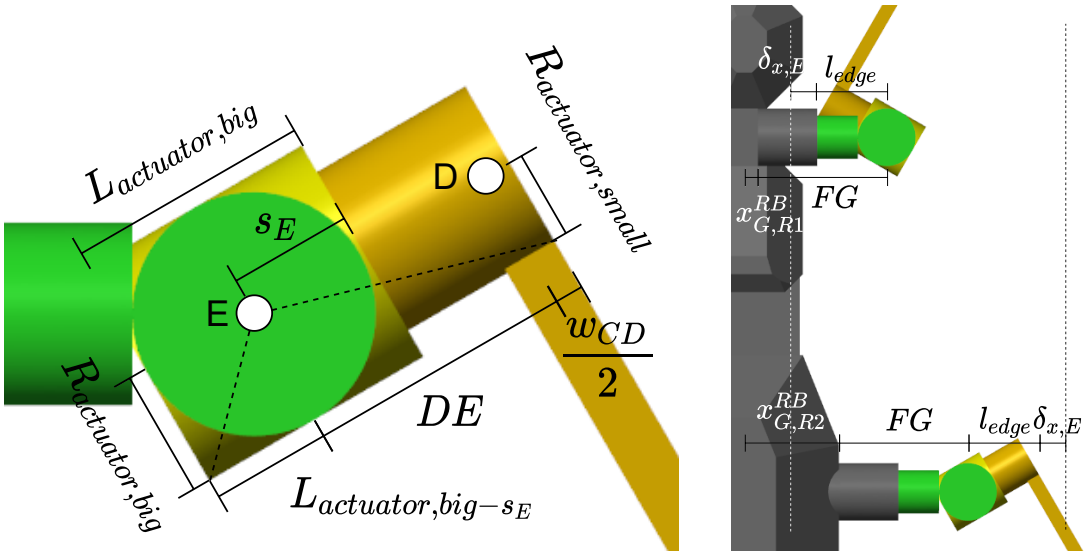
$$x_{A,straight,min} = x^{RB}_{G,R1} + FG + x_{offset} - \delta_{x,A} \tag{5.29}$$

$$x_{A,straight,max} = x^{RB}_{G,R2} + FG + x_{offset} - \delta_{x,A} \tag{5.30}$$

While steering is assumed directly the maximum allowed condition for any curvature radius $x_{lim,up}, x_{lim,down}$.

$$x_{A,steering,min} = x^{RB}_{G,R1} + FG + x_{offset} - \delta_{x,A} + x_{lim,down} \tag{5.31}$$

$$x_{A,steering,max} = x^{RB}_{G,R2} + FG + x_{offset} - \delta_{x,A} + x_{lim,up} \tag{5.32}$$



(a) Basal actuator. (b) E-band.

Figure 5.9: E bands definition.

Finally, the T-band will be defined comprised between the two E-bands.
The resulting values can be see in table 5.3.

	Lower limit [mm]	Upper limit [mm]
y	-289	933
$\mathbf{x_E}$	63.5	342.5
$\mathbf{x_{A,straight}}$	308	498
$\mathbf{x_{A,steering}}$	118	655

Table 5.3: Band limits result.

Bands heights can be defined, considering them compatible with $H_{obstacles}(x, y)$ in section 3.4. So, as vertical distances from a plane with same $\alpha_|$ of the terrain fitting plane. The bands height will be the maximum height an obstacle can have in the band.

Therefore, the A-band will be simply of height h_{stride} accounting for a margin:

$$h_{A-band} = h_{stride} - \delta_{z,A} \tag{5.33}$$

The other parameter used in the cost function is the height of the E point h_E, but the criteria to pass an obstacle is the clearance between it and the basal joint actuator,

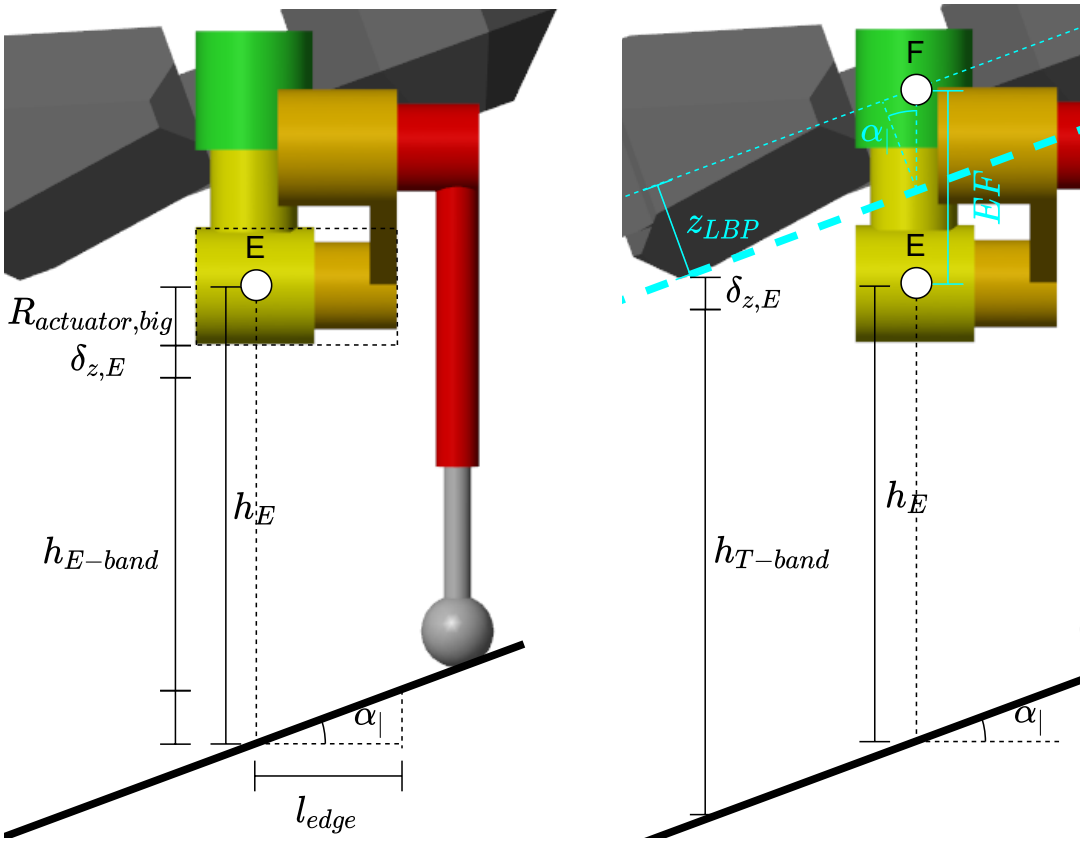
therefore the actuator dimensions must be accounted for (fig. 5.10a):

$$h_{E-band} = h_E - \left(R_{actuator,big} + l_{edge} \tan |\alpha_|| + \delta_{z,E}\right) \tag{5.34}$$

Similarly for the T-band, where a line parallel to the slope can be traced form the lowest point of the torso (fig. 5.10b), and its distance along z_{RH} is used as h_{T-band}:

$$h_{T-band} = h_E + EF + \frac{z^{RB}_{LBP}}{\cos |\alpha_||} - \delta_{z,E} \tag{5.35}$$

Where the margins are $\delta_{z,A} = 1cm$, $\delta_{z,E} = 1cm$ and the lowest point of the body in RB frame $z^{RB}_{LBP} = -44mm$.



(a) E-band height. (b) T-band height.

Figure 5.10: Bands heights.

Alternatively, the criteria on h_{stride} and h_E can directly be written, using the maximum obstacle height in the band $H_{max,band}$:

$$h_{stride} > H_{max,A} + \delta_{z,A} \tag{5.36}$$

$$h_E > H_{max,E} + R_{actuator,big} + l_{edge} \tan |\alpha_|| + \delta_{z,E} \tag{5.37}$$

$$h_E > H_{max,T} - \left(EF + \frac{z_{LBP}^{RB}}{\cos |\alpha_|} \right) + \delta_{z,E} \tag{5.38}$$

5.4. Robot mechanics simulator

The robot is simulated in Simulink using Simscape Multibody. Several components are introduced, modeled using built-in blocks of the simulator. The rover is thus modeled as a series of rigid bodies, connected by rotational joints. To make a simpler simulation, the joints are controlled imposing the kinematic (angles and derivatives), instead of dynamically (imposing torques); in this way the simulation of the motor is avoided. Nonetheless the joints torques are computed by the simulator, to provide in post processing a measurement of the energy consumption (section 5.7).

If most of the component blocks were built-in in Simscape, the simulation of the contact force with the terrain had to be built for the simulation, since the built-in contact force block can just simulate convex hulls, while a terrain is generally highly concave.
The simulation of the block used in this work, is a re-adaptation of the stick-slip model in [22] for the use on a terrain heightmap $Z(x, y)$.

The center position of the foot $\underline{\mathbf{p}}_A$ is determined by the simulator, and considering the contact point to be in the same (x,y), its z can be determined interpolating the heightmap $Z(x, y)$ and the local slopes in x and y interpolating the slope-maps along x and y $S_x(x, y)$ $S_y(x, y)$. This approximation is not true, since higher the slope further the contact point will be from the center of the foot, but has been considered acceptable to improve the speed and numerical stability of the simulation.
Determined the contact point and the versor normal to the terrain, the penetration depth (p_{foot}) and speed ($\mathring{p}_{foot}$)of the foot can be determined.
The reaction force equation is

$$F_{reaction} = kp_{foot} + b\mathring{p}_{foot} \tag{5.39}$$

with $F_{reaction}$ oriented exiting normal the terrain and p_{foot} and $\mathring{p}_{foot}$ entering it. kp_{foot}, the stiffness force, is present only if the the penetration is positive, so do not act if the foot is outside the terrain. $b\mathring{p}_{foot}$, the damping force, is present only if the velocity is entering the terrain, so to avoid an effect of the terrain dragging the foot when its lifting. The slip velocity (v_{slip}) at the contact point can be computed considering foot velocity

and rotation, and thus the friction force will be

$$F_{friction} = \mu(v_{slip})F_{reaction} \tag{5.40}$$

With $F_{friction}$ opposite to v_{slip} and μ determined as a function of the v_{slip} (fig. 5.11) can then be computed and applied to the foot.

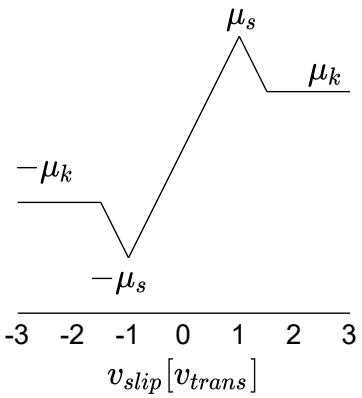


Figure 5.11: Stick-slip model.

The final contact force

$$\underline{\mathbf{F}}_{contact} = \underline{\mathbf{F}}_{reaction} + \underline{\mathbf{F}}_{friction} \tag{5.41}$$

The parameters of the contact force model have been selected to grant a reasonable terrain deformation and damping for a rigid terrain (k and b) and to grant the capability of the rover to climb the prescribed 40°limit degrees slope (μ_s, μ_d, v_{trans}) [19][23].
To greatly reduce the simulation time, due to ill-conditioning, the friction parameters have been doubled; and to evaluate if the rover has stepped on a terrain slope, that would cause slip, an analysis have been performed in post processing just based on the limiting slope of 40° (see section 5.10).
The gravity has been selected as the martian one, an all the parameters can be seen in table 5.4.

Parameter	Value
$k[\frac{N}{m}]$	11500
$b[\frac{Ns}{m}]$	1150
$\mu_s[\ \ ]$	2
$\mu_d[\ \ ]$	1.8
$v_{trans}[\frac{m}{s}]$	0.01
$g[\frac{m}{s^2}]$	3.721

Table 5.4: Mechanical simulation parameters.

5.5. Robot controller model

For the robot to traverse a virtual terrain successfully, and so perform a simulation until the goal, the controller must be able to handle different situations.
The controller programmed in Simulink generates a main motion using parameters as described in section 2.2.1, 2.2.2, 2.2.3, 3.2, but this motion is not robust to uncertainties, thus is aided with some reflexes and controls (fig. 5.12).

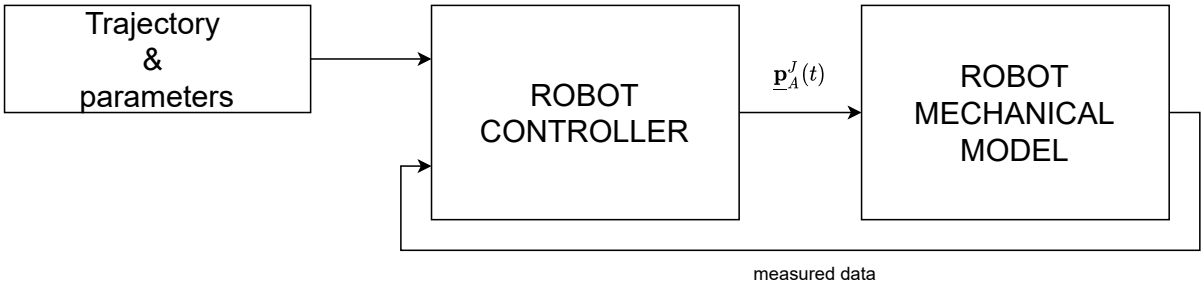


Figure 5.12: Simulation scheme.

5.5.1. Foot trajectory generator

The primary objective of the control system, is to generate the trajectory of each robot foot $\underline{\mathbf{p}}_{A,i}^{RH}(t)$ in swing and stance phase.

In swing phase (**swing controller**) the motion is generated as a 3rd grade polynomial. The starting point is the current position of the foot $\underline{\mathbf{c}}_A^{RH}$; the final position $\underline{\mathbf{f}}_A^{RH}$, is the rest position plus a translation along y of $y_+ = L_{stride}/2$, a z translation z_+ approximately accounting for the longitudinal slope, and a x translation x_+ determined by the steering

control (section 5.5.5).
Can be determined the vertical plane containing $\underline{\mathbf{c}}_A^{RH}$ and $\underline{\mathbf{f}}_A^{RH}$, and in this plane can be placed the top point of the trajectory $\underline{\mathbf{t}}_A^{RH}$, with x and y as mean of the two points and z to grant the h_{stride}.
The three points can be connected in the vertical plane with two polynomial arcs, imposing zero derivative at $\underline{\mathbf{t}}_A^{RH}$ and maximum slopes in the extreme points, avoiding stationary points in the arc.
The resulting curve will have a relation $z(x, y) = z(s)$, with s curvilinear coordinate on the xy-projection line connecting $\underline{\mathbf{c}}_A^{RH}$ to $\underline{\mathbf{f}}_A^{RH}$. Determining a velocity along the line

$$v_s = \frac{L_{projection}}{t_{swing}} \tag{5.42}$$

the z coordinate and the entire foot position $\underline{\mathbf{p}}_A^{RH}(t)$ time relation can be determined.

In stance phase (**stance controller**) the foot will follow a prescribed velocity $\underline{\mathbf{v}}_{stance}$, determined by the desired body linear and rotational velocities (section 5.5.3).

To this two basic motions are added two reflexes: one to search for contact and one to avoid obstacles.

The first one (**search controller**) works as the stance trajectory generator, but with a $\underline{\mathbf{v}}_{search} = [0, 0, -v_{search}]$ purely vertical. $v_{search} = 0.1\frac{m}{s}$.
This is done to improve stability: while others controllers prevents the start of new swing phases of other legs, to avoid an unexpected unstable position; searching as quick as possible a foothold, is needed to avoid the blocking of the rover and to reduce the time spent in this condition.

The second one (**avoidance controller**), instead use $\underline{\mathbf{v}}_{avoid} = v_{avoid}\underline{\widehat{\mathbf{n}}} - v_{y,stance}$, where $\underline{\widehat{\mathbf{n}}}$ is the versor normal to the ground at the contact point. $v_{avoid} = 0.05\frac{m}{s}$.
It allows, when hitting an obstacle, to move away from it, giving the foot room to rise without sliding on the rock. $-v_{y,stance}$ compensates the motion of the rover moving forward, so to avoid to go near to the obstacle, because dragged by the body. The clearance with the obstacle is fundamental for the type of simulation performed. Since the high attrition coefficients and the motion of the leg controlled kinematically, make a foot quick slide on a rock, able to generate massive forces, both dragging the robot in unwanted positions and generating unfeasible motor torques.

The use of this 4 trajectory generators is regulated by a **selector**.
If the phase pulse P, indicates that the leg is in swing phase, then use the swing controller, attempting to lift and lower the leg.

When the leg hits the ground, the swing phase is suppressed and the stance controller is selected.
If at the end of the swing phase the contact is still missing, the search controller is selected, vertically lowering the height of the leg until contact is found, then starting the stance phase.
If a contact is detected before the leg reached the top point, probably there is an obstacle on the path, thus the avoidance controller is selected for 0.2s to give space to the leg to lift without sliding on the rock. (fig. 5.13)

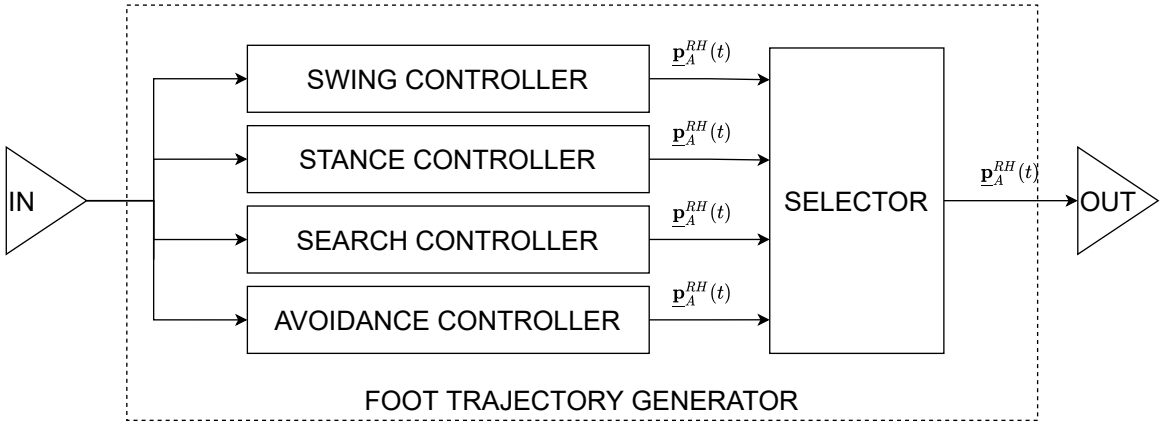


Figure 5.13: Foot trajectory generator.

5.5.2. Joints angles controller

While the foot trajectory generator in section 5.5.1 can generate a geometrical trajectory $\underline{\mathbf{p}}_A^{RH}(t)$ of the point A; the rover is actuated by revolute joints. Since the joints are actuated kinematically, the quantity to be provided to the joints are the angles of each joint and their derivatives. For this purpose, the obtained $\underline{\mathbf{p}}_A^{RH}(t)$ is converted to joint space $\underline{\mathbf{p}}_A^{J}(t) = [\varphi_D(t), \varphi_B(t), \varphi_T(t)]$ (Distal, Basal, Thorax) through an **inverse kinematic module**. Must be noted that the lean joint is not present, since is used just to keep the leg aligned vertically and is controlled separately.

A **direct kinematic module**, instead, performs the opposite operation, working as a sensor filtering unfeasible geometrical positions and providing measurement of the current position $\underline{\mathbf{c}}_A^{RH}(t)$ to the trajectory generator.

Another module (**limiting module**) validates the states in both spaces, and if one of the limits is reached the robot motion is blocked.

Bundled together, this modules compose the **leg controller** (fig. 5.14).

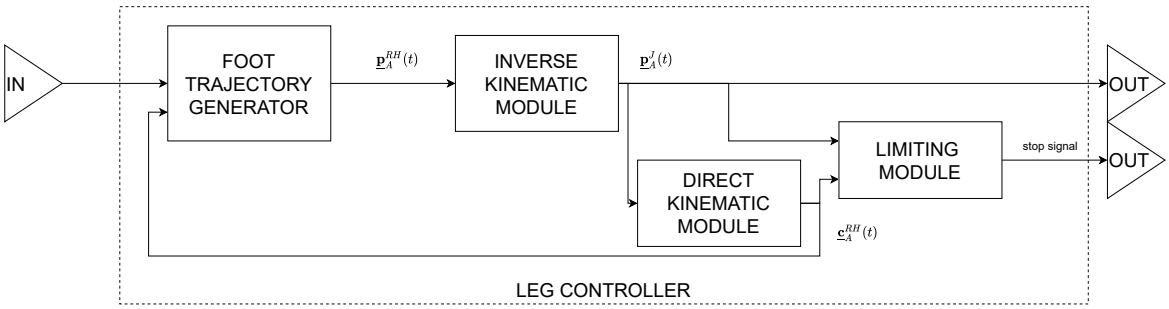


Figure 5.14: Leg controller.

5.5.3. Coordination controller

The 6 legs controllers (one for each leg), must be coordinated to generate a motion with the desired velocity, direction and rotations. To do so, a **coordinator controller** is used.

- Knowing the current position of the feet from motor positions, and being able to detect, if they are in contact or not, through a sensor in the tarsus-tibia prismatic joint; it can fit a plane through the contact points. This plane can be used to estimate the current h_E, the φ_{pitch} and $\alpha_|$. While the φ_{roll} can be computed measuring the gravity vector. From this values can be computed the errors and, using a proportional gain, the desired velocities.

$$\omega_{pitch} = P_{pitch}\left(\varphi_{pitch,desired} - \varphi_{pitch,measured}\right) \tag{5.43}$$

$$\omega_{roll} = P_{roll}\left(\varphi_{roll,desired} - \varphi_{roll,measured}\right) \tag{5.44}$$

$$v_z = P_z\left(h_{E,desired} - h_{E,measures}\right) \tag{5.45}$$

  With the body oriented parallel to the ground in pitch ($\varphi_{pitch,desired} = \alpha_|$), horizontal in roll ($\varphi_{roll,desired} = 0$) and gains $P_{pitch} = 1$, $P_{roll} = 1$, $P_z = 10$.

- The remaining angular velocity contribution is due to the steering. Using prescribed R_c and $v_{advance}$ the angular velocity can be computed.

- Including all this computed velocity contributions and the ones due to the prescribed $v_{forward}$ and $v_{lateral}$ (section 5.5.5), the velocity of the body can be obtained and, the velocity to be held by each foot in stance phase ($\underline{\mathbf{v}}_{stance}$) can be computed.

- A module selects the y_{offset} interpolating ID_{gait}, $\alpha_|$ and L_{stride} (see section 5.9).

- A module selects the height of each leg top point to grand the h_{stride}

- A module, while steering, estimates the lateral motion needed by the legs, and determine the x_+ in section 5.5.1, so to center the foot lateral motion in x_{offset}.
- It also aids the search controller (section 5.5.1), by changing the $\underline{\mathbf{v}}_{stance}$ to $[0, 0, v_{search}]$, to block the advancement and lower the body, if after 0.2s in search, a leg has not found a foothold yet.
- It imposes $\underline{\mathbf{v}}_{stance} = 0$, if any leg reaches the limits.

This controller plus the 6 leg controllers compose the **legs motion generator** (fig. 5.15).

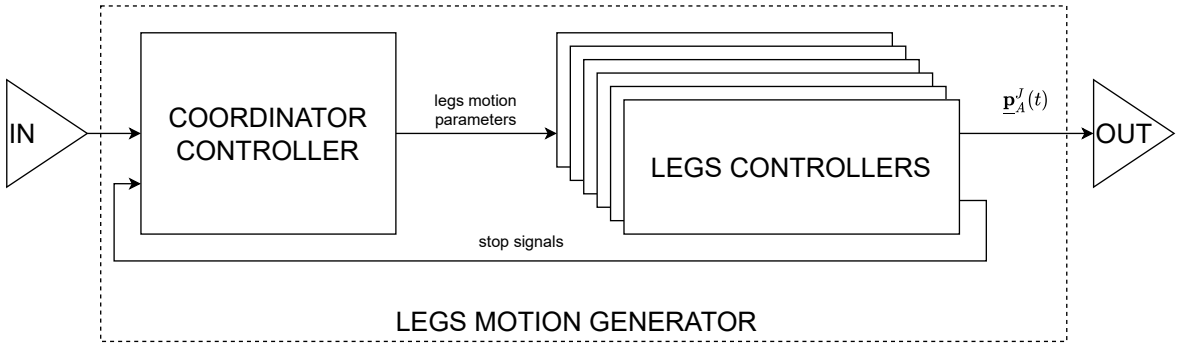


Figure 5.15: Legs motion generator.

5.5.4. Trajectory follower

As stated in chapter 4, the result of the path planning is a trajectory containing a list of $[x, y, \vartheta]$ positions and the corresponding list of parameters $[h_{stride}, h_E, L_{stride}, ID_{gait}, v_{forward}]$. The **trajectory follower**, determines which is the current position of the rover along the trajectory $\underline{\mathbf{t}} = [x, y, \vartheta]$, which point is aiming to reach $\underline{\mathbf{t}}_{next}$ (next point) and the parameters for this branch. (fig. 5.16)

5.5.5. Steering controller

Two of the parameters required by the leg motion generator, are the curvature radius R_C and the lateral velocity $v_{lateral}$, that are provided by the **steering controller**.

The curvature radius is generated to follow a prescribed trajectory in two possible ways:

1. If the current direction of the robot is pointing toward the trajectory ($\underline{\mathbf{t}}_{next}$), it generates a circle passing for the current position, tangent to it and tangent to the trajectory ($\underline{\mathbf{t}}_{next}$ direction).

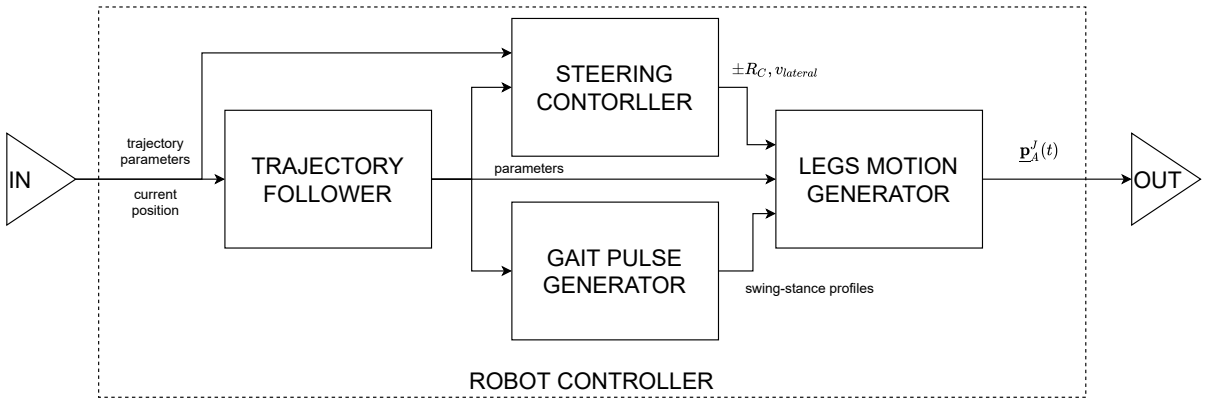


Figure 5.16: Robot controller.

2. If the current direction is not pointing towards the trajectory ($\underline{\mathbf{t}}_{next}$), then a proportional gain is used, with x_{error}^{RB} being the distance from the trajectory to the rover RB position:

$$R_C = \frac{1}{P_{steer} x_{error}^{RB}} \tag{5.46}$$

with $P_{steer} = 100\frac{1}{m^2}$

Using just this controller brings some limitations, in fact, while not pointing toward the trajectory, it will steer to realign to the trajectory using (2), when it will be parallel to it, the controller is switched to(1), but the tangent circle will have almost infinite R_C, therefore moving parallel to the trajectory and not closing any eventual gap. To compensate this, a lateral velocity $v_{lateral}$ is added, when the angle with trajectory is almost parallel ($|\text{angle}| < 5°$):

$$v_{lateral} = P_{v,lateral} x_{error}^{RB} \tag{5.47}$$

with $P_{v,lateral} = 5\frac{1}{s}$. (fig. 5.16)

5.5.6. Gait pulse generator

To generate the proper leg motion sequence, and so the proper gaits, a controller must generate a series of pulses defining swing phases and stance phases for each leg.
The **gait pulse generator** performs this by modulating a series of predetermined patterns (section 2.2.2), so to have the required t_{cycle}.
To grant the stability of the robot, each swing phase is started only when all the legs are in contact with the ground.(fig. 5.16)

5.5.7. Lean joint controller

The lean joint is directly controlled so to align the leg to the vertical, using a proportional controller with gain 1.

5.6. Terrain generation

The terrain is generated and used as an heightmap $Z_{terrain}(x, y)$, both for the simulation and the path planning, wide 5x5m and with resolution of 1cm.
The algorithm must be able to generate the terrain randomly, to repeatedly test the path-planning on different terrains.
To do so, the terrain is generated by adding randomized features, like hills and rocks.

Hills can be modeled as 2D axially symmetric Gaussian functions:

$$h(x, y) = Ae^{-\frac{(x - x_0)^2 + (y - y_0)^2}{2\sigma^2}} \tag{5.48}$$

Where (x_0, y_0) denotes the position of the maximum (or minimum) of the curve, A its peak height (or peak depression) and σ will relate to the slopes of the sides of the hill. Setting these parameters, is possible to generate a hill (or a pit), defining position, dimension and slope. The single hill is bounded to not have a maximum slope higher than a limiting value **max allowed slope** $\alpha_{max,hill}$ (20° in this work), to avoid the generation of totally unfeasible terrains. For this purpose, can be noted that cutting the 2D Gaussian with a plane perpendicular to the xy-plane and passing for (x_0, y_0) in any direction, will result into a 1D Gaussian with the same parameters. For example, imposing $y = y_0$, can be obtained

$$h(x, y_0) = Ae^{-\frac{(x - x_0)^2}{2\sigma^2}} \tag{5.49}$$

And by computing its second derivative and imposing it to 0, the point with maximum derivative can be found, resulting $x = \sigma$. Using the first derivative, the maximum derivative can be computed:

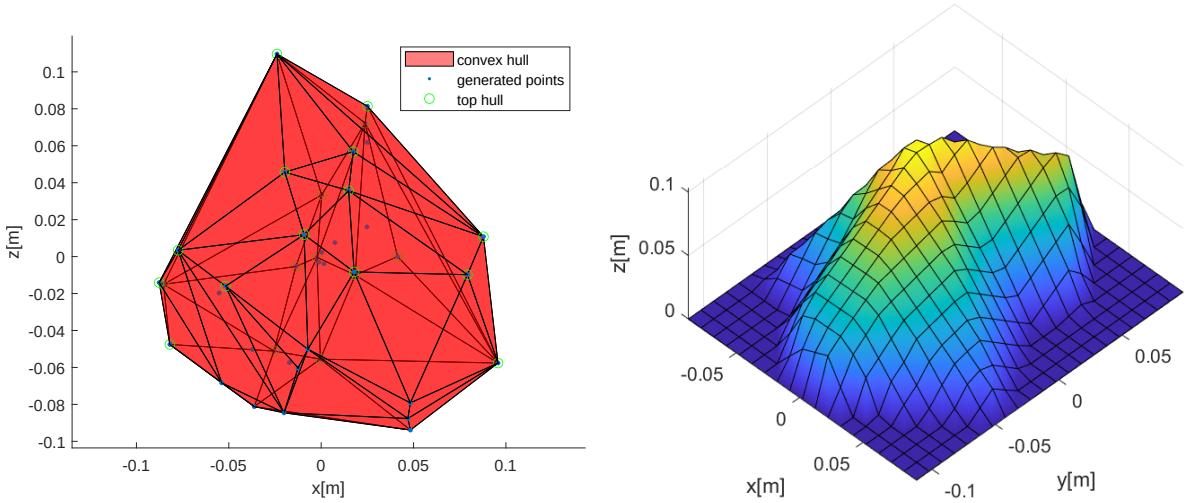
$$h'_{max}(x, y_0) = -A\frac{1}{\sigma\sqrt{e}} \tag{5.50}$$

And since maximum allowed slope must be respected, and the first derivative is the tangent of the slope, a limiting A can be computed:

$$A_{max} = \sqrt{e} \cdot \sigma \tan(\alpha_{max,hill}) \tag{5.51}$$

A random A of the hill can be generated between $+A_{max}$ and $-A_{max}$, while the center (x_0, y_0) is positioned randomly on the terrain, if the hill is small ($2\sigma <= L_{max,terrain}$); otherwise the center is positioned even outside the terrain, but the center of the terrain must be inside the maximum slope circumference of the hill, so to have a uniform slope effect. This means that the range to spawn the hill center must be 2σ.
The resulting heightmap of the hill $h(x, y)$ is then summed to the terrain heightmap, and the lowest point of the map is redefined as $z = 0$.
This is done for each hill that must be added, in this simulation have been used 12 hills with σ as 100, 10 and 10 random values between 1 and 3 (the random values change for each terrain generated).

After the terrain morphology has been generated, rocks must be added.
A rock can be generated as a convex hull of a cloud of random points. Defining the variables R_{mean}, n_{points} and E, a random point in spherical coordinates can be generated with a random φ in $[0, 2\pi]$ and ϑ in $[-\pi/2, \pi/2]$ (uniform distribution), while the radius have a probability distribution with as mean R_{mean} and $\sigma = R_{mean} \cdot E$. E represents the "edgyness" of the rock, smaller values means the rock will tend to be spherical, with bigger values will tend to be like a shard.
Generated the points, their convex hull is generated, the upper points of the hull are obtained(fig. 5.17a) and the rock is converted into an heightmap (fig. 5.17b).



(a) Rock generated through points.

(b) Heightmap of the rock.

Figure 5.17: Rock generation.

Finally the rock is positioned on a random point of the terrain, positioned with its zero at the height of the point on the terrain, and the union between terrain and rock maps is

performed (in each (x,y) is kept the highest z).
This operation is performed for several rocks and different groups of rocks.
Each rock R_{mean} is generated randomly, using a Gaussian distribution with mean $R_{up,mean}$ and σ_{up}, and is generated a number of rocks n_{rocks} so to approximately cover a certain percentage of terrain. Assuming that each rock will occupy a an area of about $\pi R^2_{up,mean}$ then

$$n_{rocks} = \rho_{coverage} \cdot \frac{A_{terrain}}{\pi} \mathrm{R}_{up,mean}{}^2 \tag{5.52}$$

The $\rho_{coverage}$ itself is randomized an and is the absolute value of a number generated with normal distribution with mean 0 and $\sigma = \sigma_\rho$.
This is performed for 3 groups of rocks: big rocks, rocks that cannot be traversed by the rover, working as obstacles; medium rocks, that can be hovered by robot E or T bands, but cannot be stepped on; and small rocks, like pebbles that can be stepped on, but creates problems for the stability of the rover.
The parameters used on each rock set can be seen in table 5.5.

	Small	**Medium**	**Big**
$\mathbf{R_{up,mean}}[m]$	0.02	0.1	0.2
$\sigma_{\mathbf{up}}[m]$	0.01	0.05	0.2
E[-]	0.1	0.2	0.2
$\mathbf{n_{points}}$[-]	40	40	40
σ_ρ[-]	0.05	0.05	0.01

Table 5.5: Rocks groups generation parameters.

Generated the terrain heightmap $Z_{terrain}(x, y)$, the local slopes along x and y are computed in each point of the map ($S_x(x, y)$,$S_y(x, y)$), is computed also a map of the total local slope, and is converted into a boolean map of the points, where the slope is higher than the one that grants foothold stability for the rover $\Gamma_{terrain}(x, y)$.

5.7. Power post processing

Since this work is about energy consumption, after a simulation has been run, the power consumption must be estimated.
Simulink and Simscape compute during the simulation the torque profile $\widetilde{T}_i(t)$ for each joint and their angular velocities $\widetilde{\omega}_i(t)$. Since the motors are connected to joints through

an harmonic gear [24, 19, 21] the values must be converted into motor values through

$$T_i(t) = \frac{\widetilde{T}_i(t)}{n_{gear}} \omega_i(t) = \widetilde{\omega}_i(t) n_{gear} \tag{5.53}$$

Since the motor itself is not simulated in Simulink, and the leg are controlled kinematically, in can generate any arbitrary force, but in reality motors does have limits. Therefore defined the max motor torque T_{max}, is computed if there are periods where is requested a force above the limits, and how much they last. If the peaks last less than 0.2 seconds, than the simulation is considered acceptable. This is done to add a bit of compliance, since, being controlled in positions, the joints will generates for some instants extremely high forces, like on impacts. In reality it would not be a problem, because those instantaneous movement requested for brief instants, are not present.
The $T_i(t)$ is capped with T_{max} and its derivative $\mathring{T}_i(t)$ is computed. The electric motor equations are

$$\begin{cases} L\mathring{i} + Ri + K\omega = V \\ J\mathring{\omega} + b\omega - Ki = -T_L \end{cases} \tag{5.54}$$

$J\mathring{\omega}$ can be comprised in T_L, since simulink already computes it, and adding $P = Vi$ the power can be computed as

$$\begin{cases} I = \dfrac{T_L + b\omega}{K} \\ P = \left(L\dfrac{\mathring{T}}{K} + Ri + K\omega \right) i \end{cases} \tag{5.55}$$

Using parameters in table 5.6 and T is the $T_i(t)$. The power is capped to 10kW just to avoid numerical errors to make explode the intergal. All the joint powers are summed and is added also the **no load power** P_{noload}, the power consumed by the robot without generating torques, obtaining the **total power** $P_{tot}(t)$. Then, the total power is integrated to obtain the total energy consumption.
Parameters have been computed from the motor datasheet[25] and the remaining ones (b), have been estimated to try to obtain the same power profile in [20]. The result are always similar in behaviour, but in absolute value they are accurate only at high torques and rotation velocities. This might happen because the data in [20] are referred to experiments, while the simulation uses only theoretical data. For the purpose of this work, just a behaviour simulating a motor would be enough, but must be highlighted one thing: in the working region of the rover, the joints power demand simulated is approximately half of the one measured in [20], but since must be added the P_{noload}, the

Parameter	Values
$\mathbf{K}[mN/A]$	58
$\mathbf{R}[m\Omega]$	540
$\mathbf{H}[\mu H]$	490
$\mathbf{b}[Nm/s]$	0.0017*
$\mathbf{T_{max}}[Nm]$	0.96
$\mathbf{n_{gear}}[-]$	100
$\mathbf{V_{motor}}[V]$	48
$\mathbf{P_{noload}}[W]$	66.5**

* Estimated attempting to obtain a profile like in [20].
** For the entire rover.

Table 5.6: Motor and gear parameters [19, 20, 25]

difference is much less significant (like 110W against 130W). But since the experiments are performed on Earth, while the simulation runs with Mars gravity , this difference is even higher. This can be a problem, because the power consumption is dominated by the contribution of the no load power, causing the algorithm to mainly optimize the path by increasing the speed and reducing the travel time. In fact the only parameters to add a significant variations to the consumption, are the slopes and the velocity, while the others despite behave as expected, their effect on the consumption is so marginal that are not considered for optimization. Probably this algorithm can work with high variation on joint consumption, but a most relevant benchmark must be created.

5.8. Stability post processing

To evaluate the success and the quality of a path simulation, an other analysis can be performed on the stability.
The stability of the rover can be measured with the Static Stability Margin S_m (see section 2.2.2). During the simulation Simulink measures the position for each foot, if they are in contact to the ground and computes the position of the varying center of mass (COM) of the rover, accounting also for legs positions. Furthermore can detect if a contact is stable or unstable (above the limit slope) and accounts the foot as not in contact. With this data the support polygon can be computed, and measuring the minimum distance of the rover COM projection from the support polygon edges, the stability margin can be computed for each time instant.

The stability margin profile can be converted into a mean stability margin of the path with a time-weighted mean.
Furthermore, since $S_m \leq 0$, are possible, especially due to unstable contact, and represent unstable configurations; they can be identified, and the fraction of time during the simulation when the rover is unstable, can be computed.

5.9. Y offset-slope-gait relation analysis

As said in section 5.5, the y_{offset} for the robot is evaluated by the controller on the run, based on the slope and the gait. The controller uses a lookup table to obtain the value. In this section will be discussed the way to obtain this table.
First is created an exploration grid, made of longitudinal slopes $\alpha_| = [-40, -35, -30, -25, -20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]$, stride lengths $L_{stride} = [0.1, 0.25]$ and the four gaits described in section 2.2.2: slow wave (1), fast wave(2), ripple(3) and tripod(6). For each combination between the three parameters is run a bisection algorithm. The algorithm run a simulation at each iteration, trying to find the y_{offset} that maximize the mean stability margin (see section 5.8). It works with a tolerance of 1cm and search in the range of [-3dm,+3dm]. The final result will be a matrix of optimal y_{offset}. The matrix is filtered of the spurious non monotonic results, since to in crease stability, with increasing slope the robot must increase y_{offset}. And the results will be grouped for each stride length-gait combination. Will be performed an interpolation along the slope to convert all slope slamples in m and q parameters of:

$$y_{offset} = m\alpha_| + q \tag{5.56}$$

So the final result are table 5.7

$\mathbf{L_{stride}}[mm]$:	**0.1**	**0.1**	**0.25**	**0.25**
	m[mm/°]	**q**[m]	**m**[mm/°]	**q**[m]
Slow wave (1)	0.5000	0.0160	0.5417	0.0187
Fast wave (2)	0.4688	0.0222	0.6458	0.0299
Ripple (3)	0.1250	0.0389	0.5417	0.0472
Tripod (6)	0.0625	0.0354	0.4375	0.0458

Table 5.7: y_{offset} lookup table.

5.10. Gait-roughness relation analysis

As seen in chapter 3, the gaits are selected using a missing contact coverage factor ρ_{MCC}. This result comes from a stability analysis section 5.8.
The idea is that when a foot steps on a rock, it creates an unfeasible foothold, so the leg is not considered for the support polygon generation. This can cause the stability margin to become negative, and so the rover to be unstable. If this happens to a foot during tripod gait, with just 3 footholds, it immediately becomes unstable. Instead, for a wave gait to become unstable, 3 feet must be unstable. Therefore, a rougher terrain requires a more stable gait, and this analysis wants to find a criteria to select the gait.
An exploration is performed this time for flat terrain with only small rocks with $\rho_{coverage} = [0, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 0.8, 1]$ (terrain generation parameter described in section 5.6), and the four gaits described in section 2.2.2: slow wave (1), fast wave(2), ripple(3) and tripod(6).
A simulation is performed for each combination and, performing the stability analysis (section 5.8), the percentage of time when the rover is in an unstable position ($S_m \leq 0$) can be obtained. Has been chosen for this value to stay under 5%, so to maintain stability, yet grant a bit of tolerance to the simulation.
Therefore, for each gait, the highest values of $\rho_{coverage}$ that can stay under 5% are set as limiting values.
Must be noted that the $\rho_{coverage}$ is the value used to generate the terrain, while ρ_{MCC} is the one used in the cost function. The first does account the entire rock as occupying terrain, while the second only the part with a steep slope, so, generally, the border. Therefore, the ρ_{MCC} for each test must be computed and on those, must be chosen the limit values. Values shown in table 5.8.

Gait	$\rho_{\mathbf{MCC,max}}[\%]$
Slow wave (1)	32.56
Fast wave (2)	21.34
Ripple (3)	15.55
Tripod (6)	0.52

Table 5.8: ρ_{MCC} boundaries table.

5.11. Cost Of Transport matrix evaluation

Similarly to section 5.9 and section 5.10, the analysis can be performed to obtain the $COT(\alpha_{|}, \alpha_{\_}, h_{stride}, h_E, L_{stride}, \text{gait}, v_{forward})$.

Several combinations are generated using parameters in table 5.9. The ones respecting the constraints in section 5.2.1, 5.2.2, 5.2.3 are maintained and are simulated. Each combination is simulated for a duration of 3 cycles, on smooth terrain, and the data of the last 2 cycles, are used for post processing (section 5.7, 5.8) obtaining a COT to put in the matrix. NaN if the simulation was not feasible.

Parameters	Values
$\alpha_{\|}$ [°]	-40 -30 -20 -10 0 10 20 30 40
$\alpha_{\_}$ [°]	0 10 20 30 40
$h_{stride}\,[cm]$	5 15 25 35 45
$h_E\,[cm]$	5 15 25 35 45
$L_{stride}\,[cm]$	5 10 15 20 25
ID_{gait}	6 3 2 1
$v_{forward}\left[\frac{cm}{s}\right]$	1 3 5 7 9

Table 5.9: Parameters to evaluate the COT matrix.

Furthermore, the post processing computes the final error in following the trajectory, and the presence of torques outside the limits. The few values with an error on the trajectory higher than 10 cm, have been discarded, while no element presented a torque outside the limits.

5.12. Algorithms parameters

In table 5.10 are reported the parameters, used to test the algorithms and obtain the result reported in chapter 6.

Parameters	RRT*	RRCT*	RRHT*
$L_{sampling}[dm]$	1	1	1
$d_{extend}[dm]$*	5	5	-3
$d_{connect}[dm]$*	5	5	4
$\sigma_{exp}[-]$	0.2	0.5	0.5
$\sigma_{opt}[-]$	0.8	0.5	0.5
$R_{C,min}[dm]$	1.5	1.5	1.5
$R_{C,max}[dm]$	inf	5	5

* Remember the distances have a different definition for the algorithms (chapter 4).

Table 5.10: Algorithms parameters.

6 | Results

Defined the simulation instruments in chapter 5, the validation of the path planning algorithm can be performed. For this scope, 304 terrains have been generated (section 5.6), the target direction has been defined as +y, and on each terrain the 3 algorithms in chapter 4 have been used to to obtain the best path.

The performance have been measured considering:

- The **total energy** needed to reach the target E_{tot}. This value is the "real" one, the one simulated as in section 5.7.
- The **absolute error** between the E_{tot} and the energy estimated by the path planner $E_{tot,est}$:

$$e_{abs,E_{tot}} = E_{tot,est} - E_{tot} \tag{6.1}$$

  so if the error is negative, means that the planner underestimated the energy required.

- The **relative error**:

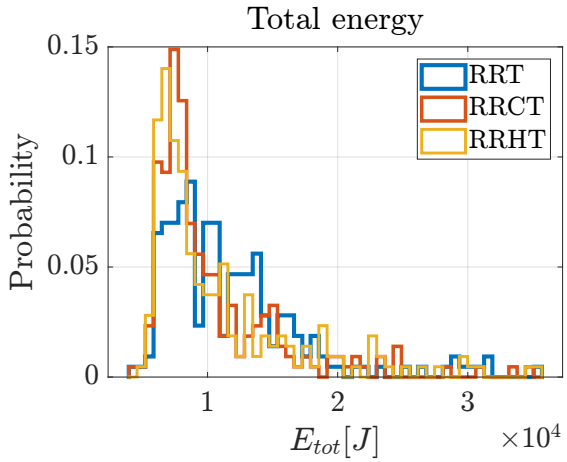
$$e_{rel,E_{tot}} = \frac{e_{abs,E_{tot}}}{E_{tot}} \tag{6.2}$$

- The number of calls to the cost function (single point calls) (chapter 3), to reach a certain stage.
- The execution time needed, to reach a certain stage.

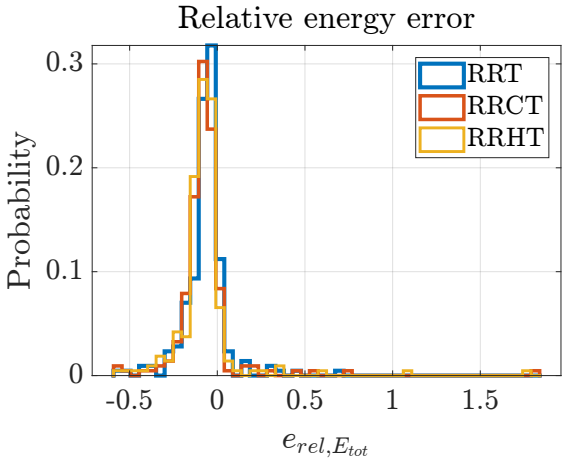
Their distribution will be shown in fig. 6.1, 6.2, 6.3, while their mean values over the samples will be shown in table 6.1,6.2, 6.3, 6.4.

A premise is that the mean values are computed over samples, feasible both for RRT* and RRCT* to be consistently comparable for each terrain, while the fail of RRHT* is not a criteria to discard a sample, because being it in a very early stage, has some problems and would reduce too much the number of samples. The resulting comparisons show, that RRCT* and RRHT* find generally cheaper paths (fig. 6.1a,table 6.1). Precisely, the difference of the mean E_{tot} between RRT* and RRCT* is of 1447 J, while between RRT*

and RRHT* of 909 J. While not being huge differences, that means that RRT* paths are 14% more expensive respect to RRCT* paths(in mean), and 8.3% more expensive than RRHT* paths. This is expected since (as said in section 4.2), RRT* is not actually able to evaluate curves, while RRCT* and RRHT* can, and so can better optimize the path. The estimation errors $e_{rel,E_{tot}}$ and $e_{abs,E_{tot}}$ are not significantly different (fig. 6.1b, fig. 6.1c, table 6.1), this is reasonable, since they all use the same cost function (chapter 3).
The difference, instead, is relevant for RRT* before the path post-processing, as can be seen in fig. 6.1d and table 6.1, where, as already said, the algorithm does not evaluate turns.



(a) Total energy consumed during the path simulated in Simscape.



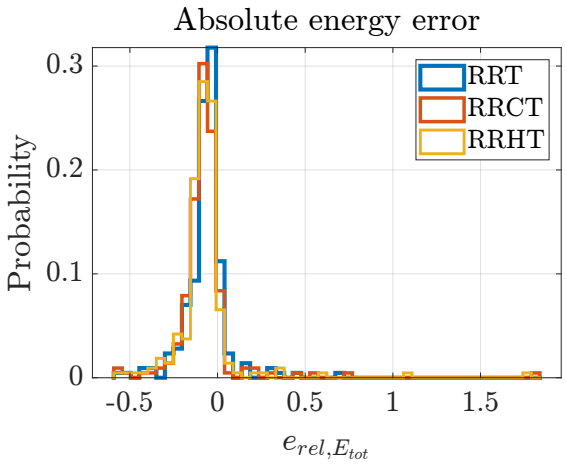
(b) Relative error between energy simulated ("real") and energy estimated by the path planning.

Figure 6.1: Energy consumption and estimation between the algorithms pt1.

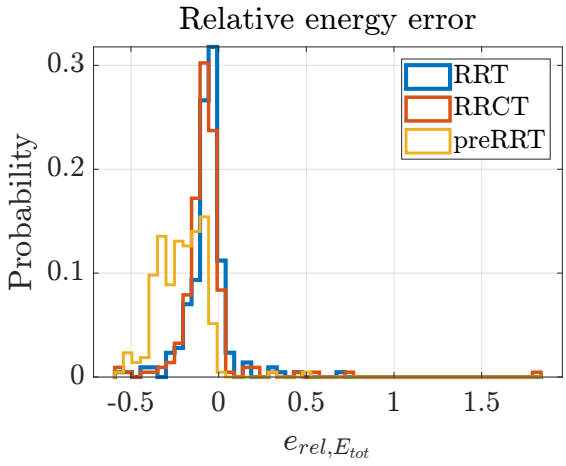
	preRRT*		RRT*		RRCT*		RRHT*	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
$\mathbf{E_{tot}}[J]$	"	"	11845	5606	10398	5161	10936	5952
$\mathbf{e_{rel,E_{tot}}}[-]$	-0.216	0.136	-0.067	0.125	-0.073	0.184	-0.071	0.190
$\mathbf{e_{abs,E_{tot}}}[\mathbf{J}]$	2727	2551	1256	2112	1328	2201	1449	2440

Table 6.1: Total energy consumed simulated data. Including the "real" energy consumed and the errors on the estimation. preRRT* refers to the RRT* result before the post processing of the path (see section 4.1.4).

Looking at the number of cost function calls to reach said results, we can measure them



(c) Absolute error between energy simulated ("real") and energy estimated by the path planning.



(d) Relative error between energy simulated ("real") and energy estimated by the path planning showing energy estimated by RRT* without the post processing (section 4.1.4).

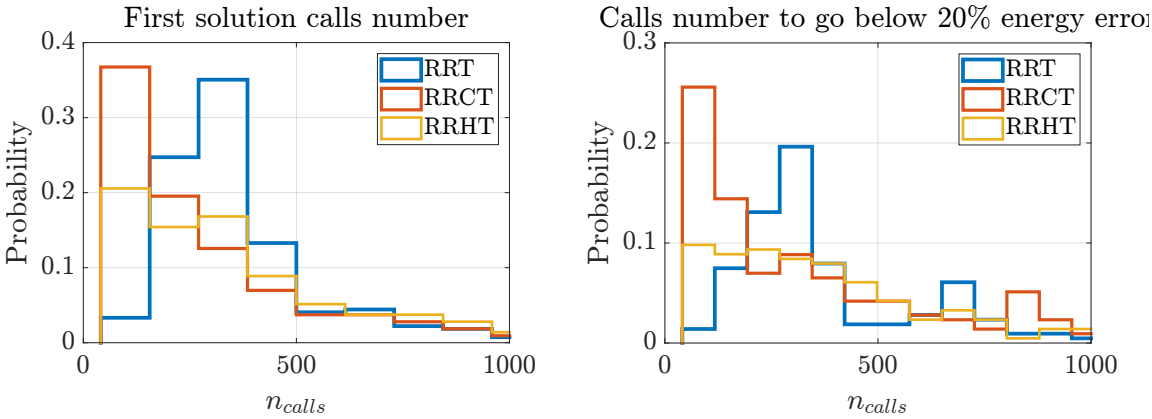
Figure 6.1: Energy consumption and estimation between the algorithms pt2.

at 4 milestones. Them are: the first path, when a path maximum 20% or 10% more expensive than the optimal path is found, and the end of the path planning.

At all milestones RRCT* has the lowest mean amount of calls (fig. 6.2, table 6.2), this is possibly due to the bigger d_{extend} and $d_{connect}$ parameters forced by the RRCT*, creating less dense trees. Furthermore, the path optimization algorithm for RRCT* generates a lot less circles respect to RRT* optimization points, therefore the lighter optimization terminates first the code. The last point is confirmed by the fact that RRT* has less calls than RRHT* in all milestones, except the last, since the optimization process is lighter and and terminates quickly.

Same discussion for the calls can be done for the time.
The analysis is the same as for the calls (fig. 6.3, table 6.3), but must be added that in this case the partial milestones of the RRT* do not account for the time of path post processing, causing the RRT* time to find first path to be lower than RRCT*, while having an higher number of iterations.

These data are useful in the case of path planning while moving. If the algorithm needs several seconds to compute an optimal path, instead of waiting for the algorithm to finish, wasting a lot of energy standing still, the rover could start moving before the final path is found, knowing that the path will not be optimal, but already reasonably cheap.



(a) Number of calls to find a first feasible path

(b) Number of callsto find a path with relative error on the energy lower than 20%.

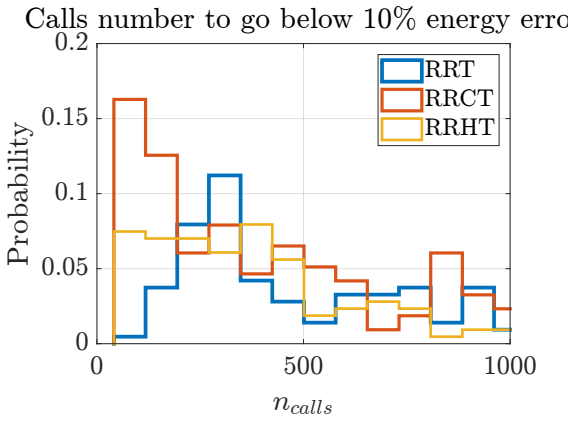
Figure 6.2: Simulations number of cost function calls distribution comparison between the algorithms pt1.

	RRT*	RRCT*	RRHT*
Calls to first path	711	564	841
Calls to 20% error	1078	648	1144
Calls to 10% error	1570	803	1432
Calls to end	2767	1251	2678

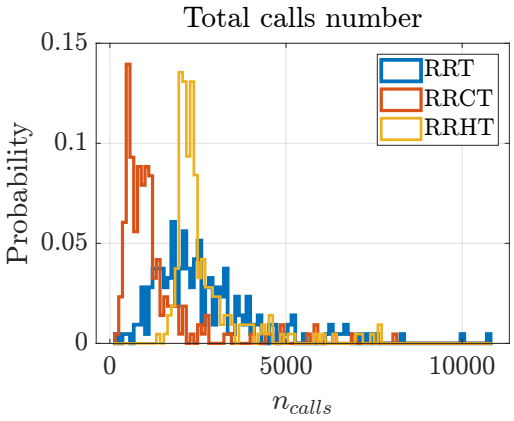
Table 6.2: Path planning cost function calls. Means of the number of calls to the cost function for the path planning algorithm. Are computed reported calls of when: the first path is found, the estimation of the energy has an error lower than 20% and 10% and when the algorithm terminates.

As last thing, is important to evaluate the ability of the path planner to find a feasible path.

As seen in table 6.4 the RRT* has the highest planing success rate of 89%. This is expected, as said in section 4.3, since, thanks to its capability to use small d_{extend}, and the use of thinner windows (because it does not account steering section 4.1), allows it to sneak the path in narrower passages. While increasing the ability to find a path, this reduces the reliability of the path generated. In fact, only 34% of the paths generated by the RRT*, are able to go through the post processing, without finding unfeasible positions. While the evaluation is overly conservative and the rover is still able to reach the destination, in some applications this unrealiablity is not acceptable.



(c) Number of calls to find a path with relative error on the energy lower than 10%.

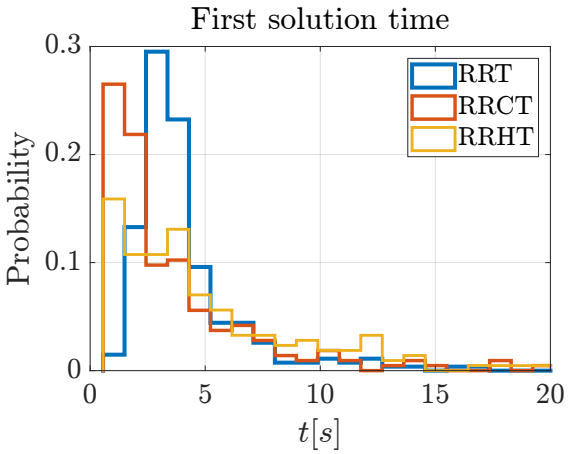


(d) Total number of calls to reach path planning terminal conditions.

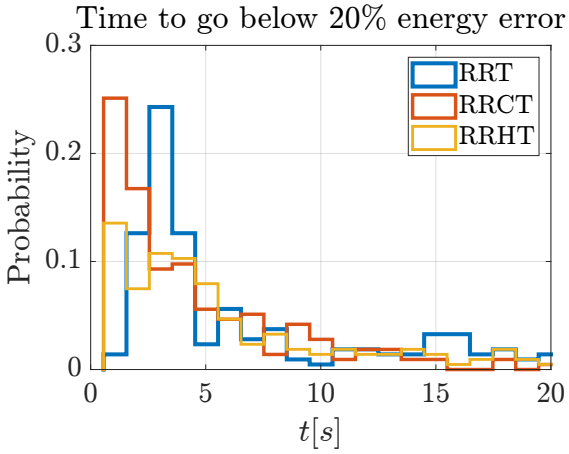
Figure 6.2: Simulations number of cost function calls distribution comparison between the algorithms pt2.

RRCT* and RRHT* have more difficulties in finding a path, the success rate is 71%. If is expected for the RRCT*, the RRHT* fails to reach its goal in improving this parameter. This might be due to the fact that RRT* does not evaluate curves at all, using slimmer windows, and so being unreachable by RRHT*; but is likely that since RRHT* has been drafted hastily, it is badly optimized and retains some errors.

Lastly, can be checked if the stability criteria defined in section 5.10, is respected.
In 10.7% of the terrains RRT* and RRHT*, while RRCT* in 7%, are not able to keep the unstable time below 5% (table 6.4). This is probably due to the fact that the possibility to step on a small rock is very statistical, and thus a criteria on it can reduce the probability, but not make it disappear.

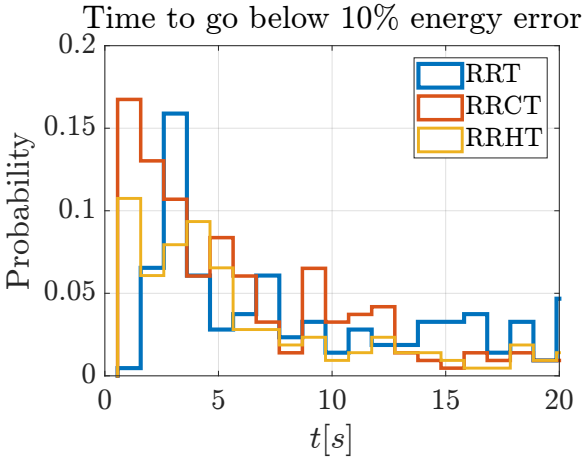


(a) Times to find a first feasible path

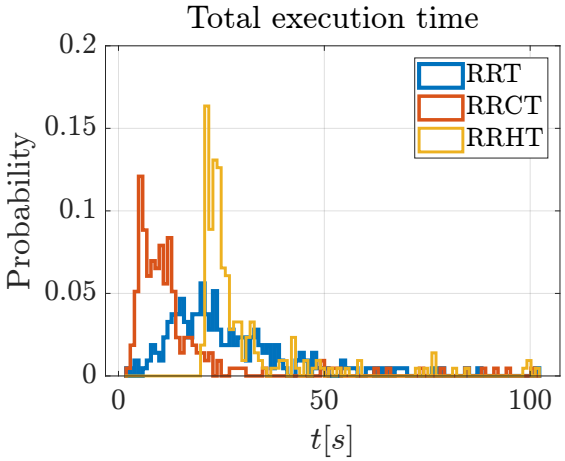


(b) Times to find a path with relative error on the energy lower than 20%.

Figure 6.3: Simulations times distribution comparison between the algorithms pt1.



(c) Times to find a path with relative error on the energy lower than 10%.



(d) Total time to reach path planning terminal conditions.

Figure 6.3: Simulations times distribution comparison between the algorithms pt2.

	RRT*		RRCT*		RRHT*	
	mean	**std**	**mean**	**std**	**mean**	**std**
Time to first path [s]	6.3	10.6	7.3	14.7	10.1	15.9
Time to 20% error [s]	10.7	-	8.16	-	13.5	-
Time to 10% error [s]	15.4	-	9.9	-	16.6	-
Time to end [s]	30.0	16.1	14.7	16.6	30.3	15.2

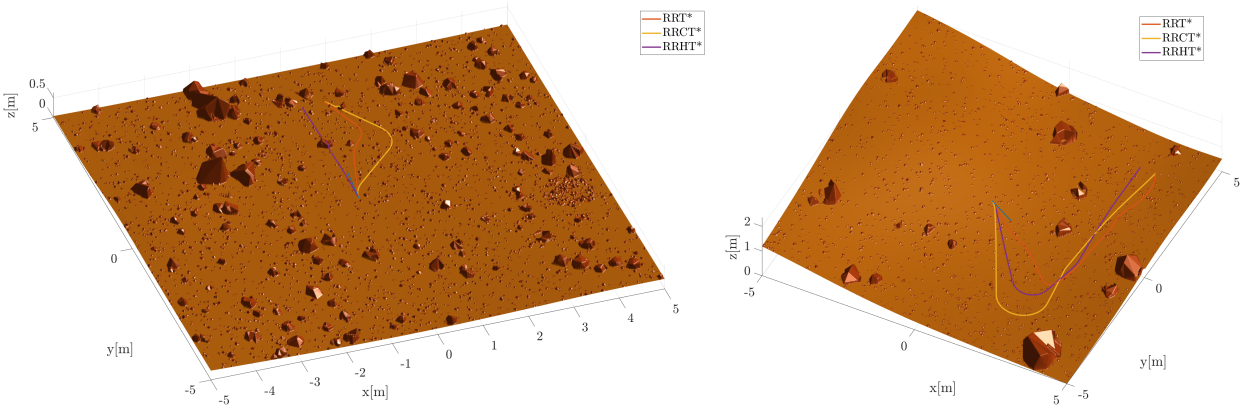
Table 6.3: Path planning time. Mean and standard deviation of the execution times for the path planning algorithm. Are computed reported times of when: the first path is found, the estimation of the energy has an error lower than 20% and 10% and when the algorithm terminates. Some data are not reported for low relevance.

	preRRT*	RRT*	RRCT*	RRHT*
Planning success [%]	89*	34*	71	71
Paths over 5% of instability [%]	"	10.7	7.0	10.7

The RRT is often able to find a solution, but during the path post processing some unfeasibilities are found, the path generally still works, but the safety estimation is not conservative (section 4.1.4).

Table 6.4: Other data. Including the percentage of terrain that the algorithm was able to plan and the percentage of simulation, where the criteria of the instablity during the par must be lower than 5% is not respected (section 5.10).

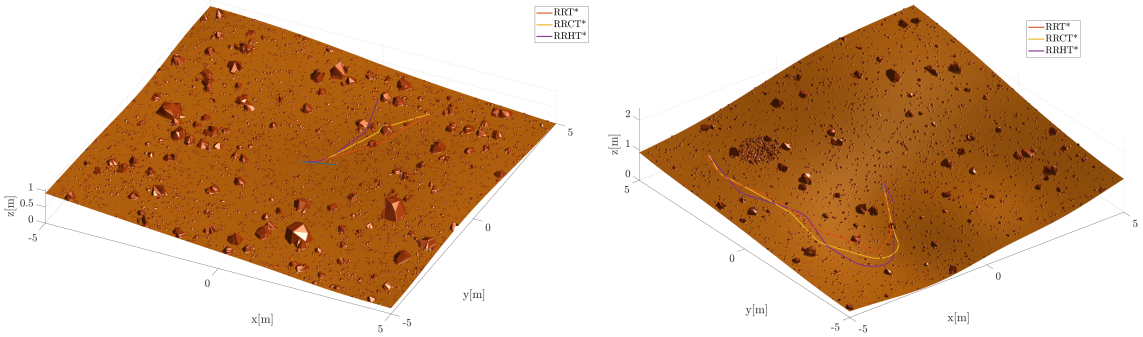
Performed an analysis on the statistic of all the simulations, another analysis can be performed on some of the terrain samples, to evaluate if some unexpected problems appear and check the behaviour of the planners fig. 6.4.



(a) Terrain with multiples medium rocks, the rover can pass over them.

(b) RRCT* uses bigger radii of curvature, respect to RRT*, thus reducing the consumption.

Figure 6.4: Simulation on terrains examples pt1.



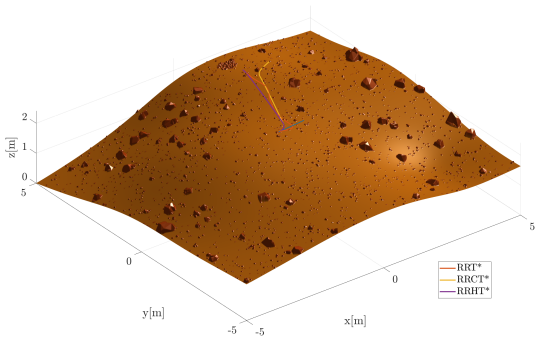
(c) Terrain with multiples medium rocks, the rover can pass over them.

(d) RRCT* and RRHT* have less and softer turns, reducing consumption.

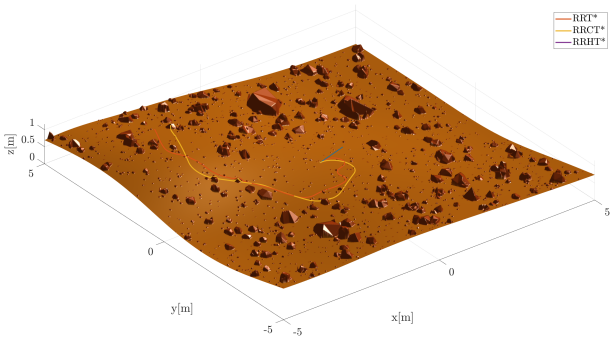
Figure 6.4: Simulation on terrains examples pt2.

What the terrain shows are just confirmations of what already said, just one think must be added.

Often the terminal part of the path does have a small sudden and pointless curve, can be clearly seen on flat terrain in fig. 6.4i and fig. 6.4m or with slopes in fig. 6.4e. This is due to the rough way selected to determine a target. In fact, the direction has been modeled ass a series of points, that can be possible targets. While this works nicely in the areas far from the goal line, it creates problems near the targets points, because the arrival

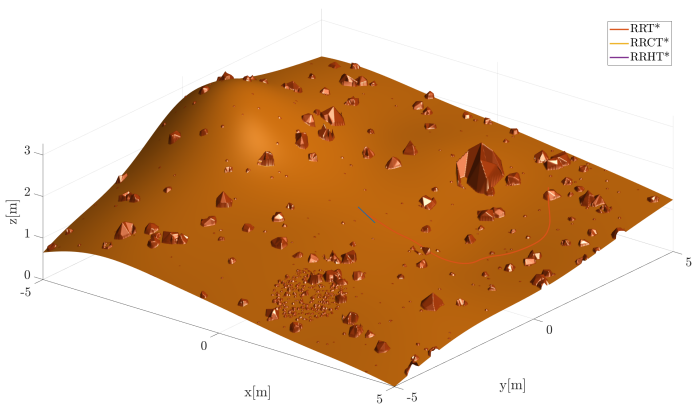


(e) Medium-high slope.

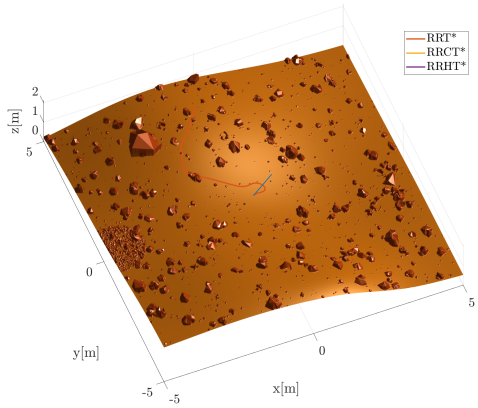


(f) Very harsh terrain due to medium rocks, RRHT* fails.

Figure 6.4: Simulation on terrains examples pt3.



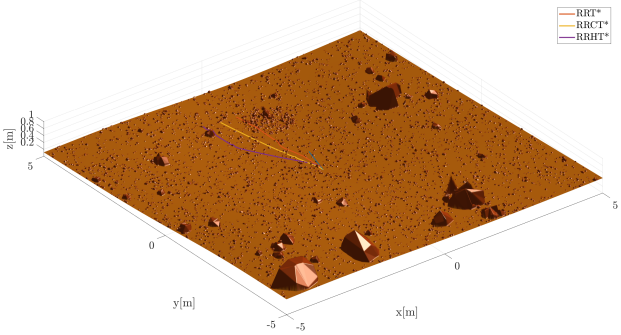
(g) Harsh terrain due to medium rocks and slopes, only RRT* successful.



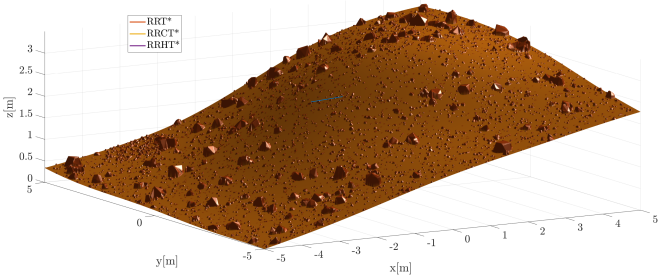
(h) Harsh terrain due to medium rocks and slopes, only RRT* successful.

Figure 6.4: Simulation on terrains examples pt4.

condition is to exactly pass for those point, therefore causing the path to turn. Putting an area as a target position, as in common RRTs, probably would solve the problem.

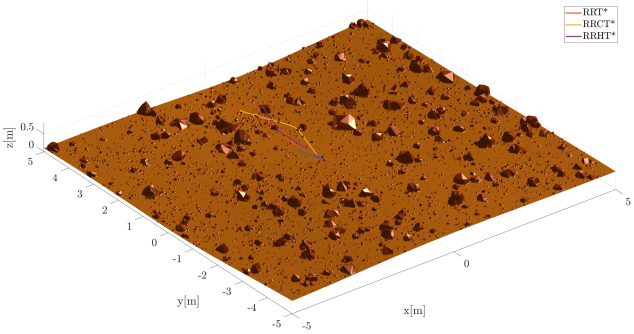


(i) Almost flat terrain, with several small rocks.

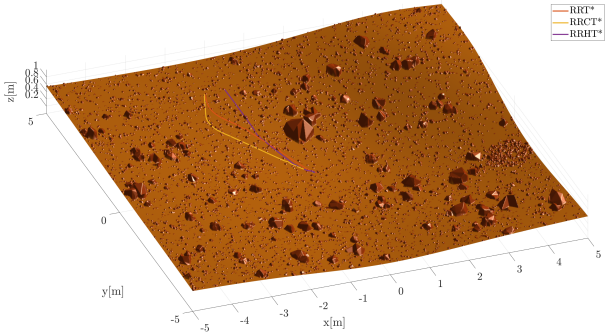


(j) Impossible terrain due to excessive slope, all fails.

Figure 6.4: Simulation on terrains examples pt5.

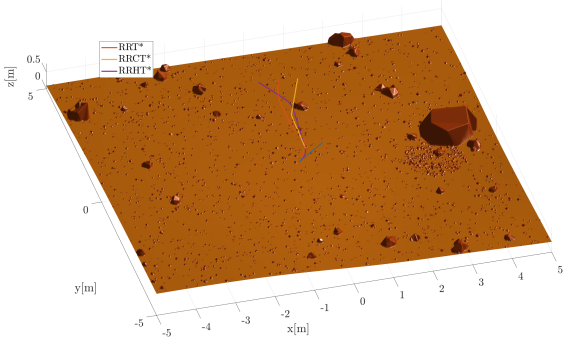


(k) Terrain with several medium rocks.

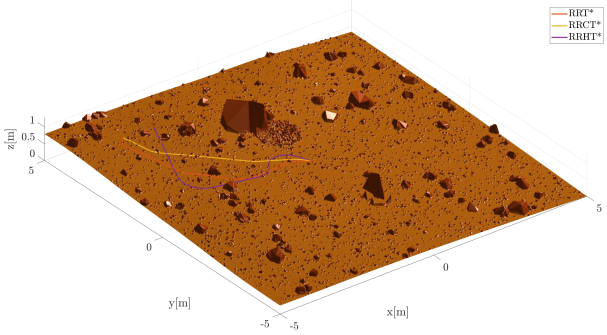


(l) Terrain with some medium rocks and a gap that can be exploited.

Figure 6.4: Simulation on terrains examples pt6.



(m) Almost flat terrain, yet the trejectories are not pointing to the goal with straigh lines.



(n) Harsh terrain.

Figure 6.4: Simulation on terrains examples pt7.

7 | Conclusions and future development

This thesis presents a path planning algorithm for legged robots, able to optimize paths to reduce energy consumption.
The result shows that the main goals are reached. The path planning is correctly performed, the energy consumption is estimated with reasonable error and the developed algorithms works better than others.

Must be remembered, that the cost function use a cost matrix that has been evaluated by simulations (section 5.11). The computation of a wider matrix, with more dense samples and some statistic evaluation, could allow higher prediction accuracy and accessibility to much more terrains.

While being promising for multiple applications, the validity of this work is still limited, since has been tested and validated just in certain conditions.
As said in section 3.7, the rover must be able to determine its position and to follow a trajectory relatively tightly. Furthermore, the validation has been performed on rigid and monolithic terrain, measurement errors of the terrain and state are not considered, and line of sight obstruction has been neglected.
Nevertheless, as explained in section 3.7, this limitations can possibly be lifted by further validations and modules additions.

The first effort in validating different benchmark robots or models, should be directed to have a robot with more variable power consumption. The possibility of this work to be flawed by low energy consumption in the actuators, is the first to be addressed.

The algorithms studied, retains some limitations, therefore must be improved. In particular, the RRHT* needs a deeper study and optimization, and could be developed in the direction of becoming a Dubin curve-based RRT* algorithm, adding procedures developed in this thesis, to the Dubin curve-based RRT, making it the most interesting alternative at the moment.

This work is just a preliminary exploration of the topic, it gave some answers on the energy-based path planning, but they are not resolutive and all aspects must be expanded.

Bibliography

[1] G. Genta, *INTRODUCTION TO THE MECHANICS OF SPACE ROBOTS*. Springer, 2012.

[2] M. Luneckas, T. Luneckas, D. Udris, D. Plonis, R. Maskeliunas, and R. Damasevicius, "Energy-efficient walking over irregular terrain: a case of hexapod robot," *Metrology and Measurement Systems*, vol. vol. 26, no. No 4, pp. 645–660, 2019.

[3] B. Jin, C. Chen, and W. Li, "Power consumption optimization for a hexapod walking robot," *Journal of Intelligent Robotic Systems*, vol. 71, pp. 195–209, 2013.

[4] T. Homberger, M. Bjelonic, N. Kottege, and P. Borges, "Terrain-dependant control of hexapod robots using vision," 10 2016.

[5] M. Prágr, P. Čížek, and J. Faigl, "Cost of transport estimation for legged robot based on terrain features inference from aerial scan," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1745–1750, 2018.

[6] N. Kottege, C. Parkinson, P. Moghadam, A. Elfes, and S. P. N. Singh, "Energetics-informed hexapod gait transitions across terrains," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5140–5147, 2015.

[7] D. Belter, P. Łabęcki, and P. Skrzypczyński, "Adaptive motion planning for autonomous rough terrain traversal with a walking robot," *Journal of Field Robotics*, vol. 33, no. 3, pp. 337–370, 2016.

[8] A. Stelzer, H. Hirschmüller, and M. Görner, "Stereo-vision-based navigation of a six-legged walking robot in unknown rough terrain," *The International Journal of Robotics Research*, vol. 31, no. 4, pp. 381–402, 2012.

[9] D. Wooden, M. Malchano, K. Blankespoor, A. Howardy, A. A. Rizzi, and M. Raibert, "Autonomous navigation for bigdog," in *2010 IEEE International Conference on Robotics and Automation*, pp. 4736–4741, 2010.

[10] K. Otsu, G. Matheron, S. Ghosh, O. Toupet, and M. Ono, "Fast approximate clear-

ance evaluation for rovers with articulated suspension systems," *Journal of Field Robotics*, vol. 37, 07 2019.

[11] O. Toupet, T. Sesto, M. Ono, S. Myint, J. Vander Hook, and M. McHenry, "A ros-based simulator for testing the enhanced autonomous navigation of the mars 2020 rover," pp. 1–11, 03 2020.

[12] N. Abcouwer, S. Daftry, S. Venkatraman, T. Sesto, O. Toupet, R. Lanka, J. Song, Y. Yue, and M. Ono, "Machine learning based path planning for improved rover navigation (pre-print version)," 11 2020.

[13] D. Helmick, A. Angelova, and L. Matthies, "Terrain adaptive navigation for planetary rovers," *Journal of Field Robotics*, vol. 26, no. 4, pp. 391–410, 2009.

[14] MATLAB, "Path planning with a* and rrt | autonomous navigation, part 4," 2020.

[15] S. M. LaValle and J. James J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[16] A. Becker, "Rrt, rrt* random trees," 2018.

[17] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1105.1186, 2011.

[18] D. Živojević and J. Velagić, "Path planning for mobile robot using dubins-curve based rrt algorithm with differential constraints," in *2019 International Symposium ELMAR*, pp. 139–142, 2019.

[19] S. Bartsch, T. Birnschein, M. Langosz, J. Hilljegerdes, D. Kuehn, and F. Kirchner, "Development of the six-legged walking and climbing robot spaceclimber," *Journal of Field Robotics*, 05 2012.

[20] S. Bartsch, "Development, control, and empirical evaluation of the six-legged robot spaceclimber designed for extraterrestrial crater exploration," *KI - $Ku_nstlicheIntelligenz, vol.$ 28, pp.* $127 - -131, 062014$.

[21] J. Hilljegerdes, P. Kampmann, S. Bosse, and F. Kirchner, "Development of an intelligent joint actuator prototype for climbing and walking robots," 09 2009.

[22] S. Miller, "Simscape multibody contact forces library," 2021.

[23] S. McCloskey, "Development of legged, wheeled, and hybrid rover mobility models to facilitate planetary surface exploration mission analysis," 2007.

[24] T. Tuttle, "Understanding and modeling the behavior of a harmonic drive gear transmission," 08 2005.

[25] TQ-SYSTEMS GMBH, *Motor Parameters ILM50x08*, 2021. Rev. 200.

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
d_{extend}	extension distance	m
$d_{connect}$	connection distance	m
$\underline{\mathbf{q}}$	state	-
t_{swing}	swing time	s
t_{stance}	stance time	s
t_{cycle}	cycle time	s
β	duty factor	-
L_{stride}	stride length	m
h_{stride}	stride height	m
σ_{stride}	stride slant	rad
S_m	static stability margin	m
Δz_{MJ}	height difference between middle and other legs G joints	m
x_{offset}	x offset	m
y_{offset}	y offset	m
z_{offset}	z offset	m
h_E	E height	m
RB	body frame	-
RH	local horizon frame	-
R_C	trajectory curvature radius	m
$v_{forward}$	forward velocity	m/s
$\alpha_{\mid}$	longitudinal slope	rad
$\alpha_{\_}$	transversal slope	rad
$h_{obstacle,max}$	max obstacle height	m
ρ_{MCC}	missing contact coverage factor	-
COT	cost of transport	J/m

$Z(x, y)$	heightmap	m
$S_x(x, y)$	slope along x map	rad
$S_y(x, y)$	slope along y map	rad
$H(x, y)$	relative heightmap	m
$\Gamma(x, y)$	missing contact matrix	-
$L_{sampling}$	trajectory sampling length	m
d_{start}	starting distance	m
σ_{opt}	optimization bias	-
σ_{exp}	exploration bias	-
e	cost from parent	J
\$	cost from start	J
ϑ_c^a	point absolute angle	rad
ϑ_c^o	point oriented angle	rad
ρ_S	stator welding distance	m
ρ_R	rotor welding distance	m
x_E	E-band boundaries	m
$x_{A,straight}$	A-band boundaries on straight line	m
$x_{A,steering}$	E-band boundaries on curve	m
$\delta_x, \delta_y, \delta_z$	margin along x,y,z direction	m
p_{foot}	penetration depth	m
$\mathring{p}_{foot}$	penetration velocity	m/s
v_{slip}	slip velocity	m/s
k	stiffness coefficient	N/m
b	damping coefficient	Ns/m
μ_s	static friction coefficient	-
μ_k	dynamic friction coefficient	-
v_{trans}	transition velocity	m/s
g	gravitational acceleration	m/s2
$\underline{\mathbf{p}}_A^{RH}$	position of A in RH	m
$\underline{\mathbf{p}}_A^J$	position of A in joint angles space	rad
$\underline{\mathbf{v}}_{stance}$	stance velocity	m/s
ID_{gait}	gait index	-
P_{xx}	proportional gain of xx	-

E	edginess	-
$\rho_{coverage}$	rock coverage factor	
$\tilde{T}_i$	joint torque of joint i	Nm
$\tilde{\omega}_i$	joint rotational velocity of joint i	rad/s
T_i	motor torque of joint i	Nm
ω_i	motor rotational velocity of joint i	rad/s
L	motor equivalent inductance	H
R	motor equivalent resistance	ω
K	motor constant	N/A
b	damping rotational coefficient	Nm/s
T_L	load torque	
T_{max}	maximum motor torque	Nm
n_{gear}	gear ratio	-
V	voltage	V
i	current	A
P_{noload}	no load power	
E_{tot}	total energy consumption	J
$E_{tot,est}$	estimated total energy consumption	J
$e_{abs,E_{tot}}$	absolute error on energy estimation	J
$e_{rel,E_{tot}}$	relative error on energy estimation	-

List of Abbreviations

Abbreviation	Description
ACE	Approximate Clearance Evaluation
ENav	Enhanced AutoNav
COT	Cost Of Transport
MCC	Missing Contact Coverage
RRT	Rapidly-exploring Random Trees
RRCT	Rapidly-exploring Random Christmas Trees
RRHT	Rapidly-exploring Random Holly Trees

Acknowledgements

Thanks to my parents for the continued support...
...since the very beginning.