Hierarchical representation of the landmarks detected in a visual scene during the navigation of a vehicle via Landmark-Tree map coding

Niama Elkhbir[1] under the supervision of Nicolas Cuperlier[2]

***Abstract*— In this paper, we will present two methods for navigating mobile robots, both based on biological findings that can replace SLAM methods. We will present the advantages of each method, and implement the algorithm replacing an already existing location system for a vehicle, which will then allow us to evaluate the performance of this new model.**

I. INTRODUCTION

Mobile robotics has been approached by researchers for several decades, starting with tests in indoor environments. Nowadays, outdoor navigation solutions for mobile robotics such as autonomous vehicles are of increasing interest to researchers because they can provide great solutions to industry; they can help reduce the number of crashes on our roads then increase safety, they can also reduce pollution and emissions by up to 90%.

In mobile robotics, the localization is usually solved by some approaches, named SLAM (Simultaneous Localization And Mapping) based on incrementally building a map of the environment in which the robot can be localized. They perform well in indoor application. However, they have limitations (such as scalability, evolving in different kinds of environment and at very different speeds..) that prevent them to achieve the performances needed for autonomous driving.

To remedy these problems with SLAM algorithms, several alternative methods exist. This article focuses on two of them called attention mechanism and Landmark-Tree Map. They will be presented in the following paragraph.

[1]Niama Elkhbir, Master 2 SIC student, CY Cergy Paris Université, 95000 Cergy Cedex, France. niama.elkhbir@ensea.fr

[2] Nicolas Cuperlier, Neurocybernetic team, ETIS laboratory, UMR 8051, 95000 Cergy Cedex, France. nicolas.cuperlier@cyu.fr

Our goal will be to modify and evaluate the existing visual localization system for a vehicle by replacing the landmarks coding done in [1] by a tree called Landmark-Tree Map. This will allow us to address the problem of the high resource cost of metric maps that do not scale very well. It will also make us able to easily free memory in case of a continuously growing map while still preserving the dominant information.

II. NAVIGATION METHODS

Here, we present two methods of navigation that are of interest to our work.

A. Attention mechanism [1]

This method handles the problem by taking inspiration from biological models (lab mice, ants...). As can be seen in figure 1, this solution is based on two types of data : visual data (images provided by a camera for example) and directional data. Directional data are simply used to decide whether an object is seen under a certain azimuth. To do this, a new network is built having as input data the coordinates on the image of the object in question and the orientation data of the robot. The output layer of this network is then a set of neurons coding for the belonging of the wanted azimuth to a sector of the unit circle, the sectors being taken in general equidistantly distributed, of size 2pi/N with N the number of output neurons.

For visual data, an image is first processed by a gradient filter and then convoluted with Gaussian difference windows. From the result of this filtering are then extracted the local maximums by a local competition mechanism. The rectangular neighborhood of a maximum is what we call landmark. landmarks can be conceived as the areas that draw our attention most to a given image.

The list of landmarks extracted from an image contains the information indicating the coordinates on the image of each landmark and the nature of this landmark. The first type of information is sent to the neural network which allows us to find the azimuth under which each landmark is observed.

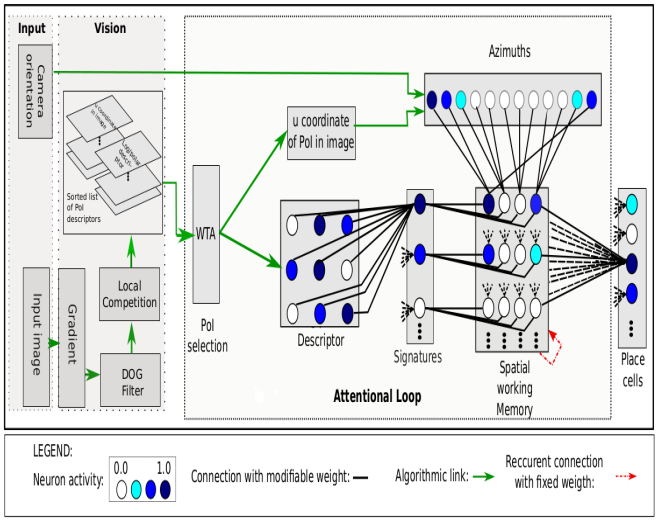


Figure 1 : From [1], Architecture of an algorithm using the attention mechanism. First, a vision chain successively filters the input image and ends with a local competition mechanism leading to a sorted list of Points of Interest (PoI), each characterizing a visual landmark. Second, each of the PoI belonging to a same image is sequentially selected, via an attentional loop, by a mechanism of winner takes all (WTA) according to its saliency level. The visual signature (what information) of each PoI is then computed from a log-polar descriptor, as well as the azimuth (where information) under which it is seen (computed from the camera orientation and the position of the PoI in the image along the u axis). Next, the product of each signature and azimuth couple is added to a Spatial Working Memory (SWM). The resulting pattern of activity of the SWM represents the current vehicle location that can thus be learned by a neural layer coding for place cells. A binary signal (l(t)) synchronizes the learning of the signatures, SWM and PC layers (not shown on the figure but see the text for a description).

The nature of the landmark (a flower, a cat, a house...) is extracted by applying the transformation log-polar to the landmark. This transformation has the particularity of being invariant by rotation, i.e. its value ignores the azimuth under which the object is seen (a flower should be perceived as a flower no matter what angle from which it is seen). For a given image, the set of values of the landmarks contained in this image obtained after transformation is called the descriptor of this image, it's the input layer of a neural network that decides whether an object belongs to the image. The set of natures of objects contained in an image is encoded by the output layer of this network, and that's called an image signature.

In order to know if we are in a place already visited or not, we should combine the signature and the azimuths under which each landmark is seen in order to decide if we have already visited a place where the same type of object is seen from the same angle. If such a place has already been visited, a neuron coding for this place must excite itself by exceeding a certain threshold. This type of neurons is called place cells, they are found in the hippo-campus of all mammals, that's why this algorithm is said bio inspired.

If the place is not already visited, the attention mechanism aims at looping this mechanism in order to synchronize the recruitment of neurons at the signature level (new type of objects perceived) and at the level of the place cell layer (new place visited). We can therefore see on one hand that learning is done in a single step, and on the other hand that we must imperatively use a synchronisation signal that allows to switch between learning phase and latency phase. It is important to note that the neurons coding for the places are stacked in the memory : the memory layer has a stack structure.

The attention mechanism allows a mobile robot to learn how to navigate without being trained beforehand (one-shot learning). This property makes this method more advantageous compared to the usual deep learning methods that require several teaching periods. However, sometimes this model can be less performant than deep network solutions but it's still more rapidly functional.

B. Landmark-Tree Map [2]

Landmark-Tree Map algorithm allows a robot to navigate and explore using a tree labeled with a set of landmarks. The construction of this tree is done by the algorithm whose pseudo-code is represented in figure 2.

The core idea is to put the detected landmarks in a tree structure (the Landmark-Tree Map), sorting

them from global to local information. Considering, that the angles of far distant landmarks remain static as the robot advances, we can use this information for our purpose.

The Landmark-Tree Map structure allows an optimization of both space and calculation time (the time taken to calculate the return path of a robot for example).

Algorithm 1: Append Viewframe To Tree

```
input        : a viewframe V, a Landmark-Tree T
output       : a Landmark-Tree T'
precondition : V is not empty
postcondition: |T| ≤ |T'|
if T.root not set then
    T.root = new Node(∅)
Node_current = T.root
while V has elements do
    {SameElements} = shared elements in
        Node_current.latestChild and V
    {RemainingElementsNode} =
        Node_current.latestChild \ {SameElements}
    {RemainingElementsViewframe} = V \ {SameElements}
    if {SameElements} is empty then
        Node_current.children ← new Node(V)
        V = ∅
    else
        if Node_current.latestChild ⊂ V then
            if Node_current.latestChild has children then
                Node_current = Node_current.latestChild
                V = {RemainingElementsViewframe}
            else
                if {RemainingElementsViewframe} ⊈ ∅ then
                    Node_current.latestChild.children ← new
                        Node(∅)
                    Node_current.latestChild.children ← new
                        Node ({RemainingElementsViewframe})
                V = ∅
        else
            TemporaryNode = new
                Node({RemainingElementsNode})
            move children from Node_current.latestChild to
                TemporaryNode
            Node_current.latestChild.children ← TemporaryNode
            Node_current.latestChild.children ← new
                Node({RemainingElementsViewframe})
            Node_current.latestChild = {SameElements}
            V = ∅
return T'
```

Figure 2 : Pseudo-code for the construction of the landmarks tree, precisely, for adding a view frame to the tree, from [2].

The characteristics of the Landmark-Tree Map are as follows :

- A visited place (a perceived image) corresponds to a leaf of the tree. The set of landmarks perceived in this place corresponds to the set of landmarks encountered while walking along the path connecting the root of the tree to this leaf.
- Leaves of the sub-tree whose root is a landmark located at a node inside the tree correspond to the set of places all containing the landmarks encountered while walking along the path connecting the root of the tree at the node in question. Since the sub-tree whose root is the root of the tree is the tree itself, the landmark that corresponds to the root of the tree must belong to all the places visited, and this is not general, such a place does not generally exist. To remedy this, the root of a Landmark-Tree Map is noted : None.
- When the robot moves, objects far away don't disappear from its view (the horizon for example). Such objects thus remain in the image and the two visited places must have in common all the landmarks located very far away in the first place. So the more we go down in the landmarks tree, the more the landmarks encountered tend to disappear more quickly during the navigation of the robot.
- When a new landmark is perceived during the navigation of the robot, the path on the landmarks tree bifurcates to the right : the navigation of the robot is from the leftmost to the rightmost leaf.
- When adopting a landmarks tree memory structure, some memory can be freed -while avoiding to generate all ambiguity between places- by removing the landmarks that tend to disappear the fastest, and thus those that are located the lowest in the tree.
- The landmarks tree map structure allows the optimization of the memory space avoiding the duplication of landmarks present in several places.

III. Methodology

The main idea of our work is to replace the stack structure used in the neuronal layers in figure 1 by a tree structure coding for landmarks, in

order to take advantage of the strengths of both algorithms. The recruitment of new neurons will then be done by the algorithm of figure 2 during the synchronized learning phases.

It was planned to work on the Promethe Neural Simulator that belongs to the UCP laboratory. A network of neurons coding for the attention mechanism had already been completed by the neurocybernetic team. I've also already coded the Landmark-Tree Map function on Promethe. For reasons of fluidity and because of some problems found on the structure of the code and boxes of Promethe related to my Landmark-Tree Map function, we decided, with my supervisor Mr. Cuperlier, to code the Landmark-Tree Map function in Python as well as the steps that precede Landmark-Tree Map, namely image processing and extraction of their descriptors. All the details of the work initially done under Promethe will be found in the appendices, from appendix 1.1 to appendix 1.6 and appendix 2. The details of the work done under Python will be presented in the following sections.

A. Main types of objects

1) Landmark: A landmark is defined by a pair of angle and descriptor. Except that the neural network gives activities representing the probability that the image is rotated by a certain angle, this angle is represented by a list of size 360 which gives for each element (degree of freedom) its own activity. The landmark is then defined by a list of angles and descriptors. The cosine or sine of the angles of a landmark is calculated as the average of the cosines or sines of all the angles weighted by their activities. To compare two landmarks, we use the function __eq__ of the class Landmark. This function is based on a boolean function c(Li,Lj) that reveals, whether a landmark Li corresponds to landmark Lj [3]. It can be computed by :

$$\mathrm{c}(L_i, L_j) = \begin{cases} 1 & \text{if } \| \mathrm{d}(\boldsymbol{d}_i,\ \boldsymbol{d}_j) \| < \zeta_{\boldsymbol{d}} \quad \bigwedge \quad \| \arccos\left(\boldsymbol{l}_i^T \boldsymbol{l}_j\right) \| < \zeta_{\boldsymbol{\phi}} \\ 0 & \text{else} \end{cases}$$

with d(.) the distance of Hamming compared to an applied threshold. The equality condition has been slightly modified, keeping the first condition relating to the Hamming distance between descriptors of the two landmarks, however, the condition relating to the arc cos of the scalar product between angle vectors of the two landmarks has been replaced by a condition relating to the scalar product between angle vectors of the two landmarks, because in practice it happens that this scalar product goes out of the definition domain of the arc cos. The new condition is justified by the fact that if the landmarks were perfectly equal their angle vectors would be perfectly equal, and therefore the scalar product between the two angle vectors would be equal to one. Thus, the original condition relating to the angle vectors can be replaced by an increase in the absolute deviation between the scalar product of the two angle vectors and one.

See appendix 3.

2) Viewframe: A view frame is defined as a list of landmarks. It has an add method that allows to add a landmark to it, and a t method that allows to compute the vector of directions considering the x and y coordinates as the coordinates on the average over the present landmarks.

See appendix 4.

3) Image: We deal with large databases, hence the need to detail the abstraction of each image and its processing.

Each image is instantiated by the path to its database folder. An image can be abstracted in its landmarks, i.e. in its view frame. The key points of an image carry several information about the location of the landmark : the angle and the answer. The answer for example represents the importance rate of the landmark in the image. As for the descriptor, it carries information on the nature of the landmark. An image has the method getKeyPointsAndDescriptors that returns two lists of the same size encoding the key point and descriptor pairs of each landmark using BRISK. An image also has the getViewframe method that allows to create a view frame according to the following method : from each key point and descriptor pair, we build a landmark that we then add to the view frame of the image and finally we return the view frame of the image. A first test of the use of BRISK is done on the image in figure 3.1.

See appendix 5.

Figure 3.1 : Test image.

The key points are visualized as small circles on the images using the save_keypoints method. An integer parameter M controls the number of landmarks. The more M is large the fewer landmarks we get. The result of the test with M=10 is shown in figure 3.2 and with M=200 in figure 3.3 .

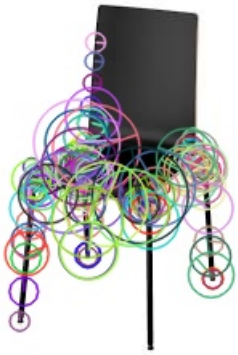


Figure 3.1 : Detected key points with M=10.

Figure 3.1 : Detected key points with M=200.

4) Ltreemap: A Landmark-Tree Map map is defined recursively. Its root is either a landmark or none, its children are a list of ltreemaps. An empty ltreemap is defined as a root none and its children as an empty list. A ltreemap has the add method that allows to add a new view frame to it. It also has an internal attribute called nbViewframes allowing an acceleration of the calculations. When adding a new view frame, nbViewframes is imperatively inserted of 1. Recursivity means that when adding a view frame to the tree, if the root of no child is already contained in the view frame, then the view frame must be added as a vertical flat tree to the list of children in the main tree, otherwise the root of the child contained in the view frame is removed and the remaining view frame is added to that child. To do this, we must be able to build a vertical flat tree from a view frame, which is done by the toTree method.

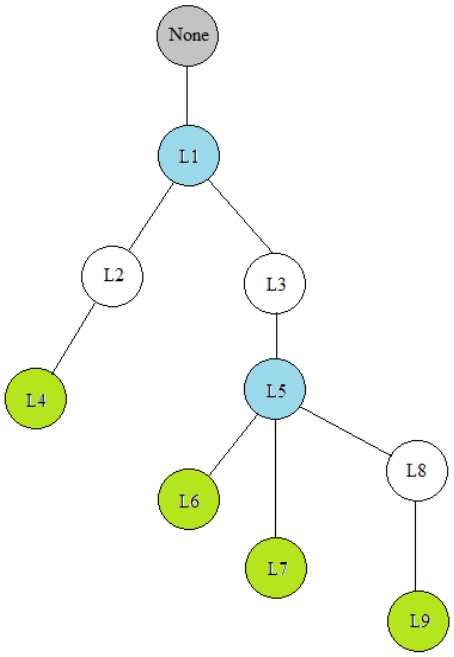


Figure 4 : Tree of test. The root is None by convention. The first branch starting on the left can be read None,L1,L2,L4 and presents our first view frame. The second branch is None,L1,L3,L5,L6, and so on for the remaining two branches. We can notice that the bifurcations are at L1 and L5. The green leaves represent the number of view frames.

A ltreemap has an nbBiff method that calculates the number of bifurcations by deep exploration. It also has the nbLeafs and getViewframes methods which respectively allow to compute the number of leaves of the tree and the number of images (viewframes), while carrying out a deep exploration. Deep exploration is used instead of recursive calls because landmark trees are generally deep, and a recursive method on such trees will result in heap overflows. A first test on a tree

that we build on figure 4 and on the function customize.py is used initially with an __eq__ function comparing two landmarks by an equality and not by the boolean function c(Li,Lj). The equality is less flexible than the boolean function and only works on such custom examples. The purpose of this example is then limited to checking our implementation of the ltreemap. This test confirms the correctness of the algorithm and returns as number of bifurcations 2 and as number of view frames and leaves 4.

See appendices 6.1, 6.2 and 6.3 .

B. Test on a real database

Tests were carried out on a real database that could be found on [5] composed of 1001 images taken from the camera of a vehicle during its navigation.

A first test was performed on an image of the database to observe its keypoints and places of interest where landmarks are placed. The test result is shown in figure 5 with an M=100.

Figure5 : Test image. We notice that the landmarks are concentrated on the car in the center and on the zone delimiting the trees and the horizon. They are also present on the roadsides.

This test aims on the construction of the Landmark-Tree Map corresponding to this database. We make several tests on samples of this database while modifying the thresholds of the image classes, angles and descriptors of the landmark class. The image class threshold controls the minimum response to keep a key point. The thresholds of the angles and descriptors of a landmark are those of the boolean function c(Li,Lj). Thresholds were set to be the smallest possible while maintaining at least two bifurcations on the first ten images.

The processing of the database is done in the main.py function. We process each image of our sample by first extracting its landmarks to get its view frame and then adding this view frame to our landmarks tree. The tests performed on a sample of 50 images gives for number of bifurcations 4 and for number of sheets and view frames 50. The thresholds that were set led to an average depth of 1178.14, which is potentially high compared to the average depth expected for images containing an average of 400 landmarks.

A plot of the vehicle direction vector as a function of the positions x and y has been made, based on the following formula of the navigation vector [3] :

$$\boldsymbol{t} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{l}_i - \boldsymbol{l}'_i)$$

with N the number of visible landmarks and li the unit vector in the reference frame pointing to landmark Li and li' the unit-length measurement vector pointing to the same landmark in the current frame. Note that li the unit vector pointing in the direction of landmark i, is described as a vector of two coordinates : (cos(angle),sin(angle)).

The plotting of this curve didn't give any analysable results, because the positions are defined as cosine and sine, therefore are always between -1 and 1, and furthermore the variations over time are very small.

See appendices 7 and 8.

IV. Discussion and Conclusion

It is true that for mobile robots operating in outdoor environments such as cars, this method based on a landmarks tree has several advantages compared to methods based on metric maps, due to its flexibility regarding memory limitations encountered with other methods.

But better than that, the Landmark-Tree Map allows to distinguish distal and proximal landmarks, to know, based on the order of the landmarks in the tree, which are the most stable positions for the mobile robot. This implementation of

the Landmark-Tree Map also allows to free up memory if necessary by pruning local information which is negligible for long distance navigation.

Based on the tests carried out in this paper, this algorithm gives promising results provided that the number of landmarks drawn on the images is sufficiently large and that the thresholds are well determined.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Mr. Nicolas Cuperlier, for proposing, supervising and guiding this project. I also present my thanks to the whole team of researchers from the ETIS laboratory of CY Cergy Paris université for their help, for their availability and for this opportunity which presents our first step into the world of research.

REFERENCES

[1] Y. Espada, N. Cuperlier, G. Bresson, O. Romain (2019). "From Neurorobotic Localization to Autonomous Vehicles".

[2] M. Augustine, E. Mair, A. Stelzer, F. Ortmeier, D. Burschka, M. Suppa (2012). "Landmark-Tree Map : a Biologically Inspired Topological Map for Long-Distance Robot Navigation".

[3] E. Mair, M. Augustine, B. Jäger, A. Stelzer, C. Brand, D. Burschka, M. Suppa (2014). "A Biologically Inspired Navigation Concept Based on the Landmark-Tree Map for Efficient Long-Distance Robot Navigation".

[4] B. Siciliano and O. Khatib (2008). "Springer Handbook of Robotics, Springer Verlag."

[5] https://cloud-etis.ensea.fr/index.php/s/3iSJYr6bELQcf6y .

```
//Author: Elkhbir Niama
/*#define DEBUG*/
#include <libx.h>
#include <Struct/prom_images_struct.h>
#include <stdlib.h>
#include <string.h>

#include <Kernel_Function/find_input_link.h>
#include <Kernel_Function/prom_getopt.h>
#include <public_tools/Vision.h>

typedef struct tree Tree;
typedef struct list List;
typedef struct viewframe ViewFrame;


// Landmark definition
struct landmark
{
        float[361] angles;
        float[256] descriptors;
};

// Viewframe definition : a list of landmarks
struct viewframe
{
        ViewFrame *next;
        Landmark landmark;
        int size;
};

// Landmark-treemap definition : a tree of landmarks
struct tree
{
    List *childs;
    Landmark landmark;
};

struct list
{
    Tree *node;
    List *next;
};

// Constructors :

Landmark *cons(float[361] *agls, float[256] *descs)
{
        Landmark *elem;
        if ((elem = malloc(sizeof *elem)) == NULL)
                return NULL;
        elem->angles = agls;
        elem->descriptors = descs;
        return elem;
}
```

Appendix 1.1 : Part 1 of f_ltree_map.c function initially written for the Promethe script.

```
ViewFrame *cons(ViewFrame *vf, Landmark *ldmk)
{
    ViewFrame *elem;
    if ((elem = malloc(sizeof *elem)) == NULL)
        return NULL;
    elem->next = vf;
    elem->landmark = ldmk;
    elem->size = vf->size + 1;
    return elem;
}

Tree *cons(List *chlds, Landmark *ldmk)
{
    Tree *elem;
    if ((elem = malloc(sizeof *elem)) == NULL)
        return NULL;
    elem->childs = chlds;
    elem->landmark = ldmk;
    return elem;
}

List *cons(Tree *tree, List *list)
{
    List *elem;
    if ((elem = malloc(sizeof *elem)) == NULL)
        return NULL;
    elem->node = tree;
    elem->next = list;
    return elem;
}

typedef struct data_ltree_map
{
        int angleGpe;
        int descriptorGpe;
        int syncGpe;
        int image_changedGpe;
        Tree *tree;
        ViewFrame previousViewframe;
        ViewFrame viewframe;
        float tx;
        float ty;
} Data_ltree_map;

// Compares the current values of the two structures

bool areEqual(Landmark l1, Landmark l2)
{
        res = true;
        for(i=0;i<361;i++){
                res = res && ((l1->angles)[i] == (l2->angles)[i])
        }
        for(i=0;i<256;i++){
                res = res && ((l1->descriptors)[i] == (l2->descriptors)[i])
```

Appendix 1.2 : Part 2 of f_ltree_map.c function initially written for the Promethe script.

```
bool areEqual(Landmark l1, Landmark l2)
{
        res = true;
        for(i=0;i<361;i++){
                res = res && ((l1->angles)[i] == (l2->angles)[i])
        }
        for(i=0;i<256;i++){
                res = res && ((l1->descriptors)[i] == (l2->descriptors)[i])
        }
        return res;
}

bool hasTheSameVal(Tree tree, ViewFrame viewframe)
{
        return areEqual(tree->landmark,viewframe->landmark);
}

// Verifies if the viewframe is already in the ltreemap

bool viewframe_in_ltreemap(Tree tree, ViewFrame viewframe)
{
        // Generic depth tree exploration : use of stack structure
        List pile = List(tree,NULL);
        ViewFrame copyViewframe = viewframe;
        while(pile != NULL){
                Tree curTree = pile->node;
                pile = pile->next;
                if(curTree->landmark != NULL){
                        curViewframe = copyViewframe;
                        while(curViewframe != NULL){
                                if(hasTheSameVal(curTree, curViewframe)){
                                        curViewframe = curViewframe->next;
                                        break;
                                }
                        }
                }
                List childs = curTree->childs;
                while(childs != NULL){
                        pile = List(childs->node,pile);
                        childs = childs->next;
                }
        }
        return (copyViewframe != NULL);
}

// Appends the viewframe to the ltreemap

void append_viewframe_to_ltree_map(Tree tree, ViewFrame viewframe)
{
        // Generic depth tree exploration : use of stack structure
        List pile = List(tree,NULL);
        ViewFrame copyViewframe = viewframe;
        while(pile != NULL){
                Tree curTree = pile->node;
                pile = pile->next;
```

Appendix 1.3 : Part 3 of f_ltree_map.c function initially written for the Promethe script.

```
void append_viewframe_to_ltree_map(Tree tree, ViewFrame viewframe)
{
        // Generic depth tree exploration : use of stack structure
        List pile = List(tree,NULL);
        ViewFrame copyViewframe = viewframe;
        while(pile != NULL){
                Tree curTree = pile->node;
                pile = pile->next;
                if(curTree->landmark != NULL){
                        curViewframe = copyViewframe;
                        while(curViewframe != NULL){
                                if(hasTheSameVal(curTree, curViewframe)){
                                        curViewframe = curViewframe->next;
                                        break;
                                }
                        }
                }
                List childs = curTree->childs;
                while(childs != NULL){
                        pile = List(childs->node,pile);
                        childs = childs->next;
                }
        }

        Tree subtreeToAdd = NULL;
        while(copyViewframe != NULL){
                subtreeToAdd = Tree(List(subtreeToAdd,NULL),copyViewframe->landmark);
                copyViewframe = copyViewframe->next;
        }
        // Where to add?
}

void new_ltree_map(int gpe)
{
        int i=0;
        Data_ltree_map *my_data = NULL;
        if(def_groupe[gpe].data == NULL){
                my_data = (Data_ltree_map *) malloc(sizeof(Data_ltree_map));
                i = 0;
                l = find_input_link(gpe, i);
                while (l != -1)
                {
                        if (strcmp(liaison[l].nom, "-angle") == 0)
                                mydata->angleGpe = liaison[l].depart;
                        if (strcmp(liaison[l].nom, "-descriptor") == 0)
                                mydata->descriptorGpe = liaison[l].depart;
                        if (strcmp(liaison[l].nom, "-image_changed") == 0)
                                mydata->image_changedGpe = liaison[l].depart;
                        if (strcmp(liaison[l].nom, "-sync") == 0)
                                mydata->syncGpe = liaison[l].depart;
                        i++;
                        l = find_input_link(gpe, i);
                }
                Tree emptyTree = {NULL, NULL};
```

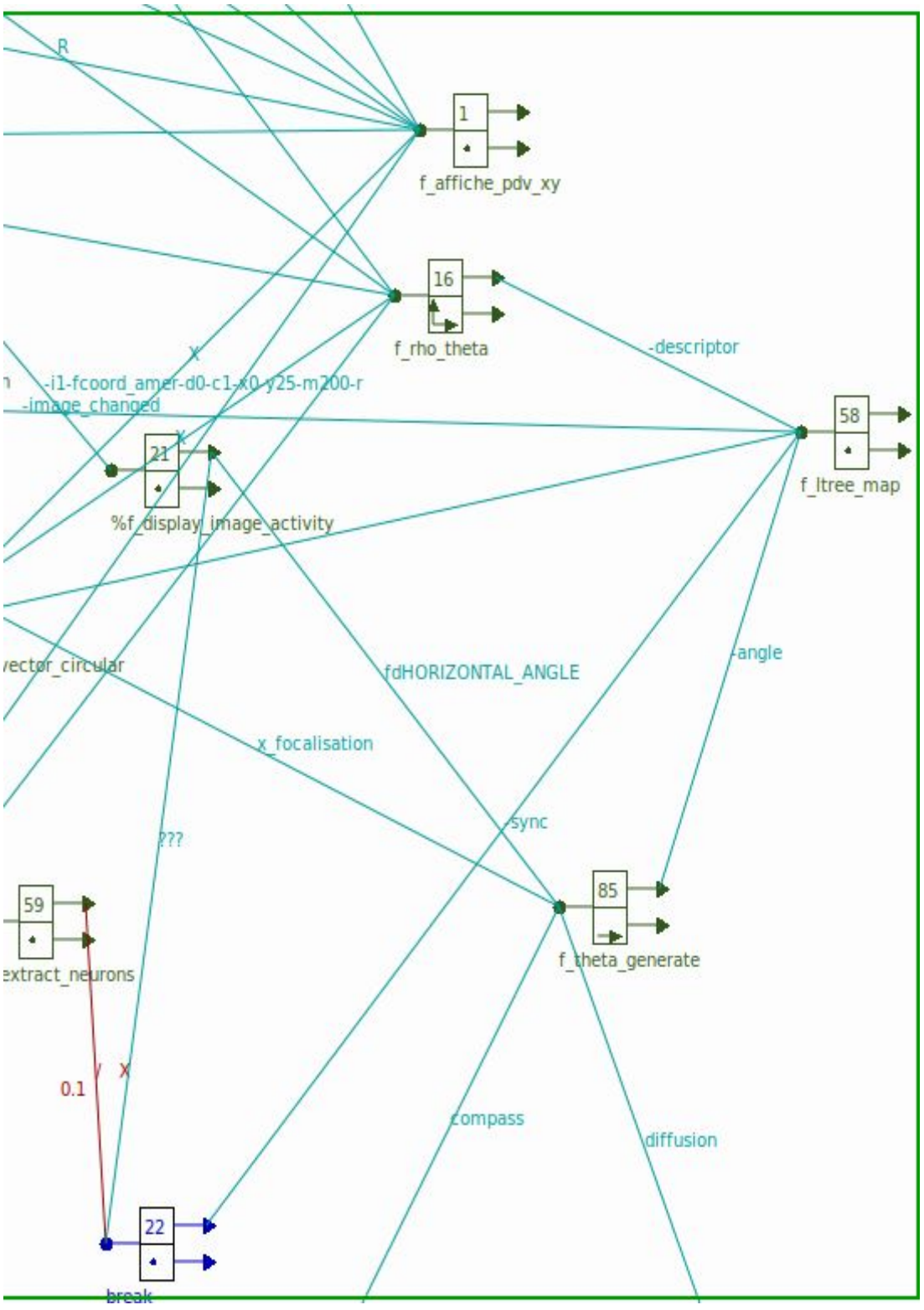
Appendix 1.4 : Part 4 of f_ltree_map.c function initially written for the Promethe script.

```
                }
                Tree emptyTree = {NULL, NULL};
                my_data->tree = emptyTree;
                my_data->previousViewframe = NULL;
                my_data->viewframe = NULL;
                my_data->tx=0;
                my_data->ty=0;
        }
        def_groupe[gpe].data = my_data;
}

void function_ltree_map(int gpe)
{
        int image_changedGpe,syncGpe,angleGpe,descriptorGpe;
        mydata = (Data_ltree_map*) def_groupe[gpe].data;
        Data_ltree_map *my_data = NULL;
        image_changedGpe = mydata->image_changedGpe;
        syncGpe = mydata->syncGpe;
        angleGpe = mydata->angleGpe;
        descriptorGpe = mydata->descriptorGpe;

        if(neurone_s1[def_groupe[image_changedGpe].premier_ele] == 0){
                float[361] angles;
                float[256] descriptors;

for(i=def_groupe[angleGpe].premier_ele;i<def_groupe[angleGpe].premier_ele+def_groupe[angleGpe].nbre;i
+){
                        angles[i] = neurone_s1[i];
                }

for(i=def_groupe[descriptorGpe].premier_ele;i<def_groupe[descriptorGpe].premier_ele+def_groupe[descri
+){
                        descriptors[i] = neurone_s1[i];
                }
                Landmark landmark = Landmark(angles, descriptors);
                mydata->viewframe = ViewFrame(mydata->viewframe, landmark);
        }
        else {
                Tree mytree = mydata->tree;
                ViewFrame myviewframe = mydata->viewframe;
                if(neurone_s1[def_groupe[syncGpe].premier_ele] == 1 && !
viewframe_in_ltreemap(mytree, myviewframe)){
                        append_viewframe_to_ltree_map(mytree,myviewframe);
                }
                // Direction's calculation :
                int N = (mydata->viewframe)->size;
                int previousN = (mydata->previousViewframe)->size;
                float Tx = 0;
                float Ty = 0;
                ViewFrame curX = (mydata->viewframe);
                while(curX != NULL){
                        // Which angle do we have to choose in the angles between 0 and 360 degrees?
                        Tx = Tx + std::cos((//TODO));
                        curX = curX->next;
```

Appendix 1.5 : Part 5 of f_ltree_map.c function initially written for the Promethe script.

```
                // Direction's calculation :
                int N = (mydata->viewframe)->size;
                int previousN = (mydata->previousViewframe)->size;
                float Tx = 0;
                float Ty = 0;
                ViewFrame curX = (mydata->viewframe);
                while(curX != NULL){
                        // Which angle do we have to choose in the angles between 0 and 360 degrees?
                        Tx = Tx + std::cos((//TODO));
                        curX = curX->next;
                }
                ViewFrame curY = (mydata->viewframe);
                while(curY != NULL){
                        // Which angle do we have to choose in the angles between 0 and 360 degrees?
                        Ty = Ty + std::sin((//TODO));
                        curY = curY->next;
                }
                // Same for previous :
                float prevTx = 0;
                float prevTy = 0;
                ViewFrame prevCurX = (mydata->viewframe);
                while(prevCurX != NULL){
                        // Which angle do we have to choose in the angles between 0 and 360 degrees?
                        prevTx = prevTx + std::cos((//TODO));
                        prevCurX = prevCurX->next;
                }
                ViewFrame prevCurY = (mydata->viewframe);
                while(prevCurY != NULL){
                        // Which angle do we have to choose in the angles between 0 and 360 degrees?
                        prevTy = prevTy + std::sin((//TODO));
                        prevCurY = prevCurY->next;
                }
                mydata->tx = (prevTx/previousN)-(Tx/N);
                mydata->ty = (prevTy/previousN)-(Ty/N);
                // Updating :
                mydata->previousViewframe = myviewframe;
                mydata->viewframe = NULL;
        }
}

void destroangleGpey_ltree_map(int gpe)
{
        int i;
        for (i = 0; i < def_groupe[gpe].nbre + 1; i++)
        {
                free(((prom_images_struct *) def_groupe[gpe].ext)->images_table[i]);
                ((prom_images_struct *) def_groupe[gpe].ext)->images_table[i] = NULL;
        }
        free(def_groupe[gpe].data);
        free(def_groupe[gpe].ext);
        def_groupe[gpe].data = NULL;
        def_groupe[gpe].ext = NULL;
        dprints("exiting %s (%s line %i)\n", __FUNCTION__, __FILE__, __LINE__);
}
```

Appendix 1.6 : Part 6 of f_ltree_map.c function initially written for the Promethe script.



Appendix 2 : Extract of coeos on place_cells.symb function. Our f_ltree_map function is directly linked to the function f_theta_generate.c giving angles of the landmark and f_rho_theta giving descriptors of the landmarks.

Landmark.py

```
6  # Globals :
7
8  # Imports :
9  from math import cos, sin, pi
10
11 ##
12
13 class Landmark:
14
15     ANGLES_THRESHOLD = 0.01
16     DESCRIPTOR_THRESHOLD = 0.38
17
18     # angles : list of floats, with size 360
19     # descriptors : list of floats, with size 64
20     def __init__(self , angles = [], descriptor = []):
21         self.angles = angles
22         self.descriptor = descriptor
23
24     def __eq__(self, l):
25         angles_size = len(self.angles)
26         descriptor_size = len(self.descriptor)
27         return isinstance(l,Landmark) and angles_size ==
   len(l.angles) and descriptor_size == len(l.descriptor) and
   abs(1-(self.cos()*l.cos()+self.sin()*l.sin())) <=
   Landmark.ANGLES_THRESHOLD and sum([1 for i in
   range(descriptor_size) if self.descriptor[i] !=
   l.descriptor[i]])/descriptor_size <=
   Landmark.DESCRIPTOR_THRESHOLD
28
29     # Mean cosinus of the angles of self
30     def cos(self):
31         Z = sum([a for a in self.angles])
32         if Z != 0:
33             return sum([self.angles[i]*cos(i*pi/180) for i
   in range(len(self.angles))])/Z
34         else:
35             return 0
36
37     # Mean sinus of the angles of self
38     def sin(self):
39         Z = sum([a for a in self.angles])
40         if Z != 0:
41             return sum([self.angles[i]*sin(i*pi/180) for i
   in range(len(self.angles))])/Z
42         else:
43             return 0
```

Appendix 3 : Function Landmark.py .

Viewframe.py

```
"""
@Author : Niama ELKHBIR
@Last update : 21/03/2020
"""
## TODO :

# Globals :
path = '//home//hp-elitebook-8570p//Desktop//Project//'

# Imports :
from os import chdir
chdir(path)
from Classes.Landmark import *
from numpy import array

##

# The viewframe is an image abstracted as its landmarks
class Viewframe:

    # landmarks : list of landmarks
    def __init__(self , landmarks):
        self.landmarks = landmarks

    # adding a new landmark to self
    def add(self,landmark):
        self.landmarks.append(landmark)

    # direction vector of self
    def t(self):
        N = len(self.landmarks)
        if N != 0:
            return array([sum([l.cos() for l in
self.landmarks])/N , sum([l.sin() for l in self.landmarks])/
N])
        else:
            return array([0, 0])
```

Appendix 4 : Function Viewframe.py .

Image.py

```
@Last update : 25/03/2020
"""
## TODO :

# Globals :
path = '//home//hp-elitebook-8570p//Desktop//Project//'

# Imports :
from os import chdir
chdir(path)
from Classes.Landmark import *
from Classes.Viewframe import *
import cv2
from math import floor

##

class Image:

    N = 41
    THRESHOLD = 50 # mustn't be less than 31!

    def __init__(self, path):
        self.path = path
        self.viewframe = None

    def getKeyPointsAndDescriptors(self):
        img = cv2.imread(self.path)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
        return
cv2.BRISK_create(Image.N).detectAndCompute(gray,None)

    def getViewframe(self):
        if self.viewframe is None:
            kp, descriptors =
self.getKeyPointsAndDescriptors()
            self.viewframe = Viewframe([])
            for i in range(len(kp)):
                if kp[i].response >= Image.THRESHOLD:
                    angles = [0 for i in range(360)]
                    angles[floor(kp[i].angle)] = 1
                    lm = Landmark(angles,
list(descriptors[i]))
                    self.viewframe.add(lm)
        return self.viewframe
```

Appendix 5 : Function Image.py.

Ltreemap.py

```
6
7   # Globals :
8   path = '//home//hp-elitebook-8570p//Desktop//Project//'
9
10  # Imports :
11  from os import chdir
12  chdir(path)
13  from Classes.Landmark import *
14  from Classes.Viewframe import *
15  from numpy import array
16  import matplotlib.pyplot as plt
17
18  ##
19
20  class Ltreemap:
21
22      # root : landmark
23      # childs : list of Ltreemaps
24      def __init__(self , root , childs):
25          self.root = root
26          self.childs = childs
27          self.nbViewframes = 0
28          self.viewframes = []
29          for c in childs:
30              for v in c.viewframes:
31                  self.viewframes.append(Viewframe([root]+v.landmarks))
32                  self.nbViewframes += 1
33
34      # adds the viewframe to self
35      def add(self , viewframe):
36          # Recursivity :
37
38          if viewframe.landmarks == []:
39              return
40
41          if self.childs != []:
42              for c in self.childs:
43                  if c.root in viewframe.landmarks:
44
45                      c.add(Viewframe([l for l in viewframe.landmarks if l !=
    self.root]))
46
47                      self.nbViewframes += 1
48                      self.viewframes.append(viewframe)
49                      return
50
51          flatten = Ltreemap.toTree(Viewframe([l for l in viewframe.landmarks if l
    != self.root]))
52          if flatten != []:
```

Appendix 6.1 : Function Ltreemap.py .

Ltreemap.py

```
50
51          flatten = Ltreemap.toTree(Viewframe([l for l in viewframe.landmarks if l
    != self.root]))
52          if flatten != []:
53
54              self.childs.append(flatten.pop())
55
56              self.nbViewframes += 1
57              self.viewframes.append(viewframe)
58              return
59
60      # Changes a viewframe to a flat vertical tree
61      def toTree(viewframe):
62          res = []
63          n = len(viewframe.landmarks)
64          for i in range(n):
65              res = [Ltreemap(viewframe.landmarks[n-i-1],res)]
66          return res
67
68      # Counts the number of biffurcations in a tree
69      def nbBiff(self):
70          # DE : depth exploration
71          pile = [] #DE
72          pile.append(self) #DE
73
74          nb = 0
75
76          while pile != []: #DE
77              cur = pile.pop() #DE
78
79              if len(cur.childs)>1:
80                  nb += 1
81
82              for c in cur.childs: #DE
83                  pile.append(c) #DE
84
85          return nb
86
87      # Counts the number of leafs in a tree, must be equal to the number of
    viewframes !
88      def nbLeafs(self):
89          # DE : depth exploration
90          pile = [] #DE
91          pile.append(self) #DE
92
93          nb = 0
94
95          while pile != []: #DE
96              cur = pile.pop() #DE
```

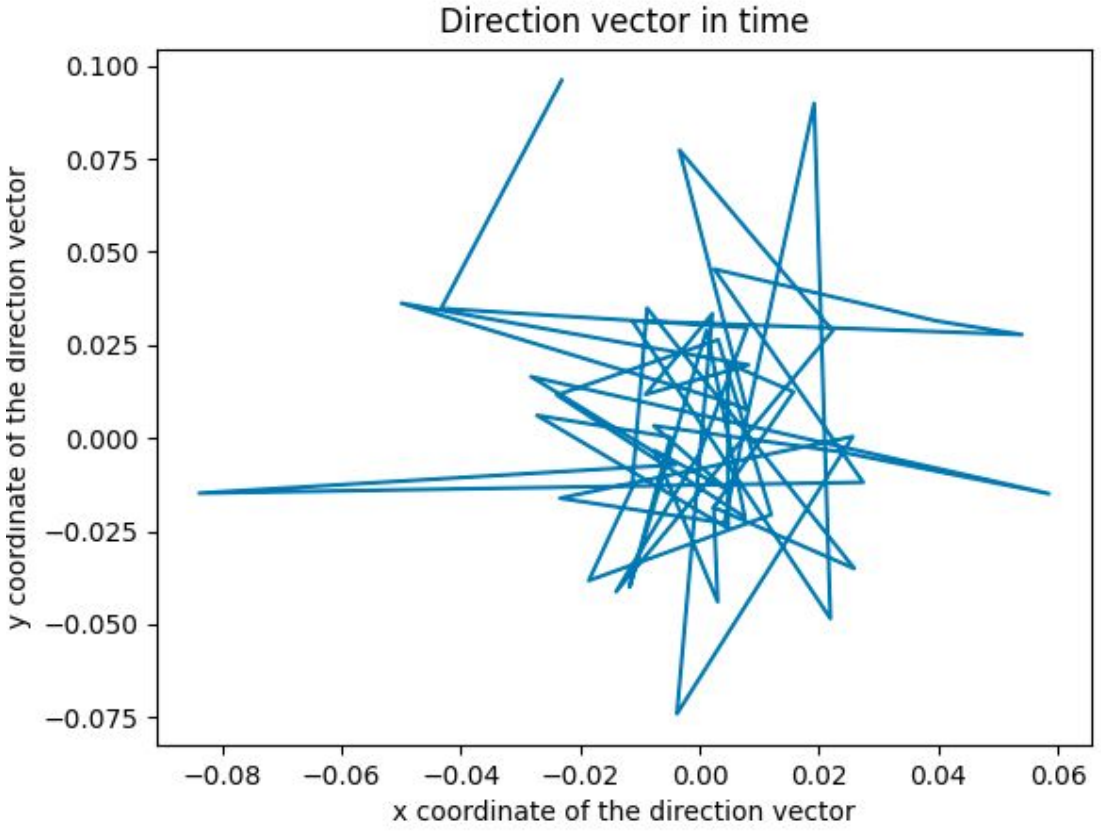
Appendix 6.2 : Function Ltreemap.py .

```
88      def nbLeafs(self):
89          # DE : depth exploration
90          pile = [] #DE
91          pile.append(self) #DE
92
93          nb = 0
94
95          while pile != []: #DE
96              cur = pile.pop() #DE
97
98              if len(cur.childs)<1:
99                  nb += 1
100
101             for c in cur.childs: #DE
102                 pile.append(c) #DE
103
104         return nb
105
106     # Plots the y-coordinates of the direction vector as a function of the x-
    coordinates of the direction vector
107     def plotDirectionVector(self):
108         # X = np.array([i for i in range(self.nbViewframes)])
109         T2 = array([v.t() for v in self.viewframes])
110         T1 = array(list(T2)[1:]+[array([0,0])])
111         T = T2-T1
112         Tx = [t[0] for t in T]
113         Ty = [t[1] for t in T]
114         plt.plot(Tx,Ty)
115         plt.title("Direction vector in time")
116         plt.xlabel("x coordinate of the direction vector")
117         plt.ylabel("y coordinate of the direction vector")
118         plt.show()
119
120     def meanDepth(self):
121         return 1+(sum([len(v.landmarks) for v in self.viewframes]))/
    self.nbViewframes
```

Appendix 6.3 : Function Ltreemap.py .

```
Processing: 46
Processing: 47
Processing: 48
Processing: 49
Done !
Number of biffurcations : 4
Number of leafs : 50
Number of viewframes (static method) : 50
Number of viewframes (dynamic method) : 50
Mean depth : 1178.14
```

Appendix 7 : Test results on a 50 images sample.



Appendix 8 : Direction vector on a 50 images sample. Changes are included in the [-1.1] area.