Report

Simulation based on ray tracing for the modeling of dynamic wideband channel
Localization application

Student: Taguhi CHALUMYAN

Mentor: Bernard UGUEN

Academic year 2010-2011

Master's degree 2 Telecommunications Systems

Words of Gratitude

Firstly, I want to thank my mentor Bernard UGUEN who guided, supported and cooperated during my internship program. Thanks to his advices and recommendations this internship has had vivid progress and development.

Special thanks to Miss Roxana-Elena BURGHELEA and Mr. Stephan AVRILLON for continuous attention and their presence during my work activities. Thanks to their assistance in all professional, technical and personal issues that contributed to find answers and solutions in workplace and to successfully realize my internship program.

I express my deepest gratitude towards my professor Christian BROUSSEAU for guiding me in the professional field and for provision of consistent attention.

I thank Ms. Noëlle Le Ber for her warm attention and advices that empowered me to be integrated not only into the student life, but also in the professional field and day by day activities in laboratory. Positive fillings and good mood empowered and motivated me to further my steps in professional project implementation phases.

I express deepest gratitude to Gerard LAUNAY, Eric POTTIER and Emil OLIVIER who were responsible for formations during the two study years. Thanks for provision of counseling and administrative assistance, as well as many thanks to professional staff at Master Electronics and Telecommunications and Master System of Telecommunications for their excellent and available lecturing.

Thanks to members of jury for both evaluation of the final document and of being present during the work presentation and defense.

I thank very much my mother Paytsar GULIYAN, my sister Gohar CHALUMYAN and my husband Antoine VOISINE for their continuous support and assistance during my study years.

Contents

Table of figure

List of the acronyms

The list below gives the meaning of acronyms used in this manuscript. For reasons of legibility, the meaning of acronyms is generally reminded only its first appearance in the text.

2D	*two dimensions*
3D	*three dimensions*
AoA	*Angles of Arrival*
AoD	*Angles of Departure*
FDTD	*Finite Difference Time Domain*
Gc	*Connection graph*
GPS	*Global Positioning System*
Gr	*Room graph*
Gs	*Structure graph*
Gt	*Topological graph*
Gv	*Visibility graph*
IETR	*Institute of Electronics and Telecommunications of Rennes*
MIMO	*Multiple Input Multiple Output*
OFDM	*Orthogonal Frequency Division Multiplexing*
OG	*Geometrical optics*
RAT	*Radio Access Technique*
RT	*Ray Tracing*
UMTS	*Universal Mobile Telecommunication System*
UTD	*Uniform theory of the Diffraction*
UWB	*Ultra Wide Band*
WiFi	*Wireless Fidelity*
WLAN	*Wireless Local Area Network*
WMAN	*Wireless Metropolitan Area Network*

Abstract

Location sensing systems have become a principal research and development direction for academic and industry people. It is required to have fast and performance localization systems for both outdoor and indoor applications. Some specific examples of indoor applications depending on localization are: location detection of the equipment in a hospital, location detection of firemen in a building on fire and so on. For localization in outdoor environments, the well-known Global Positioning System (GPS) is used for military and civilian applications (commerce, scientific uses, tracking, and surveillance).

RT (Ray Tracing) techniques are commonly used to estimate the propagation channel for different indoor/outdoor localization applications. Such a RT based simulation tool PyRay, was developed before at the Institute of Electronics and Telecommunication of Rennes (IETR). PyRay is a tool which aims at modeling the propagation channel for multi-standard radio systems and/or UWB. It uses a combination of a 3D-RT method and GO-UTD (Geometric Optical - Uniform Theory of Diffraction) in order to calculate the propagation channel matrix which links the transmitted outgoing field to the received incoming field for each ray. PyRay can be used for different applications like monitoring of interferences, positioning, localization applications and so on.

This work proposes two new methods which use the geometry of indoor environment to accelerate the localization. Both of them use data from deterministic channel simulator PyRay. The first method converts indoor representation from 3D to 2D considering that z axis of Cartesian coordinate system is constant. Then, it uses the Theory of Graphs to anticipate the existence of possible rays between the walls and diffraction corners of indoor construction. Finally, using a transmitter and a receiver reference point, allows the identification of all possible ray paths between this transmitter and this receiver. The second method uses a transmitter and a receiver coordinates, identifies the walls and the diffraction corners of the indoor environment which are close to transmitter and receiver locations and eliminates the rest of indoor construction. Then, it verifies the direct visibility relationship (see paragraph 3) and anticipates the existence of possible rays between the transmitter and the receiver.

The comparison of the results from PyRay simulator and the results from the proposed models shows that the results of the used new methods are highly reliable and can be used in different wireless indoor localization applications in order to archive other goals as radio coverage. These methods can be used in channel simulator before RT to facilitate calculation and decrease the time required for simulation.

Keywords: Mobility model, indoor propagation, wireless communication, localization, ray signature, rays anticipation, Ultra Wide Band (UWB), Multiple Input Multiple Output (MIMO).

Introduction

Recently, localization has become one of the primary questions in the field of indoor and outdoor communication systems. It is highly demanded to have new, fast and successful technology which can be used in different applications such as first aid, data transfer, and military applications and so on. There are lots of localization models and devices such as ray tracing, continuous data transfer, and statistical resolution to find the coordinates and so on. One of the main problems in localization is the required time. More and more localization applications require faster working drivers.

A propagation channel simulator PyRay based on RT, was developed before during some thesis and different projects at the IETR. The PyRay platform consists of several blocks which interact among themselves, mostly implemented in Python language, but it also appeals to some external blocks of calculation written in C language. Based on the combination of a 3D-RT method and GO-UTD, PyRay is used for indoor different wireless indoor applications like monitoring of interferences, positioning, localization applications and so on.

The primary objective of this work is to reduce the localization time of a propagation channel simulator. To achieve good results, this work provides two methods which collect all information about the indoor environment construction and anticipate the existence of possible rays between a transmitter and a receiver. This information can be used in an existing channel simulator of indoor environment before RT and short localization time. Both proposed methods use data from deterministic channel simulator PyRay. For geometry fast resolutions Shapely, a BSD-licensed Python package, is used for both models. Shapely package is used for manipulation and analysis of planar geometric objects, the necessary calculation process becomes easier and faster.

The first method is based on the Theory of Graphs and uses NetworkX, which is a Python package. NetworkX is used for the creation, manipulation and study of the structure, dynamics and functions of complex networks. Using NetworkX methods and considering that z axis of Cartesian coordinate system is constant; the indoor representation is converted from 3D to 2D. Thanks to combination of NetworkX and Shapely, possible ray paths between walls and diffraction corners of indoor construction, are found. Lastly, using a transmitter and a receiver the existence of possible rays between this transmitter and this receiver can be anticipated.

The second method uses transmitter and receiver coordinates, identifies walls and diffraction corners of indoor environment which are in the proximity of the transmitter and receiver locations and eliminates the other part of the inside construction. After checking direct

visibility relationship, which is developed during this work, the possible ray paths between the transmitter and the receiver are built by using these calculated ‘‘signatures’’. A ray signature specifies an ordered sequence of points that are crossed by the ray. The shortest ‘‘ray path’’ (the one with the shortest delay) can be taken as a reference in calculating the delay times for the rest of the rays. In fact, the anticipated rays may correspond to real rays in case there is a real wireless communication between the transmitter and the receiver. Thanks to the above mentioned representation of indoor environment, where the transmitter and the receiver are placed, localization is done very quickly. Obtained results are validated with the results of propagation channel PyRay simulator. Comparison results show that the anticipated rays have the shortest delay time, pass through the shortest distance and carry the highest power or the highest energy. New way utilization of geometry in indoor environment facilitates and reduces the simulation time. Nothing limits the use of these methods, in order to achieve other goals like radio coverage.

In the first part of this stage report, the deterministic channel simulation tool PyRay (from where the initial data is taken) is presented. Then, some basic definitions, for the development of this work, are given. Some graph models are also proposed in this research for representing indoor environments. An innovative method for finding out the shortest ray is detailed. The final results are validated by PyRay simulation results.

1. PyRay

1.1. Introduction

This chapter describes the deterministic channel simulation tool PyRay, which is used during this research work. This simulator was developed during several previous thesis and projects, with more focus on theoretical aspects. This tool aims to model radio propagation channel for multi-standard systems and/or UWB. It is essentially used for applications inside buildings, although its use may be extended to outdoor applications. This tool evolves gradually towards a platform of heterogeneous radio simulation, integrating the mobility of radio nodes, in view of its exploitation in diverse application contexts, as localization.

Globally, the PyRay simulator can be divided into three strictly connected parts:

- the first part – the emission (the transmitter, the emitted signal)
- the second part – the wireless propagation channel (the common medium used to transmit information in any system of radio communication)
- the third part - the reception (the receiver, the received signal)

To learn about these three parts of the simulator, see [1].

The architecture of the proposed simulator is detailed in the first section. The second section presents the description of the simulation platform of the propagation channel.

1.2. Architecture of PyRay simulator

This section is dedicated to the description of the PyRay channel simulation tool, developed at IETR.

PyRay is conceived in order to take into account all the physical interactions associated to the electromagnetic waves propagation phenomena (transmission, specular reflection, refraction and diffraction), as well as the properties of the constituent materials of the various walls, rooms and/or furniture inside the propagation environment. Also, the simulator is conceived to take into account the vectorial description of the antennas on a wide frequency band, as well as the possible antenna multi-configurations at the emission and/or the reception part.

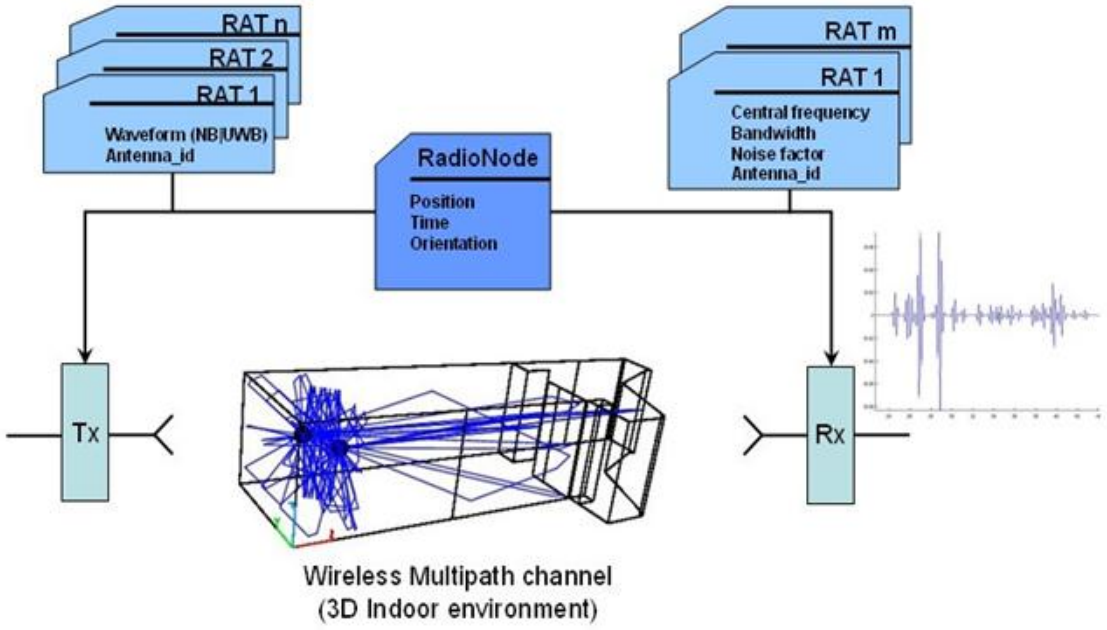


Figure 1.2.1: Architecture of PyRay simulator

The simulator architecture is presented in the figure 1.2.1. PyRay is implemented in Python and C programming languages, by making use of scientific Numpy, Scipy and Matplotlib packages.

The legibility and the simplicity of its syntax, the numerous and varied secondary libraries, the easiness of integration with other programming languages and libraries, are only some reasons among many others that justify the choice of Python for our work development. Moreover, the object-oriented programming language is in perfect accord with our work and allows to build not only specific objects associated to different elements of the radio scene, but also a flexible simulation tool which satisfies all the constraints and specifications of cognitive radio systems.

As shown in figure 1.2.1., the simulator is decomposed in three major parts: the transmission radio node, the multipath propagation channel and the reception radio node. This modular approach offers the possibility to better manage the tool complexity, to isolate propagation channel and antenna effects and to easy modify simulation's particular parameters.

The radio nodes are defined as RadioNode objects and they may denote either a transmitter, either a receiver, without any difference.

The RadioNode class defines the followings specific parameters:

- the spatiotemporal coordinates:

  - Position;
  - Orientation;
  - Mobility Model.
- the RATs (Radio Access Technologies) implemented on the node:
  - Radio standard (WIFI, WiMax, UMTS, etc.);
  - Wave shape (impulse radio, OFDM, etc.);
  - Frequency band.
- the characteristics of the components:
  - Type of antennas;
  - Antenna frequency band;
  - Noise factor.

For a transmitter-receiver particular connection, the propagation channel is described by the RadioLink class and is determined by the 3D-RT (3D-Ray Tracing) method. The 3D-RT method is based on the Geometrical Optics (GO) and the Uniform Theory of Diffraction (UTD). Angular information, attenuation and delay are accessible from the RadioLink class.

The figure 1.2.2 shows the principle of the channel simulator.

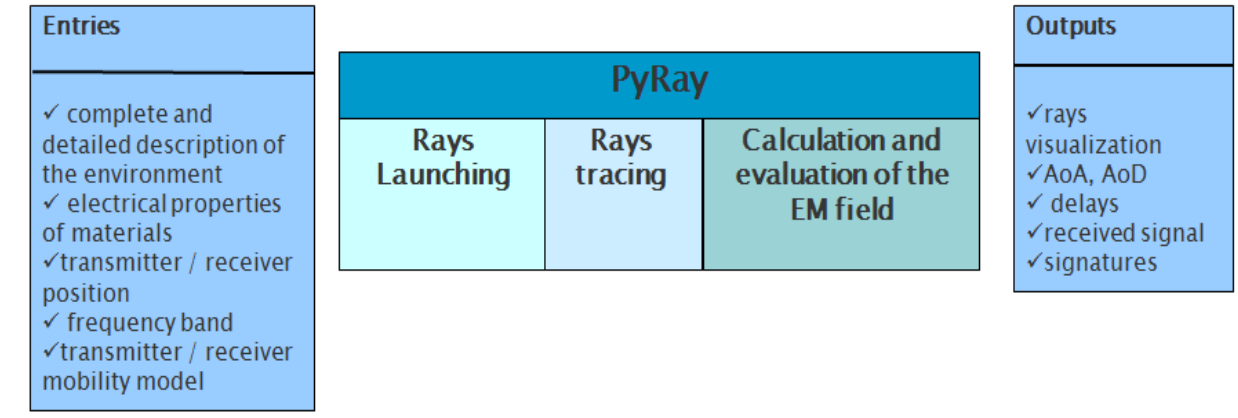


Figure 1.2.2: Principle of PyRay simulator

As shown in figure 1.2.2., the 3D-RT simulation tool needs a detailed description of the environment in which the simulation is realized (including walls and/or furniture constituent materials, transmitter(s) and receiver(s) locations, their mobility models, characteristic radio access technology and antennas), in order to calculate the propagation delays, AoD (Angles of Departure) and AoA (Angles of Arrival) for every considered radio link. These last results are

registered in files and are available and exploitable at any time.

In conclusion, PyRay tool is a particularly well adapted code for developing and testing diverse transmitter-receiver architectures, allowing the integration of several Radio Access Technologies (RATs).

2. The PyRay Simulation Platform

The purpose of this section is to describe the PyRay simulation platform. This platform utilized in this work twice: at the beginning and at the end. Firstly to create these two proposed models the initial data concerning to the indoor environment construction is taken from PyRay Simulation platform. Secondly and finely, the results of these models are validated by a result of a simulation of PyRay.

The PyRay platform consists of a set of functional blocks that collaboratively interact to simulate radio propagation channel. PyRay is mostly implemented in Python language, but it also calls some external programs written in C language. The choice of Python corresponds to the need for having a flexible simulation tool, easy to integrate with other software applications. This programming language not only has a large variety of library modules, but it also offers the possibility to create your own libraries.

Furthermore, the main Simulation concept is defined. The description of the environment detailed.

2.1. Definition of a Simulation

The Simulation class, described on the figure 2.1.1, defines a channel simulation and groups together all the ASCII parameter files are used during the simulation process, as well as the radio scene objects that are necessary for the execution of the simulation:

- files:
  - the configuration file “fileconf” contains the list of all simulation parameter files and their corresponding directories;
  - the “filestr”, “fileslab”, “filemat” files describe the simulation environment: geometry, dimensions, position and structure of walls, physical properties of the constituent materials;
  - the “filepalch”, “filepatra”, “filepatud” files define the parameters to be used in every RT simulation task: ray launching, ray tracing, calculation and evaluation of the

electromagnetic field;

- the “filespaTx” and “filespaRx” files contains spatial information for the transmitters and receivers: position, orientation and mobility model;
- the “fileantTx”, “fileantRx” files are the transmission, respectively the reception antenna file.

- objects:
  - RadioNode denotes either a transmitter, either a reception radio node;
  - IndoorStr denotes the environment of the propagation;
  - GrRay3D denotes a set of 3D rays linking a Tx point to a Rx point.

A Simulation is divided into four successive simulation steps which are modifying it in time. The order in which these simulation steps are executed is very important; previous simulation results influence both next simulation steps and Simulation object.

A Simulation object is characterized by the integer parameter “progress”, indicating the progress of the radio channel simulation process as follows:

- progress = 0: initialization of the simulation
- progress = 1: ray launching
- progress = 2: rays tracing
- progress = 3: calculation of the 3D rays with the Uniform Theory of Diffraction (TUD)
- progress = 4: evaluation of the electromagnetic field

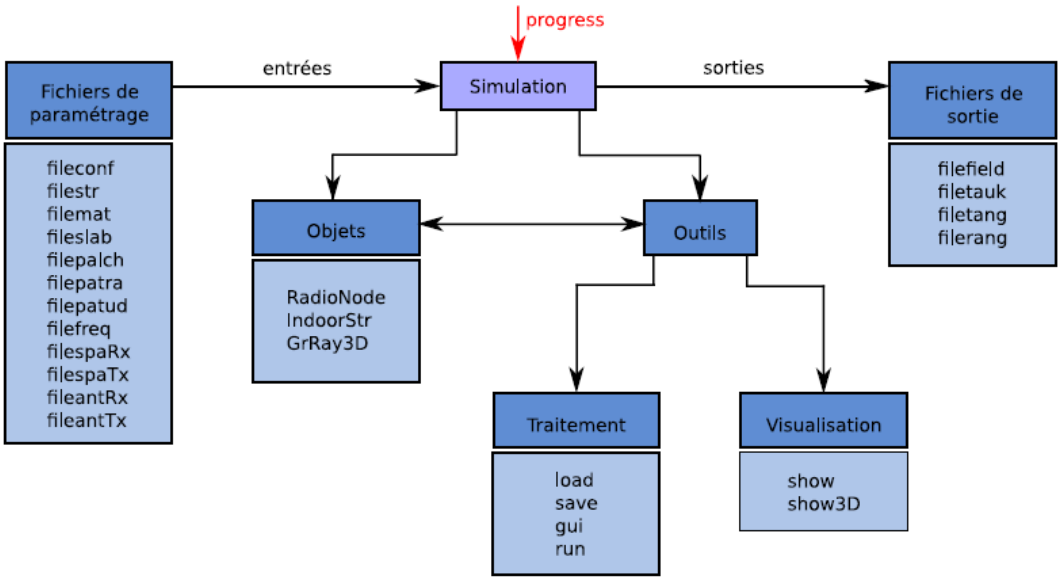


Figure 2.1.1: Definition of a Simulation

A Simulation is complete when progress = 4.

As illustrated in figure 2.1.1., the Simulation class uses some particular methods for data treatment and display, among which we distinguish:

- save: saves a Simulation in “filesimul” file, containing all the names of input and output files associated with this Simulation, as well as the state of the evolution of the Simulation process;
- load: loads a Simulation from a existing “filesimul” file;
- gui: modifies/changes current Simulation parameters;
- run: executes a Simulation. This function chains four simulation steps (ray lauching, ray tracing, 3D-TUD ray calculation and field evaluation). In order to execute “run” function, the user has to specify in the argument of the function, the desired degree in the progress of the simulation, corresponding to the “progress” parameter. The results of every simulation step are written in ASCII files, availables and exploitables at any time:
  - the “filelch” file: contains all rays that are calculated with ray launching algorithm;
  - the “filetra” file: contains all rays that are calculated with ray tracing algorithm;
  - the “filetud” file: contains all 3D-TUD rays;
  - the “filetauk” file: contains the delays of propagation on every ray;
  - the “filetang” and “filerang” file: contains the AoD at the transmitter, respectively AoA at the receiver for each ray;
  - the “filefield” file: is obtained after the evaluation of the field.

The figure 2.1.2 shows the overview diagram for the method “run” of the Simulation class. When using “run” method, it is very important to make sure that all the previous simulation steps were realized before executing the advanced ones. For example, we cannot proceed to the calculation of the 3D-TUD rays without ending ray launching and ray tracing simulation steps before.

- show: 2D display of rays and environment of simulation;
  - show3D: 3D display.

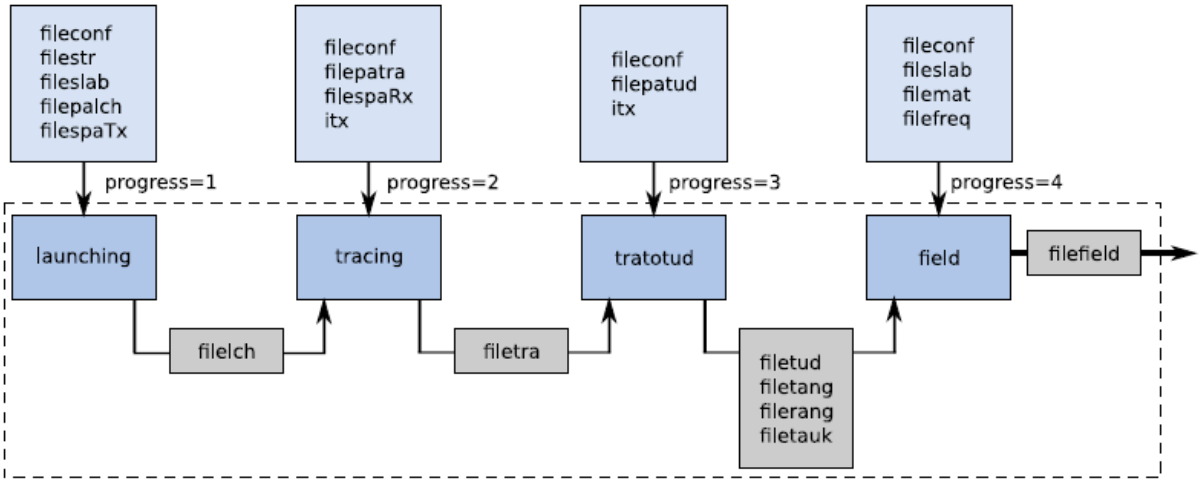


Figure 2.1.2: Run function of the Simulation class

2.2. Description of the environment

The modeling of the propagation environment is made using objects of IndoorStr class type. These objects are built from several files: “filestr” of geometrical data, “fileslab” of walls thicknesses and layers and “filemat” of materials and their corresponding physical properties. The relationship between these files is explained in figure 2.2.1.

The “filestr” file contains geometric details of the propagation environment, such as:

- the number of points and their coordinates expressed in meters;
- the number of segments (walls), their start and end nodes, their lengths, the indications in connection with the “fileslab” file;
- the number of co-segments (if doors, windows, etc.), the segments on which they are, their gaps with regard to the axes of segments, their lengths, the indications in connection with the “fileslab” file.

The geometrical model of the environment first requires a 2D description. The 3D representation follows this 2D structure, by introducing the concept of ground, ceiling and co-segment. Then, every wall of this 3D environment is modeled as an assembly of homogeneous, flat dielectric layers, defined in “fileslab” file by specific parameters, such as:

- the number of layers and their thickness;
- the indications in connection with the “filemat” file;
- the name and the color used for the graphic representation.

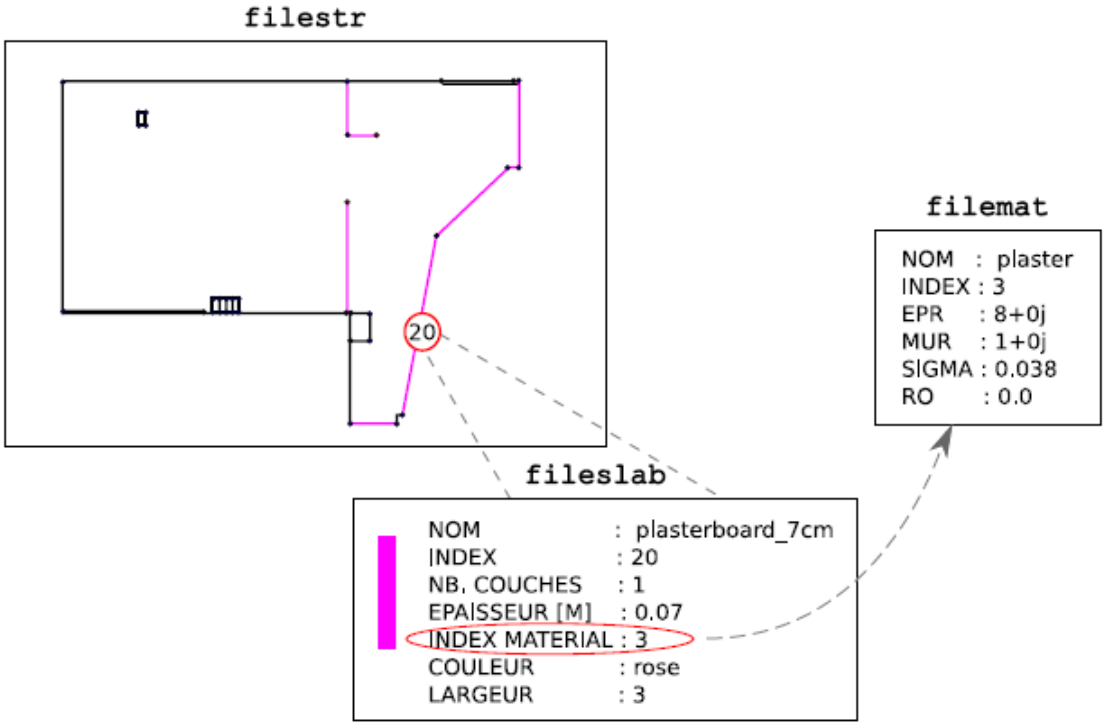


Figure 2.2.1: Modeling the propagation environment

This multi- layer model is a way of approximation for the possible discontinuities of the building materials which may have complicated stratified structures (reinforced concrete, tiled floor, etc.). Layers referenced in the "fileslab" file are constituted by materials whose electrical characteristics are given in the "filemat" file, such as:

- the relative permittivity ε_r, the relative permeability μ_r, the electric conductivity σ, the roughness factor ρ;
- the reflection and transmission coefficients;
- the name.

Electrical characteristics of materials and their frequency dependence are grouped in the Layout class developed during this work.

The table 2.2.1 groups together some of the most commonly used materials in indoor environment modeling and gives their characteristics.

Id	Material	ε_r	μ_r	σ	Color
-1	Metal	-1-1j	1+1j	1	
1	Air	1+0j	1+0j	0	
2	Brick	4.1+0j	1+0j	0.3	
3	Plaster	8+0j	1+0j	0.038	
4	Glass	3.8+0j	1+0j	0	
5	Concrete	5.5+0j	1+0j	0.487	
6	Reinforced concrete	8.7+0j	1+0j	3	
7	Wood	2.84-0.02j	1+0j	0	
8	Pierre	8.7+0j	1+0j	3	

Table 2.2.1: Characteristics of some usual materials

For wood material, figure 2.2.2 presents the dependence of the reflection and transmission coefficients on the incidence angle θ in the case of 2.4 GHz frequency.

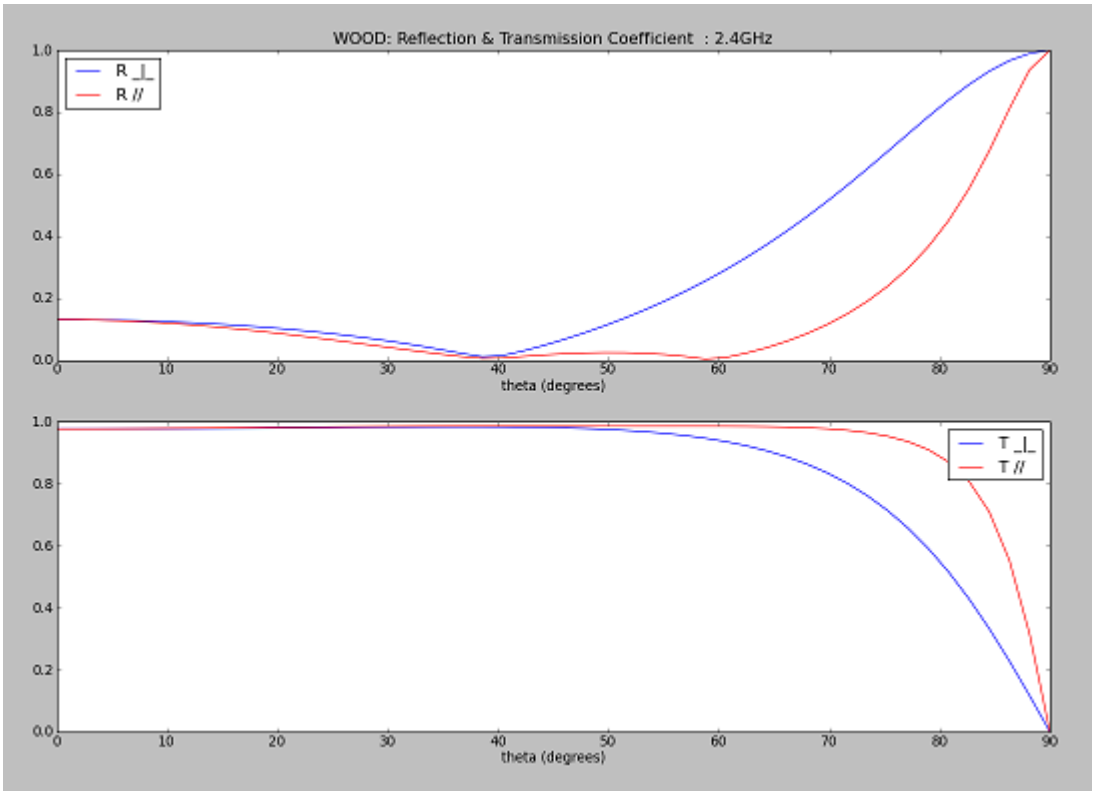


Figure 2.2.2: Dependence of the reflection and transmission coefficients on the incident angle θ in case of 2.4 GHz frequency, for wood material.

Figure 2.2.3 gives an example of a 2D and 3D representation of a propagation environment simulated with PyRay.

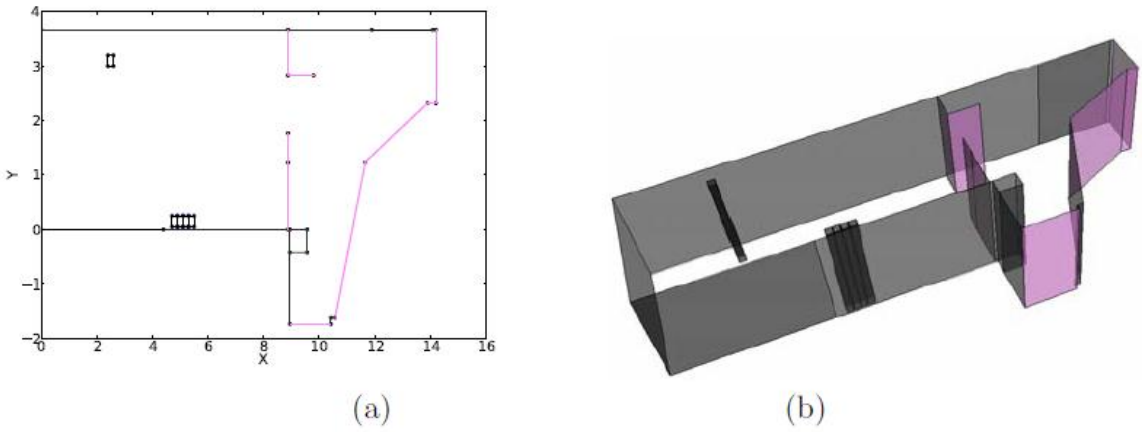


Figure 2.2.3: The propagation environment of type IndoorStr object, corresponding to the simulation: (a) 2D display; (b) 3D display

The modeling of the propagation environment is a key factor that influences any RT-3D based method of estimating radio propagation channel [5]. The description of the propagation environment requires compromise solutions in order to provide realistic representations that are compatibles with the constraints imposed by the simulation tool and also by the physical model on which they are based.

3 Concepts and Tools

Several concepts/tools used in our research work, such as Direct/Indirect Visibility Relationship, Signature, and Allowed Mobility Zone, are defined in this chapter. These basic definitions are essentials in our future development work, as they contribute to increase further work effectiveness and to faster its accomplishment.

In order to easily introduce these concepts/tools, a model of a 3-room indoor environment is used in the 2D measurement system (see figure 3.1). In this indoor environment, the rooms are polygons, numerated N1, N2 and N3. In each room every wall segment is also numerated using positive numbers. The diffraction corners are numerated using negative numbers. The sub-segments are colored with red color to foster comprehension.

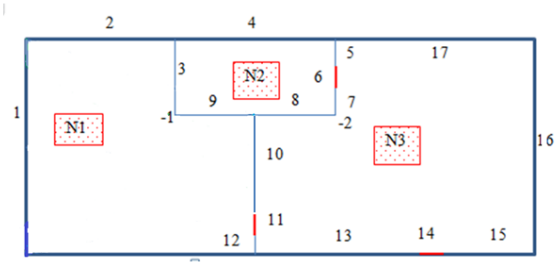
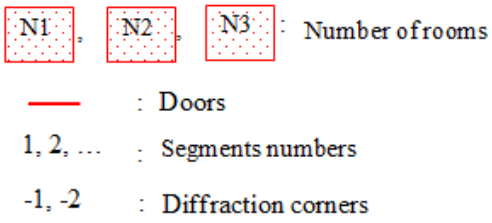


Figure 3.1: The 3-room indoor environment structure.



3.1 Direct visibility relationship between two walls

The direct visibility relationship between two walls is defined using the 3-room indoor environment from figure 3.1.1. In this particular environment, “-1” and “-2” diffraction corners are considered. These diffraction corners play an important role especially in N1 and N3 rooms; that’s why no special attention is given to them in N2 room.

For every particular pair of walls, a polygon is built as follows: the extremities of the walls are connected; the polygon diagonals as well as the middle points of the walls are connected. After defining a tolerance length and identifying those points which are situated in the the extremities of the walls at a distance equivalent to the tolerance length, a new polygon is built and its diagonals are drawn.

The direct visibility relationship between two walls is defined as a possibility of having at least one complete geometric connection belonging to the polygon formed by the walls/diffraction corners coordinate for each room (see figure 3.1.2).

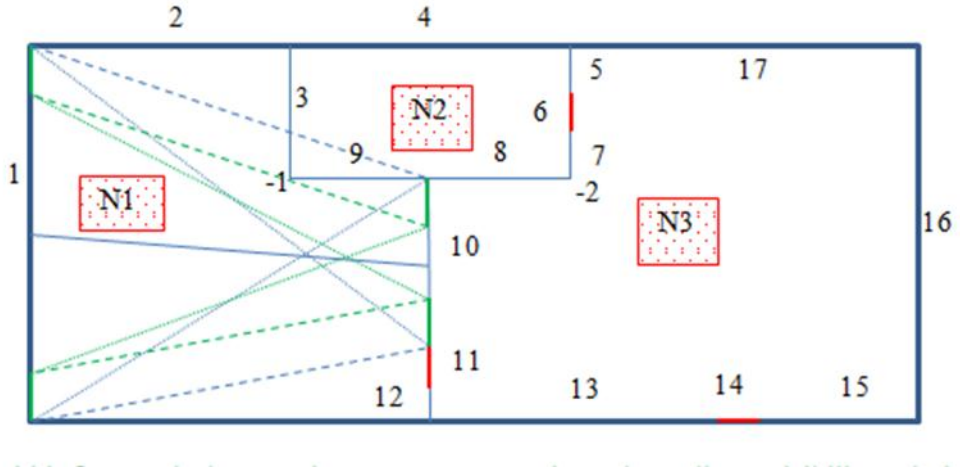


Figure 3.1.1: Check the direct visibility relationship between two walls of the 3-room indoor environment.

:

N1, N2, N3 : Number of rooms

: Doors

: Tolerance length

: Connection of middle points of the walls

: Connection of the extremities of the walls

: Connection of the extremities of the walls built by using the tolerance

: Connection of the diagonals of the polygon

: Connection of the diagonals of the polygon built by using the tolerance length

3.2 Direct visibility relationship between two diffraction corners

A direct visibility relationship between two diffraction corners is defined as a straight line existing between the two points that does not cross with the other infrastructures of the indoor structure (such as walls, doors and so on).

3.3 Direct visibility relationship between a wall and a diffraction corners

The direct visibility relationship between a wall and a diffraction corner is defined using the 3-room indoor environment model shown in figure 3.1. In this case, wall boundaries and middle points are first linked to the diffraction corners. Then, the points which are located in the extremities of the walls at a distance which corresponds to the tolerance length are identified, and linked to the diffraction corners (see figure 3.3.1)

Visibility relationship between a wall and a diffraction corner is named for the cases where one of these geometric connections is completely allocated within the polygon.

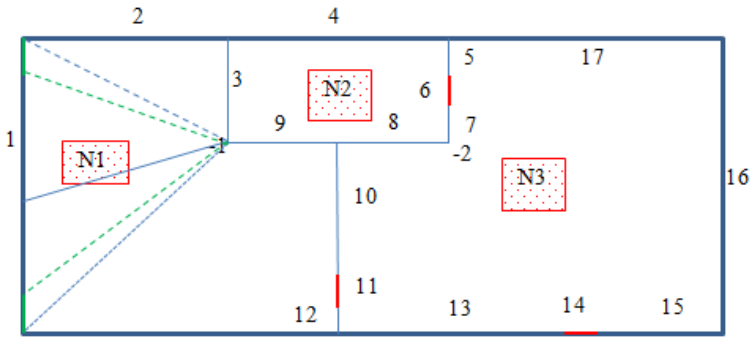


Figure 3.3.1: Check the direct visibility relationship between a wall and a diffraction corner of the 3-room indoor environment.

N1, N2, N3 : Number of rooms

: Doors

: Tolerance length

: Connection of middle points of the wall to the diffraction corner

: Connection of the extremities of the wall to the diffraction corner

: Connection of the extremities of the wall to the diffraction corner built by using the tolerance length

3.4 Indirect visibility between two walls and/or between a wall and a diffraction corner

Checking indirect visibility is based on data from direct visibility that correspondingly shows direct visibility among segments, sub-segments and diffraction angle belonging to each room.

As an example, if in the room N1 the condition for the direct visibility is satisfied between wall number “1” and wall number “10”, as well as it is satisfied in the room N3 between the wall number “10” and the wall number “16”, then it can be said that the wall number “1” and the wall number “16” are checking the indirect visibility condition.

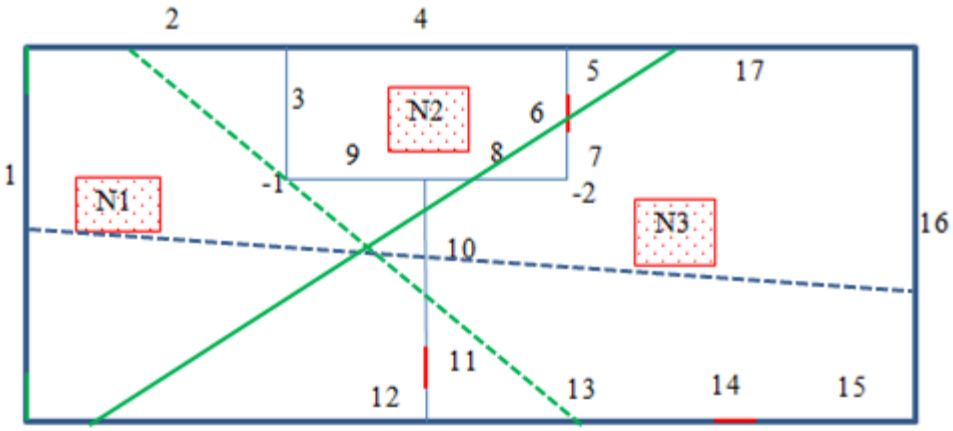


Figure 3.4.1: An example of indirect visibility relationship.

: Shows indirect relationship between wall 12 and 17

: Shows indirect relationship between wall 2 and 13

: Shows indirect relationship between wall 1 and 16

In figure 3.4.1, there is an indirect visibility relationship between wall number “2” and wall number “13” that is expressed via diffraction angle “-1” and wall number “10”, because of direct visibility between wall number “2” and wall number “10”, then direct visibility between wall number “10” and wall number “13”.

3.5 Signature

Let's us consider a transmission point Tx located nearby the wall number "1" and a reception point Rx located nearby the wall number "16" of the indoor environment shown in figure 3.1. In this case, it would be possible that the emitted rays pass through walls number "1", "10" and "16" before arriving to the receiver (see red ray in Figure 3.5.1).

Thus, various rays between Tx-Rx can be built based on direct visibility relationship and then based on indirect visibility relationship.

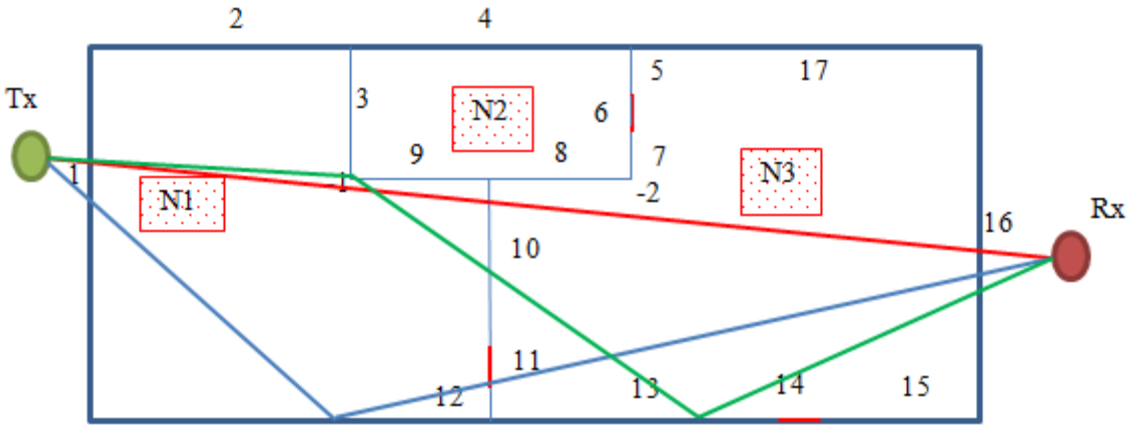


Figure 3.5.1: Finding out indirect visibility between walls and between walls and diffraction corners in 3-room indoor environment.

Indirect visibility is among the environment elements which could be crossed by an emitted ray.

In figure 3.5.1, a ray signature is obtained for a given transmission-reception pair, as a succession of integer numbers associated to all structures that are crossed by the ray. For example, the blue ray emitted from Tx transmitter passes through wall number "1", then it is reflected on the wall number "12", then passes through wall number "11", intersects wall number "16" and arrives at the receiver Rx . So, the blue ray's signature is: Tx, 1, 12, 11, 16, Rx. The green ray signature is: Tx, 1, -1, 10, 13, 16, Rx.

Therefore a signature contains full information about all the interactions that the ray has had during its route between the transmitter and the receiver.

3.6 Creation of rays using signatures

For a given indoor environment, new signature sequences can be obtained by the use of direct and indirect visibility relationships. Afterwards, a new set of ray signatures can be built using the theory of Geometrical optics and environment geometry information, such as size, shape and orientation. An example is given in figure 3.6.1.

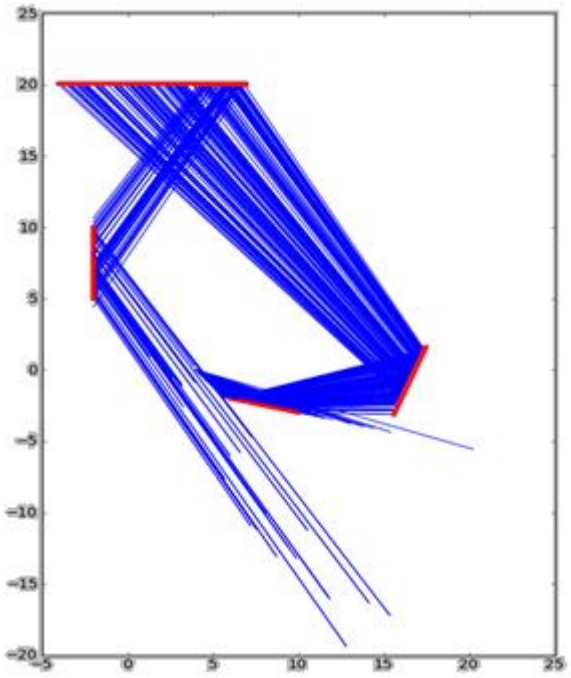


Figure 3.6.1: Building possible ray paths in an indoor environment with four walls.

3.7 Allowed Mobility Zone

In situations where transmitters and/or receivers are allowed to move freely in the indoor environment, the Allowed Mobility Zone notion is introduced. The Allowed Mobility Zone is based on the geometric parameters and position of indoor environment structure and uses the NetworkX and Shapely packages for increasing the effectiveness and making faster the proposed model.

3.8 Allowed Mobility Zone in Rooms

To decide the Allowed Mobility Zone indoor environment construction is used. For each room which presented as a polygon the coordinates of the polygon are taken. Then a length d=1cm is defined and all the coordinates of the points belonging to the polygon are recalculated

based on that length. Figure 3.8.1 shows the Allowed Mobility Zones for two rooms. For point P_1, as well as all other points that form a polygon, P_1' and other points are calculated and a new polygon is built based on the set of P_1' and other points. The newly built polygon represents the Allowed Mobility Zone for a room.

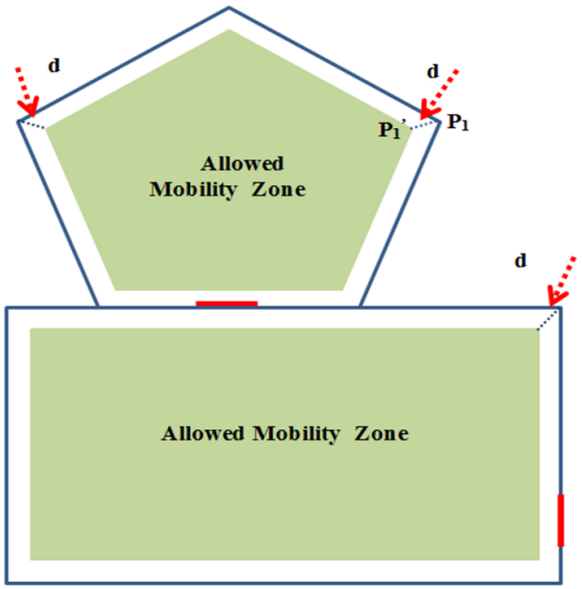


Figure 3.8.1: Presentation of Allowed Mobility Zone: The blue outlines shows two rooms, the Allowed Mobility Zone is colored in green and the doors are presented in red color.

By application of the function that classifies the Allowed Mobility Zone for all indoor environment, the Allowed Mobility Zones for all rooms can be found.

But the set of polygons provides movement opportunity for the transmitter and/or receiver only within the rooms and doesn't allow free movement from one room to another.

To allow the transmitter and/or receiver to move within the entire indoor environment (not only in the specific room, but also from one room to another throughout the doors) the Allowed Mobility Zone is defined for all the doors in indoor environment.

3.9 Allowed Mobility Zone between Doors

For each room the door coordinates included in that room are taken. To decide the position of the doors in indoor environment, than move from extremities l = 5cm and by the use of Shapely function, a polygon is received. The polygon includes the length of the doors thus a part of not allowed zone of rooms is also included in required part of the Allowed Zone.

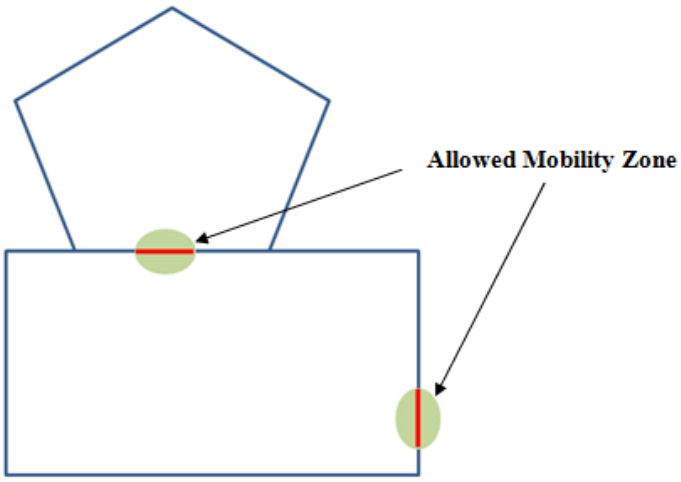


Figure 3.9.1: Presentation of the Allowed Mobility Zone between the doors. The blue outlines show two rooms, Mobility Allowed Zone is colored in green and the doors are presented in red color.

3.10 Allowed Mobility Zone

By adding the Allowed Mobility Zones in the rooms to the Allowed Mobility Zones within the doors, there is the complete required Allowed Mobility Zones within entire indoor environment. Figure 3.10.1 presents the entire Allowed Mobility Zone.

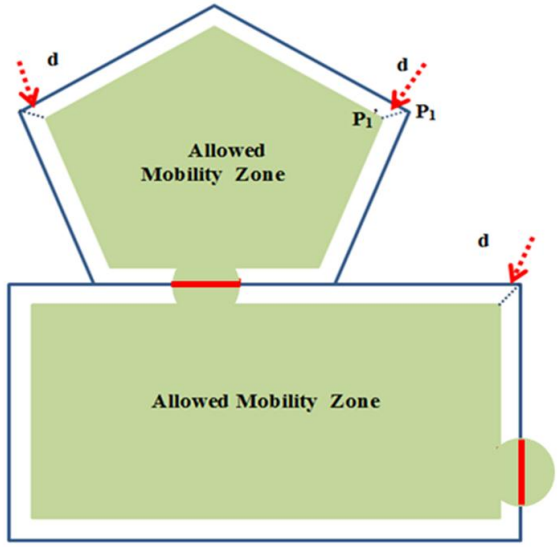


Figure 3.10.1: Presentation of the Allowed Mobility Zone: The blue outlines present the two rooms, the Allowed Mobility Zone is colored in green and the doors are presented in red color.

4 Layout

Propagation environment is modeled in the Layout class. This class is created by data from propagation canal simulator PyRay. All information of indoor environment description, presented before in PyRay description, is stored in this class and is available at every moment. One part of this registered data like indoor construction coordinates is used for the further development of this work. Whereas the second part of this information is not used, it remains accessible at every moment. This kind of solution allows Layout class to be independent and to work quickly. Layout class provides full information about the propagation environment. NetworkX package is used in order to implement Layout class methods. Thanks to NetworkX faster methods, propagation environment information is available at any time if needed.

A part of saved information in Layout class is shown in Figure 4.1.

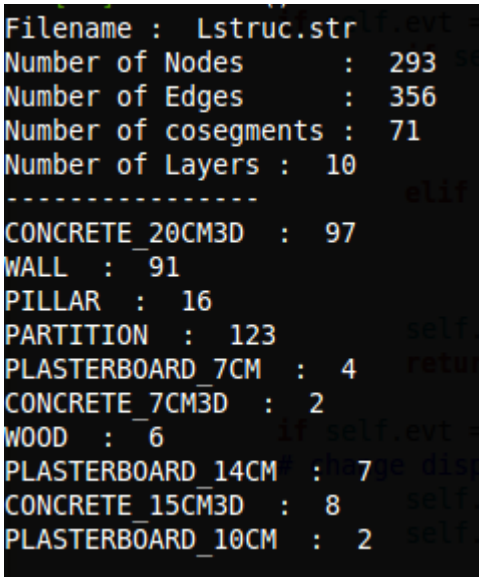


Figure 4.1: One example of saved information in Layout class.

Layout class contains main geometric data of propagation environment, like: number of nodes and their location coordinates expressed in meters, number of segments, their start and end points and their geometry, number of co-segments (doors, windows and so on), the segments on which they are located.

This class also contains several graphs, such as:

- *Gs* - Structure graph which is a 2D-representation of the indoor environment structure.
- *Gc* - Connection graph which presents viability relationships between Gs nodes and edges.
- *Gt* - Topological graph which indicates topological relationships between the rooms.
- *Gr* - Room graph which specifies the cycles of Gt graph, which contain at least one door.

- *Gm* - Mobility graph which points out the connectivity relationship between Gs rooms.
- *Gv* - Visibility graph which specifies all direct and indirect visibility relationships among Gs nodes and edges.

Some of the above mentioned graphs are presented in this work in detail and some of them are presented briefly.

4.1 Structure graph (Gs)

The structure graph Gs is a representation of the indoor environment, including all internal elements and their characteristics (like in the above-mentioned PyRay simulator). The NetworkX package is used as support to model Gs graph. Every Gs segment and sub-segment is numerated, as follows: walls are numerated by positive numbers and corresponding corners are numerated by negative numbers. This type of numeration gives an access to segment and sub-segment different characteristics, such as material, shape, size, and location.

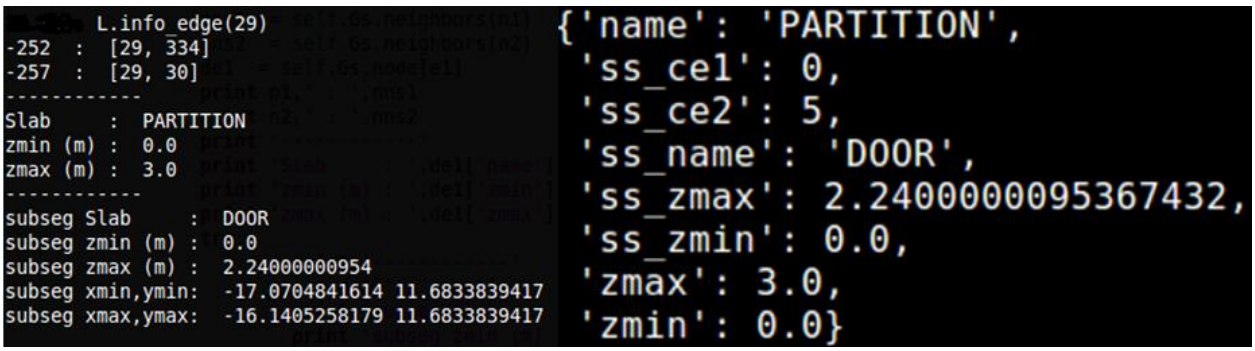


Figure 4.1.1: Example of information associated to Gs node number "29".

.

Figure 4.1.1 gives a particular example of Gs data associated to one segment number. In case of node "29", that belongs to a slab named Partition, its coordinates and neighbors' node numbers are displayed. This sub-segment is called Door and its height is 2.24m.

This indoor environment is exactly defined and numerated by 22 materials (figure 4.1.2). These materials can be found in different elements of indoor environment.

```
{-1: 'METALIC',
 0: 'ABSORBENT',
 1: 'AIR',
 2: 'WALL',
 3: 'PARTITION',
 4: 'WINDOW',
 5: 'DOOR',
 6: 'CEIL',
 7: 'FLOOR',
 8: 'WINDOW_GLASS',
 9: 'WOOD',
 10: '3D_WINDOW_GLASS',
 11: 'WALLS',
 12: 'PILLAR',
 13: 'METAL',
 14: 'CONCRETE_15CM3D',
 15: 'CONCRETE_20CM3D',
 16: 'CONCRETE_6CM3D',
 17: 'CONCRETE_7CM3D',
 18: 'PLASTERBOARD_10CM',
 19: 'PLASTERBOARD_14CM',
 20: 'PLASTERBOARD_7CM'}
```

Figure 4.1.2: Definition and numeration of materials used in indoor construction.

As in PyRay simulator case, UWB frequency dependence of each material properties is considered.

An example of an indoor environment representation using NetworkX package is displayed in figure 4.1.3(a), and the Gs nodes of the indoor environment are presented in figure 4.1.3(b).

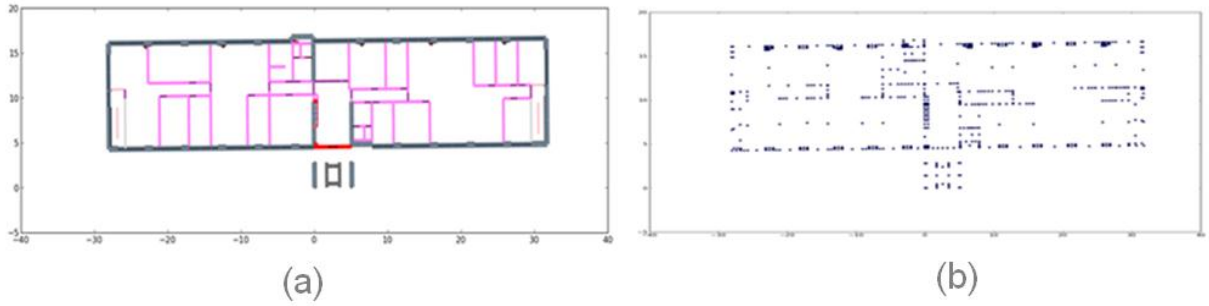


Figure 4.1.3: (a) Indoor environment plotted using NetworkX, (b) Indoor environment plotted using Gs nodes.

This kind of indoor structure modeling, as well as different geometric solutions make easier calculations and allow reducing the simulation time, thus making shorter the time required for localization.

4.2 Connection graph (Gc)

This graph indicates the visibility relationship among the nodes and the edges of the indoor structure. Thanks to Gc graph it is easily possible for each node and edge to find their nearest set of nodes and edges. It defines the Gs graph nodes depending on increase of distance firstly the nearest nodes, then near nodes and far nodes or edges. For any point considering isotropy case (when it works the same way for all the directions) all other nodes and edges are classified by the increase in distance. Based on limitations of this task our work is limited with the use of the first 20 nodes/edges. Thanks to this graph, in case of given transmitter and receiver near nodes and edges are found in short period of time in the indoor environment.

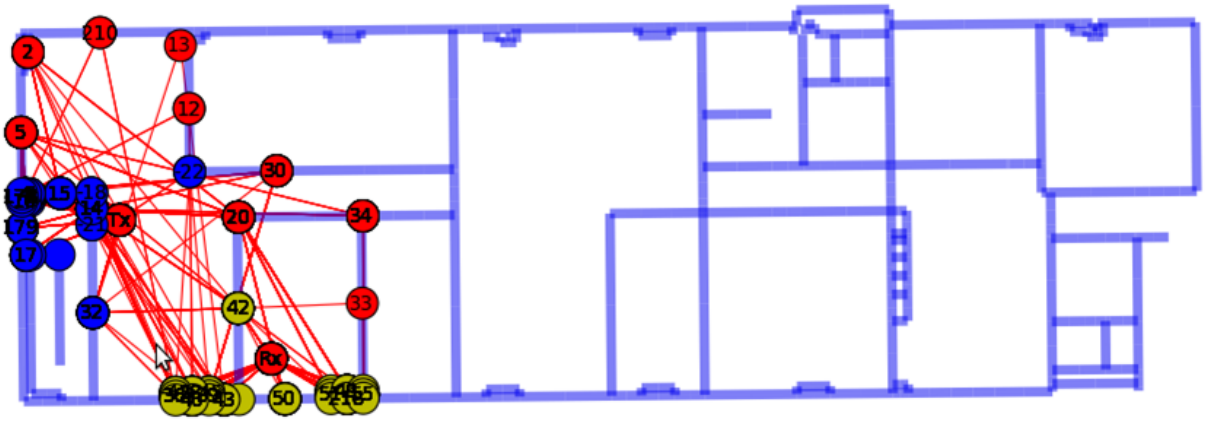


Figure 4.2.1: Near 20s (for each node/edges) for the given transmitter and receiver. The nearest points from Tx are colored in blue, the nearest points from Rx are colored in yellow, and the red ones are the intermediate points (the distance from these points is not the shortest one).

4.3 Topological graph (Gt)

The topological graph (Gt) indicates topological relationship between Gs graph cycles. It means information about location and the connectivity of every node in Gt cycles is supplied. The Gt cycles are formal combinations of closed inflexible of a given plan.

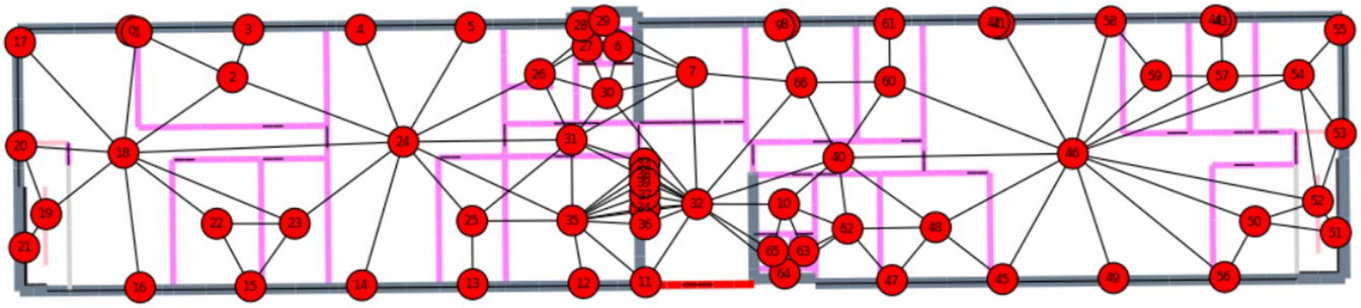


Figure 4.3.1: Gt: The red nodes are Gt cycles and black lines show the connection between the Gt cycles.

4.4 Room graph (Gr)

The room graph is founded on Gt graph. The cycles of Gt graph, which contains at last one door, is included in Gr graph and named as a room. In figure 4.4.1 a Gr graph of our indoor model is presented.

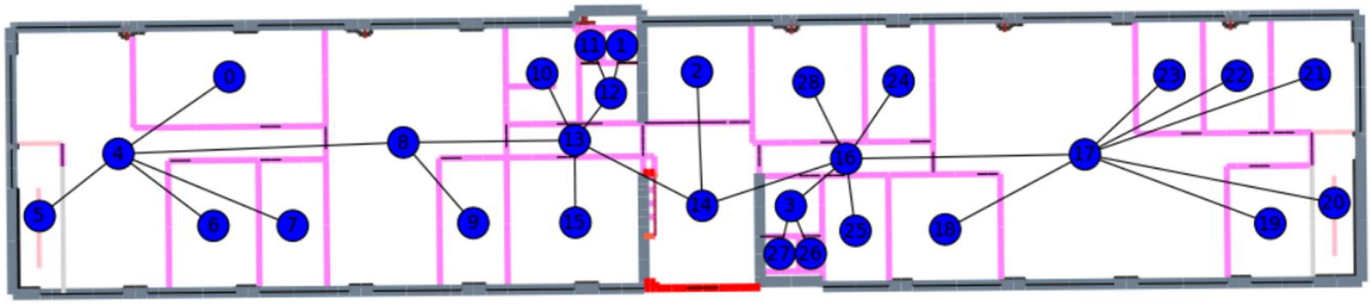


Figure 4.4.1: Gr. The blue nodes are Gr rooms and the black lines show the connection between the Gr rooms.

4.5 Presentation of a piece of indoor environment

A piece of indoor environment is presented in figure 4.5.1, to illustrate together all graphs presented before and to show the coherence among them.

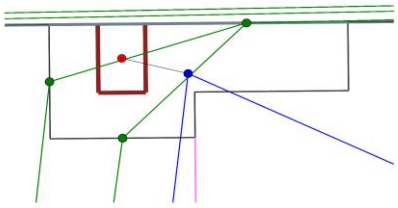


Figure 4.5.1: Gs, Gt, Gr presentation. The blue node presents a "singroom" it means a node of Gr graph, the green node is a possible "signatures" between transmitter and receiver, and the red node is a Gt cycle.

This piece is constructed by using Gs nodes. The red node presents a Gt cycle, which is a formal combination of closed object. The blue node corresponds to a Gr node which its turn corresponds to a Gt node which contains at least a door. Green nodes are related to the walls middles. The middles of walls are chosen to only facilitate the displaying of the graph. They give existence of possible rays between a transmitter and a receiver by using obtained ''signatures'' defined before.

This presentation gives a possibility to find possible routes of the rays. Fi there were a transmitter and a receiver some possible rays between them would follow routes defended by a "singroom". A singroom specifies an ordered sequence of rooms that are passed by the ray.

4.6 Visibility graph (Gv)

Gv graph contains all direct visibility relationships. In Gv graph, visibility relationship is composed of 3 connection types:

The first connection type links the walls;

The second one connects the diffraction angles;

The third one joins walls to diffraction angles and/or joins diffraction angles to walls.

And visibility relationship between a wall and diffraction angle is named for the cases where one of these geometric connections is completed in a polygon formed by Gs nodes which belong to a Gt node.

One example of a Gt node is presented in figure 4.6.1.

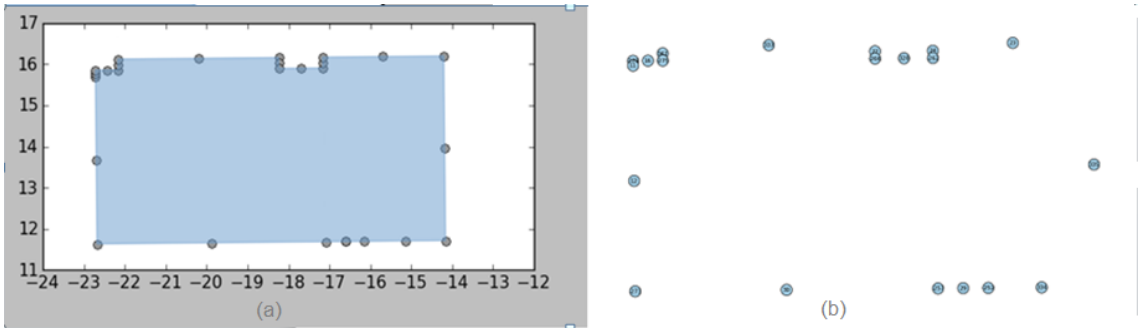


Figure 4.6.1: (a) Presentation of a Gt node by a polygon (b) Presentation of a Gt node by the numbers of Gs nodes.

There is a limitation of node numbers in the construction of Gv graph. That is, only the walls and diffraction angles are used to build Gv graph. In every Gt graph node, for every Gs node the visibility relationship with other nodes: walls and diffraction angles are found (example figure 4.6.2).

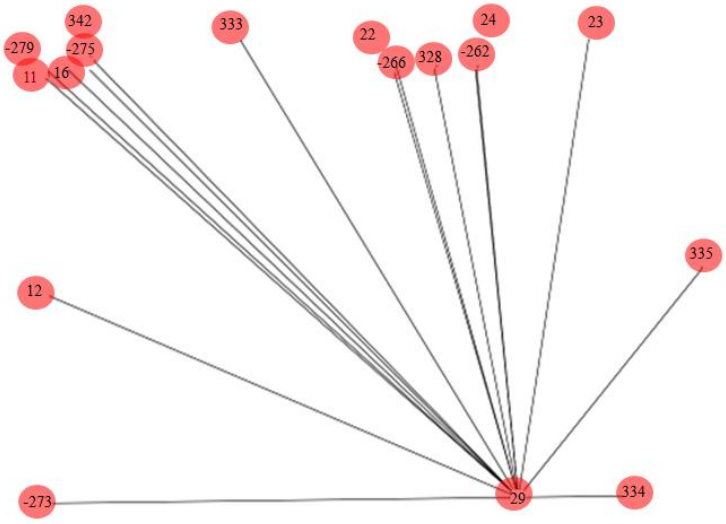


Figure 4.6.2: Visibility relationship between wall 29 and the rest of nodes in a Gt node number 2.

Some walls or diffraction angles can belong to one or more Gt nodes. Gt node numbers which contain this Gs node: a wall or a diffraction angle can be found by the application of a function from Layout class. An example for Gs 29 node is presented in figure 4.6.3.

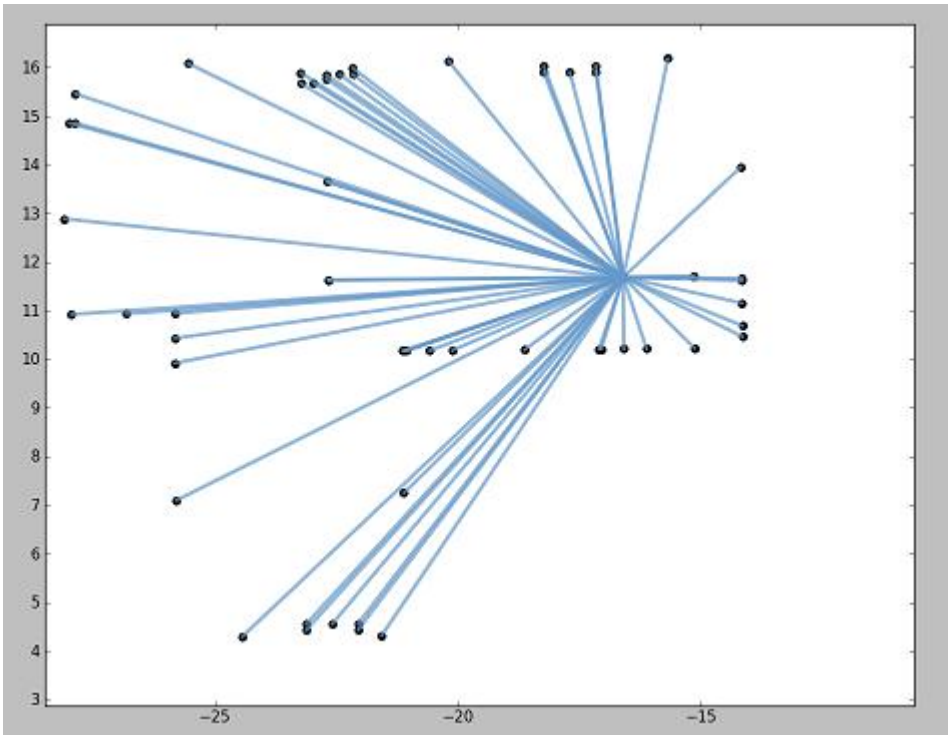


Figure 4.6.3: There are two Gt nodes which contain wall number 29: Gt node 2 and Gt node 18.

Using all visibility relationships presented before, for the entire indoor environment structure, Graph of Visibility is created. For an indoor environment, Gv graph is presented in figure 4.6.4:

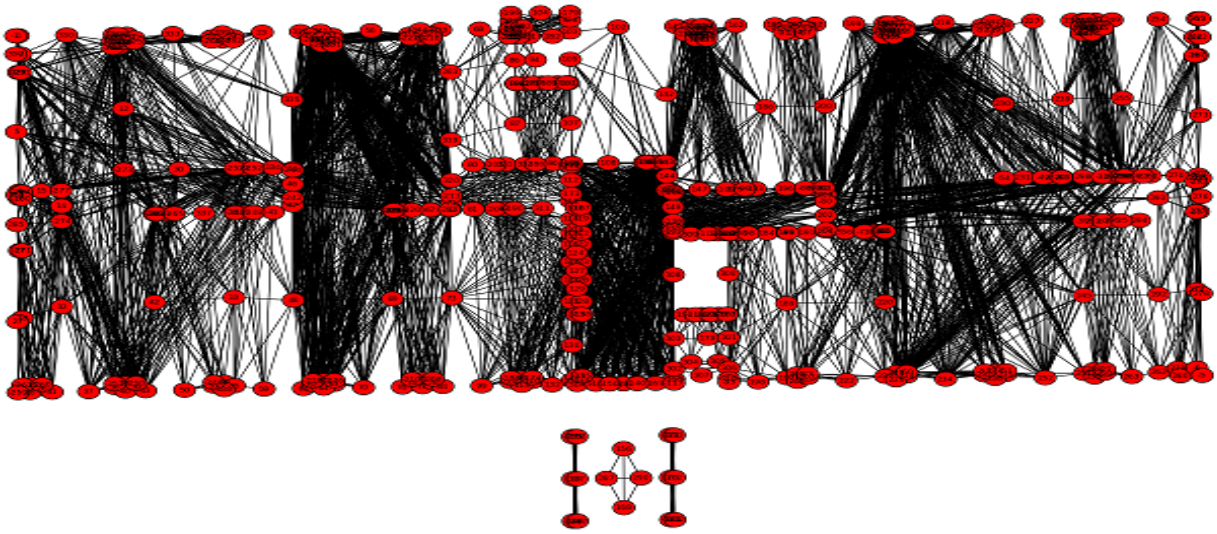


Figure 4.6.4: Gv graph example.

Of course, the creation of this graph requires a lot of time, but once it is created, this kind of

presentation of a building structure makes easier to proceed with future simulations and calculations. Thanks to this graph, we can show connectivity relationship between the transmitter and the receiver also, nodes/edges which are among the transmitter and the receiver.

5 Getting possible rays

5.1 Installation of a transmitter (Tx) and a receiver (Rx)

In indoor environment, in this work, random coordinates are used for the position of a transmitter and of a receiver. Random values are limited by the maximum and minimum values of the coordinates of the indoor construction. This kind of limitation permits the transmitter and the receiver to always be allocated within indoor construction.

5.2 Localization of a transmitter and a receiver in Gs graph structure

To localize a transmitter and a receiver in closed settings some algorithms are developed in this work. Firstly, the Gt node numbers containing the transmitter and the receiver are found. Secondly, their visibility relationships with all Gs nodes in the same Gt nodes are verified. Thirdly, the transmitter and the receiver are connected with Gv graph (see figure 5.2.1).

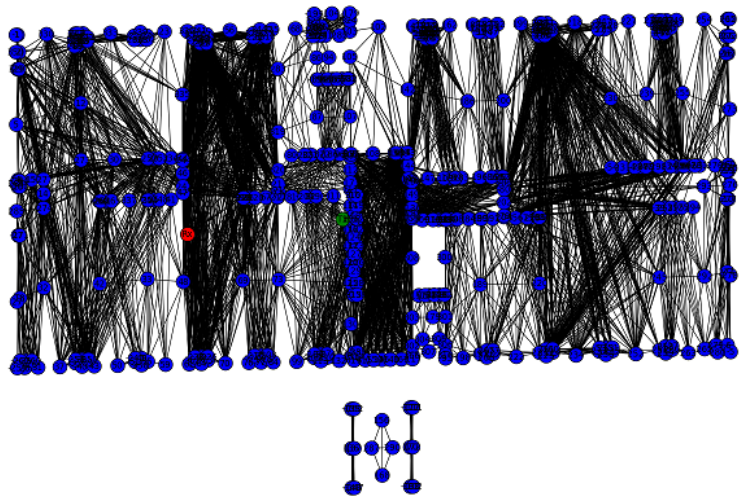


Figure 5.2.1: The transmitter and the receiver connected with Graph of Visibility.

5.3 Finding out signatures by using Graphs method

A Signature contains information on all the interactions that the ray had during the route between the transmitter and the receiver.

Using NetworkX analyses, some “signatures” can be easily found. A part of the signatures are presented in figure 5.3.1 (a):

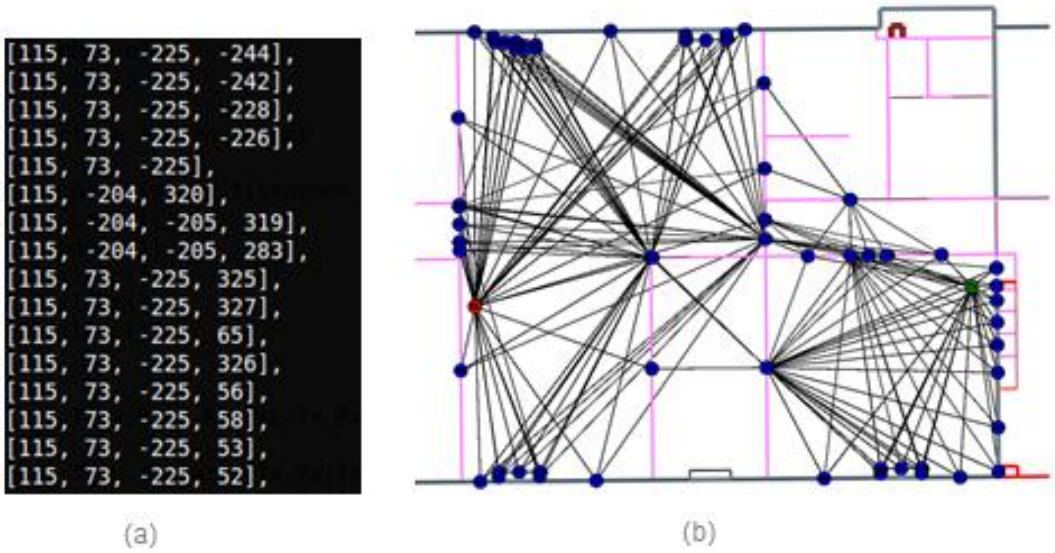


Figure 5.3.1: (a) An example of some “signatures”. (b) An example of some rays that can exist between a real transmitter and a real receiver.

Via these signatures some rays existing between the transmitter and the receiver can be anticipated (figure 5.3.1 (b)).

5.4 Finding out the shortest “signature” by using Gv graph

To find out the shortest “signature”, which can be used in the future as a reference, NetworkX function is used one more time. The result is presented in figure 5.4.1.

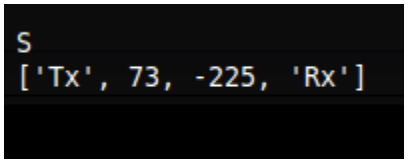


Figure 5.4.1: The shortest “signature”, between the transmitter and the receiver.

Using this ‘‘signature’’ a ray can be built. This ray is the shortest ray which can exist between a transmitter and a receiver.

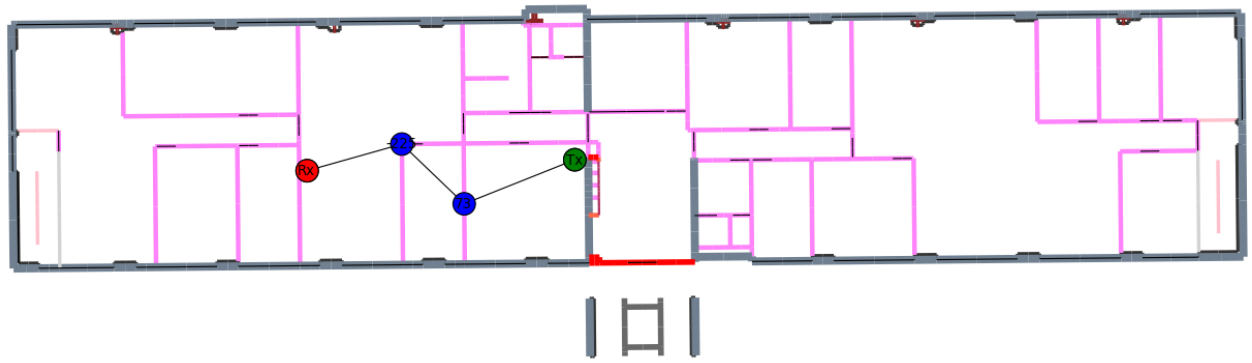


Figure 5.4.2: Anticipated shortest ray in indoor environment.

This method is reliable for different limitations:

- The shortest ray can be found without any distance limitation between a transmitter and a receiver.
- More rays can be obtained only imitated distances between a transmitter and a receiver. Limitation distance is equal to 3 rooms between a transmitter and a receiver

5.5 Finding out the shortest "signature" by using clip of environment

Finding out the shortest "signature" with the help of this model is the easiest way among nowadays other methods.

Firstly, the indoor environment is clipped. The clip of an environment is made by using a transmitter and a receiver coordinates, takes the walls and the diffraction angles of the indoor environment which are among a transmitter and a receiver and eliminates the rest of indoor construction. Secondly, the segments limited by clipped boundary are separated from the other segments. Thirdly, the direct visibility relationship between separated segments is verified.

The elimination of the walls, which are out of the clipped zone, contributes to reduce the time simulation more than it was done by graphs method.

5.6 Finding the shortest "signature"

Using clip method the shortest signature can be easily found. The result of simulation is presented in figure 5.6.1.

```
['Tx', 12, 334, 46, 69, 73, 'Rx']
```

Figure 5.6.1: Presentation of the shortest anticipated signature.

Obtained result would show that if there had been a ray emitted from a transmitter and received by a receiver, it could have some interaction with the walls numbers 12, 334, 46, 69 and 73. The numbers of these walls are classed by interaction order.

Using the shortest signature obtained by clip method, the shortest ray between a transmitter and a receiver can be created. The result of the created ray is shown in figure 5.6.2.

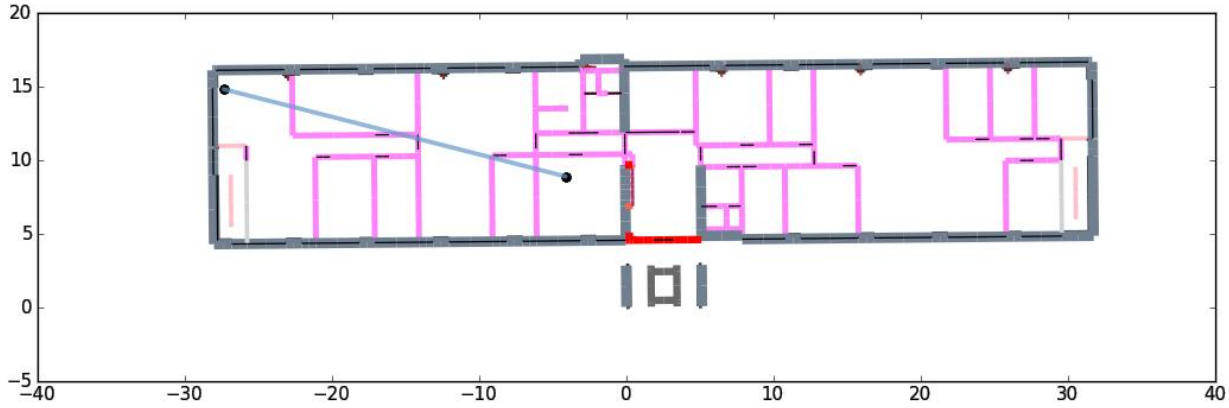


Figure 5.6.2: The shortest ray obtained by clip method.

5.7 Finding out more signatures

More rays can by created based on the shortest ray founded by clip method, and some signatures corresponding to these rays can be obtained. These creation of rays these is founded on GO and use coordinates only from clipped party of the environment. Created rays are presented in figure 5.7.1 and its corresponding signatures in figure 5.7.2.

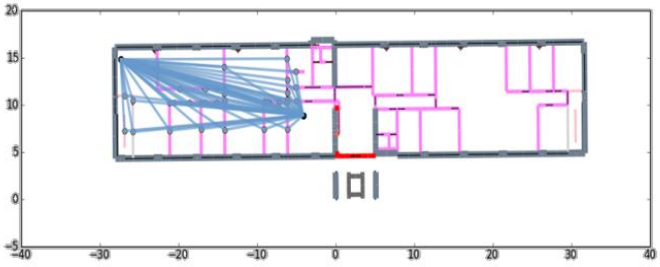


Figure 5.7.1: Presentation of the created rays in clipped zone.

```
[0, 21, 33, 48, 0],
[0, 12, 328, 24, 23, 23, 335, 0],
[0, 12, 30, 337, 33, 48, 0],
[0, 15, 19, 31, 32, 42, 33, 48, 0],
[0, 15, 32, 32, 42, 33, 48, 0],
[0, 42, 33, 48, 0]]
```

Figure 5.7.2: Presentation of Signatures which are corresponding to the clipped zone created rays.

If time of obtaining more rays is not limitated, method to obtain more rays in clipped zone can be applied for all indoor environment. By this way, there is a probability that created rays, which are based on the shortest ray, correspond to PyRay simulation result. There is illustration of these rays for all indoor environment in figure 5.7.3 and its corresponding signatures are presented in figure 5.7.4.

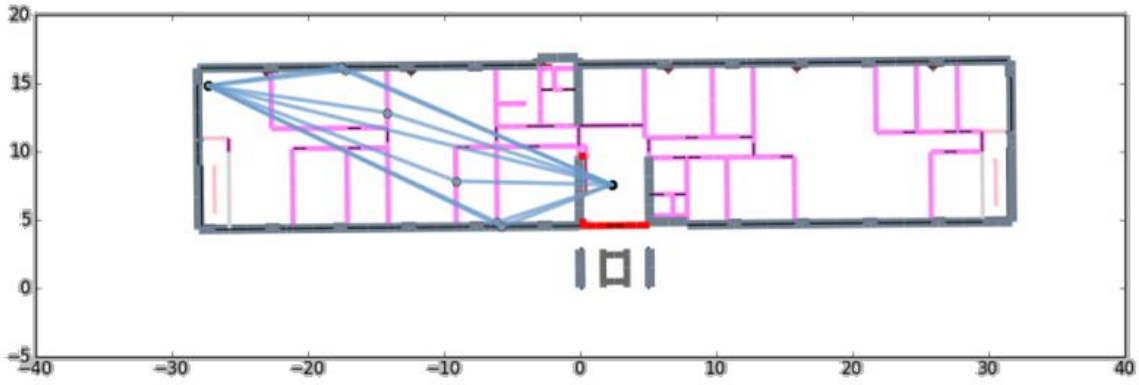


Figure 5.7.3: Presentation of created rays for complete indoor environment.

```
[0, 12, 30, 337, 33, 48, 0],
[0, 12, 30, 337, 33, 48, 69, 0],
[0, 12, 30, 337, 33, 48, 69, 73, 69, 0],
[0, 12, 335, 283, 87, 69, 67, 320, 80, 0],
[0, 12, 30, 337, 33, 48, 69, 73, 99, 69, 73, 0],
[0, 12, 335, 283, 87, 107, 69, 67, 71, 80, 87, 0],
[0, 12, 30, 337, 33, 48, 69, 73, 99, 136, 69, 75, 78, 84, 0],
[0, 12, 30, 337, 33, 48, 69, 73, 99, 136, 137, 69, 75, 78, 84, 136, 0],
[0, 12, 335, 283, 86, 94, 105, 142, 69, 73, 81, 96, 107, 0],
```

Figure 5.7.4: Presentation of Signatures which are corresponding to the created rays for indoor environment.

6 PyRay Simulation resultants and Validation of models

The PyRay Simulator Platform is used to obtain the propagation channel simulation result. To verify and finally validate the obtained result of the two developed models, the same coordinates of a transmitter and a received are placed in an indoor environment.

The result of PyRay simulation is presented in figure 6.1 and some signatures of obtained rays are presented in figure 6.2.

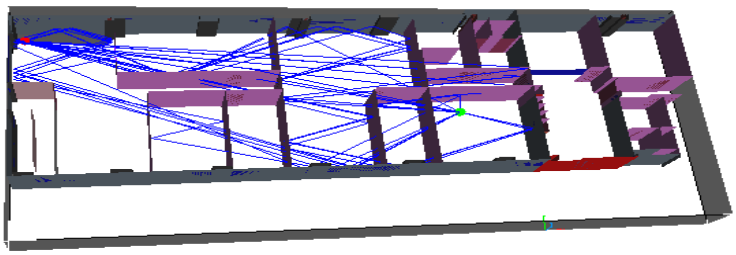


Figure 6.1: A result of PyRay simulation.

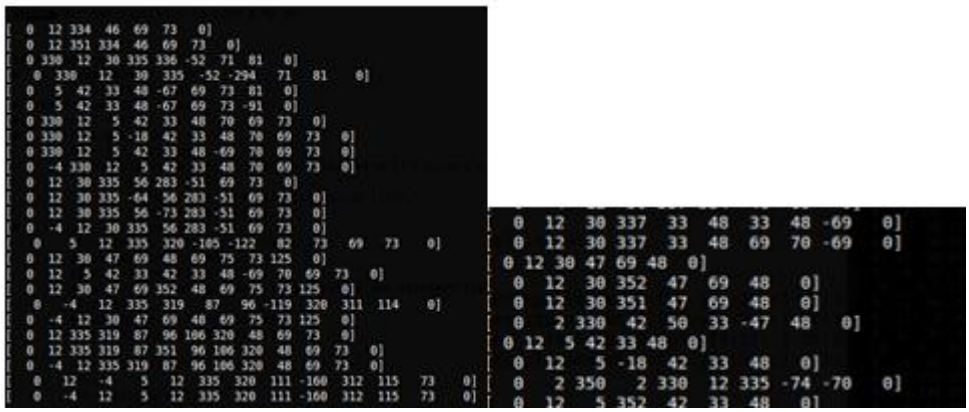


Figure 6.2: Signatures obtained by PyRay simulator.

In figure 6.2 the signatures are arranged by power intersection: the first line corresponds to the first arrived ray, which contains the highest energy compare to the other rays. The last arrived ray contains the least energy.

The results obtained by using Graphs method are not comparable with the result of PyRay simulations because there is a need for further development.

The comparison of the obtained signature from figure 5.6.1 with the first signature from figure 6.2 shows that the two ones are totally similar. It means that the shortest ray founded by the clip method corresponds to the first arrived ray in PyRay simulation. It means that in case of real transmitter and receiver in real indoor environment the same result could be achieved. It is known

that the first arrived ray is the important ray for localization applications. Thus the primary objective of this work is realized. The comparison of the signatures, corresponding to the clipped zone created rays (figure 5.7.2) with the rest of the signatures obtained by a PyRay simulation (figure 6.2), shows that looking for more signatures in clipped zone it is not useful. On the other hand, the evaluation of the signatures corresponding to the created rays for indoor environment (figure 5.7.4) with PyRay simulation result (figure 6.2) illustrates some relationship between several signatures. If the method to find more signatures in complete indoor environment is developed further, it can be attractive approach for other indoor applications: for example radio covert.

Conclusion

This project has enabled us to discover specific problems related to localization issue in indoor environment. This problem strongly depends on environment characteristics and thus necessitates the development of realistic radio propagation channel modeling simulation tools.

In this context, the project has aimed at bringing solutions for reducing calculation time in indoor localization while keeping accurate estimation. New models are proposed and fast tools for different localization applications in indoor environment are implemented. These tools use NetworkX and Shapely packages and environment geometry information from the 3D-RT based propagation channel simulator PyRay.

The obtained results have been evaluated through the comparison with the results of propagation channel simulator PyRay. It has been shown that the proposed models provide reliable results and enable us to save time in localization. More, this research topic is very complex and lots of further investigations have still to be led in order to improve indoor localization system performances for stationary and mobile transmitter and/or receiver.

If the method to find more signatures in complete indoor environment is developed further, it can be attractive approach for other indoor applications: for example radio covert.

Bibliography

[1] Roxana-Elena BURGHELEA
‘‘Contribution à la Simulation Déterministe de Canaux Radios Hétérogènes ULB : Problématique de la Prise en Compte Réaliste de l'Antenne’’ Thèse Universite de Rennes 1, 2010.

[2] Marios Raspopoulos, Stavros Stavrou, Bernard Uguen, Roxana Burghelea, Mariano García, Troels Pedersen, Gerhard Steinböck, Bernard H. Fleury, Benoît Denis, Joe Youssef, Yves Lostanlen, Álvaro Álvarez, ”Modelling of the Channel and its variability”, ICT- 217033 WHERE, Dec.2008.

[3] Roxana Burghelea, Stephane Avrillon, Bernard Uguen, “UWB antenna compact modeling using vector spherical harmonic theory”, Institut d’Electronique et Telecommunications de Rennes.

[4] Pereira C., Chartois Y., Pousset Y., and Vauzelle R., “Inuence of the level of description of the indoor environment on the characteristic parameters of a mimo channel,” Comptes Rendus Physique, Elsevier, p. 715725, 2006.

[5] Aaron D. Ames, Robert D. Gregg, Eric D.B. Wendel and Shankar Sastry, ‘‘Towards the Geometric Reduction of Controlled three-dimensional Bipedal robotic Walkers,’’ University of California, Berkeley Berkeley, CA 94720 2006.

[6] L. De Nardis, D. Domenicali, M.-G. Di Benedetto, “ Mobility model for Body Area Networks of soccer players,” INFO-COM Department, Sapienza University of Rome Via Eudossiana 18, 00184, Rome, Italy, September 2010.

[7] Wei-jen Hsu, Thrasyvoulos Spyropoulos, Konstantinos Psounis and Ahmed Helmy, Sophia-Antipolis, “Modeling Time-variant User Mobility in Wireless Mobile Networks,”IEEE INFO-COM 2007

[8] http://networkx.lanl.gov/contents.html, ‘‘NetworkX documentation’’

[9] http://pypi.python.org/pypi/Shapely, ‘‘The Shapely User Manual’’

Annex

```
# -*- coding:Utf-8 -*-
import os
import pickle
import numpy as np
import scipy as sp
import random
import matplotlib.pyplot as plt
from matplotlib import pyplot as plt
import matplotlib.colors as clr
import networkx as nx
import shapely.geometry as sh

from   shapely.ops import cascaded_union
from   descartes.patch import PolygonPatch
import Image
from   Graph import *
from   PyUtil import *
from   GeomUtil import *
from operator import *
import random
from scipy import io
from PyUtil import *

COLOR = {
    True:  '#6699cc',
    False: '#ff3333'
    }
def v_color(ob):
    return COLOR[ob.is_valid]



class Layout(object):

        """

            Layout exploits networkx to store Layout information
            Gt : Topological graph  (indicates topological
relationships between rooms)
            Gs : Structure graph
            Gc : Connection graph (indicates visbility
relationships)

            Nnode  : Number of nodes of Gs
            Nedge  : Number of edges of Gs
```

```
            pt     :
           tahe    :
           labels :

           To initialize the structure :

                L = Layout(G1)

                L.find_edgelist(edgelist,nodelist)
                L.diag(p1,p2,l,al1,al2,quadsel=0)
                L.checkvis(p,edgelist,nodelist)
                L.visilist(p)
TBD
                L.closest_edge(p,AAS)
                L.visi_papb(pa,pb,edgelist=np.array([]))
                L.showGs()                 : show structure Graph
Gs
                L.showGc()                 : show connectivity
Graph Gc (TBD)
                L.showGt()                 : show topology Graph
Gt (TBD)
                L.savestr2                 : export Layout as an
.str2 file
                L.add_fnod(p)              : add free node  p
                L.add_none(ns,alpha=0.5)   : add node on edge
                L.add_edge(n1,n2)          : add edge between n1
and n2
                L.del_node(n1)             : delete node n1
                L.del_edge(e1)             : delete edge e1
                L.edit_edge(e1)            : edit edge e1
                dico = L.subseg()          : get the dictionnary
of subseg
                L.add_fnod(p)              : add free node p
                L.add_none(ns,alpha=0.5)   : add node on edge ns
                L.add_edge(n1,n2)          : add edge between n1
and n2
                L.del_node(n1)             : delete node n1
                L.del_edge(e1)             : delete edge e1
                L.edit_edge(e1)            : edit edge e1
                 L.geomfile()               :
                L.show3()                  :

                Getting nodelist and edgelist from a zone
```

```
                nl,el = L.zone(xmin=0,ymin=0,xmax=10,ymax=10)

        L.wall_delta(x1, y1, x2, y2, delta=0.0001)
        L.Polygon_tag(nnode): return a Polygon nnode number of
node of Gt graph
        L.visi_wall_wall(nnode)   :find out connectivity between
walls
        L.diffraction_num(nnode): find the numbers and
        coordinates of diffraction corners in the Gt node
        L.visi_diff_wall(nnode): find out connectivity between
walls and diffraction angels
        L.visi_wall_diff(nnode): find out connectivity between
diffraction and walls angels
        L.visi_diff_diff(nnode): find out connectivity between
diffraction angels
        L.node_visi_direct(nnode,nn):return a dictionary with
all numbers of walls
        L.visi_direct(): return a list of all direct visibility
in the indoor environment
        L.coords_visi_cycle(nn): return the  numbers nodes of
        Gt which include nn also return the coordinates of
        nodes whit whom nn in direct  visibility
        L.visi_nodes(nnode): return a Graph of nodes and number
        of nodes which are engaged in visibility list of a Gt
        node, also show them
        L.node_visi_direct_Graph(nnode,nn): return a Graph of
        visibility: visibility between a node and other nodes
        in the same Gt node
        L.visi_all_nodes(): return a dictionary with all nodes
        of Gs which are engaged in visibility
        L.visi_all_indoor(): return graph Gv: Graph of
visibility
        L.visi_all_indoor(): return graph Gv: Graph of
visibility
        L.visi_Tx_Rx(nnode,Tx_x,Tx_y):return the numbers of
        walls and diffraction angles which are in visibility
        with Tx/Rx
        L.contains_Tx_Rx(x,y):return the number og Gt node
which contains Tx or Rx
        L.connect_Tx(Tx_x,Tx_y,node_Tx): return a graph which
        connect Tx with nodes and diffraction angels of node of
        Gt which are in visibility with Tx
        L.connect_Rx(Rx_x,Rx_y,node_Rx):return a graph which
        connect Rx     with nodes and diffraction angels of node
```

```
            of Gt which are in visibility with Rx
            L.connect_Tx_Rx(,Tx_x,Tx_y,node_Tx,Rx_x,Rx_y,node_Rx)::
            return a graph which connect Tx et Rx with nodes and
            diffraction angels of node of Gt
            L.cords_Tx_Rx(): definite the boundary of Gs graph and
            return Gs_XMAX, Gs_xmin, Gs_YMAX, Gs_ymin with +,-3,4
            for position of Tx and Rx
            L.boundary_Gs(): returns  xmax,xmin,ymax,ymin of Gs
construction
            L.Tx_Rx_pos(): give random positions to Tx and Rx in
            the Gs graph and return Tx_x,Tx_y,Rx_x,Rx_y
            L.Gv_Tx_Rx(): crate a Graph Gv_Tx_Rx,  which connect
            the graph visibility Gv with Tx and Rx
            L.one_ray(S,Tx_x,Tx_y,Rx_x,Rx_y):build a graph og a ray
by using a signature
            L.all_rays(signature): build a graph of all possible
            rays between Tx and Rx by using signatures
            L.shortest_ray(S,Tx_x,Tx_y,Rx_x,Rx_y):build a graph of
            the shortest ray by using a signature
            L.dir_ray_sign(Tx_x,Tx_y,Rx_x,Rx_y): takes transmitter
            and receiver coordinates and returns the shortest ray
            and his signature
            L.dir_ray_sign(Tx_x,Tx_y,Rx_x,Rx_y): takes transmitter
            and receiver coordinates and returns the shortest ray
            and his signature
            L.clip_all_sign(Txs,Rxs): takes transmitters and
            receivers coordinates and returns all shortest
            signatures which can exist between every transmitter
            and every receiver

        """

        def __init__( self,_filename=''):
            self._filename = _filename
                      self.Gs=nx.Graph()
            self.Gv=nx.Graph()
            self.labels={}
            self.pos={}
            self.Gt=nx.Graph()
            self.pos_v={}
            self.labels_v ={}
            self.d={}
            self.visi_dir=[]
            self.Gv_new=nx.Graph()
            self.pos_new={}
            self.labels_new={}
```

```
                self.Gc=nx.Graph()
                self.Gm=nx.Graph()
                self.labels ={}
        self.name   ={'PARTITION':[]}
        self.Gs.pos={}
                self.sl  = SlabDB()
                self.Nn  = 0
                self.Ne  = 0
                self.Nss = 0
        self.filename = 'Lstruc'
                self.display={}
                self.display['title']=''
                self.display['nodes']=False
                self.display['ndsize']=10
                self.display['ndlabel']=False
                self.display['ndlblsize']=10
                self.display['edlblsize']=10
                self.display['fontsize']=10
                self.display['edlabel']=False
                self.display['edges']=True
                self.display['ednodes']=False
                self.display['subseg']=True
                self.display['visu']=False
                self.display['thin']=False
                self.display['scaled']=True
                self.display['alpha']=0.5
                self.display['layer']=[]
                self.display['clear']=False
                self.display['activelayer']=[]
                self.display['overlay']=False
                self.display['fileoverlay']="TA-Office.png"
                self.display['box']=(-11.4,19.525,-8.58,23.41)
        self.display['subLayer']=['3D_WINDOW_GLASS','WINDOW_GLA
    SS','DOOR','WOOD']
                self.ax   = (-40,40,-5,20)
                 self.zoom = (-40,40,-5,20)

    def check(self):
                """
                Check Layout consistency
                """
                for e in self.Gs.node.keys():
                        if e > 0 :
                                n1,n2 =
array(self.Gs.neighbors(e))
                                p1    = array(self.Gs.pos[n1])
                                p2    = array(self.Gs.pos[n2])
                                for n in self.Gs.node.keys():
                                        if (n < 0) & (n1!=n) &
(n2!=n):
                                                p =
```

```
array(self.Gs.pos[n])
                                                    if
isBetween(p1,p2,p):
                                                          print
"Warning segment ",e,"contains point ",n

     def clip(self,xmin,xmax,ymin,ymax):

          """
               return the list of edges which cross or belong to
the clipping zone

               Algorith :
                    1) Determine all segment outside the clipping
zone
                     4 condition to test
                    2) Union of the 4 conditions
                    3) setdiff1d between the whole array of
segments and the segments outside

          """
          p0   = self.pt[:,self.tahe[0,:]]
          p1   = self.pt[:,self.tahe[1,:]]

          maxx = maximum(p0[0,:],p1[0,:])
          maxy = maximum(p0[1,:],p1[1,:])
          minx = minimum(p0[0,:],p1[0,:])
          miny = minimum(p0[1,:],p1[1,:])

          nxp = np.nonzero(maxx<xmin)[0]
          nxm = np.nonzero(minx>xmax)[0]
          nyp = np.nonzero(maxy<ymin)[0]
          nym = np.nonzero(miny>ymax)[0]

          iseg = arange(self.Ne)

          u1   = union1d(nyp,nym)
          u2   = union1d(nxp,nxm)
          u   = union1d(u1,u2)

          return setdiff1d(iseg,u)



        def boundary(self):

     """
          boundary()
               provides the boundary box of the structure
               return : ( xmin,xmax,ymin,ymax )
     """
```

```
                self.ax=(-10,10,-10,18)

        def help(self):
                print "L.showGs(clear=True)"
                print
"L.showGs(edlist=L.subseg()['WOOD'],dthin=False,dlabels=True)"


        def loadGr(self,G1):
                # Tag nodes
                self.labels ={}
                self.name   ={}
                # Position of the nodes
                self.Gs.pos={}
                # Dictionary of the slab
                self.sl    = G1.sl
                self.ce    = G1.ce
                self.pt    = G1.pt
                self.tahe  = G1.tahe
                self.Nn    = G1.nn
                self.Ne    = G1.en
                self.Nss   = G1.cen
                self.ax   = (-40,40,-5,20)
                self.zoom = (-40,40,-5,20)

                # Node labeling (structure of the nodes)

                for k in range(self.Nn):
                    self.Gs.add_node(-(k+1))
                    self.Gs.pos[-(k+1)]=(G1.pt[0,k],G1.pt[1,k])
                 self.labels[-(k+1)] = str(-(k+1))


                # Node labeling (structure of the edges)

                for k in range(self.Ne):
                    self.Gs.add_node(k+1,name=G1.name[k])
                        self.Gs.add_node(k+1,zmin=G1.z[0,k])
                        self.Gs.add_node(k+1,zmax=G1.z[1,k])
                    nta  = G1.tahe[0,k]
                    nhe  = G1.tahe[1,k]

     self.Gs.pos[k+1]=((G1.pt[0,nta]+G1.pt[0,nhe])/2.,(G1.pt[1,nt
a]+G1.pt[1,nhe])/2.)
                    self.Gs.add_edge(-(nta+1),k+1)
                    self.Gs.add_edge(k+1,-(nhe+1))
                    self.labels[k+1] = str(k+1)
                        if self.name.has_key(G1.name[k]):
                         self.name[G1.name[k]].append(k+1)
                        else:
                                    self.name[G1.name[k]]= [k+1]
```

```
                # Update sub-segment

                for k in self.ce:

self.Gs.add_node(k+1,ss_name=self.sl.di[self.ce[k][0]])

self.Gs.add_node(k+1,ss_ce1=self.ce[k][1])

self.Gs.add_node(k+1,ss_ce2=self.ce[k][2])

self.Gs.add_node(k+1,ss_zmin=self.ce[k][3])

self.Gs.add_node(k+1,ss_zmax=self.ce[k][4])



                # Create connectivity graph Gc
                #   update Gc with nd_nd ed_ed

                self.Gc = nx.Graph()
                self.Gc.add_nodes_from(self.Gs.nodes())
                pos   = self.Gs.pos
                ndnd  = G1.nd_nd
                nded  = G1.nd_ed
                eded  = G1.ed_ed


                Nn    = shape(ndnd)[0]
                for k in range(Nn):
                        nnp = -(k+1)
                        kvu = sp.nonzero(ndnd[k]==3)
                        nc = -kvu[0]-1
                        for l in nc:
                                self.Gc.add_edge(nnp,l)

                Ne    = shape(eded)[0]
                for k in range(Ne):
                        ne = k+1
                        kvu = sp.nonzero(eded[k]!=0)
                        nc = kvu[0]+1
                        for l in nc:
                                self.Gc.add_edge(ne,l)
                self.Gc.pos = pos
                self.display['activelayer']=self.name.keys()

        def subseg(self):

        """
                    Subseg
          dico = L.subseg()
```

```
                name <->  edgelist
            """
              dico = {}
              for k in self.Gs.node.keys():
                      dk = self.Gs.node[k]
                      if dk.has_key('ss_name'):
                              name = dk['ss_name']
                              if dico.has_key(name):
                                      dico[name].append(k)
                              else:
                                      dico[name]=[k]
              return (dico)

          ############## GRAPH PART ##############


    def wall_delta(self,x1,y1,x2,y2,delta=0.0001):

        """
            L.wall_delta(x,y)  : we define a tolerance
length:delta=0.0001,
            distinguish the points corresponding to the
distance of the tolerance
            length starting from the extremities of the walls
and return receved ponts.
            x1,y1,x2,y2 =
L.wall_delta(x1,y1,x2,y2,delta=0.0001)
        """
        ax=x2-x1
        ay=y2-y1
        a_mod=sqrt(ax**2+ay**2)
        a_ch_x=ax/a_mod
        a_ch_y=ay/a_mod

        bx=x1+ax*delta/a_mod
        by=y1+ay*delta/a_mod
        cx=x2-ax*delta/a_mod
        cy=y2-ay*delta/a_mod

        return (bx,by,cx,cy)

    def wall_delta_signs(self,x1,y1,x2,y2,delta):

        """
            L.wall_delta(x,y)  : we define a tolerance
length:delta=0.0001,
```

```
                distinguish the points corresponding to the
distance of the tolerance
                length starting from the extremities of the walls
and return receved ponts.
                x1,y1,x2,y2 =
L.wall_delta(x1,y1,x2,y2,delta=0.0001)
            """
            ax=x2-x1
            ay=y2-y1
            a_mod=sqrt(ax**2+ay**2)
            a_ch_x=ax/a_mod
            a_ch_y=ay/a_mod

            bx=x1+ax*delta/a_mod
            by=y1+ay*delta/a_mod
            cx=x2-ax*delta/a_mod
            cy=y2-ay*delta/a_mod

            return (bx,by,cx,cy)

        def Polygon_tag(self,nnode):

            """
                Polygon(nnode): return a Polygon
            nnode number of node of Gt graph
            """
            coords=[]

            for i in range(len(self.Gt.node[nnode]['vnodes'])):

        coords.append(self.Gs.pos[self.Gt.node[nnode]['vnodes'][i]])

            polygon2 = sh.Polygon(tuple(coords))

            return(polygon2)


        def visi_wall_wall(self,nnode):

            """
                L.visi_wall_wall(nnode) :find out conectivites
between walls
                takes a number of a node of graph Gt and return
                walls numbers placed in a matrix (2*27),    first
```

```
                matrix corresponds to total visibility second
                matrix corresponds to partial visibility

            _______ Used  shapely functions _________

            contains(): Returns True if the object's interior
contains the boundary
                        and interior of the other object and their
boundaries do not touch at all.
            intersects(): Returns True if the boundary and interior
of the object
                        intersect in any way with those of the other.
            disjoint(): Returns True if the boundary and interior
of the object
                        do not intersect at all with those of the
other.

            """
            polygon2 = self.Polygon_tag(nnode)

            w_w_a = [] # first wall of visility total
            w_w_b = [] # second wall of visility total
            w_w_c = [] # first wall of partial visility
            w_w_d = [] # second wall of partial visility


            N = self.Gt.node[nnode]['vnodes'][
self.Gt.node[nnode]['vnodes']>0] # Numbers of segments in Gt node
            for i in range(len(N)):
                 for j in range(len(N)):
                      if (N[i])!= (N[j]):


                           n1_x_i,n1_y_i =
self.Gs.pos[self.Gs.neighbors(N[i])[0]]
                           n2_x_i,n2_y_i =
self.Gs.pos[self.Gs.neighbors(N[i])[1]]
                           n1_x_j,n1_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[0]]
                           n2_x_j,n2_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[1]]

                           M1x = self.Gs.pos[N[i]][0]
                           M1y = self.Gs.pos[N[i]][1]
```

```
                        M2x = self.Gs.pos[N[j]][0]
                        M2y = self.Gs.pos[N[j]][1]

                        if (((n1_x_i!=n2_x_j) and
(n1_y_i!=n2_y_j)) or ((n2_x_i!=n2_x_j) and (n2_y_i!=n2_y_j)) or
((n2_x_i!=n1_x_j) and (n2_y_i!=n1_y_j)) or ((n1_x_i!=n1_x_j) and
(n1_y_i!=n1_y_j))):

                            visi_1 = sh.LineString([(n1_x_i,
n1_y_i), (n1_x_j, n1_y_j)])
                            visi_2 = sh.LineString([(n1_x_i,
n1_y_i), (n2_x_j, n2_y_j)])
                            visi_3 = sh.LineString([(n2_x_i,
n2_y_i), (n1_x_j, n1_y_j)])
                            visi_4 = sh.LineString([(n2_x_i,
n2_y_i), (n2_x_j, n2_y_j)])


                            if (polygon2.contains(visi_1)==True)
or (polygon2.contains(visi_2)==True) or
(polygon2.contains(visi_3)==True)
or(polygon2.contains(visi_4)==True):

                                n1xi,n1yi,n2xi,n2yi=
self.wall_delta(n1_x_i,n1_y_i,n2_x_i,n2_y_i)
                                n1xj,n1yj,n2xj,n2yj=
self.wall_delta(n1_x_j,n1_y_j,n2_x_j,n2_y_j)
                                #visibility between center -
center of wall and center - extremes of wall
                                visi_5 =
sh.LineString([(M1x,M1y), (M2x, M2y)])
                                visi_6 =
sh.LineString([(M1x,M1y), (n1_x_j, n1_y_j)])
                                visi_7 =
sh.LineString([(M1x,M1y), (n2_x_j, n2_y_j)])
                                visi_8 = sh.LineString([(M2x,
M2y),(n1_x_i,n1_y_i)])
                                visi_9 = sh.LineString([(M2x,
M2y),(n2_x_i,n2_y_i)])


                                vi_del_1 =
sh.LineString([(n1xi, n1yi), (n1xj, n1yj)])
                                vi_del_2 =
```

```
sh.LineString([(n1xi, n1yi), (n2xj, n2yj)])
                                    vi_del_3 =
sh.LineString([(n2xi, n2yi), (n1xj, n1yj)])
                                    vi_del_4 =
sh.LineString([(n2xi, n2yi), (n2xj, n2yj)])
                                    if
(polygon2.contains(vi_del_1)==True) or
(polygon2.contains(vi_del_2)==True) or
(polygon2.contains(vi_del_3)==True) or
(polygon2.contains(vi_del_4)==True) or
(polygon2.contains(visi_5)==True) or
(polygon2.contains(visi_6)==True) or
(polygon2.contains(visi_7)==True) or
(polygon2.contains(visi_8)==True) or
(polygon2.contains(visi_9)==True):

                                         w_w_a.append(N[i])
                                         w_w_b.append(N[j])
                                    else:
                                         pass

                               else:
                                         pass


          tvis_w_w=vstack([w_w_a,w_w_b])
          return (w_w_a,w_w_b) #retourn walls numbers of  partial
visibility


     def diffraction_num(self,nnode):

          """
               L.diff_num(nnode): find the numbers and
               coordinatesof diffraction corners in the Gt node

          """
          ncoin,ndiff= self.buildGc()
          coor=[]
          diff_num=[]
          for i in range(len(self.Gt.node[nnode]['vnodes'])):
                    if
ndiff.__contains__(self.Gt.node[nnode]['vnodes'][i]):
```

```
            coor.append(self.Gs.pos[self.Gt.node[nnode]['vnodes'][i]])

            diff_num.append(self.Gt.node[nnode]['vnodes'][i])
                return (diff_num,coor)


        def visi_diff_wall(self,nnode):

                """
                        L.visi_diff_wall(nnode): find out conectivites
bitween walls and diffraction angels
                        takes a number of a node of graph Gt and return
                        walls numbers placed in a matrix (2*27),    first
                        matrix corresponds to total visibility second
                        matrix corresponds to partial visibility

                        ____ some shapely functions are used _____

                contains(): Returns True if the object's interior
contains the boundary
                                and interior of the other object and their
boundaries do not touch at all.
                intersects(): Returns True if the boundary and interior
of the object
                                intersect in any way with those of the other.
                disjoint(): Returns True if the boundary and interior
of the object
                                do not intersect at all with those of the
other.

                """

                diff_num = self.diffraction_num(nnode)[0]
                polygon2 = self.Polygon_tag(nnode)
                d_w_a = [] # first diffraction angle of visility total
                d_w_b = [] # second wall of visility total
                d_w_c = [] # first diffraction angle of partial
visility
                d_w_d = [] # second wall of partial visility
                N = self.Gt.node[nnode]['vnodes'][
self.Gt.node[nnode]['vnodes']>0] # Numbers of segments in Gt node
                for i in range(len(diff_num)):
                                for j in range(len(N)):
```

```
                        diff_x_i = self.Gs.pos[diff_num[i]][0]
                        diff_y_i = self.Gs.pos[diff_num[i]][1]
                        n1_x_j,n1_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[0]]
                        n2_x_j,n2_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[1]]

                        M2x = self.Gs.pos[N[j]][0]
                        M2y = self.Gs.pos[N[j]][1]
                        if ((diff_x_i!=n1_x_j) and
(diff_x_i!=n2_x_j)) and ((diff_y_i!=n1_y_j) and
(diff_y_i!=n2_y_j)):
                            coo_pili=[(diff_x_i, diff_y_i),
(n1_x_j,n1_y_j),(n2_x_j, n2_y_j)]

                            visi_1 = sh.LineString([(diff_x_i,
diff_y_i), (n1_x_j,n1_y_j)])
                            visi_2 = sh.LineString([(diff_x_i,
diff_y_i), (n2_x_j,n2_y_j)])

                            n1xj,n1yj,n2xj,n2yj=
self.wall_delta(n1_x_j,n1_y_j,n2_x_j,n2_y_j)
                            visi_3 = sh.LineString([(diff_x_i,
diff_y_i), (n1xj, n1yj)])
                            visi_4 = sh.LineString([(diff_x_i,
diff_y_i), (n2xj, n2yj)])

                            visi_5 = sh.LineString([(diff_x_i,
diff_y_i), (M2x, M2y)])



                            if (polygon2.contains(visi_1)==True)
or (polygon2.contains(visi_2)==True)or
(polygon2.contains(visi_3)==True)or
(polygon2.contains(visi_4)==True)or
(polygon2.contains(visi_5)==True):
                                d_w_a.append(diff_num[i])
                                d_w_b.append(N[j])

                            else:
                                pass

        return (d_w_a,d_w_b) # retourn diff angel et num walls
```

```
of  partial visibility

     def visi_wall_diff(self,nnode):

          """
               L.visi_wall_diff(nnode): find out conectivites
bitween walls and diffraction angels
               takes a number of a node of graph Gt and return
               walls numbers placed in a matrix (2*27),     first
               matrix corresponds to total visibility second
               matrix corresponds to partial visibility

          ____ some shapely functions are used _____

          contains(): Returns True if the object's interior
contains the boundary
                    and interior of the other object and their
boundaries do not touch at all.
          intersects(): Returns True if the boundary and interior
of the object
                    intersect in any way with those of the other.
          disjoint(): Returns True if the boundary and interior
of the object
                    do not intersect at all with those of the
other.
          """

          diff_num = self.diffraction_num(nnode)[0]
          polygon2 = self.Polygon_tag(nnode)
          w_d_a = [] # wall number of visility total
          w_d_b = [] # diffraction angle numbers of total
visility
          w_d_c = [] #  wall number of partial visility
          w_d_d = [] # diffraction angle numbers of partial
visility
          N = self.Gt.node[nnode]['vnodes'][
self.Gt.node[nnode]['vnodes']>0] # Numbers of segments in Gt node
          for j in range(len(N)):
                    for i in range(len(diff_num)):
                         diff_x_i,diff_y_i =
self.Gs.pos[diff_num[i]]

                         n1_x_j,n1_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[0]]
```

```
                        n2_x_j,n2_y_j =
self.Gs.pos[self.Gs.neighbors(N[j])[1]]


                        M2x = self.Gs.pos[N[j]][0]
                        M2y = self.Gs.pos[N[j]][1]

                        if ((n1_x_j!=diff_x_i) and
(n2_x_j!=diff_x_i)) and ((n1_y_j!=diff_y_i) and
(n2_y_j!=diff_y_i)):
                            coo_pili=[(n1_x_j,n1_y_j),(n2_x_j,
n2_y_j),(diff_x_i, diff_y_i)]
                            visi_pol = sh.Polygon(coo_pili)

                            visi_1 = sh.LineString([(diff_x_i,
diff_y_i), (n1_x_j,n1_y_j)])
                            visi_2 = sh.LineString([(diff_x_i,
diff_y_i), (n2_x_j,n2_y_j)])

                            n1xj,n1yj,n2xj,n2yj=
self.wall_delta(n1_x_j,n1_y_j,n2_x_j,n2_y_j)
                            visi_3 = sh.LineString([(diff_x_i,
diff_y_i), (n1xj, n1yj)])
                            visi_4 = sh.LineString([(diff_x_i,
diff_y_i), (n2xj, n2yj)])

                            visi_5 = sh.LineString([(diff_x_i,
diff_y_i), (M2x, M2y)])

                            if
(polygon2.contains(visi_pol)==True) or
(polygon2.contains(visi_1)==True)
or(polygon2.contains(visi_2)==True)
or(polygon2.contains(visi_3)==True)
or(polygon2.contains(visi_4)==True)
or(polygon2.contains(visi_5)==True):
                                w_d_a.append(N[j])
                                w_d_b.append(diff_num[i])

                            if
polygon2.disjoint(visi_pol)==True:
                                pass

                            if
```

```
polygon2.intersects(visi_pol)==True:
                                        w_d_c.append(N[j])
                                        w_d_d.append(diff_num[i])


                                   else:
                                        print 'barev gcic'
                              else:


                                   pass

          return (w_d_c,w_d_d) # retourn  num walls and diff
angels of  partial visibility



     def visi_diff_diff(self,nnode):

          """
               L.visi_diff_diff(nnode): find out conectivites
bitween diffraction angels
               takes a number of a node of graph Gt and return the
numbers of diffractions
               corners placed in a matrix (2*27), first matrix
corresponds to total visibility
               second matrix corresponds to partial visibility

               ____ some shapely functions are used _____

          contains(): Returns True if the object's interior
contains the boundary
                    and interior of the other object and their
boundaries do not touch at all.
          intersects(): Returns True if the boundary and interior
of the object
                    intersect in any way with those of the other.
          disjoint(): Returns True if the boundary and interior
of the object
                    do not intersect at all with those of the
other.

          """
          polygon2 = self.Polygon_tag(nnode)
```

```
          diff_num = self.diffraction_num(nnode)[0]
          d_d_a = [] # first diffraction angle of visility total
          d_d_b = [] # second diffraction angle of visility total
          d_d_c = [] # first diffraction angle of partial
visility
          d_d_d = [] # second diffraction angle of partial
visility


          for i in range(len(diff_num)):
               for j in range(len(diff_num)):
                    if diff_num[i]!= diff_num[j]:
                         diff_x_i,diff_y_i =
self.Gs.pos[diff_num[i]]
                         diff_x_j,diff_y_j =
self.Gs.pos[diff_num[j]]

                         if (diff_x_i!=diff_x_j or
diff_y_i!=diff_y_j):
                              line = sh.LineString([(diff_x_i,
diff_y_i), (diff_x_j, diff_y_j)])
                              if polygon2.contains(line)==True:
                                   d_d_a.append(diff_num[i])
                                   d_d_b.append(diff_num[j])

                              if polygon2.disjoint(line)==True:
                                   pass

                              if polygon2.intersects(line)==True:
                                   d_d_c.append(diff_num[i])
                                   d_d_d.append(diff_num[j])


                              else:
                                   pass
                         else:
                              pass

          return (d_d_a,d_d_b)

     def node_visi_direct(self,nnode,nn):

          """
               L.node_visi_direct(nnode,nn):return a dictionary
```

```
with all numbers of walls
                and diffraction angels with whom nnode has direct
visibility in a node Gt
                _____________________________________________

                     ex: s=L.node_visi_direct(2,29)
                In s[29]
                Out[6]: [12, 11, 16, 342, 333, 22, 328, 24, 23,
                335, 334, -273, -279, -275, -266, -262]
           """
           a1_w_w,a2_w_w = self.visi_wall_wall(nnode)
           a1_d_w,a2_d_w = self.visi_diff_wall(nnode)
           a1_d_d,a2_d_d = self.visi_diff_diff(nnode)
           a1_w_d,a2_w_d = self.visi_wall_diff(nnode)

           a = []
           b = []
           c = []
           d = []
           vis = {}
           nk = {}
           for i in range(len(a1_w_w)):
                if (a1_w_w[i]==nn):
                     a.append(a2_w_w[i])
                else:
                     pass
           for j in range(len(a1_d_w)):
                if (a1_d_w[j]==nn):
                     b.append(a2_d_w[j])
                else:
                     pass
           for k in range(len(a1_d_d)):
                if (a1_d_d[k]==nn):
                     c.append(a2_d_d[k])
                else:
                     pass
           for h in range(len(a1_w_d)):
                if (a1_w_d[h]==nn):
                     d.append(a2_w_d[h])
                else:
                     pass
           vis[nn] = a+b+c+d
           nk[nn] = vis
```

```
            return (vis[nn])


        def visi_direct(self):

            """
                L.visi_direct(): return a list of all direct
visibility in the indoor environment
                (diffraction angels and walls of indoor environment
are involved)

            """
            ndiff = self.buildGc()[1]
            visi_dir = []

            for i in range(len(self.Gt.node)):
                for j in range(len(self.Gt.node[i]['vnodes'])):
                        if
ndiff.__contains__(self.Gt.node[i]['vnodes'][j]):

        visi_dir.append(self.node_visi_direct(i,self.Gt.node[i]['vno
des'][j]))

            self.visi_dir = visi_dir

        def coords_visi_cycle(self,nn):

            """
                L.coords_visi_cycle(nn): return the  numbers nodes
of Gt which include nn
                also return the coordinates of nodes whit whom nn
in direct visibility

                        _____________________________ ex
29_________________

                        num_cycle=[2, 18]
            """
            coords = []
            n = [] # varification of apartenance of the numbe
rwall/diffraction angle
            num_cycle = [] # number of the cycls which containe
number of wall or number of the diffraction angle
```

```
            visi_dir_con = []
            for i in range(len(self.Gt.node)):
                      n.append(self.node_visi_direct(i,nn))

            for i in range(len(self.Gt.node)):
                       if len(n[i])>1:
                 num_cycle.append(i)# 29 is in cycle 2 and cycle 18


            for i in range(len(num_cycle)):
                 for j in range(len(n[num_cycle[i]])):

      coords.append(self.Gs.pos[n[num_cycle[i]][j]])
            for i in range(len(coords)):

      visi_dir_con.append(sh.LineString([(self.Gs.pos[nn][0],
self.Gs.pos[nn][1]), (coords[i][0], coords[i][1])]))
            return (num_cycle,coords,visi_dir_con)


      def visi_nodes(self,nnode):

            """
                 L.visi_nodes(nnode): return a Graph of nodes and
number of nodes
                 which are engaged in visibility list of a Gt node,
also show them
                 __________ Ex______

                 L.visi_nodes(29)

                  (<networkx.classes.graph.Graph object at
0xb4113cc>,
                 [-202, -200, -199, -179, 354, 353, 352, 83, 85,
323, 104, 324, 103, 282, 281])

            """
            G=nx.Graph()
            ncoin,ndiff= self.buildGc()
            diff_num=[]
            pos_num=[]
            pos={}
            labels={}
            for i in range(len(self.Gt.node[nnode]['vnodes'])):
```

```
                if
ndiff.__contains__(self.Gt.node[nnode]['vnodes'][i]):

     diff_num.append(self.Gt.node[nnode]['vnodes'][i])
          N =
self.Gt.node[nnode]['vnodes'][self.Gt.node[nnode]['vnodes']>0] #
Numbers of segments in Gt node
          for j in range(len(N)):
               pos_num.append(N[j])
          numb = diff_num+pos_num  # numbers of walls/diffraction
angels in a node of Gt


          for i in range(len(numb)):
                    G.add_node(numb[i])

     pos[numb[i]]=(self.Gs.pos[numb[i]][0],self.Gs.pos[numb[i]][1
])
                    labels[numb[i]] = str(numb[i])

          colors=range(20)

          return (G,numb)

     def node_visi_direct_Graph(self,nnode,nn):

          """
               node_visi_direct_Graph(nnode,nn): return a Graph of
visibility:
               visibility between a node and other nodes in the
               same Gt node which are in visibility with nn

               _____________ EX _____________

               L.node_visi_direct_Graph(nnode=2,nn=29)
          """

          G = nx.Graph()
          nvi_by_nn = self.node_visi_direct(nnode,nn)
          pos = {}
          labels =  {}
          G.add_node(nn)
          for i in range(len(nvi_by_nn)):
               G.add_node(nvi_by_nn[i])
```

```
                pos[nvi_by_nn[i]] = (self.Gs.pos[nvi_by_nn[i]][0],
self.Gs.pos[nvi_by_nn[i]][1])
                labels[nvi_by_nn[i]] = str(nvi_by_nn[i])
                G.add_edge(nn,nvi_by_nn[i])
                pos[nn] = (self.Gs.pos[nn][0],self.Gs.pos[nn][1])

        nx.draw(G,pos,font_size=8)
        plt.show()
        return(G)

    def visi_all_nodes(self):

                """
                    L.visi_all_nodes(): return a dictionary with
                    all nodes of Gs which are engaged in
                    visibility.
                """
                d={}

                for j in range(len(self.Gt.node)):
                        for i in
range(len(self.visi_nodes(j)[1])):

    d[self.visi_nodes(j)[1][i]]=self.node_visi_direct(j,self.vis
i_nodes(j)[1][i])
                self.d=d

    def visi_all_indoor(self):

        """
                L.visi_all_indoor(): return graph Gv: Graph of
visibility.
        """

        Gv = nx.Graph()
        pos = {}
        labels = {}

        nnum_all=self.d
        a=nnum_all.keys()
        for i in range(len(nnum_all)):
                for j in range(len(nnum_all[a[i]])):
                Gv.add_node(a[i])
```

```
     pos[a[i]]=(self.Gs.pos[a[i]][0],self.Gs.pos[a[i]][1])
                    labels[a[i]] = str(a[i])
                    Gv.add_node(nnum_all[a[i]][j])

     pos[nnum_all[a[i]][j]]=(self.Gs.pos[nnum_all[a[i]][j]][0],se
lf.Gs.pos[nnum_all[a[i]][j]][1])
                    labels[nnum_all[a[i]][j]] =
str(nnum_all[a[i]][j])
                    Gv.add_edge(a[i],nnum_all[a[i]][j])

          self.Gv_new=Gv
          self.pos_new=pos
          self.labels_new=labels


     def build_visi_all_indoor(self):

          """
               L.visi_all_indoor(): return graph Gv: Graph of
visibility.
          """
          Gv     = nx.Graph()
          pos    = {}
          labels = {}

          nnum_all=self.d
          a=nnum_all.keys()
          for i in range(len(nnum_all)):
                    for j in range(len(nnum_all[a[i]])):
                    Gv.add_node(a[i])

     pos[a[i]]=(self.Gs.pos[a[i]][0],self.Gs.pos[a[i]][1])
                    labels[a[i]] = str(a[i])

                    Gv.add_node(nnum_all[a[i]][j])

     pos[nnum_all[a[i]][j]]=(self.Gs.pos[nnum_all[a[i]][j]][0],se
lf.Gs.pos[nnum_all[a[i]][j]][1])
                    labels[nnum_all[a[i]][j]] =
str(nnum_all[a[i]][j])
                    Gv.add_edge(a[i],nnum_all[a[i]][j])

          self.Gv=Gv
          self.labels=labels
```

```
            self.pos=pos

    def buildGv_new(self):

            """
                L.buildGv_new(): return graph Gv: Graph of
visibility.
            """

            nnum_all = self.d
            a = nnum_all.keys()
            for i in range(len(nnum_all)):
                        for j in range(len(nnum_all[a[i]])):
                        self.Gv_new.add_node(a[i])
                        self.Gv_new.pos[a[i]] = (self.Gs.pos[a[i]][0],
self.Gs.pos[a[i]][1])
                        self.Gv_new.labels[a[i]] = str(a[i])
                        self.Gv_new.add_node(nnum_all[a[i]][j])
                        self.Gv_new.pos[nnum_all[a[i]][j]] =
(self.Gs.pos[nnum_all[a[i]][j]][0],self.Gs.pos[nnum_all[a[i]][j]]
[1])
                        self.Gv_new.labels[nnum_all[a[i]][j]] =
str(nnum_all[a[i]][j])
                        self.Gv_new.add_edge(a[i],nnum_all[a[i]][j])

    def visi_Tx_Rx(self,nnode,p_Tx):

            """
                L.visi_Tx_Rx(nnode,p_Tx):return the numbers of
                walls and diffraction angles which are in
                visibility with Tx/Rx
            """
            Tx_x=p_Tx[0]
            Tx_y=p_Tx[1]
            diff_num= self.diffraction_num(nnode)[0]
            polygon2=self.Polygon_tag(nnode)
            visi_Tx_w=[]
            visi_Tx_diff=[]

    N=self.Gt.node[nnode]['vnodes'][self.Gt.node[nnode]['vnodes'
]>0] # Numbers of segments in Gt node
            for k in range(len(N)):
                n1_x_k,n1_y_k =
self.Gs.pos[self.Gs.neighbors(N[k])[0]]
```

```
                n2_x_k,n2_y_k =
self.Gs.pos[self.Gs.neighbors(N[k])[1]]

                M_w_x,M_w_y = self.Gs.pos[N[k]]

                n1xk,n1yk,n2xk,n2yk=
self.wall_delta(n1_x_k,n1_y_k,n2_x_k,n2_y_k)
                visi_1 = sh.LineString([(Tx_x, Tx_y), (n1_x_k,
n1_y_k)])
                visi_2 = sh.LineString([(Tx_x, Tx_y), (n2_x_k,
n2_y_k)])
                visi_3 = sh.LineString([(Tx_x, Tx_y), (M_w_x,
M_w_y)])
                visi_4 = sh.LineString([(Tx_x, Tx_y), (n1xk,
n1yk)])
                visi_5 = sh.LineString([(Tx_x, Tx_y), (n2xk,
n2yk)])
                if (polygon2.contains(visi_1)==True) or
(polygon2.contains(visi_2)==True)or
(polygon2.contains(visi_3)==True)or
(polygon2.contains(visi_4)==True)or
(polygon2.contains(visi_5)==True):

                    visi_Tx_w.append(N[k])


          for n in range(len(diff_num)):
                diff_x_n=self.Gs.pos[diff_num[n]][0]
                diff_y_n=self.Gs.pos[diff_num[n]][1]
                visi_1 = sh.LineString([(Tx_x, Tx_y),
(diff_x_n,diff_y_n)])
                if (polygon2.contains(visi_1)==True):
                    visi_Tx_diff.append(diff_num[n])
          visi_Tx = visi_Tx_diff+visi_Tx_w
          return(visi_Tx)


     def contains_Tx_Rx(self,p_Tx):

          """

                L.contains_Tx_Rx(x,y):return the number og Gt node
which contains Tx or Rx
```

```
                    _________________  ATANTION ! _________________

                         L.contains_Tx_Rx(p_Tx)[0]
                    """
                    Tx_x,Tx_y=p_Tx
                    n=[]
                    for i in range(len(self.Gt.node)):
                         if
(self.Gt.node[i]['polyg'].disjoint(sh.Point(Tx_x,Tx_y))==False):
                              n.append(i)
                         else:
                                   pass

                    return (n)

               def connect_Tx(self,p_Tx,node_Tx):

                    """
                         L.connect_Tx(Tx_x,Tx_y,node_Tx): return a graph
which connect Tx
                         with nodes and diffraction angels of node of Gt
                    """
                    Tx_x,Tx_y = p_Tx
                    Tx = str('Tx')
                    pos = {}
                    labels = {}
                    G_Tx = nx.Graph()
                    G_Tx.add_node('Tx',node_color='y')
                    pos['Tx'] = (Tx_x,Tx_y)
                    labels['Tx'] = str(Tx)
                    visi_list=self.visi_Tx_Rx(node_Tx,p_Tx)
                    for i in range(len(visi_list)):
                         G_Tx.add_node(visi_list[i])
                         pos[visi_list[i]] =
(self.Gs.pos[visi_list[i]][0],self.Gs.pos[visi_list[i]][1])
                         labels[visi_list[i]] = str(visi_list[i])
                         G_Tx.add_edge('Tx',visi_list[i])
                    return (G_Tx,pos,labels)

               def connect_Rx(self,p_Rx,node_Rx):

                    """
                         L.connect_Rx(p_Rx,node_Rx):return a graph which
connect Rx
```

```
                with nodes and diffraction angels of node of Gt
            """
            Rx_x,Rx_y = p_Rx
            Rx = str('Rx')
            pos = {}
            labels = {}
            G_Rx = nx.Graph()
            G_Rx.add_node('Rx',node_color='o')
            pos['Rx'] = (Rx_x,Rx_y)
            labels['Rx'] = str(Rx)
            visi_list = self.visi_Tx_Rx(node_Rx,p_Rx)
            for i in range(len(visi_list)):
                G_Rx.add_node(visi_list[i])

    pos[visi_list[i]]=(self.Gs.pos[visi_list[i]][0],self.Gs.pos[
visi_list[i]][1])
                labels[visi_list[i]] = str(visi_list[i])
                G_Rx.add_edge('Rx',visi_list[i])
            return (G_Rx,pos,labels)


    def connect_Tx_Rx(self,p_Tx,node_Tx,p_Rx,node_Rx):

            """
            L.connect_Tx_Rx(p_Tx,node_Tx,p_Rx,node_Rx): return a
    graph which
                 connect Tx et Rx with nodes and diffraction angels
            of node of Gt
            """
            Tx_x,Tx_y = p_Tx
            Rx_x,Rx_y = p_Rx

            node_Tx = self.contains_Tx_Rx(p_Tx)[0]
            node_Rx = self.contains_Tx_Rx(p_Rx)[0]
            Tx = str('Tx')
            Rx = str('Rx')
            pos = {}
            labels = {}
            G_Tx_Rx = nx.Graph()
            G_Tx_Rx.add_node('Tx',node_color='y')
            pos['Tx'] = (Tx_x,Tx_y)
            labels['Tx'] = str(Tx)
            visi_list= self.visi_Tx_Rx(node_Tx,p_Tx)
            for i in range(len(visi_list)):
```

```
                G_Tx_Rx.add_node(visi_list[i])
                pos[visi_list[i]] = (self.Gs.pos[visi_list[i]][0],
self.Gs.pos[visi_list[i]][1])
                labels[visi_list[i]] = str(visi_list[i])
                G_Tx_Rx.add_edge('Tx',visi_list[i])

          G_Tx_Rx.add_node('Rx',node_color='o')
          pos['Rx']=(Rx_x,Rx_y)
          labels['Rx'] = str(Rx)
          visi_list= self.visi_Tx_Rx(node_Rx,p_Rx)
          for j in range(len(visi_list)):
               G_Tx_Rx.add_node(visi_list[j])
               pos[visi_list[j]]=( self.Gs.pos[visi_list[j]][0],
self.Gs.pos[visi_list[j]][1])
               labels[visi_list[j]] = str(visi_list[j])
               G_Tx_Rx.add_edge('Rx',visi_list[j])

          return (G_Tx_Rx,pos)

     def signature_Tag(self,Gv_Tx_Rx,node_Tx,p_Tx,node_Rx,p_Rx):

          """
               L.signature_Tag(Gv_Tx_Rx,node_Tx,p_Tx,node_Rx,p_Tx
               ):return possible signatures between Tx and Rx
          """
          Tx=str('Tx')
          Rx=str('Rx')
          Tx_x,Tx_y = p_Tx
          Rx_x,Rx_y = p_Rx

          signature=[]
          if node_Tx==node_Rx:

               if
self.Polygon_tag(node_Tx).contains(sh.LineString([(Tx_x,Tx_y),
(Rx_x,Rx_y)]))==True:

                         signature_shortest= ['Tx','Rx']
               else:

     signature_shortest=nx.dijkstra_path(Gv_Tx_Rx,'Tx','Rx')
          else:

     signature_shortest=nx.dijkstra_path(Gv_Tx_Rx,'Tx','Rx')
```

```
            visi_list_Tx=self.visi_Tx_Rx(node_Tx,p_Tx)
            visi_list_Rx=self.visi_Tx_Rx(node_Rx,p_Rx)
            for i in range(len(visi_list_Tx)):
                 for j in range(len(visi_list_Rx)):

     signature.append(nx.dijkstra_path(Gv_Tx_Rx,visi_list_Tx[i],v
isi_list_Rx[j]))

            return (signature_shortest,signature)


     def coords_Tx_Rx(self):

            """
                 L.coords_Tx_Rx(): definite the boundary of Gs graph
and
                 return Gs_XMAX, Gs_xmin, Gs_YMAX, Gs_ymin
                  with +,-3,4 for position of Tx and Rx
            """
            gsx = []
            gsy = []
            kay = self.Gs.pos.keys()
            for i in range(len(self.Gs.pos)):
                 gsx.append(self.Gs.pos[kay[i]][0])
                 gsy.append(self.Gs.pos[kay[i]][1])


            Gs_XMAX=max(gsx)-3
            Gs_xmin=min(gsx)+3
            Gs_YMAX=max(gsy)-4
            Gs_ymin=min(gsy)+4

            return (Gs_XMAX,Gs_xmin,Gs_YMAX,Gs_ymin)

     def boundary_Gs(self):

            """
                 L.boundary_Gs(): returns  xmax,xmin,ymax,ymin of Gs
construction
            """
            gsx = []
            gsy = []
            kay = self.Gs.pos.keys()
```

```
            for i in range(len(self.Gs.pos)):
                gsx.append(self.Gs.pos[kay[i]][0])
                gsy.append(self.Gs.pos[kay[i]][1])
            xmax = max(gsx)
            xmin = min(gsx)
            ymax = max(gsy)
            ymin = min(gsy)
            return (xmax,xmin,ymax,ymin)

    def Tx_Rx_pos(self):

            """
                L.Tx_Rx_pos(): give random positions to Tx and Rx
in the Gs graph and
                return p_Tx,p_Rx
            """
            Gs_XMAX,Gs_xmin,Gs_YMAX,Gs_ymin = self.coords_Tx_Rx()

            Tx_x = random.uniform(Gs_xmin,Gs_XMAX)
            Tx_y = random.uniform(Gs_ymin,Gs_YMAX)
            Rx_x = random.uniform(Gs_xmin,Gs_XMAX)
            Rx_y = random.uniform(Gs_ymin,Gs_YMAX)
            p_Tx = np.array([Tx_x,Tx_y])
            p_Rx = np.array([Rx_x,Rx_y])

            return (p_Tx,p_Rx)


    def Gv_Tx_Rx(self,Tx_x,Tx_y,Rx_x,Rx_y):

            """
                Gv_Tx_Rx(,Tx_x,Tx_y,Rx_x,Rx_y) crate a Graph
Gv_Tx_Rx,
                which connect the graph visibility Gv with Tx and
Rx
                return(Gv_Tx_Rx)
            """
            Tx=str('Tx')
            Rx=str('Rx')
            Gv_Tx_Rx=nx.Graph()
            pos={}
            labels={}
            node_Tx= self.contains_Tx_Rx(Tx_x,Tx_y)[0]
            node_Rx=self.contains_Tx_Rx(Rx_x,Rx_y)[0]
```

```
            nnum_all=self.visi_all_nodes()
            a=nnum_all.keys()

            # Connect Tx to a cycle of Gt

            Gv_Tx_Rx.add_node('Tx',node_color='y')
            pos['Tx']=(Tx_x,Tx_y)
            labels['Tx'] = str(Tx)
            visi_list= self.visi_Tx_Rx(node_Tx,Tx_x,Tx_y)
            for i in range(len(visi_list)):
                 Gv_Tx_Rx.add_node(visi_list[i])
                 pos[visi_list[i]]=( self.Gs.pos[visi_list[i]][0],
self.Gs.pos[visi_list[i]][1])
                 labels[visi_list[i]] = str(visi_list[i])
                 Gv_Tx_Rx.add_edge('Tx',visi_list[i])

            # Connect Rx to a cycle of Gt

            Gv_Tx_Rx.add_node('Rx',node_color='o')
            pos['Rx']=(Rx_x,Rx_y)
            labels['Rx'] = str(Rx)
            visi_list=self.visi_Tx_Rx(node_Rx,Rx_x,Rx_y)
            for i in range(len(visi_list)):
                 Gv_Tx_Rx.add_node(visi_list[i])

     pos[visi_list[i]]=(self.Gs.pos[visi_list[i]][0],self.Gs.pos[
visi_list[i]][1])
                 labels[visi_list[i]] = str(visi_list[i])
                 Gv_Tx_Rx.add_edge('Rx',visi_list[i])

            # Cration of a grapf with all visibilitys

            for i in range(len(nnum_all)):
                 for j in range(len(nnum_all[a[i]])):
                      Gv_Tx_Rx.add_node(a[i])

     pos[a[i]]=(self.Gs.pos[a[i]][0],self.Gs.pos[a[i]][1])
                      labels[a[i]] = str(a[i])
                      Gv_Tx_Rx.add_node(nnum_all[a[i]][j])

     pos[nnum_all[a[i]][j]]=(self.Gs.pos[nnum_all[a[i]][j]][0],se
lf.Gs.pos[nnum_all[a[i]][j]][1])
                      labels[nnum_all[a[i]][j]] =
str(nnum_all[a[i]][j])
```

```
                    Gv_Tx_Rx.add_edge(a[i],nnum_all[a[i]][j])

          H=Gv_Tx_Rx.subgraph('Rx')
          Y=Gv_Tx_Rx.subgraph('Tx')
          nx.draw(Gv_Tx_Rx,pos,node_color='b')

nx.draw_networkx_nodes(Y,pos,font_size=8,node_color='g')

nx.draw_networkx_edges(Y,pos,font_size=8,node_color='g')

nx.draw_networkx_nodes(H,pos,font_size=8,node_color='r')

nx.draw_networkx_edges(H,pos,font_size=8,node_color='r')
          plt.draw()
          plt.show()
          return (Gv_Tx_Rx)

def Gv_Tx_Rx(self,p_Tx,p_Rx):

          """
               L.Gv_Tx_Rx(p_Tx,p_Rx): crate a Graph Gv_Tx_Rx,
               which connect the graph visibility Gv with Tx and
          Rx
               return(Gv_Tx_Rx)
          """
          Tx_x,Tx_y = p_Tx
          Rx_x,Rx_y = p_Rx

          Tx = str('Tx')
          Rx = str('Rx')

          Gv,pos_v,labels_v=self.loadlo()
          Gv_Tx_Rx = Gv
          node_Tx = self.contains_Tx_Rx(p_Tx)[0]
          node_Rx = self.contains_Tx_Rx(p_Rx)[0]

          # Connect Tx to a cycle of Gt

          Gv_Tx_Rx.add_node('Tx',node_color='y')
          pos_v['Tx'] = (Tx_x,Tx_y)
          labels_v['Tx'] = str(Tx)
          visi_list_Tx= self.visi_Tx_Rx(node_Tx,p_Tx)
          for i in range(len(visi_list_Tx)):
               Gv_Tx_Rx.add_node(visi_list_Tx[i])
```

```
     pos_v[visi_list_Tx[i]]=(self.Gs.pos[visi_list_Tx[i]][0],self
.Gs.pos[visi_list_Tx[i]][1])
                labels_v[visi_list_Tx[i]] = str(visi_list_Tx[i])
                Gv_Tx_Rx.add_edge('Tx',visi_list_Tx[i])

          # Connect Rx to a cycle of Gt

          Gv_Tx_Rx.add_node('Rx',node_color='o')
          pos_v['Rx'] = (Rx_x,Rx_y)
          labels_v['Rx'] = str(Rx)
          visi_list_Rx = self.visi_Tx_Rx(node_Rx,p_Rx)
          for i in range(len(visi_list_Rx)):
                Gv_Tx_Rx.add_node(visi_list_Rx[i])
                pos_v[visi_list_Rx[i]]=(
self.Gs.pos[visi_list_Rx[i]][0], self.Gs.pos[visi_list_Rx[i]][1])
                labels_v[visi_list_Rx[i]] = str(visi_list_Rx[i])
                Gv_Tx_Rx.add_edge('Rx',visi_list_Rx[i])


          H = Gv_Tx_Rx.subgraph('Rx')
          Y = Gv_Tx_Rx.subgraph('Tx')

          nx.draw(Gv_Tx_Rx,pos_v,node_color='b')

     nx.draw_networkx_nodes(Y,pos_v,font_size=8,node_color='g')

     nx.draw_networkx_edges(Y,pos_v,font_size=8,node_color='g')

     nx.draw_networkx_nodes(H,pos_v,font_size=8,node_color='r')

     nx.draw_networkx_edges(H,pos_v,font_size=8,node_color='r')
          self.showGs()
          plt.draw()
          plt.show()
          return (Gv_Tx_Rx)

     def one_ray(self,S,p_Tx,p_Rx):

          """
               L.one_ray(S,p_Tx,p_Rx):build a graph og a ray by
using a signature
               ____________________________________
```

```
                type(S)='list'
                type(p_Rx) = 'numpy.ndarray'
            """
            Tx_x,Tx_y = p_Tx
            Rx_x,Rx_y = p_Rx

            Tx = str('Tx')
            Rx = str('Rx')
            G_sign = nx.Graph()
            pos = {}
            labels = {}
            G_sign.add_node('Tx',node_color='g')
            pos['Tx'] = (Tx_x,Tx_y)
            labels['Tx'] = str(Tx)
            G_sign.add_node('Rx',node_color='r')
            pos['Rx'] = (Rx_x,Rx_y)
            labels['Rx'] = str(Rx)
            for i in range(len(S)):
                G_sign.add_node(S[i])

        pos[S[i]]=(self.Gs.pos[S[i]][0],self.Gs.pos[S[i]][1])
                labels[S[i]] = str(S[i])

            for i in range(len(S)-1):
                j=i+1
                G_sign.add_edge(S[i],S[j])

            G_sign.add_edge('Tx',S[0])
            G_sign.add_edge('Rx',S[len(S)-1])
            # Calculation of the length of a ray by using sou-rays
            dist=[]
            for i in range(len(S)-1):
                    j=i+1
                    dist.append(sh.Point(self.Gs.pos[S[i]][0],
self.Gs.pos[S[i]][1]).distance(sh.Point(self.Gs.pos[S[j]][0],
self.Gs.pos[S[j]][1])))

        dist.append(sh.Point(Tx_x,Tx_y).distance(sh.Point(self.Gs.po
s[S[0]][0], self.Gs.pos[S[0]][1])))

        dist.append(sh.Point(Rx_x,Rx_y).distance(sh.Point(self.Gs.po
s[S[len(S)-1]][0], self.Gs.pos[S[len(S)-1]][1])))
            dist_sum = sum(dist)
```

```
            H = G_sign.subgraph('Rx')
            Y = G_sign.subgraph('Tx')
            nx.draw(G_sign,pos,node_color='b')

        nx.draw_networkx_nodes(Y,pos,font_size=8,node_color='g')

        nx.draw_networkx_edges(Y,pos,font_size=8,node_color='g')

        nx.draw_networkx_nodes(H,pos,font_size=8,node_color='r')

        nx.draw_networkx_edges(H,pos,font_size=8,node_color='r')
            self.showGs()
            plt.draw()
            plt.savefig("ray.png")
            plt.show()

            return (G_sign,dist_sum)

        def all_rays(self,signature,p_Tx,p_Rx):

            """
                L.all_rays(self,signature,p_Tx,p_Rx): build a graph
of all possible
                rays between Tx and Rx by using signatures
            """

            Tx_x,Tx_y = p_Tx
            Rx_x,Rx_y = p_Rx
            sign = []
            dist_reng = []
            dist_all_i = []

            for i in range(len(signature)):

        dist_all_i.append(self.one_ray(signature[i],p_Tx,p_Rx)[1])
            a=np.argsort(dist_all_i)

            for k in range(len(signature)):

                self.one_ray(signature[k],p_Tx,p_Rx)[0]
            plt.draw()
            self.showGs()
            plt.show()
```

```
    def shortest_ray(self,S,p_Tx,p_Rx):

        """
            L.shortest_ray(S,p_Tx,p_Rx):build a graph of the
shortest ray by using a signature
            ____________________________________________

            type(S)='list'
            type(p_Rx) = 'numpy.ndarray'
        """
        Tx_x,Tx_y = p_Tx
        Rx_x,Rx_y = p_Rx

        Tx=str('Tx')
        Rx=str('Rx')
        G_sh_ray=nx.Graph()
        pos={}
        labels={}
        G_sh_ray.add_node('Tx',node_color='g')
        pos['Tx']=(Tx_x,Tx_y)
        labels['Tx'] = str(Tx)
        G_sh_ray.add_node('Rx',node_color='r')
        pos['Rx']=(Rx_x,Rx_y)
        labels['Rx'] = str(Rx)
        for i in range(len(S)-2):
            j=i+1
            G_sh_ray.add_node(S[j])
            pos[S[j]]=( self.Gs.pos[S[j]][0],
self.Gs.pos[S[j]][1])
            labels[S[j]] = str(S[j])

        for i in range(1,len(S)-2):
                j=i+1
                G_sh_ray.add_edge(S[i],S[j])
        G_sh_ray.add_edge('Tx',S[1])
        G_sh_ray.add_edge('Rx',S[len(S)-2])


        H = G_sh_ray.subgraph('Rx')
        Y = G_sh_ray.subgraph('Tx')
        nx.draw(G_sh_ray,pos,node_color='b')

    nx.draw_networkx_nodes(Y,pos,font_size=8,node_color='g')
```

```
        nx.draw_networkx_edges(Y,pos,font_size=8,node_color='g')

        nx.draw_networkx_nodes(H,pos,font_size=8,node_color='r')

        nx.draw_networkx_edges(H,pos,font_size=8,node_color='r')
            self.showGs()
            plt.draw()
            plt.show()

            return (G_sh_ray)

        def dir_ray_sign(self,p_Tx,p_Rx):

            """
                L.dir_ray_sign(p_Tx,p_Rx): takes transmitter and
receiver coordinates
                and returns the shortest ray and his signature
            """
            Tx_x,Tx_y = p_Tx
            Rx_x,Rx_y = p_Rx
            xmin = min(Tx_x,Rx_x)
            xmax = max(Tx_x,Rx_x)
            ymin = min(Tx_y,Rx_y)
            ymax = max(Tx_y,Rx_y)

            seg= self.clip(xmin,xmax,ymin,ymax)

            # Creation of direct signature and ray

            sign=[]
            sign_Tx_Rx=[]
            dist=[]
            sign1=[]
            line = sh.LineString([(Tx_x, Tx_y), (Rx_x, Rx_y)])
            for i in range(len(seg)):
                n1,n2= self.Gs.neighbors(seg[i]+1)
                line_i = sh.LineString([(self.Gs.pos[n1][0],
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])
                if line.crosses(line_i)==True:
                    sign1.append(seg[i]+1)
                    dist.append(sh.Point(Tx_x,
Tx_y).distance(sh.LineString([(self.Gs.pos[n1][0],
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])))
```

```
                dist1=np.argsort(dist)
            for i in range(len(dist)):
                sign.append(sign1[dist1[i]])

            fig = plt.figure(1, dpi=90)
            ax = fig.add_subplot(1,1,1)

            self.plot_coords(ax, line)
            self.plot_bounds(ax, line)
            self.plot_line(ax, line)
            sign_Tx_Rx.append('Tx')

            for i in range(len(sign)):
                        sign_Tx_Rx.append(sign[i])

            sign_Tx_Rx.append('Rx')

            self.showGs()
            plt.draw()
            plt.show()
            return (sign_Tx_Rx)

    def clip_ray_sign(self,p_Tx,p_Rx):

            """
                L.dir_ray_sign(p_Tx,p_Rx): takes transmitter and
receiver coordinates
                and returns the shortest ray and his signature
            """
            Tx_x,Tx_y = p_Tx
            Rx_x,Rx_y = p_Rx
            xmin = min(Tx_x,Rx_x)
            xmax = max(Tx_x,Rx_x)
            ymin = min(Tx_y,Rx_y)
            ymax = max(Tx_y,Rx_y)

            seg= self.clip(xmin,xmax,ymin,ymax)

            # Creation of direct signature and ray

            sign=[]
            sign_Tx_Rx=[]
            dist=[]
            sign1=[]
```

```
            line = sh.LineString([(Tx_x, Tx_y), (Rx_x, Rx_y)])
            for i in range(len(seg)):
                n1,n2= self.Gs.neighbors(seg[i]+1)
                line_i = sh.LineString([(self.Gs.pos[n1][0],
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])
                if line.crosses(line_i)==True:
                    sign1.append(seg[i]+1)
                    dist.append(sh.Point(Tx_x,
Tx_y).distance(sh.LineString([(self.Gs.pos[n1][0],
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])))
                dist1=np.argsort(dist)
            for i in range(len(dist)):
                sign.append(sign1[dist1[i]])

            sign_Tx_Rx.append('Tx')

            for i in range(len(sign)):
                    sign_Tx_Rx.append(sign[i])

            sign_Tx_Rx.append('Rx')

            return (sign_Tx_Rx)


    def clip_all_sign_ray(self,Txs,Rxs):


            """
                L.clip_all_sign(Txs,Rxs): takes transmitters and
                receivers coordinates in type 'numpy.ndarray' and
                returns all shortest signatures which can existe
                between every transmitter and every receiver

            __________EX_________

            Txs = np.array([[-27.3569, 14.7833000002, 1.5],[-
24.3661, 12.3050000002, 1.5]])

            Rxs = np.array([[-4.14179999998, 8.86029999983, 1.5],[-
27.3569, 15.7833000002, 1.5]])

            """

            fig = plt.figure(1, dpi=90)
```

```
            ax = fig.add_subplot(1,1,1)
            signs_all=[]
            for i in range(len(Txs)):
                 for j in range(len(Rxs)):
                      Tx_x,Tx_y=Txs[:,0][i],Txs[:,1][i]
                      Rx_x,Rx_y=Rxs[:,0][j],Rxs[:,1][j]
                      line =
sh.LineString([(Tx_x,Tx_y),(Rx_x,Rx_y)])
                      plot(Tx_x, Tx_y, 'o', color='#6699cc',
zorder=1)
                      plot(Rx_x, Rx_y, 'o', color='#ffcc33',
zorder=1)
                      self.plot_coords(ax, line)
                      self.plot_bounds(ax, line)
                      self.plot_line(ax, line)
                      p_Tx=array([Tx_x,Tx_y])
                      p_Rx=array([Rx_x,Rx_y])

     signs_all.append(self.clip_ray_sign(p_Tx,p_Rx))


            self.showGs()
            plt.draw()
            plt.show()
            return (signs_all)


     def displot(pt,ph,col='black'):

            """
                 pt: tail points array (2 x (2*Nseg))
                 ph: head points array (2 x (2*Nseg))

            """
            Nseg = np.shape(pt)[1]
            pz = np.empty((2,))
            pn = np.zeros((2,))
            for i in range(Nseg):
                 pz = np.vstack((pz,pt[:,i],ph[:,i],pn))

            m1 = np.array([0,0,1])
            mask = np.kron(np.ones((2,Nseg)),m1)
            pzz = pz[1:,:].T
            vertices = np.ma.masked_array(pzz,mask)
```

```
     plt.plot(vertices[0,:],vertices[1,:],color=col,linewidth=3)

     COLOR = {
     True:  '#6699cc',
     False: '#ffcc33'
      }

     def v_color(self,ob):
          return COLOR[ob.is_simple]

     def plot_coords(self,ax, ob):

          x, y = ob.xy
          ax.plot(x, y, 'o', color='#999999', zorder=1)

     def plot_bounds(self,ax, ob):

          x, y = zip(*list((p.x, p.y) for p in ob.boundary))
          ax.plot(x, y, 'o', color='#000000', zorder=1)

     def plot_line(self,ax, ob):

          x, y = ob.xy
          ax.plot(x, y, color=v_color(ob), alpha=0.7,
linewidth=3, solid_capstyle='round', zorder=2)



     def loadlo(self,_filelo='exemple.lo'):

          """
                    L.loadlo()
                                        Load Layout's lo file which
contains Gv graph Values and reconstruct Gv graph
          """
          Gv_re = nx.Graph()
          pos = {}
          labels = {}
          filelo = getlong(_filelo,'layout')
          if os.path.isfile(filelo):
               data = io.loadmat(filelo,appendmat=False)
               # Reconstructs Gv from data
               Gv_lab = data['lab']
```

```
                pos_val = data['p_val']
                pos_keys = data['p_keys']
                Gv_edges = data['edges']
                Gv_nodes = data['node']

                for i in range(len(pos_keys)):
                                Gv_re.add_node(Gv_nodes[i][0])
                                pos[pos_keys[i][0]] =
(pos_val[i][0],pos_val[i][1])
                                labels[Gv_lab[i][0]] =
str(Gv_lab[i][0])


                for j in range(len(Gv_edges)):
                     Gv_re.add_edge(Gv_edges[j,0],Gv_edges[j,1])



          return (Gv_re,pos,labels)


     def savelo(self,_filelo='exemple.lo'):

          """
               L.savelo(_filelo)
               _filelo is an *.lo file
               Create L.lo file

               ----------------------
               L.savelo('example.lo')
               ----------------------
          """
          # Create Gv
          self.build_visi_all_indoor()

          # Create lo file
          filelo=getlong(_filelo,'layout')

          # Writing into lo file
          if os.path.isfile(filelo):
               print filelo,' already exist'
          else:
               print 'create ',filelo,' file'
```

```
                data=dict(lab=
self.labels.keys(),node=self.Gv.nodes(),edges=self.Gv.edges(),p_v
al=self.pos.values(),p_keys=self.pos.keys())

                io.savemat(filelo,data,appendmat=False)

    def union_Tx_Rx(self,p_Tx,p_Rx):

        """
            L.union_Tx_Rx(Tx_x,Tx_y,Rx_x,Rx_y)
             Union of G_Tx( G_Tx = Gv nodes with Tx in Gt node)
and
             G_Rx ( G_Rx = Gv nodes with Tx in Gt node)graphs

            __Used methodes from shapely___

            G_Tx=L.connect_Tx(p_Tx,node_Tx)
            G_Rx=L.connect_Rx(p_Rx,node_Rx)
        """
        Tx_x,Tx_y = p_Tx
        Rx_x,Rx_y = p_Rx
        pos = {}
        labels = {}
        node_Tx = self.contains_Tx_Rx(p_Tx)[0]
        node_Rx = self.contains_Tx_Rx(p_Rx)[0]
        G_Tx,posTx,labelsTx = self.connect_Tx(p_Tx,node_Tx)
        G_Rx,posRx,labelsRx = self.connect_Rx(p_Rx,node_Rx)
        pos.update(posTx,**posRx)
        labels.update(labelsTx,**labelsRx)
        G_Tx_Rx = nx.union(G_Tx,G_Rx)
        return (G_Tx_Rx,pos,labels)

    def max_min(self,x1,x2):

        """
            L.max_min(self,p_Tx,p_Rx):return max and min valus
of two points

        """

        xmin=min(x1,x2)
        xmax=max(x1,x2)

        return (xmin,xmax)
```

```
    def delta(self,x1,x2):

        """
            L.delta(x1,x2): calculate the different between two
numbers
        """
        xmin,xmax= self.max_min(x1,x2)
        if (xmin>0 and xmax>0):
            deltx=xmax-xmin
        if (xmin<0 and xmax<0):
            deltx=abs(xmin)-abs(xmax)
        if (xmin<=0 and xmax>=0):
            deltx=xmax+abs(xmin)
        else:
            pass
        return (abs(deltx))


    def clip_Gv(self,p_Tx,p_Rx):

        """
            S.clip( Tx_x,Tx_y,Rx_x,Rx_y):return the list of
            edges from Gv graph which cross or belong to the
            clipping zone

        Algorith :
            1) Determine all segment outside the clipping zone
               4 condition to test
            2) Union of the 4 conditions
            3) setdiff1d between the whole array of segments
and the segments outside
        """

        p0=[]
        p1=[]
        Gv_re,pos_re,labels_re = self.loadlo()
        Gv_nodes = Gv_re.nodes()
        Tx_x,Tx_y = p_Tx
        Rx_x,Rx_y = p_Rx

        for i in range(len(Gv_nodes)):
```

```
                nn= self.Gs.neighbors(Gv_nodes[i]) # number of node
                    p0.append(self.Gs.pos[nn[0]])
                    p1.append(self.Gs.pos[nn[1]])
            xmin,xmax=self.max_min(p_Tx[0],p_Rx[0])
            ymin,ymax=self.max_min(p_Tx[1],p_Rx[1])

            p0=array(p0)
            p1=array(p1)
            maxx = maximum(p0[:,0],p1[:,0])
            maxy = maximum(p0[:,1],p1[:,1])
            minx = minimum(p0[:,0],p1[:,0])
            miny = minimum(p0[:,1],p1[:,1])

            nxp = np.nonzero(maxx<xmin)[0]
            nxm = np.nonzero(minx>xmax)[0]
            nyp = np.nonzero(maxy<ymin)[0]
            nym = np.nonzero(miny>ymax)[0]


            Ne=len(Gv_nodes)
            iseg = arange(Ne)

            u1   = union1d(nyp,nym)
            u2   = union1d(nxp,nxm)
            u   = union1d(u1,u2)
            sers=setdiff1d(iseg,u)
            return (sers)


    def Gv_ch(self,sers):

            """
                L.Gv_ch(sers):creation Gv_ch which is the clip of
Gv graph
            """
            Gv_re,pos,labels= self.loadlo()
            Gv_ch=nx.Graph()
            a=array(Gv_re.edges())[:,0]
            b=array(Gv_re.edges())[:,1]
            pos = {}
            labels = {}

            for i in range(len(sers)):
```

```
                if Gv_re.__contains__(sers[i])==True:
                    Gv_ch.add_node(sers[i])

    pos[sers[i]]=(self.Gs.pos[sers[i]][0],self.Gs.pos[sers[i]][1
])
                    labels[sers[i]] = str(sers[i])
                    for j in range(len(b[find(a==sers[i])])):
                        Gv_ch.add_edge(sers[i],sers[j])




        return (Gv_ch,pos,labels)


    def signatutes_U_m(self,p_Tx,p_Rx):

        """
            L.signatutes_U_m(self,p_Tx,p_Rx): use transmitter
and receiver coordinates
            and return some of possible signatures between Tx
and Rx

                _______! use ______

            L.all_rays(signature=L.signatutes_U_m,p_Tx,p_Rx)
to display the rays
        """
        Gv_re,pos,labels=self.loadlo()
        NroomTx=self.contains_Tx_Rx(p_Tx)[0]
        NroomRx=self.contains_Tx_Rx(p_Rx)[0]
        visi_Tx=self.visi_Tx_Rx(NroomTx,p_Tx)
        visi_Rx=self.visi_Tx_Rx(NroomRx,p_Rx)

        sign=[]
        for i in range(len(visi_Tx)):
            for j in range(len(visi_Rx)):
                if (i!=j):
                    try:

    path=nx.dijkstra_path(Gv_re,visi_Tx[i],visi_Rx[j])
                        sign.append(path)
                    except:
                        pass
```

```
                                print 'No path
between',visi_Tx[i],visi_Rx[j]
                    else:
                         path=[visi_Tx[i]]
                         sign.append(path)

          return (sign)

def segs_cross(self, line):

          """
               L.segs_cross(line): return segment numbers crossed
by a line
          """
          segs_cross=[]

          nodes= self.Gs.nodes()

          for i in range(len(nodes)):
               if nodes[i]>=0:
                    n1,n2= self.Gs.neighbors(nodes[i])#+1 for
coherancy with PyRay
                    line_i = sh.LineString([(self.Gs.pos[n1][0],
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])
                    if line.crosses(line_i)==True:
                         segs_cross.append(nodes[i])

               else:
                    pass

          return (segs_cross)


     def sign_cross(self,pTx,pRx,P,seg_num):

          """
               L.sign_cross(pTx,pRx,P,seg_num): return a possible
               signature at list with one reflexion
          """
          px=P.x
          py=P.y
          Tx_x, Tx_y=pTx
          Rx_x, Rx_y=pRx
          line_Tx_p = sh.LineString([(Tx_x, Tx_y), (px,py)])
```

```
            line_Rx_p = sh.LineString([ (px,py),(Rx_x, Rx_y)])
            segs_Tx_p= self.segs_cross(line_Tx_p)
            segs_Rx_p= self.segs_cross(line_Rx_p)
            dist_Tx=[]
            dist_Rx=[]
            sign=[]
            for i in range(len(segs_Tx_p)):
                 d=
sh.Point(Tx_x,Tx_y).distance(sh.Point(self.Gs.pos[segs_Tx_p[i]][0
],self.Gs.pos[segs_Tx_p[i]][1]))
                 dist_Tx.append(d)
            distTx=np.argsort(dist_Tx)
            for j in range(len(segs_Rx_p)):
                 la=sh.Point(self.Gs.pos[segs_Rx_p[j]][0],
self.Gs.pos[segs_Rx_p[j]][1]).distance(sh.Point(Tx_x,Tx_y))
                 dist_Rx.append(la)
            distRx=np.argsort(dist_Rx)
            sign.append(0)
            for k in range(len(segs_Tx_p)):
                 sign.append(segs_Tx_p[distTx[k]])
            sign.append(seg_num)
            for h in range(len(segs_Rx_p)):
                 sign.append(segs_Rx_p[distRx[h]])
            sign.append(0)
            return (sign)


     def cross_shortest(self,line):

            """

                 L.cross_shortest(segs_cross): return all points
crossed by the line
                 line cross the shortest ray in the middle of the
                 shortest and they form an angle of 90 degree
                 together
            """

            segs_cross= self.segs_cross(line)

            p_int=[]
            for k in range(len(segs_cross)):
                 n1,n2=self.Gs.neighbors(segs_cross[k])
                 line_k = sh.LineString([(self.Gs.pos[n1][0],
```

```
self.Gs.pos[n1][1]), (self.Gs.pos[n2][0], self.Gs.pos[n2][1])])
                p_int.append(line.intersection(line_k))

            return (p_int)
```

```
            ############## MOBILITY PART ##############
    def delta_point(self,pt,d=0.01):

        """
            L.delta_point(pt,d=0.01): calculate new point to
build a polygon for mobility zone

        """

        x,y=pt
        if (x >= 0 and y >= 0):
            x1 = x - d
            y1 = y - d
            pt1=array([x1,y1])
        if (x <  0 and y <  0):
            x1 = x + d
            y1 = y + d
            pt1=array([x1,y1])
        if (x <  0 and y >  0):
            x1 = x + d
            y1 = y - d
            pt1=array([x1,y1])
        if (x >  0 and y <  0):
            x1 = x - d
            y1 = y + d
            pt1=array([x1,y1])
        return (pt1)

    def polygon_delta(self,nnode):

        """
            L.polygon_delta(nnode): build a polygon for
            mobility permission zone for each Gt node
        """
        coords=[]
        N= self.Gt.node[nnode]['vnodes']
        for i in range(len(N)):

    coords.append(self.delta_point(self.Gs.pos[N[i]],d=0.01))

        polygon = sh.Polygon(tuple(coords))
        return (polygon)
```

```
    def new_coo_d(self,x1,x2,d=0.5):

        """
            L.new_coo_d(self,x1,x2,d=0.5): find new coordinates
with a distance of d
        """
        if (x1<x2):

            x1_sh=x1+d
            x2_sh=x2-d
        if (x1>x2):
            x1_sh=x1-d
            x2_sh=x2+d

        else:
            pass
        return (x1_sh,x2_sh)


    def poly_door(self,nnode):

        """
            L.poly_door(nnode): create a polygon with door
space
        """

        nd=self.Gr.node[nnode]['doors'][0]
        n1,n2=self.Gs.neighbors(nd)
        x1,y1=self.Gs.pos[n1]
        x2,y2=self.Gs.pos[n2]
        deltx=self.delta(x1,x2)
        delty=self.delta(y1,y2)


        if deltx>delty:
            x1,x2=self.new_coo_d(x1,x2)
            y1=y1
            y2=y2
        if deltx<delty:
            y1,y2=self.new_coo_d(y1,y2)
            x1=x1
            x2=x2
        else:
```

```
                print 'another  construction type'


            line = sh.LineString([(x1, y1), (x2, y2)])
            dilated = line.buffer(0.5)
            return (dilated)

        def moby_allowed_zone(self):

            """
                L.moby_allowed_zone(): defined mobility  allowed
Zone

                ___________ USE_______

                zone=L.moby_allowed_zone()
                Pxy = Point(0.0, 0.0): Tx/Rx coordinates
                if (zone.contains(Pxy)==True):
                          print 'In the ZONE!!!'
                else:
                              print 'Out of the ZONE'

            """
            polygon=[]
            dilated=[]
            for i in range(len(self.Gt.node)):
                polygon.append(self.polygon_delta(i))


            for j in range(len(self.Gr.node)):
                dilated.append(self.poly_door(j))



            polygons = polygon+dilated
            u = cascaded_union(polygons)
            return (u)
```