ELEC-H304 PHYSICS OF TELECOMMUNICATIONS

Ray-Tracing

Authors:
Alban Dietrich
Yassine Sahli
Geoffrey Vandeplas

Professor:
Philip De Doncker

May 9, 2019

Table of Contents

1 Intro

The goal of this project is to calculate the power received by a receiver connected to a Wifi station inside a building. The program makes it possible to visualize this captured power in a 2-dimensional representation of the building, as well as to determine the coverage area of the Wifi base station.

To this end, the interior of the building has been discretized and the program (developed in PYTHON) sums for each point the contributions of all the electromagnetic waves passing through it. Only the direct rays, transmitted or reflected by the walls were taken into account. Diffraction has been neglected.

The first part of this report will present the theoretical models used to describe the physical phenomena present in this problem. The second part will give explanations on the algorithms used. Finally, results will be presented and discussed in the last part.

2 Theoretical model

The study of the electromagnetic power distribution at each point of the floor involves only two spatial dimensions (xy plane). Polarization considerations are avoided by assuming the transmitting antennas to be lossless vertical $\frac{\lambda}{2}$ dipoles. The electric field then has only one component in z, parallel to the obstacles, with phasor $\underline{E}$. The antenna is idealized and emits waves isotropically. The transmitter is at the same height as the receiver: we only consider waves propagating in the horizontal plane.

2.1 Source used

The transmitter used in this project is a WiFi base station operating at 5 GHz (802.11ac type) and whose base power is 20dBm.

The basic quantities specific to the source are given below:

$$\textit{Antenna gain: } G_{TX} = \frac{Z_0}{\pi R_a} = \frac{\sqrt{\frac{\mu_0}{\epsilon_0}}}{\pi R_a} = 1.64 \tag{1}$$

$$\textit{Base Power: } P_{TX} = 20dBm = 0.1W \tag{2}$$

$$\textit{Wave number: } \beta = \frac{2\pi f}{c} = 104.72m^{-1} \tag{3}$$

$$\textit{Wavelength: } \lambda = \frac{2\pi}{\beta} = 0.06m \tag{4}$$

$$\begin{gathered}\textit{Equivalent height: } \vec{h_e} = -\frac{\lambda}{\pi}\vec{1_z} = -\frac{c}{\pi f}\vec{1_z} = -0.0191\vec{1_z} \\ \left\|\vec{h_e}\right\| = 0.019m\end{gathered} \tag{5}$$

With R_a the antenna resistance worth 73Ω (corresponding to the radiation resistance), Z_0 the vacuum impedance, and c the speed of light.

The limit from which waves can be considered locally planar is the far-field boundary r_{ff}, conventionally given by:

$$r_{ff} = Max\{1, 6\lambda; 5D; \frac{2D^2}{\lambda}\} = 5 \cdot D = 0.15m \tag{6}$$

where D is the maximum characteristic dimension of the emitter, here assumed equal to $\frac{\lambda}{2} = 0.03m$. Given that the characteristic size of a WiFi base station is of the order of

$15cm$, the far field approximation is made for all the buildings in which the coverage area is studied.

2.2 Direct waves

Direct rays are those going in a straight line from the transmitter to the receiver without encountering any obstacles. The principle of conservation of energy imposes that the intensity radiated by a source decreases with the inverse of the square of the distance d traveled. Indeed, the energy emitted by the source propagates in all directions, which implies that its flow through any sphere containing the source, i.e. the energy leaving it, must therefore remain constant. . Since the surface of a sphere is proportional to d^2, the intensity of the wave decreases in $\frac{1}{d^2}$ and therefore its electric field in $\frac{1}{d}$. In addition, this electric field propagating at the speed of light, there is then a propagation delay between the transmitter and the receiver. Considering at the receiver, the field emitted by the transmitter at time $tt' = t - \frac{d}{c}$, amounts, in the phasor formalism, to accumulating a phase delay $\omega\frac{d}{c} = \beta d$ and therefore to multiply the electric field phasor by $e^{-j\beta d}$. We then have :

$$\underline{E}\sqrt{60 \cdot G_{TX}(\theta_{TX}, \phi_{TX}) \cdot P_{TX}} \cdot \frac{e^{-j\beta d}}{d} \tag{7}$$

where G_{TX} is the gain of the transmitting antenna in the direction of the receiver, P_{TX} is the power of the transmitter, d the distance between the receiver and the transmitter, β the wave number. In the considered case, the transmitter gain is constant in all directions and is given by (1).

2.3 Multi-path components

These rays are those that have been transmitted or reflected by an obstacle, or a combination of both. When a wave comes into contact with a wall, the charges on the surface of the latter are excited proportionally to the incident field and in turn emit an electromagnetic wave. These waves depend on the properties of the obstacle (thickness, permittivity, etc.) as well as on the angle of incidence of the initial wave. Their gain is not homogeneous in space: two directions are preferred corresponding to reflection and transmission.

Of the energy absorbed by the wall, part is therefore reflected, another is transmitted and finally part is dissipated in the form of heat. To obtain the phasor of the waves generated by the obstacle, it suffices to correct the equation (7) by the coefficients of reflections Γ_i and transmissions T_i which model the loss of power by relation to the

incident radius for each obstacle i encountered.

$$\underline{E} = \prod_{i=1}^{N_R} \Gamma_i \cdot \prod_{j=1}^{N_T} T_i \cdot \sqrt{60 \cdot G_{TX}(\theta_{TX}, \phi_{TX}) \cdot P_{TX}} \cdot \frac{e^{-j\beta d}}{d} \tag{8}$$

where N_R and N_T are respectively the number of consecutive reflections and transmissions undergone by the wave before arriving at the calculation point and d is the total distance traveled by the wave before arriving at the receiver.

The coefficients Γ_m and T_m depend on the angle of incidence θ_i relative to the normal to the wall and are given by:

$$\Gamma_m(\theta_i) = \Gamma_\perp(\theta_i) + (1 - \Gamma_\perp^2(\theta_i)) \frac{\Gamma_\perp(\theta_i) e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}}{1 - \Gamma_\perp^2(\theta_i) e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}} \tag{9}$$

$$T_m(\theta_i) = \frac{(1 - \Gamma_\perp^2(\theta_i)) e^{-\gamma_m s}}{1 - \Gamma_\perp^2(\theta_i) e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}} \tag{10}$$

Or:

- θ_i is the angle between the trajectory of the incident ray and the normal to the wall and θ_t is the angle between the trajectory of the transmitted ray and the normal to the wall (respectively outside and inside the wall), as shown in figure 1.

- γ_m is the (complex) constant of propagation in the wall, it represents the weak losses which occur there and depends on the properties of the material:

$$\gamma_m = j\omega\sqrt{\mu_0 \hat{\epsilon}} \tag{11}$$

  with $\hat{\epsilon} = \epsilon - j\frac{\sigma}{\omega}$, ϵ being the permittivity of the wall and σ its conductivity. ω is the pulsation of the incident wave, given by $2\pi f$, f being its frequency.

- s is the distance traveled by the ray in the wall. The diagram 1 shows:

$$s = \frac{l}{\cos\theta_t} \tag{12}$$

- $\Gamma_\perp$ is the reflection coefficient when the incident ray is perpendicular to the wall. It is given by the following formula:

$$\Gamma_\perp = \frac{Z_2 \cos\theta_i - Z_1 \cos\theta_t}{Z_2 \cos\theta_i + Z_1 \cos\theta_t} \tag{13}$$

  where Z_m is the impedance of the middle m, equal to $\sqrt{\frac{\mu_0}{\hat{\epsilon}_m}}$. Here, middle 2 matches the inside of the wall while middle *1* matches the outside of the wall.

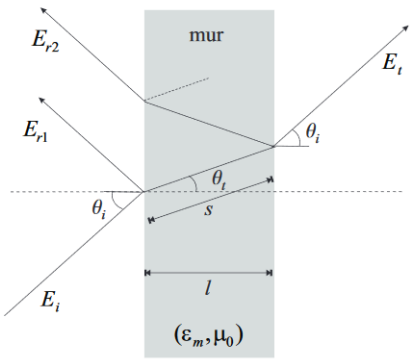

Figure 1: Transmission and reflection by a wall of thickness l

2.4 Power Received

To calculate the electromagnetic power received at a point , it is necessary to sum the contributions of the various incident electric fields associated with each possible path between the transmitter(s) and the receiver, before calculating the module square of the resulting field:

$$P_{RX} = \frac{1}{8R_a} |\sum_{n=1}^{N} \vec{h_e} \cdot \underline{\vec{E_n}}(\vec{r})|^2 \tag{14}$$

where P_{RX} is the power collected at the receiver, R_a the antenna resistance of the transmitter, N the number of rays incident at the point considered, $\vec{h_e}$ the equivalent height of the 'antenna and $\underline{\vec{E_n}}(\vec{r})$ the phasor of the incident wave.

Since the received power varies very quickly with the position, it is more practical to calculate an average power received in a local area around $\vec{r}$ of the order of a square meter. To obtain this average power, it suffices to sum the square modules of the phasors, rather than taking the square module of their sum:

$$< P_{RX} >= \frac{1}{8R_a} \sum_{n=1}^{N} |\vec{h_e} \cdot \underline{\vec{E_n}}(\vec{r})|^2 \tag{15}$$

The powers will be expressed in dBm, conventionally for their use in telecommunications.

$$P[dBm] = 10 \log \frac{P[W]}{1mW} \tag{16}$$

It is assumed that the receiver used has a sensitivity varying linearly with the bit rate outside the saturation zone between $-82dBm$ and $-51dBm$. Both being expressed in logarithmic scale, the bit rate follows the following law:

$$DB = 12.226 \cdot P_{RX}[dBm] + 1056.5 \tag{17}$$

Knowing the power received, and therefore the bit rate, makes it possible to determine the coverage area of the base station. For a received power less than $-82dBm$, the connection is impossible. Between $-82dBm$ and $-51dBm$, the connection gradually improves according to the equation (17). Once $-51dBm$ is reached, the connection is optimal and the bit rate saturates.

3 Code Description

This Ray-Tracing software was implemented in PYTHON, given that this language has the advantage of offering simplicity of writing for the design of this digital simulation. In addition, several libraries associated with PYTHON offer interesting features for it.

In order to carry out the geometric construction calculations, the simulation is done on the real Cartesian plane, where each walls and emitters have coordinates (x,y).

3.1 Map Design

To simplify the calculations, the map was discretized into cells of modifiable size, whose position is given by Cartesian coordinates. This map is created with an Excel file *Map2.xlsx* (figure 2). Each box colored blue represents a wall, the red boxes represent their ends and the green boxes represent the transmitters. The values noted in the walls represent their thickness and type, and those noted in the emitters represent their phase. This file allows you to modify the card where the received power is studied easily.

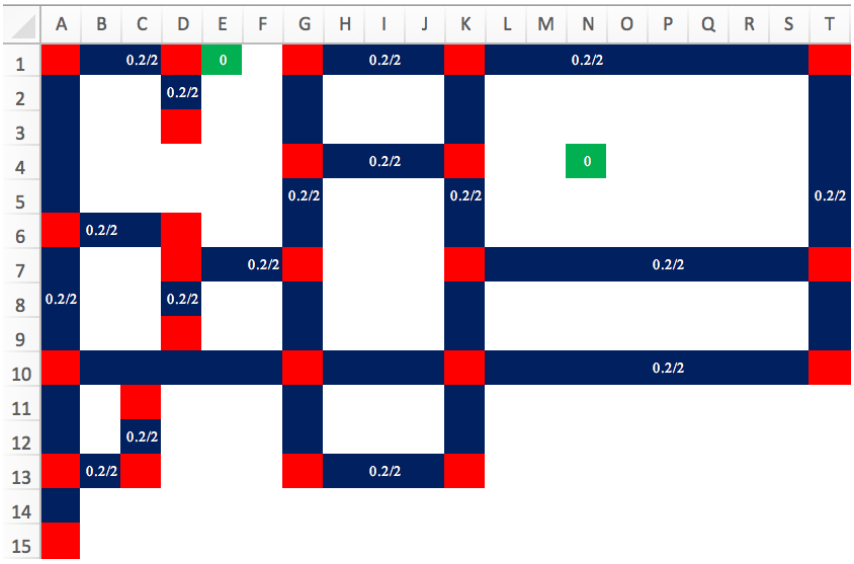

Figure 2: Excel file *Map2.xlsx*: discretized map

The three types of walls considered are brick, concrete, and partition whose useful characteristics for the calculations of reflection and transmission coefficients are given below.

Type	Relative Permittivity ϵ_r	Conductivity σ
Brick	4.6	0.02
Concrete	5	0.014
Partition	2.25	0.04

The *EditeurMap.py* file uses the *openpyxl* library to read the Excel file and create from it the text files *walls.txt* and *map.txt* . These contain respectively the information relating to the walls (position, characteristics, etc.) and those relating to the card (transmitter(s), dimensions, number of cells).

These text files are then used in the main code to produce the Ray-Tracing type simulation .

3.2 Power Calculation

The program calculates the power captured on each cell by summing the contributions of the various incident fields on it. It determines the average power in a local area using the formula 15, summing the powers contributed by each ray individually. It can also calculate the power at the center of each cell, with the formula 14 which takes into account the phase shifts between the phasors of the incident rays. The program determines the phasors associated with several different types of ray:

Direct beam

The phasors of these rays are determined using the formula (7). The distance traveled is given by the coordinates of the transmitter and the receiver.

Transmitted Ray

The program goes through all the walls on the map and, if the ray intersects one of them before arriving at the target cell, it calculates the angle of incidence of the ray on it. The corresponding transmission coefficient is then determined using the equation (10). The latter, with the coordinates of the transmitter and that of the receiver, is used for the calculation of the phasor via the equation (8).

Reflected Ray

To calculate the contributions of the reflected rays, the *image method* was used. The latter makes it possible to find the angle of incidence θ_i as well as the distance traveled d by the ray in order to calculate the electric field which constitutes it.

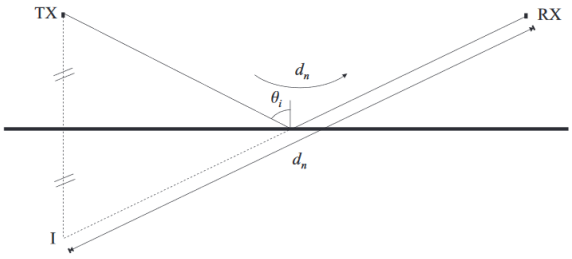

Figure 3: Illustration of image method for simple reflection: TX is transmitter, RX is receiver, I is image transmitter

This method consists in considering that the ray is emitted by the image of the original emitter, located symmetrically with respect to the reflecting wall. The angle of incidence as well as the total distance traveled are kept as shown in the figure 3: they are calculated using the I image transmitter in the same way as for transmission, the coefficient of reflection given by the formula (9).

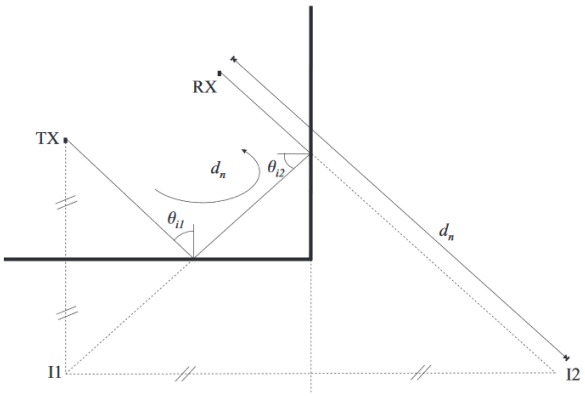

Figure 4: Illustration of the image method for a double reflection: TX is the transmitter, RX the receiver, I1 the first image transmitter, I2 the second image transmitter

In the case of multiple reflections (figure 4), the contribution of the ray is obtained by using the method of images in an iterative way: for each obstacle, a new image emitter, symmetrical to the initial emitter with respect to to the considered obstacle, is created. These will consist of the "sources" of the rays having been reflected once.

Then, for each of the image transmitters, the process begins again: new image transmitters are created in the same way for each of the previous images. These are the "sources" of the rays having undergone two reflections.

The image transmitters contain in memory the various walls by which the symmetries have been carried out, in the order of operations. For each of these image transmitters, the point of intersection between the transmitter-receiver segment and the last obstacle kept in memory is determined. If this point does not exist, this issuer is no longer taken into account in the calculation. The angle of incidence on the obstacle is then calculated as well as the corresponding reflection coefficient. If there are other intersections with obstacles located between the reflecting obstacle and the receiver, the transmission by these is taken into account and the calculations of the transmission coefficients are carried out. The process is then repeated with the transmitter symmetrical to the previous one. The process stops when reflection with the initial real transmitter is confirmed. The total distance traveled is determined using the last image transmitter. The electric field is finally given by (8).

Using a recursive function in the program increases the simulation loading time. For this reason, an iterative method has been preferred to the recursive function: the code repeats the process described above for as many reflections considered requested. To simplify the calculations, this number has been increased to 3, the following reflections being neglected.

3.3 Simulation display

The simulation is displayed by the program using the *matplotlib* library. The functions of this library allow the display of mathematical graphs. They receive a list containing all the cells on the map as well as the power assigned to them, and represent them in a color-coded figure. It is possible to display the calculated received power, the coverage area, or the bit rate as desired.

3.4 Optimization

The program is able, from a given map, to calculate a simulation for a multitude of transmitter locations. For each of these simulations, it counts the number of cells where the connection is optimal (power captured above $-51dBm$) as well as the number of cells where it is good (power captured between $-82dBm$ and $-51dBm$). It then returns the simulation that provided the best results, thus giving the optimal position of the transmitter.

4 Elementary cases

A validation of the code on elementary cases was carried out by comparison with analytical results calculated below.

4.1 Propagation in the void

Analytical result

For a wave traveling a distance $d = 5m$, the phasor of the electric field is given by (7):

$$\underline{E} = 0.6274 \ e^{-523.6j} \ \frac{N}{C} \tag{18}$$

Knowing that this electric field only has one component in $\vec{1_z}$, it is possible to determine the theoretical average power by (15):

$$< P_{RX} >= \frac{1}{584} | -0.019 \ \vec{1_z} \cdot 0.6274 \, e^{-523.6j} \vec{1_z}|^2 = 2.433 \ 10^{-7} W \tag{19}$$

Passing in dBm, we have:

$$< P_{RX} >= -36.14 \ dBm \tag{20}$$

Numerical Result

By placing only the transmitter in the center of an empty card, the simulation of the propagation in vacuum of the waves emitted by the source is observed. As predicted by theory, this propagation is isotropic.

At 5 meters from the transmitter, the average power received in a local area is:

$$< P_{RX} >_{num}= -36.18 \ dBm \tag{21}$$

This value corresponds well with that found analytically.

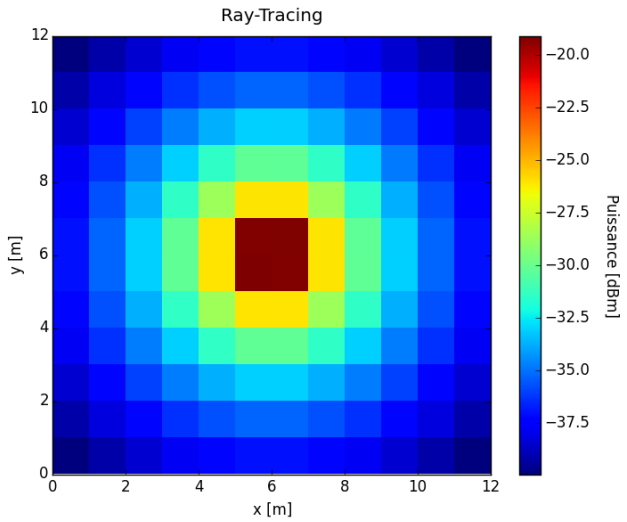

Figure 5: Simulation for a transmitter placed in a vacuum

4.2 Simple transmission

We consider a ray going from the source to an emitter located 5 meters further. A concrete wall 10cm thick is placed perpendicular to the path of the ray between the source and the transmitter.

Analytical result

To obtain the phasor of the electric field associated with the ray, we use the formula (8) where the transmission coefficient T is given by (10). We start by calculating the resistances of the media inside and outside the wall:

$$Z_1 = Z_{empty} = \sqrt{\frac{\mu_0}{\epsilon_0}} = 376.73\Omega \tag{22}$$

$$Z_2 = Z_{béton} = \sqrt{\frac{\mu_0}{\epsilon_{béton} - j\frac{\sigma}{\omega}}} = 168.48 + 0.85j \ \ \Omega \tag{23}$$

Since θ_i is zero, Snell's second law tells us that θ_t is too:

$$\sin\theta_t = \sqrt{\frac{\epsilon_0}{\epsilon_{béton}}}\sin\theta_i \longleftrightarrow \theta_t = 0° \tag{24}$$

This allows you to find the reflection coefficient $\Gamma_\perp$ using (13):

$$\Gamma_\perp = -0.382 + 0.002d \tag{25}$$

The propagation constant in concrete is given by:

$$\gamma_m = j\omega\sqrt{\mu_0(\epsilon_{béton} - j\frac{\sigma}{\omega})} = 1.18 + 234.32d \tag{26}$$

And knowing that the distance traveled in the wall s is equal to its thickness, the formula (10) gives for the transmission coefficient:

$$T(0) = -0.067 + 0.68d \tag{27}$$

Knowing this coefficient makes it possible to calculate the phasor of the electric field associated with the ray having traveled 5 meters and having been transmitted by the concrete wall using the formula (8):

$$\underline{E} = T \cdot 0.6274 \ e^{-523.6j} = 0.39 - 0.177j \ \frac{N}{VS} \tag{28}$$

We obtain the average power:

$$< P_{RX} >= 1.134 \cdot 10^{-7} W \tag{29}$$

Passing in dBm, we have:

$$< P_{RX} >= -39.45 \ dBm \tag{30}$$

Numerical result

We place a transmitter in the center of a room separated in two by a 10cm thick concrete wall. We observe that the power captured is less on the right side of the wall. It therefore absorbs energy, in accordance with what the theory predicts. 5 meters from the source, on the other side of the obstacle, we have:

$$< P_{RX} >_{num}= -39.50 \ dBm \tag{31}$$

This value is consistent with what the theory predicts.

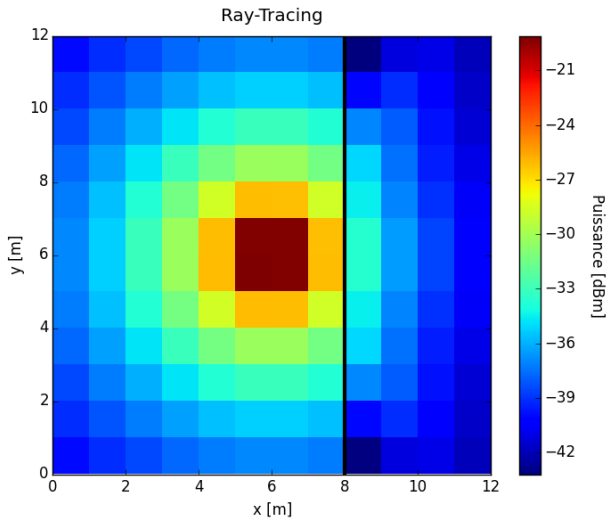

Figure 6: Simulation for a transmitter placed in front of a wall

4.3 Simple Reflection

The TX transmitter and the RX receiver are both placed 2.5 meters above a 0.1 meter thick wall and are spaced 5 meters as shown in figure 7. The angle of incidence of the ray on the wall is therefore 45°.

Analytical result

The impedance and propagation constant of the concrete wall are given by (23). By Snell's law, we determine the transmission angle $\theta_t = 18.435°$. The reflection coefficient Γ is given by (13) and (9):

$$\Gamma_\perp = -0.5 + 0.0019j \tag{32}$$

$$\Gamma = -0.388 - 0.254j \tag{33}$$

The total distance traveled by the ray is $d = 5\sqrt{2}$ m, we then have :

$$\underline{E} = 3.1368 \cdot \Gamma \cdot \frac{e^{-j\beta d}}{d} = (-0.356346 - 0.064593j)\frac{N}{VS} \tag{34}$$

$$< P_{RX} >= 8.1073 \cdot 10^{-8}\ W \Longleftrightarrow < P_{RX} >= -40.911\ dBm \tag{35}$$

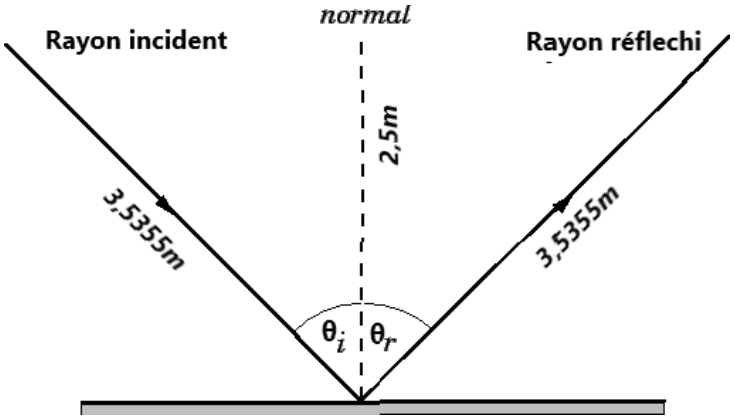

Figure 7: Simple reflection of an incident ray at 45°

Numerical result

The previously explained arrangement is reproduced digitally to perform the calculation of the power coming from a reflected ray. By isolating the contribution of this ray, the program obtains:

$$< P_{RX} >_{num} = -41.59 \ dBm \tag{36}$$

What corresponds to the analytical answer by knowing that for the analytical method several rounded ones were carried out. The figure 8 illustrates the power distribution in this same layout considering all possible contributions. It can be seen that the reflection here has little influence on the average power perceived by the receiver, as one might expect by comparing the power of the direct ray, calculated in the study of vacuum propagation in (21), and that of the reflected ray.

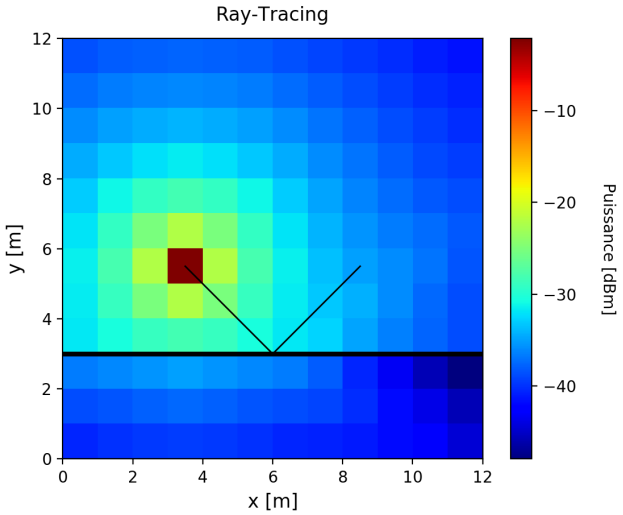

Figure 8: Reflected ray illustration

4.4 Double or more reflection

Since for multiple reflections, the method is iterative, the calculations carried out are the same as those carried out previously. After 2 reflections, we can observe below that the power contribution is negligible.

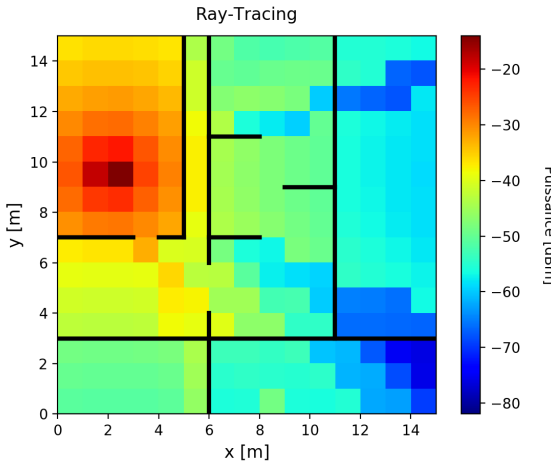

Figure 9: After 0 thinking

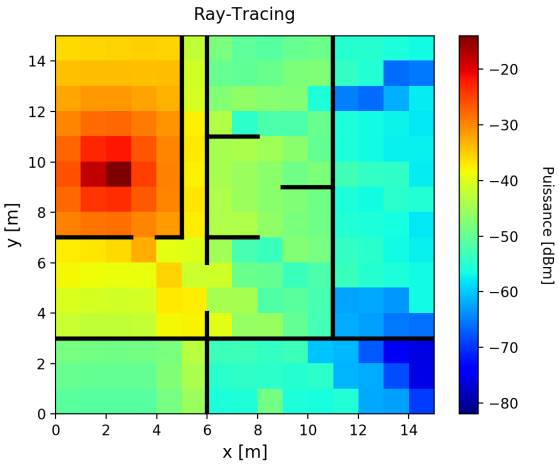

Figure 10: After 1 thought

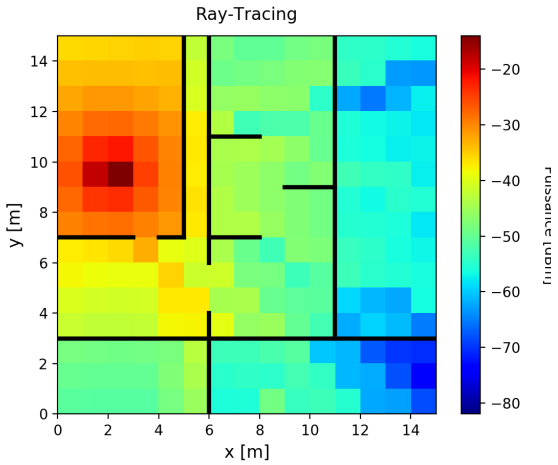

Figure 11: After 2 thoughts

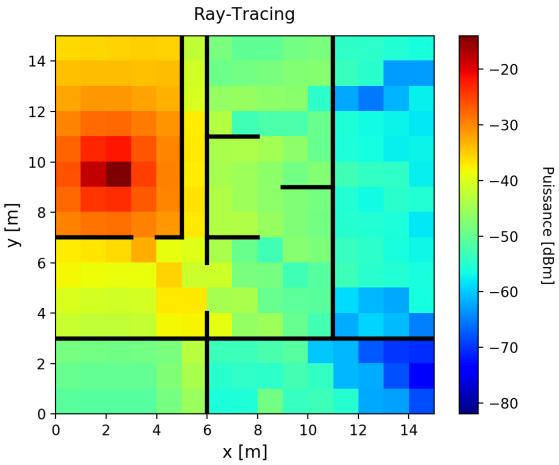

Figure 12: After 3 thoughts

5 Apps

In this section of the report, several examples of application of the program are presented.

5.1 House floor

In this example, we are interested in the first floor of a large house of $20m \times 20m$ of average complexity. The simulation was performed three times:

- With brick walls $20cm$ thick.
- With concrete walls $20cm$ thick.
- With partition walls $20cm$ thick.

In each of these simulations, the transmitter was placed at the coordinate (5,16).

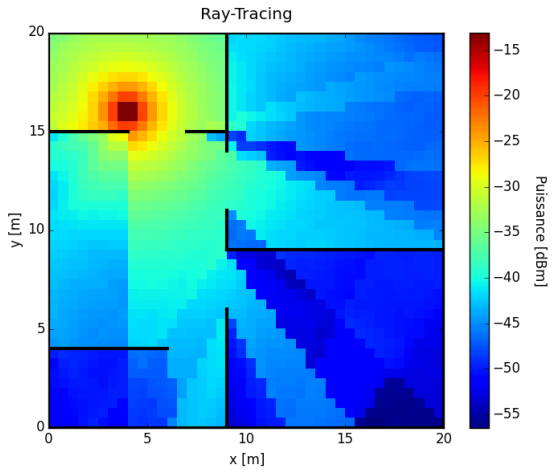

Figure 13: Captured Power *(brick walls)*

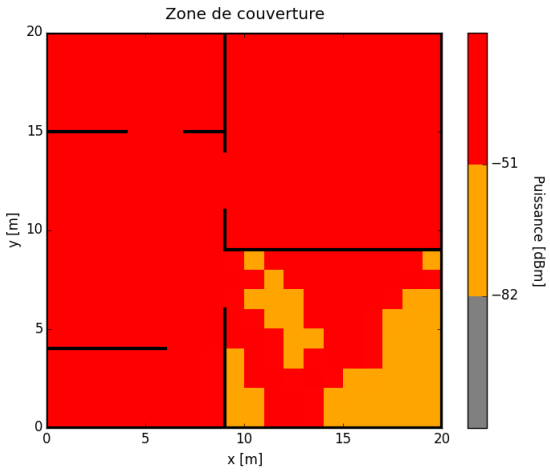

Figure 14: Coverage area *(brick walls)*

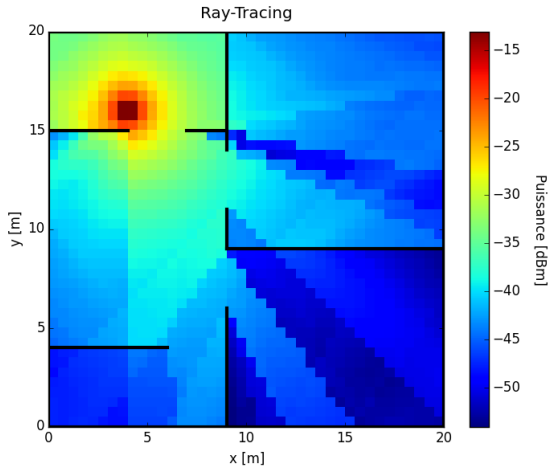

Figure 15: Captured Power *(concrete walls)*

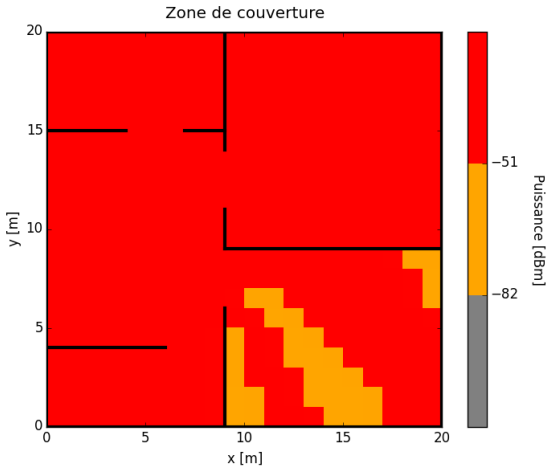

Figure 16: Coverage area *(concrete walls)*

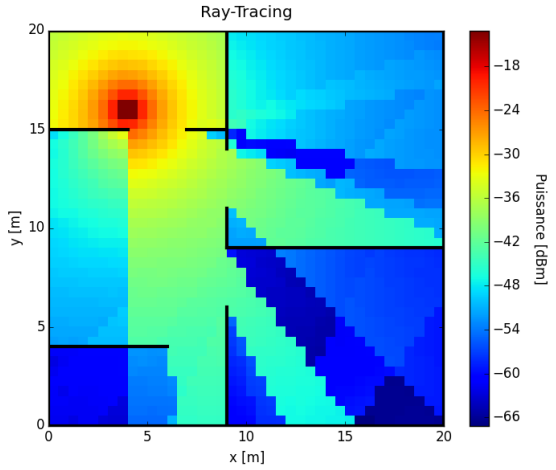

Figure 17: Captured power *(bulkhead walls)*

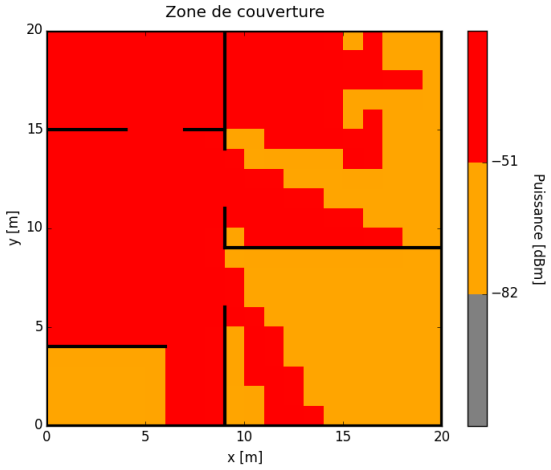

Figure 18: Coverage area *(partition walls)*

It is observed that for each type of wall, the entire floor is in the coverage area of the source, ie the power captured exceeds $-82dBm$. However, this is not done the same way for all wall types:

- Concrete walls don't reflect waves as much as the other two. They transmit them more than the others and consequently have the lowest minimum power among the different wall types. The coverage area in the concrete floor is also the one in which the connection is most often optimal, ie it exceeds the saturation value $-51dBm$.
- The partition walls do not transmit waves as much as the other two, and this affects the coverage area on this floor: it is the one where the connection is less

often optimal. The floor in the partition is also the one where the minimum power captured is the lowest.

- Brick walls are between the two previous ones in terms of minimum power input and optimal connection.

We therefore conclude, as predicted by theory, that the power captured by the receiver varies when the material used for the walls changes. The coverage area can therefore be a criterion to take into account when constructing a wall in a building, and the Ray-Tracing program can help determine the type of wall to use.

5.2 Wifi Connection Optimization

In this section, the code optimization function is used.

House floor

First of all, consider the case of the house floor presented in the previous point. For walls made of brick, 20cm thick, the result is as follows:

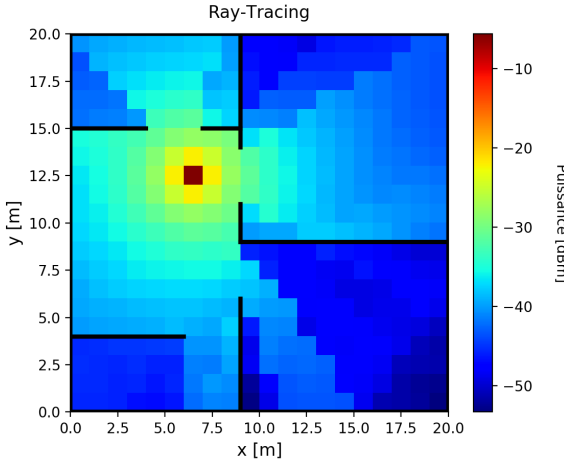

Figure 19: Captured power *(Optimal case)*

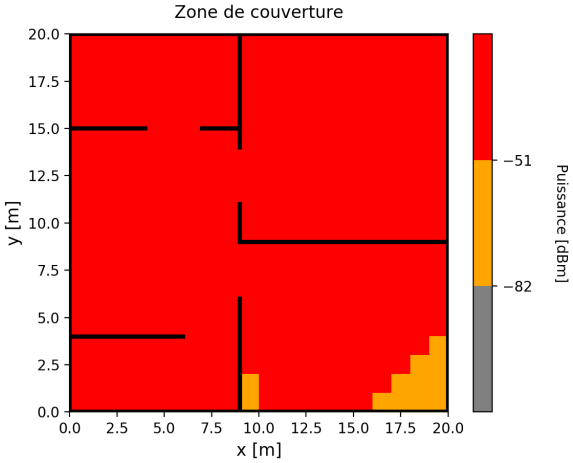

Figure 20: Coverage Area *(Optimal Case)*

We observe that the transmitter has been placed at (6.5;12.5) by the program. The zone in which the connection is optimal is, as expected, larger than that seen in the previous example, in figure 14. In the 21 figure below, the bitrate is shown in the optimal case. By comparing it to the coverage area above (figure 20), we can see that the good and optimal connection areas correspond well.

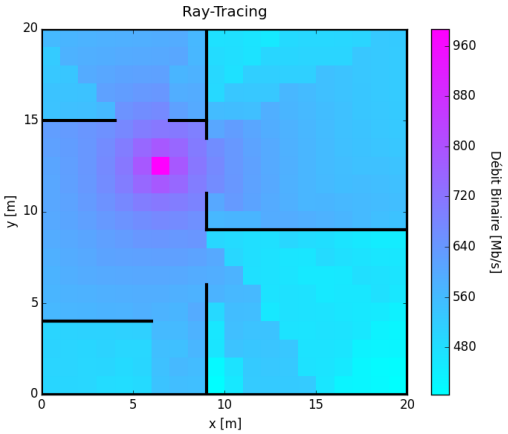

Figure 21: Bitrate *(Optimal case)*

University building

We propose to use the program in order to find the most optimal location of a transmitter on the floor of a university building. On the figure 22 is the plan of the ground floor of a university building proposed during a project competition at the Université Libre de Bruxelles.

To create the simulation map, we based ourselves on the blue colored part of this plan. All walls on the map are assumed to be $30cm$ thick concrete.

The result of the search for the optimal position of the transmitter carried out by the program is given in figure 23. To have a maximum surface in the coverage area, the WiFi source must therefore be located at coordinates (25.5; 31).

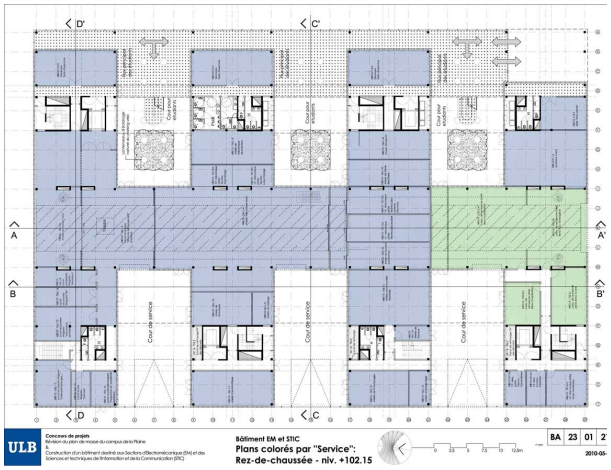

Figure 22: Shot of building interior

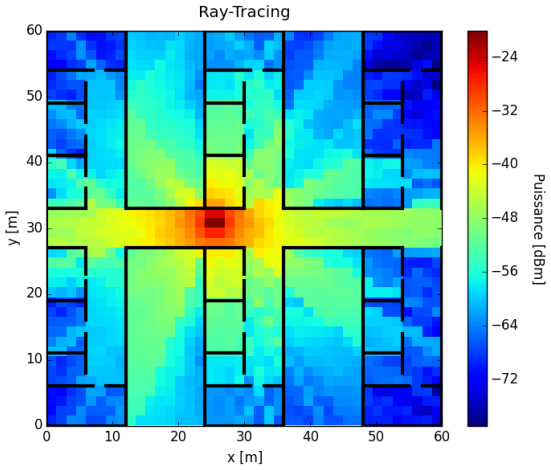

Figure 23: Captured power (Optimal case)

Below, the coverage area as well as the bit rate in the building are displayed. We observe by the figure 24 that all the parts have a good connection. The same goes for the areas outside the building: the courses in which the students can go also need a good connection, which is the case here.

It can also be seen in the two figures that only two rooms, as well as the hallway, enjoy an optimal connection. If a certain room has a specific need for an optimal connection, it will therefore be necessary either to move the transmitter or to add a source in order to maintain a good connection in the whole area of $60m^2$.

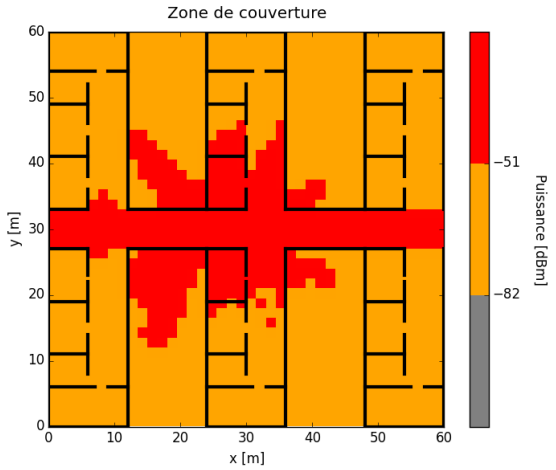

Figure 24: Coverage Area (Best Case)

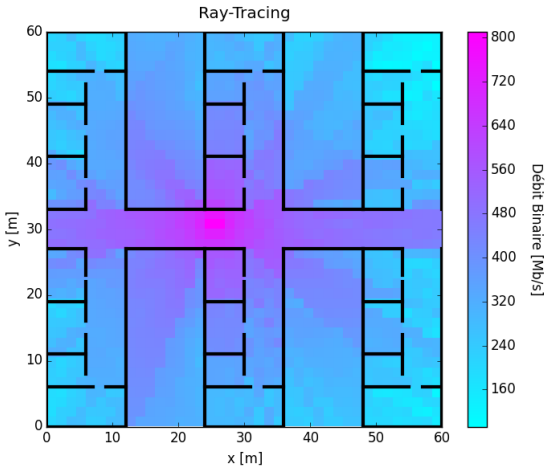

Figure 25: Bitrate (Optimal Case)

5.3 Interference with a reflected wave

We consider here a basic example, that of a transmitter placed in a vacuum facing a partition wall.

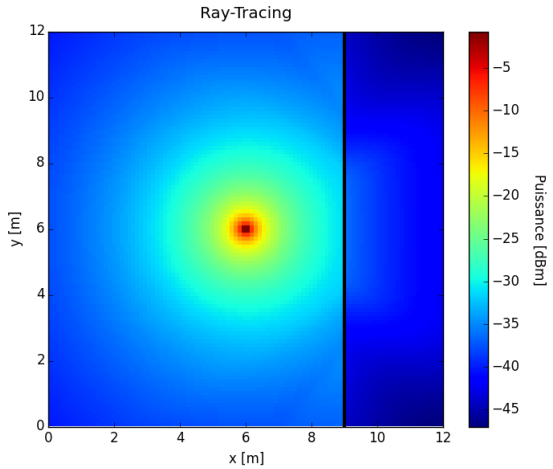

Figure 26: Power calculated by averaging over a local area

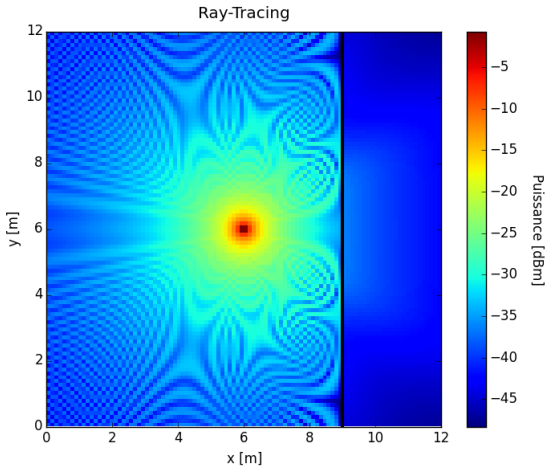

Figure 27: Power calculated taking interference into account

First, we observe as expected that the resulting power varies very quickly in space, the use of an average in a local area is therefore very necessary because if the receiver has a characteristic size of about $50cm$, as in in the case of a computer for example, knowing a power whose value varies every centimeter is not practical.

Second, we can see the interference between the direct rays and those reflected from

the wall. The latter having traveled a longer distance, they are in phase shifts. The variations in power due to these phase shifts occur at most, every $10cm$. This is still too fast to be practical, given the size of the receiving devices. Despite these interferences, the average power is indeed correct to represent the non-local power: the scale is identical and the zones of good reception or not are the same.

5.4 Interference between two sources

The implementation of a second transmitter, contrary to what one might think on board, can have regrettable consequences on the power captured by the receiver. Indeed, the two emitters emitting at the same frequency, there are regions of space where the field coming from an emitter constantly opposes that coming from the second. In the case where these are placed fairly close to each other, the calculation of the average power as carried out previously does not make much sense since the local power varies very little in space. The power in the center of the cell makes it possible to clearly observe the interference fringes.

Below, we place the emitters on the $y = 5$ axis at a distance $\frac{\lambda}{2}$ from each other. At zero phase shift between the two transmitters, we expect to have a maximum power on the axis of symmetry between the two receivers, where the two fields add up perfectly in phase. The results obtained by simulation for different phase shifts between the emitters are shown in the figure 28.

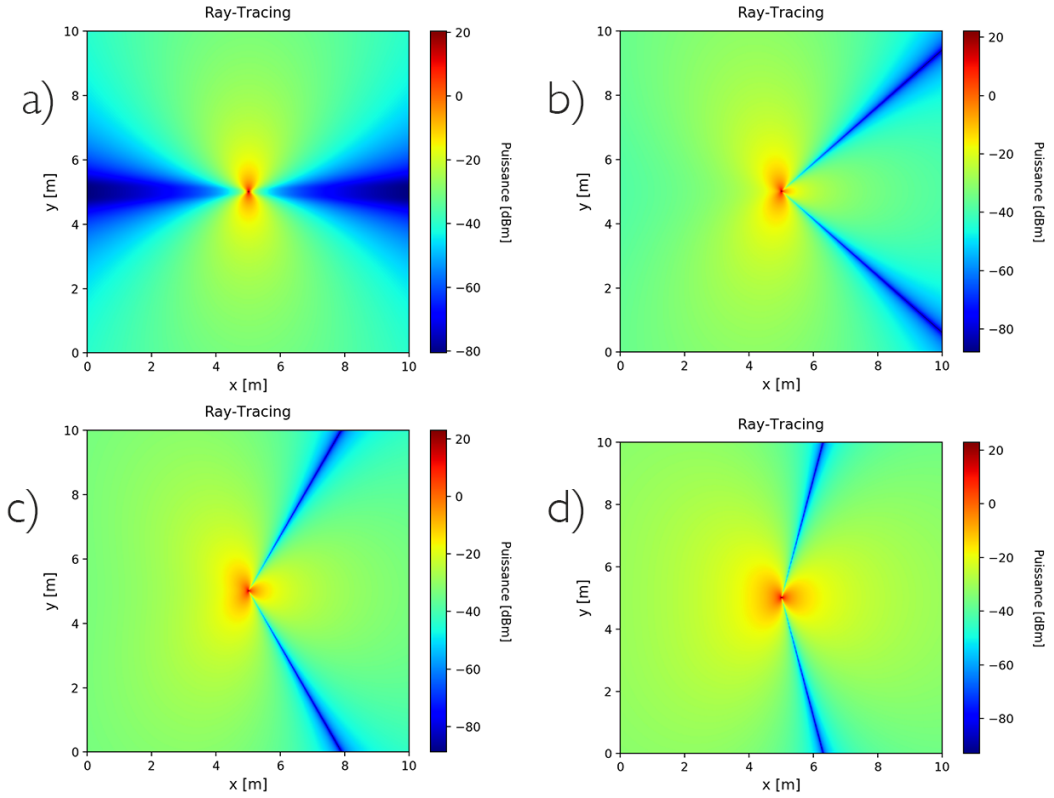

Figure 28: Power emitted by two emitters, $\frac{\lambda}{2}$ apart and phase shifted respectively by a) 0°, b) 45°, c) 90°and d) 135°

We can notice that by arranging the transmitters in a particular way and by shifting them, we can play on the directivity of the signal. There are indeed directions for which the gain of the system is maximum and others where it is almost zero. It is therefore possible, using two sources, to isolate areas in which the Wifi connection would not be established, and the Ray-Tracing program makes it possible to predict these areas. This can be useful in different cases, such as having a Wifi connection in a house without it reaching the children's room.

6 Conclusion

Ray-Tracing program presented here makes it possible to calculate and visualize, in accordance with theory, the amplitude of the electric field, the power received and the bit rate in each cell for a multitude of architectures. The program can, in general, display interference figures for a wide frequency spectrum other than the 5 GHz of the Wifi network considered and also obtain the location of the transmitter which maximizes the coverage area on a given map. This is very useful for anyone looking for an optimal connection or minimal installation costs (minimizing the number of transmitters).

7 Appendix

CalculationPower.py

```
"""

FILE THAT TAKES CARE OF ALL THE CALCULATIONS TO DETERMINE THE POWER

"""

import cmath as cm
from SpatialOperations import *
from CreateMapElements import wallListe, listeCellules, nbreCellX,
    nbreCellY
import matplotlib.pyplot as p


def coefReflexPerp(resMur, aI, aT):
"""
:param resWall: Resistance of the wall
:param aI: Angle of incidence
Transmission angle
:return: Perpendicular reflection coefficient
"""
z1 = m.sqrt(muO/epsO)
z2 = resWall
return (z2*m.cos(aI)-z1*m.cos(aT))/(z2*m.cos(aI)+z1*m.cos(aT))


def coefficientTransmitionCalculation(wall, point1, point2):
"""
:param wall: wall through which the ray passes
:param point1: point 1 of the segment intersecting the wall
:param point2: point 2 of the segment intersecting the wall
:return: Returns the transmission coefficient
"""
aI = calculateAngleIncident(point1, point2, wall)
aT = calculateAngleTransmit(aI, epsO, wall.eps)
epsTilde = complex(wall.eps, -wall.sigma/puls)
wall.setResistance(epsTilde)
refCoef = coefReflexPerp(wall.resistance, aI, aT)
```

```
beta = pulse/c
s = wall.width/m.cos(aT)
# gamma = A + jB
A = puls * m.sqrt(muO * wall.eps / 2) * (m.sqrt(1 + (wall.sigma / (
    puls * wall.eps)) ** 2) - 1) ** 0.5
B = puls * m.sqrt(muO * wall.eps / 2) * (m.sqrt(1 + (wall.sigma / (
    puls * wall.eps)) ** 2) + 1) ** 0.5
gamma = complex(A,B)
return ((1-refCoef**2)*cm.exp(-gamma*s))/(1-refCoef**2*cm.exp(-2*
    gamma*s)*cm.exp(complex(0, 2*beta*s*m.sin(aT)*m.sin(aI))))

def coefficientReflexionCalcul(wall, point1, point2):
"""
:param wall: wall through which the ray passes
:param point1: point 1 of the segment intersecting the wall
:param point2: point 2 of the segment intersecting the wall
:return: Returns the reflection coefficient
"""
# Calculation of the reflection coefficient
aI = computeAngleIncident(point1, point2, wall)
aT = calculateAngleTransmit(aI, epsO, wall.eps)
epsTilde = complex(wall.eps, -wall.sigma / pulse)
wall.setResistance(epsTilde)
refCoef = coefReflexPerp(wall.resistance, aI, aT)
beta = pulse/c
s = wall.width / m.cos(aT)
#gamma = A + jB
A = puls * m.sqrt(muO * wall.eps / 2) * (m.sqrt(1 + (wall.sigma / (
    puls * wall.eps)) ** 2) - 1) ** 0.5
B = puls * m.sqrt(muO * wall.eps / 2) * (m.sqrt(1 + (wall.sigma / (
    puls * wall.eps)) ** 2) + 1) ** 0.5
gamma = complex(A,B)

num = (1-refCoef**2)*refCoef*cm.exp(-2*gamma*s)*cm.exp(complex(0.2*
    beta*s*m.sin(aT)*m.sin( have)))
den = 1-refCoef**2*cm.exp(complex(0.2*beta*s*cm.sin(aT)*cm.sin(aI)))
return refCoef+num/ den


def calculEField(coefT, dist):
"""
```

```
:param coefT: total coefficient (multiplication of the different
    coefficients if there has been reflection or transmission)
:param dist: distance from transmitter to receiver
:return: returns the E field multiplied by the height of the antenna
"""
# Electric field calculation
he = -c / f / m.pi
beta = pulse / c
if (dist == 0):
distance = 0.15
E = coefT * m.sqrt(60 * GTX * PTX) * cm.exp(complex(0, -beta * dist))
    / dist
return he* E

def powerLocalArea(coefT, dist):
"""
:param coefT: total coefficient (multiplication of the different
    coefficients if there has been reflection or transmission)
:param dist: distance from transmitter to receiver
:return: returns the squared module of the field multiplied by
    different factors (local area approximation)
"""
# Calculation of power by approximating local areas
he = -c/f/m.pi
beta = puls / c
if(dist == 0):
distance = 0.15
E = coefT * m.sqrt(60 * GTX * PTX) * cm.exp(complex(0, -beta * dist))
    / dist
return 1/(8*Ra)*abs((he*E))**2

def powerTotal (transmitter, local zone):
"""
:param emitter: variable containing the emitter
:param zoneLocale: boolean that tells us if we are using the formula
    in the local zone or not
:return: no return. Stores the calculated power in the power
    attribute of the cell
"""
for cell in listCells:
coefT = 1
```

```
[x, y] = cell. getPos()
coefT *=ReflTransFunct(sender, x, y)
dist = distCalc(transmitter.getPos(), [x,y])
if(localZone):
then =LocalZonepower(coefT, dist)
cell.setPower(then)
else:
fieldE = calculationFieldE(coefT, dist)*cm.exp(complex(0,emitter.
    phase))
cell.setField(fieldE)


def conversionBinaryDebit(power):
"""
:power param: power
:return: returns the bit rate conversion of the power in Mb/s
"""
return 379/31*power+32752/ 31


def ReflTransFunct(emitter, x, y):
"""
:transmitter param: transmitter
:param x: position in x of the cell
:param y: position in y of the cell
:return: returns the global coefficient
"""
global showRay #Boolean indicating whether we want to display the
    rays or not
interBis = 0
interBisBis = 0
interMemory = list()
interMemoryBis = list()
xlist = list()
ylist = list()
coefTBisBis = 1
nR = len(transmitter.getListWalls())
wall = 1
oldWall = 1
[x2, y2] = emitter.getPos()[:]
[x1, y1] = [x, y][:]
```

```
listWalls = emitter.getListWalls()

while(nR>0):
if(len(listWalls)!=1):
oldWall = listWalls[nR - 2]
wall = listWalls[nR - 1]
inter = intersectionCalculation(wall.posStart, wall.posEnd, [x1, y1],
    [x2, y2])
if (inter != 0):
[xi, yi] = inter[:]

coefTBisBis *= coefficientReflexionCalcul(wall, [x1, y1], [xi, yi])

#Show Rays
if([x, y] == [8.5, 5.5] and len(sender.getListWalls())==2 and
   showRadius):
xlist.extend([x1, xi])
ylist.extend([y1, yi])

for wall in wallList:
if(interBis!=0):
interMemory = interBis[:]
interBis = intersectionCalculation(wall.posStart, wall.posEnd, [x1,
   y1], [xi, yi])
if (wall != wall and wall!= oldMur and interBis != 0 and interBis!=
   interMemory):
coefTBisBis *= coefficientTransmitionCalcul(wall, [x1, y1], interBis)
[x1, y1] = inter[:]
[x2, y2] = calculateSymmetry(wall, [x2, y2])[:]

else:
coefTBisBis = 0
break
nR-=1

if(nR==0):
for wall in wallList:
if(interBisBis!=0):
interMemoryBis = interBisBis[:]
interBisBis = intersectionCalculation(wall.posStart, wall.posEnd, [x1
   , y1], [x2, y2])
```

```
if (wall != wall and interBisBis != 0 and interBisBis!=interMemoryBis
    ):
coefTBisBis *= coefficientTransmitionCalcul(wall, [x1, y1],
    interBisBis)

# Show rays
if(coefTBisBis!=0 and [x, y] == [8.5, 5.5] and len(emitter.
    getListeMurs())==2 and showRayon):
xlist.extend([x1, x2])
ylist.extend([y1, y2])
p.plot(xlist, ylist, color='k', linewidth=1)
showRadius = False

return coefTBisBis

def makeGraphList(CellList, boolDebitBin, Localarea):
"""
:param listCells: a list containing all cells
:param boolDebitBin: boolean indicating whether we calculate the bit
    rate or the power in dBm
:param zoneLocale: boolean indicating whether we use the power
    formula in local zone or not
:return: returns the list which contains all the data to display the
    map
"""
bigList = list()
for j in range(numCellY):
smallList = list()
for i in range(numCellX):
cell = listCells[i + j*numCellX]
if (not localzone):
then = 1 / (8 * Ra) * abs(cell.fieldE) ** 2
cell.setPower(then)
power = 10 * m.log10(cell.power * 1000)
if(boolDebitBin):
smallList.append(conversionBinaryDebit(power))
else:
smallList.append(power)
bigList.append(smallList)
return bigList
```

```
# Definition of constants
muO = 4*m.ft*10**(-7); epsO = 8.854*10**(-12)

showRayon = False # Display of the reflected ray yes or no

Ra=73; GTX = m.sqrt(muO/epsO)/(m.pi*Ra); PTX=0.1; f = 5*10**(9); c
    =299792458

pulse = 2 * m.pi * f
```

Cell.py

```
"""
DEFINITION OF THE CELL OBJECT
"""

from Object import Object as obj

class Cell(obj):

def __init__(self, position):
"""
:param position: initialize the position
"""
obj.__init__(self, position)
self.power = 0
self.champE = 0

def setCellSize(self, cellSize):
"""
:param sizeCell: the size of the cell
:return: no return
"""
self.cellsize = cellsize

def setPower(self, power):
"""
:power param: power which is added to the total power of the cell
:return: no return
"""
self.power += power

def setField(self, fieldE):
"""
:param fieldE: field E which is added to the total field E of the
    cell
:return: no return
"""
self.fieldE += fieldE
```

```
def resetPower(self, power):
"""
:power param: resets the cell power to the power value
:return: no return
"""
self.power = power
```

CreateMapElements.py

```
"""

FILE WHICH CREATES THE ELEMENTS PRESENT ON THE MAP

"""

import Cell as cel
import Wall as w
import Emitter as em
from OperationsSpaciales import calculSymetry
import math as m
import MapEditor as edM

def createCells():
"""
:return: returns list containing all cells
"""
listCells = list()
for i in range(nbreCellY):
for j in range(numCellX):
# Centered in the center of the cell
iReal = i / nbreCellY * mapSizeY + cellSize / 2;
jReal = j / numberCellX * mapSizeX + cellSize / 2;
cell = cell.Cell([jReal, iReal])
listCells.append(cell)
return listCells



def creationListeEmitterImg(emitter, wallListeBis):
"""
:param sender: sender
:param wallListeBis: the list of walls by which the transmitter will
   be reflected
:return: returns the list of all image transmitters
"""
wallListBis = wallListBis[:]
numWall = len(emitter.getListWalls())

if(numWall!=0):
```

```
wallListBis.remove(emitter.getListWalls()[nberWall-1])
for wall in wallListBis:
newEmet = em.Emitter(calculationSymmetry(wall, emitter.getPos()))
for wall in emitter.getListWalls():
newEmit.addWallList(wall)
newEmit.addWallList(wall)
if(numWall==0):
listEmetImage.append(newEmet) # To change put just emitter there we
    do it to observe the pos
else:
if (worm(emitter, emitter.getListWalls()[numWall - 1], wall)):
listEmetImage.append(newEmet) #To change put just emitter there we do
     it to observe the pos


def ver(emitterImg, oldMur, newMur):
"""
:param emitterImg: image emitter
:param oldMur: old wall by which the emitter was reflected
:param newMur: new wall by which the emitter will be reflected
:return: returns whether or not this sender image is relevant. If yes
    , it is created. If not, it is destroyed.
"""
if(oldWall.posStart[1]==oldWall.posEnd[1] and newWall.posStart[1]==
    newWall.posEnd[1]):
dOld = emitterImg.getPos()[1]-oldMur.posStart[1]
dN = newWall.posStart[1]-oldWall.posStart[1]
if(dOld*dN>=0):
return False
if (oldWall.posStart[0] == oldWall.posEnd[0] and newWall.posStart[0]
    == newWall.posEnd[0]):
dOld = emitterImg.getPos()[0] - oldMur.posStart[0]
dN = newMur.posStart[0] - oldMur.posStart[0]
if (dOld * dN>=0):
return False

return True

# Boolean saying if we go into optimization mode or not
optimizationMode = False
listOpti = list()
```

```
# Call the map editor to create the map
edM.editMapBis("MapEdit.xlsx")

# Opening of the map file with for info the size of the map and the
    discretization
map_file = open("map.txt", "r")

mapEmetCoord = map_file.read().strip().split("\n")
emetCoord = mapEmetCoord[0].split("/")
emitCoord.remove("")

mapCoord = mapEmitCoord[1].split(" ")
emitDepart = list()

for elem in emetCoord:
# Creation of base transmitter(s)
elemAttribute = elem.split(" ")
emitter = em.Emitter([float(elemAttribut[0]), float(elemAttribut[1])
    ])
transmitter.setPhase(float(elemAttribut[2])*m.pi/180)
emitDepart.append(emitter)


mapSizeX = float(mapCoord[0])
mapSizeY = float(mapCoord[1])

discretization = float(mapCoord[2])

sizeCell = mapSizeX/discretization

nbreCellX = int(discretization)
numCellY = int(mapSizeY/sizeCell)

map_file.close()

# Definition of reflection number
NBRE_MAX_REFL = 3

# Creation of lists that contain all the transmitters created
listEmitImage = list()
```

```
listEmitImageTot = list()

# List containing all the data
listData = list()


# Open the walls.txt file to create a list of walls
wall_file = open("walls.txt", "r")
wallCoord = wall_file.read().strip().split("\n")
wallList = list()

wall_file.close()

if(wallCoord[0]!=''):
for a in wallCoord:
elements = a.split(" ")
wall = w.Wall(elements[:4], float(elements[4]), float(elements[5]),
    float(elements[6]))
wallList.append(wall)

# Creating the cell list
listCells = creationCells()

# Creation
if(optimizationMode):
# Optimization part
# We create all the image transmitters created from the total list of
    transmitters
listTotEmitters = list()
numDisc = 10
sizeX = int(mapSizeX/numDisc)
sizeY = int(mapSizeY/nberDisc)
for b in range(0, numDisc):
for a in range(0, numDisc):
emetStart = [em.Issuer([a*sizeX+0.5, b*sizeY+0.5])]
listTotSenders.extend(sendStart[:])
for e in listTotEmitters:
listEmitImage = list()
listEmitImageTot = list()
listEmitImageTot.append(e)
# We add all the image transmitters created to the total list
```

```
if(NBRE_MAX_REFL > 0):
creationListIssuerImg(e, wallList)
listEmetImageTot.extend(listEmetImage[:])


for i in range(NBRE_MAX_REFL-1):
listEmBis = listEmetImage[:]

listEmitImage = list()

for emit in listEmBis:
creationListImitter(emet, wallList)
listEmetImageTot.extend(listEmetImage[:])
listOpti.append(listEmetImageTot[:])

else:
# Classic Mode
# We create all the image transmitters created from the total list of
    transmitters

listEmetImageTot.extend(emetDepart)

if (NBRE_MAX_REFL > 0):
for a in emetDepart:
creationListImitter(a, wallList)
listEmetImageTot.extend(listEmetImage)

for i in range(NBRE_MAX_REFL - 1):
listEmBis = listEmetImage[:]

listEmitImage = list()
for emit in listEmBis:
creationIssuerListImg(emet, wallList)
listEmetImageTot.extend(listEmetImage)
```

MapEditor.py

```
"""
MAP EDITOR CREATING A MAP FROM AN EXCEL FILE
"""

import openpyxl as op

"""
Different colors with their hexadecimal translation and their
    functionality:
-green: transmitter FF00B050
- blue: middle wall FF002060
-red: wall end FFFF0000
"""

def editMapBis(name):
"""
excel file name
:return: returns the list of walls with their characteristics and the
    emitter(s) with their characteristics
"""
posWallH = list()
vWallPos = list()
posEmitters = list()

wb = op.load_workbook(name, read_only=True)
ws = wb.active

for max_row, row in enumerate(ws, 1):
if all(c.value is None for c in row):
break

maxRow = ws.max_row

# Creation of horizontal walls
for i in range(ws.min_row, ws.max_row+1):
value = 0
listExtreme = list()
```

```
for j in range(ws.min_column, ws.max_column+1):
try:
if(ws.cell(i, j).fill.start_color.index == "FFFF0000"):
if(len(listExtreme)==4):
a = listEnd[0]
b = listEnd[1]
listExtreme = [a,b]
if (j != 1):
if (not ws.cell(i, j-1).fill.start_color.index == "FFFF0000"):
listExtreme.extend([str((j - 1)*dim), str((maxRow - i)*dim)])
else:
listExtreme.extend([str((j - 1)*dim), str((maxRow - i)*dim)])

elif(ws.cell(i,j).fill.start_color.index == "FF002060"):

val = ws.cell(i, j).value
if(val != None):
value = value
else:

if(len(listExtreme)==4):
width = value. split("/")[0]
type = value.split("/")[1]
wallType = checkWallType(type)
listExtreme.extend([width, wallType[0], wallType[1]])
posWallH.append(listExtreme)
listExtreme = list()

if(j == ws.max_column):
if (len(listExtreme) == 4):
width = value. split("/")[0]
type = value.split("/")[1]
wallType = checkWallType(type)
listExtreme.extend([width, wallType[0], wallType[1]])
posWallH.append(listExtreme[:])
listExtreme = list()

if(ws.cell(i, j).fill.start_color.index == "FF00B050"):
posEmitters.append([str((j - 1)*dim), str((maxRow - i)*dim), ws.cell(
    i, j).value])
except:
```

```
pass

# Creation of the vertical walls
for i in range(ws.min_column, ws.max_column+1):
value = 0
listExtreme = list()
for j in range(ws.min_row, ws.max_row+1):
try:
if(ws.cell(j,i).fill.start_color.index == "FFFF0000"):

if (len(listExtreme) == 4):
a = listExtremity[0]
b = listExtremity[1]
listExtremity = [a, b]

if(j!=1):
if(not ws.cell(j-1, i).fill.start_color.index == "FFFF0000"):
listExtreme.extend([str((i - 1)*dim), str((maxRow - j)*dim)])
else:
listExtreme.extend([str((i - 1)*dim), str((maxRow - j)*dim)])



elif(ws.cell(j,i).fill.start_color.index == "FF002060"):

val = ws.cell(j, i).value
if(val != None):
value = value


else:
if(len(listExtremite)==4):
width = value.split("/")[0]
type = value.split("/")[1]
WallType = checkWallType(type)
listExtreme.extend([width, wallType[0], wallType[1]])
posWallV.append(listExtreme)
listExtremity = list()


if(j == ws.max_row):
```

```
if (len(listExtreme) == 4):
width = value. split("/")[0]
type = value.split("/")[1]
wallType = checkWallType(type)
listExtreme.extend([width, wallType[0], wallType[1]])
posWallV.append(listExtreme[:])
listExtremity = list()
except:
pass
wb.close()
posWallH.extend(posWallV)
createFileWalls(posWallH)
createFileMap(ws.max_column, ws.max_row, posEmitters)

def checkWallType(number):
"""
:param number: the number that corresponds to a specific material
:return: returns the characteristics of the material (brick, concrete
    , or partition)
"""
count = int(count)
char = list()
if(number==1):
#Brick
char = ['4.6','0.02']
elif(number==2):
#Concrete
char = ['5', '0.014']
else:
#Partition
char = ['2.25', '0.04']

return character

def createFileWalls(listWalls):
"""
:param listWalls: the list of walls on the map
:return: no return. We create the wall file
"""
wall_file = open("walls.txt", "w")
for wall in listWalls:
```

```
for elem in wall:
wall_fichier.write(elem)
wall_file.write(' ')
wall_file.write('\n')
wall_file.close()

def createFileMap(maxX, maxY, posEmitters):
"""
:param maxX: the size in X of the map
:param maxY: the size in Y of the map
:param posEmitters: the list of emitters containing their position
   and their phase
:return: no return. We create the map file
"""
map_file = open("map.txt", "w")
for posEmit in posEmitters:
map_fichier.write(emitPos[0] + " " + emitPos[1]+ " " + str(emitPos
   [2]) + "/")
map_file.write("\n"+str((maxX-1)*dim))
map_file.write(" ")
map_file.write(str((maxY-1)*dim))
map_file.write(" ")
map_fichier.write("100") # Discretization
map_file.close()

# Size of a cell. Multiply all the cells by the dimension
dim = 4
```

Issuer.py

```
"""

DEFINITION OF THE TRANSMITTER OBJECT

"""

from Object import Object as obj

class Emitter(obj):

def __init__(self, position):
"""
:param position: position initialization
"""
obj.__init__(self, position)
self.walllist = list()
self.phase = 0

def addWallList(self, wall):
"""
:param wall: a wall by which the emitter was reflected
:return: no return. Updates the list of walls the emitter was
   reflected from
"""
self.walllist.append(wall)

def getListWalls(self):
"""
:return: returns the list of walls by which the emitter was reflected
"""
return self.walllist

def setPhase(self, phase):
"""
:param phase: the phase of the transmitter
:return: no return
"""
self.phase = phase
```

Menu.py

```
"""

MENU MANAGING THE DIFFERENT FUNCTIONALITIES

"""

# Import necessary files and libraries
from matplotlib import colors
from CreateMapElements import listEmetImageTot, mapSizeX, mapSizeY,
    listOpti, modeOptimisation
from PowerCalculation import *

def optimization(listDatas, upperthreshold, lowerthreshold):
"""
possible transmitter position
:param highthreshold: the threshold above which the power saturates
:param lowthreshold: the threshold below which the power saturates
:return: returns the list containing the data of the transmitter
    having the best position
"""
totalMax = -10000000000000 # Total starting power
elementPos = -1
elementPosMemory = 0
for subListData in listData:
elementPos += 1
total = 0
for subSubList in subListData:
for elem in subSubList:
if(elem>thresholdHigh):
total+=highthreshold
elif(elem<lowthreshold):
total+=low threshold
else:
total+=elem
if(total==ThresholdHigh*len(underListData)*len(underSubList)):
# Case where there is network everywhere in the map. Stop the
    function and return the position
print("Area already perfect")
break
```

```
if(total>totalMax):
totalMax = total
elementPosMemory = elementPos

return elementPosMemory



# HAND

# BASIC VARIABLES

# Definition of bounds for optimization
thresholdOptHigh = -51
thresholdOptLow = -82

# Local zone yes or no
LocalArea = True

# Bit rate option or not
booleanDebitBin = False

# Coverage area
coveragearea = False

# Definition of the minimum and maximum flow requested
terminalMinFlow1 = 54; terminalMinFlow2 = 454; minBound = -82

# Definition of the color palette
name = 'jet'; namePaletteColor = "Power [dBm]"

# Definition of the saturation limit / Minimum and maximum power
terminalMinPower1 = -82; terminalMinPower2 = -51

lowerbound = -1000; terminalSup = 1000

# Conversion to binary if requested
if(booleanDebitBin):
terminalMin = conversionBinaryDebit(terminalMin)
terminalMin1 = terminalMinFlow1
terminalMin2 = terminalMinFlow2
```

```
#Setting the color palette for bit rate
name='inferno'
namePaletteColor = "Bitrate [Mb/s]"
else:
terminalMin1 = terminalMinPower1
terminalMin2 = terminalMinPower2

# Definition of the graph, of these axes, ...
fig = p.figure()
fig.canvas.set_window_title('Ray-tracing project')

p.xlabel('x [m]', fontsize=12)
p.ylabel('y [m]', fontsize=12)

# Data calculation
advancement = 1
if(optimizationMode):
# Browse different possible transmitters to find the best possible
    position
listDataTot = list()
for listElem in listOpti:
for cell in cellList:
cell.resetPower(0)

listData = list()
for emit in listElem:
totalElem = len(listElem)*len(listOpti)
powerTotal(emit, localZone)
print(str(progress) + "/" + str(totalElem)) # Display program loading
progress += 1

# Creation of the list containing the data
listData = makeListGraph(listCells, booleanDebitBin, localZone)
listDataTot.append(listData[:])
posNum = optimization(listDataTot, upperOptthreshold,
    lowerOptthreshold)
listData = listDataTot[posNum]

else:
for emit in listEmitImageTot:
powerTotal(emit, localZone)
```

```
print(str(progress)+"/"+str(len(listEmetImageTot))) # Displays the
    loading of the program
advancement+=1

# Creation of the list containing the data
listData = makeListGraph(listCells, booleanDebitBin, localZone)

if(areaOfCoverage):
# Display of coverage area
nameFunction = "Coverage area"
cmap = colors.ListedColormap(['gray', "orange", 'red'])
bounds = [boundLower, boundMin1, boundMin2, boundSup]
norm = colors.BoundaryNorm(bounds, cmap.N)
img = p.imshow(listData, interpolation='none', origin='lower', cmap=
    cmap, norm=norm, extent=[0, mapSizeX, 0, mapSizeY])
cbar = p.colorbar(img, cmap=cmap, norm=norm, boundaries=bounds, ticks
    =[boundMin1, boundMin2])
cbar.ax.set_ylabel(namePaletteColor, rotation=270, labelpad=25)
else:
# Map display
nameFunction = " Ray-Tracing "
p.imshow(listeData,interpolation='none',cmap=p.get_cmap(name), origin
    ='lower', vmin = boundMin, extent=[0, mapSizeX, 0, mapSizeY])

cbar = p.colorbar()
cbar.ax.set_ylabel(namePaletteColor, rotation=270, labelpad=25)


# Display of walls
for elem in wallList:
p.plot([elem.posStart[0], elem.posEnd[0]], [elem.posStart[1], elem.
    posEnd[1]], color='k', linewidth=3)

p.title(FunctionName, y=1.02)
p.show()
```

Object.py

```
"""

DEFINITION OF AN OBJECT PRESENT ON THE MAP

"""

class Object:

def __init__(self, position):
"""
:param position: position initialization
"""
self.position = position

def setPos(self, pos):
"""
:param pos: the position of the object
:return: no return
"""
self. position = pos;

def getPos(self):
"""
:return: returns the position of the object
"""
return self. position;
```

SpacialOperations.py

```
"""

SPACE OPERATIONS / GEOMETRY

"""

import math as m


def distCalculate(point1, point2):
"""
:param dot1: dot 1
:param dot2: dot 2
:return: returns the distance between these two points
"""
return m.sqrt((dot1[0]-dot2[0])**2+(dot1[1]-dot2[1])**2)

def calculationSymmetry(wall, posEmitter):
"""
:param wall: wall by which the transmitter will be mirrored
:param posEmitter: the position of the emitter
:return: returns the position of the image emitter
"""
positionEmetImage = [0, 0]
if (wall.posStart[0] == wall.posEnd[0]):
# Wall v
positionEmitImage[0] = wall.posStart[0] + (wall.posStart[0] -
    posEmitter[0])
positionEmitImage[1] = posEmit[1]
else:
# Wall h
positionEmitImage[0] = posEmit[0]
positionSendImage[1] = wall.posStart[1] + (wall.posStart[1] -
    posSender[1])

return positionEmitImage


def intersectionCalculate(wallPosStart, wallPosEnd, cellPos, posEmit)
```

```
    :
"""
:param murPosDebut: position of the first end of the wall
:param wallPosEnd: position of the second end of the wall
cell position
:param posEmet: position of the transmitter
:return: returns the intersection if there is one otherwise returns 0
"""
x=0; y = 0
output = 1 # inter (1) or not (0)
res1 = paramLines(wallPosStart, wallPosEnd)
res2 = paramLines(cellPos, emitPos)
if(res1[0]):
if(res2[0]):
output = 0
x = res1[2]
y = res2[1]*x + res2[2]
elif(res2[0]):
x = res2[2]
y = res1[1] * x + res1[2]
elif(res2[1]==res1[1]):
output = 0
else:
x = (res2[2]-res1[2])/(res1[1]-res2[1])
y = res1[1]*x + res1[2]

if(output):
if (not (interOnSegment(cellPos, emitPos, [x,y]) and interOnSegment(
    wallPosStart, wallPosEnd, [x,y]))):
# Inter step on the segment
output = 0
if(output):
return [x,y]

return exit

def interSurSegment(p1Bis, p2Bis, pInter):
"""
:param p1Bis: point 1
:param p2Bis: dot 2
:param pInter: point of intersection between these 2 points
```

```
:return: returns whether or not the intersection is on the wall in
    question, i.e. on the segment [p1Bis, p2Bis]
"""
p1 = p1Bis[:]
p2 = p2Bis[:]
output = 1
if(p1[0] == p2[0]):
if(p2[1]<p1[1]):
inventory = p1[1]
p1[1] = p2[1]
p2[1] = inventory
if(pInter[1]<p1[1] or pInter[1]>p2[1]):
output = 0
else:
if(p2[0]<p1[0]):
inventory = p1[0]
p1[0] = p2[0]
p2[0] = inventory
if (pInter[0] < p1[0] or pInter[0] > p2[0]):
output = 0

return exit



def paramRights(p1, p2):
"""
:param p1: dot 1
:param p2: dot 2
:return: returns res = true if the line is vertical otherwise False
    and the slope and the ordinate at the origin of the line
"""
res=False
a = 0; b = 0

if (p1[0]==p2[0]):
res=True
b = p1[0]
else:
a = (p2[1] - p1[1]) / (p2[0] - p1[0])
b = p1[1] - a * p1[0]
```

```
return [res, a, b]



def angleBetweenPoints(point1, point2):
"""
:param dot1: dot 1
:param dot2: dot 2
:return: returns the angle the segment [point1, point2] and the
    normal
"""
if(dot2[0] == dot1[0]):
angle = m.ft/2
else:
angle = m.atan((point2[1]-point1[1])/(point2[0]-point1[0]))

if(angle<0):
angle+=m.ft
return angle

def calculateAngleTransmit(angleI, epsI, epsT):
"""
:param angleI: angle of incidence
:param epsI: epsilon of incident medium
:param epsT: epsilon of the transmission medium
:return: returns the transmission angle
"""
return m.asin(m.sqrt(epsI/epsT)*m.sin(angleI))

def calculatesAngleIncident(pos1, pos2, wall):
"""
:param pos1: position 1
:param pos2: position 2
:param wall: wall that is intersected with segment [pos1, pos2]
:return: returns the incident angle with the wall
"""
angleBetweenPos = angleBetweenTwoPoints(pos1, pos2)
angleWall = angleBetweenTwoPoints(wall.posStart, wall.posEnd) + m.ft
    / 2
if (wallangle > m.pi):
wallangle -= m.ft
```

```
angleI = m.fabs(angleWall - angleBetweenPos)

if (angleI > m.pi / 2):
angleI -= m.pi

return m.fabs(angleI)
```

Wall.py

```
"""

DEFINITION OF THE WALL OBJECT (MAP WALLS)

"""

import math as m
import cmath as cm

class Wall():

def __init__(self, position, width, epsR, sigma):
"""
:param position: initialization of the wall position (contains these
   two ends: [pos1, pos2])
:param width: the width of the wall
:param epsR: its relative epsilon
:param sigma: its conductivity
"""
self.startPos = [float(position[0]), float(position[1])]
self.endPos = [float(position[2]), float(position[3])]
self.eps = 8.854*10**(-12)*epsR
self.width = width
self.sigma = sigma

def setResistance(self, epsTilde):
"""
:param epsTilde: epsilon tilde
:return: returns nothing. Initializes the resistor.
"""
self.resistance = complex(cm.sqrt(4 * m.pi * 10 ** (-7) / epsTilde))
```

$$\omega_0 \tag{37}$$