ECOLE
POLYTECHNIQUE
DE BRUXELLES

Réalisation d'un logiciel de ray-tracing

ELEC-H304 - Physique des télécommunications

Auteurs :
Blanquez Victor
Dietrich colin

Professeur
DE DONCKER Philippe

année académique 2020 - 2021

Table des matières

1 Introduction **1**

2 Modèle théorique **1**

2.1 Hypothèses 1

2.2 Description des antennes 2

2.2.1 Hauteur équivalente 2

2.2.2 Champ Électrique 3

2.2.3 Puissance rayonnée 4

2.2.4 Réception 4

2.3 Le ray-tracing 5

2.3.1 Onde directe 5

2.3.2 Coefficient de réflexion et de transmission 5

2.3.3 Composantes multi-trajets 6

2.3.4 Tension reçue pour l'onde directe et pour une réflexion 7

2.3.5 Méthode des images 8

3 Logiciel **9**

3.1 Description du code 9

3.2 Vérification analytique des résultats de la simulation 11

3.2.1 Onde directe 12

3.2.2 Transmission à travers un mur 12

3.2.3 Réflexion sur un mur 13

3.2.4 Combinaison des deux cas précédents 14

3.2.5 Conclusion 15

3.3 Résultats du Metropolitan Museum 15

4 Optimisation **15**

4.1 Définition du problème 16

4.2 Algorithme 17

4.3 Résultats 19

5 Conclusion **21**

Appendices **22**

A Représentation des résultats de l'optimisation **22**

B Code **23**

1 Introduction

Ce rapport s'inscrit dans le cadre du cours de Physique des télécommunications (ELEC-H304). Il a pour but le développement d'un logiciel de ray-tracing permettant le calcul de propagations des ondes électromagnétiques. Ainsi, le logiciel permet de calculer la puissance captée par un appareil connecté à un point d'accès Wifi, à l'intérieur d'un bâtiment. Il est également capable d'établir la zone de couverture d'une station de base d'une petite cellule indoor d'un réseau 5G, ainsi que le débit binaire reçu par un récepteur relativement à sa position.

La première partie de ce rapport présente les concepts théoriques utiles au développement du code de calcul. Celle-ci permet de déterminer les paramètres du problème à partir de la modélisation des antennes, et les calculs relatifs à la propagation des ondes.
Ensuite, une description du logiciel est faite, et une vérification des résultats est effectuée pour des cas élémentaires. Le résultat pour le Metropolitan Museum est présenté à la fin de cette partie.
Enfin, un algorithme génétique basé sur la méthode de réplication des chromosomes est proposé afin de trouver une zone de couverture optimale, le nombre de sources et leurs positions respectives.

2 Modèle théorique

Lorsque l'on étudie la propagation des ondes électromagnétiques, les développements analytiques deviennent souvent très complexes. Les méthodes de simulations numériques telles que le ray-tracing permettent alors de pallier ce problème.
La suite de cette section présente les différents aspects théoriques utiles à la conception du logiciel de ray-tracing.

2.1 Hypothèses

Afin de simplifier les développements analytiques et d'optimiser le programme de simulation numérique, différentes hypothèses seront admises dans la suite du projet :

1. **Hypothèses sur les antennes :**
   Plusieurs hypothèses sont faites pour les antennes :
   — On considère un réseau 5G fonctionnant à la fréquence $f = 27GHz$ avec des émetteurs et des récepteurs correspondant à des antennes dipôles $\frac{\lambda}{2}$ sans pertes et orientées selon l'axe z.
   — On suppose que la puissance d'émission des antennes vaut $G_{TX} = 20dBm$.
   — On admet que les émetteurs et récepteurs sont tous situés à une même hauteur du sol.
   — On étudie seulement les ondes planes rayonnées par les antennes se propageant dans le plan horizontal (x,y).

2. **Approximation des champs lointains :**
   On suppose que l'approximation des champs lointains peut être appliquée à l'ensemble du bâtiment dans lequel la zone de couverture est étudiée. La frontière entre les régions de champ proche et de champ lointain étant définie par convention à une distance r_{ff} de l'émetteur telle que :
   $$r_{ff} \;=\; \mathbf{Max}\,\{1,6\lambda\,;\,5D\,;\,\frac{2D^2}{\lambda}\} \;=\; 5D \;=\; 0,0278m \tag{1}$$
   avec $D = \frac{\lambda}{2} = 0,00556m$ la dimension caractéristique maximale de l'émetteur.
   Cette hypothèses est donc valide tant que le récepteur se trouve à une distance de plus de $2,78cm$ de l'émetteur, ce qui sera toujours vérifié.
3. **Ondes planes :**
   Les ondes sont supposées localement planes. En effet, comme il a été vu au cours, chaque interaction d'une onde avec l'environnement peut être étudiée en supposant que l'onde est localement plane, à la condition que la distance aux sources soit suffisante, ce qui est donc bien vérifié avec l'hypothèse des champs lointains.
4. **Interactions avec l'environnement :**
   Les interactions des ondes planes avec l'environnement sont calculées grâce aux modèles vus au cours. Ainsi, seules les ondes ayant subi une, deux ou trois réflexions, et ayant subi des transmissions au travers des murs sont prises en compte. La diffraction est négligée.
   Enfin, le mobilier, les portes et les personnes présents à l'intérieur du bâtiment ne sont pas non plus pris en compte.

2.2 Description des antennes

Comme dit précédemment, les sources utilisées sont des antennes dipôles $\frac{\lambda}{2}$ sans pertes et orientées verticalement selon l'axe z. Cette antenne peut être considérée comme constituée de deux brins métalliques de longueur $\frac{\lambda}{4}$ disposés parallèlement et bout à bout tel que celle-ci soit alimentée en son milieu.

2.2.1 Hauteur équivalente

Les caractéristiques de rayonnement et de réception d'une antenne sont synthétisées dans la notion de hauteur équivalente :

$$\vec{h}_e(\theta,\phi) = \frac{-1}{\underline{I_a}} \int_D \vec{J}(\vec{r})\; e^{\,j\beta z\,(\vec{1}_r\,.\,\vec{1}_z)}\; dV \tag{2}$$

où (θ,ϕ) est la direction d'incidence de l'onde plane, $\beta = \frac{2\pi}{\lambda}$ est le nombre d'onde, D est le volume de l'antenne, $\vec{J}(\vec{r})$ est la densité de courant au sein de l'antenne en émission, et $\underline{I}_a$ est le courant

aux bornes d'accès de l'antenne en émission.

En considérant que l'antenne peut être modélisée par une ligne bifilaire ouverte dont on aurait écarté les brins métalliques de longueur $L = \frac{\lambda}{4}$, et en supposant que le courant serait resté invariable lors de cette opération, on peut faire une approximation du courant $\underline{I}(z) = \underline{I}_a cos(\beta z)$ tel que $\beta L = \frac{\pi}{2}$. De plus, l'antenne étant filiforme, l'intégrale de l'équation (2) devient une intégrale de ligne avec $\vec{r} = z\vec{1}_z$ tel que :

$$\begin{aligned} \vec{h}_e(\theta,\phi) &= \frac{-1}{\underline{I_a}} \int_{-\frac{\lambda}{4}}^{\frac{\lambda}{4}} \underline{I}(z)\, e^{\,j\beta z\,(\vec{1}_r\,.\,\vec{1}_z)}\; dz.\vec{1}_z \\ &= \frac{-\lambda}{\pi}\, \frac{cos[\frac{\pi}{2}\, cos(\theta)]}{sin^2(\theta)}\, \vec{1}_z \end{aligned} \tag{3}$$

Enfin, on remarque que la hauteur équivalente ne dépend que de θ et donc pas de l'orientation dans le plan d'incidence (x,y). Bien plus, le plan d'incidence étant orthogonal à la direction $\vec{1}_z$ c'est à dire $\theta = \frac{\pi}{2}$, on peut conclure que dans notre cas, la hauteur équivalente ne dépend que de la fréquence :

$$\vec{h}_e = \frac{-\lambda}{\pi}\, \vec{1}_z \tag{4}$$

2.2.2 Champ Électrique

En appliquant l'approximation des champs lointains (cf. (1)) sur le potentiel vecteur $\vec{A}(\vec{r})$, on peut déterminer le champ électrique émis par les antennes à partir de l'expression $\underline{\vec{E}} = j\omega\, (\underline{\vec{A}} \times \vec{1}_r) \times \vec{1}_r$. Ainsi, en considérant l'impédance du vide $Z_0 = \sqrt{\frac{\mu_0}{\epsilon_0}}$:

$$\begin{cases} \underline{\vec{A}}(\vec{r}) = \dfrac{\mu_0}{4\pi} \displaystyle\int_D \underline{\vec{J}}(\vec{r'})\, \frac{e^{-j\beta\,|\vec{r}-\vec{r'}|}}{|\vec{r}-\vec{r'}|}\, dV' = \underline{\vec{A}}(\vec{r}) = \frac{\mu_0}{4\pi}\, \frac{e^{-j\beta r}}{r} \int_D \underline{\vec{J}}(\vec{r'})\, e^{-j\beta\,\vec{r'}.\vec{1}_r}\, dV' \\ \underline{\vec{E}} = j\omega\, \underline{I}_a\, \vec{h}_e(\theta,\phi)\, \dfrac{\mu_0}{4\pi} \dfrac{e^{-j\beta r}}{r} = j\, \dfrac{Z_0\, \underline{I}_a}{2\pi}\, \dfrac{cos[\frac{\pi}{2}\, cos(\theta)]}{sin(\theta)}\, \dfrac{e^{-j\beta r}}{r}\, \vec{1}_\theta \end{cases} \tag{5}$$

On observe donc bien que, dans la région de champ lointain, la distribution angulaire du rayonnement est indépendante de la distance à l'antenne, c'est à dire que le rayonnement est isotrope dans le plan d'incidence. Bien plus, les antennes dipôles $\frac{\lambda}{2}$ sont orientées verticalement de façon à ce que, dans le plan horizontal (x,y), les vecteurs $\vec{1}_\theta$ et $\vec{1}_z$ aient la même direction (car $\theta = \frac{\pi}{2}$). Elles rayonnent donc un champ électromagnétique polarisé linéairement selon l'axe z de telle sorte que la composante du champ soit bien perpendiculaire au plan d'incidence. Ainsi l'expression finale du champ électrique rayonné dans le plan (x,y), c'est à dire pour $\theta = \frac{\pi}{2}$ est :

$$\underline{\vec{E}} = j\, \frac{Z_0\, \underline{I}_a}{2\pi}\, \frac{e^{-j\beta r}}{r}\, \vec{1}_z \tag{6}$$

2.2.3 Puissance rayonnée

Le vecteur de Pointing $|\vec{S}(r,\theta,\phi)| = \frac{1}{2Z_0}\,|\underline{\vec{E}}|^2$ dont le module représente localement la densité surfacique de puissance rayonnée par une antenne permet de calculer la puissance rayonnée par unité d'angle et donc de calculer la puissance totale rayonnée :

$$\begin{cases} U(\theta,\phi) \;=\; r^2\,|\vec{S}(r,\theta,\phi)| & = \; Z_0\,\dfrac{|\underline{I}_a|^2}{8\pi^2}\,[\dfrac{cos[\frac{\pi}{2}\,cos(\theta)]}{sin(\theta)}]^2 \\ P_{ar} \;=\; \displaystyle\int_0^{2\pi}\int_0^{\pi} U(\theta)\,sin(\theta)\;d\theta\,d\phi & = \; \dfrac{3}{32}\,Z_0\,|\underline{I}_a|^2 \end{cases} \tag{7}$$

De plus, on a posé l'hypothèse que les antennes sont sans pertes, c'est à dire que leur rendement $\eta = \frac{P_{ar}}{P_a} = 1$ avec P_a la puissance totale consommée par l'antenne et P_{ar} la puissance totale rayonnée par l'antenne. sachant que la résistance totale de l'antenne vaut $R_a = R_{ar} + R_{al}$ avec R_{ar} la résistance de rayonnement et R_{al} la résistance ohmique, on peut en conclure que la résistance ohmique est nulle. La résistance de rayonnement vaut donc :

$$R_{ar} \;=\; 2\,\frac{P_{ar}}{|\underline{I}_a|^2} \;=\; 71\Omega \tag{8}$$

Enfin, on peut déterminer la directivité D et le gain G_{TX} de l'antenne dans le plan d'incidence, c'est à dire pour $\theta = \frac{\pi}{2}$, où ils ont une valeur maximale :

$$\begin{cases} D(\theta) = \dfrac{U(\theta)}{\frac{P_{ar}}{4\pi}} & = \; \dfrac{16}{3\pi} \;=\; 1,7 \\ G_{TX} = \eta\,D(\theta) & = \; 1,7 \end{cases} \tag{9}$$

2.2.4 Réception

L'équation (6) nous donne l'expression de l'onde plane incidente sur l'antenne réceptrice, qui est donc de la forme $\underline{\vec{E}}_i = E_i\,e^{-j\beta r}\,\vec{1}_\theta$. On peut alors calculer la tension induite aux bornes de l'antenne en appliquant le théorème de réciprocité :

$$\begin{aligned} \underline{V}_{OC}(0) \;&=\; \frac{-1}{\underline{I}_a}\int_D \vec{J}(\vec{r})\,\cdot\underline{\vec{E}}_i(\vec{r})\;dV \\ &=\; \frac{-1}{\underline{I}_a}\int_D \vec{J}(\vec{r})\,e^{\,j\beta\,(\vec{r}\,.\,\vec{1}_r)}\;dV\;\cdot\vec{E}_i(0) \\ &=\; \vec{h}_e(\theta,\phi).\underline{\vec{E}}_i(0) \end{aligned} \tag{10}$$

où $\vec{E}_i(0)$ est la valeur du champ électrique aux bornes de l'antenne.

2.3 Le ray-tracing

Le ray-tracing est un algorithme permettant de simuler la propagation des ondes électromagnétiques. Pour ce faire, l'algorithme suit les ondes, appelées composantes multi-trajets, entre l'émetteur et le récepteur. Les obstacles peuvent ainsi être modélisés en un ensemble de formes simples permettant de calculer analytiquement les coefficients de réflexion et de transmission.
Afin de simplifier les développements, nous considérons le résultat (6) dans le plan d'incidence tel que le champ électrique soit polarisé selon l'axe z et tel que tous les obstacles lui soient parallèles. Ainsi, seule la composante selon z de $\underline{E}$ sera calculée dans la suite des développements.

2.3.1 Onde directe

Les ondes directes correspondent aux rayonnements allant directement de l'émetteur vers le récepteur sans passer au travers d'obstacles. Ainsi, la densité de puissance incidente sur l'antenne réceptrice en espace libre (c'est à dire en l'absence de tout obstacle) vaut :

$$S = G_{TX} \frac{P_{TX}}{4\pi d^2} \tag{11}$$

où d est la distance parcourue par l'onde directe, G_{TX} est le gain de l'antenne (9), et P_{TX} est la puissance de l'émetteur. Sachant que l'impédance du vide $Z_0 = 120\pi[\Omega]$ et que $|\vec{S}(r,\theta,\phi)| = \frac{1}{2Z_0}|\vec{\underline{E}}|^2$, on peut déterminer le champ électrique capté à l'antenne réceptrice :

$$\underline{E} = \sqrt{60 G_{TX} P_{TX}} \, \frac{e^{-j\beta d}}{d} \tag{12}$$

2.3.2 Coefficient de réflexion et de transmission

Considérons une onde se propageant dans l'air de permittivité proche de celle du vide ϵ_0. Lorsque celle-ci rencontre un mur de permittivité ϵ avec un angle d'incidence θ_i, une partie de celle-ci va être réfléchie avec un angle θ_r tandis que l'autre partie sera transmise avec un angle θ_t. Ces angles peuvent être déterminés à partir des lois de Snell :

$$\begin{cases} \theta_i = \theta_r \\ \sqrt{\epsilon_0}\, sin\theta_i = \sqrt{\epsilon}\, sin\theta_t \end{cases} \tag{13}$$

les coefficients de réflexions Γ et de transmissions T permettent de tenir compte des pertes de puissances associées aux interactions avec les obstacles. Ainsi, en considérant le cas où l'onde est polarisée perpendiculairement au plan d'incidence (cf. figure 1) :

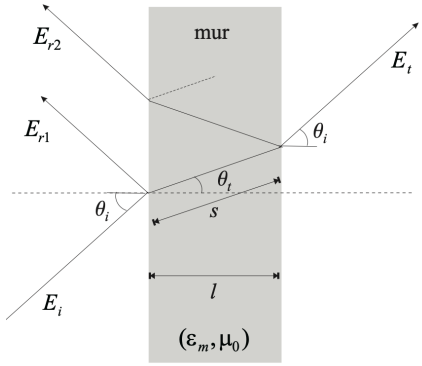


FIGURE 1 – Transmission et réflexion par un mur d'épaisseur l.

$$\begin{cases} \Gamma_\perp = \dfrac{Z\ cos\theta_i\ -\ Z_0\ cos\theta_t}{Z\ cos\theta_i\ +\ Z_0\ cos\theta_t} \\ T_\perp = \dfrac{2Z\ cos\theta_i}{Z\ cos\theta_i\ +\ Z_0\ cos\theta_t} \end{cases} \tag{14}$$

où $Z = \sqrt{\frac{\mu_0}{\epsilon - j\frac{\sigma}{\omega}}}$ est l'impédance du mur et Z_0 est l'impédance du vide. Cependant, à l'intérieur du mur, l'onde subit également de multiples réflexions et transmissions. Ainsi, si l'on tient compte de ces multiples contributions internes, on peut calculer les coefficients totaux :

$$\begin{cases} \Gamma_m(\theta_i) = \Gamma_\perp(\theta_i)\ -\ [1 - \Gamma_\perp^2(\theta_i)]\ \dfrac{\Gamma_\perp(\theta_i)\ e^{-2j\beta_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}}{1\ -\ \Gamma_\perp^2(\theta_i)\ e^{-2j\beta_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}} \\ T_m(\theta_i) = \dfrac{[1 - \Gamma_\perp^2(\theta_i)]\ e^{-j\beta_m s}}{1\ -\ \Gamma_\perp^2(\theta_i)\ e^{-2j\beta_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}} \end{cases} \tag{15}$$

où $s = \frac{l}{cos\theta_t}$ est la distance parcourue par l'onde dans le mur. Dans le cas où le mur présente de faibles pertes, il faut remplacer $j\beta_m$ par la constante de propagation dans le mur $\gamma_m = j\omega\sqrt{\mu_0\,(\epsilon - j\,\frac{\sigma}{\omega})}$ avec σ la conductivité du mur et ω la pulsation de l'onde incidente.

2.3.3 Composantes multi-trajets

Les composantes multi-trajets correspondent aux rayonnements ayant subi plusieurs réflexions et transmissions avec les murs (la diffraction n'est pas prise en compte). Pour calculer le champ électrique capté à l'antenne réceptrice, il suffit de corriger l'expression pour l'onde directe (12) par les coefficients de réflexion et de transmission correspondants :

$$\underline{E} = \prod_{i=0}^{N_r} \Gamma_{mi} \prod_{i=0}^{N_t} T_{mi}\ \sqrt{60 G_{TX} P_{TX}}\ \frac{e^{-j\beta d}}{d} \tag{16}$$

où N_r et N_t sont respectivement le nombre de réflexions et de transmissions subies par l'onde, Γ_{mi} est le coefficient de réflexion de la $i$ème composante multi-trajet, et T_{mi} correspond à la $i$ème transmission subie par l'onde.

En utilisant (10), on peut déterminer la tension induite au récepteur situé en $\vec{r}$ résultant de toutes les composantes multi-trajets :

$$\underline{V}_{OC}(\vec{r}) \;=\; \sum_{n=1}^{N} \vec{h}_e(\theta,\phi)\,.\,\underline{\vec{E}}_n(\vec{r}) \tag{17}$$

où N est le nombre de composantes multi-trajets captées par le récepteur, et $\underline{\vec{E}}_n$ le champ électrique associé. Cela permet donc de calculer la puissance totale reçue par le récepteur :

$$P_{RX} = \frac{1}{8R_a}\,|\sum_{n=1}^{N} \vec{h}_e(\theta,\phi)\,.\,\underline{\vec{E}}_n(\vec{r})|^2 \tag{18}$$

Cependant, cette puissance varie très rapidement selon la position. C'est pourquoi on calcule une puissance moyenne dans une zone locale autour du récepteur (en général de l'ordre du mètre carré) :

$$< P_{RX} > = \;\frac{1}{8R_a}\;\sum_{n=1}^{N} |\vec{h}_e(\theta,\phi)\,.\,\underline{\vec{E}}_n(\vec{r})|^2 \tag{19}$$

Les puissances calculées en watt peuvent être converties en décibels :

$$P[dbm] = 10\log\frac{P[W]}{1mW} \tag{20}$$

Enfin, on considère des antennes dont la sensibilité en décibels varie linéairement avec le débit binaire tel que toute communication soit impossible en-dessous de -82 dbm, et que celle-ci soit optimale et sature au-delà de -73 dbm.

sensibilité	débit binaire
-82 dbm	4 Mb/s
-73 dbm	320 Mb/s

2.3.4 Tension reçue pour l'onde directe et pour une réflexion

Dans le cas de l'onde directe, on peut utiliser les équations (12) et (17) pour déterminer la tension résultant à l'antenne réceptrice :

$$\underline{V}_{OC_1} = \frac{-\lambda}{\pi}\,\sqrt{60G_{TX}P_{TX}}\;\frac{e^{-j\beta d}}{d} \tag{21}$$

et pour le cas d'une onde ayant subi une réflexion sur un mur, on utilise également les équations (14), (15), et (16) :

$$\underline{V}_{OC_2} = \frac{-\lambda}{\pi}\,\Gamma_m\,\sqrt{60 G_{TX} P_{TX}}\,\frac{e^{-j\beta d}}{d} \tag{22}$$

On peut alors faire la somme de $\underline{V}_{OC_1} + \underline{V}_{OC_2}$ pour obtenir la tension d'une antenne réceptrice captant ces deux composantes simultanément.

2.3.5 Méthode des images

La méthode des image est une technique géométrique permettant de calculer les angles d'incidence et la distance totale des composantes multi-trajets. Pour ce faire, on détermine les antennes images des antennes émettrices en faisant une symétrie orthogonale par rapport au mur réflecteur. Le cas pour deux réflexions est représenté figure 2. On a ainsi une première antenne image symétrique de l'antenne émettrice par le mur horizontal, et une seconde antenne image symétrique de la première image par le mur vertical. Le rayon réfléchi est tracé en considérant qu'il est émis par la seconde antenne image.

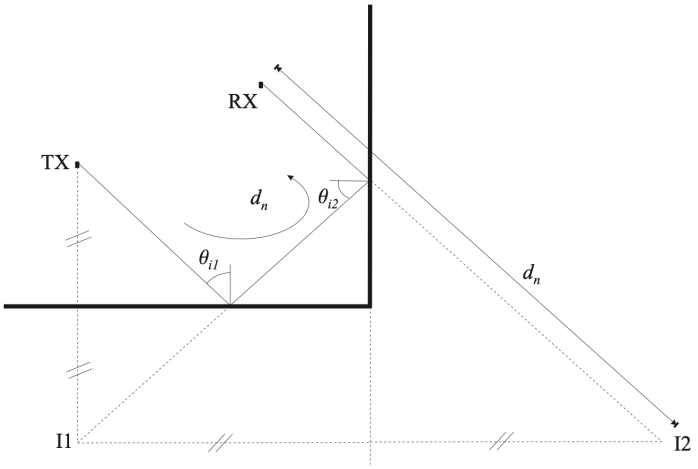


FIGURE 2 – Calcul de l'angle d'incidence et de la distance parcourue par la méthode des images pour une réflexion double. TX est l'émetteur, RX est le récepteur, et I1, I2 les antennes images

3 Logiciel

Le logiciel a été réalisé en c++ car le paradigme de programmation orientée objet permet une modélisation plus aisée du programme et la puissance de calcul fournie par ce langage permet d'obtenir des temps d'exécution optimaux.

3.1 Description du code

Création de la carte

On commence par retranscrire la carte à 2 dimensions du lieu dans lequel s'effectue le ray-tracing en choisissant les dimensions de celle-ci et en stockant les informations relatives aux murs dans un fichier texte ("*wall.txt*"). Chaque ligne du fichier correspond à un mur et contient les coordonnées (x, y) des extrémités du mur, son épaisseur, sa permittivité relative, et sa conductivité (voir figure 3).

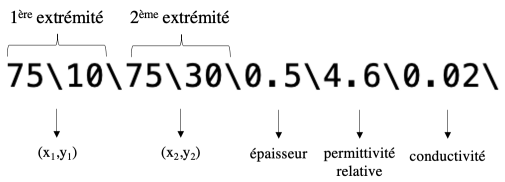


FIGURE 3 – Définition d'un mur dans un fichier texte

Modes d'utilisation

Le logiciel implémente le ray-tracing et possède trois modes de fonctionnement différents :

- **Mode 1** : Il permet l'affichage de tous les rayons qui vont atteindre le récepteur pour un ou plusieurs émetteurs et le calcul de la puissance reçue par celui-ci. L'utilisateur peut choisir la position des émetteurs et du récepteur.
- **Mode 2** : Il permet la visualisation de la zone de couverture et l'intensité de celle-ci pour un ou plusieurs émetteurs. La zone de couverture est calculée en effectuant une discrétisation de la carte. En effet, le logiciel calcule la puissance reçue par un récepteur placé dans chacun des éléments de la discrétisation. L'utilisateur choisit la position des émetteurs et la valeur de discrétisation de la carte, celle-ci influe grandement sur le temps de calcul.
- **Mode 3** : Il détermine le nombre d'émetteurs, ainsi que leurs positions, assurant une zone de couverture optimale, pour une carte donnée. Cette détermination s'effectue à l'aide d'un algorithme génétique dont la fonctionnement est expliqué à la section 4.

Dans tous les cas, il est possible de choisir le nombre d'émetteurs, leur fréquence et le nombre maximum de réflexions que les rayons peuvent effectuer. Ce dernier est fixé à 3 dans les cas présentés, selon nos hypothèses.

Affichages graphiques

Le logiciel utilise la bibliothèque *Gnuplot* afin d'afficher les éléments graphiques.
Pour le premier mode, la couleur des rayons affichés est verte pour les rayons d'onde directe, bleue pour une réflexion, orange pour deux réflexions et rouge pour trois réflexions.
Pour le deuxième mode, il est possible d'observer la zone de couverture en décibels [dBm] ou en mégabit par seconde [Mb/s]. Il est important d'observer que les affichages graphiques ne se font que pour des amplitudes allant de -73 à -82 dBm, et de 40 à 320 Mb/s car le réseau sature au-delà -73 dBm ou 320 Mb/s et la communication est impossible en dessous de -82 dBm ou 40 Mb/s.

Exemples

Voici les résultats obtenus pour l'exécution des deux situations suivantes. Tout d'abord, un émetteur de $27GHz$ est positionné en $(25, 75)$ et les murs forment une pièce 80x80 m^2 contenant deux sous-pièces de respectivement 40x40 m^2 et 50x20 m^2.
Le mode 1 permet de visualiser la figure 4, qui représente les rayons pour un récepteur placé en (20,40) et captant une puissance de $-74dBm$. Le mode 2 permet de visualiser la figure 5, qui représente la zone de couverture.

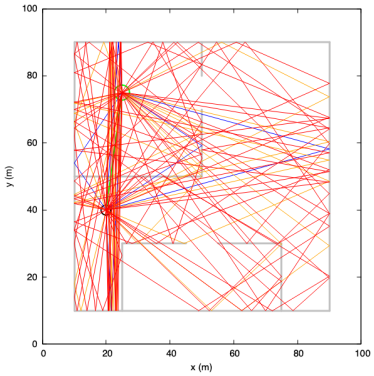


FIGURE 4 – Ray-tracing : récepteur en (20,40)

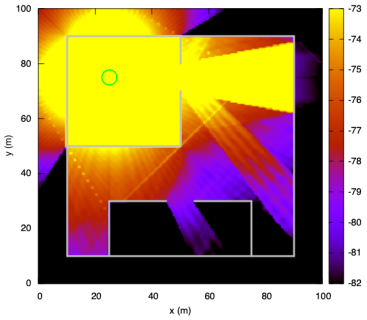


FIGURE 5 – Zone de couverture 1

Ensuite un deuxième émetteur est placé en $(50, 20)$. La figure 6 représente les rayons pour un récepteur placé en (40,20) et captant une puissance de $-56dBm$. La zone de couverture résultante est représentée figure 7.

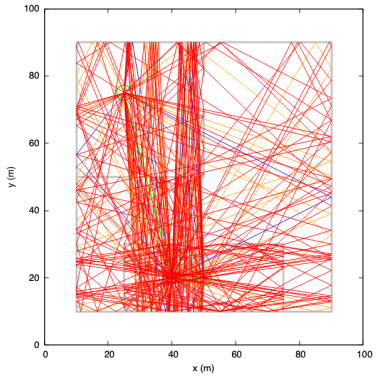


FIGURE 6 – Ray-tracing : récepteur en (40,20)

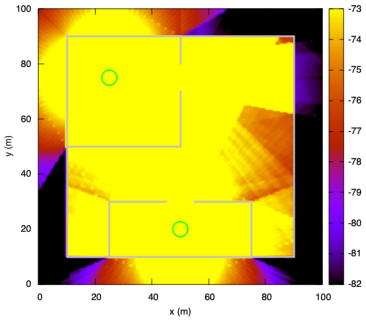


FIGURE 7 – Zone de couverture 2

3.2 Vérification analytique des résultats de la simulation

Afin de valider les puissances calculées par le logiciel, on compare les résultats obtenus aux valeurs déterminées par calcul analytique, pour des cas simples de propagation. Dans les différents cas traités ci-dessous, la source et le récepteur sont séparés d'une distance $d_1 = 6m$ et on considère que l'ensemble des murs sont en briques et d'une épaisseur $s = 0.5m$ Les différentes valeurs nécessaires aux calculs sont reprises dans le tableau ci-dessous.

f	27 GHz	β	565,87815 m^{-1}	$\epsilon_{r,brique}$	4,6
G_{TX}	1,7	h_e	-0,003534 m	σ_{brique}	0,02 S/m
P_{TX}	0,1 W	Z_0	120π Ω	s	0,5 m
R_{ar}	71 Ω	d_1	6 m		

La hauteur équivalente est calculée à partir de l'équation (4) et sachant que les murs des différents cas vérifiés sont en briques, on peut déjà calculer l'impédance caractéristique Z de ceux-ci, ainsi que la constante de propagation γ_m.

$$Z = \sqrt{\frac{\mu_0}{\epsilon_r \epsilon - j\frac{\sigma}{2\pi f}}} = 175,65 + j0,2541$$

$$\gamma_m = j2\pi f\sqrt{\mu_0(\epsilon_r \epsilon - j\frac{\sigma}{2\pi f})} = 1,7565 + j1213,67$$

3.2.1 Onde directe

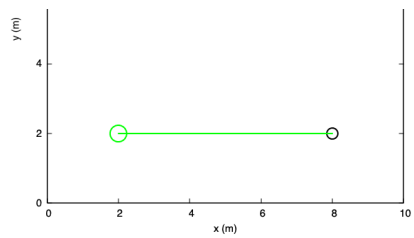


FIGURE 8 – Onde directe

Dans le cas d'une onde ni réfléchie, ni transmise à travers un mur, la puissance moyenne captée par le récepteur est donnée par (19) sans coefficients.

$$< P_{RX} >= \frac{h_e^2\, 60\, G_{TX}\, P_{TX}}{8R_a} \left|\frac{1}{d_1}\right|^2 = 6,2224.10^{-9}\, W = -52,0604\, dBm$$

La puissance obtenue par le logiciel est -52,0604 dBm.

3.2.2 Transmission à travers un mur

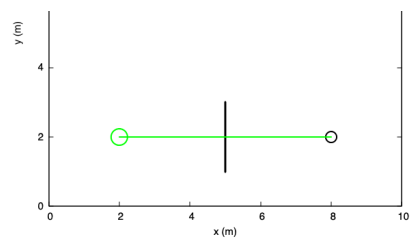


FIGURE 9 – Onde trasmise à travers un mur

Pour calculer la puissance moyenne reçue par l'émetteur, il faut calculer le coefficient de transmission à travers le mur en brique $T_{m1}(\theta_i)$.
Puisque $\theta_i = 0 = \theta_t$, le coefficient de réflexion pour une polarisation perpendiculaire vaut :

$$\Gamma_\perp = \frac{Z\, cos\theta_i\ -\ Z_0\, cos\theta_t}{Z\, cos\theta_i\ +\ Z_0\, cos\theta_t} = \frac{Z - Z_0}{Z + Z_0} = -0,3643 + j0,000627$$

et

$$T_{m1} = \frac{(1 - \Gamma_\perp^2)\, e^{-\gamma_m s}}{1\ -\ \Gamma_\perp^2\, e^{-2\gamma_m s}} = -0,3152 + j0,1834$$

La puissance moyenne vaut donc,

$$< P_{RX} >= \frac{h_e^2\, 60\, G_{TX}\, P_{TX}}{8R_a} \left|\frac{T_{m1}}{d_1}\right|^2 = 8,2752.10^{-10}\, W = -60,8222\, dBm$$

La puissance obtenue par le logiciel est -60,8232 dBm.

3.2.3 Réflexion sur un mur

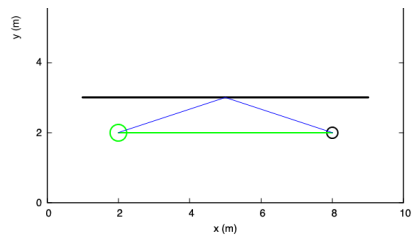


FIGURE 10 – Onde directe et onde réfléchie sur un mur

On commence par calculer la distance parcourue par l'onde réfléchie $d_2 = 2\sqrt{3^2 + 1^2} = 6,3246m$ et les angles d'incidence et de transmission de la réflexion sur le mur.

$$\theta_i = \arctan \frac{3}{1} = 1,249\, rad$$

$$\theta_t = \arcsin(\frac{1}{\sqrt{\epsilon_r}}\ \sin\theta_i) = 0,4582\, rad$$

On calcule ensuite l'épaisseur équivalente,

$$s = \frac{l}{\cos\theta_t} = 0,5575\, m$$

et le coefficient de réflexion pour une polarisation perpendiculaire,

$$\Gamma_\perp = \frac{Z\ cos\theta_i\ -\ Z_0\ cos\theta_t}{Z\ cos\theta_i\ +\ Z_0\ cos\theta_t} = -0,7118 + j0,000351$$

ce qui nous permet de calculer le coefficient de réflexion sur le mur $\Gamma_{m2}(\theta_i)$

$$\Gamma_{m2} = \Gamma_\perp\ -\ (1 - \Gamma_\perp^2)\ \frac{\Gamma_\perp\ e^{-2\gamma_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}}{1\ -\ \Gamma_\perp^2\ e^{-2\gamma_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}} = -0,7178 - j0,04939$$

La puissance moyenne est égale à la somme des puissances reçues pour chaque onde (19),

$$< P_{RX} >= \frac{h_e^2\ 60\ G_{TX}\ P_{TX}}{8R_a}\ \left(\left|\frac{1}{d_1}\right|^2 + \left|\frac{\Gamma_{m2}}{d_2}\right|^2\right) = 9,1251.10^{-9}\, W = -50,3976\, dBm$$

La puissance obtenue par le logiciel est -50,394 dBm.

3.2.4 Combinaison des deux cas précédents

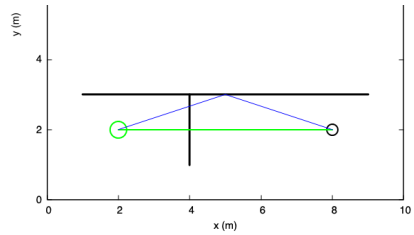


FIGURE 11 – Onde transmise et onde transmise et réfléchie

Puisque nous connaissons les distances d_1 et d_2, le coefficient de transmission de l'onde directe T_{m1} et le coefficient de réflexion de l'onde réfléchie Γ_{m2}, il nous reste à calculer le coefficient de transmission T_{m2} de l'onde réfléchie à travers le mur parallèle à l'axe y, pour pouvoir calculer la puissance moyenne.
Les angles θ_i et θ_t de la transmission à travers le mur valent :

$$\theta_i = \arctan \frac{\frac{1}{3}}{1} = 0,3218\, rad$$

$$\theta_t = \arcsin(\frac{1}{\sqrt{\epsilon_r}}\ \sin\theta_i) = 0,1480\, rad$$

On calcule ensuite l'épaisseur équivalente,

$$s = \frac{l}{\cos\theta_t} = 0,5055\, m$$

et le coefficient de réflexion pour une polarisation perpendiculaire,

$$\Gamma_\perp = \frac{Z\ cos\theta_i\ -\ Z_0\ cos\theta_t}{Z\ cos\theta_i\ +\ Z_0\ cos\theta_t} = -0,3823 + j0,0006176$$

ce qui nous permet de calculer le coefficient de transmission $T_{m2}(\theta_i)$

$$T_{m2} = \frac{(1-\Gamma_\perp^2)\ e^{-\gamma_m s}}{1\ -\ \Gamma_\perp^2\ e^{-2\gamma_m s}\ e^{j\beta\ 2s\ sin\theta_t\ sin\theta_i}} = -0,2129 + j0,2899$$

La puissance moyenne est égale à la somme des puissances reçues pour chaque onde (19),

$$< P_{RX} >= \frac{h_e^2\ 60\ G_{TX}\ P_{TX}}{8R_a}\ \left(\left|\frac{T_{m1}}{d_1}\right|^2 + \left|\frac{\Gamma_{m2}.T_{m2}}{d_2}\right|^2\right) = 1,2029.10^{-9}\ W = -59,1978\, dBm$$

La puissance obtenue par le logiciel est -59,1989 dBm.

3.2.5 Conclusion

On observe que les résultats obtenus par le logiciel correspondent à ceux obtenus par calculs. Les erreurs sont de l'ordre du centième de dBm pour les calculs de puissances de ces cas simples et peuvent être expliquées par les arrondis effectués lors des deux méthodes de calculs. On conclut donc que le logiciel est fonctionnel pour ces cas simples.

3.3 Résultats du Metropolitan Museum

Le calcul de la zone de couverture d'un réseau 5G fonctionnant à $27GHz$ émis par un émetteur dipôle $\frac{\lambda}{2}$, placé en (100,45) [1] dans le plan du MET et vérifiant les hypothèses de la section 2, est représenté à la figure 12 pour une discrétisation de 0,5x0,5 $[m^2]$.

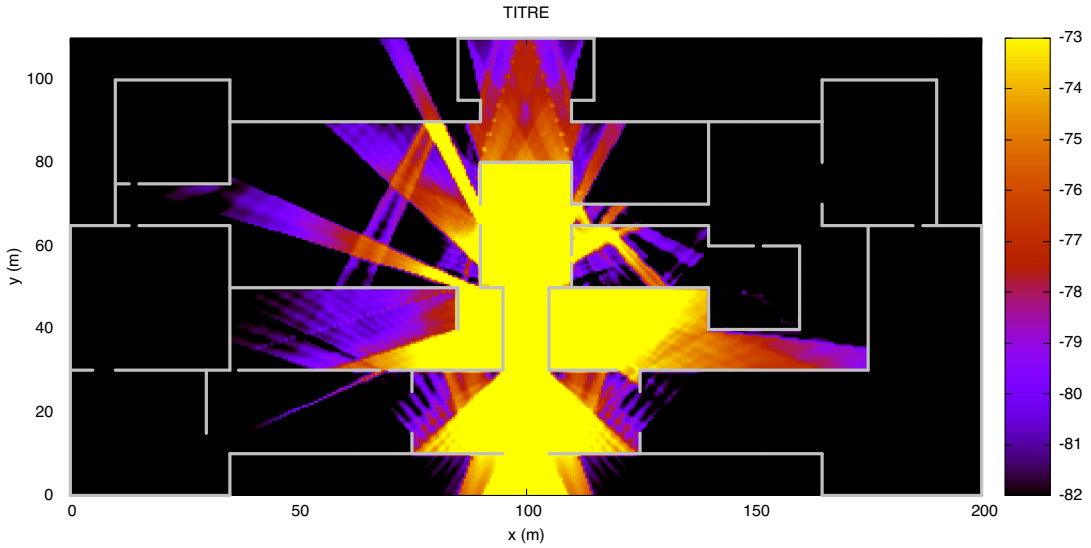


FIGURE 12 – Zone de couverture du Metropolitan Museum pour un émetteur placé en (100,45)

Le logiciel prend par exemple environs deux heures pour calculer la zone de couverture de la figure 12 avec une discrétisation de 1x1 $[m^2]$.

4 Optimisation

L'objectif de cette section est de déterminer le nombre et le positionnement des émetteurs afin que le réseau 5G couvre de manière optimale l'étage du musée. Cela doit être réalisé avec le minimum d'émetteurs possible.

1. cette position correspond à (100,65) dans l'énoncé

Ainsi, afin de résoudre ce problème avec un temps d'exécution raisonnable, nous utilisons un algorithme génétique inspiré de la méthode de réplication des chromosomes. En effet, celui-ci ne nécessite pas de déterminer tous les cas possibles pour le positionnement des sources (ce qui prendrait un temps de calcul très long). De plus, il permet de ne pas devoir d'abord fixer le nombre de sources pour ensuite déterminer leur positionnement optimal. L'algorithme, la théorie, et les images présentés dans la suite de la section sont grandement inspirés de l'article *"Wireless Heterogeneous Transmitter Placement Using Multiobjective Variable-Length Genetic Algorithm"*[2].

4.1 Définition du problème

Environnement

Le problème de positionnement des sources est constitué de trois éléments :

1. la map : elle est divisée en différentes cellules dont le nombre varie selon la discrétisation $n[m]$. Ainsi, si on définit sa longueur L et sa largeur l, on a un nombre $N = \frac{L}{n} \text{x} \frac{l}{n}$ de cellules de positions (x_i, y_j) avec $i = 0, ..., \frac{L}{n}$ et $j = 0, ..., \frac{l}{n}$. Elle contient également l'ensemble des informations relatives au positionnement des murs.
2. l'émetteur : un émetteur est caractérisé par sa position (x, y) et son type. Dans notre cas, on ne considère qu'un seul type d'émetteur : l'antenne dipôle $\frac{\lambda}{2}$.
3. le récepteur : chaque récepteur correspond à une cellule de la map. Il correspond à un point en lequel il est possible de calculer une puissance moyenne captée.

Objectifs

Le problème consiste en trois objectifs principaux :

— maximiser l'aire totale $A[m^2]$ de la zone couverte par les émetteurs
— minimiser le nombre total E d'émetteurs
— minimiser l'aire $I[m^2]$ résultante de l'intersection des zones de couvertures des différents émetteurs présents sur la map

Ainsi, si on définit $f_1(A)$ comme étant le pourcentage total de la map couverte par les émetteurs, et $f_2(I)$ comme étant le pourcentage de l'intersection des différentes zones couvertes par les émetteurs par rapport à la zone de couverture totale, le problème consiste à résoudre :

$$\begin{cases} \min \ f\left[1 - f_1(A)\ ,\ f_2(I)\ ,\ E\right] \\ f_1(A) = A/S \\ f_2(I) = I/A \end{cases} \tag{23}$$

où S est l'aire totale de la map.

2. C. Ting, C. Lee, H. Chang and J. Wu, "Wireless Heterogeneous Transmitter Placement Using Multiobjective Variable-Length Genetic Algorithm", in IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 39, no. 4, pp. 945-958, Aug. 2009.

4.2 Algorithme

Comme dit précédemment, l'algorithme utilisé est un algorithme génétique inspiré du mode de réplication des chromosomes et leur sélection naturelle au fil des générations.

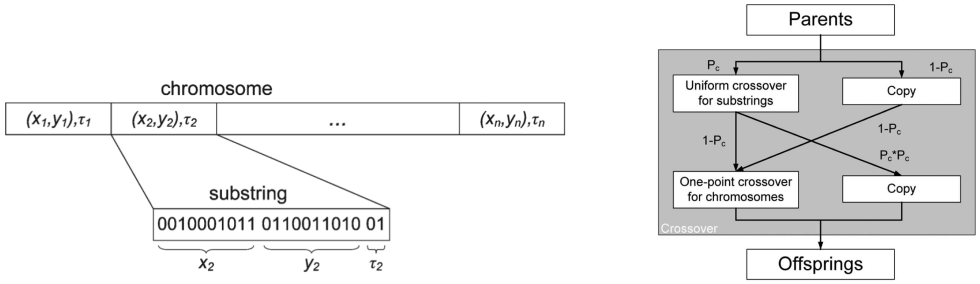
Représentation d'un chromosome

Un chromosome (voir figure 13a) représente un ensemble T d'émetteurs t de position (x_t, y_t) répartis sur la map. Chaque émetteur, ainsi que sa position correspondante, sont appelés une branche du chromosome. Les chromosomes représentent donc chacun une solution possible au problème et sont tous caractérisés par 3 éléments :

— le nombre d'émetteurs n_t

— l'aire totale $A[m^2]$ de la zone couverte par ses émetteurs

— l'aire $I[m^2]$ résultante de l'intersection des zones de couvertures des différents émetteurs du chromosome



(a) représentation d'un chromosome et ses branches

(b) fonctionnement général de l'algorithme

FIGURE 13 – schémas récapitulatifs de l'algorithme

Fonctionnement de l'algorithme

Les étapes suivies par l'algorithme peuvent être résumées dans l'ordre suivant (voir figure 13b) :

1. **Initialisation** : consiste à initialiser un premier ensemble n_c de chromosomes dont le nombre peut être fixé par l'utilisateur. De plus, pour la première génération, une borne supérieure et inférieure sont temporairement appliquées au nombre d'émetteurs par chromosomes. Dans notre cas, la limite inférieure est de 2, et la limite supérieure est de 4. Il est important de remarquer que pour les autres générations, le nombre d'émetteurs par chromosome est libre. Enfin, les positions (x_t, y_t) de chacun des émetteurs est générée aléatoirement tout en respectant les dimensions l et L de la map.
2. **Croisement** : consiste à prendre une paire de deux chromosomes parents afin de les répliquer ensemble, et ainsi d'obtenir deux chromosomes enfants à partir de ceux-ci. Deux types de

croisements sont utilisés : le croisement uniforme et le croisement à point .
Le premier (voir figure 14a), se réalisant avec une probabilité P_c, consiste à sélectionner une branche (donc un émetteur) aléatoirement à chaque chromosome parent, pour ensuite intervertir la position selon x de l'un avec la position selon y de l'autre et inversement :

$$\begin{cases} \text{chromosome 1 :} \ (x_{t1}, y_{t1}) \ \rightarrow \ (x_{t1}, y_{t2}) \\ \text{chromosome 2 :} \ (x_{t2}, y_{t2}) \ \rightarrow \ (x_{t1}, y_{t2}) \end{cases} \tag{24}$$

Le second (voir figure 14b), se réalisant avec une probabilité $1 - P_c$, consiste à couper chaque chromosome parent en deux pour générer deux chromosomes enfants. Ainsi, l'un est la fusion du premier morceau du premier chromosome avec le second morceau du second chromosome et l'autre l'inverse. Les chromosomes n'ayant subi aucun croisement sont alors simplement copié dans la génération suivante.

3. **Mutation** : se réalise avec une probabilité relativement faible P_m. Elle consiste à prendre un émetteur aléatoire d'un chromosome, et à lui ajouter une position aléatoire selon x et selon y en respectant les dimensions de la map.

$$\begin{cases} \text{nombre aléatoire :} \ (x_{rand} \ , \ y_{rand}) \\ \text{mutation :} \ (x_t, y_t) \ \rightarrow \ (x_t + x_{rand} \ , \ y_t + y_{rand}) \end{cases} \tag{25}$$

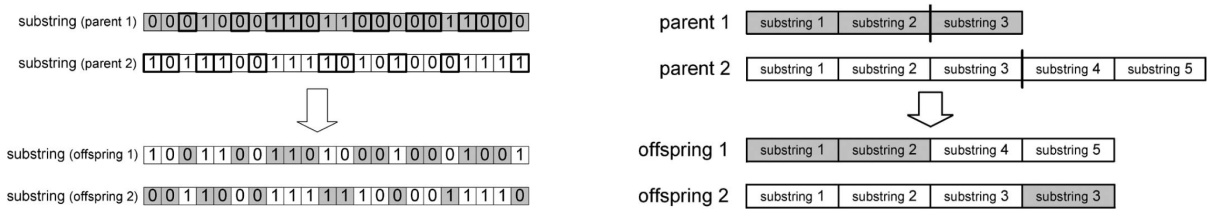
4. **Sélection** : consiste à prendre avec une probabilité égale des paires de chromosomes dans l'ensemble de la population des chromosomes. Elle les fait ensuite s'affronter un à un selon les critères donné en (23). Chaque critère est pondéré par un poids respectif p_A, p_I, et p_{nt} de telle sorte que le chromosome qui obtient les meilleures performances dans le plus de critères l'emporte :

$$\begin{cases} \text{performance 1} \ = \ p_A f_1(A_1) \ - \ p_I f_2(I_1) \ - \ p_{nt} n_{t1} \\ \text{performance 2} \ = \ p_A f_1(A_2) \ - \ p_I f_2(I_2) \ - \ p_{nt} n_{t2} \end{cases} \tag{26}$$

Ainsi si performance 1 > performance 2, alors le chromosome 1 est sélectionné pour la prochaine génération, et inversement.

5. **Élitisme** : consiste à copier directement le meilleur chromosome d'une génération directement dans la génération suivante sans qu'il soit modifié. Le critère qui permet de déterminer le meilleur chromosome est le même que pour la sélection (26). Il est important de remarquer que le fait que le chromosome soit directement copié dans la génération suivante n'empêche pas d'autres chromosomes de réaliser des croisements avec celui-ci.

Enfin, ces cinq étapes s'exécutent de manière itérative jusqu'à ce qu'il ne subsiste plus qu'un seul chromosome, qui est alors le résultat du problème.



(a) croisement uniforme

(b) croisement à un point

FIGURE 14 – schéma des différents types de croisement entre chromosomes

4.3 Résultats

Afin de pouvoir utiliser l'algorithme, il a été nécessaire de déterminer les paramètres du problème. Pour ce faire, plusieurs tests on été réalisés avec différentes valeurs afin de définir des coefficients satisfaisants. Ceux-ci sont représentés dans le tableau suivant :

$P_c = 0{,}7$	$P_m = 0{,}4$	$p_A = 1$	$p_I = 0{,}1$	$p_{nt} = 0{,}1$

Les paramètres de l'algorithme ayant été trouvés, il a fallu s'assurer que celui-ci fournisse des solutions qui convergent bien toutes vers un résultat optimal. Pour ce faire, onze tests ont été réalisés avec à chaque fois une population initiale de 512 chromosomes. La population de chromosomes étant divisée par deux lorsqu'on passe d'une génération à une autre, un total de neuf générations est nécessaire pour obtenir le chromosome final. La figure 15 présente l'évolution du pourcentage moyen de la zone de couverture $f_1(A)$ en fonction des générations. Les premières générations étant toutes générées aléatoirement, on observe bien que la moyenne des pourcentages de la zone de couverture de leurs chromosomes sont relativement faibles (environs 55%). On peut également remarquer que les différents tests ont tendance à converger dans un même intervalle de valeurs (aux environs de 75% à 85%), ce qui valide en partie l'efficacité de la sélection des chromosomes réalisée par l'algorithme. Enfin, sur les onze tests réalisés, dix chromosomes finaux possédaient un total de trois sources, et un seul en possédait quatre (courbe jaune sur la figure 15).

Les onze résultats de l'algorithme génétique sont représentés dans l'annexe A. On observe que les chromosomes possédants trois émetteurs (seul la figure 25 a quatre émetteurs) ont tendance à se positionner aux même endroits, à savoir la zone supérieure droite, la zone inférieure au milieu, et la zone supérieure droite. Bien plus, sept chromosomes sur les 10 (possédant trois sources) possèdent leurs émetteurs dans les trois même pièces. Bien que les positionnements des sources dans la pièce varient, celles-ci semblent être un très bon indicateur pour trouver la solution optimale. Finalement, nous sélectionnons le résultat à trois émetteurs le plus performants (figure 27), qui se trouve dans ces trois pièces. Il possède une zone de couverture d'environ 85% et nous cherchons ensuite à déplacer légèrement ses sources dans les pièces ou il se trouve, pour améliorer la couverture et obtenir la

figure 16 dont les sources se trouvent en (38,70), (100,24) et (170,80).

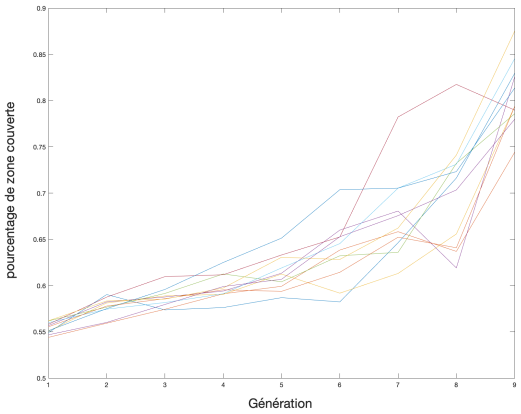


FIGURE 15 – Graphique représentant l'évolution du pourcentage de la zone de couverture moyenne en fonction de la génération

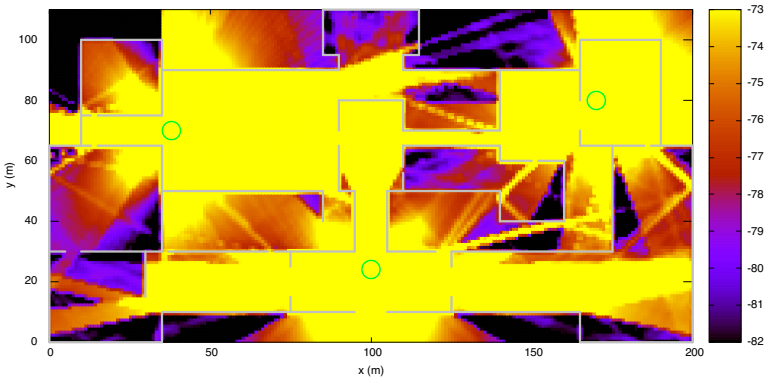


FIGURE 16 – Résultat optimal

Pour conclure, notre démarche a consisté à utiliser l'algorithme génétique afin de tendre vers une solution optimale. Ainsi, on a pu observer que les solutions fournies par l'algorithme se rapprochaient, pour la plupart, approximativement vers une structure commune. Nous avons alors décider de garder cette structure pour en modifier légèrement la disposition afin d'optimiser d'avantage la zone de couverture.

Pour aller plus loin, plusieurs améliorations pourraient être apportée. Tout d'abord, afin d'augmenter la précision, une voie possible serait de diminuer la discrétisation de la map, celle-ci étant de 10 mètres pour les résultats obtenus. De plus, augmenter le nombre de générations permettrait d'observer une meilleure convergence des résultats. Enfin, une détermination plus précise des

paramètres du problèmes permettrait de rendre l'algorithme plus efficace.

5 Conclusion

Après avoir présenté les concepts théoriques relatifs à la propagation d'ondes électromagnétiques, les hypothèses simplificatrices et les concepts relatifs au ray-tracing, un logiciel implémentant celui-ci a été réalisé. Des comparaisons ont permis la validation du fonctionnement de celui-ci dans des cas simples.
Une fois le logiciel implémenté, nous avons pu commencer l'installation d'émetteurs dans le Le Metropolitan Museum (MET). Une fois la répartition de puissance calculée et affichée pour un émetteur placé au centre du musée, il était nécessaire d'optimiser celle-ci, l'objectif étant de pouvoir recevoir partout le plus grand débit binaire possible en plaçant les stations de base de façon optimale, ainsi qu'avoir le moins de stations. L'algorithme de génétique a été implémenté, ses résultats et sa convergence ont été analysés, pour finalement déterminer une solution efficace.

Appendices

A Représentation des résultats de l'optimisation

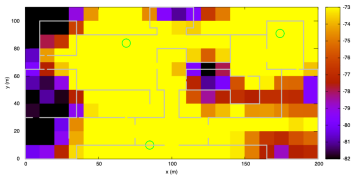


FIGURE 17 – sélection 1

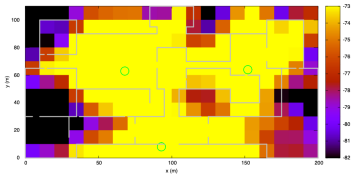


FIGURE 18 – sélection 2

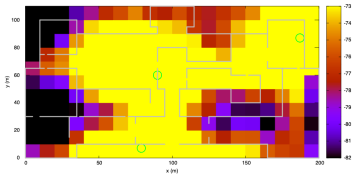


FIGURE 19 – sélection 3

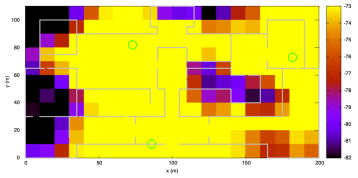


FIGURE 20 – sélection 4

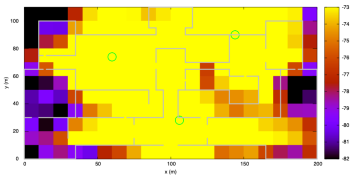


FIGURE 21 – sélection 5

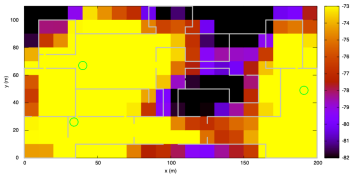


FIGURE 22 – sélection 7

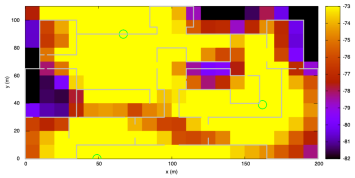


FIGURE 23 – sélection 8

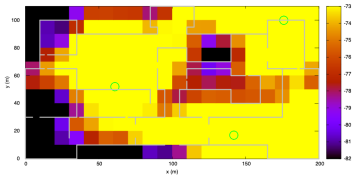


FIGURE 24 – sélection 9

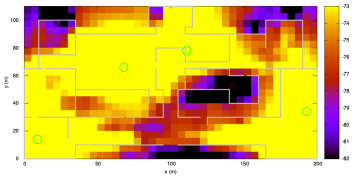


FIGURE 25 – sélection 10

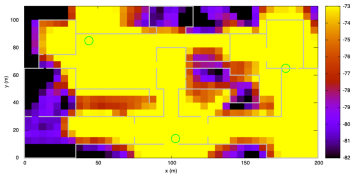


FIGURE 26 – sélection 11

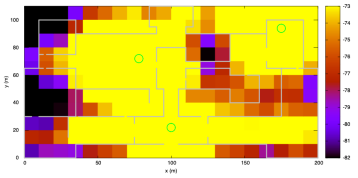


FIGURE 27 – sélection 6

B Code

Dans le fichier "main.cpp", l'utilisateur peut choisir l'option qu'il utilise en changeant la variable "option" en 0,1 ou 2.

```
#include <iostream>
#include <vector>
#include <array>
#include <complex>
#include "Wall.h"
#include "Beam.h"
#include "Graphics.h"
#include "createMap.h"
#include "Source.h"
#include "utils.h"
#include "Map.h"
#include "Optimization.h"

using namespace std;

int main()
{
    /** ----- DECLARATION DES VARIABLES ----- */

    // nombre de reflexions maximums
    int nbRef = 3;
    // classe qui gere la visualisation graphique
    Graphics graph;
    // on cree la liste des walls a partir du fichier .txt
    vector<Wall> listWalls; createListWalls(&listWalls);
    // liste des sources et sources images
    vector<vector<Source> > listSources;
    // nombre de sources initiales
    const int nbSourceinit = 3;
    // position des sources initiales
    double positionSourceInit[nbSourceinit*2] = {38.0, 70.0, 100.0, 23.0, 170.0,
    80.0};
    // position du recepteur
    double pCell[2] = {40.0,20.0};
    Cell *cell = new Cell(pCell);
    // choix option
    int option = 0;

    /** ----- INITIALISATION ----- */

    // on initialise la liste des sources
    initialiselistSources(nbRef, nbSourceinit, positionSourceInit, &listSources);
```

```
    // on cree la liste des sources images
    createListSources(&listWalls, &listSources, nbRef);
    // on ajuste les impedances des murs    partir de la frequence de la source
    for(Wall &wall : listWalls) {
        wall.setImpedance(listSources[0][0].f);
        if(wall.sigma == 0.014) {
            graph.drawLine(wall.position[0], wall.position[1], wall.position[2], wall.position[3], 0);
        } else {
            graph.drawLine(wall.position[0], wall.position[1], wall.position[2], wall.position[3], 100);
        }
    }

    /** ----- CALCUL RAY-TRACING ----- */

    // L'option 0 calcule la puissance re ue par le recepteur pour les
    // differentes sources et affiche les rayons atteignant le recepteur
    if(option == 0) {
        // liste des faisceaux
        vector<Beam> listBeams;
        // affichage des sources et du recepteur
        for(int i = 0; i < nbSourceinit; i++){
            graph.addPoint(positionSourceInit[2*i],positionSourceInit[2*i+1],0);
        }
        graph.addPoint(pCell[0],pCell[1],1);
        // onde directe
        for(int i = 0; i < listSources[0].size(); i++) {
            listSources[0][i].directWave(cell, &listWalls, &listBeams, i);
        }
        // reflexions
        for(int i = 0; i < nbRef; i++) {
            rayTracing(cell, &listWalls, &listBeams, &listSources[i+1], i);
        }
        // calcul puissance captee et affichage des rayons
        for(Beam &beam : listBeams) {
            beam.electricField(listSources[0][0].beta, listSources[0][0].G, listSources[0][0].P_watt);
            cell->updatePower(listSources[0][0].R_a, listSources[0][0].h_e, beam.E);

            int typeOfBeam = beam.size-1;
            for(int j = beam.size-1; j > 0; j--) {
                graph.drawLine(beam.listPositions[j][0], beam.listPositions[j-1][0], beam.listPositions[j][1], beam.listPositions[j-1][1], typeOfBeam);
            }
        }
        std::cout << "power watt : " << cell->P_watt << std::endl;
```

```
        std::cout << "power dbm : " << cell->P_dbm << std::endl;
        std::cout << "d bit binaire : " << cell->bitRate << std::endl;
        // affichage graphique
        graph.printGraph(200.0, 110.0);
    }

    // L'option 1 calcule et affiche la repartition spatiale de puissance
    // pour les diff rentes sources
    else if (option == 1) {
        //affichage sources et cell
        for(int i = 0; i < nbSourceinit; i++){
            graph.addPoint(positionSourceInit[2*i],positionSourceInit[2*i+1],0);
        }
        // map contenant toutes les cell
        Map map = Map(1, 200.0, 110.0);
        for(int i = 0; i < map.nbRow; i++) {
            for(int j = 0; j < map.nbCol; j++) {
                // liste des faisceaux
                vector<Beam> listBeams;
                // onde directe
                for(int h = 0; h < listSources[0].size(); h++) {
                    listSources[0][h].directWave(&map.matrix[i][j], &listWalls, &listBeams, h);
                }
                // reflexions
                for(int h = 0; h < nbRef; h++) {
                    rayTracing(&map.matrix[i][j], &listWalls, &listBeams, &listSources[h+1], h);
                }
                // calcul puissance captee
                for(Beam &beam : listBeams) {
                    beam.electricField(listSources[0][0].beta, listSources[0][0].G, listSources[0][0].P_watt);
                    (map.matrix[i][j]).updatePower(listSources[0][0].R_a, listSources[0][0].h_e, beam.E);
                }
            }
            // indicateur de progression (nbr de lignes effectues/nbr de lignes totales)
            std::cout << "iterarion : " << i << " / " << map.nbRow << std::endl;
        }
        // affichage graphique
        map.createMapFile();
        graph.printMap(map.width, map.height);
    }

```

```
    // L'option 2 selectionne un repartition optimale  de sources a l'aide d'un algorithme
    // genetique
    else if (option == 2) {
        // on initialise la position initiale de la generation pseudo-aleatoire
        srand (time(NULL));
        // nombre de chromosomes dans la 1ere generation
        int nbChromosomes = 256;
        // classe qui gere la selection et reproduction d'une generation
        Optimization opt(listWalls, 3, 10, 200, 110);
        // generation initiale
        opt.initialisation(nbChromosomes);
        // (log(nbChromosomes)/log(2)) generations
        while(opt.generation < log(nbChromosomes)/log(2)) {
            // indicateur de progression
            std::cout << "--- generation : " << opt.generation << std::endl;
            opt.updateGeneration();
        }
        std::cout << "---- END ---- " << std::endl;

        // repatition optimale
        for (int i = 0; i < opt.listChromosome[opt.generation][0].nbSources; i++) {
            std::cout << "source : " << i << std::endl;
            std::cout << " - posx : " << opt.listChromosome[opt.generation][0].positions[i][0] << std::endl;
            std::cout << " - posy : " << opt.listChromosome[opt.generation][0].positions[i][1] << std::endl;
        }

    }

    // conversion et affichage en Mbps
    else if (option == 3) {
        convertPowerFileToMbsFile("dataPowerFinal4.txt");
        graph.printMapMbps(200.0, 110.0);
    }

    return 0;
}
```

Listing 1 – main.cpp

La classe "Source.cpp" modélise les émetteurs et les images de ces émetteurs par la méthode des images.

```
#include <iostream>
#include <vector>
#include <array>
```

```
#include <complex>
#include "Source.h"
#include "Wall.h"
#include "Beam.h"
#define pi 3.14159265358979323846264338327950288 4
#define c 299792458.0

/**
 * Constructeur
 * @param pPosition : position de la source
 * @param pIsImage : booleen qui est vrai si la source est une image
 */
Source::Source(double pPosition[2], bool pIsImage)
{
    // position de la source (x,y)
    memcpy(position, pPosition,2*sizeof(double));
    // frequence d'emission de l'antenne
    f = 27.0e9;
    // nombre d'onde
    beta = (2.0*pi*f) / c;
    // hauteur equivalente
    h_e = -2.0/beta;
    // puissance d'emission en d cibels
    P_dbm = 20.0;
    // puissance d'emission en watts -> Ptx
    P_watt = pow(10,P_dbm/10)*1.0e-3;
    // resistance de rayonnement
    R_a = 71.0;
    // rendement de l'antenne
    rend = 1.0;
    // gain de l'antenne -> Gtx
    G = rend * (16.0/(3.0*pi));
    // booleen qui est vrai si la source est une image
    isImage = pIsImage;
}
/**
 * Calcule la position de l'image de la source associee a un mur
 * @param pWall : mur associe a la source image
 * @param image : position de la source image passee par reference pour etre modifiee
 */
void Source::imageSource(Wall pWall, double *image)
{
    // pente du mur
    double alphaMur = (pWall.position[3] - pWall.position[2]) / (pWall.position[1] - pWall.position[0]);
    //pente de la droite perpendiculaire au mur
    double alphaPerp = -(1/alphaMur);
```

```
    // point d'intersection
    double x, y;

    if(isinf(alphaMur)) {
        x = pWall.position[0];
        y = position[1];
    } else if(isinf(alphaPerp)) {
        x = position[0];
        y = pWall.position[2];
    } else{
        x = (position[1] - pWall.position[2] + alphaMur*pWall.position[0] - alphaPerp*position[0]) / (alphaMur - alphaPerp);
        y = alphaPerp * (x-position[0]) + position[1];
    }
    image[0]= x + (x-position[0]);
    image[1]= y + (y-position[1]);
}
/**
 * Cree le rayon direct entre la source et le recepteur et l'ajoute a la liste des rayons.
 * Calcule le coefficient de transmission total du rayon.
 * @param cell : recepteur
 * @param listWalls : liste des murs
 * @param listBeams : liste des rayons, passee par reference pour etre modifiee
 * @param n : indice de la source initiale dans le liste des sources
 */
void Source::directWave(Cell *cell, std::vector<Wall> *listWalls, std::vector<Beam> *listBeams, int n) {
    // liste des positions du rayon
    std::vector<std::array<double,2> > listPosBeam;
    listPosBeam.push_back((std::array<double,2>){position[0], position[1]});
    listPosBeam.push_back((std::array<double,2>){cell->position[0], cell->position[1]});
    // on cree le faisceau d'onde directe et on l'ajoute a la liste
    (*listBeams).push_back(Beam(listPosBeam));
    // on verifie si il traverse des murs et on change le coefficient total en fonction
    (*listBeams)[n].coeff *= (*listBeams)[n].updateTransferCoefficient((*listWalls), 0);
}
```

Listing 2 – Source.cpp

La classe "Cell.cpp" modélise les récepteurs.

```
#include <iostream>
#include <complex>
#include "Cell.h"
```

```
#include "Graphics.h"

/**
 * Constructeur
 * @param pPosition : position du recepteur
 */
Cell::Cell(double *pPosition)
{
    P_watt = 0;
    memcpy(position, pPosition,2*sizeof(double));
}
/**
* Ajoute la contribution du champ E a la puissance de reception.
* @param Ra : resistance de rayonnement
* @param he : hauteur equivalente
* @param E : champ electrique propage par une onde sur le recepteur
*/
void Cell::updatePower(double Ra, double he, std::complex<double> E) {
    // ajout de la contribution du champ E a la puissance
    P_watt += (1/(8*Ra))*pow(abs(he*E),2);

    // conversion
    P_dbm = 10 * log10(P_watt/(1e-3));
    if(P_dbm > -73) {
        bitRate = 320;
    } else if(P_dbm < -82) {
        bitRate = 40;
    } else {
        double alpha = (320.0 - 40.0) / (-73.0 + 82.0);
        bitRate = alpha*(P_dbm + 73.0) + 320.0;
    }
}
```

Listing 3 – Cell.cpp

La classe "Beam.cpp" modélise les rayons du ray-tracing.

```
#include <iostream>
#include <cmath>
#include <complex>
#include <vector>
#include <array>
#include <cmath>
#include "Beam.h"
#include "Wall.h"
#define pi 3.14159265358979323846264338327950288 4

/**
```

```
 * Constructeur
 * @param pListPositions : liste des positions du rayon
 */
Beam::Beam(std::vector<std::array<double,2> > pListPositions)
{
    // // nombre de reflexions subies par le faisceau + 2 (nbr total de positions)
    size = pListPositions.size();
    // positions intermediaires des reflexions du faisceau
    listPositions = std::vector<std::array<double,2> >(pListPositions);
    // coefficient de reflexion total du faisceau
    coeff = 1;
}
/**
 * Ajoute une position a la liste des positions
 * @param newPos : position a ajouter
 */
void Beam::addR(std::array<double,2> newPos) {
    size += 1;
    listPositions.push_back(newPos);
}
/**
 * Calcule l'angle que fait le rayon avec la normale du mur
 * @param wallPosition : position du mur
 * @param n : indice dans la liste des positions de la derniere position du rayon
    avant l'arrivee sur le mur
 * @return : angle incident
 */
double Beam::angleCalculator(double *wallPosition, int n) {
    // composantes vecteur mur
    double v_a = wallPosition[1] - wallPosition[0];
    double u_a = wallPosition[3] - wallPosition[2];
    // norme vecteur mur
    double d_a = sqrt(pow(v_a,2) + pow(u_a,2));
    // composantes vecteur faisceau
    double v_b = listPositions[n+1][0] - listPositions[n][0];
    double u_b = listPositions[n+1][1] - listPositions[n][1];
    // norme vecteur faisceau
    double d_b = sqrt(pow(v_b,2) + pow(u_b,2));
    // calcul de l'angle incident entre le faisceau et la normale au mur
    double angle_i = pi/2 - acos((v_a*v_b + u_a*u_b) / (d_a*d_b));
    return abs(angle_i);
}
/**
 * Calcule le coefficient de transmission total du rayon direct (sans reflexion).
 * @param listWalls : liste des murs
 * @param n : indice dans la liste des positions de la position de la position de
    depart du rayon
```

```
 * @return : coefficient de transmission
 */
std::complex<double> Beam::updateTransferCoefficient(std::vector<Wall> listWalls, int n) {
    // on cree le coefficient
    std::complex<double> transferCoefficient = 1.0;
    std::vector<std::array<double,2> > listIntersection;
    // pour chaque mur on verifie si le rayon le traverse
    for (Wall &wall : listWalls) {
        bool condx = (listPositions[n+1][0] >= std::min(wall.position[0],wall.position[1]) && listPositions[n+1][0] <= std::max(wall.position[0],wall.position[1]));
        bool condy = (listPositions[n+1][1] >= std::min(wall.position[2],wall.position[3]) && listPositions[n+1][1] <= std::max(wall.position[2],wall.position[3]));
        bool condx2 = (listPositions[n][0] >= std::min(wall.position[0],wall.position[1]) && listPositions[n][0] <= std::max(wall.position[0],wall.position[1]));
        bool condy2 = (listPositions[n][1] >= std::min(wall.position[2],wall.position[3]) && listPositions[n][1] <= std::max(wall.position[2],wall.position[3]));
        if(! ((condx && condy) || (condx2 && condy2)) ) {
            double intersection[2];
            // true si il existe un point d'intersection
            if(wall.transmissionIntersectionPoint((std::array<double,4>){listPositions[n][0],listPositions[n+1][0],listPositions[n][1],listPositions[n+1][1]}, intersection)) {
                for (int i = 0; i < listIntersection.size(); i++) {
                    if(( intersection[0] != listIntersection[i][0]) || (intersection[1] != listIntersection[i][1]) ) {
                        // angle incident entre le faisceau et la normale au mur
                        double angle_i = angleCalculator(wall.position, n);
                        // on ajoute le coeff de transmission du au mur
                        transferCoefficient *= wall.transmissionCoefficientCalculator(angle_i);
                        listIntersection.push_back((std::array<double,2>){intersection[0], intersection[1]});
                    }
                }
                if(listIntersection.size() == 0) {
                    // angle incident entre le faisceau et la normale au mur
                    double angle_i = angleCalculator(wall.position, n);
                    // on ajoute le coeff de transmission du au mur
                    transferCoefficient *= wall.transmissionCoefficientCalculator(angle_i);
                    listIntersection.push_back((std::array<double,2>){intersection[0], intersection[1]});
                }
            }

        }
```

```
    }
    return transferCoefficient;
}
/**
 * Calcule la distance totale parcourue par le rayon
 * @return : distance
 */
double Beam::getDistance() {
    double distance = 0;
    for(int i = 1; i < size; i++) {
        distance += sqrt(pow(listPositions[size-i][0]-listPositions[size-(i+1)][0],2)
    + pow(listPositions[size-i][1]-listPositions[size-(i+1)][1],2));
    }
    return distance;
}
/**
 * Calcule le champ electrique du rayon
 * @param beta : nombre d'onde
 * @param Gtx : gain de l'antenne
 * @param Ptx : puissance d'emission
 */
void Beam::electricField(double beta, double Gtx, double Ptx) {
    std::complex<double> e(0, -beta);
    double dn = getDistance();
    // si on est sur la source -> on lui donne la valeur juste a cote
    if(dn == 0.0) {
        dn = 0.1;
    }
    // champ electrique capte par le recepteur
    E = coeff * sqrt(60*Gtx*Ptx)*exp(e*dn)/(dn);
}
```

Listing 4 – Beam.cpp

La classe "Wall.cpp" modélise les murs de la carte.

```
#include <iostream>
#include <cmath>
#include <complex>
#include <array>
#include "Wall.h"
#define pi 3.14159265358979323846264338327950288 4
#define c 299792458.0
#define eps_0 (8.85418782 * 1e-12)
#define Z_0 (120.0 * 3.14159)
#define mu_0 (4.0 * 3.14159265358979323846264338327950288 4 * 1e-7)

/**
```

```
 * Constucteur
 * @param pPosition : positions des extremites du mur
 * @param pWidth : epaisseur
 * @param pEps_r : permittivite relative
 * @param pSigma : conductivite
 */
Wall::Wall(double *pPosition, double pWidth, double pEps_r, double pSigma)
{
    // position de d part du mur
    memcpy(position, pPosition,4*sizeof(double));
    // epaisseur du mur
    width = pWidth;
    // permittivite relative du mur
    eps_r = pEps_r;
    // permitivite du mur
    eps = eps_r * eps_0;
    // conductivite du mur
    sigma = pSigma;
}
/**
 * Calcule le coefficient de r flexion du rayon qui se reflechit sur le mur.
 * @param angle_i : angle d'incidence par rapport    la normale
 * @return coefficient de reflexion
 */
std::complex<double> Wall::reflexionCoefficientCalculator(double angle_i)
{
    // angle de transmission
    double angle_t = snellLaw(angle_i);
    // distance de propagation dans le mur
    double s = width/cos(angle_t);
    // coefficient de reflexion de  l interface  air/mur
    std::complex<double> gamma1 = (Z*cos(angle_i)-Z_0*cos(angle_t)) / (Z*cos(angle_i)
    + Z_0*cos(angle_t));
    // definition des deux arguments des 2 exponentielles
    std::complex<double> e1 = -2.0*gamma_m*s;
    std::complex<double> j(0.0,1);
    std::complex<double> e2 = beta*2*s*sin(angle_t)*sin(angle_i) * j;
    // coefficient de reflexion total
    std::complex<double> gamma_tot = gamma1 - (1.0-pow(gamma1,2)) * ((gamma1*exp(e1)*
    exp(e2)) / (1.0-pow(gamma1,2)*exp(e1)*exp(e2)));
    return gamma_tot;
}
/**
 * Calcule le coefficient de transmission du rayon qui traverse le mur.
 * @param angle_i : angle d'incidence par rapport    la normale
 * @return coefficient de transmission
 */
```

```
std::complex<double> Wall::transmissionCoefficientCalculator(double angle_i)
{
    // angle de transmission
    double angle_t = snellLaw(angle_i);
    // distance de propagation dans le mur
    std::complex<double> s = width/cos(angle_t);
    // coefficient de reflexion de  l interface  air/mur
    std::complex<double> gamma1 = (Z*cos(angle_i)-Z_0*cos(angle_t)) / (Z*cos(angle_i)
    + Z_0*cos(angle_t));
    // definition des deux arguments des 2 exponentielles
    std::complex<double> e1 = -gamma_m*s;
    std::complex<double> j(0.0,1);
    std::complex<double> e2 = beta*2*s*sin(angle_t)*sin(angle_i) * j;
    // coefficient de transmission total
    std::complex<double> gamma_tot = ((1.0-pow(gamma1,2)) * exp(e1)) / (1.0 - pow(
    gamma1,2)*exp(2.0*e1)*exp(e2));
    return gamma_tot;
}
/**
 * Calcule l'angle de transmission a l'aide de la loi de Snell.
 * @param angle_i : angle d'incidence par rapport    la normale
 * @return angle de transmission
 */
double Wall::snellLaw(double angle_i)
{
    // quotient des permittivit s
    double frac = eps_0 / eps;
    // angle de transmission
    double angle_t = asin(sqrt(frac) * sin(angle_i));
    return angle_t;
}
/**
 * Verifie que le point d'intersection de la droite prolongeant le mur et de
 * la droite passant par 2 points existe et le calcule.
 * @param pPosition : position des 2 points de la droite
 * @param intersection : position de l'intersection, passee par reference pour etre
    modifiee
 * @return 1 si ce point existe et 0 sinon
 */
int Wall::intersectionPoint(std::array<double,4> pPosition, double *intersection)
{
    // coefficient angulaire du mur
    double alpha = (position[3] - position[2]) / (position[1] - position[0]);
    // coefficient angulaire du faisceau
    double pAlpha = (pPosition[3] - pPosition[2]) / (pPosition[1] - pPosition[0]);
    // points d'intersection
    double x, y;
```

```
    // 2 segments paralleles
    if(alpha == pAlpha) {
        return 0;
    }
    // Cas avec intersection
    else if(isinf(alpha)) {
        x = position[0];
        y = pAlpha*(x-pPosition[0]) + pPosition[2];
    } else if(isinf(pAlpha)) {
        x = pPosition[0];
        y = alpha*(x-position[0]) + position[2];
    } else if(alpha == 0) {
        y = position[2];
        x = (y-pPosition[2]) / pAlpha + pPosition[0];
    } else if(pAlpha == 0) {
        y = pPosition[2];
        x = (y-position[2]) / alpha + position[0];
    } else {
        x = (pPosition[2] - position[2] + alpha*position[0] - pAlpha*pPosition[0]) / (alpha - pAlpha);
        y = pAlpha * (x-pPosition[0]) + pPosition[2];
    }
    intersection[0] = x;
    intersection[1] = y;
    return 1;
}
/**
 * Verifie qu'il existe et calcule le point de reflexion sur le mur par la methode des images.
 * @param positionImage : position de la source image visee par le rayon
 * @param positionCell : position de depart du rayon
 * @param intersection : position du point de reflexion sur le mur, passee par reference pour etre modifiee
 * @return 1 si ce point existe sur le mur et 0 sinon
 */
int Wall::reflexionIntersectionPoint(double positionImage[2], double positionCell[2], double *intersection) {
    // calcul le point d'intersection
    std::array<double,4> positionImageCell = {positionImage[0], positionCell[0], positionImage[1], positionCell[1]};
    if(intersectionPoint(positionImageCell, intersection)) {
        double x = intersection[0]; double y = intersection[1];
        // epsilon qu'on ajoute ou en l ve pour se d barraser de l'erreur de repr sentation en base binaire
        double epsilon = 1e-5;
        // verifie que le point d'intersection est sur le mur et le faisceau
        bool condx = (x >= std::min(position[0],position[1]) && x <= std::max(position
```

```
    [0],position[1]) && x >= std::min(positionImage[0],positionCell[0]) && x <= std::max(positionImage[0],positionCell[0]));
        bool condy = (y >= std::min(position[2],position[3])  && y <= std::max(position[2],position[3]) && y >= std::min(positionImage[1],positionCell[1]) && y <= std::max(positionImage[1],positionCell[1]));

        if(condx && condy) {
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
/**
 * Verifie qu'il existe et calcule le point de transmission sur le mur.
 * @param pPosition : position des extremites du rayon
 * @param intersection : position du point de transmission sur le mur, passee par reference pour etre modifiee
 * @return 1 si ce point existe  sur le mur et 0 sinon
 */
int Wall::transmissionIntersectionPoint(std::array<double,4> pPosition, double * intersection) {
    // calcul le point d'intersection
    if(intersectionPoint(pPosition, intersection)) {
        double x = intersection[0]; double y = intersection[1];
        // epsilon qu'on ajoute ou enl ve pour se d barraser de l'erreur de repr sentation en base binaire
        double epsilon = 1e-5;
        // verifie que le point d'intersection est sur le mur et le faisceau
        bool condx = (x >= std::min(position[0],position[1]) - epsilon && x <= std::max(position[0],position[1]) + epsilon && x >= std::min(pPosition[0],pPosition[1]) - epsilon && x <= std::max(pPosition[0],pPosition[1]) + epsilon);
        bool condy = (y >= std::min(position[2],position[3]) - epsilon && y <= std::max(position[2],position[3]) + epsilon && y >= std::min(pPosition[2],pPosition[3]) - epsilon && y <= std::max(pPosition[2],pPosition[3]) + epsilon);
        if(condx && condy) {
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
/**
```

```
 * Assigne la valeur des attributs suivants du mur : le nombre d'onde, sa constante
 * de propagation et son impedance.
 * @param pF : frequence de la source
 */
void Wall::setImpedance(double pF)
{
    // nombre d'onde de la source
    beta = (2.0*pi*pF) / c;
    // epsilon tilde
    std::complex<double> esp_tild(eps, -sigma / (2*pi*pF));
    // constante de propagation du mur
    std::complex<double> j(0.0,1);
    gamma_m = (2*pi*pF) * sqrt(mu_0*esp_tild) * j;
    // impedance du mur
    Z = sqrt(mu_0 / esp_tild);
}
```

Listing 5 – Wall.cpp

Le fichier "utils.cpp" contient plusieurs fonctions permettant de gérer la création des sources images et le Ray-tracing.

```
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>
#include "utils.h"
#include "Source.h"
#include "Wall.h"
#include "Beam.h"
/**
 * Cree la liste des sources et y place les sources initiales
 * @param nbRef : nombre de reflexions
 * @param nbSourceinit : nombre de sources initiales
 * @param positionSourceInit : positions des sources initiales
 * @param listSources : liste des sources
 */
void initialiselistSources(int nbRef, int nbSourceinit, double *positionSourceInit,
    std::vector<std::vector<Source> > *listSources) {
    for(int i = 0; i<nbRef+1; i++) {
        // une liste pour les sources d'ordre i
        std::vector<Source> sources1;
        // ajout des sources initiales (ordre 0)
        if(i == 0) {
            for(int j = 0; j<nbSourceinit; j++) {
                double pSource[2] = {positionSourceInit[2*j], positionSourceInit[2*j
    +1]};
                sources1.push_back(Source(pSource, false));
```

```
            }
        }
        listSources->push_back(sources1);
    }
}
/**
 * Remplit la liste des sources avec les sources images
 * @param listWalls : liste des murs
 * @param listSources : liste des sources
 * @param nbRef : nombre de reflexions
 */
void createListSources(std::vector<Wall> *listWalls, std::vector<std::vector<Source> > *listSources, int nbRef) {
    for(int i = 0; i < nbRef; i++) {
        for(int j = 0; j<(*listSources)[i].size(); j++) {
            // pour chaque source, on calcule la source image de chaque mur
            for (Wall &wall : (*listWalls)) {
                // le mur par rapport pour lequel on cherche une source image doit etre different
                // du mur de la source initial -> le cas echeant on retombe sur la source parente
                if ((*listSources)[i][j].wallImage != &wall) {
                    // position de la source image
                    double image[2];
                    (*listSources)[i][j].imageSource(wall, image);
                    // on cree la source image
                    Source newSourceImage = Source(image, true);
                    newSourceImage.wallImage = &wall;
                    // remplit la liste de sources parentes
                    for (Source *parentSource : (*listSources)[i][j].listParentSources) {
                        newSourceImage.listParentSources.push_back(parentSource);
                    }
                    newSourceImage.listParentSources.push_back(&((*listSources)[i][j]));
                    // ajout a la liste de sources
                    (*listSources)[i+1].push_back(newSourceImage);
                }
            }
        }
    }
}
/**
 * Cree l'ensemble des rayons reflechis nR fois et les ajoute a la liste des rayons.
 * @param cell : recepteur
 * @param listWalls : liste des murs
 * @param listBeams : liste des rayons, passee par reference pour etre modifiee.
```

```
 * @param listSources : liste des sources
 * @param nR : nombre de reflexion
 */
void rayTracing(Cell *cell, std::vector<Wall> *listWalls, std::vector<Beam> *listBeams, std::vector<Source> *listSources, int nR) {
    // on itere pour chacune des sources images (d'ordre nR)
    for (Source &source : (*listSources)) {
        // intersection mur faisceau
        double intersection[2];
        // faisceau contenant tous les points intermediaires
        Beam beam = Beam();
        // check si il y a une intersection
        if (source.wallImage->reflexionIntersectionPoint(source.position, cell->position, intersection)) {
            // on ajoute les 2 premieres intersections
            beam.addR((std::array<double, 2>){cell->position[0], cell->position[1]});
            beam.addR((std::array<double, 2>){intersection[0], intersection[1]});
            // on verifie que la source n'est pas sur le mur -> le cas echeant l'intersection serait au meme endroit
            // que la source ce qui donnerait une valeur 'nan' pour angle_i
            if( (intersection[0] != cell->position[0]) && (intersection[1] != cell->position[1]) ) {
                // coefficient de reflexion
                std::complex<double> coeffR = 1;
                if(backTracing(&beam, &coeffR, source)) {
                    // angle incident
                    double angle_i = beam.angleCalculator(source.wallImage->position,0);
                    // on update le coeff de reflexion
                    beam.coeff *= coeffR;
                    beam.coeff *= source.wallImage->reflexionCoefficientCalculator(angle_i);
                    // on update le coeff de transmission pour chaque partie directe du rayon
                    for(int n = 0; n<beam.size-1; n++) {
                        beam.coeff *= beam.updateTransferCoefficient(*listWalls, n);
                    }
                    // ajout a la liste
                    (*listBeams).push_back(beam);
                } else {
                    continue;
                }
            }
        }
    }
}
/**
```

```
 * Verifie que par la methode des images que le chemin de reflexion est valide
 * et calcule le coefficient total.
 * @param beam : rayon reflechi
 * @param coeff : coefficient de reflexion
 * @param image : source image
 * @return : 1 si le chemin est valide et 0 sinon
 */
int backTracing(Beam *beam, std::complex<double> *coeff, Source image) {
    // intersection mur faisceau
    double inter1[2];
    // nbre reflexions
    int n = 1;
    // on itere sur toutes les sources parentes
    for(int i = image.listParentSources.size()-1; i>=0; i--) {
        // intersection mur faisceau
        double inter[2];
        // check si la source parente est une image
        if(image.listParentSources[i]->isImage) {
            double posInter[2] = {beam->listPositions[beam->size-1][0], beam->listPositions[beam->size-1][1]};
            // check si il y'a une intersection
            if (image.listParentSources[i]->wallImage->reflexionIntersectionPoint(image.listParentSources[i]->position, posInter, inter)) {
                // ajout des positions
                beam->addR((std::array<double, 2>){inter[0], inter[1]});
                // ckeck si intersection egale a l'intersection precedente
                if(inter[0] == posInter[0] && inter[1] == posInter[1]) {
                    return 0;
                }
                // angle incident
                double angle_i = beam->angleCalculator(image.listParentSources[i]->wallImage->position,beam->size-2);
                // coeff reflexion
                (*coeff) *= image.listParentSources[i]->wallImage->reflexionCoefficientCalculator(angle_i);
                n++;
            } else {
                return 0;
            }
        } else {
            beam->addR((std::array<double, 2>){image.listParentSources[i]->position[0], image.listParentSources[i]->position[1]});
        }
    }
    return 1;
```

```
}
```

Listing 6 – utils.cpp

La classe "Map.cpp" modélise la carte utilisée pour l'option 1 (Mode 2 ou enocre le calcul zone de couverture).

```
#include <iostream>
#include <fstream>
#include <vector>
#include "Map.h"
#include "Cell.h"

/**
 * Constructeur
 * @param pDiscretisation : precision choisie pour la discretisation
 * @param pWidth : largeur
 * @param pHeight : longueur
 */
Map::Map(double pDiscretisation, double pWidth, double pHeight) {
    // precision choisie pour la discr tisation
    discretisation = pDiscretisation;
    // largeur de la map
    width = pWidth;
    // longueur de la map
    height = pHeight;
    createMap();
}
/**
 * Cree la map en ajoutant un recepteur a chaque element de la matrice
 */
void Map::createMap() {
    // nbre de lignes / colonnes
    nbRow = floor((height/discretisation)) + 1;
    nbCol = floor((width/discretisation)) + 1;
    // pour chaque element de discretisation, on ajoute une cell
    for(int i = 0; i < nbRow; i++) {
        std::vector<Cell> vec;
        matrix.push_back(vec);
        for(int j = 0; j < nbCol; j++) {
            // position du recepteur
            double pPosition[2] = {j*discretisation, i*discretisation};
            Cell cell = Cell(pPosition);
            matrix[i].push_back(cell);
        }
    }
}
/**
```

```
 * Ouvre un fichier .txt pour y stocker les donnees de puissance en fonction de la position
 */
void Map::createMapFile() {
    std::ofstream file;
    file.open ("dataPower.txt");
    // Verifie que l'ouverture s'est deroulee correctement
    if(file) {
        for (int j = 0; j < nbCol; j++) {
            for(int i = 0; i < nbRow; i++) {
                file << matrix[i][j].position[0] << "\t" << matrix[i][j].position[1] << "\t" << matrix[i][j].P_dbm << std::endl;
            }
            file << "\n";
        }
    } else {
        std::cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << std::endl;
    }
}
```

Listing 7 – Map.cpp

La fonction "createListWalls" crée la liste a partir d'un fichier texte.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "createMap.h"
#include "Wall.h"

/**
 * Fonction qui remplit la liste des murs a l'aide d'un fichier .txt contenant les
 * caracteristiques des murs.
 * @param listWalls : liste des murs, passee par reference pour etre modifiee
 * @return un entier qui informe de l'ouverture correcte du fichier
 */
int createListWalls(std::vector<Wall> *listWalls) {
    //ouverture du fichier
    std::ifstream file;
    file.open ("wall.txt");
    // Verifie que l'ouverture s'est deroulee correctement
    if(file)
    {
        // stocke chacunes des lignes du fichier
        std::string line;
        // continue tant qu'il y'a encore une ligne a lire
```

```
        while(getline(file, line))
        {
            // liste des caracteristique d'un mur
            std::string list[7];
            double listCaracteristcs[7];
            // iterateur
            int i = 0;
            for (const char &c: line){
                if(c != '\\'){
                    list[i] += c;
                }
                else{
                    listCaracteristcs[i] = stod(list[i]);
                    i++;
                }
            }
            // positions du mur
            double positions[4] = {listCaracteristcs[0], listCaracteristcs[2], listCaracteristcs[1], listCaracteristcs[3]};
            // creation du mur
            Wall newWall(positions, listCaracteristcs[4], listCaracteristcs[5], listCaracteristcs[6]);
            // ajout a la liste des murs
            (*listWalls).push_back(newWall);
        }
        return 1;
    }
    else
    {
        std::cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << std::endl;
        return 0;
    }
    file.close();
}

int convertPowerFileToMbsFile(std::string fileName) {

    // matrice contenant toutes les cell
    std::vector<std::vector<std::array<double, 3> > > matrix;
    std::ifstream file;
    file.open (fileName);

    // Verifie que l'ouverture s'est deroulee correctement
    if(file)
    {
        bool newSet = true;
```

```
68          // stocke chacunes des lignes du fichier
69          std::string line;
70          // iterateur
71          int index = 0;
72          // continue tant qu'il y'a encore une ligne a lire
73          while(getline(file, line))
74          {
75              if(newSet) {
76                  newSet = false;
77                  std::vector<std::array<double, 3> > vec;
78                  matrix.push_back(vec);
79              }
80              // liste des caract ristique d'un mur
81              std::string list[3];
82              double listCaracteristcs[3];
83              // iterateur
84              int i = 0;
85              for (const char &c: line)
86              {
87                  if(c != '\t')
88                  {
89                          list[i] += c;
90                  }
91                  else
92                  {
93                      i++;
94                  }
95              }
96
97              if(line != "") {
98                  // on convertit les nombres  r  c u p  r  s  en double
99                  for (int n = 0; n < 3; n++) {
100                     listCaracteristcs[n] = stod(list[n]);
101                 }
102                 // on ajoute les valeurs    la matrice
103                 matrix[index].push_back((std::array<double, 3>){listCaracteristcs[0],
    listCaracteristcs[1], listCaracteristcs[2]});
104             } else {
105                 newSet = true;
106                 index ++;
107             }
108         }
109
110         // c r e  le fichier contenant les mb/s
111         createMbsFile(matrix);
112         file.close();
113         return 1;
```

```
    }
    else
    {
        std::cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << std::endl;
        return 0;
    }
}

void createMbsFile(std::vector<std::vector<std::array<double, 3> > > matrix) {
    std::ofstream file;
    file.open ("dataPowerMbs.txt");

    // Verifie que l'ouverture s'est deroulee correctement
    if(file) {
        for(int i = 0; i < matrix.size(); i++) {
            for(int j = 0; j < matrix[i].size(); j++) {
                // on convertit les watts en Mb/s
                double bitRate;
                if(matrix[i][j][2] > -73.0) {
                    bitRate = 320;
                } else if(matrix[i][j][2] < -82.0) {
                    bitRate = 40;
                } else {
                    double alpha = (320.0 - 40.0) / (-73.0 + 82.0);
                    bitRate = alpha*(matrix[i][j][2] + 73.0) + 320.0;
                }
                file << matrix[i][j][0] << "\t" << matrix[i][j][1] << "\t" << bitRate << "\n";
            }
            file << "\n";
        }
    } else {
        std::cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << std::endl;
    }
}
```

Listing 8 – createMap.cpp

La classe "Graphics.cpp" permet de gérer les modélisations graphiques

```
#include <iostream>
#include <fstream>
#include <string>
#include <array>
#include "Graphics.h"

```

```
/**
 * Constructeur
 */
Graphics::Graphics() {
    std::cout << "Cr ation de l'objet graphique" << std::endl;
}
/**
 * Ajoute la commande qui dessine une ligne d'extr mit s (x1,y1),(x2,y2)
 * et dont la couleur et la taille d pendent de l'argument (type)
 */
void Graphics::drawLine(double x1, double x2, double y1, double y2, int type) {
    std::string color;
    int size;
    switch(type) {
        case 0 :
            color = "grey";
            size = 3;
            break;
        case 100:
            color = "grey";
            size = 3;
            break;
        case 1:
            color = "green";
            size = 2;
            break;
        case 2:
            color = "blue";
            size = 1;
            break;
        case 3:
            color = "orange";
            size = 1;
            break;
        case 4:
            color = "red";
            size = 1;
            break;
        case 5:
            color = "pink";
            size = 1;
            break;
        case 6:
            color = "brown";
            size = 1;
            break;
        default:
```

```
            color = "black";
            size = 4;
    }
    std::string command = "set arrow nohead from " + std::to_string(x1) + "," + std::to_string(y1) + " to ";
    command += std::to_string(x2) +"," + std::to_string(y2) + " lc '" + color + "' lw " + std::to_string(size) + " front\n";
    std::cout << command;
    commandLine.push_back(command);
}
/**
 * Ajoute la commande qui dessine un point de coordonn es (x,y)
 * et dont la couleur et la taille d pendent de l'argument (type)
 */
void Graphics::addPoint(double x, double y, int type) {
    std::string color;
    double size;
    switch(type) {
        case 0:
            color = "green";
            size = 1.5;
            break;
        case 3:
            color = "black";
            size = 1.5;
            break;
        default:
            color = "black";
            size = 1;
    }
    std::string command = "set object circle at " + std::to_string(x) + "," + std::to_string(y) + " radius char ";
    command +=  std::to_string(size) + "fillstyle empty border lc rgb '"+ color + "' lw 2 front\n";
    commandLine.push_back(command);
}
/**
 * Permet l'affiche du graphe a l'aide de gnuplot en creant un fichier
 * de commandes et en le remplissant.
 * @param width : largeur
 * @param height : hauteur
 */
void Graphics::printGraph(double width, double height) {

    //Cr ation du fichier de commandes
    FILE *gnuplot = popen("gnuplot -persistent", "w");
    if(gnuplot == NULL) {
```

```
        //printf("\n-----ERREUR-----\nIMPOSSIBLE D'OUVRIR LE FICHIER DANS LA CREATION DU FICHIER DE COMMANDE POUR GNUPLOT\n");
    }

    fprintf(gnuplot, "reset\n");
    fprintf(gnuplot, "set terminal qt size %d, %d font 'Helvetica,15'\n", (int)((width/height)*600), 600);
    fprintf(gnuplot, "unset key\n");
    fprintf(gnuplot, "set xlabel 'x (m)'\n");
    fprintf(gnuplot, "set ylabel 'y (m)'\n");
    fprintf(gnuplot, "set xrange[%f:%f]\n", 0.0, width);
    fprintf(gnuplot, "set yrange[%f:%f]\n", 0.0, height);
    for(int i = 0; i < commandLine.size(); i++) {
        fprintf(gnuplot, "%s", commandLine[i].c_str());
    }
    fprintf(gnuplot, "plot 1/0\n");
    fclose(gnuplot);
}

/**
 * Permet l'affiche de la Map a l'aide de gnuplot en creant un fichier
 * de commandes et en le remplissant.
 * @param width : largeur
 * @param height : hauteur
 */
void Graphics::printMap(double width, double height) {
    //Cr ation du fichier de commandes
    FILE *gnuplot = popen("gnuplot -persistent", "w");
    if(gnuplot == NULL) {
        //printf("\n-----ERREUR-----\nIMPOSSIBLE D'OUVRIR LE FICHIER DANS LA CREATION DU FICHIER DE COMMANDE POUR GNUPLOT\n");
    }
    std::cout << "test : " << (width/height)*600 << std::endl;
    fprintf(gnuplot, "reset\n");
    fprintf(gnuplot, "set terminal qt size %d, %d font 'Helvetica,15'\n", (int)((width/height)*600), 600);
    fprintf(gnuplot, "unset key\n");
    fprintf(gnuplot, "set xlabel 'x (m)'\n");
    fprintf(gnuplot, "set ylabel 'y (m)'\n");
    fprintf(gnuplot, "set xrange[%f:%f]\n", 0.0, width);
    fprintf(gnuplot, "set yrange[%f:%f]\n", 0.0, height);
    fprintf(gnuplot, "set cbrange[%f:%f]\n", -82.0, -73.0);
    fprintf(gnuplot, "set pm3d map\n");
    //fprintf(gnuplot, "set palette defined ( 0 '#FF8000', 1 '#FFFF00', 2 '#80FF00', 3 '#00FF00', 4 '#00FF80', 5 '#00FFFF', 6 '#0080FF', 7 '#0000FF', 8 '#7F00FF', 9 '#FF00FF', 10 '#FF007F', 11 '#FF0000' )\n");
    //fprintf(gnuplot, "set palette rgbformulae 22,13,-31\n");
```

```
    for(int i = 0; i < commandLine.size(); i++) {
        fprintf(gnuplot, "%s", commandLine[i].c_str());
    }
    fprintf(gnuplot, "splot 'dataPower.txt' using 1:2:3 with pm3d\n");
    //fprintf(gnuplot, "plot 1/0\n");
    //fprintf(gnuplot, "splot 'dataPowerFinal1.txt' using 1:2:($3 >= -83 ? $3 : 'none') with pm3d\n");
    fclose(gnuplot);
}

/**
 * Permet l'affiche de la Map en Mbps a l'aide de gnuplot en creant un fichier
 * de commandes et en le remplissant.
 * @param width : largeur
 * @param height : hauteur
 */
void Graphics::printMapMbps(double width, double height) {
    //Cr ation du fichier de commandes
    FILE *gnuplot = popen("gnuplot -persistent", "w");
    if(gnuplot == NULL) {
        //printf("\n-----ERREUR-----\nIMPOSSIBLE D'OUVRIR LE FICHIER DANS LA CREATION DU FICHIER DE COMMANDE POUR GNUPLOT\n");
    }
    std::cout << "test : " << (width/height)*600 << std::endl;
    fprintf(gnuplot, "reset\n");
    fprintf(gnuplot, "set terminal qt size %d, %d font 'Helvetica,15'\n", (int)((width/height)*600), 600);
    fprintf(gnuplot, "unset key\n");
    fprintf(gnuplot, "set xlabel 'x (m)'\n");
    fprintf(gnuplot, "set ylabel 'y (m)'\n");
    fprintf(gnuplot, "set xrange[%f:%f]\n", 0.0, width);
    fprintf(gnuplot, "set yrange[%f:%f]\n", 0.0, height);
    fprintf(gnuplot, "set cbrange[%f:%f]\n", 40.0, 320.0);
    fprintf(gnuplot, "set pm3d map\n");
    //fprintf(gnuplot, "set palette defined ( 0 '#FF8000', 1 '#FFFF00', 2 '#80FF00', 3 '#00FF00', 4 '#00FF80', 5 '#00FFFF', 6 '#0080FF', 7 '#0000FF', 8 '#7F00FF', 9 '#FF00FF', 10 '#FF007F', 11 '#FF0000' )\n");
    //fprintf(gnuplot, "set palette rgbformulae 22,13,-31\n");
    for(int i = 0; i < commandLine.size(); i++) {
        fprintf(gnuplot, "%s", commandLine[i].c_str());
    }
    fprintf(gnuplot, "splot 'dataPowerMbs.txt' using 1:2:3 with pm3d\n");
    //fprintf(gnuplot, "plot 1/0\n");
    //fprintf(gnuplot, "splot 'dataPowerFinal1.txt' using 1:2:($3 >= -83 ? $3 : 'none') with pm3d\n");
    fclose(gnuplot);
```

```
}
```

Listing 9 – Graphics.cpp

La classe "Chromosome.cpp" modélise un chromosome de l'algorithme génétique

```
#include "Chromosome.h"
#include <vector>
#include <array>

/**
 * Constructeur
 * @param pNbSources : nombre de sources que poss de le chromosome
 * @param pPositions
 */
Chromosome::Chromosome(int pNbSources, std::vector<std::array<double,2> > pPositions)
    {
    // nombre de source que poss de le chromosome
    nbSources = pNbSources;
    // zone de couverture totale du chromosome
    area = 0;
    // intersection entre les sources du chromosome
    overlap = 0;
    // positions des sources du chromosome
    positions = pPositions;
}

/**
 * cette fonction met a jour la zone de couverture et l'overlap du chromosome
 * @param pPosition : position en laquel la puissance est superieure a -82[dbm]
 * @param pArea : aire ajoutee
 */
void Chromosome::updateCoverage(std::array<double,2>  pPosition, double pArea) {
    // booleen -> true si la cellule n'est pas deja couverte par une autre source du
    chromosome
    bool check = true;

    // on verifie que les autres sources du chromosomes ne couvre pas deja cette
    cellule
    for(int i = 0; i < listCells.size(); i++) {
        if (listCells[i][0] == pPosition[0] && listCells[i][1] == pPosition[1]) {
            check = false;
            overlap += 1;
        }
    }
    // sinon on ajoute l'aire a la zone de couverture du chromosome
    if(check) {
        area += 1;
```

```
        listCells.push_back(pPosition);
    }
}
```

Listing 10 – Chromosome.cpp

La classe "Optimisation.cpp" implémente l'algorithme génétique

```
#include <iostream>
#include <vector>
#include "Optimization.h"
#include "Wall.h"
#include "Beam.h"
#include "Source.h"
#include "utils.h"
#include "Chromosome.h"

/**
 * Constructeur
 * @param pListWalls : liste des murs sur la map
 * @param pNbRef : nombre de reflexons maximales
 * @param pDiscretisation : discretisation de la map
 * @param pWidth : largeur de la map
 * @param pHeight : hauteur de la map
 */
Optimization::Optimization(std::vector<Wall>  &pListWalls, int pNbRef, int
    pDiscretisation, double pWidth, double pHeight) {

    // liste des murs present sur la map
    listWalls = pListWalls;
    // nombre de reflexions maximales utilisees par l'algorithme
    nbRef = pNbRef;
    // largeur de la map
    width = pWidth;
    // hauteur de la map
    height = pHeight;
    // discretisation de la map
    discretisation = pDiscretisation;
    // probabilite pour le croisement uniforme
    Pc = 0.7;
    // probabilite pour la mutation
    Pm = 0.4;
    // numero de la generation
    generation = 0;
    // on calcul l'aire totale de la map en fonction de la discretistation
    double nbRow = floor((height/discretisation)) + 1;
    double nbCol = floor((width/discretisation)) + 1;
    totalArea = nbRow*nbCol;
```

```
}

/**
 * Cette fonction permet de passer d'une generation a une autre
 * cela ce fait en trois etapes :
 * le croisement, la mutation et la selection
 */
void Optimization::updateGeneration() {

    /* --- CROISEMENT --- */
    // probabilite pour le croisement a un point
    double Po = 1 - Pc;
    // conversion de la probabilite Pc en un nombre entier de chromosomes
    int nb_Pc = (int)(Pc*(double)(listChromosome[generation].size()));
    if(nb_Pc%2 == 1) { nb_Pc -= 1; }
    // conversion de la probabilite 1 - Pc en un nombre entier de chromosomes
    int anti_Pc = (int)(Po*(double)(listChromosome[generation].size()));
    if(anti_Pc%2 == 1) { anti_Pc += 1; }
    // conversion de la probabilite Pm en un nombre entier de chromosomes
    int nb_Pm = (int)(Pm*(double)(listChromosome[generation].size()));
    if(nb_Pc%2 != 1) { nb_Pc += 1; }

    // on realise le croisement uniforme sur les chromosomes
    for (int i = 0; i < nb_Pc/2; i++) {
        uniformCrossover(&listChromosome[generation][2*i], &listChromosome[generation
    ][2*i+1]);
    }
    // on realise le croisement a un point sur les chromosomes
    for (int i = 0; i < anti_Pc/2; i++) {
        onePointCrossover(&listChromosome[generation][2*i], &listChromosome[generation
    ][2*i+1]);
    }

    /* --- MUTATION --- */
    // on realise la mutation sur les chromosomes
    for (int i = 0; i < nb_Pm; i++) {
        mutation(&listChromosome[generation][i]);
    }

    // on melange l'ensemble des chromosomes avant de les selectionner
    shuffle(&listChromosome[generation]);

    /* --- HELITISME --- */
    // on reporte le meilleur chromosome de la generation precedente
    // a une position aleatoire dans la generation suivante
    int posRand = rand() % (listChromosome[generation].size());
    listChromosome[generation][posRand] = Chromosome(best.nbSources, best.positions);
```

```
    // on calcul la zone couverture pour chacun des chromosomes
    initialiseChromosome();

    /* --- SELECTION --- */
    // valeur moyenne de la zone de couverture pour la generation
    double mean = 0;
    // on it re sur tous les chromosomes pour calculer la zone de couverture moyenne
    for(int i = 0; i < listChromosome[generation].size(); i++) {
        // cette condition initialise le meilleur chromosome a la premi re generation
        if(generation == 0 && i == 0) {
            best = Chromosome(listChromosome[generation][i].nbSources, listChromosome[generation][i].positions);
            best.area = listChromosome[generation][i].area;
            best.overlap = listChromosome[generation][i].overlap;
        }
        mean += listChromosome[generation][i].area/totalArea;
    }
    // affichage console
    std::cout << "moyenne de l'aire de la generatioin " << generation << " = " << mean << " et : " << mean/listChromosome[generation].size() << std::endl;

    // on augmente le numero de la generation
    generation += 1;
    // on selectionne les chromosomes en les faisant s'affronter 2 par 2
    for (int i = 0; i <  listChromosome[generation-1].size()/2; i++) {
        std::vector<Chromosome> vInit;
        listChromosome.push_back(vInit);
        Chromosome newChro = selection(listChromosome[generation-1][2*i], listChromosome[generation-1][2*i+1]);
        listChromosome[generation].push_back(newChro);
    }

    // affichage console
    std::cout << "---- BEST DE LA GENERATION : aire = " << best.area/totalArea << " - overlap =  " << best.overlap/best.area << " - nbSources = " << best.nbSources << " - listCells size = " << best.listCells.size() << std::endl;
    for (int i = 0; i < best.nbSources; i++) {
        std::cout << "source : " << i << " - posx : " << best.positions[i][0] << " - posy : " << best.positions[i][1] << std::endl;
    }
}

/**
 * cette fonction prend une liste de chromosomes en entree
 * et retourne cette  m me  liste melangee
 * @param listBis : liste de chromosomes a melanger
```

```
 */
void Optimization::shuffle(std::vector<Chromosome> *listBis) {
    std::vector<Chromosome> tempList = *listBis;
    for (int i = 0; i < (*listBis).size(); i++) {
        (*listBis)[i] = tempList[tempList.size()-(i+1)];
    }
}

/**
 * cette fonction permet d'initialiser la premi re
 * generation de chromosomes
 * @param nbChromsome : nombre de chromosomes dans la generation
 */
void Optimization::initialisation(int nbChromsome) {
    // on cree tous les chromosomes et on les ajoutes a la liste initiale
    for (int i = 0; i < nbChromsome; i++) {
        // on gen re un nombre aleatoire de sources initiales
        // -> compris entre 2 et 4
        int nRand = rand() % (3) + 2;
        std::vector<std::array<double,2> > listPositions;
        for (int j = 0; j < nRand; j++) {
            // on gen re une position aleatoire selon x et y pour chaque source
            double xRand = rand() % ((int)(width) + 1);
            double yRand = rand() % ((int)(height) + 1);
            listPositions.push_back((std::array<double,2>){xRand, yRand} );
        }
        std::vector<Chromosome> vInit;
        listChromosome.push_back(vInit);
        // on ajoute le chromosome a la liste initiale
        listChromosome[generation].push_back(Chromosome(nRand, listPositions));
    }
}

/**
 * Cette fonction prend deux chromosomes en entree
 * et renvoie le meilleur des deux
 * @param chromosome1
 * @param chromosome2
 * @return
 */
Chromosome Optimization::selection(Chromosome &chromosome1, Chromosome &chromosome2) {
    // on definit les poids pour les differents param tres de selection :
    // -> zone de couverture, intersection des aires, et nombre de sources
    double weightArea = 1, weightOverlap = 0.1, weightNbSources = 0.1;
    // calcul de la performance des deux chromosomes
    double comparateur1 = weightArea*(chromosome1.area/totalArea) - weightOverlap*(
    chromosome1.overlap/chromosome1.area) - weightNbSources*((double)chromosome1.
```

```
    nbSources);
    double comparateur2 = weightArea*(chromosome2.area/totalArea) - weightOverlap*(chromosome2.overlap/chromosome2.area) - weightNbSources*((double)chromosome2.nbSources);
    // calcul de la performance du meilleur chromosome actuel de la generation
    double comparateurBest = weightArea*(best.area/totalArea) - weightOverlap*(best.overlap/best.area) - weightNbSources*((double)best.nbSources);

    // on verifie que les deux chromosomes ne sont pas plus performants que le
    // meilleur chromosome actuel -> sinon on change le meilleur chromosome
    if(comparateurBest <= comparateur1) {
        best = Chromosome(chromosome1.nbSources, chromosome1.positions);
        best.area = chromosome1.area;
        best.overlap = chromosome1.overlap;
    }
    if(comparateurBest <= comparateur2) {
        best = Chromosome(chromosome2.nbSources, chromosome2.positions);
        best.area = chromosome2.area;
        best.overlap = chromosome2.overlap;
    }

    // affichage console
    std::cout << "----- SELECTION -----" << std::endl;
    std::cout << "chromosome 1 : aire = " << chromosome1.area/252 << " - overlap =  " << chromosome1.overlap/chromosome1.area << " - nbSources = " << weightNbSources*((double)chromosome1.nbSources) << " - listCells size = " << chromosome1.listCells.size() << std::endl;
    for (int i = 0; i < chromosome1.nbSources; i++) {
        std::cout << "source : " << i << " - posx : " << chromosome1.positions[i][0] << " - posy : " << chromosome1.positions[i][1] << std::endl;
    }
    std::cout << "chromosome 2 : aire = " << chromosome2.area/252 << " - overlap =  " << chromosome2.overlap/chromosome2.area << " - nbSources = " << weightNbSources*((double)chromosome2.nbSources) << " - listCells size = " << chromosome2.listCells.size() << std::endl;
    for (int i = 0; i < chromosome2.nbSources; i++) {
        std::cout << "source : " << i << " - posx : " << chromosome2.positions[i][0] << " - posy : " << chromosome2.positions[i][1] << std::endl;
    }

    // on retourne le chromosome le plus performant
    if(comparateur1 > comparateur2) {
        return Chromosome(chromosome1.nbSources, chromosome1.positions);
    } else {
        return Chromosome(chromosome2.nbSources, chromosome2.positions);
    }
}
```

```
/**
 * cette fonction realise le croisement uniforme
 * @param chromosome1
 * @param chromosome2
 */
void Optimization::uniformCrossover(Chromosome *chromosome1, Chromosome *chromosome2) {
    // on gen re une source aleatoirement dans chacun des chromosomes
    int c1 = rand() % (int)(chromosome1->nbSources);
    int c2 = rand() % (int)(chromosome2->nbSources);

    // on intervertit les positions selon x et y l'un avec l'autre
    chromosome1->positions[c1][1] = chromosome2->positions[c2][1];
    chromosome2->positions[c2][0] = chromosome1->positions[c1][0];
}

/**
 * cette fonction realise le croisement a un point
 * @param chromosome1
 * @param chromosome2
 */
void Optimization::onePointCrossover(Chromosome *chromosome1, Chromosome *chromosome2) {
    // on verifie que les chromosomes poss dent plus d'une source
    // -> sinon  a  n'est la peine de faire le croisement (nb : ce cas est t r s  rare)
    if(chromosome1->nbSources > 1 && chromosome2->nbSources > 1) {
        // on gen re une position aleatoire en lequel on coupe les deux chromosomes
        int c1 = rand() % (chromosome1->nbSources-1) + 1;
        int c2 = rand() % (chromosome2->nbSources-1) + 1;

        // on decoupe chacun des chromosomes en deux
        std::vector<std::array<double,2> > copyFirst1 = slice(chromosome1->positions, 0, c1);
        std::vector<std::array<double,2> > copyLast1 = slice(chromosome1->positions, c1, chromosome1->nbSources);
        std::vector<std::array<double,2> > copyFirst2 = slice(chromosome2->positions, 0, c2);
        std::vector<std::array<double,2> > copyLast2 = slice(chromosome2->positions, c2, chromosome2->nbSources);

        // on fusionne les differentes parties ensemble pour croiser les chromosomes entre eux
        chromosome1->positions.clear();
        copy(copyFirst1, copyLast2, &(chromosome1->positions));
        chromosome1->nbSources = chromosome1->positions.size();
        chromosome2->positions.clear();
```

```
        copy(copyFirst2, copyLast1, &(chromosome2->positions));
        chromosome2->nbSources = chromosome2->positions.size();
    }
}

/**
 * cette fonction realise la mutation sur un chromosome
 * @param chromosome
 */
void Optimization::mutation(Chromosome *chromosome) {
    // on selectionne une source du chromosome aleatoirement
    int index = rand() % (chromosome->nbSources);
    // on gen re une valeurs aleatoire selon x et y
    // dans les limites des dimensions de la map
    int xRand = rand() % ((int)(width)+1) - chromosome->positions[index][0];
    int yRand = rand() % ((int)(height)+1) - chromosome->positions[index][1];

    // on ajoute les valeurs aleatoires a la position de la source
    chromosome->positions[index][0] += xRand;
    chromosome->positions[index][1] += yRand;
}

/**
 * cette fonction prend un vecteur en enree
 * et rentourne un vecteur possedant ses valeurs
 * entre l'indice m et n
 * @param v : vecteur a decouper
 * @param m : position de depart de la decoupe
 * @param n : position finale de la decoupe
 * @return : retourne le vecteur decoupe a partir de v
 */
std::vector<std::array<double,2> > Optimization::slice(std::vector<std::array<double
    ,2> > const &v, int m, int n) {
    std::vector<std::array<double,2> > vec;
    for(int i = 0; m+i < n; i++) {
        vec.push_back(v[m+i]);
    }
    return vec;
}

/**
 * cette fonction fusionne deux vecteurs v1 et v2 entre
 * eux pour former un nouveau vecteur vout
 * @param v1 : premi re partie a fusionner
 * @param v2 : seconde partie a fusionner
 * @param vOut : vecteur resultant de la fusion de v1 et v2
 */
```

```
void Optimization::copy(std::vector<std::array<double,2> > const &v1, std::vector<std
    ::array<double,2> > const &v2, std::vector<std::array<double,2> > *vOut) {
    for(int i = 0; i < v1.size(); i++) {
        vOut->push_back(v1[i]);
    }
    for(int i = 0; i < v2.size(); i++) {
        vOut->push_back(v2[i]);
    }
}

/**
 * Cette fonction initialise les chromosomes avant
 * de faire la selection
 */
void Optimization::initialiseChromosome() {
    for (int i = 0; i < listChromosome[generation].size(); i++) {
        for (int j = 0; j < listChromosome[generation][i].nbSources; j++) {
            coverageArea(listChromosome[generation][i].positions[j], &(listChromosome[
    generation][i]));
        }
        std::cout << "avancement " << i << " / " << listChromosome[generation].size()
    << std::endl;
    }
}

/**
 * Cette fonction calcule la zone de couverture d'une source pour un chromosome
 * -> elle est optimisee de sorte a ne pas devoir calculer la puissance
 *  captee sur l'entierete de la map
 * @param positionSourceInit : position de la source dont on calcul la zone de
    couverture
 * @param chromosome : chromosome dont on calcul la zone de couverture
 */
void Optimization::coverageArea(std::array<double,2> positionSourceInit, Chromosome *
    chromosome) {
    // liste des sources / source image
    std::vector<std::vector<Source> > listSources;
    // on initialise la liste des sources
    double posInit[2] = {positionSourceInit[0], positionSourceInit[1]};
    initialiselistSources(nbRef, 1, posInit, &listSources);
    // on cree la liste des sources images
    createListSources(&listWalls, &listSources, nbRef);
    // surface totale calculee
    double sUnitaire = pow(discretisation, 2);
    // conditions pour limiter la limite de surface calculee selon l'axe y
    int cond1y = 0, cond2y = 0, cond3y = 0, cond4y = 0;
    double x = listSources[0][0].position[0] - (int)listSources[0][0].position[0] %
```

```
discretisation;
double y = listSources[0][0].position[1] - (int)listSources[0][0].position[1] %
discretisation;
// on it re sur la map pour calculer la zone de couverture
for(int i = 0; i <= height; i += discretisation) {
    // conditions pour limiter la limite de surface calculee selon l'axe x
    int cond1x = 0, cond2x = 0, cond3x = 0, cond4x = 0;
    for (int j = 0; j <= width; j += discretisation) {
        // conditions permettant de verifier que la position n'est pas en dehors
        // de la map et de ne pas calculer plusieurs fois une m me zone
        bool check1 = false, check2 = false, check3 = false, check4 = false;
        // position du recepteur
        std::array<double,2>  pCell1, pCell2, pCell3, pCell4;

        // on initialise les conditions
        if(i == 0 && j == 0) {
            pCell1 = (std::array<double,2>){x+j,y+i};
            check1 = true;
        } else if(j == 0) {
            if(y+i <= height && cond1x < 1) {
                pCell1 = (std::array<double,2>){x+j,y+i};
                check1 = true;
            }
            if(y-i >= 0 && cond2x < 1) {
                pCell2 = (std::array<double,2>){x+j,y-i};
                check2 = true;
            }
        } else if(i == 0) {
            if(x+j <= width && cond1x < 1) {
                pCell1 = (std::array<double,2>){x+j,y+i};
                check1 = true;
            }
            if(x-j >= 0 && cond3x < 1) {
                pCell3 = (std::array<double,2>){x-j,y-i};
                check3 = true;
            }
        } else {
            if(x+j <= width && y+i <= height && cond1x < 1) {
                pCell1 = (std::array<double,2>){x+j,y+i};
                check1 = true;
            }
            if(x+j <= width && y-i >= 0 && cond2x < 1) {
                pCell2 = (std::array<double,2>){x+j,y-i};
                check2 = true;
            }
            if(x-j >= 0 && y-i >= 0 && cond3x < 1) {
                pCell3 = (std::array<double,2>){x-j,y-i};
```

```
                    check3 = true;
                }
                if(x-j >= 0 && y+i <= height && cond4x < 1) {
                    pCell4 = (std::array<double,2>){x-j,y+i};
                    check4 = true;
                }
            }

            // on calcule la zone de couverture dans 4 directions differentes en m me
    temps
            if(check1) {
                if(calculatePower(listSources, pCell1) > -82.0) {
                    chromosome->updateCoverage(pCell1, sUnitaire);
                } else {
                    cond1x += 1;
                    if(j == 0) { cond1y += 1; }
                }
            }
            if(check2) {
                if(calculatePower(listSources, pCell2) > -82.0) {
                    chromosome->updateCoverage(pCell2, sUnitaire);
                } else {
                    cond2x += 1;
                    if(j == 0) { cond2y += 1; }
                }
            }
            if(check3) {
                if(calculatePower(listSources, pCell3) > -82.0) {
                    chromosome->updateCoverage(pCell3, sUnitaire);
                } else {
                    cond3x += 1;
                    if(j == 0) { cond3y += 1; }
                }
            }
            if(check4) {
                if(calculatePower(listSources, pCell4) > -82.0) {
                    chromosome->updateCoverage(pCell4, sUnitaire);
                } else {
                    cond4x += 1;
                    if(j == 0) { cond4y += 1; }
                }
            }
            if(!check1 && !check2 && !check3 && !check4) {
                break;
            }
        }
        if(cond1y >= 1 && cond2y >= 1 && cond3y >= 1 && cond4y >= 1) {
```

```
            break;
        }
    }
}

/**
 * cette fonction calcul la puissance captee a la position pCell
 * @param listSources : liste des sources images
 * @param pCell : position en laquelle on calcul la puissance
 * @return : retounre la puissance captee
 */
double Optimization::calculatePower(std::vector<std::vector<Source> > listSources, std
    ::array<double,2>  pCell) {
    // creation recepteur
    double positionCell[2] = {pCell[0], pCell[1]};
    Cell *cell = new Cell(positionCell);
    // liste des faisceaux
    std::vector<Beam> listBeams;
    // onde directe
    for(int i = 0; i < listSources[0].size(); i++) {
        listSources[0][i].directWave(cell, &listWalls, &listBeams,i);
    }
    // reflexions
    for(int i = 0; i < nbRef; i++) {
        rayTracing(cell, &listWalls, &listBeams, &listSources[i+1], i);
    }
    // calcul puissance captee
    for(Beam &beam : listBeams) {
        beam.electricField(listSources[0][0].beta, listSources[0][0].G, listSources
    [0][0].P_watt);
        cell->updatePower(listSources[0][0].R_a, listSources[0][0].h_e, beam.E);
    }
    return cell->P_dbm;
}
```

Listing 11 – Optimisation.cpp