ECOLE
POLYTECHNIQUE
DE BRUXELLES

ELEC-H304 Rapport de Projet: Réalisation d'un logiciel de ray-tracing Wi-Fi IEEE 802.11ay

Lucas Placentino
Salman Houdaibi

Cours:
ELEC-H304 — Philippe De Doncker

Année académique:
2023-2024

Table des matières

Introduction

Ce rapport présente le projet de *ray tracing* donné dans le cadre du cours d'ELECH304 - Physique des Télécommunications, cours donné en troisième année de bachelier en cursus d'ingénieur civil en option électronique et télécommunication et en option physique. L'objectif du projet est de réaliser un code de calcul qui permet d'analyser la couverture réseau d'antennes Wi-Fi IEEE 802.11ay. Celle-ci est simulée via ray-tracing en utilisant la méthode des images. Plusieurs simplifications physiques et géométriques sont détaillées puis prises en compte dans la réalisation. L'espace dans lequel la simulation est faite est un appartement composé de plusieurs pièces, avec des murs de matériau différents et donc de propriétés physiques différentes.

Une application dotée d'une interface graphique a été réalisée afin de simplifier le lancement et la paramétrisation de la simulation.

L'objectif final est d'observer la couverture du réseau dans l'appartement entier, afin d'ensuite optimiser le placement de la station de base pour maximiser cette couverture à une vitesse la plus élevée possible.

Chapitre 1

Modélisation

1.1 Hypothèses physiques

1.1.1 Hypothèses de départ

Pour simplifier la réalisation du code, plusieurs simplifications géométriques et physiques ont été prises en compte.

Spécificités du réseau

Notre couverture de base est assurée par une antenne dipôle $\lambda/2$, celle-ci sera considérée sans perte. Elle émettra avec une fréquence de 60 GHz (suivant la norme Wi-Fi IEEE 802.11 ay) et avec une puissance de 20 dBm. Pour la réception, nous considérons un seuil minimal de -90 dBm et un seuil maximal de -40 dBm. Les débits binaires correspondants valent respectivement 50 Mb/s et 40 Gb/s.

Géométrie du problème

Nous ne prendrons en compte que deux dimensions de l'espace, c'est à dire que nous considérons que les antennes sont placées à une même hauteur sur l'axe z et n'émettent et ne recoivent que dans la dimension transverse à celui-ci. Nous ne considérons donc que les dimensions (x, y) de nos murs.

Simulation de la couverture de base

L'étude de la couverture de base de la borne Wi-Fi se fait via la méthode des images (détaillée plus loin) qui nous permettra de tracer le trajet des rayons entre l'antenne émettrice et l'antenne réceptrice. On considérera les rayons transmis à travers les murs, et les ondes réfléchies avec un maximum de deux réflexions. On ne prendra pas en compte la diffraction des ondes.

1.1.2 Hypothèse de champ lointain

Afin d'utiliser nos méthodes de tracé de rayon, nous allons déterminer dans quel cadre théorique on se place. Les dimensions de notre problème nous imposent de travailler sous les hypothèses de *champ lointain*.

La frontière de champ lointain est définie avec

$$r_f = \max\left\{1.6\lambda, 5D, \frac{2D^2}{\lambda}\right\}$$

avec D la dimension du circuit émetteur et λ la longueur d'onde déduite de la fréquence de travail f. Nos émetteurs sont des émetteurs de type $\lambda/2$, leur dimension vaut donc approximativement $\lambda/2$. Notre fréquence de travail est celle du Wi-Fi IEEE 802.11ay et donc $f = 60\mathrm{GHz}$.

On peut donc évaluer λ via

$$\lambda = \frac{c}{f} = 5 \cdot 10^{-3}\mathrm{m}$$

On en déduit que $r_f = 5D = 1.25\mathrm{cm}$ et que donc notre approximation est donc tout à fait valable car notre espace est découpé en carrés de 0.5m de largeur.

1.1.3 Propriétés des antennes $\lambda/2$

Les propriétés des antennes dipôles, dites $\lambda/2$, ont été étudiées au cours, dans le chapitre relatif aux antennes, d'où nous déduirons nos équations. Dans notre cas, l'antenne n'a pas de pertes et une résistance $R_a = 73\Omega$. Placée sur l'axe z, sa hauteur équivalente s'exprime comme

$$\vec{h}_e(\theta, \phi) = -\frac{\lambda \cos\left(\frac{\pi}{2}\cos\theta\right)}{\pi \sin^2\theta}\vec{1}_z$$

Cependant, comme énoncé plus haut, nous n'allons considérer que deux dimensions de l'espace, et donc nous prendrons $\theta = \frac{\pi}{2}$.

$$\vec{h}_e\left(\frac{\pi}{2}, \phi\right) = -\frac{\lambda}{\pi}\vec{1}_z$$

Nous prendrons cette valeur en compte dans le calcul des puissances.

1.2 Propagation des ondes

Les formules de cette section sont issues du cours, principalement du chapitre relatif aux calculs de coefficients de réflexion, transmission et diffraction.

1.2.1 Propriétés des murs

Les champs se propageront soit dans l'air (assimilé au vide) soit en traversant et/ou en se réfléchissant sur des murs. Pour calculer les effets relatifs à ces interactions entre les murs et les ondes, on utilisera leurs propriétés électriques. Ces propriétés sont données par la constante de propagation dans un matériau

$$\gamma_m = \alpha_m + j\beta_m = j\omega\sqrt{\mu_0\tilde{\varepsilon}}$$

avec $\tilde{\varepsilon}$ la permittivité complexe ($\tilde{\varepsilon} = \varepsilon_r\varepsilon_0 - j\frac{\sigma}{\omega}$, où ε_r est la permittivité relative, et σ la conductivité) du matériau, ω la pulsation de l'onde, et μ_0 la perméabilité du vide, repris dans ce tableau [Table 1.1] :

Matériau	ε_r	σ [S/m]
brique	3.95	0.073
béton	6.4954	1.43
cloison	2.7	0.05346
vitre	6.3919	0.00107
paroi métallique	1	10^7

TABLE 1.1 – Propriétés des différents matériaux [PYP08]

1.2.2 Réflexions et transmissions

Angle transmis

Pour calculer l'angle transmis θ_t à l'interface entre deux matériaux on utilise la loi de Snell-Descartes qui s'exprime via

$$\sin\theta_t = \sqrt{\frac{\varepsilon_{r1}}{\varepsilon_{r2}}}\sin\theta_i$$

avec θ_i l'angle incident, et ε_r la permittivité relative. En pratique, nous assimilerons l'air au vide et prendrons donc $\varepsilon_{r1} = 1$ pour celui-ci.

Impédance

Pour évaluer nos coefficients de transmission et de réflexion, on définit l'impédance d'un matériau Z_m comme suit

$$Z_m = \sqrt{\frac{\mu}{\tilde{\varepsilon}}} = \sqrt{\frac{\mu_0}{\varepsilon - j\frac{\sigma}{\omega}}}$$

avec μ_0 la perméabilité du vide, ε la permittivité du matériau ($\varepsilon = \varepsilon_r\varepsilon_0$), σ sa conductivité et ω la pulsation de l'onde électromagnétique.

Coefficient de transmission

Le coefficient de transmission à travers un obstacle se calcule avec cette formule

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

où $\Gamma_\perp$ est le coefficient de réflexion de la polarisation perpendiculaire, qui se calcule comme suit

$$\Gamma_\perp(\theta_i) = \frac{Z_m\cos\theta_i - Z_0\cos\theta_t}{Z_m\cos\theta_i + Z_0\cos\theta_t}$$

Coefficient de réflexion

Le coefficient de réflexion se calcule comme suit :

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

où $\Gamma_\perp$ se calcule de la même manière que pour le coefficient de transmission.

1.2.3 Directivité et gain

Les antennes n'émettent pas de manière isotrope. Leur efficacité dépend de la direction dans laquelle elles émettent/reçoivent l'onde. Pour quantifier cela, on fait appel à un coefficient appelé le gain d'antenne $G(\theta,\phi)$, qui défini à partir de l'efficacité d'antenne η et de sa directivité $D(\theta,\varphi)$ (définie comme le quotient de son intensité rayonnée dans une direction U et de son intensité rayonnée moyenne $P/4\pi$). Mis sous forme d'équation nous avons

$$G(\theta,\varphi) = \eta D(\theta,\varphi) = \eta\frac{U(\theta,\varphi)}{P_{ar}/4\pi}$$

Or, nous avons supposé par hypothèse que notre antenne a un rendement maximal $\eta = 1$. Nous avons une antenne $\lambda/2$, et comme toutes nos antennes émettent et reçoivent dans le plan horizontal, nous savons évaluer la directivité dans cette direction (qui est la direction maximale pour ce genre d'antennes), qui vaut donc :

$$D(\theta = \frac{\pi}{2}) = \frac{16}{3\pi} \approx 1.7 = G(\theta = \frac{\pi}{2})$$

1.2.4 Champ électrique et puissance

Champ électrique

L'amplitude du champ électrique de l'onde électromagnétique reçue au point de réception sera impactée par toutes les réflexions et transmissions qu'elle subira durant le trajet qui la mène au récepteur. Pour modéliser cela, on se sert des coefficients appropriés définis précédemment, l'amplitude du champ sera

$$\underline{E_n} = T_1 T_2 \dots \Gamma_1 \Gamma_2 \dots D_1 D_2 \dots \sqrt{60 G_{\mathrm{TX}} P_{\mathrm{TX}}} \frac{e^{-j\beta d_n}}{d_n} \tag{1.1}$$

avec d_n la distance parcourue par l'onde, D_n les coefficients de diffraction (qui ne seront pas pris en compte pour le projet donc vaudront 1) et P_{TX} la puissance de l'antenne transmettrice. Les autres paramètres de l'équation sont ceux définis plus haut.

Puissance reçue

L'onde électromagnétique, quand elle est réceptionnée au point d'antenne, a une valeur de puissance calculable comme suit

$$P_{\mathrm{RX}} = \frac{1}{8R_a} \left| \sum_{n=1}^{N} \vec{h}_e(\theta_n, \varphi_n) \cdot \overrightarrow{\underline{E}}_n(\vec{r}) \right|^2$$

Cependant, pour évaluer la puissance moyenne reçue sur une zone locale nous considérons

$$< P_{\mathrm{RX}} > = \frac{1}{8R_a} \sum_{n=1}^{N} \left| \vec{h}_e(\theta_n, \varphi_n) \cdot \overrightarrow{\underline{E}}_n(\vec{r}) \right|^2$$

Avec tous les termes définis dans les sections précédentes. Qui, une fois développés donneront la formule utilisée dans l'implémentation que voici (1.2)

$$< P_{\mathrm{RX}} > = \frac{60\lambda^2}{8\pi^2 R_a} P_{\mathrm{TX}} G_{\mathrm{TX}} \sum_{n=1}^{N} \left| \Gamma_1 \Gamma_2 \dots T_1 T_2 \dots \frac{e^{-j\beta d_n}}{d_n} \right|^2 \tag{1.2}$$

1.3 Méthode des images

Pour modéliser toutes les ondes (d'une ou deux réflexions) qui sont transmises de l'émetteur au récepteur, nous utilisons la méthode des images. Cette méthode repose sur la notion d'antennes images, qui sont des antennes fictives placées de manière à simuler les effets des réflexions.

1.3.1 Principe de la méthode

Le principe de la méthode des images est de remplacer les réflexions des ondes sur les surfaces par des sources fictives, appelées images, situées de l'autre côté de ces surfaces. Chaque réflexion sur une surface peut être modélisée par une image située symétriquement par rapport à cette surface.

1.3.2 Détermination du point de réflexion

Considérons une antenne émettrice TX, placée en un point du côté droit d'un mur. La position du point de réflexion peut être calculée comme suit :

$$P_r \equiv \vec{x}_0 + t\vec{u} \text{ avec } t = \frac{d_y(r_{ix} - x_0) - d_x(r_{iy} - y_0)}{u_x d_y - u_y d_x}$$

avec donc, $\vec{x}_0$ la position origine du mur, $\vec{u}$ le vecteur directionnel du mur, r_{ix} et r_{iy} les coordonnées de l'antenne réceptrice, et d_x, d_y, u_x, et u_y les composants des vecteurs directionnels.

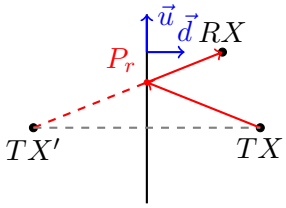


FIGURE 1.1 – Antenne image TX' de l'émettrice TX

1.3.3 Cas à deux réflexions

Pour le cas à deux réflexions, il suffit de prendre d'abord en compte l'image du premier mur sur lequel on se reflète (I_1 sur le schéma), et puis à partir de ce point image, on détermine son propre point image par rapport au second mur sur lequel on se reflète (I_2).

On considère alors que le point de réflexion sur le second mur (Pr_2) est l'intersection entre ledit mur et le récepteur, et pour le premier point de réflexion (Pr_1), il s'agit de l'intersection entre la première image et Pr_2. Ceci est illustré sur la figure 1.2. Il est pertinent de noter que la distance totale du rayon (composé des trois traits verts) est égale à la distance entre le récepteur (RX) et le dernier point image (I_2).

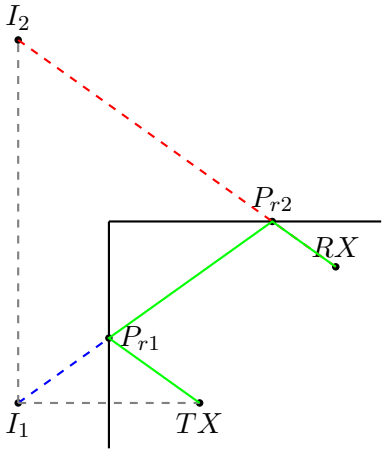


FIGURE 1.2 – Méthode des images pour deux réflexions entre TX et RX

Chapitre 2

Implémentation

2.1 Interface utilisateur

Une interface graphique (visible sur la Figure [2.1]) a été créée afin de faciliter l'exécution de la simulation pour un utilisateur selon ses paramètres choisis, sans devoir recompiler le programme, ni entrer ses paramètres via une ligne de commande.

L'utilisateur peut choisir les coordonnées de la station de base ainsi que le mode de simulation :

— Une *heatmap* de la couverture,

— Des tracés de rayons jusqu'à une seule cellule réceptrice.

Le mode *Coverage Heatmap* permet d'avoir le rendu de la couverture de l'appartement, où l'utilisateur peut choisir la résolution des cellules réceptrices carrées de la carte (leur largeur, allant de 0,5m à 0,0625m).

Quant au mode *Rays to Single Cell*, il permet de visualiser tous les rayons simulés vers une unique cellule (de largeur 0,5m), où l'utilisateur peut choisir les coordonnées de cette cellule.

Le groupe de boutons du dessous contrôle les paramètres de la/des stations de base. L'utilisateur peut rajouter et enlever des émetteurs (en gardant un émetteur au minimum), et modifier leurs coordonnées et leurs puissances.

Le haut de l'interface contient un large bouton *Run simulation* qui permet de lancer la simulation avec les paramètres rentrés. Au-dessus de ce bouton se trouve un cadre qui affichera si la simulation s'est finie avec succès, ainsi que son temps d'exécution en millisecondes. En dessous du bouton se trouve une barre de progression de la simulation. Ceci est visible Figure [2.4].

En haut de la fenêtre se trouve trois menus :

— Le premier (*Menu*) permet de sauvegarder une image de la simulation après son exécution, et permet également de réinitialiser l'application ou de la fermer.

— Le second (*Info*) permet d'avoir des informations sur l'application, et donne aussi un lien vers le Github du projet.

— Le troisième (*TP4-Simulation*) permet de lancer le rendu de la simulation de l'exercice 1 du TP4, ainsi que de sauvegarder une image de celui-ci.

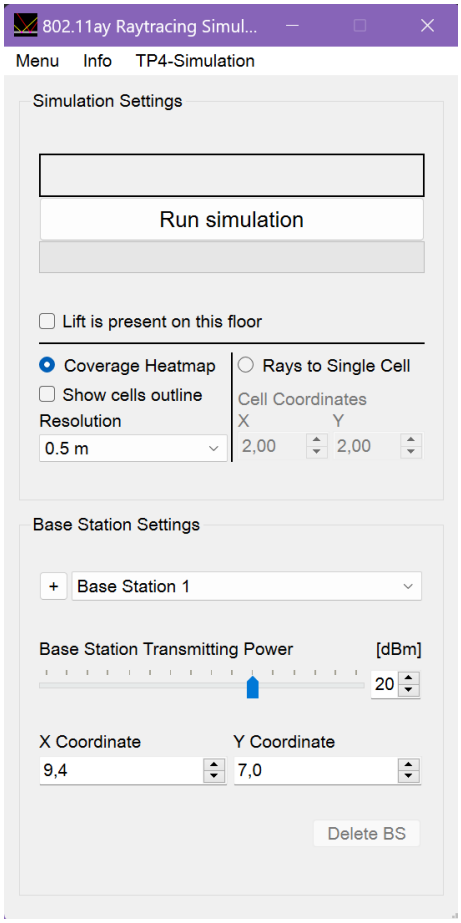


FIGURE 2.1 – Interface utilisateur du programme

2.2 Algorithme

L'algorithme est simplement une itération sur une matrice de cellules (cellule = récepteur) qui calcule la puissance reçue par chacune. Pour chaque récepteur, un rayon direct est tracé depuis l'émetteur vers celui-ci, les coefficients d'éventuelles transmissions à travers des obstacles rencontrés sont calculés. Ensuite, les rayons à plusieurs réflexions sont pris en compte, en vérifiant si l'émetteur (ou ses images) et le récepteur sont du même côté de l'obstacle. Si c'est le cas, l'image de l'émetteur (ou de son image) par rapport à l'obstacle est déterminée, et le point de réflexion est calculé. Les segments de rayon entre le transmetteur, les points de réflexion et le récepteur sont créés, et les coefficients de réflexion sont calculés en fonction de l'angle d'incidence et des propriétés du matériau de l'obstacle. D'éventuelles transmissions après réflexion sont aussi vérifiées et calculées. La puissance totale reçue par chaque récepteur est ensuite calculée en additionnant les contributions de tous les rayons (direct et réfléchis), tel que décrit l'équation [1.2].

2.3 Affichage

Après l'exécution d'une simulation, l'application affiche à l'utilisateur une visualisation graphique de cette simulation. Un exemple est visible Figure [2.3] (pour un mode *Rays to Single Cell*). Elle présente une carte de l'appartement, les axes, ainsi qu'une légende. Les

murs/parois sont représentés par des traits de couleurs et épaisseurs différentes représentant leurs matériaux. L'émetteur est représenté par un point blanc. Les cellules réceptrices seront représentées par des carrés de couleurs indiquant leurs débits binaires selon le dégradé affiché dans la légende.

L'échelle de couleur (un dégradé à 5 points[1], utilisé communément dans les *heatmaps*) a été choisie pour représenter le logarithme du débit binaire, ou la puissance en dBm. Elle a donc été limitée à un minimum de 50Mb/s (ou -90dBm) et à un maximum de 40Gb/s (ou -40dBm).

Il est possible d'afficher davantage d'informations dans une info-bulle sur l'émetteur ou sur une cellule réceptrice lors de leur survol avec la souris (voir Figure [2.2]).

FIGURE 2.2 – Exemple d'info-bulle au survol (curseur souris caché)

Pour une mode de simulation *Rays to Single Cell*, la cellule réceptrice est représentée comme un carré magenta, les points de réflexions sont en vert, et les rayons sont coloriés en fonction de leur nombre de réflexions. Le programme affiche alors ceci [Figure 2.3] :

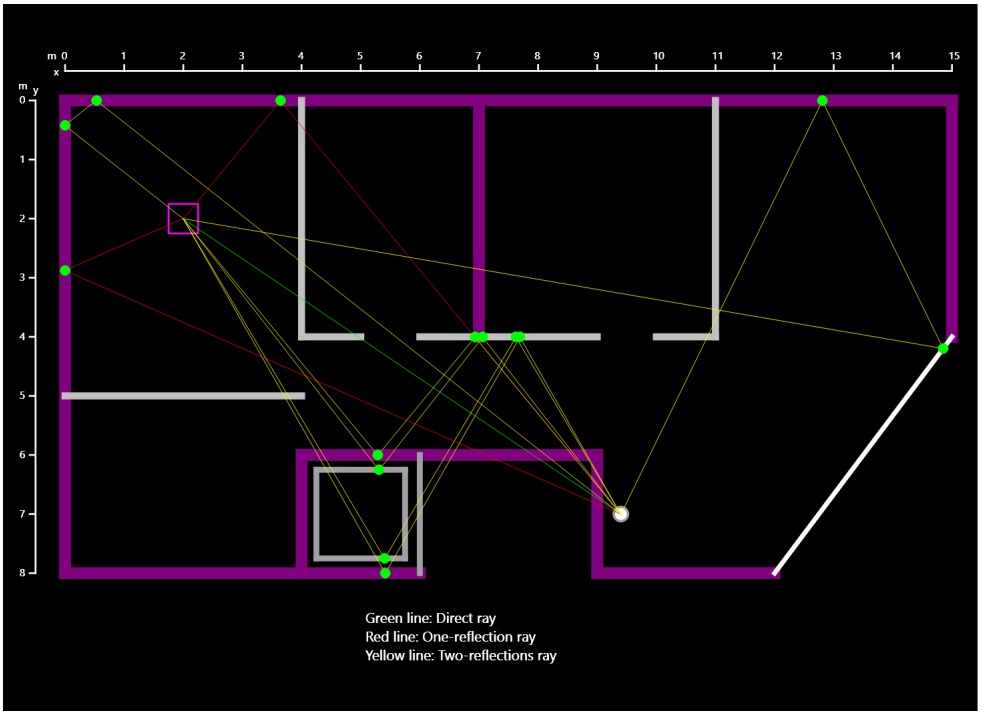


FIGURE 2.3 – Simulation *Rays to Single Cell* pour une cellule en (2,2)m et un émetteur en (9.4,7)m

1. 0% : bleu, 25% : cyan, 50% : vert, 75% : jaune, 100% : rouge

2.4 Ajouts futurs possibles

Les ajouts et améliorations futurs possibles sont :

— Sélection du nombre maximal de réflexions des rayons (nécessitant un algorithme de ray-tracing récursif)
— Importation de différentes cartes d'appartement
— Diffraction des ondes

2.5 Performances

Le temps d'exécution pour une simulation exemple, de résolution de cellule de $0.125\text{m} \times 0.125\text{m}$ est aux alentours des 500ms [Figure 2.4], avec une utilisation de 2,8GB de RAM, soit des performances tout à fait acceptables. Une simulation à la plus basse résolution (de base) de $0.5\text{m} \times 0.5\text{m}$ prend une trentaine de millisecondes. [2]

Le temps d'exécution a été calculé grâce à un objet `QTimer`, lancé après une pression du bouton *Run simulation* et stoppé juste avant l'affichage graphique.

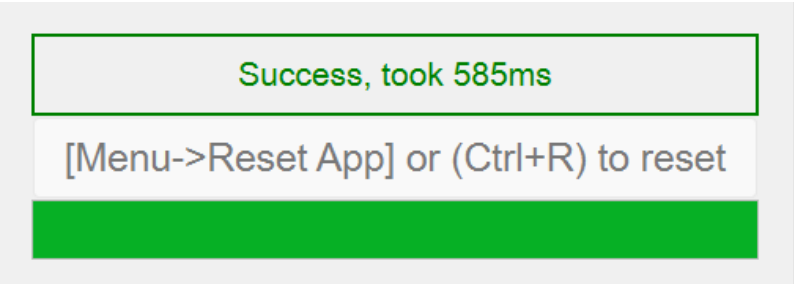


FIGURE 2.4 – Temps d'exécution simulation de résolution 0.125m

Le temps d'exécution pour une simulation *Rays to Single Cell*, ou pour la simulation du TP4, tourne autour des 2ms.

2. Exécution sur un laptop doté d'un CPU AMD Ryzen 7 5800U avec 16GB de RAM, en mode *économie d'énergie*.

Chapitre 3

Validation

Remarque : lors de la simulation de la situation de l'exercice 1 du TP4, nous ne pouvons pas utiliser la puissance sur une zone locale $< P_{RX} >$ (équation 1.2), mais bien P_{RX} en un point de l'espace (équation 3.1).

$$P_{RX} = \frac{60\lambda^2}{8\pi^2 R_a} P_{TX} G_{TX} \left| \sum_{n=1}^{N} \Gamma_1 \Gamma_2 \Gamma_3 \dots T_1 T_2 T_3 \frac{e^{-j\beta d_n}}{d_n} \right|^2 \tag{3.1}$$

Le calcul du champ électrique utilise l'équation (1.1).

L'exercice 1 du TP4 utilise un $P_{TX}G_{TX} = 1.64$, une fréquence à $868.3\mathrm{MHz}$, une résistance d'antenne $R_a = 73\Omega$, ainsi que des murs ayant comme permittivité relative $\varepsilon_r = 4.8$ et comme conductivité $\sigma = 0.018\mathrm{S/m}$.

3.1 Calculs pour rayon à deux réflexions, cas A

La méthode pour obtenir les valeurs du cas à deux réflexions est celle brièvement présentée dans la section correspondante, et le cas à déterminer est d'ailleurs le même que la figure correspondante.

3.1.1 Positions des réflexions

La position des images antenne s'obtient par symétrie, $I_1 = (-32, 10)$ et ensuite $I_2 = (-32, 150)$. On commence par déterminer le point P_{r2} qui est l'intersection entre le mur horizontal le plus élevé sur l'axe y (mur B) et $\overrightarrow{I_2RX}$. On commence par évaluer $\overrightarrow{I_2RX}$:

$$\overrightarrow{I_2RX} = (79, -85) \rightarrow \|\overrightarrow{I_2RX}\| = 116.04 \rightarrow \frac{\overrightarrow{I_2RX}}{\|\overrightarrow{I_2RX}\|} = (0.68, 0.73)$$

Avec cela, on peut déterminer P_{r2}, dont on connaît la coordonnée $y = 80$ et dont on veut déterminer la coordonnée x.

$$P_{r2} = I_2 + \frac{\overrightarrow{I_2RX}}{\|\overrightarrow{I_2RX}\|} t \rightarrow t = 95.56 \rightarrow x = 33.06 \rightarrow P_{r2} = (33.06, 80)$$

Et on peut déterminer de la même manière, la coordonnée de P_{r1}, connaissant celle de P_{r2}, le vecteur est donc $\overrightarrow{I_1P_{r2}}$. Nous obtenons

$$\overrightarrow{I_1P_{r2}} = (65.06, 70) \rightarrow \|\overrightarrow{I_1P_{r2}}\| = 95.57 \rightarrow \frac{\overrightarrow{I_1P_{r2}}}{\|\overrightarrow{I_1P_{r2}}\|} = (0.68, 0.73)$$

On connait cette fois ci la coordonnée $x = 0$ du point de réflexion, et on cherche y.

$$P_{r1} = I_1 + \frac{\overrightarrow{I_1P_{r2}}}{\|\overrightarrow{I_1P_{r2}}\|} t \rightarrow t = 47 \rightarrow y = 44.43 \rightarrow P_{r1} = (0, 44.43)$$

Il nous reste finalement à déterminer P_t le point d'intersection entre $\overrightarrow{P_{r1}\mathrm{TX}}$ et le mur vertical passant par $A = (0, 20)$.

$$\overrightarrow{P_{r1}\mathrm{TX}} = (-32, 34.43) \rightarrow \|\overrightarrow{P_{r1}\mathrm{TX}}\| = 47 \rightarrow \frac{\overrightarrow{P_{r1}\mathrm{TX}}}{\|\overrightarrow{P_{r1}\mathrm{TX}}\|} = (-0.68, 0.73)$$

On connaît la coordonnée $y = 20$,

$$P_t = \mathrm{TX} + \frac{\overrightarrow{P_{r1}\mathrm{TX}}}{\|\overrightarrow{P_{r1}\mathrm{TX}}\|} t \rightarrow t = 13.65 \rightarrow x = 22.71 \rightarrow P_{r1} = (22.71, 20)$$

3.1.2 Coefficients

Nous évaluerons les coefficients en prenant le même ordre que celui du calcul des points de réflexion et transmission. Nous commençons par déterminer Γ_2 associé à la réflexion en P_{r2}, avec la normale est $(0, -1)$ nous avons

$$\cos\theta_i = \langle \frac{\overrightarrow{I_2RX}}{\|\overrightarrow{I_2RX}\|}, (0, -1) \rangle = 0.73 \rightarrow \sin\theta_i = \sqrt{1 - \cos^2\theta_i} = 0.68$$

L'obtention des composantes transmises dans le mur se font via la loi de Snell, avec $\varepsilon_r = 4.8$

$$\sin\theta_t = \frac{1}{\sqrt{\varepsilon_r}} \sin\theta_i = 0.31 \rightarrow \cos\theta_t = 0.95$$

La distance parcourue par le rayon dans le mur s, est par conséquent

$$s = \frac{l}{\cos\theta_t} = 0.16$$

Avec $Z_m = (171.57 + 6.65j)\Omega$ et $Z_0 = 377\Omega$, le coefficient $\Gamma_\perp$ de polarisation perpendiculaire vaut

$$\Gamma_\perp = \frac{Z_m \cos\theta_i - Z_0 \cos\theta_t}{Z_m \cos\theta_i + Z_0 \cos\theta_t} = -0.48 + 0.015j$$

On peut donc, pour finir, évaluer Γ_2

$$\Gamma_m = \Gamma_\perp - \frac{(1 - \Gamma_\perp^2)\Gamma_\perp e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}}{1 - \Gamma_\perp^2 e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}} = -0.42 + 0.25j$$

Les calculs pour Γ_1 se font de la même manière que précédemment, il nous suffit alors d'indiquer les résultats intermédiaires. Avec le mur de normale $(1, 0)$, on obtient $\cos\theta_i = 0.68$, $\sin\theta_i = 0.73$, $\sin\theta_t = 0.33$, $\cos\theta_t = 0.94$ et $s = 0.16$. Qui donnent $\Gamma_\perp = -0.51 + 0.0144j$, qui permet d'obtenir

$$\Gamma_1 = -0.47 + 0.25j$$

Il nous reste plus qu'à évaluer T_1 le coefficient de transmission. Les valeurs trigonométriques des angles incidents et transmis sont facilement obtenables avec la même méthode que précédemment $\cos\theta_i = \langle \frac{\overrightarrow{P_{r1}\mathrm{TX}}}{\|\overrightarrow{P_{r1}\mathrm{TX}}\|}, (0, 1) \rangle = 0.73$, $\sin\theta_i = 0.68$, $\sin\theta_t = 0.31$, $\cos\theta_t = 0.95$ et $s = 0.16$. Par conséquent, $\Gamma_\perp = -0.51 + 0.014j$ et pour finir

$$T_1 = \frac{(1 - \Gamma_\perp^2) e^{-\gamma_m s}}{1 - \Gamma_\perp^2 e^{-2\gamma_m s} e^{j\beta 2s \sin\theta_t \sin\theta_i}} = 0.63 + 0.089j$$

3.1.3 Champ et puissance

L'amplitude du champ se calcule, pour notre cas, via

$$\underline{E_n} = \Gamma_1\Gamma_2 T_1 \sqrt{60 P_{\mathrm{TX}} G_{\mathrm{TX}}} \frac{e^{-j\frac{2\pi f}{c} d_n}}{d_n} = 4.4519 \cdot 10^{-4} - 2.1816 \cdot 10^{-5} j$$

Avec la distance d_n étant celle qui sépare RX de TX, c'est aussi la distance qui sépare RX de I_2, et vaut donc $\|\overrightarrow{I_2RX}\| = 116.04$.
La puissance vaut donc

$$P_{RX} = \frac{60\lambda^2}{8\pi^2 R_a} P_{TX} G_{TX} \left| \Gamma_1\Gamma_2 T_1 \frac{e^{-j\beta d_n}}{d_n} \right|^2 = 4.1145 \cdot 10^{-12} W$$

Ce qui donne, en dB, $< P_{\mathrm{RX}} >= -83.85\mathrm{dBm}$.

3.2 Comparaison calculs et simulation

Nous pouvons comparer les résultats de notre simulateur avec les résultats calculés manuellement afin de valider notre implémentation. Un tableau comparatif [Table 3.1] présente ces données.

	Grandeur	Calculs	Simulation	Erreur
Direct	$\lvert\underline{E}\rvert$ [V/m]	$4,031 \cdot 10^{-3}$	$4,0124 \cdot 10^{-3}$	$0,40\%$
	P_{RX} [W]	$3,33 \cdot 10^{-10}$	$3,32965 \cdot 10^{-10}$	$0,01\%$
1 réflexion (A)	$\lvert\underline{E}\rvert$ [V/m]	$7,0819 \cdot 10^{-4}$	$7,07611 \cdot 10^{-4}$	$0,08\%$
	P_{RX} [W]	$1,04 \cdot 10^{-11}$	$1,03557 \cdot 10^{-11}$	$0,42\%$
1 réflexion (B)	$\lvert\underline{E}\rvert$ [V/m]	$6,7821 \cdot 10^{-4}$	$6,78351 \cdot 10^{-4}$	$0,02\%$
	P_{RX} [W]	$9,53 \cdot 10^{-12}$	$9,51694 \cdot 10^{-12}$	$0,14\%$
2 réflexions (A)	$\lvert\underline{E}\rvert$ [V/m]	$4,4572 \cdot 10^{-4}$	$4,46271 \cdot 10^{-4}$	$0,12\%$
	P_{RX} [W]	$4,1145 \cdot 10^{-12}$	$4,11896 \cdot 10^{-12}$	$0,11\%$

TABLE 3.1 – Comparaison valeurs calculs et simulation selon le rayon

On peut observer que notre simulateur se rapproche beaucoup des valeurs réelles calculées à la main, avec une erreur ne dépassant pas 0,5%. Ceci permet donc de valider l'implémentation de notre simulateur.

3.3 Affichage simulation des rayons

La Figure [3.1] présente le tracé des rayons simulés. Le récepteur est représenté comme un carré bleu, l'émetteur comme un rond blanc, les points de réflexions comme des points magenta, les murs comme des traits gris, et les rayons sont différenciés par leur couleur représentant leur nombre de réflexions :

— Vert : rayon direct (0 réflexion)
— Rouge : rayon à une réflexion
— Jaune : rayon à deux réflexions

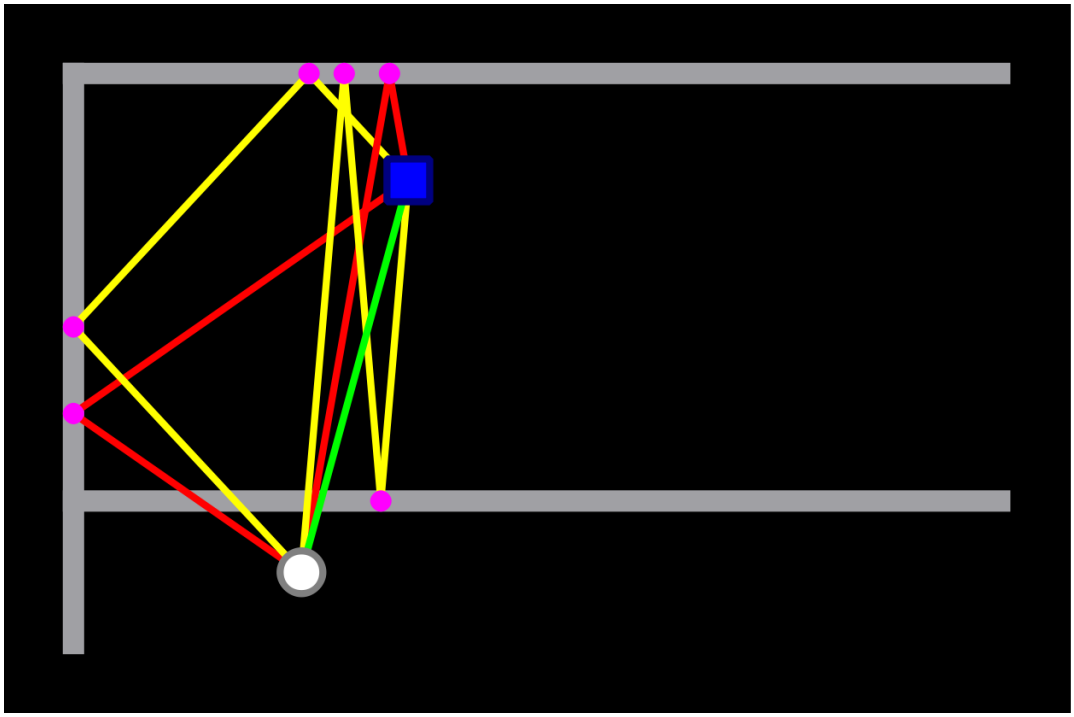


FIGURE 3.1 – Simulation Ray-Tracing de l'exercice 1, TP4

Comme le simulateur complet de l'appartement, il est possible d'afficher plus d'informations du récepteur, tel que ses coordonnées ou encore sa puissance reçue, lors du survol de la souris par-dessus.

Chapitre 4

Analyse performance

4.1 Résultats

Les résultats ont été réalisés avec la simulation à sa plus haute résolution (soit des cellules de $0,0625\text{m} \times 0,0625\text{m}$). L'emplacement par défaut est à (9.4,7)m.

4.1.1 Emplacement par défaut (salon)

Sans ascenseur

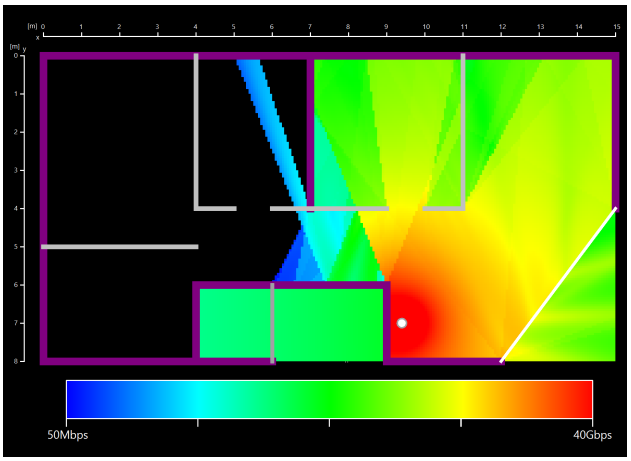


FIGURE 4.1 – Couverture débit binaire station de base salon, sans ascenseur

Avec ascenseur

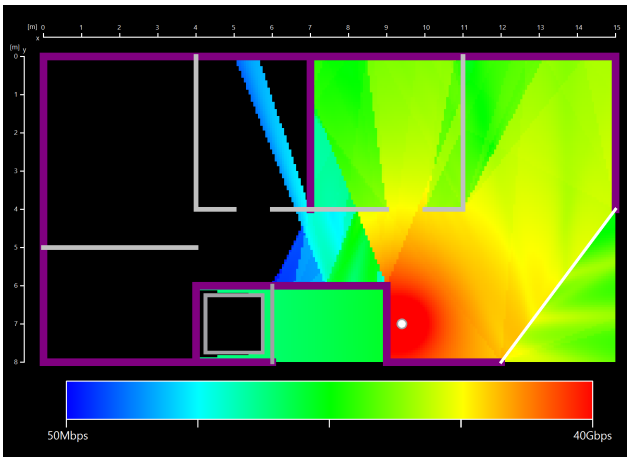


FIGURE 4.2 – Couverture débit binaire station de base salon, avec ascenseur

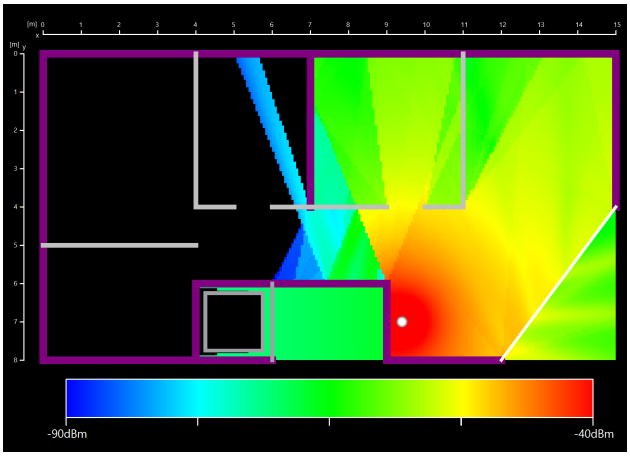


FIGURE 4.3 – Couverture puissance station de base salon, avec ascenseur

On observe qu'il n'y a pas de différence visuelle entre la couverture puissance et la couverture débit binaire car l'échelle est définie comme le logarithme du débit binaire, ou la puissance en dBm, donc l'amplitude de couleur est la même.

4.2 Suggestions placement station de base

Afin de trouver un emplacement optimal pour la station de base, il est intéressant de réaliser un algorithme trouvant cette position. Nous profitons de la vitesse d'exécution de notre simulation, en déterminant l'emplacement ayant la plus haute moyenne des puissances des cellules, via une méthode d'optimisation en *brute-forcing*, c'est-à-dire nous itérons à travers toutes nos positions de station de base, et gardons seulement la meilleure. Notre algorithme d'optimisation nous indique qu'un emplacement optimal est (6.5, 4.5)m, la résolution basse a été utilisée, soit des cellules de $0,5\text{m} \times 0,5\text{m}$.

Cet emplacement optimal **théorique** au milieu de l'appartement offre une large couverture, dû à son placement central et à sa plus grande distance à tous les murs en béton.

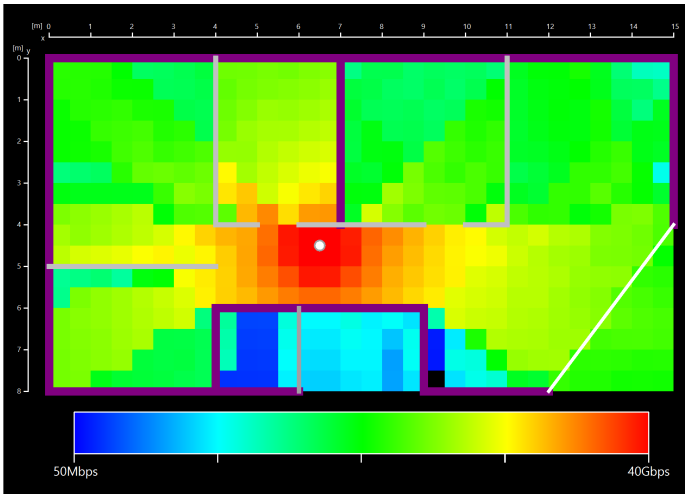


FIGURE 4.4 – Couverture débit binaire station de base optimale, sans ascenseur

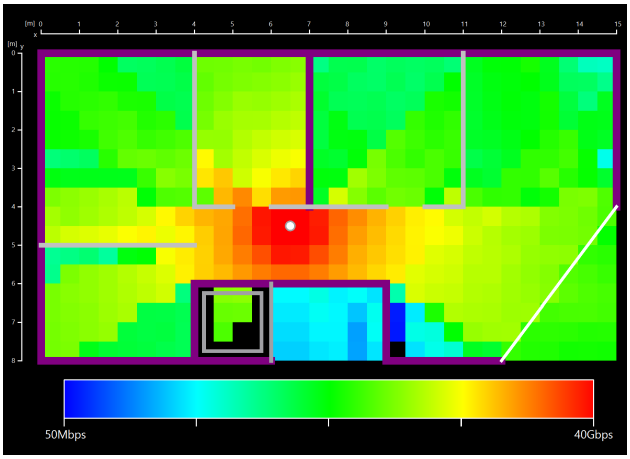


FIGURE 4.5 – Couverture débit binaire station de base optimale, avec ascenseur

Un emplacement au milieu d'une pièce est évidemment non favorable en pratique, car loin de prises de courant et réseau, et nécessite aussi sans doute de placer la station de base au plafond ce qui engendre un travail supplémentaire lors de l'installation et un aspect esthétique moindre. Nous privilégions alors un emplacement proche d'un mur, sans doute sur un meuble, par soucis pratique.

L'emplacement proche de celui du projet par défaut par exemple à (9.4, 6.4)m semble être un emplacement réaliste et efficace et donc meilleur à utiliser en **pratique**. La station de base se trouve dans le salon de l'appartement, un lieu où est souvent situé le raccordement internet, et là où sont majoritairement utilisés les appareils sans-fil lors de besoins élevés en débit binaire (téléchargements, vidéos, etc). De plus, la cuisine et la salle à manger sont bien couvertes (deux autres pièces qui sont plus enclines à une utilisation du réseau Wi-Fi par les habitants, contrairement aux chambres et salle de bain). Voir Figures [4.1] et [4.2] pour une couverture depuis un emplacement similaire.

4.3 Suggestion nombre de stations de base

Utilisons l'emplacement réaliste situé à (9.4, 6.4)m pour la première station de base, tel que détaillé au point précédent.

Pour optimiser davantage la couverture, il est proposé d'ajouter une deuxième antenne près de (2, 4.75)m. Cette disposition permettra de minimiser les zones de faible signal, en assurant une meilleure couverture dans les autres parties de la maison, notamment les chambres et les zones éloignées de la station de base principale. Tout en gardant les avantages pratiques réalistes que nous obtenions avec la position du premier émetteur. En envisageant l'utilisation d'un système mesh (c'est-à-dire plusieurs émetteurs pour un même réseau), nous pouvons améliorer la couverture et réduire l'effet des obstacles. Un système mesh permet en effet de distribuer plusieurs stations de base dans tout l'appartement, garantissant une meilleure couverture. Cela assure une distribution plus homogène du signal Wi-Fi, optimisant ainsi les performances du réseau pour tous les habitants sur l'ensemble de la surface.

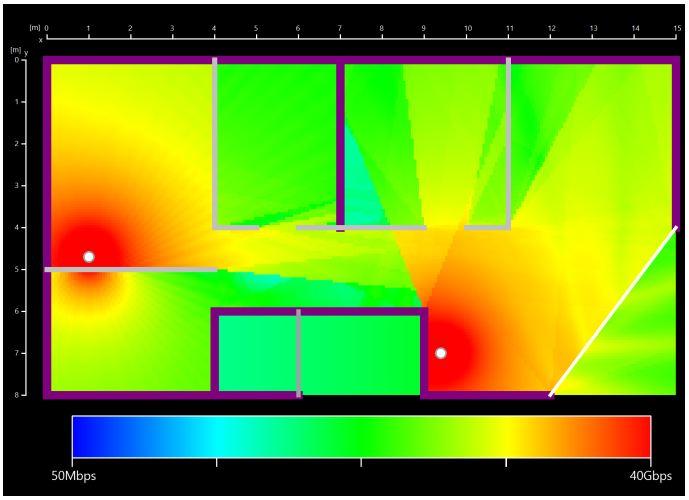


FIGURE 4.6 – Exemple de couverture débit binaire pour deux stations de base, sans ascenseur

Cependant, comme notre modèle ne prend pas en compte d'éventuelles interférences entre les différentes stations de base, il est donc **théoriquement** avantageux d'avoir le plus de station de base possible, ce qui n'est pas réaliste.

4.4 Critique de la simulation

Tout d'abord, on peut observer qu'un débit binaire non-nul est présent à l'intérieur de l'ascenseur, juste à côté ainsi que dans la cage. Ce résultat ne devrait pas se produire à cause de la présence d'épais murs en béton ainsi que de la/des parois métalliques constituant l'ascenseur. Cette erreur a malheureusement une origine inconnue, il se peut qu'elle survienne à cause d'un **bug** des coins des murs, où des ondes passeraient. Bien des essais et des réécritures ont été tentées, mais sans succès. Essayer de résoudre ce problème fut une des choses qui prit le plus de temps durant la réalisation du projet, plus de détails sur les différents essais de résolution sont présents dans l'**Annexe [A]**.

Ensuite, cette simulation repose sur de nombreuses simplifications et approximations. En effet, l'appartement ne présente pas d'obstacle de type meubles, ne possède aucune autre fenêtre (en plus de la baie vitrée oblique) et les murs en béton sont normalement recouverts d'une couche de plâtre (affectant alors la réflexion différemment du béton).

De plus, la simulation considère une antenne parfaitement omnidirectionnelle dans le plan étudié.

Enfin, la simulation ne prend pas en compte la diffraction, et ignore les technologies maintenant très répandues dans les antennes Wi-Fi tel que le *beam-forming* ou encore le *MIMO*.

Chapitre 5

Optimisations code

L'utilisation du **C++** pour notre projet nous donne un grand avantage au niveau performance de notre programme, c'est en effet un langage qui, après compilation, est connu pour être très optimisé pour la rapidité grâce à son faible niveau d'abstraction et son absence de *garbage collector*[1]. Ceci nous a permis d'avoir un temps d'exécution déjà très faible sans avoir implémenté d'optimisation spécifique. Cet avantage couplé au manque de temps nous ont finalement conduit à ne pas mettre en place une des optimisations possibles.

Cependant, il reste intéressant de mentionner la discussion au sein du groupe faite en début de projet pour comparer les différentes optimisations, qui est résumée dans le tableau ci-dessous (Table [5.1]) :

	Optimisation du temps de calcul	Difficulté de mise en place
1. Parallélisation (CPU)	grande	moyenne
2. GPU	très grande	grande
3. Mise en cache	faible	faible
4. Abandon anticipé	incertaine	moyenne

TABLE 5.1 – Comparaison méthodes d'optimisations

Explications de chaque méthode d'optimisation Table [5.1] :

1. Parallélisation (CPU) : diviser le calcul entre plusieurs threads pour traiter les cellules simultanément.
2. GPU : utiliser le processeur graphique pour une parallélisation massive, par exemple avec l'API Nvidia CUDA.
3. Mise en cache : stocker les valeurs calculées fréquemment, comme les $\sin(\theta_i)$ et les coefficients de réflexion Γ_m pour un même angle d'incidence sur un même matériau, pour éviter de les recalculer.
4. Abandon anticipé : arrêter le calcul des rayons lorsque la puissance devient trop faible pour contribuer significativement à la couverture.

1. Le *garbage collector* est un mécanisme de gestion automatique de la mémoire qui récupère et libère les espaces mémoire non utilisés pour éviter les fuites de mémoire et optimiser l'utilisation des ressources. Un *garbage collector* est généralement assez lourd en performance, comparé à une gestion manuelle de la mémoire.

Conclusion

En conclusion, ce projet de ray-tracing pour le Wi-Fi IEEE 802.11ay nous a permis de développer un simulateur capable d'analyser la couverture réseau dans un environnement résidentiel prédéfini. Grâce à l'utilisation de la méthode des images et à l'utilisation du langage C++, le simulateur offre des résultats précis avec une erreur de moins de 0,5% par rapport aux calculs manuels, et ce en un temps très réduit. La meilleure position pour la ou les stations de base, du point de vue pratique et théorique, a été identifiée, garantissant une couverture optimale de l'appartement. Bien que certaines limitations, comme l'absence de diffraction et les anomalies près de l'ascenseur, subsistent, le projet atteint ses objectifs principaux et propose une base solide pour des améliorations futures.

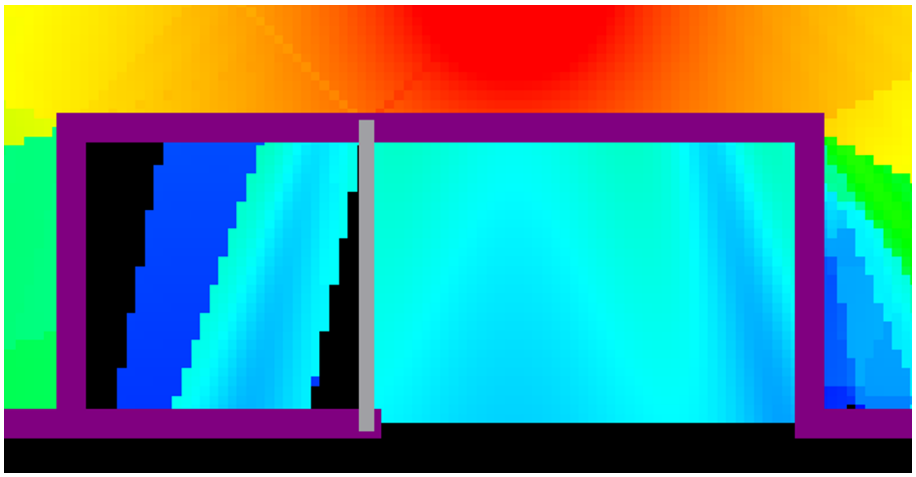
Annexes

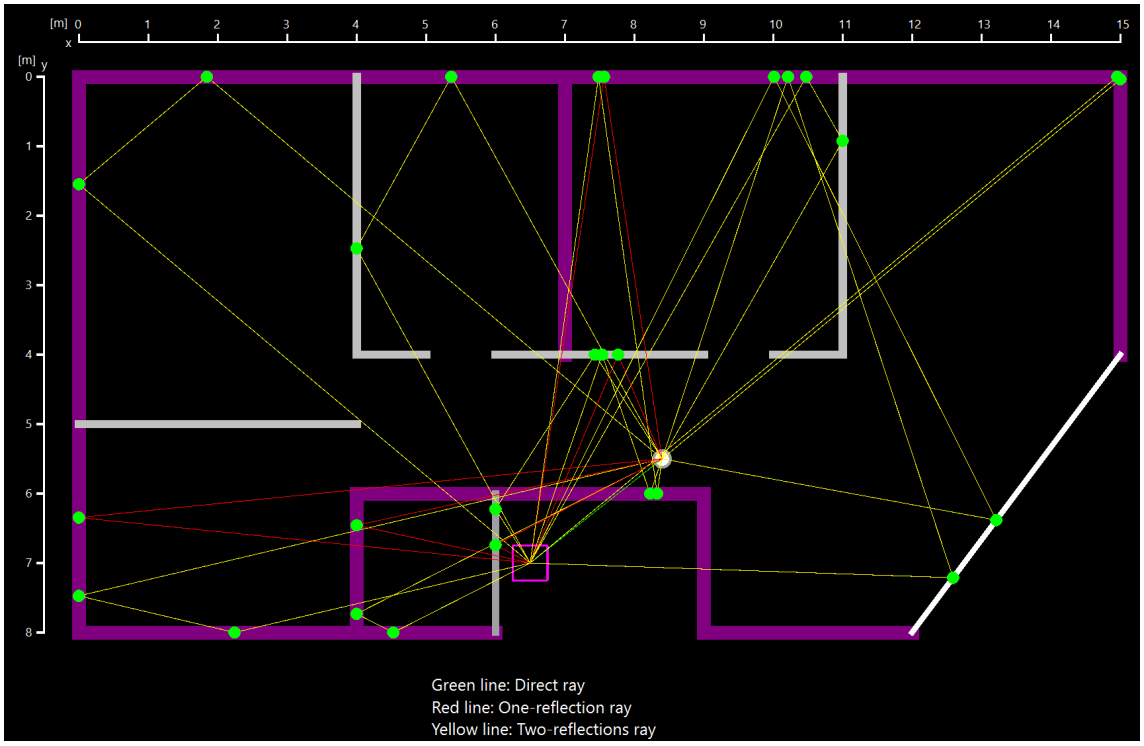
A Discussion du débit binaire près de l'ascenceur

Théoriquement, on devrait observer un blocage du signal dans cette zone en raison des matériaux utilisés. Cependant, nous avons rencontré un problème inattendu de valeur de puissance trop élevée (environ $-68\mathrm{dBm}$ pour une station de base en $(9.4, 7)m$ avec $20\mathrm{dBm}$ de puissance d'émission).

Nous avons tenté de le résoudre de plusieurs manières. Nous avons déplacé les murs pour vérifier s'il y avait un problème au niveau des coins. Nous avons vérifié avec notre fonction *Rays to Single Cell*, les rayons tracés à cet endroit. Nous avons comparé et cherché l'erreur avec des camarades, à plusieurs reprises et pendant longtemps, sans succès. Plusieurs matériaux ont été testés, et nous avons vérifié les propriétés physiques des murs à de nombreuses reprises.
En outre, nous avons revérifié toutes les fonctions de calcul des coefficients et réécrit plusieurs parties du code plusieurs fois dans l'espoir de trouver la faille. Enfin, différentes positions de la station de base et plusieurs stations ont été testées.
Malgré tous ces efforts, le problème persiste et reste un point d'investigation pour des travaux futurs.

Ce qui reste étrange tout de même, est que c'est la seule zone dans laquelle une si grande erreur de débit binaire persiste.




[m]
x
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
[m]
y
0
1
2
3
4
5
6
7
8
Green line: Direct ray
Red line: One-reflection ray
Yellow line: Two-reflections ray

B Code

Cette partie de l'annexe comprend la quasi-totalité du code du projet.

B.1 main.cpp

Lance simplement l'interface utilisateur de l'aplication.

```
#include "mainwindow.h"

#include <QApplication>
//#include "algorithme.h"

int main(int argc, char *argv[])
{
    //Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);
    QCoreApplication::setOrganizationName("Lucas Placentino and Salman Houdaibi");
    QCoreApplication::setApplicationName("802.11ay Raytracing Simulator");
    QCoreApplication::setApplicationVersion(QT_VERSION_STR);

    MainWindow mainWin;
    mainWin.show();

    //runAlgo();

    return app.exec();
}

```

B.2 simulation.cpp

Ce fichier contient toute l'implémentation de la logique de la simulation.
Le *header file* ***.h*** définissant la classe **Simulation** se trouve dans le Github du projet.

```
#include "simulation.h"

#include <QDir>
#include <QtMath>

#include "parameters.h"


Simulation::Simulation(bool show) {
    // class constructor
    this->show = show;

    createWalls();
}

void Simulation::createWalls()
{
    //qDebug("Creating walls...");

    // Delete old obstacles vector objects (avoid memory leak)
    if(!this->obstacles.empty()) {
        // The vector is not empty
        for(auto ptr : this->obstacles) {
            delete ptr;
        }
        this->obstacles.clear();
    }

    // Add obstacles:
    QList<Obstacle*> concrete_walls;
    ObstacleType concrete = ConcreteWall;
    qreal thickness = 0.3; // 30cm
    concrete_walls.append(new Obstacle(QVector2D(0,0), QVector2D(15,0), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(15,0), QVector2D(15,4), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(7,0), QVector2D(7,4), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(0,0), QVector2D(0,8), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(0,8), QVector2D(6,8), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(4,6), QVector2D(9,6), concrete,
    ↪  thickness));
    concrete_walls.append(new Obstacle(QVector2D(4,6), QVector2D(4,8), concrete,
    ↪  thickness));
```

```
40      concrete_walls.append(new Obstacle(QVector2D(9,6), QVector2D(9,8), concrete,
        ↪  thickness));
41      concrete_walls.append(new Obstacle(QVector2D(9,8), QVector2D(12,8), concrete,
        ↪  thickness));
42
43      QList<Obstacle*> drywall_walls;
44      ObstacleType drywall = DryWall;
45      thickness=0.1; // 10cm
46      drywall_walls.append(new Obstacle(QVector2D(4,0), QVector2D(4,4), drywall,
        ↪  thickness));
47      drywall_walls.append(new Obstacle(QVector2D(4,4), QVector2D(5,4), drywall,
        ↪  thickness));
48      drywall_walls.append(new Obstacle(QVector2D(6,4), QVector2D(9,4), drywall,
        ↪  thickness));
49      drywall_walls.append(new Obstacle(QVector2D(10,4), QVector2D(11,4), drywall,
        ↪  thickness));
50      drywall_walls.append(new Obstacle(QVector2D(11,0), QVector2D(11,4), drywall,
        ↪  thickness));
51      drywall_walls.append(new Obstacle(QVector2D(0,5), QVector2D(4,5), drywall,
        ↪  thickness));
52
53      thickness=0.05; // 5cm
54      Obstacle* glass_window = new Obstacle(QVector2D(12,8),QVector2D(15,4), Window,
        ↪  thickness); // this one is diagonal
55
56      thickness=0.05; // 5cm
57      Obstacle* metal_lift_door = new Obstacle(QVector2D(6,6),QVector2D(6,8), MetalWall,
        ↪  thickness);
58
59      // /!\ The lift is only added to the obstacles if enabled
60      QList<Obstacle*> all_obstacles;
61      all_obstacles.append(concrete_walls);
62      all_obstacles.append(drywall_walls);
63      all_obstacles.append(glass_window);
64      all_obstacles.append(metal_lift_door); // last one is the metal lift door
65
66      this->obstacles = all_obstacles;
67      qDebug() << "Walls created";
68  }
69
70  void Simulation::run(QProgressBar* progress_bar)
71  {
72      // TODO: compute everything
73      this->timer.start();
74      qDebug() << "Simulation::run() - single cell simulation: " <<
        ↪  (this->showRaySingleCell); // still TODO: single cell simulation
75
76      qDebug() << "P_TX:" << P_TX << "W," << P_TX_dBm << "dBm";
77      qDebug() << "G_TX:" << G_TX;
78      qDebug() << "beta:" << beta_0;
79      qDebug() << "lambda:" << lambda << "m";
```

```
80      qDebug() << "frequency:" << freq << "Hz";
81      qDebug() << "omega:" << omega << "rad/s";
82
83      //this->cells_matrix.clear();
84      for (QList<Receiver*> cells_line : this->cells) {
85          qDeleteAll(cells_line);
86      }
87      this->cells.clear();
88      qDeleteAll(this->obstacles);
89      this->obstacles.clear();
90
91      createWalls();
92      if (this->lift_is_on_floor) { // Adds the lift metal walls if set as present
93          qDebug() << "Lift is on this floor.";
94
95          QList<Obstacle*> lift_walls;
96          qreal thickness = 0.05; // 5cm
97          lift_walls.append(new Obstacle(QVector2D(4.25,6.25),QVector2D(5.75,6.25),
            ↪  MetalWall, thickness));
98          lift_walls.append(new Obstacle(QVector2D(4.25,6.25),QVector2D(4.25,7.75),
            ↪  MetalWall, thickness));
99          lift_walls.append(new Obstacle(QVector2D(5.75,6.25),QVector2D(5.75,7.75),
            ↪  MetalWall, thickness));
100         lift_walls.append(new Obstacle(QVector2D(4.25,7.75),QVector2D(5.75,7.75),
            ↪  MetalWall, thickness));
101
102         this->obstacles.append(lift_walls);
103     }
104
105     if (!this->showRaySingleCell) {
106         createCellsMatrix();
107     } else {
108         // single cell simulation
109         Receiver* rx = new Receiver(this->singleCellX,this->singleCellY,0.5, true);
110         QPen singleCellPen = QPen(Qt::magenta);
111         singleCellPen.setWidthF(10*0.03);
112         rx->graphics->setPen(singleCellPen);
113         this->cells = {{rx}};
114     }
115
116     //Transmitter* base_station = this->baseStations[0]; // TODO: feature : multiple
        ↪  transmitters?
117
118     //if (this->cells_matrix.isEmpty()) {
119     if (this->cells.isEmpty()) {
120         qWarning("no cells provided (simulation.cells matrix is empty)");
121         throw std::exception();
122     }
123
124     for (Transmitter* tx : this->baseStations) {
125         int i=0;
```

```
126         for (QList<Receiver*> cells_line : this->cells){
127             // loops over each line
128             for (Receiver* cell : cells_line) {
129                 computeDirect(cell, *tx);
130                 computeReflections(cell, *tx);
131                 progress_bar->setValue(i/(cells_line.length()*this->cells.length()));
132                 i++;
133             }
134         }
135     }
136     qDebug() << this->cells.length()*this->cells[0].length() << "cells,";// << i <<
    ↪  "computed";
137
138     //end of simulation
139     this->simulation_time = this->timer.elapsed();
140     qDebug() << "Simulation time:" << this->simulation_time << "ms";
141
142     if (this->show) {
143         showView();
144     }
145
146 }
147
148 void Simulation::computeReflections(Receiver* _RX, const QVector2D& _TX)
149 {
150     // Makes the whole reflections computation, summary:
151     // For each wall, check if TX (or its image, for the 2nd reflection) and RX are on the
    ↪  same
152     // side of the wall, if so computes TX's (or its image's) image with this wall, then
    ↪  computes
153     // the reflection point which is the intersection of the line between the image and RX
    ↪  and the
154     // wall. Creates the segments from TX to each reflection point to finally RX and
    ↪  creates a new
155     // Ray object made of these segments.
156
157     // 1st reflection :
158     //for (Wall* wall: wall_list) { // could use this instead
159     for (int i=0; i<this->obstacles.length(); i++) {
160         Obstacle* wall = this->obstacles[i];
161
162         // check if same side of wall, if false, then no reflection only transmission
163         if (checkSameSideOfWall(wall->normal,_TX,*_RX)) {
164             //same side of this wall, can make a reflection
165             //qDebug() << "Same side of wall TX and RX:" << wall << _TX.toPointF() <<
                ↪  _RX->toPointF() ;
166             Ray* ray_1_reflection = new Ray(_TX.toPointF(), _RX->toPointF());
167
168             QVector2D _imageTX = computeImage(_TX, wall);
169             //qDebug() << "_image:" << _imageTX.x() << _imageTX.y();
170
```

```
171             QVector2D _P_r = calculateReflectionPoint(_imageTX,*_RX,wall);
172
173             // CHECK IF REFLECTION IS ON THE WALL AND NOT ITS EXTENSION:
174             RaySegment* test_segment = new
                ↪  RaySegment(_imageTX.x(),_imageTX.y(),_RX->x(),_RX->y());
175             if (!checkRaySegmentIntersectsWall(wall, test_segment)) {
176                 // RAY DOES NOT TRULY INTERSECT THE WALL (only the wall extension) ignore
                    ↪  this one-reflection ray at this wall
177                 //qDebug() << "ignore";
178                 delete ray_1_reflection;
179                 delete test_segment;
180                 ////continue; // break out of this forloop instance for this wall
181                 goto second_reflection; // don't make a 1 reflection ray with this wall,
                    ↪  proceed to two-reflections rays
182             }
183             delete test_segment;
184
185             //qDebug() << "P_r" << _P_r;
186
187             // create ray segments between points
188             QList<RaySegment*> ray_segments;
189             ray_segments.append(new RaySegment(_TX.x(),_TX.y(),_P_r.x(),_P_r.y())); //
                ↪  first segment
190             ray_segments.append(new RaySegment(_P_r.x(),_P_r.y(),_RX->x(),_RX->y())); //
                ↪  last segment
191
192             ray_1_reflection->segments = ray_segments;
193             addReflection(ray_1_reflection,_imageTX,*_RX,wall);
194             checkTransmissions(ray_1_reflection,{wall});
195
196             if (this->showRaySingleCell) {
197                 this->singleCellSimReflectionPoints.append(_P_r);
198             }
199
200             //qDebug() << "ray_1_refl distance:" << QVector2D(*_RX - _imageTX).length();
201             ray_1_reflection->distance = QVector2D(*_RX-_imageTX).length();
202             //qDebug() << "Ray's (1refl) total coeffs:" <<
                ↪  ray_1_reflection->getTotalCoeffs();
203             _RX->all_rays.append(ray_1_reflection);
204         }
205
206         second_reflection:
207         QVector2D _imageTX = computeImage(_TX, wall);
208         //qDebug() << "_image:" << _imageTX.x() << _imageTX.y();
209         // 2nd reflection
210         for (Obstacle* wall_2 : this->obstacles) {
211             // check that the second wall is not the same as the first wall and that
                ↪  imageTX and RX are at the same side of this second wall
212             if (wall_2 != wall && checkSameSideOfWall(wall_2->normal,_imageTX,*_RX)) {
```

```
213                 //qDebug() << "Same side of wall imageTX and RX --- wall_2:" <<
                    ↪  wall_2->line.p1() << wall_2->line.p2() << ", imageTX:" <<
                    ↪  _imageTX.toPointF() << ", RX:" << _RX->toPointF() ;
214                 Ray* ray_2_reflection = new Ray(_TX.toPointF(),_RX->toPointF());
215
216                 QVector2D _image_imageTX = computeImage(_imageTX,wall_2);
217                 //qDebug() << "_image_image:" << _image_imageTX.x() <<
                    ↪  _image_imageTX.y();
218
219                 QVector2D _P_r_2_last =
                    ↪  calculateReflectionPoint(_image_imageTX,*_RX,wall_2);
220                 QVector2D _P_r_2_first =
                    ↪  calculateReflectionPoint(_imageTX,_P_r_2_last,wall);
221                 if (_P_r_2_last.x()==_P_r_2_first.x() &&
                    ↪  _P_r_2_last.y()==_P_r_2_first.y()) {
222                     //qDebug() << "------> P_r_2_last = P_r_2_first !!!)";
223                 }
224
225                 RaySegment* test_segment_1 = new
                    ↪  RaySegment(_image_imageTX.x(),_image_imageTX.y(),_RX->x(),_RX->y());
226                 RaySegment* test_segment_2 = new
                    ↪  RaySegment(_imageTX.x(),_imageTX.y(),_P_r_2_last.x(),_P_r_2_last.y());
227                 if (!checkRaySegmentIntersectsWall(wall_2, test_segment_1) ||
                    ↪  !checkRaySegmentIntersectsWall(wall,test_segment_2)) {
228                     //qDebug() << "ignore";
229                     // RAY DOES NOT TRULY INTERSECT THE WALL (only the wall extension)
                        ↪  ignore this two-reflections ray at this wall
230                     delete ray_2_reflection;
231                     delete test_segment_1;
232                     delete test_segment_2;
233                     continue; // break out of this forloop instance for this wall
234                 }
235                 delete test_segment_1;
236                 delete test_segment_2;
237
238                 //qDebug() << "P_r_2_first" << _P_r_2_first;
239                 //qDebug() << "P_r_2_last" << _P_r_2_last;
240
241                 QList<RaySegment*> ray_segments_2;
242                 ray_segments_2.append(new
                    ↪  RaySegment(_TX.x(),_TX.y(),_P_r_2_first.x(),_P_r_2_first.y()));
243                 ray_segments_2.append(new
                    ↪  RaySegment(_P_r_2_first.x(),_P_r_2_first.y(),_P_r_2_last.x(),_P_r_2_last.y()));
244                 ray_segments_2.append(new
                    ↪  RaySegment(_P_r_2_last.x(),_P_r_2_last.y(),_RX->x(),_RX->y()));
245
246                 ray_2_reflection->segments = ray_segments_2;
247                 addReflection(ray_2_reflection,_imageTX,_P_r_2_last,wall);
248                 addReflection(ray_2_reflection,_image_imageTX,*_RX,wall_2);
249                 checkTransmissions(ray_2_reflection,{wall,wall_2});
250
```

```
251                 if (this->showRaySingleCell) {
252                     this->singleCellSimReflectionPoints.append(_P_r_2_first);
253                     this->singleCellSimReflectionPoints.append(_P_r_2_last);
254                 }
255 
256                 //qDebug() << "ray_2_refl distance:" << QVector2D(*_RX -
                    ↪  _image_imageTX).length();
257                 ray_2_reflection->distance = QVector2D(*_RX-_image_imageTX).length();
258                 //qDebug() << "Ray's (2refl) total coeffs:" <<
                    ↪  ray_2_reflection->getTotalCoeffs();
259                 _RX->all_rays.append(ray_2_reflection);
260             }
261         }
262     }
263 }
264 
265 // compute the image position
266 QVector2D Simulation::computeImage(const QVector2D& _point, Obstacle* wall) {
267     // returns the coordinates of _point's image with wall
268     QVector2D new_origin = QVector2D(wall->line.p1()); // set origin to point1 of wall
269     //qDebug() << "new coords" << wall->line.p1();
270     QVector2D _normal = wall->normal; // normal to the wall (is normalized so it is
        ↪  relative to any origin)
271     //qDebug() << "normal" << wall->normal;
272     QVector2D new_point_coords = _point - new_origin; // initial point in new coordinates
        ↪  relative to point1 of wall
273     double _dotProduct = QVector2D::dotProduct(new_point_coords, _normal);
274     QVector2D _image_new_coords = new_point_coords - 2 * _dotProduct * _normal; // image
        ↪  point in new coordinates relative to point1 of wall
275     QVector2D _image = new_origin + _image_new_coords; // image point in absolute
        ↪  coordinates
276     //qDebug() << "image:" << _image.x() << _image.y();
277 
278     return _image;
279 }
280 
281 // compute the reflection point on the wall
282 QVector2D Simulation::calculateReflectionPoint(const QVector2D& _start, const QVector2D&
    ↪  _end, Obstacle* wall) {
283     // returns the intersection bewteen the line from _start to _end and the wall
284     QVector2D d = _end-_start;
285     //QVector2D x0(0,0); // TODO: always this ?
286     QVector2D x0 = QVector2D(wall->line.p1()); // TODO: FIXME: correct ?
287     qreal t = ((d.y()*(_start.x()-x0.x()))-(d.x()*(_start.y()-x0.y()))) /
        ↪  (wall->unitary.x()*d.y()-wall->unitary.y()*d.x());
288     QVector2D P_r = x0 + t * wall->unitary;
289     return P_r;
290 }
291 
292 void Simulation::addReflection(Ray* _ray, const QVector2D& _p1, const QVector2D& _p2,
    ↪  Obstacle* wall){
```

```
293     // computes the final Gamma coeff for the ray_segment's reflection with this wall, and
        ↪ adds it to this ray's coeffs list
294     QVector2D _d = _p2-_p1;
295     qreal _cos_theta_i = abs(QVector2D::dotProduct(_d.normalized(),wall->normal));
296     qreal _sin_theta_i = sqrt(1.0 - pow(_cos_theta_i,2));
297     qreal _sin_theta_t = _sin_theta_i / sqrt(wall->properties.relative_permittivity);
298     qreal _cos_theta_t = sqrt(1.0 - pow(_sin_theta_t,2));
299     complex<qreal> Gamma_coeff =
        ↪ computeReflectionCoeff(_cos_theta_i,_sin_theta_i,_cos_theta_t,_sin_theta_t, wall);
300     //qDebug() << "addReflection, Gamma_coeff:" << Gamma_coeff;
301     if (Gamma_coeff.real() != Gamma_coeff.real() || Gamma_coeff.imag() !=
        ↪ Gamma_coeff.imag()) {
302         qDebug() << "Gamma_coeff = NaN";
303         qDebug() << "_d" << _d;
304         qDebug() << "_cos_theta_i" << _cos_theta_i;
305         qDebug() << "_sin_theta_i" << _sin_theta_i;
306         qDebug() << "_sin_theta_t" << _sin_theta_t;
307         qDebug() << "_cos_theta_t" << _cos_theta_t;
308     }
309     _ray->addCoeff(Gamma_coeff);
310 }
311
312 complex<qreal> Simulation::makeTransmission(RaySegment* ray_segment, Obstacle* wall) {
313     // computes the final T coeff for the ray_segment's transmission with this wall, and
        ↪ adds it to this ray's coeffs list
314     QVector2D _d = QVector2D(ray_segment->p1())-QVector2D(ray_segment->p2());
315     qreal _cos_theta_i = abs(QVector2D::dotProduct(_d.normalized(),wall->normal));
316     qreal _sin_theta_i = sqrt(1.0 - pow(_cos_theta_i,2));
317     qreal _sin_theta_t = _sin_theta_i / sqrt(wall->properties.relative_permittivity);
318     qreal _cos_theta_t = sqrt(1.0 - pow(_sin_theta_t,2));
319
320     complex<qreal> T_coeff =
        ↪ computeTransmissionCoeff(_cos_theta_i,_sin_theta_i,_cos_theta_t,_sin_theta_t,wall);
321     if (T_coeff.real() != T_coeff.real() || T_coeff.imag() != T_coeff.imag()) {
322         qDebug() << "Gamma_coeff = NaN";
323         qDebug() << "_d" << _d;
324         qDebug() << "_cos_theta_i" << _cos_theta_i;
325         qDebug() << "_sin_theta_i" << _sin_theta_i;
326         qDebug() << "_sin_theta_t" << _sin_theta_t;
327         qDebug() << "_cos_theta_t" << _cos_theta_t;
328     }
329     return T_coeff;
330 }
331 void Simulation::checkTransmissions(Ray* _ray, QList<Obstacle*> _reflection_walls) {
332     // checks for every segment in this ray if they intersect a wall (which isn't a wall
        ↪ already used for a reflection by this ray)
333     // if so: adds the Transmission coefficient to this ray's coeffs list
334     for (RaySegment* ray_segment : _ray->segments) {
335         for (Obstacle* wall : this->obstacles) {
336             //qDebug() << "pwall" << &wall;
337             if (!_reflection_walls.contains(wall)) { // is NOT reflection wall
```

```
338                 if (checkRaySegmentIntersectsWall(wall, ray_segment,nullptr)) {
339                     complex<qreal> T_coeff = makeTransmission(ray_segment,wall);
340                     //qDebug() << "checkTransmission, T_coeff:" << T_coeff;
341                     _ray->addCoeff(T_coeff);
342                 }
343             }
344         }
345     }
346 }
347
348 void Simulation::computeDirect(Receiver* _RX, const QVector2D& _TX)
349 {
350     // Computes the direct ray: checks all walls between RX and TX and adds
351     // their computed transmission coefficients to the direct ray list of coeffs
352     Ray* direct_ray = new Ray(_TX.toPointF(), _RX->toPointF());
353     RaySegment* _direct_line = new RaySegment(_RX->x(), _RX->y(), _TX.x(), _TX.y());
354     for (Obstacle* wall : this->obstacles) {
355         QPointF* intersection_point = nullptr; // not used
356         if (checkRaySegmentIntersectsWall(wall, _direct_line, intersection_point)) {
357             // transmission through this wall, compute the transmission coeff
358             complex<qreal> T_coeff = makeTransmission(_direct_line,wall);
359             direct_ray->addCoeff(T_coeff);
360         }
361         //else {
362         //    continue;
363         //}
364     }
365     //qDebug() << "ray_direct distance:" << QVector2D(*_RX - _TX).length();
366     direct_ray->distance = QVector2D(*_RX-_TX).length();
367     //qDebug() << "Ray's (direct) total coeffs:" << direct_ray->getTotalCoeffs();
368     direct_ray->segments = {_direct_line};
369     _RX->all_rays.append(direct_ray);
370 }
371
372 complex<qreal> Simulation::computePerpendicularGamma(qreal _cos_theta_i, qreal
    ↪  _cos_theta_t, Obstacle* wall)
373 {
374     // returns Gamma_perpendicular
375     complex<qreal> left = wall->properties.Z_m * _cos_theta_i;
376     //qDebug() << "left perpGamma:" << left.real() << "+j" << left.imag();
377     complex<qreal> right = Z_0 * _cos_theta_t;
378     //qDebug() << "right perpGamma:" << right.real() << "+j" << right.imag();
379     //complex<qreal> Gamma_perp =
    ↪  (wall->properties.Z_m*_cos_theta_i-Z_0*_cos_theta_t)/(wall->properties.Z_m*_cos_theta_i+Z_0*_cos_the
380     complex<qreal> Gamma_perp = (left - right) / (left + right);
381
382     //qDebug() << "Gamma_perp=" << QString::number(Gamma_perp.real()) << "+j" <<
    ↪  QString::number(Gamma_perp.imag());
383     return Gamma_perp;
384 }
385
```

```
386 complex<qreal> Simulation::computeReflectionCoeff(qreal _cos_theta_i, qreal _sin_theta_i,
    ↪ qreal _cos_theta_t, qreal _sin_theta_t, Obstacle* wall)
387 {
388     // returns the reflection coefficient Gamma_m
389     qreal s = wall->thickness/_cos_theta_t;
390     complex<qreal> Gamma_perpendicular = computePerpendicularGamma(_cos_theta_i,
        ↪ _cos_theta_t, wall);
391     complex<qreal> reflection_term = exp(-2.0 * wall->properties.gamma_m * s) * exp(j *
        ↪ 2.0 * beta_0 * s * _sin_theta_t * _sin_theta_i);
392     //qDebug() << "reflection_term:" << QString::number(reflection_term.real()) << "+ j"
        ↪ << QString::number(reflection_term.imag());
393     complex<qreal> Gamma_m = Gamma_perpendicular - (1.0 - pow((Gamma_perpendicular), 2)) *
        ↪ (Gamma_perpendicular * reflection_term) / (1.0 - pow((Gamma_perpendicular), 2) *
        ↪ reflection_term);
394     //qDebug() << "Gamma_m:" << QString::number(Gamma_m.real()) << "+ j" <<
        ↪ QString::number(Gamma_m.imag());
395
396     return Gamma_m;
397 }
398
399 complex<qreal> Simulation::computeTransmissionCoeff(qreal _cos_theta_i, qreal
    ↪ _sin_theta_i, qreal _cos_theta_t, qreal _sin_theta_t, Obstacle* wall)
400 {
401     // returns the transmission coefficient T_m
402     //qDebug() << "_cos_theta_i" << _cos_theta_i;
403     //qDebug() << "_sin_theta_i" << _sin_theta_i;
404     //qDebug() << "_sin_theta_t" << _sin_theta_t;
405     //qDebug() << "_cos_theta_t" << _cos_theta_t;
406     qreal s = wall->thickness/_cos_theta_t;
407     //qDebug() << "s" << s;
408     complex<qreal> perpGamma = computePerpendicularGamma(_cos_theta_i, _cos_theta_t,
        ↪ wall);
409     //qDebug() << "perpGamma" << perpGamma.real() << "+j" << perpGamma.imag();
410     complex<qreal> T_m =
        ↪ ((1.0-pow(perpGamma,2))*exp(-(wall->properties.gamma_m)*s))/(1.0-(pow(perpGamma,2)*exp(-2.0*(wall->pr
411     //qDebug() << "TransmissionCoeff=" << QString::number(T_m.real()) << "+j" <<
        ↪ QString::number(T_m.imag());
412     return T_m;
413 }
414
415 bool Simulation::checkSameSideOfWall(const QVector2D& _normal, const QVector2D& _TX, const
    ↪ QVector2D& _RX) {
416     // returns true if _TX and _RX are on the same side of the wall (using the wall's
        ↪ normal vector)
417     // must be same sign to be true:
418     bool res = (QVector2D::dotProduct(_normal,_RX)>0 ==
        ↪ QVector2D::dotProduct(_normal,_TX)>0);
419     return res;
420 }
421
```

```
422 bool Simulation::checkRaySegmentIntersectsWall(const Obstacle* wall, RaySegment*
    ↪ ray_segment, QPointF* intersection_point) {
423     // returns true if ray_segment intersects wall
424     // the intersection_point pointer's value is set wit hthe intersection point
        ↪ coordinates if they intersect
425     int _intersection_type = ray_segment->intersects(wall->line, intersection_point); //
        ↪ also writes to intersection pointer the QPointF
426     bool intersects_wall = _intersection_type==1 ? true: false; //0: no intersection
        ↪ (parallel), 1: intersects directly the line segment, 2: intersects the infinite
        ↪ extension of the line
427     return intersects_wall;
428 }
429
430 void Simulation::showView()
431 {
432     qDebug() << "Creating graphics view...";
433     QGraphicsScene* sim_scene = createGraphicsScene();
434     this->scene = sim_scene;
435     this->view = new QGraphicsView(sim_scene); // create user's view showing the graphics
        ↪ scene
436
437     this->view->setAttribute(Qt::WA_AlwaysShowToolTips); //? maybe necessary ?
438
439     this->view->setFixedSize(990, 720);
440     this->view->scale(6, 6);
441     qDebug() << "Showing graphics view";
442     //QIcon _icon = QIcon(QDir::currentPath()+"/icon.png");
443     //view->setWindowIcon(QIcon("./assets/icon.png"));
444     view->setWindowIcon(QIcon(":/assets/icon.png"));
445     this->view->show(); // shows the graphics scene to the user
446 }
447
448 void Simulation::createCellsMatrix()
449 {
450     int max_x_count = ceil(max_x/this->resolution); // -1 ?
451     //qDebug() << "Max count of cells X:" << max_x_count;
452     int max_y_count = ceil(max_y/this->resolution); // -1 ?
453     //qDebug() << "Max count of cells Y:" << max_y_count;
454
455     qInfo() << "Creating cells matrix" << max_x_count << "x" << max_y_count << "...";
456     //qDebug() << "cells matrix initial size:" << this->cells.size();
457     for (int x_count=0; x_count < max_x_count; x_count++) {
458         //qDebug() << "Creating new line of cells_matrix...";
459         qreal x = this->resolution/2+(this->resolution*x_count);
460         QList<Receiver*> temp_list;
461         for (int y_count=0; y_count < max_y_count; y_count++) {
462             qreal y = this->resolution/2+(this->resolution*y_count);
463             temp_list.append(new Receiver(x,y,this->resolution, this->show_cell_outline));
464             //qDebug() << "cells_matrix line"<< x_count << "size:" << temp_list.size();
465         }
466         this->cells.append(temp_list);
```

```
467             //qDebug() << "cells_matrix size:" << this->cells.size();
468         }
469         qDebug() << "cells_matrix created";
470     }
471
472     Transmitter* Simulation::getBaseStation(int index)
473     {
474         if (index < 0 || index >= this->baseStations.length()) {
475             qWarning("baseStations index out of range");
476             throw std::out_of_range("baseStations index out of range");
477         }
478         //return &this->baseStations.at(index);
479         return this->baseStations.at(index);
480     }
481
482     void Simulation::deleteBaseStation(int index)
483     {
484         if (index > 0 || index < this->baseStations.size())
485         {
486             this->baseStations.erase(this->baseStations.begin()+index);
487         } else if (index == 0)
488         {
489             qDebug("Cannot delete Base Station 1");
490         } else {
491             qWarning("deleteBaseStation error: index out of range");
492             throw std::out_of_range("deleteBaseStation error: index out of range");
493         }
494     }
495
496     QList<Obstacle*>* Simulation::getObstacles()
497     {
498         return &this->obstacles;
499     }
500
501     unsigned int Simulation::getNumberOfObstacles()
502     {
503         return this->obstacles.size();
504     }
505
506     unsigned int Simulation::getNumberOfBaseStations()
507     {
508         return this->baseStations.size();
509     }
510
511     qint64 Simulation::getSimulationTime() const
512     {
513         return this->simulation_time;
514     }
515
516     QGraphicsScene *Simulation::createGraphicsScene()//std::vector<Transmitter>* TX)
517     {
```

```
518     // creates the QGraphicsScene (to give to QGraphicsView) and adds all graphics to it
519     qDebug() << "Creating graphics scene...";
520 
521     QGraphicsScene* scene = new QGraphicsScene();
522 
523     //Transmitter* TX = this->baseStations[0];
524 
525     for (QList<Receiver*> cells_line : this->cells) {
526         for (Receiver* RX : cells_line) {
527             // compute total power and set it in RX
528             qreal _rx_power = 0.0;
529             for (Transmitter* tx : this->baseStations) {
530                 qreal _pwr = RX->computeTotalPower(tx);
531                 if (_pwr > _rx_power) {
532                     // cell has more power with this transmitter
533                     _rx_power = _pwr;
534                 }
535             }
536 
537             RX->power = _rx_power;
538 
539             RX->updateBitrateAndColor();
540             QBrush _rxBrush = RX->graphics->brush();
541             if (this->showRaySingleCell) {
542                 _rxBrush.setColor(Qt::black);
543             } else {
544                 _rxBrush.setColor(RX->cell_color);
545             }
546             RX->graphics->setBrush(_rxBrush);
547 
548             // Draw RX and add its tooltip
549             float _rx_power_dBm = 10*std::log10(RX->power*1000); // TODO: correct ? *1000
                ↪  because is in Watts and need in mW :
550             //qDebug() << "RX power:" << _rx_power << "W," << _rx_power_dBm << "dBm";
551             //qDebug() << "RX bitrate:" << RX->bitrate_Mbps << "Mbps";
552 
553             qreal bitrate_Mbps = RX->bitrate_Mbps;
554             if (bitrate_Mbps >= 1000) {
555                 qreal bitrate_Gbps = bitrate_Mbps/1000;
556                 RX->graphics->setToolTip(
557                     //QString("Receiver cell:\nx=%1 y=%2\nPower: %3 mW | %4 dBm\nBitrate:
                        ↪  %5 Gbps\nDirect ray (%7 segments) coeffs list length: %6").arg(
558                     QString("Receiver cell:\nx=%1 y=%2\nPower: %3 mW | %4 dBm\nBitrate: %5
                        ↪  Gbps").arg(
559                         QString::number(RX->x()),
560                         QString::number(RX->y()),
561                         QString::number(_rx_power*1000),
562                         QString::number(_rx_power_dBm,'f',2),
563                         QString::number(bitrate_Gbps,'f',2)
564                         ////QString::number(RX->all_rays.first()->coeffsList.length()),
565                         ////QString::number(RX->all_rays.first()->segments.length())
```

```
566                         ));
567             } else {
568                 RX->graphics->setToolTip(
569                     //QString("Receiver cell:\nx=%1 y=%2\nPower: %3 mW | %4 dBm\nBitrate:
                    ↪  %5 Mbps\nDirect ray (%7 segments) coeffs list length: %6").arg(
570                     QString("Receiver cell:\nx=%1 y=%2\nPower: %3 mW | %4 dBm\nBitrate: %5
                    ↪  Mbps").arg(
571                         QString::number(RX->x()),
572                         QString::number(RX->y()),
573                         QString::number(_rx_power*1000),
574                         QString::number(_rx_power_dBm,'f',2),
575                         QString::number(bitrate_Mbps)
576                         //QString::number(RX->all_rays.first()->coeffsList.length()),
577                         //QString::number(RX->all_rays.first()->segments.length())
578                         ));
579             }
580             scene->addItem(RX->graphics);
581             //qDebug() << "RX.graphics:" << RX->graphics->rect();
582         }
583     }
584     qDebug() << "All receiver cells added to scene.";
585
586     // Draw all walls in wall_list
587     for (Obstacle* wall : this->obstacles){
588         //qDebug() << "Adding wall to scene...";
589         scene->addItem(wall->graphics);
590     }
591     qDebug() << "All walls added to scene.";
592
593     // TODO: feature : multiple transmitters ?
594     //for (Transmitter* TX : this->baseStations) {
595     //    // Draw TX and add its tooltip
596     //    TX->graphics->setToolTip(QString("Transmitter\nx=%1
    ↪  y=%2\nG_TX*P_TX=%3").arg(QString::number(TX->x()),QString::number(TX->y()),QString::number(TX->gain*
597     //    scene->addItem(TX->graphics);
598     //    //qDebug() << "TX.graphics:" << TX->graphics->rect();
599     //}
600
601     for (Transmitter* TX : this->baseStations) {
602         TX->graphics->setToolTip(QString("Transmitter:\nx=%1 y=%2\nGain: %3\nPower:
        ↪  %4W").arg(QString::number(TX->x()),QString::number(TX->y()),QString::number(TX->gain),QString::nu
603         //qDebug() << "TX.graphics:" << TX->graphics->rect();
604         scene->addItem(TX->graphics);
605         qDebug() << "Transmitter added to scene.";
606     }
607
608     if (this->showRaySingleCell) {
609         // Draw all rays (their segments) from the all_rays list
610
611         for (Ray* ray :this->cells[0][0]->all_rays) {
612             qDebug() << "Drawing ray";
```

```
613             for (QGraphicsLineItem* segment_graphics : ray->getSegmentsGraphics()) {
614                 scene->addItem(segment_graphics);
615             }
616         }
617 
618         //// --- TEST only direct ---
619         //#ifdef DEBUG_SINGLE_CELL_DIRECT_RAY
620         //for (QGraphicsLineItem* gra :
        ↪  this->cells[0][0]->all_rays.first()->getSegmentsGraphics()) {
621         //    scene->addItem(gra);
622         //}
623         //qDebug() << "number of coeffs:" <<
        ↪  this->cells[0][0]->all_rays.first()->coeffsList.length();
624         //qDebug() << "T_m=" <<
        ↪  this->cells[0][0]->all_rays.first()->coeffsList.first().real() << "+j" <<
        ↪  this->cells[0][0]->all_rays.first()->coeffsList.first().imag();
625         //#endif
626         //// ---
627 
628         qDebug() << "All rays added to scene.";
629 
630         for (QVector2D point : this->singleCellSimReflectionPoints) {
631             qreal width = 0.04;
632             QGraphicsEllipseItem* reflection_graphics = new
                ↪  QGraphicsEllipseItem(10*(point.x()-width),10*(point.y()-width),2*width*10,2*width*10);
633             reflection_graphics->setPen(QPen(Qt::green));
634             reflection_graphics->setBrush(QBrush(Qt::green));
635             reflection_graphics->setToolTip("Reflection point");
636             scene->addItem(reflection_graphics);
637         }
638 
639         qDebug() << "All reflection points added to scene.";
640     }
641 
642     addLegend(scene);
643 
644     scene->setBackgroundBrush(QBrush(Qt::black)); // black background
645 
646     qDebug() << "Scene created.";
647 
648     return scene;
649 }
650 
651 void Simulation::addLegend(QGraphicsScene* scene)
652 {
653     qDebug() << "Adding legend to scene";
654 
655     QPen legendPen(Qt::white);
656     legendPen.setWidthF(0.3);
657 
658     if (!this->showRaySingleCell) {
```

```
659
660         QLinearGradient gradient;
661         QRectF rect(6,85,138,10); // gradient rectangle scene coordinates
662         QGraphicsRectItem* gradient_graphics = new QGraphicsRectItem(rect);
663
664         gradient.setCoordinateMode(QGradient::ObjectBoundingMode);
665         gradient.setColorAt(0.0, QColor::fromRgb(255,0,0)); // blue
666         gradient.setColorAt(0.25, QColor::fromRgb(255,255,0)); // cyan
667         gradient.setColorAt(0.5, QColor::fromRgb(0,255,0)); // green
668         gradient.setColorAt(0.75, QColor::fromRgb(0,255,255)); // yellow
669         gradient.setColorAt(1.0, QColor::fromRgb(0,0,255)); // red
670         gradient.setStart(1.0, 0.0); // start left
671         gradient.setFinalStop(0.0, 0.0); // end right
672
673         QBrush gradientBrush(gradient);
674
675         gradient_graphics->setPen(legendPen);
676         gradient_graphics->setBrush(gradientBrush);
677         scene->addItem(gradient_graphics); // draw gradient rectangle
678         qreal rect_width = rect.width();
679         for (int i=0; i<5; i++) {
680             QGraphicsLineItem* small_line = new
                ↪  QGraphicsLineItem(rect.bottomLeft().x()+0.25*i*rect_width,rect.bottomLeft().y(),rect.bottomLe
681             small_line->setPen(legendPen);
682             scene->addItem(small_line);
683         }
684
685         // text for minimum of gradient
686         //QGraphicsTextItem* min_text = new QGraphicsTextItem("-90dBm");
687         QGraphicsTextItem* min_text = new QGraphicsTextItem("50Mbps");
688         min_text->setPos(rect.bottomLeft().x()-6,rect.bottomLeft().y()+1);
689         min_text->setDefaultTextColor(Qt::white);
690         min_text->setScale(0.25);
691         // text for maximum of gradient
692         //QGraphicsTextItem* max_text = new QGraphicsTextItem("-40dBm");
693         QGraphicsTextItem* max_text = new QGraphicsTextItem("40Gbps");
694         max_text->setPos(rect.bottomRight().x()-6,rect.bottomRight().y()+1);
695         max_text->setDefaultTextColor(Qt::white);
696         max_text->setScale(0.25);
697
698         scene->addItem(min_text);
699         scene->addItem(max_text);
700     } else {
701         QGraphicsTextItem* ray_colors = new QGraphicsTextItem("Green line: Direct ray\nRed
            ↪  line: One-reflection ray\nYellow line: Two-reflections ray");
702         ray_colors->setPos(50,85);
703         ray_colors->setScale(0.2);
704         ray_colors->setDefaultTextColor(Qt::white);
705         scene->addItem(ray_colors);
706     }
707
```

```
708     // axes legends :
709     QGraphicsLineItem* x_line = new QGraphicsLineItem(0,-5,15*10,-5);
710     QGraphicsLineItem* y_line = new QGraphicsLineItem(-5,0,-5,8*10);
711     x_line->setPen(legendPen);
712     y_line->setPen(legendPen);
713     scene->addItem(x_line);
714     scene->addItem(y_line);
715     QGraphicsTextItem* x_label = new QGraphicsTextItem("x");
716     x_label->setPos(-2.5,-6.8);
717     x_label->setScale(0.15);
718     x_label->setDefaultTextColor(Qt::white);
719     scene->addItem(x_label);
720     QGraphicsTextItem* y_label = new QGraphicsTextItem("y");
721     y_label->setPos(-6,-3.7);
722     y_label->setScale(0.15);
723     y_label->setDefaultTextColor(Qt::white);
724     scene->addItem(y_label);
725     for (int x=0; x<=15; x++) {
726         QGraphicsLineItem* small_line = new QGraphicsLineItem(0+(x*10),-5,0+(x*10),-6);
727         small_line->setPen(legendPen);
728         QGraphicsTextItem* x_index = new QGraphicsTextItem(QString::number(x));
729         x_index->setPos(small_line->line().x2()-1.2,small_line->line().y2()-3.5);
730         x_index->setScale(0.15);
731         x_index->setDefaultTextColor(Qt::white);
732         scene->addItem(small_line);
733         scene->addItem(x_index);
734     }
735     QGraphicsTextItem* x_unit = new QGraphicsTextItem("[m]");
736     x_unit->setPos(-4.8,-6-3.7);
737     x_unit->setScale(0.15);
738     x_unit->setDefaultTextColor(Qt::white);
739     scene->addItem(x_unit);
740     for (int y=0; y<=8; y++) {
741         QGraphicsLineItem* small_line = new QGraphicsLineItem(-5,0+(y*10),-6,0+(y*10));
742         small_line->setPen(legendPen);
743         QGraphicsTextItem* y_index = new QGraphicsTextItem(QString::number(y));
744         y_index->setPos(small_line->line().x2()-2.3,small_line->line().y2()-2);
745         y_index->setScale(0.15);
746         y_index->setDefaultTextColor(Qt::white);
747         scene->addItem(small_line);
748         scene->addItem(y_index);
749     }
750     QGraphicsTextItem* y_unit = new QGraphicsTextItem("[m]");
751     y_unit->setPos(-6-3.4,-4.4);
752     y_unit->setScale(0.15);
753     y_unit->setDefaultTextColor(Qt::white);
754     scene->addItem(y_unit);
755
756 }
```

B.3 transmitter.cpp

Ce fichier contient toute l'implémentation de la classe **Transmitter**.
Le *header file* ***.h*** définissant la classe **Transmitter** se trouve dans le Github du projet.

```
#include "transmitter.h"

Transmitter::Transmitter(qreal x, qreal y, int selector_index, QString name){
    // Transmitter object constructor
    QBrush txBrush(Qt::white);
    QPen txPen(Qt::gray);
    txPen.setWidthF(6*0.07);

    this->selector_index = selector_index;
    this->name = name;

    this->setX(x);
    this->setY(y);
    this->graphics->setToolTip(QString("Test transmitter x=%1
    ↪  y=%2").arg(this->x(),this->y()));
    this->graphics->setBrush(txBrush);
    this->graphics->setPen(txPen);
    //this->graphics->setRect(10*x-1.5,10*y-1.5,3,3);
    this->setGraphicsRect(x,y);
    this->graphics->setAcceptHoverEvents(true);
};

void Transmitter::setGraphicsRect(qreal x,qreal y)
{
    this->graphics->setRect(10*x-1.2,10*y-1.2,2.4,2.4);
}

int Transmitter::getPower_dBm() const
{
    return 10*std::log10(this->power*1000);
}

double Transmitter::getPower() const
{
    return this->power;
}

qreal Transmitter::getG_TXP_TX() const
{
    return this->gain*this->power;
}

void Transmitter::setPower_dBm(int power_dBm)
{
    // TODO: correct ?
    this->power = (10^(power_dBm/10))/1000; // dBm to Watts
```

```
46      qDebug() << "setPower:" << this->power << "Watts";
47  }
48
49  void Transmitter::changeCoordinates(QPointF new_coordinates)
50  {
51      this->setX(new_coordinates.x());
52      this->setY(new_coordinates.y());
53      this->setGraphicsRect(this->x(),this->y());
54  }
55
56  QPointF Transmitter::getCoordinates() const {
57      return this->toPointF();
58  }
```

B.4 receiver.cpp

Ce fichier contient toute l'implémentation de la classe **Receiver**.
Le *header file* ***.h*** définissant la classe **Receiver** se trouve dans le Github du projet.

```
#include "receiver.h"

#include <QBrush>
#include <QPen>
#include "parameters.h"

static constexpr qreal max_power_dBm = -40.0;
static constexpr qreal min_power_dBm = -90.0;
static constexpr qreal max_bitrate_Mbps = 40000;
static constexpr qreal min_bitrate_Mbps = 50;

Receiver::Receiver(qreal x, qreal y, qreal resolution, bool showOutline) {
    // Receiver object constructor
    QBrush rxBrush(Qt::black);
    QPen rxPen;
    if (showOutline){
        rxPen.setColor(Qt::black);
        rxPen.setWidthF(10*0.01);
    } else {
        rxPen.setColor(Qt::transparent);
        rxPen.setWidth(0);
    }

    this->setX(x);
    this->setY(y);
    this->graphics->setToolTip(QString("Receiver x=%1 y=%2").arg(this->x(),this->y()));
    this->graphics->setBrush(rxBrush);
    this->graphics->setPen(rxPen);

    ↪  this->graphics->setRect(10*(x-resolution/2),10*(y-resolution/2),10*resolution,10*resolution);
    this->graphics->setAcceptHoverEvents(true);
}
void Receiver::updateBitrateAndColor()
{
    qreal bitrate;
    qreal power_dBm = 10*std::log10(this->power*1000);
    if (this->power != this->power) {
        bitrate = 9999999999999999.9;
        this->cell_color = QColor::fromRgb(255,192,203); // pink
        qDebug() << "--- ! ERROR: Cell has NaN power ! ---";
    } else if (power_dBm > max_power_dBm) {
        bitrate = max_bitrate_Mbps; //in Mbps, 40 Gbps max from -40 dBm
        this->cell_color = QColor::fromRgb(255,0,0); // red
    } else if (power_dBm < min_power_dBm) { // 50 Mbps at -90 dBm
        bitrate = 0; //in Mbps, no connection (0 Mbps)
        this->cell_color = Qt::black; // Qt::transparent or Qt::black or Qt::darkBlue ?
```

```
46      } else {
47          // Conversion to bitrate (beware log scale)
48          qreal max_power_mW = std::pow(10.0, max_power_dBm / 10.0);
49          qreal min_power_mW = std::pow(10.0, min_power_dBm / 10.0);
50
51          //bitrate = min_bitrate_Mbps + (((this->power - min_power_mW/1000) /
            ↪  (max_power_mW/1000 - min_power_mW/1000)) * (max_bitrate_Mbps -
            ↪  min_bitrate_Mbps));
52          // OR ?
53          qreal max_bitrate_dB = 10*log10(max_bitrate_Mbps);
54          qreal min_bitrate_dB = 10*log10(min_bitrate_Mbps);
55          qreal bitrate_dB = min_bitrate_dB + (((power_dBm - min_power_dBm) / (max_power_dBm
            ↪  - min_power_dBm)) * (max_bitrate_dB - min_bitrate_dB));
56          //qDebug() << "bitrate dB:" << bitrate_dB;
57          bitrate = pow(10.0, bitrate_dB / 10.0);
58          //qDebug() << "bitrate (Mbps):" << bitrate;
59
60          //qreal value_normalized = (power_dBm - min_power_dBm) / (max_power_dBm -
            ↪  min_power_dBm);
61          // or
62          //qreal value_normalized = (this->power*1000 - min_power_mW) / (max_power_mW -
            ↪  min_power_mW);
63          // or
64          //qreal value_normalized = (qreal(bitrate) - qreal(min_bitrate_Mbps)) /
            ↪  (qreal(max_bitrate_Mbps) - qreal(min_bitrate_Mbps));
65          // or
66          qreal value_normalized = (bitrate_dB - min_bitrate_dB) / (max_bitrate_dB -
            ↪  min_bitrate_dB);
67          //qDebug() << "value_normalized:" << value_normalized;
68
69          QColor color = computeColor(value_normalized);
70          // testing in monochrome :
71          //QColor color =
            ↪  QColor::fromRgbF(value_normalized,value_normalized,value_normalized);
72          //qDebug() << "cell color:" << color;
73
74          this->cell_color = color;
75      }
76      //// DEBUG:
77      ////this->cell_color = Qt::black; // set ALL cells to black
78
79      this->bitrate_Mbps = bitrate;
80  }
81
82  QColor Receiver::computeColor(qreal value)
83  {
84      // Maps a normalized value to a 5 color gradient
85
86      // Define colors for the heatmap gradient
87      QColor colors[] = {
88          QColor(0, 0, 255),   // Blue
```

```
89          QColor(0, 255, 255), // Cyan
90          QColor(0, 255, 0),   // Green
91          QColor(255, 255, 0), // Yellow
92          QColor(255, 0, 0)    // Red
93      };
94
95      // Determine which two colors to interpolate between
96      int index1 = value * 4;
97      int index2 = index1 + 1;
98
99      // Calculate interpolation factor
100     qreal factor = (value * 4) - index1;
101
102     // Interpolate between the two colors
103     QColor color = colors[index1].toRgb();
104     QColor next_color = colors[index2].toRgb();
105
106     int red = color.red() + factor * (next_color.red() - color.red());
107     int green = color.green() + factor * (next_color.green() - color.green());
108     int blue = color.blue() + factor * (next_color.blue() - color.blue());
109
110     return QColor(red, green, blue);
111 }
112
113
114 qreal Receiver::computeTotalPower(Transmitter* transmitter) // returns final total power
    ↪  computation for this RX
115 {
116     qreal res = 0;
117     for (Ray* ray : this->all_rays) {
118         res+=ray->getTotalCoeffs(); // sum of all the rays' total coefficients and exp
            ↪  term, returns |G*G*T|^2 *|exp|^2
119     }
120     //qDebug() << "computeTotalPower res+=ray->getTotalCoeffs" << res;
121     // multiply by the term before the sum:
122     res *= (60*pow(lambda,2))/(8*pow(M_PI,2)*Ra)*transmitter->gain*transmitter->power;
123
124     if (res != res) {
125         qDebug() << "computeTotalPower: NaN !!!";
126     }
127
128     //qDebug() << "computeTotalPower:" << res;
129     return res;
130 }
131
132
133
```

B.5 obstacle.cpp

Ce fichier contient toute l'implémentation de la classe **Obstacle** pour les murs. Le *header file* ***.h*** définissant la classe **Obstacle** se trouve dans le Github du projet.

```
#include "obstacle.h"

#include <QPen>
#include <QVector2D>
#include "parameters.h"

/*
 * The floorplan is 8m x 15m (x=0,y=0 top left to x=15,y=8 bottom right) :
 *  ___________________________
 * |      |    |      |      |
 * |      |    |      |      |
 * |      |    |      |      |
 * |      |_  _|___  _|      |
 * |______                   /
 * |       _________        /
 * |      |##I     |      /
 * |      |##I     |     /
 * -----------     ------
 *
 * _ or | : wall (concrete or drywall)
 * # : lift (metal walls)
 * I : metal door
 * / : glass window
 *
*/

Obstacle::Obstacle(QVector2D start, QVector2D end, ObstacleType material, qreal
↪  thickness)//, int id)
{
    // Wall object cosntructor
    //qDebug("Creating wall...");
    this->material = material;
    //this->id = id;
    this->thickness = thickness; // in meters
    this->line = QLineF(start.x(),start.y(), end.x(), end.y());
    QLineF graphics_line = QLineF(10*start.x(), 10*start.y(), 10*end.x(), 10*end.y());
    //qDebug() << "Wall" << id << "line:" << this->line ;
    QLineF normal_line = this->line.normalVector();
    //qDebug() << "Line" << id << "normal:" << normal_line;
    this->normal = QVector2D(normal_line.dx(),normal_line.dy()).normalized(); // !
    ↪  normalized !
    //qDebug() << "Wall" << id << "normal:" << this->normal;
    QLineF unit_line = this->line.unitVector();
    //qDebug() << "Line" << id << "unitary:" << unit_line;
    this->unitary = QVector2D(unit_line.dx(),unit_line.dy()).normalized(); // ! normalized
    ↪  !
```

```
44      //qDebug() << "Wall" << id << "unitary:" << this->unitary;
45      //qDebug("Setting Wall graphics line...");
46      this->graphics->setLine(graphics_line);
47
48      QPen pen(Qt::gray);
49      //pen.setWidthF(10*0.2);
50
51      switch (this->material) {
52      case BrickWall:
53          pen.setColor(Qt::darkRed);
54          pen.setWidthF(10*0.18);
55          this->properties.relative_permittivity = 3.95;
56          this->properties.conductivity = 0.073;
57          break;
58      case Window:
59          pen.setColor(Qt::white);
60          pen.setWidthF(10*0.08);
61          this->properties.relative_permittivity = 6.3919;
62          this->properties.conductivity = 0.00107;
63          break;
64      case MetalWall:
65          pen.setColor(Qt::gray);
66          pen.setWidthF(10*0.1);
67          this->properties.relative_permittivity = 1;
68          this->properties.conductivity = 1e7;
69          break;
70      case DryWall:
71          pen.setColor(Qt::lightGray);
72          pen.setWidthF(10*0.12);
73          this->properties.relative_permittivity = 2.7;
74          this->properties.conductivity = 0.05349;
75          break;
76      case ConcreteWall:
77          pen.setColor(Qt::darkMagenta);
78          pen.setWidthF(10*0.2);
79          this->properties.relative_permittivity = 6.4954;
80          this->properties.conductivity = 1.43;
81          break;
82      default:
83          pen.setColor(Qt::white);
84          pen.setWidthF(10*0.12);
85          this->properties.relative_permittivity = 1e-10;
86          this->properties.conductivity = 1e-10;
87          break;
88      }
89      this->graphics->setPen(pen);
90
91      this->properties.epsilon = epsilon_0 * this->properties.relative_permittivity;
92      complex<qreal> epsilon_tilde = this->properties.epsilon - j
        ↪  *(this->properties.conductivity / omega);
93      this->properties.Z_m = sqrt(mu_0 / epsilon_tilde);
```

```
94
95      this->properties.gamma_m = j * omega * sqrt(mu_0 * epsilon_tilde);
96
97      //qDebug("Wall created.");
98      //qDebug() << this->material; // 1: glass, 4: concrete, 2: metal, 3: drywall, 0:
        ↪  brick
99      //qDebug() << "Z_m" << this->properties.Z_m.real() << "+j" <<
        ↪  this->properties.Z_m.imag();
100     //qDebug() << "gamma_m" << this->properties.gamma_m.real() << "+j" <<
        ↪  this->properties.gamma_m.imag();
101 }
102
103 ObstacleType Obstacle::getMaterial()
104 {
105     return this->material;
106 }
```

B.6 ray.cpp

Ce fichier contient toute l'implémentation de la classe **Ray**.
Le *header file* ***.h*** définissant la classe **Ray** se trouve dans le Github du projet.

```
#include "ray.h"

#include <QPen>
#include "parameters.h"

Ray::Ray(QPointF start, QPointF end) {
    // Ray object constructor
    this->start=start;
    this->end=end;
}

void Ray::addCoeff(complex<qreal> coeff) {
    // add a Transmission or Reflection coefficient to this ray's list of coeffs
    //qDebug() << "Adding coeff to ray:" << coeff_module;
    // which one to use ? :
    if (coeff.real() != coeff.real() || coeff.imag() != coeff.imag() ||coeff != coeff) {
        qDebug() << "NaN coeff";
    }
    this->coeffsList.append(coeff);
    //this->totalCoeffs*=pow(abs(coeff),2);
}

qreal Ray::getTotalCoeffs() {
    // returns the total product of all of the ray's coefficients multiplied by the
    ↪  exponent term
    //qreal res = totalCoeffs;
    //res *= pow(abs(exp(-j*beta_0*this->distance)/this->distance),2); // exp term
    ////qDebug() << "getTotalCoeffs ray:" << res;
    //return res;

    qreal distance_ray = getTotalDistance();
    //qDebug() << distance_ray;
    qreal res = 1;
    complex<qreal> res_q = 1;
    for (complex<qreal> coeff : this->coeffsList) {
        res_q*=coeff; // all coeffs
    }
    res = pow(abs(res_q),2);
    res*=pow(abs(exp(-j*beta_0*distance_ray)/distance_ray),2); // exp term
    //qDebug() << "computeAllCoeffs:" << res;
    if (res != res) {
        qDebug() << "NaN Total Coeff";
    }
    return res;
}

```

```
46  QList<QGraphicsLineItem*> Ray::getSegmentsGraphics(){
47      // returns this ray's graphics: list of its segment's QGraphicsItems
48      //qDebug() << "Getting ray segments graphics";
49      QPen ray_pen;
50      ray_pen.setWidthF(0.1);
51      // set ray graphics color depending on number of reflections
52      //qDebug() << "This ray has" << this->segments.length() << "segments, so" <<
        ↪  this->segments.length()-1 << "reflections";
53      this->num_reflections=this->segments.length()-1;
54      switch (this->num_reflections){
55      case 0: // no reflections: direct ray
56          //qDebug() << "This ray is direct";
57          ray_pen.setColor(Qt::green);
58          break;
59      case 1: // 1 reflection
60          //qDebug() << "This ray has 1 reflection";
61          ray_pen.setColor(Qt::red);
62          break;
63      case 2: // 2 reflection
64          //qDebug() << "This ray has 2 reflections";
65          ray_pen.setColor(Qt::yellow);
66          break;
67      }
68      QList<QGraphicsLineItem*> ray_graphics;
69      for (RaySegment* ray_segment: this->segments) {
70          //for (int i=0; i<this->segments.length(); i++) {
71          //qDebug() << "Adding this segment to list ray_graphics";
72          ray_segment->graphics->setPen(ray_pen);
73          //qDebug() << "before" << ray_segment->graphics;
74          ray_graphics.append(ray_segment->graphics);
75          //qDebug() << "after:" << ray_graphics;
76      }
77      return ray_graphics;
78  }
79
80  qreal Ray::getTotalDistance() {
81      // returns the total distance of this ray (which is the either the sum of the
        ↪  distances of each segments,
82      // or the distance between the last image and RX)
83
84      // TODO: ?
85      //qreal d = 0;
86      //for (RaySegment* segment : this->segments) {
87      //    d += segment->distance;
88      //}
89      //return d;
90
91      // ray's distance has already been changed
92      //qDebug() << "Ray getTotalDisctance:" << this->distance;
93      return this->distance;
```

```
94 }
```

B.7 raysegment.cpp

Ce fichier contient toute l'implémentation de la classe **RaySegment**.
Le *header file* ***.h*** définissant la classe **RaySegment** se trouve dans le Github du projet.

```
#include "raysegment.h"

RaySegment::RaySegment(qreal start_x, qreal start_y, qreal end_x, qreal end_y) {
    // RaySegment object constructor
    this->setLine(start_x, start_y, end_x, end_y);
    this->graphics->setLine(start_x*10, start_y*10, end_x*10, end_y*10);
    this->distance = this->length();
}
```

B.8 parameters.h

Ce fichier contient les paramètres de la simulation à définir lors de la compilation.

```
#ifndef PARAMETERS_H
#define PARAMETERS_H

#include <complex>
#include <QtTypes>

using namespace std;

constexpr complex<qreal> j(0, 1); // useful
constexpr qulonglong c = 3*1e8;
constexpr qreal epsilon_0 = 8.854187817e-12;
constexpr qreal mu_0 = 1.256637e-6;//4 * M_PI * 1e-7; // 1.256637e-6; // ?
constexpr qreal freq = 60e9; // 60 GHz
constexpr qreal omega = 2 * M_PI * freq; // pulse
constexpr qreal wavelength = c/freq;
constexpr qreal lambda = wavelength;
const qreal Z_0 = sqrt(mu_0 / epsilon_0); // vacuum impedance

constexpr qreal max_x = 15;
constexpr qreal min_x = 0;
constexpr qreal min_y = 0;
constexpr qreal max_y = 8;
// resolution is set in simulation->resolution, user chosen at runtime

constexpr qreal beta_0 = 2*M_PI*freq/c; // beta

constexpr qreal G_TX = 1.7; // transmitter antenna gain, 1.64 or 1.7
constexpr qreal P_TX = 0.1; // transmitter power in Watts
constexpr qreal P_TX_dBm = 20; // transmitter power in dBm

#endif //PARAMETERS_H
```

B.9 algorithme.cpp

Ce fichier contient l'implémentation de l'algorithme d'optimisation de placement de station de base, selon la moyenne la plus haute des puissances de chaque cellule, et selon le nombre minimum de cellules en dessous du seuil pour un débit binaire minimum.

```
#include "algorithme.h"
#include <QDebug>
#include <limits>
#include <cmath>

void runAlgo() {

    qreal resolution = 0.5; // default is 0.5, or any resolution wanted

    int number_of_tx_x = 15-1;
    int number_of_tx_y = 8-1;

    QPointF bestSimAveragePowerTX;
    QPointF bestSimLessDisconnectedCellsTX;

    qreal best_average_pow_dbm = -99999;
    int min_num_of_disconnected_cells = 999999;

    qDebug() << "num of TXs:" << number_of_tx_x*number_of_tx_y;
    int g=0;
    for (int x=0; x<number_of_tx_x; x++) {
        for (int y=0; y<number_of_tx_y; y++) {
            Simulation sim = Simulation(false);
            sim.resolution = resolution;
            // Créez et ajoutez une base station à la simulation
            Transmitter* baseStation = new Transmitter(0.5+qreal(x),
            ↪  0.5+qreal(y),0,"opti_BS");

            sim.baseStations = {};
            sim.baseStations.append(baseStation);

            sim.run(nullptr); // run sim

            bool better = false;
            g++;
            qreal total_pow_dbm = 0;
            int num_of_disconnected_cells = 0;
            int i = 0;
            for (QList<Receiver*> cell_line : sim.cells) {
                for (Receiver* cell : cell_line) {
                    i++;

                    qreal _rx_power = cell->computeTotalPower(sim.baseStations[0]);
                    cell->power = _rx_power;
                    cell->updateBitrateAndColor();
```

```
45
46                      total_pow_dbm += 10*log10(cell->power*1000);
47                      if (10*std::log10(cell->power*1000) < -90) {
48                          num_of_disconnected_cells++;
49                      }
50                      //qDebug() << "Cell" << i;
51                  }
52              }
53              qreal average_pow_dbm;
54              if ((sim.cells.length()) * (sim.cells[0].length()) != 0) {
55                  average_pow_dbm = total_pow_dbm/((sim.cells.length()) *
                    ↪  (sim.cells[0].length()));
56              } else {
57                  average_pow_dbm = 0;
58              }
59              if (average_pow_dbm > best_average_pow_dbm) {
60                  // this sim is a better sim
61
62                  best_average_pow_dbm = average_pow_dbm;
63                  bestSimAveragePowerTX = sim.baseStations[0]->getCoordinates();
64                  better = true;
65              }
66              if (num_of_disconnected_cells < min_num_of_disconnected_cells) {
67
68                  min_num_of_disconnected_cells = num_of_disconnected_cells;
69                  bestSimLessDisconnectedCellsTX = sim.baseStations[0]->getCoordinates();
70                  better = true;
71              }
72              qDebug() << "Sim" << g << ", TX" << sim.baseStations[0]->getCoordinates() <<
                ↪  ", average power:" << average_pow_dbm << "dBm";
73              //if (!better) {
74              //if (true) {
75              //    for (QList<Receiver*> list : sim->cells) {
76              //        qDeleteAll(list);
77              //    }
78              //    qDeleteAll(sim->obstacles);
79              //    qDeleteAll(sim->baseStations);
80              //    delete sim;
81              //}
82          }
83      }
84      qDebug() << "for loop finished";
85
86      qDebug() << "Best sim average power: transmitter at" << bestSimAveragePowerTX;
87      qDebug() << "Best average power:" << best_average_pow_dbm << "dBm";
88      qDebug() << "-----------";
89      qDebug() << "Best sim min num of disconnected cells: transmitter at" <<
        ↪  bestSimLessDisconnectedCellsTX;
90      qDebug() << "Min num of disconnected cells:" << min_num_of_disconnected_cells;
91
92      //qDebug() << "best sim p*:" << &bestSimAveragePower;
```

```
93 }
```

B.10 Autres parties du code

D'autres parties du code sont disponibles sur le Github du projet :
`https://github.com/LucasPlacentino/802_11ay-Raytracing-Simulator`.

Ceci inclut :
- **mainwindow.cpp** (et son **.h**) : classe s'occupant de l'interface utilisateur.
- **tp4.cpp** : code reprenant l'implémentation de la simulation exclusivement utilisée pour le TP4, qui a servi de base initiale avant d'étendre les fonctionnalités au projet global.
- **CMakeLists.txt** : le CMake généré et utilisé par *QtCreator*.

Bibliographie

[De 24] Philippe De Doncker. “Syllabus ELEC-H304 - Physique des télécommunications”. https://uv.ulb.ac.be/course/view.php?id=116686. 2024.

[Hor23] Francois Horlin. “Syllabus ELEC-H311 - Signaux et systèmes de télécommunications”. https://uv.ulb.ac.be/course/view.php?id=116698. 2023.

[PYP08] Yosef Pinhasi, Asher Yahalom et Sergey Petnev. “Propagation of ultra wideband signals in lossy dispersive media”. In : 2008, p. 1-10. isbn : 978-1-4244-2097-1. doi : 10.1109/COMCAS.2008.4562803.