Université libre de Bruxelles

ELEC-H304

Physique des Télécommunications

Rapport de projet

Auteurs :
Matin Joël
Moli David

Professeur :
Philippe De Doncker

Année académique 2022-2023

Table des matières

Description générale

Ce projet a pour but d'implémenter un code de calcul de propagation des ondes électromagnétiques de type *ray-tracing* permettant de calculer la puissance reçue par un dispositif connecté à une station de base (BS) 5G. La carte considérée est celle de l'usine *Meet-A*

Pour ce faire, une modélisation théorique des phénomènes physiques impliqués a été apportée ainsi qu'un code de calcul pour en faire une simulation.

1 Modélisation Théorique

Cette section a pour but de présenter toute la modélisation théorique apportée afin de réaliser le projet.

Le but étant de calculer la puissance transmisse dans l'entièreté de l'usine, une formule permettant la calculer a été considérée :

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

Avec :

1. P_{RX} : La puissance (en Watt)
2. R_a : La résistance de l'antenne (Ω)
3. ϕ_n : Angle azimutal émetteur/récepteur
4. $\vec{h_e}$: hauteur équivalente de l'émetteur
5. $\underline{\vec{E_n}}(\vec{r})$: Phaseur champ électrique
6. θ_n : Angle longitudinal émetteur/récepteur

Hypothèses

Une série d'hypothèses on été considérée afin de facilité la modélisation, elles sont au nombre de 3 :

1. Le problème est envisagé sous un aspect exclusivement bidimensionnel, où la station de base et le récepteur se situent à une altitude équivalente et où seules les ondes se propageant dans le plan horizontal sont prises en considération.
2. Les ondes comportant plus de trois réflexions ne sont pas prises en considération, de même que la diffraction n'est pas prise en compte.
3. Le champ électrique est polarisé verticalement (selon l'axe z) et tous les murs lui sont parallèles.

Modélisation des émetteurs

Les émetteurs peuvent être de 3 types :

1. TX1 : Des dipôles $\lambda/2$ verticaux, sans pertes, émettant avec une puissance de 20dBm.
2. TX2 : Des dipôles $\lambda/2$ verticaux, sans pertes, émettant avec une puissance de 35dBm, et ne pouvant être placés que dans un environnement extérieur.
3. TX3 : Un réseau d'antennes avec réflecteur d'une puissance totale de 35dBm.

Chaque émetteur possède un certain gain, il est égale à :

$$G(\phi, \theta) = \eta D(\phi, \theta) = \eta \frac{U(\phi, \theta)}{P_{ar}/4\pi} \tag{2}$$

Le rendement η est considéré comme unitaire car les antennes sont sans pertes et pour une antenne $TX1$ et $TX2$ on a : $D = 1.7 = G$
Pour la $TX3$, son gain est donné par :

$$G(\theta) = G_{max}[dB] - 12(\frac{\phi - \delta}{\phi_{3dB}})^2 \tag{3}$$

avec $G_{max}[dB] = 21.5836dB$ et $\phi_{3dB} = 30^o$

La hauteur équivalente $\vec{h_e}$ est calculé pour chaque émetteur et vaut (en norme) :

$$|\vec{h_e}| = -\frac{\lambda}{\pi} \tag{4}$$

$$= -\frac{c}{f\pi} \tag{5}$$

Où f est la fréquence de chaque émetteur et est égale à $26GHz$ pour tous les émetteurs.

Le calcul des différents champs et leur modélisation son montrés à la section **(3)**

2 Description du code

Le langage de programmation utilisé pour ce code est le *C++* en raison de ses performances supérieures par rapports aux autres langages existants.

Ce code de calcul est structuré en plusieurs fichiers distincts dans le but d'en alléger la complexité. Ceux-ci sont :

`main.cpp` : Contient la fonction principale `main`

`constantes.h` : Contient l'intégralité des constantes utilisées

`init_map_tp.cpp` : Relatif à la map

`formule.cpp` : Contient les formules mathématique et autres opérations vectorielles

`ondes.cpp` : Relatif aux calculs des différents coefficients des ondes

`raytracing.cpp` : Relatif aux calculs des différents champs et puissances

`interface_graphique.cpp` : Relatif à l'interface grapghique

Chaque fichier `.cpp` est accompagné d'un fichier `.h` à l'exception de `formule.cpp`.

Afin de modéliser au mieux les différentes interactions des ondes sur la map, une série de structures ont été créées :

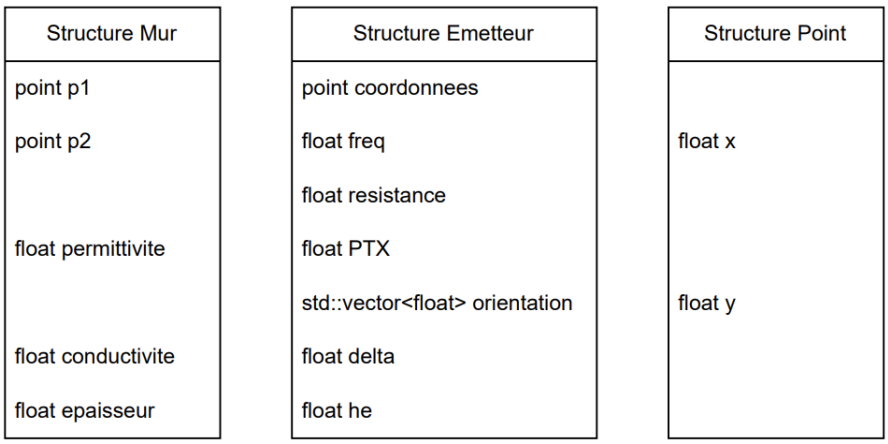

FIGURE 1 – Structures utilisées pour la mise en oeuvre du code de calcul

La fonction `main` fait appel à la fonction `puissanceMap` sur tous les points préalablement instanciés de la carte. Ceci dans le but de calculer, en tout point, la puissance reçu via l'émetteur. Cette fonction faisant elle-même appel à une série de fonctions interagissant entre-elles.

2.1 Interactions Ondes/Murs

Lorsqu'un émetteur est instancié, ce dernier émet des ondes dans différentes directions de la carte. Ces ondes interagissent via différentes manières avec les murs présents dans cette carte. Une onde peut soit être réfléchie ou transmisse à travers le mur. De plus, plusieurs réflexions sont possibles, rendant ainsi la complexité du code plus élevée.

Une série de fonction relatives aux réflexion simple, transmissions et réflexion doubles ont donc été établie afin d'en déterminer le champ que ces dernières provoquaient.

2.2 Schéma du Code de calcul

En prenant compte de ces différentes interactions, il a été possible de structurer le code de la façon suivante :

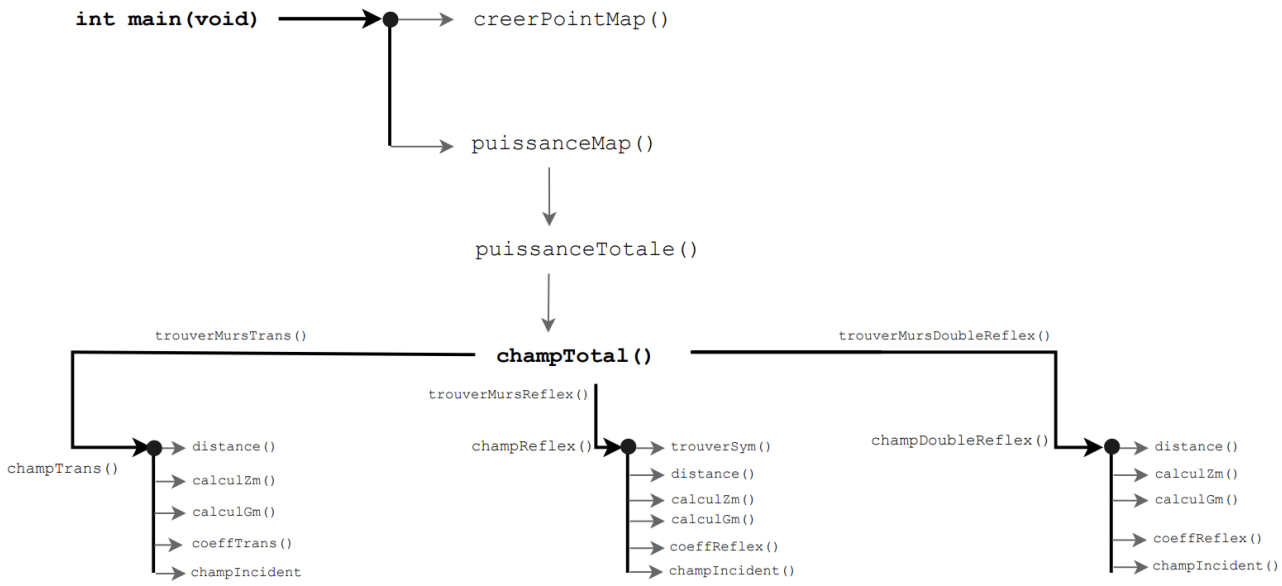

FIGURE 2 – Schéma simplifié du code de calcul

Il convient de préciser que ce schéma se lit de manière verticale, et que les fonctions alignées horizontalement sont exécutées en simultané.

2.3 Résultats

En plus d'un code de calcul, une interface graphique a été réalisée. Ceci dans le but de visualiser les résultats obtenus. A titre d'exemple, voici les résultats en considérant un émetteur TX_3 avec une orientation de 0° et positionné en (-10m ; 0.5m) :

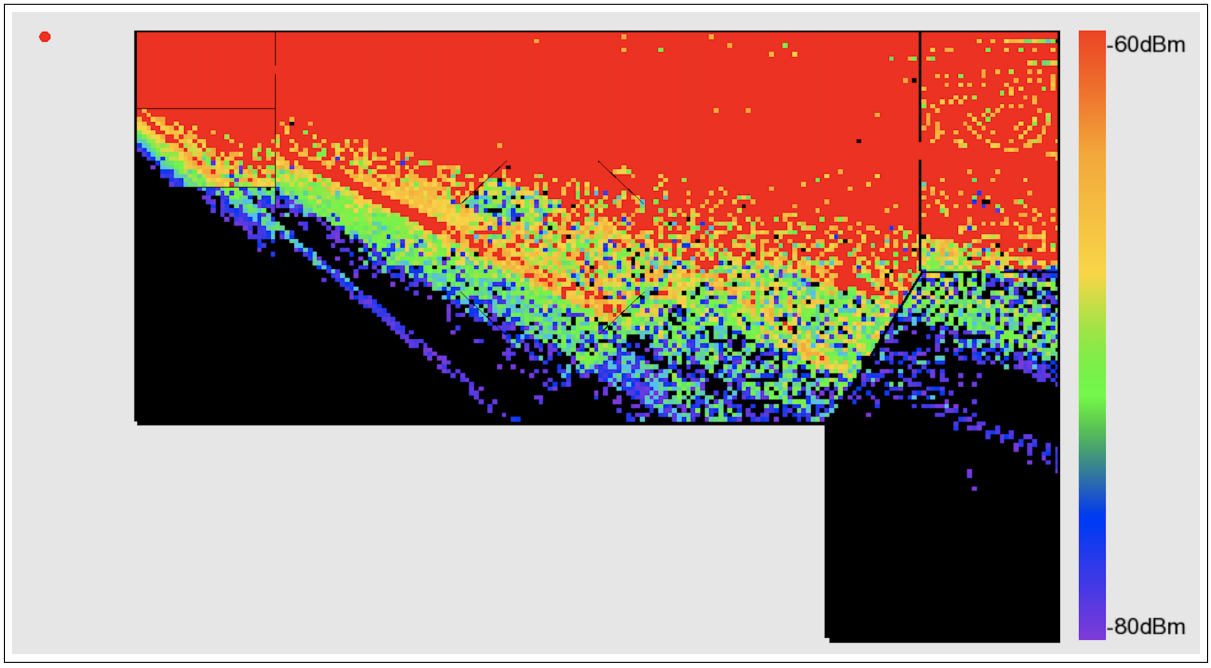

FIGURE 3 – puissance reçue depuis une station de base 5G NR 26GHz

2.4 Optimisation

Une série d'optimisation ont été pensées afin d'accélérer le temps d'exécution du code. Elles sont au nombre de 7 :

1. Limitation des fonctions non-linéaires
2. Parallélisation du code
3. Limitation des fonctions trigonométriques
4. Usage des `floats`
5. Usage de la récursivité
6. Usage de structures
7. Usage des pointeurs

Limitation des fonctions non-linéaires

Les fonctions non-linéaires ont été limitées dans le code dû à leur temps de calcul conséquent.

Parallélisation du code

Les différentes interactions ondes/murs étant indépendantes les unes des autres (une réflexion simple n'impliquant en rien une réflexion double par exemple), il a été possible de paralléliser les calculs des différents champs. Cette parallélisation a été faite dans la fonction `champTotal` à l'annexe **A.4**. Dans cette fonction, on peut y voir 3 `threads` créés afin d'éxecuter en parallèle les 3 calculs de champs possibles.

Limitation des fonctions trigonométriques

Les fonctions trigonométriques ont été évitées dans le code car celles-ci demande un long temps de calcul relativement aux autres types de fonctions. Par exemple, dans la fonction `calculTrigoReflex` de l'annexe **A.2**, pour calculer différents *sinus* et *cosinus* de différents angles, il a été préférable d'utiliser le produit scalaire plutôt que les fonctions trigonométriques.

Usage des `floats`

Les `floats` ont été priviliégiés par rapport aux `doubles` dans ce projet afin de réduire le temps de calcul. La contrepartie de ceci est que les résultats auront une précision légèrement plus faible.

Usage de la récursivité

Une fonction récursive a été implémentée afin de réduire la complexité du code. Celle-ci se trouve dans le fichier `raytracing` à l'annexe **A.4** et est nommée `verifReflex`. On peut constater que la fonction s'appelle elle-même mais avec des arguments différents. Ceci permet de ne pas avoir à écrire des lignes de codes supplémentaires.

Usage des structures

L'usage des structures a été privilégié en faveur des classes en raison de sa plus grande efficacité en termes de performance.

Usage des pointeurs

Les pointeurs on été utiles afin de minimiser la copie des variables en mémoires et ainsi la manipuler à bon escient. De plus, l'usage de pointeur permet de ne pas répéter certains calculs. Dans la fonction `calculTrigoReflex` (annexe **A.2**), les adresses de certaines variables sont passées en argument et leur contenu y est modifié à même la fonction.

3 Validation du Simulateur

La validation du simulateur s'est faite à l'aide d'un schéma simplifié de hangar.

Les paramètres pris en compte sont les suivants :

Modélisation émetteur (*TX*) :
- Antenne λ /2 en polarisation verticale.
- Résistance émetteur $R_{TX} = 73\ \Omega$
- Fréquence émetteur $f_{TX} = 868.3\ MHz$
- Pulsation émetteur $\omega_{TX} = 2\pi f_{TX}$
- Gain émetteur $G_{TX} = 1.7$
- Puissance d'émission émetteur $P_{TX} = 0.1$W

Modélisation murs :
- Permittivité relative des murs $\epsilon_r = 4.8$
- Conductivité des murs $\sigma = 0.018\ S/m$
- Epaisseur des murs $e = 15$cm

Les rayons obtenus sont les suivants :

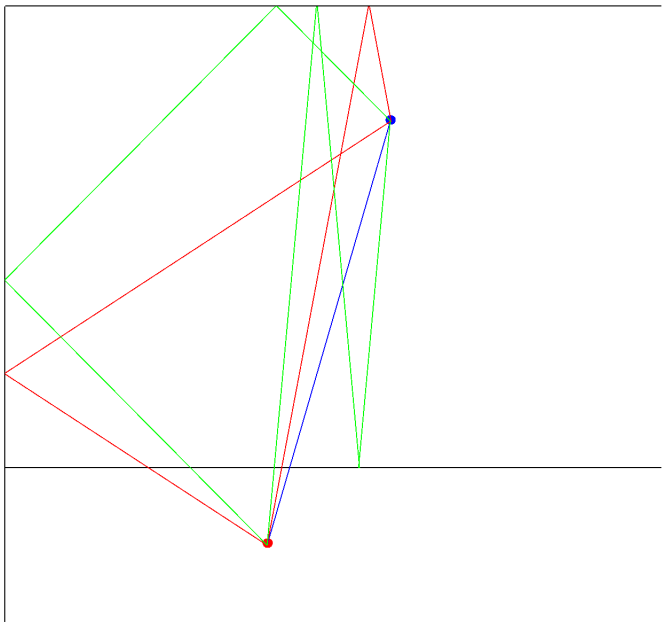

FIGURE 4 – Tracé des rayons par le simulateur dans un cas simplifié

Avec :

— : Reflexion simple — : Transmission — : Double Reflexion

Les Champs obtenus sont :

Champ	Valeur (V/m)	Point de Réflexion 1	Point de Réflexion 2
$E_{Transmission}$	3.69 10^{-3} - 1.59 10^{-3} i	/	/
$E_{Reflexion1}$	-5.43 10^{-4} - 4.53 10^{-4} i	(0 ; 32.29)	/
$E_{Reflexion2}$	-6.58 10^{-4} + 1.66 10^{-4} i	(44.35 ; 80)	/
$E_{DoubleReflexion1}$	2.675 10^{-5} - 4.455 10^{-4} i	(0 ; 44.43)	(33.06 ; 80)
$E_{DoubleReflexion2}$	5.12 10^{-5} + 5.97 10^{-5} i	(38 ; 80)	(43.14 ; 20)

3.1 Vérification des Résultats

Il est question ici de procéder au calcul de la grandeur E_{DR1}.

Les constantes considérées pour le calcul sont les suivantes :

1. ϵ_0 : La permittivité du vide
   = 8.854187 10^{-12} F/m
2. μ_0 : La perméabilité du vide
   = 4π 10^{-7} H/m
3. Z_0 : L'impédance du vide = $\sqrt{\frac{\mu_0}{\epsilon_0}}$
   = 120π Ω
4. c : La vitesse de la lumière
   = 299792458 m/s

Le champ issu d'une double réflexion est donné par :

$$E_{DR} = \Gamma_{m1}\Gamma_{m2}T_m \sqrt{60 G_{TX} P_{TX}} \ \frac{e^{-j\beta d}}{d} \tag{6}$$

Avec :

1. Γ_{mi} : Le coefficient de la i ème réflexion
2. T_m : Le coefficient de Transmission
3. β : Le nombre d'onde = $\frac{\omega_{TX}}{c}$
4. d : La distance parcourue par l'onde

Le schéma de la double réflexion peut être représenté comme suit :

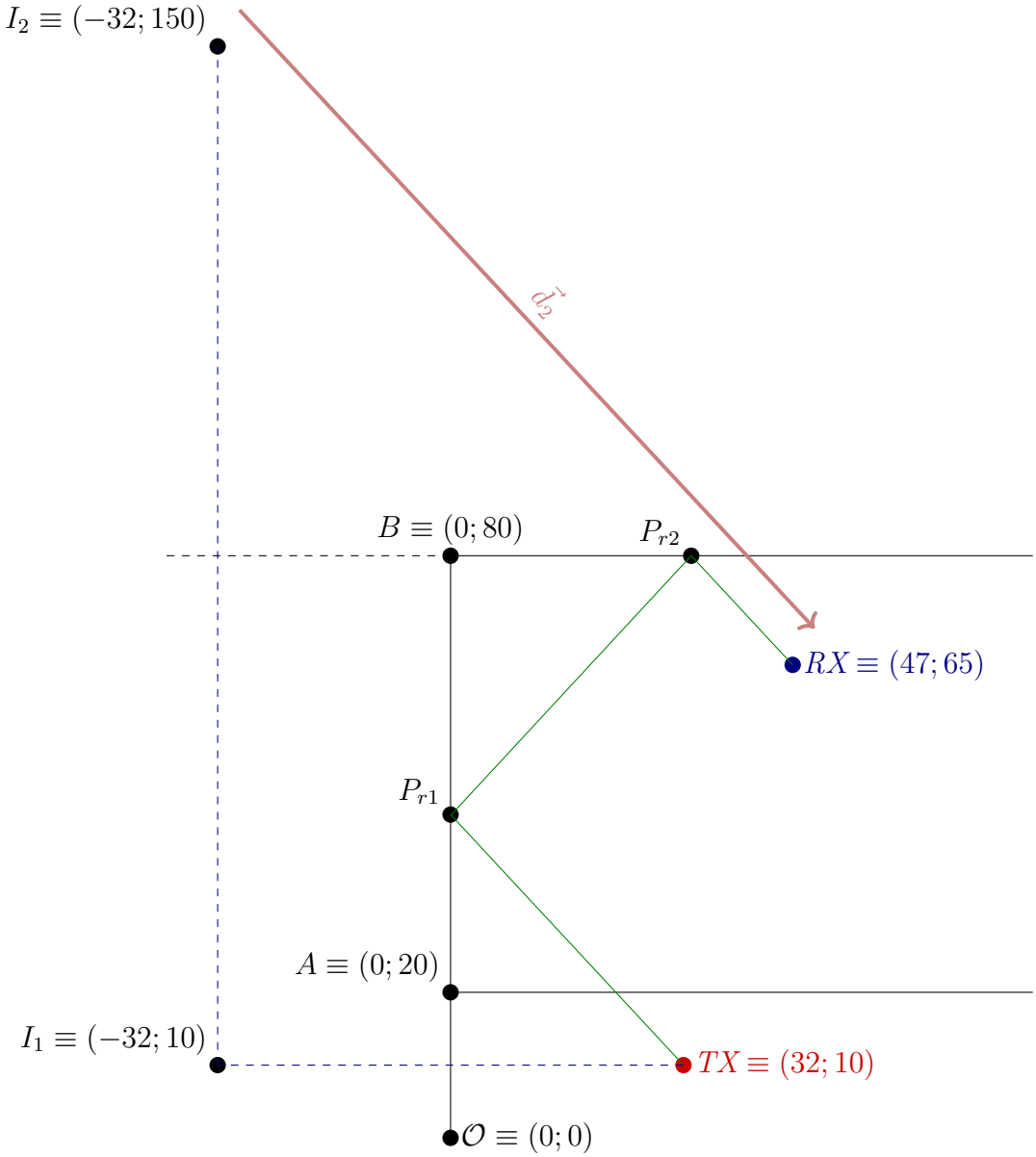

FIGURE 5 – Schéma de la Double Réflexion 1

Avec I_1 étant le symétrique de TX par rapport au mur vertical et I_2 étant le symétrique de I_1 par rapport au mur horizontal supérieur. Afin de déterminer la valeur du champ, il faut déterminer la valeur des différents coefficients de réflexion et de transmission.

Coefficient de Reflexion Γ_{m2}

Γ_{m2} est le coefficient de transmission relatif à l'incidence de l'onde sur le point de réflexion P_{r2} et est donné par :

$$\Gamma_m(\theta_i) = \Gamma_\perp(\theta_i) - (1 - \Gamma_\perp^2(\theta_i)) \frac{\Gamma_\perp(\theta_i) e^{-2\gamma_m s} e^{j\beta 2s\ sin\theta_t sin\theta_i}}{1 - \Gamma_\perp^2(\theta_i) e^{-2\gamma_m s} e^{j\beta 2s\ sin\theta_t sin\theta_i}} \tag{7}$$

Avec :

1. θ_i : L'angle que forme l'onde incidente avec la normale au mur
2. θ_t : L'angle que forme l'onde transmise avec la normale au mur
3. γ_m : La constante de propagation complexe
4. s : La distance parcourue par l'onde dans le mur

De plus :

$$\Gamma_\perp(\theta_i) = \frac{Z_m cos\theta_i - Z_0 cos\theta_t}{Z_m cos\theta_i + Z_0 cos\theta_t} \tag{8}$$

où Z_m est l'impédance du mur et vaut pour tous les murs :

$$\begin{aligned} Z_m &= \sqrt{\frac{\mu_0}{\epsilon - j\sigma/\omega_{TX}}} \\ &\approx (171,57 + 6,65j)\Omega \end{aligned}$$

Les murs ayant tous des propriétés similaires, on a, de plus, pour tous les murs :

$$\begin{aligned} \gamma_m = \alpha_m + j\beta_m &\longrightarrow \begin{cases} \alpha_m = \omega_{TX}\sqrt{\frac{\mu_0\epsilon}{2}}[\sqrt{1 + (\frac{\sigma}{\omega_{TX}\epsilon})^2} - 1]^{1/2} \\ \beta_m = \omega_{TX}\sqrt{\frac{\mu_0\epsilon}{2}}[\sqrt{1 + (\frac{\sigma}{\omega_{TX}\epsilon})^2} + 1]^{1/2} \end{cases} \\ &= 1,55 + 39,9j \end{aligned}$$

Afin de calculer les cosinus et sinus des différents angles, il est nécessaire de définir le vecteur $\vec{d_2}$ étant définit comme $\vec{r_x}$ [1] - $\vec{r_{i2}}$. Il en suit donc que $d_{2x} = 79$ et que d_{2y} = -85.
La normale au mur où se produit la deuxième réflexion est $\vec{n} = (0, 1)$.

1. $\vec{r_x}$ est le vecteur partant de l'origine allant au point RX

Il en suit :

$$
\begin{aligned}
cos\theta_{i2} &= | < \frac{\vec{d_2}}{||d_2||}, \vec{n} > | = \frac{|d_{2y}|}{||d_2||} = 0.7325 \\
sin\theta_{i2} &= \sqrt{1 - cos^2\theta_{i2}} = 0,6808 \\
sin\theta_{t2} &= \sqrt{\frac{1}{\epsilon_r}} sin\theta_{i2} = 0,3107 \\
cos\theta_{t2} &= \sqrt{1 - sin^2\theta_{t2}} = 0,9505 \\
s &= \frac{e}{cos\theta_{t2}} = 0,1578m
\end{aligned}
$$

$$
\hookrightarrow \Gamma_{\perp}(\theta_{2i}) = -0.4803 + 0,014905\,i \tag{9}
$$

Finalement :

$$
\boxed{\Gamma_{m2}(\theta_{2i}) = -0,4188 + 0,2461\,i} \tag{10}
$$

Il reste à déterminer le point de deuxième réflexion P_{r2} :

$$
P_r \equiv \vec{x_0} + t\vec{u} \tag{11}
$$

Avec $\vec{x_0} = (0, 80)$, $\vec{u} = (1, 0)$ il suit :

$$
\begin{aligned}
t_2 &= \frac{d_{2y}(r_{i2x} - x_0) - d_{2x}(r_{i2y} - y_0)}{u_x d_{2y} - u_y d_{2x}} \\
&= 33,06 \\
\implies P_{r2} &= (33,06; 80)
\end{aligned}
$$

Coeffiecient de Reflexion Γ_{m1}

Le calcul de Γ_{m1} se fait de façon analogue à celui vu précédemment. $\vec{d_1}$ est ici définit comme $\vec{p_{r2}} - \vec{r_{i1}} = (65.06, 70)$. Le mur sur lequel se situe la première réflexion est un mur vertical. Ceci implique donc que la normale au mur est $\vec{n} = (1, 0)$.

Variable	Valeur	Variable	Valeur
$cos\theta_{i1}$	0,6808	$\Gamma_{\perp}(\theta_{i1})$	-0.5048 + 0.01444 i
$sin\theta_{i1}$	0,7325	$\vec{x_0}$	(0,0)
$cos\theta_{t1}$	0,9425	$\vec{u}$	(0,1)
$sin\theta_{t1}$	0,3343	t	44,43
s	0,1592 m	P_{r1}	(0 ; 44,43)

Finalement :

$$\boxed{\Gamma_{m1}(\theta_{i1}) = -0.4708 + 0.2518\,i} \tag{12}$$

Coefficient de Transmission T_m

Les points de réflexion permettent de déterminer le chemin de l'onde étudiée. Ce chemin n'implique qu'une seule transmission avec le premier mur horizontal. Le coefficient de transmission est donné par :

$$T_m(\theta_i) = \frac{(1 - \Gamma_{\perp}^2(\theta_i)e^{-\gamma_m s})}{1 - \Gamma_{\perp}^2(\theta_i)e^{-2\gamma_m s}e^{j\beta 2s\ sin\theta_t sin\theta_i}} \tag{13}$$

où le calcul des inconnus se fait en considérant ici $\vec{d} = \vec{p_{r1}} - \vec{r_e}$ [2] = (-32, 34.43). Le mur sur lequel il y a transmission étant horizontal, il en résulte donc que la normale au mur est $\vec{n} = (0, 1)$.

2. $\vec{r_e}$ est le vecteur partant de l'origine et pointant vers le point TX

Les résultats obtenus sont les suivants :

Variable	Valeur	Variable	Valeur
$cos\theta_i$	0,7325	$sin\theta_t$	0,3107
$sin\theta_i$	0,6808	s	0,1592 m
$cos\theta_t$	0,9505	$\Gamma_\perp(\theta_i)$	-0.4803 + 0.014905i

Avec :

$$\boxed{T_m(\theta_i) = 0.62965 + 0.08894\,i} \tag{14}$$

Calcul du Champ

Avant de calculer le champ, il faut déterminer la valeur de d dans l'équation **6**. Elle est donnée par :

$$\begin{aligned} d &= ||\vec{p_{r1}} - \vec{r_e}|| + ||\vec{p_{r2}} - \vec{p_{r1}}|| + ||\vec{r_x} - \vec{p_{r2}}|| \\ &= ||\vec{r_x} - \vec{r_{i2}}|| \\ &= 116,043m \end{aligned}$$

Le champ peut ainsi être calculé :

$$E_{DR1} = (-0,4708+0,2518i)(-0,4188+0,2461i)(0,62965+0,08894i)\ \sqrt{60 * 1,64 * 10^{-3}}\ \frac{e^{-j*18.198..*116,043}}{116,043}$$

$$= \boxed{2.7145\ 10^{-5} - 4.45\ 10^{4}\ i\ \ V/m}$$

4 Résultats

Une fois vérifié, le code a pu être adapté afin de déterminer la puissance reçue en chaque points (zone de 0.5mx0.5m) de l'usine de *Meet-A*.

4.1 Couverture optimale

Afin d'obtenir une couverture satisfaisante, une antenne $TX1$ en (90, 45) a été ajoutée à l'usine. Cette antenne a été placée de façon à couvrir les zones non couvertes avec une seule antenne $TX3$:

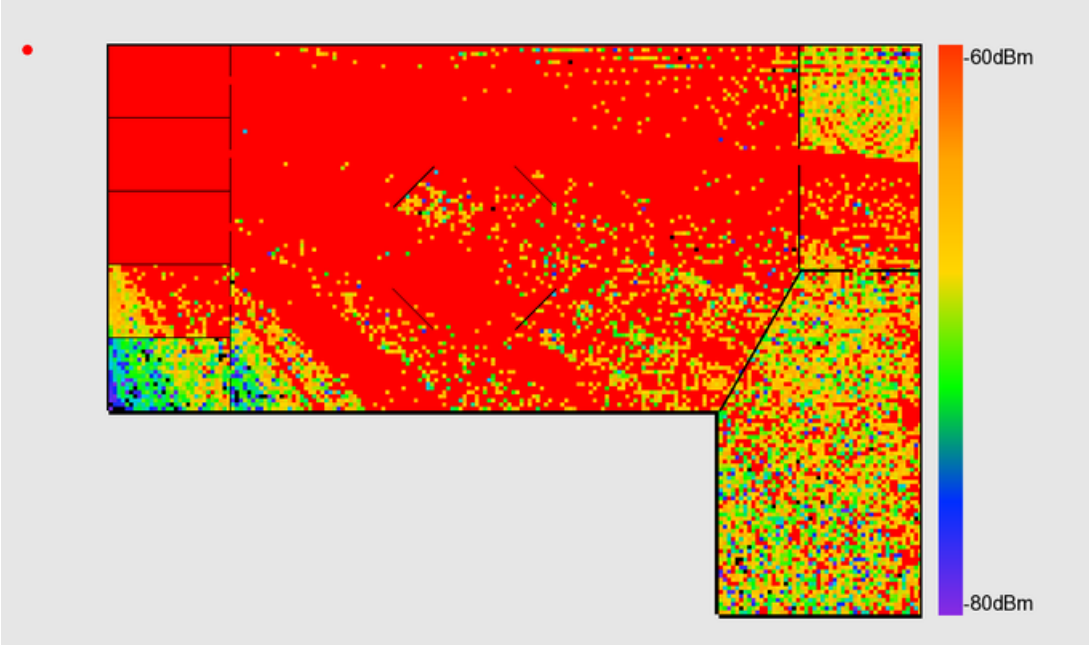

FIGURE 6 – Couverture de l'usine avec émetteur $TX3$ et $TX1$

La couverture n'est pas optimale, il faudrait placer une seconde antenne en bas à gauche de l'usine pour espérer tout couvrir.

Annexe

A Le code

A.1 formules.h

```
1
2  #include <cmath>
3  #include "init_map_tp.h"
4  #include <vector>
5  #include <numeric>
6  #include <iostream>
7  #include <complex>
8  #include <cmath>
9  #include <math.h>
10
11 float distance(point p1, point p2);
12
13 float calculNorme(std::vector<float> unVecteur);
14
15 void vecteurNorme(std::vector<float>& unVecteur);
16
17 std::vector<float> additionerVecteur(std::vector<float> v1, std::vector<float> v2);
18
19 std::vector<float> soustraireVecteur(std::vector<float> v1, std::vector<float> v2);
20
21 std::vector<float> multiplierVecteur(std::vector<float> v, float scalaire);
22 float produitScalaire(std::vector<float> v1, std::vector<float> v2);
23
24 ////////////////////////////////////////////////
25
26 std::complex<float> calcul_gm(float epsilon, float sigma, float omega);
27 std::complex<float> calcul_zm(emetteur Source, mur Mur);
28
29 point calculPtReflexion(point PSource, point Recepteur, mur Mur);
30
31 point trouverSym(point pt, mur Mur);
32
33 void calculTrigoReflex(mur MurReflex, point Recepteur, point pSource, point sym,
34     float& cosOi, float& cosOt, float& sinOi, float& sinOt, float& s);
35
```

```
36 void calculTrigoTrans(mur MurTrans, point Cible, point pSource,
37     float& cosOi, float& cosOt, float& sinOi, float& sinOt, float& s);
38
39 std::complex<float> coeffTrans(mur Mur, point Recepteur, emetteur Source, point pSource,
40     std::complex<float> Zm, std::complex<float> gm);
41
42 std::complex<float> coeffTransTotal(emetteur Source, point Cible, point pSource, float omega, std::vecto
43
44 std::complex<float> coeffReflex(mur Mur, point Recepteur, emetteur Source, point pSource,
45     std::complex<float> Zm, std::complex<float> gm, point sym);
46
47 float calculerPhi(point pSource, point Recepteur);
48
49 float calculGTX(emetteur& Source, point Recepteur);
50
51 bool sensTrigo(point p1, point p2, point p3);
52
53 bool segmentsIntersect(point p1, point p2, point p3, point p4);
54
```

A.2 formules.cpp

```cpp
#include "formule.h"

using namespace std::complex_literals;

float distance(point p1, point p2) {
    return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
}

float produitScalaire(std::vector<float> v1, std::vector<float> v2) {
    float produitScalaire = v1[0] * v2[0] + v1[1] * v2[1];
    return produitScalaire;
}

float calculNorme(std::vector<float> unVecteur) {
    return sqrt(produitScalaire(unVecteur, unVecteur));
}

void vecteurNorme(std::vector<float>& unVecteur) {
    float norme = calculNorme(unVecteur);
    unVecteur = { unVecteur[0] / norme, unVecteur[1] / norme };
}

std::vector<float> additionerVecteur(std::vector<float> v1, std::vector<float> v2) {
    std::vector<float> v3 = { v1[0] + v2[0], v1[1] + v2[1] };
    return v3;
}

std::vector<float> soustraireVecteur(std::vector<float> v1, std::vector<float> v2) {
    std::vector<float> v3 = { v1[0] - v2[0], v1[1] - v2[1] };
    return v3;
}

std::vector<float> multiplierVecteur(std::vector<float> v, float scalaire) {
    std::vector<float> kv = { v[0] * scalaire, v[1] * scalaire };
    return kv;
}
```

```
41
42  ////////////////////////////////////////////////
43
44
45  //calcul de gamma_m = alpha + ibeta
46  std::complex<float> calcul_gm(float epsilon, float sigma, float omega) {
47      float alpha = omega * sqrt(mu0 * epsilon / 2) * sqrt(sqrt(1 + (float)pow(sigma / (omega * epsilon),
48      float beta = omega * sqrt(mu0 * epsilon / 2) * sqrt(sqrt(1 + (float)pow(sigma / (omega * epsilon), 1
49      return alpha + 1if* beta;
50  }
51
52  std::complex<float> calcul_zm(emetteur Source, mur Mur) {
53      float omega = Source.freq * 2 * M_PI;
54      std::complex<float> epsilonTilde = Mur.permittivite - 1if* Mur.conductivite / omega;
55      std::complex<float> Zm = sqrt(mu0 / epsilonTilde);
56      return Zm;
57  }
58
59  // Fonction verifie //
60  point calculPtReflexion(point PSource, point Recepteur, mur Mur) {
61      std::vector<float> n = { -(Mur.p2.y - Mur.p1.y),Mur.p2.x - Mur.p1.x };
62      vecteurNorme(n);
63
64      //std::vector<float> n = { 1, 0 };
65      std::vector<float> vecteurMur = { Mur.p2.x - Mur.p1.x, Mur.p2.y - Mur.p1.y };
66      vecteurNorme(vecteurMur);                  //norme le vecteur directeur du mur
67
68      std::vector<float> re = { PSource.x - Mur.p1.x, PSource.y - Mur.p1.y };
69
70      float ren = produitScalaire(re, n);        //produit scalaire entre re et n
71      std::vector<float> ri = soustraireVecteur(re, multiplierVecteur(n, 2 * ren));
72
73      std::vector<float> rx = { Recepteur.x - Mur.p1.x, Recepteur.y - Mur.p1.y };
74      std::vector<float> d = soustraireVecteur(rx, ri);
75
76      float t = (d[1] * ri[0] - d[0] * ri[1]) / (vecteurMur[0] * d[1] - vecteurMur[1] * d[0]);
77
78      point PtReflexion = { Mur.p1.x + t * vecteurMur[0], Mur.p1.y + t * vecteurMur[1] };
79      //std::cout << "(" << PtReflexion.x << ", " << PtReflexion.y << ")" << std::endl;
80      return PtReflexion;
81  }
82
83
```

```
84  //trouve le symtrique d'un point par rapport  un mr
85  point trouverSym(point pt, mur Mur) {
86      std::vector<float> n = { -(Mur.p2.y - Mur.p1.y),Mur.p2.x - Mur.p1.x };
87      vecteurNorme(n);                    //norme le vecteur normale pour les calculs
88      std::vector<float> vecteurMur = { Mur.p2.x - Mur.p1.x, Mur.p2.y - Mur.p1.y };
89      vecteurNorme(vecteurMur);
90
91      std::vector<float> re = { pt.x - Mur.p1.x, pt.y - Mur.p1.y };
92      float ren = produitScalaire(re, n);       //produit scalaire entre re et n
93      std::vector<float> ri = soustraireVecteur(re, multiplierVecteur(n, 2 * ren));
94      //std::cout << ri[0] << ", " << ri[1] << std::endl;
95      point I = { Mur.p1.x + ri[0], Mur.p1.y + ri[1] };
96      return I;
97  }
98
99  //calcul les paramtres trigonomtriques de transmission dans le mur
100 void calculTrigoReflex(mur MurReflex, point Recepteur, point pSource, point sym, float& cosOi, float& co
101     std::vector<float> d{ Recepteur.x - sym.x, Recepteur.y - sym.y };  //vecteur entre TX et RX
102     std::vector<float> n;                          //vecteurn normal au mr
103     if (MurReflex.p1.x == MurReflex.p2.x) {                  //s'il est vertical
104         n = { 1, 0 };
105
106     }
107     else if (MurReflex.p1.y == MurReflex.p2.y) {              //s'il est horizontal
108         n = { 0, 1 };
109     }
110     else {                                         //s'il est oblique
111         n = { MurReflex.p1.y - MurReflex.p2.y, MurReflex.p1.x - MurReflex.p2.x };
112     }
113     vecteurNorme(n);
114     vecteurNorme(d);
115     cosOi = abs(produitScalaire(d, n));
116     sinOi = sqrt(1 - pow(cosOi, 2));
117     sinOt = sqrt(epsilon0 / MurReflex.permittivite) * sinOi;
118     cosOt = sqrt(1 - pow(sinOt, 2));
119     s = MurReflex.epaisseur / cosOt;
120 }
121
122
123 void calculTrigoTrans(mur MurTrans, point Cible, point pSource, float& cosOi, float& cosOt, float& sinOi
124     std::vector<float> d{ Cible.x - pSource.x, Cible.y - pSource.y };  //vecteur entre TX et RX
125     std::vector<float> n;                          //vecteurn normal au mr
126     if (MurTrans.p1.x == MurTrans.p2.x) {                  //s'il est vertical
```

```
127         n = { 1, 0 };
128
129     }
130     else if (MurTrans.p1.y == MurTrans.p2.y) {              //s'il est horizontal
131         n = { 0, 1 };
132     }
133     else {                                          //s'il est oblique
134         n = { MurTrans.p1.y - MurTrans.p2.y, MurTrans.p1.x - MurTrans.p2.x };
135     }
136     vecteurNorme(n);
137     vecteurNorme(d);
138     cosOi = abs(produitScalaire(d, n));
139     sinOi = sqrt(1 - pow(cosOi, 2));
140     sinOt = sqrt(epsilon0 / MurTrans.permittivite) * sinOi;
141     cosOt = sqrt(1 - pow(sinOt, 2));
142     s = MurTrans.epaisseur / cosOt;
143 }
144
145
146 //calcul le coefficient de transmission d'une polarisation perpendiculaire
147 std::complex<float> coeffTrans(mur Mur, point Recepteur, emetteur Source, point pSource, std::complex<fl
148     float cosOi, cosOt, sinOi, sinOt, s;
149     calculTrigoTrans(Mur, Recepteur, pSource, cosOi, cosOt, sinOi, sinOt, s);
150
151     float omega = 2 * M_PI * Source.freq;
152     float beta = omega / c;
153
154     std::complex<float> Rp = (Zm * cosOi - Z0 * cosOt) / (Zm * cosOi + Z0 * cosOt);
155     std::complex<float> Tm = ((1.F - (std::complex<float>)pow(Rp, 2)) * exp(-gm * s)) / (1.0F - (std::co
156     return Tm;
157 }
158
159 //calcule le coefficient cummule de plusieurs obstacles
160 std::complex<float> coeffTransTotal(emetteur Source, point Cible, point pSource, float omega, std::vecto
161     std::complex<float> Tm = 1;
162     for (const mur& murTrans : MursTrans) {
163         std::complex<float> zmT = calcul_zm(Source, murTrans);
164         std::complex<float> gmT = calcul_gm(murTrans.permittivite, murTrans.conductivite, omega);
165         Tm *= coeffTrans(murTrans, Cible, Source, pSource, zmT, gmT);
166     }
167     return Tm;
168 }
169
```

```
170 //calcul le coefficient de reflexion d'une polarisation perpendiculaire
171 std::complex<float> coeffReflex(mur Mur, point Recepteur, emetteur Source, point pSource, std::complex<
172     float cosOi, cosOt, sinOi, sinOt, s;
173     calculTrigoReflex(Mur, Recepteur, pSource, sym, cosOi, cosOt, sinOi, sinOt, s);
174     float omega = 2 * M_PI * Source.freq;
175     float beta = omega / c;
176 
177     std::complex<float> Rp = (Zm * cosOi - Z0 * cosOt) / (Zm * cosOi + Z0 * cosOt);
178     std::complex<float> Rm = Rp - (1.F - Rp * Rp) *
179         Rp * (exp(-2.F * gm * s) * exp(beta * 1if * 2.F * s * sinOt * sinOi)) /
180         (1.F - Rp * Rp * exp(-2.F * gm * s) * exp(beta * 1if * 2.F * s * sinOt * sinOi));
181     //std::cout << "Coefficient de Rflexion: " << Rm << std::endl;
182     return Rm;
183 }
184 
185 
186 float calculerPhi(point pSource, point Recepteur) {
187     std::vector<float> v{ Recepteur.x - pSource.x, Recepteur.y - pSource.y };  //vecteur entre TX et RX
188     vecteurNorme(v);
189     float phi = acos(v[0]);
190     return phi;
191 }
192 
193 
194 //calcule la puissance de l'metteur dans la direction du rcepteur en ca
195 float calculGTX(emetteur Source, point Recepteur, int number) {
196     if (number == 2) {
197         return 1.7;         //dans le cas du TX2
198     }
199     if (number == 3) {        //dans le cas du TX3
200         float phi3dB = M_PI / 6;
201         float phi = calculerPhi(Source.coordonnees, Recepteur);
202         float GTXdB = 21.5836 - 12 * pow((phi - Source.delta) / phi3dB, 2);
203         float GTX = (float)pow(10, GTXdB / 10);
204         return GTX;
205 
206     }
207 }
208 
209 
210 //verifie si p1 p2 p3 sont enumeres dans le sens trigonomtrique
211 bool sensTrigo(point p1, point p2, point p3) {
212     return (p3.y - p1.y) * (p2.x - p1.x) > (p2.y - p1.y) * (p3.x - p1.x);
```

```
213 }
214
215 float calculPente(point p1, point p2) {
216     return (p2.y - p1.y) / (p2.x - p1.x);
217 }
218
219 //vrifie s'il y a intersection entre segments p1p2 et p3p4: si le sens d'enumeration (trigo ou anti
220 // trigo) de p1, p2, p3 n'est pas celui de p1, p2, p4 ou que celui de p1, p3, p4 est different de celui
221 //pour chaque segment, le deuxime point se trouve de l'autre cote du deuxieme segment donc intersection
222 //d'un segment ne peut pas tre sur l'autre sinon le sens d'enumeration est le mme et il n'y a pas inter
223 bool segmentsIntersect(point p1, point p2, point p3, point p4) {
224     if ((calculPente(p3, p4) == calculPente(p3, p1)) || (calculPente(p3, p4) == calculPente(p3, p2))) {
225         return false;
226     }
227     else {
228         return ((sensTrigo(p1, p2, p3) != sensTrigo(p1, p2, p4)) && (sensTrigo(p1, p3, p4) != sensTrigo
229     }
230 }
231
```

A.3 raytracing.h

```
#pragma once
#include <complex>
#include <cmath>
#include <vector>
#include <numeric>
#include <iostream>
#include <math.h>

#include"init_map_tp.h"
#include "formule.h"
#include "onde.h"
#include "constantes.h"

bool verifReflex(point pSource, point Cible, mur Mur1, mur Mur2, int compteur);

std::vector<mur> trouverMursTrans(point pSource, point Cible, std::array<mur, nbMursMap> MursMap);

std::vector<mur> trouverMursReflex(point pSource, point Cible, std::array<mur, nbMursMap> MursMap);

std::vector<std::vector<mur>> trouverMursDoubleReflex(const point& pSource, const point& Cible, const st


std::complex<float> champTotal(const emetteur& Source, const point& Recepteur);

float puissanceTotale(emetteur& Source, point Recepteur);

std::vector<std::vector<float>> puissanceMap(std::vector<std::vector<point>> mesPointsMap, emetteur& Sou
```

A.4 raytracing.cpp

```
#include <complex>
#include <cmath>
#include <vector>
#include <numeric>
#include <iostream>
#include <math.h>

#include"init_map_tp.h"
#include "formule.h"
#include "onde.h"
#include "constantes.h"
#include "interface_graphique.h"
#include <thread>


//fonction recursive: si appelee avec compteur =1: verifie s'il y a refelxion simple
//si appelee avec compteur = 2: verifie s'il y a reflexion double en s'appelant recursivement
bool verifReflex(point pSource, point Cible, mur Mur1, mur Mur2, int compteur) {
    switch (compteur) {
    case 1: {
        point I = trouverSym(pSource, Mur1);
        return segmentsIntersect(I, Cible, Mur1.p1, Mur1.p2);
        break;
        }
    case 2: {
        bool boolean = false;
        point I1 = trouverSym(pSource, Mur1);
        if (verifReflex(I1, Cible, Mur2, Mur2, compteur-1)) {
            point PtReflexion2 = calculPtReflexion(I1, Cible, Mur2);
            if (verifReflex(pSource, PtReflexion2, Mur1, Mur1, compteur-1)) {
                boolean = true;
            }
        }
        return boolean;
        }
    }
}


//renvoie le vecteur des obstacles entre deux points
```

```
41 std::vector<mur> trouverMursTrans(point pSource, point Cible, std::array<mur, nbMursMap> MursMap) {
42     std::vector<mur> mesMursTrans{};
43     for (const mur murTrans : MursMap) {
44         if (segmentsIntersect(pSource, Cible, murTrans.p1, murTrans.p2)) {
45             mesMursTrans.push_back(murTrans);
46         }
47     }
48     return mesMursTrans;
49 }
50
51 //renvoie le vecteur des murs de reflexion entre source et recepteur
52 std::vector<mur> trouverMursReflex(point pSource, point Cible, std::array<mur, nbMursMap> MursMap) {
53     std::vector<mur> mesMursReflex{};
54     for (const mur& murReflex : MursMap) {
55         if (verifReflex(pSource, Cible, murReflex, murReflex, 1)) {
56             mesMursReflex.push_back(murReflex);
57         }
58     }
59     return mesMursReflex;
60 }
61
62 //renvoie vecteur de vecteurs de 2 murs de doubles reflexion entre source et recepteur
63 std::vector<std::vector<mur>> trouverMursDoubleReflex(const point& pSource, const point& Cible, const s
64     std::vector<std::vector<mur>> mesMursDoubleReflex{};
65     for (const mur& Mur1 : MursMap) {
66         for (const mur& Mur2 : MursMap) {
67             if (&Mur1 != &Mur2) {
68                 if (verifReflex(pSource, Cible, Mur1, Mur2, 2)) {
69                     point I1 = trouverSym(pSource, Mur1);
70                     point I2 = trouverSym(I1, Mur2);
71                     point PtReflexion2 = calculPtReflexion(I1, Cible, Mur2);
72                     point PtReflexion1 = calculPtReflexion(pSource, PtReflexion2, Mur1);
73                     mesMursDoubleReflex.push_back(std::vector<mur>{Mur1, Mur2});
74                 }
75             }
76         }
77     }
78     return mesMursDoubleReflex;
79 }
80
81
82 std::complex<float> champTotal(emetteur& Source, const point& Recepteur) {
83     point pSource = Source.coordonnees;
```

```
84      std::complex<float> ETransTotal = 0;
85      std::complex<float> EReflexTotal = 0;
86      std::complex<float> EDoubleReflexTotal = 0;
87      std::vector<mur> mesMursTrans = trouverMursTrans(pSource, Recepteur, mesMursMap);
88      std::vector<mur> mesMursReflex = trouverMursReflex(pSource, Recepteur, mesMursMap);
89      std::vector<std::vector<mur>> mesMursDoubleReflex = trouverMursDoubleReflex(pSource,
90          Recepteur, mesMursMap);
91      std::vector<std::thread> threads;
92
93      //std::cout << "taille murs double reflex:" << mesMursDoubleReflex[1].size() << std::endl;
94      if (!(mesMursTrans.empty())) {
95          threads.push_back(std::thread([&]() {
96              ETransTotal = champTransTotal(Source, Recepteur, mesMursTrans);
97              //std::cout << Recepteur.x << ", " << Recepteur.y << ": " << norm(ETransTotal) << std::endl
98              }));
99      }
100
101     if (!(mesMursReflex.empty())) {
102         threads.push_back(std::thread([&]() {
103             for (const mur& murReflex : mesMursReflex) {
104                 point PtReflexion = calculPtReflexion(pSource, Recepteur, murReflex);
105                 std::vector<mur> mursTrans1 = trouverMursTrans(pSource, PtReflexion, mesMursMap);
106                 std::vector<mur> mursTrans2 = trouverMursTrans(PtReflexion, Recepteur, mesMursMap);
107                 //std::cout << "mursTrans1 vide:" << mursTrans1.empty() << std::endl;
108                 EReflexTotal += champReflexTotal(Source, Recepteur, PtReflexion, murReflex, mursTrans1,
109             }
110             }));
111     }
112     if (!(mesMursDoubleReflex.empty())) {
113         threads.push_back(std::thread([&]() {
114             for (const std::vector<mur>& deuxMursReflex : mesMursDoubleReflex) {
115                 point I1 = trouverSym(pSource, deuxMursReflex[0]);
116                 point I2 = trouverSym(I1, deuxMursReflex[1]);
117                 //std::cout << "I1: " << I1.x << ", " << I1.y << std::endl;
118                 //std::cout << "I2: " << I2.x << ", " << I2.y << std::endl;
119                 point PtReflexion2 = calculPtReflexion(I1, Recepteur, deuxMursReflex[1]);
120                 point PtReflexion1 = calculPtReflexion(pSource, PtReflexion2, deuxMursReflex[0]);
121
122                 //std::cout << "ptreflex1 " << PtReflexion1.x << ", " << PtReflexion1.y << std::endl;
123                 ///std::cout << "ptreflex2 " << PtReflexion2.x << ", " << PtReflexion2.y << std::endl;
124                 std::vector<mur> mursTrans1 = trouverMursTrans(pSource, PtReflexion1, mesMursMap);
125                 std::vector<mur> mursTrans2 = trouverMursTrans(PtReflexion1, PtReflexion2, mesMursMap);
126                 std::vector<mur> mursTrans3 = trouverMursTrans(PtReflexion2, Recepteur, mesMursMap);
```

```
                int i = 0;
                for (const mur& murTrans : mesMursTrans) {
                    //std::cout << "Mes murs trans " << i << mesMursTrans[i].p1.x << "," << mesMursTran
                    i++;
                }
                EDoubleReflexTotal += champDoubleReflexTotal(Source, Recepteur, I1, I2, PtReflexion1, P
                    mursTrans1, mursTrans2, mursTrans3);
            }
            }));
    }

    //threads.push_back(std::thread([&]() {
      //dessinerInterface(Source.coordonnees, Recepteur, mesMursReflex, mesMursDoubleReflex); }));

    for (auto& thread : threads) {
        thread.join();
    }

    //std::cout << "Etransmission:" << ETransTotal << std::endl;
    //std::cout << "Ereflexion:"    << EReflexTotal << std::endl;
    //std::cout << "Edoublereflex:" << EDoubleReflexTotal << std::endl;

    return ETransTotal + EReflexTotal + EDoubleReflexTotal;
}


//calcule la puissance totale en un point
float puissanceTotale(emetteur& Source, point Recepteur) {
    float he = Source.he;
    std::complex<float> ETotal = champTotal(Source, Recepteur);
    //std::cout << "Etotal: " << ETotal << std::endl;
    float PTotal = 1 / (8 * Source.resistance) * std::norm(ETotal) * (float)pow(he, 2);
    //std::cout << Recepteur.x << Recepteur.y << ": " << PTotal << std::endl;
    return PTotal;
}


//calcule la puissance en chaque point de la map et retourne un vecteur avec une puissance par point
std::vector<std::vector<float>> puissanceMap(std::vector<std::vector<point>> mesPointsMap, emetteur& Sou
    std::vector<std::vector<float>> puissancesPoints = {};
    for (float k = 0; k < mesPointsMap.size(); k++) {
        puissancesPoints.push_back(std::vector<float>{});
        for (float l = 0; l < mesPointsMap[1].size(); l++) {
```

```
170             float P = puissanceTotale(Source, mesPointsMap[k][l]);
171             puissancesPoints[k].push_back(P);
172 
173         }
174     }
175     return puissancesPoints;
176 }
177 
178 
179 
```

A.5 initmaptp.h

```
#pragma once

#include <array>
#include <iostream>
#include <cmath>
#include <vector>
#include "constantes.h"

const float largeurPoint = 0.5;
const int nbPointsY1 = 45/largeurPoint;
const int nbPointsX1 = 100/largeurPoint;
const int nbPointsY2 = 25/largeurPoint;
const int nbPointsX2 = 25/largeurPoint;
const int nbMursMap = 25;

const float puissanceMax = -30;
const float puissanceMin = -80;
const float puissanceFonc = -76.66;


struct point {
    float x{};
    float y{};
};

struct emetteur {
    point coordonnees;
    float freq;
    float resistance;
    float PTX;
    std::vector<float> orientation;
    float delta;
    float he = -c / freq * 1 / M_PI;
};

struct mur {
    point p1;                   // p1.x <= p2.x et p1.y <= p2.y
    point p2;
    float permittivite;
    float conductivite;
```

```
41      float epaisseur;
42  };
43
44  extern emetteur maSourceTX1;
45  extern emetteur maSourceTX2;
46  extern emetteur maSourceTX3;
47
48  //const emetteur  maSource{ point{32, 10},868.3F * (float)pow(10,6), 73.0, 0.1, std::vector<float>(0, 0
49  const point     monRecepteur{ 47, 65 };
50
51
52  /*const std::array<mur, nbMursMap> mesMursMap = {
53      {
54      { {0.0, 0.0}, {0.0, 80.0}, 4.8* epsilon0, 0.018, 0.15},
55      { {0.0, 20.0}, {80, 20.0}, 4.8* epsilon0, 0.018, 0.15},
56      { {0.0, 80.0}, {80.0, 80.0}, 4.8* epsilon0, 0.018, 0.15}
57  }
58  };*/
59
60
61  const std::array<mur, nbMursMap> mesMursMap = {
62      {
63          //mur vertical gauche de brique
64          {{0.0, 0.0}, {0.0, 45.0}, 4.6*epsilon0, 0.02, 0.3 },
65
66          // Je commence par les murs horizontaux à gauche (comprends les 2 longs murs rouge et bleu) (6)
67          { {0.0, 0.0}, {100.0, 0.0}, 4.6 * epsilon0, 0.02, 0.3},  // *
68
69          { {0.0, 9.0}, {15, 9.0}, 2.25 * epsilon0, 0.04, 0.1},  // *
70          { {0.0, 18.0}, {15.0, 18.0}, 2.25 * epsilon0, 0.04, 0.1},  //
71          { {0.0, 27.0}, {15.0, 27.0}, 2.25 * epsilon0, 0.04, 0.1},
72          { {0.0, 36.0}, {15, 36.0}, 2.25 * epsilon0, 0.04, 0.1},
73
74          { {0.0, 45.0}, {75.0, 45.0}, 5 * epsilon0, 0.014, 0.5},
75
76          // Murs verticaux à gauche de longueur 8 (6)
77          { {15.0, 0.0}, {15.0, 4.0}, 2.25 * epsilon0, 0.04, 0.1},
78          { {15.0, 5.0}, {15, 13.0}, 2.25 * epsilon0, 0.04, 0.1},
79          { {15.0, 14.0}, {15.0, 22.0}, 2.25 * epsilon0, 0.04, 0.1},
80          { {15.0, 23.0}, {15.0, 31.0}, 2.24 * epsilon0, 0.04, 0.1},
81          { {15.0, 32.0}, {15, 40.0},  2.25 * epsilon0, 0.04, 0.1},
82          { {15.0, 41.0}, {15.0, 45.0}, 2.25 * epsilon0, 0.04, 0.1},
83
```

```
84          // J'ai finit les murs de gauche go faire les 4 obliques du milieu
85          { {35.0, 20.0}, {40.0, 15.0}, 2.25 * epsilon0, 0.04, 0.1},
86          { {35.0, 30.0}, {40, 35.0}, 2.25 * epsilon0, 0.04, 0.1},
87          { {50.0, 15.0}, {55.0, 20.0}, 2.25 * epsilon0, 0.04, 0.1},
88          { {50.0, 35.0}, {55.0, 30.0}, 2.25 * epsilon0, 0.04, 0.1},
89
90          //5 murs en rouge en haut a droite
91          { {85.0, 0.0}, {85.0, 12.8398}, 4.6 * epsilon0, 0.02, 0.3},
92          {{85.0, 14.8398}, {85, 14.8398 + 12.8398}, 4.6 * epsilon0, 0.02, 0.3},
93          { {100.0, 0.0}, {100.0, 70.0}, 4.6 * epsilon0, 0.02, 0.3},
94          { {85.0, 27.6795}, {91.5, 27.6795}, 4.6 * epsilon0, 0.02, 0.3},
95          { {93.5, 27.6795}, {100, 27.6795}, 4.6 * epsilon0, 0.02, 0.3},
96
97          //Mur oblique de longueur 20 en rouge
98          { {75.0, 45.0}, {85.0, 27.6795}, 4.6 * epsilon0, 0.02, 0.3},
99
100         //3 derniers murs en bas a droite
101         { {75.0, 45.0}, {75.0, 70.0}, 5 * epsilon0, 0.014, 0.5},
102         { {75.0, 70.0}, {100.0, 70.0}, 5 * epsilon0, 0.014, 0.5},
103     }
104
105 };
106
107
108
109 extern std::vector<std::vector<point>> mesPointsMap1;
110 extern std::vector<std::vector<point>> mesPointsMap2;
111
112 void creerPoints1(std::vector<std::vector<point>>& mesPointsMap);
113 void creerPoints2(std::vector<std::vector<point>>& mesPointsMap);
```

A.6 initmaptp.cpp

```cpp
#include "init_map_tp.h"

std::vector<std::vector<point>> mesPointsMap1{};
std::vector<std::vector<point>> mesPointsMap2{};

emetteur maSourceTX1{ point{90, 45}, 26 * (float)pow(10, 9), 73, 0.1, std::vector<float>(0, 0) };
emetteur maSourceTX2{ point{-10, 0.5}, 26 * (float)pow(10, 9), 73, 3.16277 , std::vector<float>(0, 0),
emetteur maSourceTX3{ point{-10, 0.5}, 26 * (float)pow(10, 9), 73, 3.16277 , std::vector<float>(0, 0),


void creerPoints1(std::vector<std::vector<point>>& mesPointsMap) {
    for (float i = 0; i < nbPointsX1; i++) {
        mesPointsMap.push_back(std::vector<point>{});
        for (float j = 0; j < nbPointsY1; j++) {
            mesPointsMap[i].push_back(point{ i * largeurPoint + largeurPoint/2 , j * largeurPoint + larg
        }
    }
}

void creerPoints2(std::vector<std::vector<point>>& mesPointsMap) {
    for (float i = 0; i < nbPointsX2; i++) {
        mesPointsMap.push_back(std::vector<point>{});
        for (float j = 0; j < nbPointsY2; j++) {
            mesPointsMap[i].push_back(point{ i * largeurPoint + largeurPoint/2 + 75, j * largeurPoint +
        }
    }
}


```

A.7 interfacegraphique.h

```
1
2  #pragma once
3
4  #include <iostream>
5  #include <SFML/Graphics.hpp>
6  #include <cmath>
7  #include "init_map_tp.h"
8  #include "onde.h"
9  #include "raytracing.h"
10
11
12 const int largeur = 1500;  const int hauteur = 1000;  const int facteurEchelle = 8.5;  const int epaisse
13 extern std::vector<float> monDegrade;
14 const point finPtXReel = { 105, -5 }; const point finPtYReel = { -5, 80 };
15 const point ptOrigineReel = { -15, -5 };
16
17 const sf::Color couleurMin(255, 255, 0);
18 const sf::Color couleurMax(255, 0, 0);
19
20
21 point conversionLongueurPoint(point p);
22 float distance_interface(point p1, point p2);
23 float trouverAngleMur(mur unMur);
24 float trouverAngleOnde(point p1Onde, point p2Onde);
25
26
27 sf::RectangleShape dessinerContourMap();
28 sf::RectangleShape dessinerMur(mur Mur);
29 sf::RectangleShape* stockerMursDessin(mur arrayDeMurs[nbMursMap]);
30
31 sf::CircleShape dessinerCerle(point unPoint, bool Source);
32
33
34
35 //std::vector<point> trouverCheminDirect(point pSource, point Recepteur, std::vector<mur> MursTotaux);
36 std::vector<point> trouverCheminReflex(point Source, point Recepteur, std::vector<mur> MursReflex);
37 std::vector<point> trouverCheminDoubleReflex(point pSource, point Recepteur, std::vector<std::vector<mu
38
39 sf::RectangleShape creerOnde(point p1Onde, point p2Onde, int typeOnde);
40 std::vector<sf::RectangleShape> dessinerOnde(point pSource, point Recepteur, std::vector<mur> MursRefle
```

```
41      std::vector<std::vector<mur>> MursDoubleReflex);
42
43  sf::RectangleShape dessinerCarre(point unPoint, float puissance);
44  std::vector<sf::RectangleShape> dessinerDegrade();
45  void dessinerInterface(point pSource, point Recepteur, std::vector<mur> MursReflex,
46      std::vector<std::vector<mur>> MursDoubleReflex);
47  void dessinerPuissances(point pSource, std::vector<std::vector<float>> puissancesPoints1,
48      std::vector<std::vector<float>> puissancesPoints2);
49
```

A.8 interfacegraphique.cpp

```cpp
//Created by David Moli on 19 / 04 / 2023.


#include "interface_graphique.h"
#include <thread>
#include <algorithm>
#include <string>


float largeurPointInterface = largeurPoint * facteurEchelle;

std::vector<float> monDegrade = {};

point conversionLongueurPoint(point p) {          // Je convertis la longueur en mtre en longueur de l'
    float x = (p.x + 30) * facteurEchelle;  // + 10 sinon c tro a gauche
    float y = (p.y + 25) * facteurEchelle;
    point pointReturn = { x, y };
    return pointReturn;

};

float distance_interface(point p1, point p2) {
    return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
}

float trouverAngleMur(mur unMur) {
    return (atan2(unMur.p2.y - unMur.p1.y, unMur.p2.x - unMur.p1.x) * 180) / M_PI;   // Je calcule l'an
}

float trouverAngleOnde(point p1Onde, point p2Onde) {
    return (atan2(p2Onde.y - p1Onde.y, p2Onde.x - p1Onde.x) * 180) / M_PI;   // Je calcule l'angle que
}

sf::RectangleShape dessinerContourMap() {        // Je fais le contour en gris qui dlimite la map

    sf::RectangleShape contour(sf::Vector2f(largeur - 200, hauteur - 200));
    contour.setFillColor(sf::Color(230, 230, 230));
    contour.setPosition(100, 100);
    return contour;
};
```

```
41
42
43  sf::RectangleShape dessinerMur(mur Mur) {    // Sert  dfinir le rectangle  dessiner en fonction du mur
44
45      point p1 = conversionLongueurPoint(Mur.p1);
46      point p2 = conversionLongueurPoint(Mur.p2);
47      float epaisseur = Mur.epaisseur * facteurEchelle;
48      float longueurMur = distance_interface(p1, p2);
49      sf::RectangleShape murADessiner(sf::Vector2f(longueurMur, epaisseur));
50      murADessiner.setPosition(p1.x, p1.y);
51      murADessiner.setFillColor(sf::Color(0, 0, 0));
52      murADessiner.setRotation(trouverAngleMur(Mur));
53      return murADessiner;
54  };
55
56  sf::RectangleShape* stockerMursDessin(std::array<mur, nbMursMap> arrayDeMurs) {   //Ici a me renvoit un
57
58      static sf::RectangleShape mursAPlot[nbMursMap];
59      for (int i = 0; i < nbMursMap; i++) {
60          mursAPlot[i] = dessinerMur(arrayDeMurs[i]);
61      }
62      return mursAPlot;
63  }
64  // Créer emetteur récepteur
65  sf::CircleShape dessinerCerle(point unPoint, bool Source) {
66      point pointAPlot = conversionLongueurPoint(unPoint);
67      sf::CircleShape cercleAPlot = sf::CircleShape(rayonCercle);
68      cercleAPlot.setPosition(pointAPlot.x - 5, pointAPlot.y - 3);
69      if (Source) {
70          cercleAPlot.setFillColor(sf::Color(255, 0, 0));
71      }
72      else {
73          cercleAPlot.setFillColor(sf::Color(0, 0, 255));
74      }
75      return cercleAPlot;
76  }
77  // Je cre mes chemins a partir de rectangles
78
79
80  // Ici permet de crer une onde partant d'un point  l'autre
81  sf::RectangleShape creerOnde(point p1Onde, point p2Onde, int typeOnde) {
82      point p1 = conversionLongueurPoint(p1Onde);
83      point p2 = conversionLongueurPoint(p2Onde);
```

```
84      float longueurOnde = distance_interface(p1, p2);
85
86      sf::RectangleShape ondeADessioner(sf::Vector2f(longueurOnde, epaisseurOnde));
87
88      ondeADessioner.setPosition(p1.x, p1.y);
89      ondeADessioner.setRotation(trouverAngleOnde(p1, p2));
90
91      switch (typeOnde) {
92
93      case 2:
94          ondeADessioner.setFillColor(sf::Color(255, 0, 0));      // Onde rouge si Reflexion
95          break;
96
97      case 3:
98          ondeADessioner.setFillColor(sf::Color(0, 255, 0));      // Onde verte si double Reflex
99          break;
100
101     default:
102         ondeADessioner.setFillColor(sf::Color(0, 0, 255));      // Onde bleue si Trans
103         break;
104     }
105
106     return ondeADessioner;
107
108 }
109
110
111
112 std::vector<point> trouverCheminReflex(point Source, point Recepteur, std::vector<mur> MursReflex)
113 {
114     std::vector<point> cheminReturn = {};
115     for (int i = 0; i < MursReflex.size(); i++) {
116         point PtReflexion = calculPtReflexion(Source, Recepteur, MursReflex[i]);
117         cheminReturn.push_back(PtReflexion);
118     }
119     return cheminReturn;
120 }
121
122 std::vector<point> trouverCheminDoubleReflex(point pSource, point Recepteur, std::vector<std::vector<mu
123     std::vector<point> cheminReturn = {};
124
125     for (int i = 0; i < MursDoubleReflex.size(); i++) {
126         const mur& Mur1 = MursDoubleReflex[i][0];
```

```
127         const mur& Mur2 = MursDoubleReflex[i][1];
128         point I1 = trouverSym(pSource, Mur1);
129         point PtReflexion2 = calculPtReflexion(I1, Recepteur, Mur2);
130         point PtReflexion1 = calculPtReflexion(pSource, PtReflexion2, Mur1);
131         cheminReturn.push_back(PtReflexion1);
132         cheminReturn.push_back(PtReflexion2);
133     }
134     return cheminReturn;
135 }
136 
137 
138 std::vector<sf::RectangleShape> dessinerOnde(point pSource, point Recepteur, std::vector<mur> MursReflex
139     std::vector<std::vector<mur>> MursDoubleReflex) {
140     int nOnde = 1;
141     std::vector<point> CheminReflexSimple = trouverCheminReflex(pSource, Recepteur, MursReflex);
142     std::vector<point> CheminReflexDouble = trouverCheminDoubleReflex(pSource, Recepteur, MursDoubleRefl
143     std::vector<sf::RectangleShape> mesOndes;
144 
145     sf::RectangleShape OndeDirect = creerOnde(pSource, Recepteur, 1);
146     mesOndes.push_back(OndeDirect);
147 
148     std::vector<std::thread> threads;
149 
150     if (!(CheminReflexSimple.empty())) {
151         for (int i = 0; i < CheminReflexSimple.size(); i++) {
152 
153             mesOndes.push_back(creerOnde(pSource, CheminReflexSimple[i], 2));
154             mesOndes.push_back(creerOnde(CheminReflexSimple[i], Recepteur, 2));
155         }
156     }
157 
158 
159     if (!(CheminReflexDouble.empty())) {
160 
161         for (int i = 0; i < CheminReflexDouble.size(); i++) {
162 
163             if (i == 0) {
164                 mesOndes.push_back(creerOnde(pSource, CheminReflexDouble[i], 3));
165             }
166             else if (i % 2 == 0) {
167                 mesOndes.push_back(creerOnde(pSource, CheminReflexDouble[i], 3));
168             }
169             else {
```

```
170                 mesOndes.push_back(creerOnde(CheminReflexDouble[i - 1], CheminReflexDouble[i], 3));
171                 mesOndes.push_back(creerOnde(CheminReflexDouble[i], Recepteur, 3));
172             }
173         }
174     }
175 
176 
177     return mesOndes;
178 }
179 
180 
181 
182 void dessinerInterface(point pSource, point Recepteur, std::vector<mur> MursReflex,
183     std::vector<std::vector<mur>> MursDoubleReflex)
184 {
185     // create the window
186     sf::RenderWindow window(sf::VideoMode(largeur, hauteur), "RayTransing");
187 
188 
189     //sf::View view = window.getDefaultView();
190     //view.setSize(largeur, -hauteur);
191     //window.setView(view);
192     // run the program as long as the window is open
193     while (window.isOpen())
194     {
195         // check all the window's events that were triggered since the last iteration of the loop
196         sf::Event event;
197         while (window.pollEvent(event))
198         {
199             // "close requested" event: we close the window
200             if (event.type == sf::Event::Closed)
201                 window.close();
202         }
203 
204         // clear the window with White color
205         window.clear(sf::Color::White);
206 
207         // draw everything here...
208         window.draw(dessinerContourMap());
209         // Dessine les murs
210         for (int i = 0; i < nbMursMap; i++) {
211             window.draw(stockerMursDessin(mesMursMap)[i]);  // Je plot tous mes murs
212 
```

```
213         }
214
215         window.draw(dessinerCerle(pSource, true));
216         window.draw(dessinerCerle(Recepteur, false));
217
218         // Dessine les ondes
219         std::vector<sf::RectangleShape> listeOndes = dessinerOnde(pSource, Recepteur, MursReflex, MursD
220
221         for (int j = 0; j < listeOndes.size(); j++) {
222             window.draw(listeOndes[j]);
223         }
224
225         // end the current frame
226         window.display();
227     }
228 }
229
230
231 sf::RectangleShape dessinerCarre(point unPoint, float puissance) {
232     point pointAPlot = conversionLongueurPoint(unPoint);
233     sf::RectangleShape rectAPlot(sf::Vector2f(largeurPointInterface, largeurPointInterface));
234     rectAPlot.setPosition(pointAPlot.x-largeurPointInterface/2, pointAPlot.y-largeurPointInterface/2 );
235     float puissancedBm = 10 * log10(puissance*pow(10, 3));
236     if (puissancedBm < puissanceFonc) {
237         std::cout << unPoint.x << ", " << unPoint.y << ": " << puissancedBm << std::endl;
238     }
239     float t = (float)(puissancedBm + 80) / 20.0f;
240     float r, g, b;
241
242     if (puissancedBm > puissanceMax) {
243         std::cout << unPoint.x << ", " << unPoint.y << ": " << puissancedBm << std::endl;
244         r = 255;
245         g = 255;
246         b = 255;
247     }
248     if (t < 0) {
249         r = 0;
250         g = 0;
251         b = 0;
252     }
253     else if (t < 0.2) {
254         // Mauve à Bleu
255         r = 138 + t * (0 - 138) / 0.2;
```

```
256             g = 43 + t * (47 - 43) / 0.2;
257             b = 226 + t * (255 - 226) / 0.2;
258         }
259         else if (t < 0.4) {
260             // Bleu à Vert
261             r = 0 + (t - 0.2) * (0 - 0) / 0.2;
262             g = 191 + (t - 0.2) * (255 - 191) / 0.2;
263             b = 255 + (t - 0.2) * (0 - 255) / 0.2;
264         }
265         else if (t < 0.6) {
266             // Vert à Jaune
267             r = 0 + (t - 0.4) * (255 - 0) / 0.2;
268             g = 255 + (t - 0.4) * (215 - 255) / 0.2;
269             b = 0 + (t - 0.4) * (0 - 0) / 0.2;
270         }
271         else if (t < 0.8) {
272             // Jaune à Rouge
273             r = 255 + (t - 0.6) * (255 - 255) / 0.2;
274             g = 215 + (t - 0.6) * (165 - 215) / 0.2;
275             b = 0 + (t - 0.6) * (0 - 0) / 0.2;
276         }
277         else {
278             // Rouge
279             r = 255;
280             g = 0;
281             b = 0;
282         }
283         rectAPlot.setFillColor(sf::Color((int)r, (int)g, (int)b));
284 
285         return rectAPlot;
286     }
287 
288 
289 
290     std::vector<sf::RectangleShape> dessinerDegrade() {
291         point p1 = conversionLongueurPoint({ 105, 0 });
292         point p2 = conversionLongueurPoint({ 105, 70 });
293         float epaisseur = 3 * facteurEchelle;
294         float longueur = distance_interface(p1, p2);
295 
296         std::vector<sf::RectangleShape> vecteurClipRect{};
297         float bas = 0;
298         float r = 0, g = 0, b = 0;
```

```
299     for (float t = 1; t >=0; t -=0.001) {
300         if (t < 0.2) {
301             // Mauve à Bleu
302             r = 138 + t * (0 - 138) / 0.2;
303             g = 43 + t * (47 - 43) / 0.2;
304             b = 226 + t * (255 - 226) / 0.2;
305         }
306         else if (t < 0.4) {
307             // Bleu à Vert
308             r = 0 + (t - 0.2) * (0 - 0) / 0.2;
309             g = 47 + (t - 0.2) * (255 - 47) / 0.2;
310             b = 255 + (t - 0.2) * (0 - 255) / 0.2;
311         }
312         else if (t < 0.6) {
313             // Vert à Jaune
314             r = 0 + (t - 0.4) * (255 - 0) / 0.2;
315             g = 255 + (t - 0.4) * (215 - 255) / 0.2;
316             b = 0 + (t - 0.4) * (0 - 0) / 0.2;
317         }
318         else if (t < 0.8) {
319             // Jaune à Rouge
320             r = 255 + (t - 0.6) * (255 - 255) / 0.2;
321             g = 215 + (t - 0.6) * (165 - 215) / 0.2;
322             b = 0 + (t - 0.6) * (0 - 0) / 0.2;
323         }
324         else {
325             // Rouge
326             r = 255 + (t - 0.8) * (255 - 255) / 0.2;
327             g = 165 + (t - 0.8) * (50-165) / 0.2;
328             b = 0;
329         }
330         sf::RectangleShape clipRect(sf::Vector2f(longueur / 1000, epaisseur));
331         clipRect.setRotation(90);
332         bas+=0.001;
333         clipRect.setPosition(p1.x, p1.y + bas*longueur);
334         clipRect.setFillColor(sf::Color((int)r, (int)g, (int)b));
335         vecteurClipRect.push_back(clipRect);
336     }
337     return vecteurClipRect;
338 }
339
340
341
```

```
342 void dessinerPuissances(point pSource, std::vector<std::vector<float>> puissancesPoints1,
343     std::vector<std::vector<float>> puissancesPoints2) {
344 
345 
346     // create the window
347     sf::RenderWindow window(sf::VideoMode(largeur, hauteur), "RayTransing");
348 
349     // run the program as long as the window is open
350     while (window.isOpen())
351     {
352         // check all the window's events that were triggered since the last iteration of the loop
353         sf::Event event;
354         while (window.pollEvent(event))
355         {
356             // "close requested" event: we close the window
357             if (event.type == sf::Event::Closed)
358                 window.close();
359         }
360 
361         // clear the window with White color
362         window.clear(sf::Color::White);
363 
364         // draw everything here...
365         window.draw(dessinerContourMap());
366 
367         sf::Font font;
368         if (!font.loadFromFile("arial.ttf"))
369         {
370             // Erreur lors du chargement de la police
371             std::cout << "YA PROBLM" << std::endl;
372         }
373 
374         sf::Text maxdB;
375         maxdB.setString("-60dBm");
376         maxdB.setFont(font);
377         maxdB.setCharacterSize(20);
378         maxdB.setFillColor(sf::Color(0, 0, 0));
379         point pointText = conversionLongueurPoint({ 105, (float)0});
380         maxdB.setPosition(pointText.x, pointText.y);
381 
382         sf::Text mindB;
383         mindB.setString("-80dBm");
384         mindB.setFont(font);
```

```
385         mindB.setCharacterSize(20);
386         mindB.setFillColor(sf::Color(0, 0, 0));
387         point pointText2 = conversionLongueurPoint({ 105, (float)67 });
388         mindB.setPosition(pointText2.x, pointText2.y);
389 
390         window.draw(mindB);
391         window.draw(maxdB);
392 
393         for (float i = 0; i < puissancesPoints1.size(); i++) {
394             for (float j = 0; j < puissancesPoints1[1].size(); j++) {
395                 window.draw(dessinerCarre(mesPointsMap1[i][j], puissancesPoints1[i][j]));
396             }
397         }
398         for (float m = 0; m < puissancesPoints2.size(); m++) {
399             for (float n = 0; n < puissancesPoints2[1].size(); n++) {
400                 window.draw(dessinerCarre(mesPointsMap2[m][n], puissancesPoints2[m][n]));
401             }
402         }
403         // Dessine les murs
404         for (int i = 0; i < nbMursMap; i++) {
405             window.draw(stockerMursDessin(mesMursMap)[i]);  // Je plot tous mes murs
406 
407         }
408 
409         window.draw(dessinerCerle(pSource, true));
410 
411         std::vector<sf::RectangleShape> vecteurClipRect = dessinerDegrade();
412         for (int x = 0; x < vecteurClipRect.size(); x++) {
413             window.draw(vecteurClipRect[x]);
414         }
415 
416         // end the current frame
417         window.display();
418     }
419 }
```

A.9 onde.h

```
#pragma once

#include <cmath>
#include "init_map_tp.h"
#include <vector>
#include <numeric>
#include <iostream>
#include <complex>
#include <cmath>
#include <math.h>



struct onde {
    point ptDepart;
    point ptArrivee;
};

std::complex<float> champIncident(emetteur& Source, point Recepteur, float d);

float puissanceDirect(emetteur& Source, point Recepteur);

std::complex<float> champTransTotal(emetteur& Source, const point& Recepteur, const std::vector<mur>& Mu

std::complex<float> champReflex(emetteur& Source, const point& Recepteur, const mur& MurReflexion);

std::complex<float> champReflexTotal(emetteur& Source, const point& Recepteur, const point& PtReflexion
    const mur& MurReflex,const std::vector<mur>& MursTrans1, const std::vector<mur>& MursTrans2);

std::complex<float> champDoubleReflex(emetteur& Source, const point& Recepteur, const mur& MurReflex1,
    const mur& MurReflex2, const point& I1, const point& I2, const point& PtReflexion2);

std::complex<float> champDoubleReflexTotal(
    emetteur& Source, const point& Recepteur,
    const point& I1, const point& I2, const point& PtReflexion1, const point& PtReflexion2,
    const std::vector<mur>& DeuxMursReflex,
    const std::vector<mur>& MursTrans1, const std::vector<mur>& MursTrans2, const std::vector<mur>& Murs
```

A.10 onde.cpp

```cpp
#include "onde.h"
#include "constantes.h"
#include <algorithm>
#include "formule.h"


using namespace std::complex_literals;

//calcule le champs reçu en chemin direct sans obstacle
std::complex<float> champIncident(emetteur& Source, point Recepteur, float d) {
    float GTX = calculGTX(Source, Recepteur);
    float PTX = Source.PTX;
    float omega = Source.freq * 2 * M_PI;
    float beta = omega / c;
    std::complex<float> expo= std::exp(- beta * d * 1if);
    std::complex<float> Ei = sqrt(60.F * GTX * PTX) * expo / d;  //attention changer PTX*GTX
    return Ei;
}

float puissanceDirect(emetteur& Source, point Recepteur) {
    float he = Source.he;
    float d = distance(Source.coordonnees, Recepteur);
    float Ecarre = std::norm(champIncident(Source, Recepteur, d));
    float Puissance = (1 / (8 * Source.resistance)) * Ecarre * pow(he, 2);
    return Puissance;
}

//calcule le champ de tranmission en prenant tous les obstacle en compte
std::complex<float> champTransTotal(emetteur& Source, const point& Recepteur, const std::vector<mur>& Mu

    const point pSource = Source.coordonnees;
    const float d = distance(pSource, Recepteur);
    const float omega = Source.freq * 2 * M_PI;
    std::complex<float> Tm = 1;
    for (const mur& murTrans : MursTrans) {
        std::complex<float> zmT = calcul_zm(Source, murTrans);
        std::complex<float> gmT = calcul_gm(murTrans.permittivite, murTrans.conductivite, omega);
        Tm *= coeffTrans(murTrans, Recepteur, Source, pSource , zmT, gmT);
        //std::cout << "coeff de transmission: " << Tm << std::endl;
    }
```

```
41      std::complex<float> Etot = Tm * champIncident(Source, Recepteur, d);
42      return Etot;
43  }
44
45  //calcule le champs de la réflexion sans obstacle
46  std::complex<float> champReflex(emetteur& Source, const point& Recepteur, const mur& MurReflexion) {
47
48      const point pSource = Source.coordonnees;
49      const point I1 = trouverSym(pSource, MurReflexion);
50      const float dr = distance(I1, Recepteur);
51      const float epsilon = MurReflexion.permittivite;
52      const float sigma = MurReflexion.conductivite;
53      const float omega = Source.freq * 2 * M_PI;
54
55      std::complex<float> gm = calcul_gm(epsilon, sigma, omega);
56      std::complex<float> Zm = calcul_zm(Source, MurReflexion);
57      std::complex<float> Rm = coeffReflex(MurReflexion, Recepteur, Source, pSource, Zm, gm, I1);
58
59      std::complex<float> EReflex = Rm * champIncident(Source, Recepteur, dr);
60      return EReflex;
61  }
62
63  //calcule le champs de reflexion en prenant en compte les obstacles
64  std::complex<float> champReflexTotal(emetteur& Source, const point& Recepteur, const point& PtReflexion
65      const mur& MurReflex, const std::vector<mur>& MursTrans1, const std::vector<mur>& MursTrans2){
66
67      float omega = 2 * M_PI * Source.freq;
68      float resistanceSource = Source.resistance;
69      float he = Source.he;
70      point pSource = Source.coordonnees;
71
72      std::complex<float> EReflex = champReflex(Source, Recepteur, MurReflex);
73      std::complex<float> Tm1 = coeffTransTotal(Source, PtReflexion, pSource, omega, MursTrans1);
74      std::complex<float> Tm2 = coeffTransTotal(Source, Recepteur, PtReflexion, omega, MursTrans2);
75
76      std::complex<float> Etot = EReflex * Tm1 * Tm2;
77      return Etot;
78
79  }
80
81
82  //calcule le champ de double reflexion sans obstacle
83  std::complex<float> champDoubleReflex(emetteur& Source, const point& Recepteur, const mur& MurReflex1,
```

```
84      const mur& MurReflex2, const point& I1, const point& I2, const point& PtReflexion2) {
85
86      const point pSource = Source.coordonnees;
87      const float epsilon1 = MurReflex1.permittivite;
88      const float epsilon2 = MurReflex2.permittivite;
89      const float sigma1 = MurReflex1.conductivite;
90      const float sigma2 = MurReflex2.conductivite;
91      const float omega = Source.freq * 2 * M_PI;
92      const float drr = distance(I2, Recepteur);
93
94      //std::cout << "    " << omega << std::endl;
95
96      std::complex<float> gm1 = calcul_gm(epsilon1, sigma1, omega);
97      std::complex<float> Zm1 = calcul_zm(Source, MurReflex1);
98      std::complex<float> Rm1 = coeffReflex(MurReflex1, PtReflexion2, Source, pSource, Zm1, gm1, I1);  //
99
100     std::complex<float> gm2 = calcul_gm(epsilon2, sigma2, omega);
101     std::complex<float> Zm2 = calcul_zm(Source, MurReflex2);
102     std::complex<float> Rm2 = coeffReflex(MurReflex2, Recepteur, Source, I1, Zm2, gm2, I2);          //
103
104     std::complex<float> EDoubleReflex = Rm1 * Rm2 * champIncident(Source, Recepteur, drr);
105     return EDoubleReflex;
106
107 }
108
109
110
111 /// <summary>
112 ///
113 /// </summary>
114 /// <param name="Source"></param>
115 /// <param name="Recepteur"></param>
116 /// <param name="I1"></param>
117 /// <param name="I2"></param>
118 /// <param name="PtReflexion1"></param>
119 /// <param name="PtReflexion2"></param>
120 /// <param name="DeuxMursReflex"></param>
121 /// <param name="MursTrans1"></param>
122 /// <param name="MursTrans2"></param>
123 /// <param name="MursTrans3"></param>
124 /// <returns></returns>
125 std::complex<float> champDoubleReflexTotal(
126     emetteur& Source, const point& Recepteur,
```

```
127     const point& I1, const point& I2, const point& PtReflexion1, const point& PtReflexion2,
128     const std::vector<mur>& DeuxMursReflex,
129     const std::vector<mur>& MursTrans1, const std::vector<mur>& MursTrans2, const std::vector<mur>& Murs
130
131     const float omega = 2 * M_PI * Source.freq;
132     const point pSource = Source.coordonnees;
133     const std::complex<float> EDoubleReflex = champDoubleReflex(Source, Recepteur, DeuxMursReflex[0], D
134
135     std::complex<float> Tm1 = coeffTransTotal(Source, PtReflexion1, pSource, omega, MursTrans1);
136     //std::cout << "coeff de transmission1: " << Tm1 << std::endl;
137
138     std::complex<float> Tm2 = coeffTransTotal(Source, PtReflexion2, PtReflexion1, omega, MursTrans2);
139     //std::cout << "coeff de transmission2: " << Tm2 << std::endl;
140
141     std::complex<float> Tm3 = coeffTransTotal(Source, Recepteur, PtReflexion2, omega, MursTrans3);
142     //std::cout << "coeff de transmission3: " << Tm3 << std::endl;
143
144     //std::cout << std::endl << EDoubleReflex * Tm1 * Tm2 * Tm3 << std::endl;
145     return EDoubleReflex * Tm1 * Tm2 * Tm3;
146 }
```

A.11 main.cpp

```
#include "raytracing.h"
#include "interface_graphique.h"


int main(void) {
    //crees les points de la map
    creerPoints1(mesPointsMap1);
    creerPoints2(mesPointsMap2);


    std::vector<std::vector<float>> puissancesPoints1 = puissanceMap(mesPointsMap1, maSourceTX3);
    std::vector<std::vector<float>> puissancesPoints2 = puissanceMap(mesPointsMap2, maSourceTX3);
    std::vector<std::vector<float>> puissancesPoints3 = puissanceMap(mesPointsMap1, maSourceTX1);
    std::vector<std::vector<float>> puissancesPoints4 = puissanceMap(mesPointsMap2, maSourceTX1);

    for (int i = 0; i<puissancesPoints1.size(); i++) {
        for (int j = 0; j<puissancesPoints1[0].size(); j++) {
            puissancesPoints1[i][j] = std::max(puissancesPoints3[i][j], puissancesPoints1[i][j]);
        }
    }
    for (int i = 0; i < puissancesPoints2.size(); i++) {
        for (int j = 0; j < puissancesPoints2[0].size(); j++) {
            puissancesPoints2[i][j] = std::max(puissancesPoints4[i][j], puissancesPoints2[i][j]);
        }
    }


    //std::transform(puissancesPoints1.begin(), puissancesPoints1.end(), puissancesPoints3.begin(),
       // puissancesPoints1.begin(), std::plus<double>());
    //std::transform(puissancesPoints2.begin(), puissancesPoints2.end(), puissancesPoints4.begin(),
        //puissancesPoints2.begin(), std::plus<double>());
    dessinerPuissances(maSourceTX2.coordonnees, puissancesPoints1, puissancesPoints2);

    return 0;
}


```