



Département des Technologies de
l'information et de la communication (TIC)
Informatique et systèmes de communication
Informatique logicielle

Travail de Bachelor (Rapport Intermédiaire)

Editeur d'abaque de Smith

Développement d'un outil moderne et multiplateforme

Étudiant	Edwin Häffner
Enseignant responsable	Prof. Bertrand Hochet
Année académique	2024-25

Yverdon-les-Bains, le 08.01.2026

Département des Technologies de l'information et de la communication (TIC)
Informatique et systèmes de communication
Informatique logicielle
Étudiant : Edwin Häffner
Enseignant responsable : Prof. Bertrand Hochet

Travail de Bachelor 2024-25
Editeur d'abaque de Smith

Résumé publiable

Ce travail de Bachelor a pour but de développer un logiciel pour faciliter l'utilisation de l'abaque de Smith, qui permet d'effectuer graphiquement une adaptation d'impédance. L'objectif est de proposer un outil plus moderne et surtout multi-plateforme qui permet à l'utilisateur de comprendre ce qu'il fait, à l'instar des outils de simulation automatique.

Au terme de ce travail, l'application est pleinement fonctionnelle. Les fonctionnalités principales demandées dans le cahier des charges ont pu être mises en place. L'application permet d'effectuer manuellement et intuitivement le mécanisme d'adaptation d'impédance en plaçant des composants passifs (R, L, C, lignes de transmission, stubs) en série ou en parallèle. L'utilisateur peut visualiser en temps réel les trajectoires sur l'abaque ainsi que le schéma électrique du circuit, accompagnés des paramètres calculés (VSWR, Return Loss, coefficient de réflexion,...).

L'utilisateur peut importer et exporter des fichiers S1P, ce qui permet de visualiser et d'adapter le comportement du circuit selon différentes fréquences mesurées ou simulées. L'outil permet de mettre en évidence jusqu'à trois bandes de fréquences sur le fichier S1P et permet la visualisation en temps réel de chaque modification du circuit d'adaptation sur l'entièreté des points affichés sur l'abaque. La fonction de balayage fréquentiel (sweep) permet d'analyser la réponse du circuit sur une plage de fréquences, et les résultats peuvent être exportés au format S1P pour une réutilisation ultérieure.

Finalement, l'application offre des outils pratiques comme le tuning des composants, l'utilisation d'un catalogue de composants normalisées CEI 60063 créé par l'utilisateur, la prise en compte du facteur de qualité des composants pour des simulations plus réalistes, la

gestion de l'historique (undo/redo), et la sauvegarde complète des projets sur lesquels travaille l'utilisateur.

Étudiant : Date et lieu : Signature :

Edwin Häffner

Enseignant responsable : Date et lieu : Signature :

Prof. Bertrand Hochet

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le 08.01.2026

PRÉAMBULE

Edwin Häffner

Authentification

Le soussigné, Edwin Häffner, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées

Yverdon-les-Bains, le 08.01.2026

Edwin Häffner

AUTHENTIFICATION

Cahier des charges

Résumé du problème

L'abaque de Smith est un outil très important dans le domaine de la conception et de l'analyse d'antennes. Il permet de visualiser et de manipuler les impédances complexes en les projetant sur le plan du coefficient de réflexion.

Dans un monde parfait, une antenne devrait transférer toute sa puissance vers le milieu de propagation, l'air par exemple, sans aucune perte. Cependant, en réalité, l'impédance des antennes est très rarement adaptée à celle de la ligne de transmission qui leur fournit le signal. Cette différence d'impédance crée des ondes stationnaires et un phénomène de réflexion. Cette réflexion, en plus d'être potentiellement dangereuse pour la source de courant (puisque une partie de la puissance émise est renvoyée vers celle-ci), diminue également l'efficacité de l'antenne.

Pour pallier ce problème, on peut adapter une impédance à une autre en ajoutant, en série ou en parallèle (shunt), des condensateurs ou des inductances. C'est là que l'abaque de Smith entre en jeu, il aide l'utilisateur à choisir les bons composants afin de réduire le coefficient de réflexion et d'obtenir une adaptation optimale, c'est à dire maximiser le transfert de puissance de la source à la charge.

Actuellement, les étudiants en systèmes embarqués utilisent un logiciel développé par Fritz Dellisperger, un professeur de la Haute école spécialisée bernoise (BFH) aujourd'hui à la retraite, pour leurs laboratoires sur les antennes. Ce logiciel, exclusif à Windows et appelé **Smith V4.1**, n'est plus maintenu depuis 2018 et présente plusieurs problèmes d'affichage et de dimensionnement, rendant son interface peu ergonomique et difficile à utiliser.

Problématique

Comment fournir aux étudiants un logiciel intuitif, simple d'utilisation et agréable à prendre en main pour manipuler numériquement un abaque de Smith ?

Solutions existantes

Il existe bien entendu d'autres logiciels, natifs ou en ligne, qui permettent de manipuler un abaque de Smith.

Par exemple, **SimSmith** [1] est un outil très complet développé par AE6TY [1] est un outil multiplateforme développé en Java. L'outil semble très complet, quoique pas très intuitif au premier abord.

linSmith [2] est aussi une bonne alternative, avec une interface utilisateur agréable. Son gros point faible est qu'il ne tourne que sous Linux.

Il existait également **Iowa Hills Smith Chart**, mais le développement semble être tombé en désuétude et il n'y a apparemment plus de moyen de télécharger l'application actuellement.

Finalement, il existe des solutions purement en ligne telles que QuickSmith¹ et Online Smith Chart Tool². Ces outils fonctionnent mais ne proposent pas forcément les fonctionnalités voulues pour l'étendue de ce TB. Aussi avoir une application tournant nativement sur sa machine et hors ligne est un plus.

Solutions possibles

Pour ce travail de bachelor, la meilleure option semble être de développer une solution dédiée, directement avec les utilisateurs de l'abaque de Smith à l'HEIG. Ça permettra de remplacer les logiciels obsolètes ou qui ne font pas exactement ce qu'on veut. En plus, avoir une solution multiplateforme est un point vraiment important.

Cahier des charges

Objectifs

Les objectifs principaux du projet sont :

- Fournir un logiciel multiplateforme (volonté de ne pas forcer l'utilisateur à un système d'exploitation).
- Avoir un logiciel avec une interface moderne, intuitive, adaptée à l'enseignement à l'HEIG (pas besoin d'être exhaustif dans les fonctionnalités, seulement ce dont on a besoin).
- Fonctionnalités similaires à Smith.exe pour ne pas être dépaysé.
- Avoir un logiciel évolutif dans le code.

¹Lien du site : <https://quicksmith.online/>

²Lien du site : <https://onlinesmithchart.com/>

Spécifications fonctionnelles

Fonctionnalités minimales :

- Affichage d'un abaque de Smith interactif.
- Placement d'impédances et/ou d'admittances.
- Placement de composants passifs en série ou en parallèle (R, L, C, ligne, stub).
- Chargement de fichiers S1p.
- Fonction Sweep et tuning.

Fonctionnalités avancées :

- Édition de plusieurs circuits simultanément (jusqu'à 10) à partir d'une même paire [charge, source].
- Choix de couleur pour la trajectoire de chaque circuit.
- Fenêtre "Cursor" : curseur aimanté sur les points calculés.
- Zoom et pan paramétrables.
- Export de fichiers S1p issus d'un sweep.
- Tuning basé sur des valeurs normalisées CEI 60063.
- (Optionnel) Affichage Z, Y, Mod/Arg pour les points intermédiaires.

Déroulement

Le travail commence le 3 novembre 2025 et se termine le 16 janvier 2026. L'entièreté de ce travail sera réalisé à plein temps.

Un rendu intermédiaire noté est demandé le 25 novembre 2025 à 15 heures et le rendu final est prévu pour le 16 janvier 2026 à 12 heures.

CAHIER DES CHARGES

Table des matières

Préambule	5
Authentification	7
Cahier des charges	9
Résumé du problème	9
Problématique	9
Solutions existantes	10
Solutions possibles	10
Cahier des charges	10
Objectifs	10
Spécifications fonctionnelles	11
Déroulement	11
1 Introduction	17
1.1 Origine	17
1.2 Actualité	18
2 Principes théoriques	19
2.1 Pourquoi l'abaque de Smith ?	19
2.1.1 Quantifier la réflexion	20
2.1.2 Représentation graphique	21
2.1.3 Lire l'abaque	22
2.1.4 Utiliser l'abaque	24
3 État de l'art	27
3.1 Fonctionnalités souhaitées	27
3.2 Logiciel déjà existant	27

TABLE DES MATIÈRES

3.2.1 Smith (Smith.exe)	27
3.3 Bibliothèques et frameworks graphiques pour applications multiplateformes	28
3.3.1 Qt (C++/Python)	29
3.3.2 JavaFX	29
3.3.3 GTK	29
3.3.4 Flutter	30
3.3.5 Tauri (Rust)	30
3.4 Choix et justification	31
3.5 Bibliothèque de sérialisation	31
4 Planification	33
4.1 Planification initiale	33
4.2 Planification semaine par semaine	33
4.2.1 Semaine 1 du 03.11.2025 au 09.11.2025 : Début du travail de bachelor	33
4.2.2 Semaine 2 du 10.11.2025 au 16.11.2025 : Prise en main et bases	33
4.2.3 Semaine 3 du 17.11.2025 au 23.11.2025 : Interaction de base sur l'abaque	33
4.2.4 Semaine 4 du 24.11.2025 au 30.11.2025 : Finalisation de l'interface et jalon ..	34
4.2.5 Semaine 5 du 01.12.2025 au 07.12.2025 : Gestion des données (Import)	34
4.2.6 Semaine 6 du 08.12.2025 au 14.12.2025 : Fonctionnalités avancées	34
4.2.7 Semaine 7 du 15.12.2025 au 23.12.2025 : Export data et rédaction du rapport ..	34
4.2.8 Semaine 8 du 24.12.2025 au 30.12.2025 : Finalisation des fonctionnalités et tests	34
4.2.9 Semaine 9 du 05.01.2026 au 11.01.2026 : Redaction et correction de problèmes	35
4.2.10 Semaine 10 du 12.01.2026 au 16.01.2026 : Finalisation et rendu	35
4.3 Rendu final	35
5 Architecture	37
5.1 Partie Vue	38
5.1.1 MainController	39
5.1.2 SmithChartInteractionController	40
5.1.3 Filtrage des fichiers S1P	41
5.2 Partie Modèle	42
5.2.1 Données et éléments du circuit	42
5.2.2 Les unités	43
5.2.3 Les points de données	43
5.2.4 Gestion des fichiers S1P	44

5.2.5 Les calculs mathématiques	44
5.2.6 Gestion des « undo/redo »	47
5.2.7 Les composants discrets	48
5.2.8 Le management du projet	48
5.3 Partie ViewModel	49
5.3.1 Les JavaFX Properties	49
5.3.2 Le recalculation des éléments de l'abaque	50
5.3.3 Gestion de plusieurs circuits	51
5.3.4 Utilisation du pattern Singleton	51
6 État de l'application au rapport intermédiaire	53
6.1 Déploiement	54
6.2 Le ViewModel	54
6.2.1 Le binding	54
6.2.2 La chaîne de calcul d'impédance	55
6.2.3 L'interaction souris et l'historique	55
6.3 Dessin de l'abaque	55
6.3.1 L'abaque lui-même	56
6.3.2 Les points d'impédance	56
6.3.3 Le tracé du circuit d'impédance	56
7 Conclusion intermédiaire	57
Bibliographie	59
Outils utilisés	63
Programmation	63
Conception	63
Gestion de versions	64
Rapport	64
Utilisation de LLMs	64

TABLE DES MATIÈRES

1 Introduction

Avant de parler de la conception d'un logiciel ou quoi que ce soit d'autre, il faut tout d'abord comprendre ce qu'est l'abaque de Smith et d'où cet outil provient.

1.1 Origine

L'abaque de Smith tire son nom de son inventeur, l'ingénieur américain Phillip Hagar Smith. Né le 29 avril 1905 à Lexington, dans le Massachusetts, il grandit avec une vraie passion pour la radio. Pendant son cursus universitaire, il a même monté sa propre station amateur (avec l'identifiant 1ANB) en utilisant des composants faits maison.[3]

Logiquement, il poursuit des études d'ingénierie électrique et, en 1928, il obtient son diplôme à l'université Tufts, qui aboutit à son premier emploi aux laboratoires téléphoniques Bell dans le département de la recherche radio. L'une de ses missions consistait à adapter des antennes aux lignes de transmission, le problème que l'abaque de Smith cherche justement à résoudre.

Dans un premier temps, Smith développe une méthode rapide pour pouvoir calculer les impédances des lignes de transmission en utilisant un graphique rectangulaire. Cependant, ce graphique rectangulaire présentait rapidement ses limites. Smith, entouré de deux de ses collègues, E.B. Ferrell et J.W. McRae, développe alors une transformation mathématique permettant de projeter les impédances sur un plan circulaire correspondant au coefficient de réflexion³, offrant ainsi une représentation beaucoup plus claire et pratique. [4]

C'est en 1939 que Smith publie dans le volume 12 du journal « Electronics » officiellement l'abaque de Smith, qui est toujours et encore utilisé aujourd'hui, et c'est cet abaque que nous allons étudier et digitaliser.

³Note : Explication dans la partie des principes théoriques

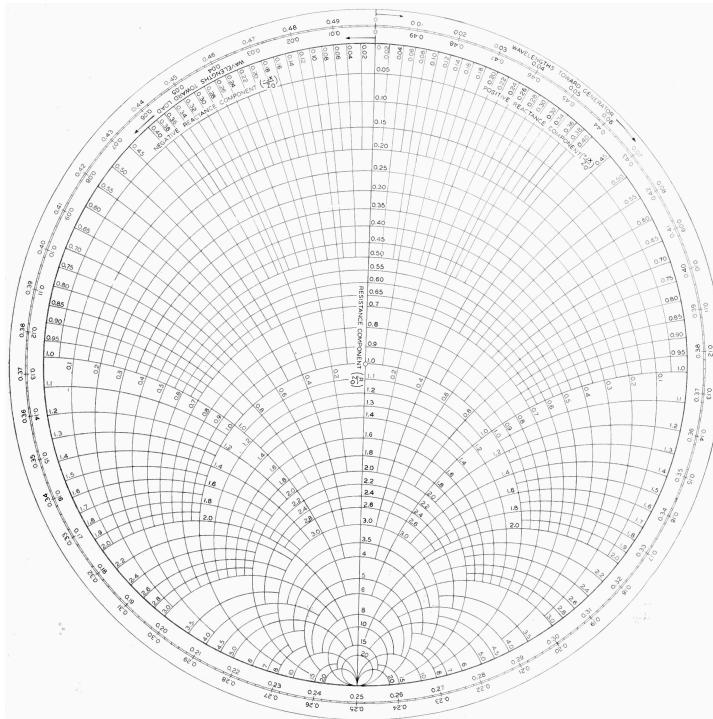


Fig. 1. – L'abaque de Smith tel que présenté dans le journal « Electronics »

1.2 Actualité

En 2025, plus de 85 ans après sa première publication, l'abaque de Smith reste un outil incontournable pour les ingénieurs et les étudiants du monde entier. Si son adoption fut lente au début, elle a été considérablement accélérée par le MIT Radiation Laboratory, qui l'a massivement utilisé dès 1940 pour le développement du radar durant la Seconde Guerre mondiale. [4]

Aujourd'hui, même à l'ère du numérique, son importance n'a pas diminué. Elle offre une manière visuelle et intuitive de comprendre des phénomènes complexes, comme l'adaptation d'impédance, permettant à presque n'importe qui de concevoir un circuit d'antenne fonctionnel. C'est l'affichage de référence lorsqu'on veut faire de l'adaptation d'impédance. C'est dans ce contexte que ce travail de bachelor prend tout son sens, moderniser un outil fondamental pour répondre aux besoins actuels des étudiants et des ingénieurs.

2 Principes théoriques

Pour pouvoir développer une solution d'abaque de Smith numérique, il faut tout d'abord comprendre comment celle-ci fonctionne et comment on l'utilise. Ce chapitre a pour but de poser les bases théoriques nécessaires à la compréhension de l'outil.

2.1 Pourquoi l'abaque de Smith ?

L'abaque de Smith existe dans le contexte de l'utilisation et la transmission des fréquences radio (RF). Dans ce domaine, on a souvent affaire à des circuits qui opèrent à hautes fréquences et dont le comportement est décrit par une grandeur complexe, l'impédance.

L'impédance Z est une grandeur complexe qui allie une partie réelle, la résistance R totale d'un circuit, avec une partie imaginaire qui s'appelle la réactance X . [5]

$$Z = R + jX$$

Ensuite cette réactance peut soit être capacitive X_C (comme un condensateur) ou bien elle peut être inductive X_L (comme une bobine). La valeur de la résistance est généralement considérée comme constante par rapport à la fréquence, tandis que la réactance en dépend fortement, la réactance inductive augmente avec la fréquence, alors que la réactance capacitive diminue.

$$X_C = -\frac{1}{2\pi f C} \quad X_L = 2\pi f L$$

En radiofréquence, l'objectif est de transférer un signal le plus efficacement possible entre une source émettrice et une charge (une antenne, par exemple). Pour ce faire, il faut assurer que le transfert de puissance soit maximal. Si l'on envoie 100W de puissance RF, on veut que notre antenne rayonne l'entièreté de ces 100W.

Si ce n'est pas le cas, on parle de désadaptation d'impédance. L'énergie qui n'est pas transmise à la charge est alors réfléchie vers la source, ce qui induit deux problèmes majeurs: une perte d'efficacité, car une partie de la puissance transmise n'est tout simplement pas rayonnée et un danger pour la source qui peut être endommagée par cette réflexion.

Pour obtenir ce transfert de puissance maximal, l'impédance de la charge doit être le conjugué complexe de l'impédance de la source. Malheureusement, ce cas idéal n'arrive que très rarement en pratique. Il faut alors insérer un circuit d'adaptation entre la source et la charge. Le rôle de ce circuit est de transformer l'impédance de la charge pour qu'elle soit adaptée à sa source.

$$\text{Adaptation maximale : } Z_{\text{Source}}^* = Z_{\text{Charge}}$$



Fig. 2. – On adapte la partie réactive ($+jX$ et $-jX$) et maximise le transfert de puissance

2.1.1 Quantifier la réflexion

Cette désadaptation est quantifiable mathématiquement avec le coefficient de réflexion gamma (Γ). Ce coefficient est calculé avec cette formule :

$$\Gamma = \frac{Z_L - Z_0}{Z_L + Z_0}$$

Où Z_L est l'impédance de la charge et Z_0 est l'impédance de la ligne de transmission (50 Ohm dans les cas les plus courants). Avec cette formule, on peut voir que, si l'impédance de la source et de la charge sont les mêmes, on obtient alors 0, une adaptation parfaite. Alors qu'envers, plus la différence entre Z_0 et Z_L est grande, plus le module du coefficient se rapprochera du 1, une réflexion alors totale de l'énergie.

Physiquement, les conséquences de cette réflexion sont l'apparition d'onde stationnaire. Ces ondes sont des pics de courant qui arrivent lorsqu'un milieu câblé reçoit un courant de retour en plus d'un courant aller. Ce qu'il se passe, c'est que les deux fréquences vont se superposer et former des ondes stationnaires.

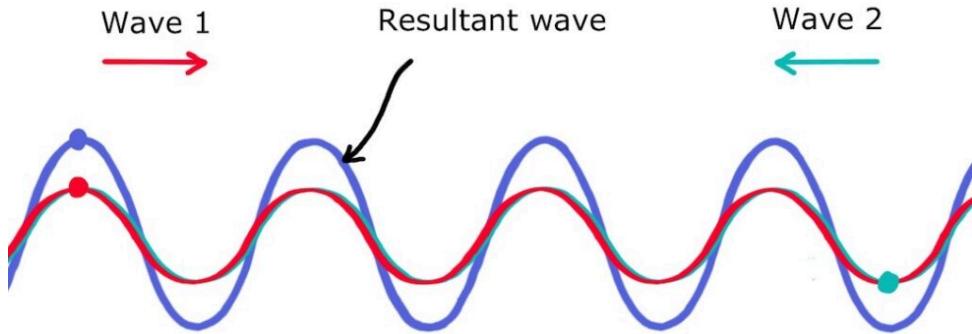


Fig. 3. – Phénomène des ondes stationnaires [6]

Ce sont ces ondes stationnaires qui sont les causes physiques des problèmes mentionnés plus haut. Et à partir de ces ondes stationnaires, on peut calculer le VSWR (Voltage Standing Wave Ratio) qui est une mesure du rapport entre l'amplitude maximale et l'amplitude minimale que peut atteindre l'onde stationnaire.

$$\text{VSWR} = \frac{1+|\Gamma|}{1-|\Gamma|}$$

Finalement, à partir de cette mesure, on peut aussi calculer le montant de puissance du signal qui est retourné à la source, une mesure appelée Return Loss, qui est exprimée en décibel.

$$\text{Return Loss} = 20 \log_{10} \left(\frac{\text{VSWR} + 1}{\text{VSWR} - 1} \right)$$

2.1.2 Représentation graphique

Sans entrer dans des détails trop exhaustifs pour le projet, les calculs pour trouver les bons composants étaient longs et fastidieux, et on a vite commencé à utiliser des représentations graphiques pour trouver ces composants.

La représentation graphique est une représentation du point d'impédance de la charge Z_L par rapport à l'impédance de référence de la ligne Z_0 . On pose alors sur l'axe des ordonnées la relation imaginaire, la réactance X . Et sur l'axe des abscisses, la partie réelle, la résistance.

Il est complexe de représenter cette relation sur le plan cartésien, car on utilise d'une part seulement le côté droit du plan, étant donné que la résistance ne peut être négative, mais surtout qu'autant les valeurs de réactance comme celles de résistance peuvent pointer vers l'infini, rendant impossible la visualisation de toutes les impédances sur une seule feuille. [7]

C'est là que l'abaque de Smith offre la solution. Au lieu de représenter directement le plan des impédances, l'abaque de Smith est une représentation graphique du plan du coefficient de réflexion complexe Γ .

Comme le module de Γ est toujours compris entre 0 et 1, toutes les valeurs possibles peuvent être contenues à l'intérieur d'un cercle de rayon 1. Smith a ensuite développé une transformation mathématique qui permet de superposer des lignes de résistance et de réactance constantes sur ce même plan circulaire. On obtient ainsi un diagramme où chaque point correspond à la fois à une valeur d'impédance unique et son coefficient de réflexion associé.

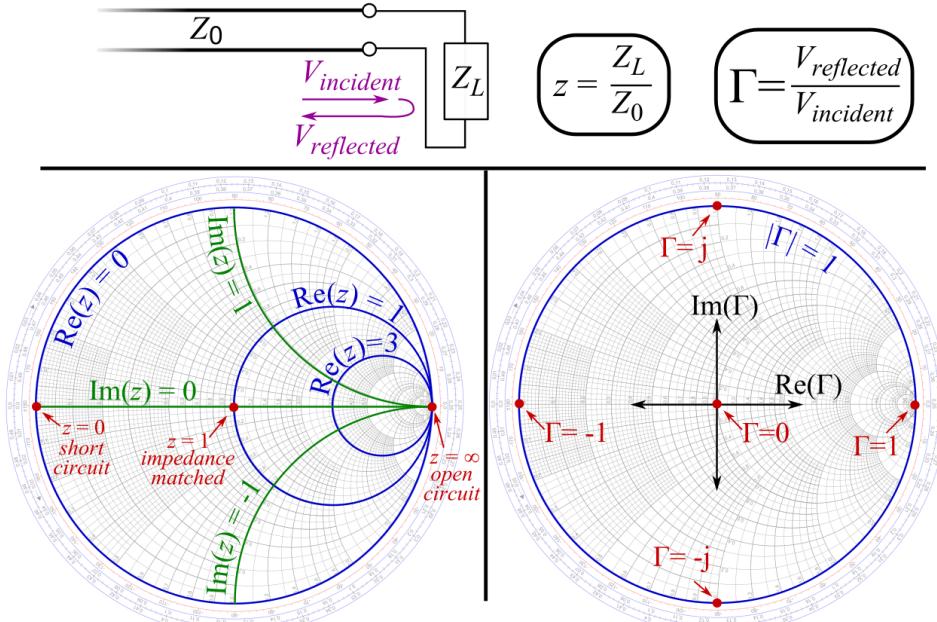


Fig. 4. – Explication graphique de l'abaque [8]

2.1.3 Lire l'abaque

Tout comme sur le schéma sur un plan cartésien, l'axe des ordonnées est l'axe des nombres imaginaires et l'axe des abscisses est l'axe des nombres réels, la résistance. Du côté gauche complètement réel de l'abaque, on a une résistance nulle, un court-circuit. Et du côté droit complètement réel, on a une résistance « infinie », en gros un circuit ouvert. À partir de là, on peut voir que l'abaque est dessiné avec deux types de lignes. D'abord, il y a des cercles qui se touchent tous au point de circuit ouvert. Ce sont les cercles de résistance constante. Chaque point se trouvant sur un même cercle partage exactement la même partie résistive. [9]

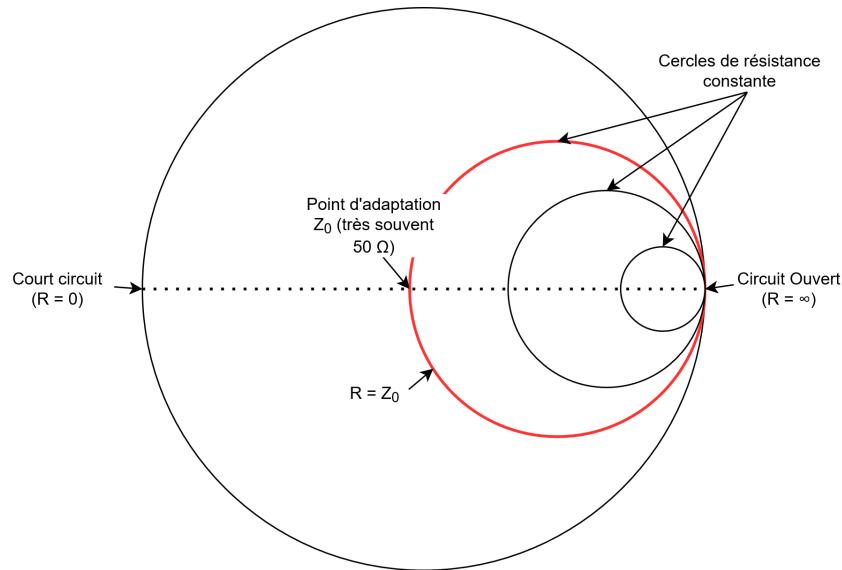


Fig. 5. – Représentation des cercles de résistance constante sur l'abaque de Smith

Ensuite, des courbes partent toutes du même point de circuit ouvert et s'étendent vers l'extérieur. Ces courbes représentent les lignes de réactance constante. La moitié supérieure de l'abaque correspond aux réactances inductives ($+jX$), tandis que la moitié inférieure représente les réactances capacitives ($-jX$). Tout comme pour les cercles, chaque point situé sur la même courbe possède la même partie réactive.

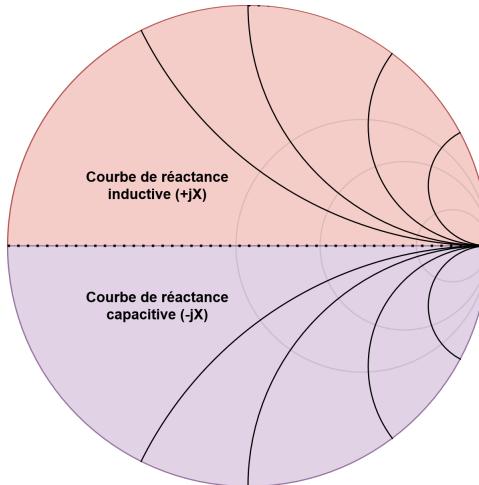


Fig. 6. – Représentation des courbes de réactance constante

Jusqu'ici, on a parlé de l'abaque d'impédance, aussi appelé la forme Z de l'abaque. Il est parfait pour visualiser l'ajout de composants en série, car les impédances s'additionnent. Mais pour pouvoir pleinement utiliser l'outil dans le cas de la recherche de circuits d'adaptation, il nous faut aussi pouvoir gérer les composants en parallèle (shunt).

Pour cela, on utilise aussi un abaque d'admittance (Y) qui est simplement l'inverse de l'impédance ($Y = \frac{1}{Z}$). La partie réelle de l'admittance est appelée la conductance et la partie imaginaire la susceptance.

Graphiquement, cet abaque est exactement le même que l'abaque des impédances, mais pivoté de 180 degrés, on peut alors superposer les deux abaques dans un graphe combiné qui est aussi appelé abaque $Z - Y$. C'est cette visualisation de l'abaque qui est le plus pratique et qui est utilisée majoritairement par les logiciels actuels. Cela sera aussi cet abaque que nous allons dessiner dans notre version du logiciel.

2.1.4 Utiliser l'abaque

Maintenant, pour l'utiliser, rien de plus simple, on mesure l'impédance de notre composant de charge. On positionne cette mesure sur l'abaque. Dans le cas d'un abaque papier, cette mesure devra être normalisée par rapport à l'impédance de référence. Par exemple, si notre impédance mesurée est de $75 + j50$ Ohms dans un système à 50 Ohms, cette impédance se trouvera sur un point normalisé de $1.5 + j1$ sur l'abaque.

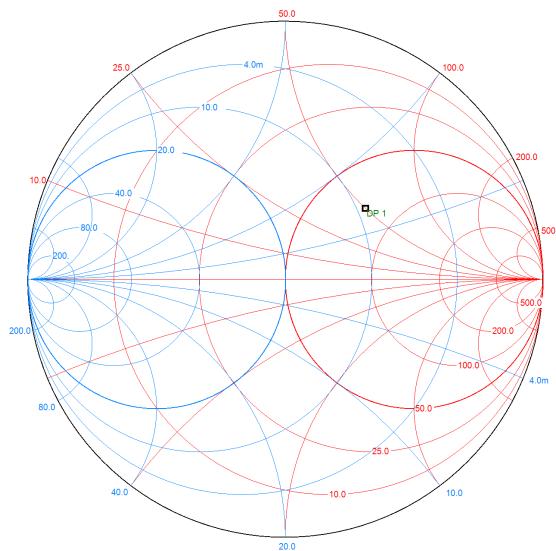


Fig. 7. – Impédance de $75 + j50$ Ohm

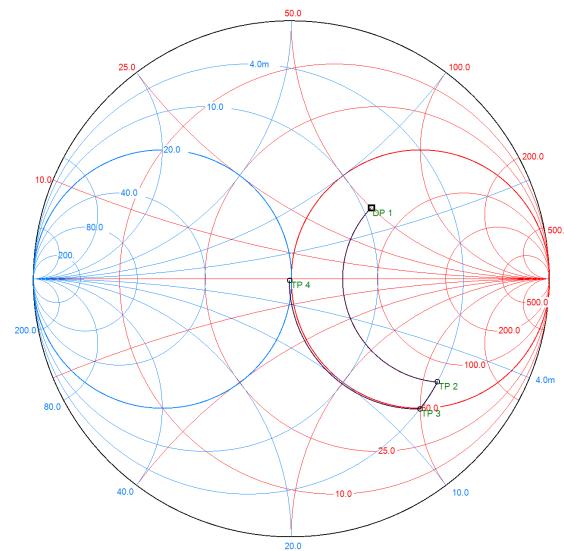


Fig. 8. – Impédance avec circuit d'adaptation⁴

⁴Source : Logiciel Smith.exe

Ensuite, le but du jeu est de ramener ce point le plus proche du centre de l'abaque possible, le point d'adaptation parfaite. Et pour y arriver, on va ajouter des composants, des inductances et des condensateurs, soit en série, soit en parallèle. Ce n'est pas le seul moyen de pouvoir adapter la charge, mais c'est le moyen sur lequel ces explications vont se reposer.

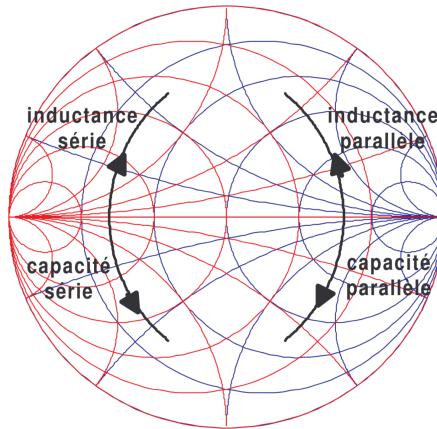
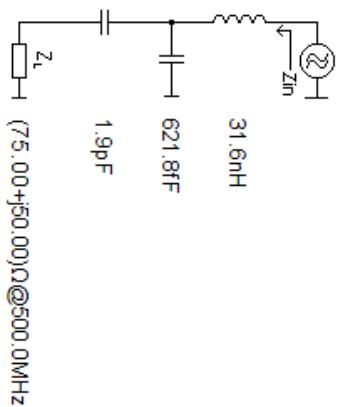


Fig. 9. – Effet d'ajouts d'inductances et de capacités en série et parallèle [9]

Selon le composant qu'on ajoute et selon son placement dans le circuit, l'impédance vue depuis la source va changer de position sur l'abaque. En se basant sur l'exemple de notre impédance mesurée, une des possibilités est d'utiliser le circuit ci-dessous pour réaliser l'adaptation:



1. Ajout d'un condensateur en série de 1.9pF pour rejoindre le cercle de conductance qui à l'échelle de 50 Ohms indique une conductance de 4 millisiemens constante.
2. Ajout d'un condensateur en parallèle de 621.8 fF pour ensuite rejoindre le cercle de résistance constante de 50 Ohms, c'est grâce à celui-ci que nous allons pouvoir nous rapprocher du centre.
3. Finalement, l'ajout d'une inductance en série de 31.6 nH pour nous ramener vers le centre de l'abaque en suivant le cercle de résistance constante de 50 Ohms.

C'est en cela que l'abaque de Smith est extrêmement puissant.

PRINCIPES THÉORIQUES

3 État de l'art

3.1 Fonctionnalités souhaitées

Le but de ce projet est de proposer un outil moderne et complet pour l'utilisation de l'abaque de Smith. Parmi les fonctionnalités attendues, il doit être possible d'éditer plusieurs circuits par projet, et non pas être limité à un seul.

3.2 Logiciel déjà existant

L'idée initiale de ce travail de bachelor est de moderniser l'outil existant développé par Fritz Dellisperger, sur lequel s'appuient actuellement les étudiants pour leurs laboratoires. Le produit final s'en inspirera largement tout en cherchant à corriger ses principales limites et en ajoutant les nouvelles fonctionnalités souhaitées.

3.2.1 Smith (Smith.exe)

Smith, ou plus couramment **Smith.exe** [10], est un logiciel dont la dernière mise à jour date de 2018. Il fonctionne uniquement sous Windows.

ÉTAT DE L'ART

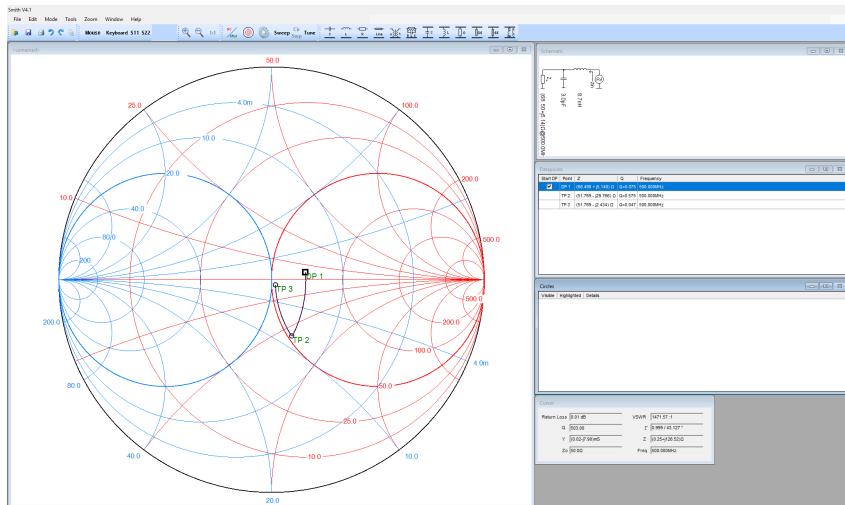


Fig. 10. – Aperçu de Smith.exe

Le principal problème de ce logiciel est son absence de support sur d'autres plateformes, comme Linux ou macOS. Son ergonomie, basée sur un système de sous-fenêtres daté, rend son utilisation peu intuitive. De plus, plusieurs fonctionnalités manquent par rapport aux besoins définis pour ce projet.

Mais, vu que le projet va largement s'inspirer de celui-ci, il est important de comprendre comment il fonctionne.

3.3 Bibliothèques et frameworks graphiques pour applications multi-plateformes

Afin de développer une application moderne et accessible sur plusieurs systèmes d'exploitation, il est important de choisir des technologies adaptées. Cette section présente une analyse des principaux frameworks et bibliothèques permettant la création d'interfaces graphiques multi-plateformes.

Dans le cadre de ce projet, le public cible étant constitué d'étudiants et d'ingénieurs, le choix technologique doit répondre à plusieurs exigences. L'application doit avant tout être multiplateforme afin d'assurer une compatibilité avec Windows, macOS et Linux. Elle doit également être moderne visuellement, de manière à offrir une interface intuitive et agréable à utiliser. Enfin, elle doit rester performante pour garantir une expérience fluide, même sur des machines pas très performantes.

Ces trois critères serviront de base pour évaluer les différentes solutions envisagées dans la suite de ce travail.

3.3.1 Qt (C++/Python)

Qt est un framework multiplateforme très répandu, utilisé tout aussi bien dans l'industrie que dans le monde académique. Il offre une excellente performance, une interface native sur chaque système et il possède une large communauté active.

Le fait que Qt puisse être utilisé en C++ ou en Python est un avantage, bien que Python puisse poser des limites de performance dans le cadre de calculs intensifs.

Après quelques essais, l'interface graphique de Qt Designer et les modules de dessins de Qt semblent être vraiment bien adaptés au projet. Le seul mauvais point notable est que, pour pouvoir compiler de façon multiplateforme, il faut disposer de différentes machines sur lesquelles compiler. C'est-à-dire avoir une machine Windows, Linux (avec ses différents distro) et MacOS. Ce n'est pas un problème avec les machines virtuelles, mais cela représente tout de même un frein, surtout que l'affichage pourrait ne pas être exactement le même selon sur quel OS le logiciel final tourne.

3.3.2 JavaFX

JavaFX est un framework de développement d'applications multiplateforme, puisqu'il repose sur Java et sa fameuse Machine virtuelle (JVM), fidèle au slogan « Write once, run anywhere ».

C'est aussi un framework très complet, enrichi par une communauté active qui propose de nombreux composants graphiques prêts à l'emploi (comme MaterialFX [11] ou AtlantaFX [12]). C'est un atout majeur pour obtenir facilement une interface moderne et agréable à utiliser.

L'un des plus grands avantages de Java, c'est qu'il est multiplateforme par nature. Pas besoin de s'embêter avec la cross compilation, on produit un seul fichier qui fonctionnera partout où la JVM est installée, et quasiment tout appareil a Java d'installé.

Et question performance, les dernières versions de Java sont très efficaces. On a pu le constater en réalisant des simulations de recherche de chemin optimum dans des graphes dans le cadre du cours « Optimisation et Simulation » donné par J.F. Héches.

Finalement, après un premier test, la prise en main de JavaFX s'est révélée intuitive et bien adaptée aux besoins du projet.

3.3.3 GTK

GTK, autrefois GIMP ToolKit est un toolkit qui permet d'effectuer des GUI majoritairement sur Linux, mais est aussi compatible avec Windows et MacOS. C'est un outil très utilisé

dans la communauté Linux, son avantage est que sa librairie fonctionne avec plusieurs langages différents, C, JavaScript, Perl, Python, Rust, etc...

Cela peut-être une bonne alternative, même s'il est possible que le support soit limité sur les plateformes autres que Linux.

3.3.4 Flutter

Flutter est un framework créé par Google pour faire des applications multiplateformes. D'après les retours sur Reddit et ailleurs, Flutter a l'air solide avec des APIs matures et un bon rendu graphique.

Le principal frein est que le langage utilisé est le Dart. L'apprentissage d'un nouveau langage représenterait une charge non négligeable au travail. En plus, Flutter a été pensé d'abord pour le mobile, le support desktop étant quelque chose de plus récent, même si apparemment bien supporté.

De plus, une considération par rapport au projet est la maintenabilité future. Lorsque mon projet sera terminé, sûrement d'autres étudiants ou même ingénieurs seront susceptibles d'ajouter de nouvelles fonctionnalités. Sachant que le logiciel est destiné aux ingénieurs en électrique, il vaudrait mieux se focaliser sur une solution utilisant des langages avec lesquels ils sont déjà à l'aise, le C, le C++ et possiblement le Java.

3.3.5 Tauri (Rust)

Tauri est un framework qui utilise une architecture du type web pour le frontend et un backend en Rust. Vu que l'application tourne en backend avec Rust et en frontend avec une technologie web tel que React, Vue, etc. Ce choix peut-être le bon vu que ça peut être un bon entraînement à Rust et l'utilisation de technologie Web m'est plus que familière.

Un grand plus de ce framework est aussi la performance et sa moindre empreinte mémoire.

Après avoir fait quelques recherches, on peut voir que, si l'on veut faire quelque chose de graphique avec Tauri comme dessiner l'abaque de Smith, l'utilisation d'une bibliothèque JavaScript est obligatoire pour le faire, ce qui pose un grand frein pour l'adaptation de ce framework. De plus, le backend ne sera pas très complexe dans ce projet or, avec Tauri, toute la partie graphique doit être réalisée en JavaScript via une technologie web (React, Vue, etc.), ce qui ajoute une couche supplémentaire et ne semble pas pertinent ici. Une solution où le dessin et l'interactivité sont gérés directement dans le langage principal du framework et qui n'utilise si possible pas de JavaScript est préférable.

3.4 Choix et justification

Finalement, après avoir passé en revue ces différents frameworks, ce qui semble le plus adapté pour l'envergure de ce travail de bachelor reste JavaFX.

Qt était un bon concurrent, de par sa performance et sa maturité. Mais le problème de déploiement, de compilation multi-plateforme représente un obstacle trop important pour un projet solo mené dans un temps limité. Trop de temps serait potentiellement perdu lorsqu'une compilation devrait être fait.

C'est la raison pour laquelle l'utilisation de Java est logique ici, pas besoin de se focaliser sur la compilation, juste besoin de produire un seul exécutable compatible avec toutes les machines, tant qu'elles ont Java d'installé.

Les autres solutions ont été écartées, car soit le langage n'était pas adapté aux besoins qui dépassent l'étendue de ce TB, notamment si le travail est repris par d'autres étudiants de la filière électrique.

Donc, en conclusion, le projet d'abaque de Smith sera codé en utilisant du Java et en utilisant le framework JavaFX, qui est plus que suffisant pour l'étendue de ce travail.

3.5 Bibliothèque de sérialisation

Lors de la conception, un choix a dû être fait concernant la bibliothèque de sérialisation pour permettre de sauvegarder et charger les projets (les composants du circuit, la fréquence, etc.).

Mon choix s'est rapidement porté sur Jackson. C'est une bibliothèque que j'avais déjà utilisée sur d'autres projets, ce qui m'a permis d'être efficace sans avoir à réaliser un comparatif exhaustif avec d'autres solutions. Elle est simple d'utilisation, toujours activement maintenue, et la vitesse de sérialisation n'était pas un critère critique pour ce projet donc pas besoin de faire des tests par rapport à cela.

De plus, Jackson s'est révélé particulièrement adapté grâce à sa gestion facile du polymorphisme, pas besoin de grandement changer le code pour que cela fonctionne. Cela m'a permis de sauvegarder ma liste de composants variés (Résistances, Lignes, Condensateurs, ...) très simplement, sans avoir à écrire de code complexe pour différencier chaque type d'objet dans le fichier de sauvegarde.

ÉTAT DE L'ART

4 Planification

4.1 Planification initiale

Ce document détaille la planification semaine par semaine pour la réalisation du travail de bachelor. Le projet se déroule à temps plein sur une période de 10 semaines, avec un rendu final fixé au **vendredi 16 janvier 2026 à 12h00**. La planification intègre des phases de développement, de test, de rédaction et une marge de sécurité pour les imprévus.

4.2 Planification semaine par semaine

4.2.1 Semaine 1 du 03.11.2025 au 09.11.2025 : Début du travail de bachelor

Durant la première semaine de travail, l'idée est de se focaliser sur les aspects techniques et théoriques de l'abaque de Smith, comprendre ses enjeux et son utilisation. Un début de rédaction du rapport est aussi nécessaire avec l'aspect historique et une analyse des différents outils possibles à utiliser pour la conception du logiciel de l'abaque de Smith.

Finalement à la fin de la semaine, un framework doit être choisi pour ensuite commencer à mettre en place les différentes étapes du développement.

4.2.2 Semaine 2 du 10.11.2025 au 16.11.2025 : Prise en main et bases

- Gestion et validation de la planification totale.
- Début de la conception du projet avec un diagramme de classe UML.
- Prise en main de l'outil JavaFX et mise en place des éléments clés :
 - Dessin de l'abaque de Smith.
 - Ajout manuel d'impédances sur l'abaque.
 - Système de mémoire pour les points ajoutés.

4.2.3 Semaine 3 du 17.11.2025 au 23.11.2025 : Interaction de base sur l'abaque

- Dessin du chemin du circuit en fonction des éléments ajoutés (inductance, capacité, série/shunt).

- Implémentation du curseur aimanté sur les points calculés.
- Affichage des informations du curseur dans une fenêtre dédiée (“Cursor”) comme Smith.exe le fait.
- Finalisation et relecture du rapport.

4.2.4 Semaine 4 du 24.11.2025 au 30.11.2025 : Finalisation de l’interface et jalon

- **Rendu du rapport intermédiaire le 25 novembre.**
- Affichage schématique du circuit avec ses composants.
- Implémentation des fonctions de **Zoom** et **Pan** paramétrables pour la navigation.
- Ajout des composants manquants, résistances, lignes et stubs avec leurs comportements respectifs.
- Amélioration générale de l’interface graphique.

4.2.5 Semaine 5 du 01.12.2025 au 07.12.2025 : Gestion des données (Import)

- Charger et afficher les fichiers S-parameter (S1p).
- Recherche sur la structure du format de fichier S1p.
- Création d’un parser pour lire et interpréter les données.
- Traduction des données en points sur l’abaque.

4.2.6 Semaine 6 du 08.12.2025 au 14.12.2025 : Fonctionnalités avancées

- Implémentation de la fonction de **Sweep** (balayage en fréquence).
- Développement de la fonction de **Tuning** de base (ajustement des valeurs).
- Mise à jour du diagramme de classe UML pour refléter l’architecture actuelle.

4.2.7 Semaine 7 du 15.12.2025 au 23.12.2025 : Export data et rédaction du rapport

- Implémentation de l’export de fichiers S1p issus d’un sweep.
- Recherche et implémentation des notations utilisant les valeurs normalisées CEI 60063.
- Début de la rédaction du rapport final sur la base du rapport intermédiaire, début d’écriture des parties liées à l’implémentation et à l’architecture.

4.2.8 Semaine 8 du 24.12.2025 au 30.12.2025 : Finalisation des fonctionnalités et tests

- Permettre la gestion de plusieurs circuits simultanément en partant du même projet, garder la même base source, charge.
- (Optionnel si le temps est disponible) Ajouter les options de personnalisation de l’interface (couleurs, etc.).
- Début d’une phase de tests et de debuggage de l’application générale.

4.2.9 Semaine 9 du 05.01.2026 au 11.01.2026 : Redaction et correction de problèmes

- Correction des bugs critiques identifiés et amélioration/finalisation des features si besoin.
- Rédaction du corps du rapport (architecture, implémentation, choix techniques).

4.2.10 Semaine 10 du 12.01.2026 au 16.01.2026 : Finalisation et rendu

- Finalisation de la rédaction (conclusion, résumé).
- Relecture complète du rapport et préparation de la version finale du code.
- Préparation du rendu.

4.3 Rendu final

Le rendu final du travail de bachelor est fixé au **vendredi 16 janvier 2026, à 18h00 au plus tard**.

PLANIFICATION

5 Architecture

L'application est un logiciel qui doit afficher des éléments graphiques complexes (l'abaque) et permettre à l'utilisateur de visualiser des données pour effectuer l'adaptation d'impédance.

Comme annoncé dans l'état de l'art, la bibliothèque JavaFX permet de concevoir ce type d'application en utilisant le pattern « MVVM » (Model-View-ViewModel).

ARCHITECTURE

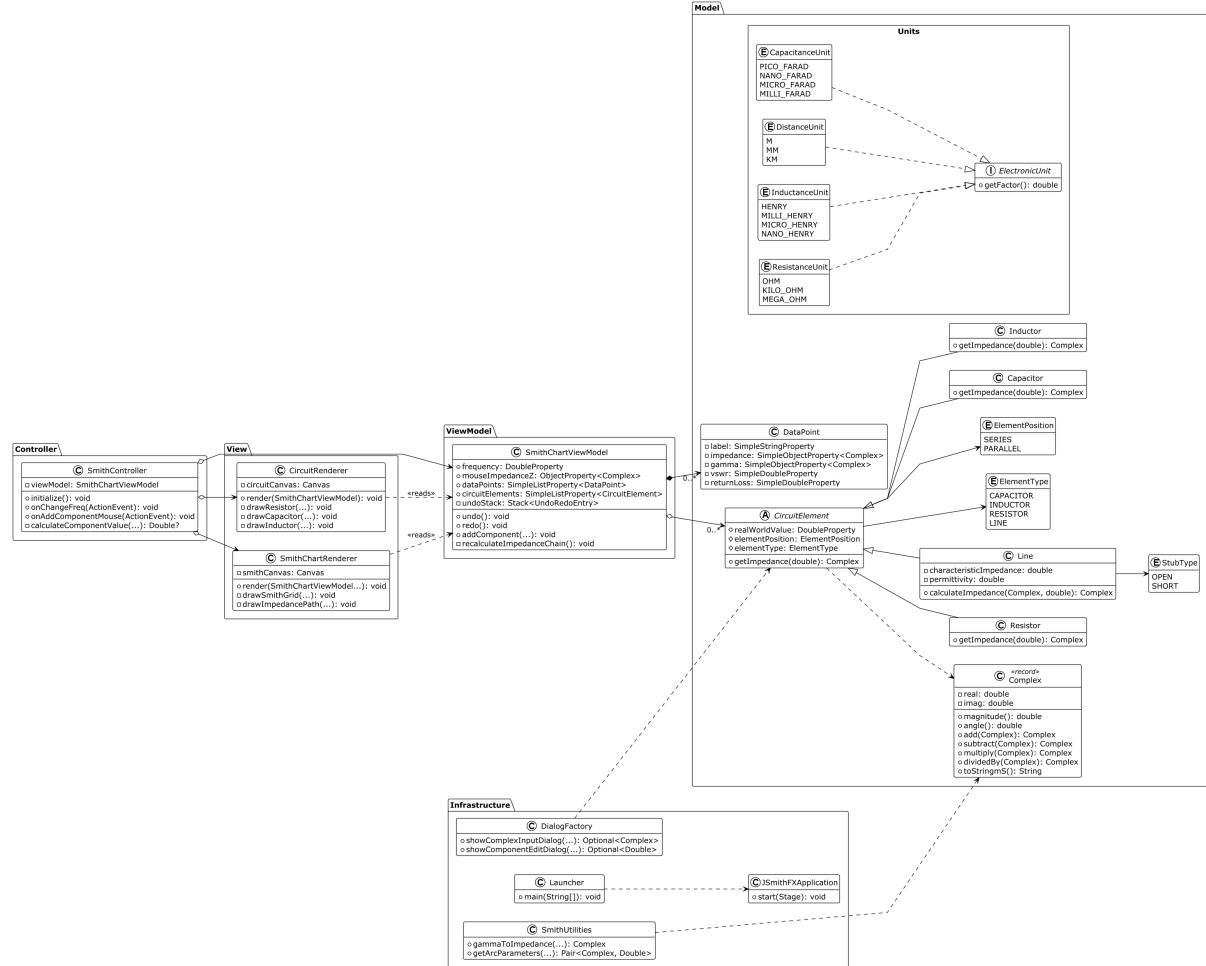


Fig. 11. – Schéma simplifié de l'architecture du projet

5.1 Partie Vue

La vue de l'application est, comme son nom l'indique, ce que l'utilisateur peut voir. C'est la partie émergée de l'iceberg. Dans une application JavaFX, elle se manifeste principalement par un fichier **.fxml**. C'est ce fichier qui est édité via l'outil Scene Builder pour modifier graphiquement l'interface utilisateur.

Ce fichier fxml est complété par une bibliothèque d'éléments d'interface utilisateur moderne AtlantaFX et d'une feuille CSS pour davantage personnaliser l'interface.

En plus de ce fichier qui gère les éléments simples, il a fallu mettre en place plusieurs classes spécifiques pour chaque élément, visibles sur le diagramme :

- SmithChartRenderer : Cette classe s'occupe de dessiner l'abaque de Smith sur un Canvas. Elle redessine la grille, les cercles VSWR et les tracés d'impédance à chaque changement de données.
- SmithChartLayout : Une classe utilitaire qui gère les conversions entre coordonnées gamma (complexe) et coordonnées pixel à l'écran. Elle centralise les calculs de positionnement sur le canvas.
- CircuitRenderer : De la même manière, cette classe dessine le schéma électrique du circuit en fonction des composants ajoutés (résistances, lignes, etc.).
- S1PPlotterWindow : Une fenêtre dédiée pour afficher les données d'un fichier S1P sous forme de diagramme cartésien (Log Magnitude / Fréquence).
- ChartPoint : Une classe utilitaire qui sert à gérer la détection de la souris sur les points du graphique (« hit detection ») pour afficher les infobulles (tooltips).
- Les Dialogues personnalisés : Plusieurs classes (comme SweepDialog ou ComplexInputDialog) ont été créées pour demander des informations complexes à l'utilisateur. Vu leur complexité, il était nécessaire de créer des classes supplémentaires pour les organiser.

Enfin, une classe utilitaire StageController a été ajoutée pour gérer le titre de la fenêtre principale, notamment pour indiquer visuellement à l'utilisateur si le projet contient des modifications non sauvegardées (ajout d'un astérisque * dans le titre).

5.1.1 MainController

La logique de l'interface est gérée par les contrôleurs. JavaFX fournit une possibilité de faire le lien entre les différents boutons et interactions avec le contrôleur principal de l'application, le MainController, qui fait le lien entre les boutons de l'interface et les fonctions du programme grâce à l'annotation @FXML. C'est cette classe que JavaFX consulte pour effectuer les actions lorsqu'on appuie sur des boutons.

Le MainController s'occupe principalement de :

- **Gestion des menus et actions utilisateur** : Importer/exporter des fichiers S1P, créer/sauvegarder des projets, gérer les circuits multiples, etc.
- **Binding avec le ViewModel** : Synchroniser les champs de texte (fréquence, impédance) avec les propriétés observables du ViewModel, mettre à jour les tables de données, etc.

- **Orchestration des dialogues** : Ouvrir les fenêtres de configuration (sweep, ajout de composants, édition de valeurs) et traiter leurs résultats.
- **Gestion de l'état de l'interface** : Afficher/masquer des sections selon le contexte (panneau S1P lorsqu'un fichier est chargé, panneau de sweep lors d'une simulation, etc.).
- **Coordination avec CircuitRenderer** : Redessiner le schéma électrique à chaque modification du circuit.

Cependant, vu la richesse des interactions possibles sur l'abaque (zoom, pan, ajout à la souris), le MainController devenait trop chargé. Une classe SmithChartInteractionController a donc été introduite pour déléguer toute la logique spécifique aux interactions avec l'abaque de Smith. Le MainController reste ainsi focalisé sur l'orchestration globale de la fenêtre et des menus.

5.1.2 SmithChartInteractionController

Dès qu'une action touchant à l'abaque est réalisée, le contrôleur appelle la fonction redrawSmithCanvas. Dans la version 1.0 de ce projet, cette fonction est sollicitée à de très nombreux endroits (plus de 30 fois). Elle demande ensuite au SmithChartRenderer de redessiner l'abaque entièrement. Le dessin complet est très rapide, donc le refaire entièrement n'est pas un problème (environ 0,05 ms lorsque l'abaque est vide et environ 0,1 ms lorsqu'il y a un fichier S1P chargé et que l'on change les composants, ce qui déplace tous les points sur l'abaque).

Pour éviter de surcharger le processeur avec des calculs inutiles (par exemple lors d'un redimensionnement rapide de la fenêtre ou d'un mouvement de souris), un mécanisme de **debouncing** a été mis en place. L'opération active un flag sur le ViewModel, nommé `isRedrawing`.

L'opération de dessin est ensuite déléguée au thread d'interface via la fonction JavaFX Platform.runLater. Si une nouvelle demande de dessin arrive alors que le flag est encore à true, elle est ignorée. Une fois le dessin terminé, le drapeau repasse à false. Cette optimisation permet d'éliminer énormément de demandes de dessin superflues et d'éviter des freezes de l'interface utilisateur. Après quelques tests simples d'utilisation, dépendamment de ce qui est effectué, cette fonction de debouncing permet d'éviter le dessin inutile d'environ 40 pourcents des demandes de dessin (entre 10 et 80 pourcents selon les actions).

C'est aussi ce contrôleur qui gère la fonctionnalité extrêmement importante d'ajout à la souris (`handleMouseMagnetization`). Lorsqu'un utilisateur ajoute un composant visuellement, le curseur ne se déplace pas librement, il doit se magnétiser au comportement souhaité du composant qu'on ajoute. Par exemple, un condensateur en parallèle va suivre un cercle de conductance constante et afficher en temps réel l'effet qu'a le composant sur le circuit d'adaptation. Le contrôleur calcule en temps réel la projection de la souris sur ces cercles mathématiques pour garantir que les valeurs affichées restent physiquement cohérentes.

Cette fonction complexe utilise de nombreuses fonctions de la classe SmithCalculator pour effectuer les calculs qui seront expliqués plus bas dans le rapport.

Le système projette le mouvement de la souris sur le vecteur tangent au cercle, convertit ce déplacement linéaire en changement d'angle, puis recalcule la valeur du composant à chaque mouvement. Pour les lignes de transmission, le système permet une rotation infinie autour du cercle, tandis que pour les autres composants, l'angle est borné entre le point de départ et les limites physiques (circuit ouvert ou court-circuit).

Au début du projet, le mécanisme de magnétisation de la souris déplaçait directement le curseur de l'utilisateur à l'aide de la classe Robot de JavaFX. Cependant, lors de tests sur une machine Linux, un problème critique est apparu. Pour des raisons de sécurité de la plateforme Wayland (source : <https://bugs.openjdk.org/browse/JDK-8307779>), le système d'exploitation empêchait le déplacement programmatique de la souris. Cette fonctionnalité était donc inutilisable sur certaines plateformes.

Pour résoudre ce problème, il a fallu mettre en place un système de souris virtuelle basé sur deux Canvas superposés. L'un est le smithCanvas, le canvas principal qui contient l'abaque de Smith, les tracés d'impédance et les points de données. Ce canvas n'est redessiné que lorsque les données changent. Ensuite la nouveauté, le cursorCanvas, c'est un canvas transparent superposé au premier, dédié aux éléments interactifs temporaires (curseur virtuel, tooltips).

Lors de l'ajout d'un composant à la souris, un curseur virtuel est affiché sur le cursorCanvas à la position magnétisée correcte. L'utilisateur voit ainsi où le composant sera placé sans que la souris système ne bouge. Sur les machines Windows, la souris est tout de même déplacée mais est rendue invisible pour que le curseur personnalisé soit mis en évidence.

Le cursorCanvas est également utilisé pour afficher les informations lorsque l'utilisateur survole un point de l'abaque. Le système de détection (ChartPoint) vérifie si la souris se trouve assez près d'un point et déclenche l'affichage d'une petite fenêtre avec les informations du point (impédance, VSWR, facteur de qualité, etc.).

5.1.3 Filtrage des fichiers S1P

Le logiciel permet de simplement mettre en évidence des parties du fichier S1P importé sur l'interface. Après des discussions sur l'utilité de ces mises en évidence, le système a été mis en place de façon à avoir trois plages de mise en évidence au maximum. Essayer de créer un circuit d'adaptation qui permet de faire trois plages est déjà très complexe, donc il était inutile d'en choisir plus.

Ces plages de fréquences sont définies par l'utilisateur via des contrôles de type RangeSlider dans l'interface. Chaque filtre peut être activé ou désactivé indépendamment via des propriétés

booléennes (filter1Enabled, filter2Enabled, filter3Enabled) stockées dans le ViewModel. Pour chaque filtre actif, on définit une fréquence minimale et maximale (freqRangeMinF1/freqRangeMaxF1, etc.).

Lors du dessin de l'abaque, le SmithChartRenderer parcourt tous les points S1P transformés et appelle la méthode whichFrequencyRange(frequency) du ViewModel pour déterminer dans quelle plage se trouve chaque point. Cette méthode retourne 1, 2 ou 3 si le point appartient à l'un des trois filtres actifs, ou -1 si aucun filtre ne correspond.

Selon le résultat, le point est dessiné avec une couleur et un style différent, si le point ne se trouve dans aucune de ces plages, il est alors mis en retrait avec une opacité moindre pour éviter d'avoir trop de points sur l'abaque, l'utilisateur peut se focaliser sur les fréquences sur lesquelles il travail.

5.2 Partie Modèle

Le modèle est le squelette des données de l'application, c'est la logique métier de l'application. Il contient les différents composants (CircuitElement), les unités, et la gestion des nombres complexes. C'est ici qu'on définit comment se comportent les éléments dans le plan physique.

5.2.1 Données et éléments du circuit

Le but de l'abaque de Smith est d'élaborer un circuit d'adaptation. Modéliser les différents composants est donc très important. Le logiciel permet d'ajouter quatre types de composants différents :

Les composants classiques (Résistance, Condensateur et Inducteur) : Ces composants se comportent de façon similaire, on peut facilement calculer leur impédance complexe à partir de la fréquence du circuit que l'on construit. Ensuite on choisit si on ajoute cette impédance en série (une simple addition) ou bien en parallèle (et on utilise la fonction de SmithCalculator appelée addParallelImpedance qui effectue ce calcul $Z_{Total} = \frac{1}{\left(\frac{1}{Z_{Start}} + \frac{1}{Z_{Comp}}\right)}$, Z Comp étant l'impédance du composant qu'on ajoute).

En plus de ça, un facteur de qualité (Q) a été mis en place pour les condensateurs et les inducteurs qui permet d'avoir des simulations de circuit d'adaptation plus réalistes. Ce facteur induit une résistance parasite :

- En série, on calcule une résistance de perte $R_s = \frac{|X|}{Q}$.
- En parallèle, on calcule une résistance de perte $R_p = |X| \cdot Q$.

Puisque la réactance (X) d'un condensateur est négative, nous utilisons sa valeur absolue pour garantir une résistance toujours positive. Selon la configuration choisie, cette résistance est combinée à la réactance pure pour former l'impédance réelle du composant.

Les lignes de transmission : Ces lignes se comportent très différemment des autres composants. Leur effet change drastiquement selon l'impédance d'entrée de la ligne, c'est pour cela que la manière de calculer l'impédance ne peut pas se faire de façon isolée. La classe Line implémente donc une méthode calculateImpedance(currentImpedance, frequency) qui prend en compte l'impédance actuelle du circuit, l'impédance au départ de la ligne.

Pour une ligne série (sans stub), on utilise la formule de transformation d'impédance [13] :

$$Z_{in} = Z_0 \frac{Z_L + Z_0 \tanh(\gamma l)}{Z_0 + Z_L \tanh(\gamma l)}$$

où $\gamma = \alpha + j\beta$ est la constante de propagation complexe (qui prend en compte les pertes via le facteur de qualité réutilisé comme perte en dB/m), et $\beta = \frac{2\pi f}{c} \sqrt{\epsilon_r}$ est la constante de phase.

Pour les stubs (court-circuit ou circuit ouvert), on travaille sur l'abaque des admittances. Un stub court-circuité donne $Y_{in} = \frac{Y_0}{\tanh(\gamma l)}$ tandis qu'un stub ouvert donne $Y_{in} = Y_0 \tanh(\gamma l)$. Cette admittance est ensuite ajoutée en parallèle au circuit.

Tous ces composants héritent de la classe abstraite CircuitElement qui définit les propriétés communes tel que la valeur réelle du composant, la position (série ou parallèle). Cette position permet aussi de savoir si une ligne va être un stub ou non. Le facteur de qualité est aussi mis dans cette classe abstraite.

5.2.2 Les unités

Chaque composant utilise des unités différentes. Pour faciliter la rédaction du code et pour avoir un code plus « propre », un système de gestion des unités a été mis en place. Toutes les unités implémentent une interface ElectronicUnit qui présente la fonction getFactor() permettant d'obtenir le facteur de conversion de l'unité choisie vers son unité de base (ex : les picofarads ont comme facteur 1E-12). Ensuite, différents énumérations implémentent cette interface :

- CapacitanceUnit : pF, nF, µF, mF (conversion vers les farads)
- InductanceUnit : H, mH, µH, nH (conversion vers les henrys)
- ResistanceUnit : Ω, kΩ, MΩ (conversion vers les ohms)
- DistanceUnit : mm, m, km (conversion vers les mètres, utilisé pour les lignes de transmission)
- FrequencyUnit : Hz, kHz, MHz, GHz (conversion vers les hertz)

5.2.3 Les points de données

Lorsqu'on ajoute un composant sur l'abaque, on affiche aussi ses informations dans une partie de l'interface nommées « DataPoint ». C'est une feature pour avoir accès directement aux informations de chaque point du circuit. La classe DataPoint encapsule toutes les informations

nécessaire, tel que la fréquence du point, son nom (le label affiché), son impédance complexe, son coefficient de réflexion, le VSWR (Voltage Standing Wave Ratio), la perte de retour (Return Loss) en dB et finalement le facteur de qualité de chaque point.

Ces datapoints sont aussi utilisés pour stocker les informations qui ne font pas partie du circuit, comme par exemple les informations liés aux sweeps ou bien aux fichiers S1P.

5.2.4 Gestion des fichiers S1P

La classe TouchstoneS1P est un parser développé pour cette application qui est capable de lire et d'exporter les fichiers .s1p selon le standard Touchstone. Ces fichiers commencent par une ligne d'options (précédée de #) qui spécifie :

- L'unité de fréquence (Hz, kHz, MHz, GHz)
- Le type de paramètre (S, Y, Z, H, G - par défaut S pour les paramètres de répartition)
- Le format des données (DB pour dB/angle, MA pour magnitude/angle, RI pour réel/imaginaire - par défaut MA)
- La résistance de référence (R suivi de la valeur, par défaut 50Ω)

La méthode statique parse(file) lit le fichier ligne par ligne. Elle commence par parser les options via parseOptionLine(), puis convertit chaque ligne de données en un DataPoint. Elle gère les différents formats de données (DB, MA, RI) via la méthode calculateComplexValue(), convertit ensuite les paramètres S en impédance grâce à calculateImpedance(), et finalement calcule le vrai gamma avec calculateGammaFromZ().

Cette classe permet aussi d'exporter les SWEEPS de fréquence du circuit crée sur l'application en fichier S1P. Les points du sweep sont alors exporté selon de format S MA R (qui sont les paramètres par défaut de ces fichiers).

5.2.5 Les calculs mathématiques

Vu que l'abaque de Smith est une projection du plan complexe, il a fallu mettre en place une classe qui gère ces nombres. La classe Complex est un record java qui représente un nombre complexe avec sa partie réelle et imaginaire. En plus de cette représentation, elle implémente toutes les opérations nécessaires pour le projet.

Ensuite, une grande partie des calculs mathématiques utilisés pour l'abaque de Smith se trouvent dans la classe SmithCalculator. On y trouve les fonctions de conversions d'impédance au coefficient de réflexion et inversement : gammaToImpedance(gamma, z0) qui calcule $Z = Z_0 \frac{1+\Gamma}{1-\Gamma}$ et impedanceToGamma(z, z0) qui calcule $\Gamma = \frac{z-z_0}{z+z_0}$. La classe fournit aussi des méthodes pour calculer le VSWR (Voltage Standing Wave Ratio) via la formule $\frac{1+|\Gamma|}{1-|\Gamma|}$ et le Return Loss en dB via $-20 \log_{10}(|\Gamma|)$ à partir du gamma.

Pour permettre à la vue de dessiner les trajectoires des composants lors de l'ajout à la souris, la méthode `getArcParameters()` détermine le centre et le rayon du cercle que doit suivre le composant sur l'abaque. Le comportement est très différent selon que le composant ajouté est une ligne de transmission ou un composant classique RLC.

Pour les lignes de transmission en série, on a deux cas de figure :

Si l'impédance caractéristique de la ligne ajoutée (Z_L) est la même que celle du système (Z_0), on obtient un centre situé à l'origine de l'abaque (0, 0) et le rayon correspond simplement à la magnitude du coefficient de réflexion du point de départ (cercle de VSWR constant).

Si les impédances caractéristiques diffèrent, il faut trouver le centre décalé du cercle. Pour cela, on calcule d'abord le coefficient de réflexion de l'impédance actuelle par rapport à l'impédance de la Ligne Z_L (et non par rapport à Z_0) : $\Gamma_{\text{rel}} = \frac{Z_{\text{actuelle}} - Z_L}{Z_{\text{actuelle}} + Z_L}$. On récupère sa magnitude $\rho_L = |\Gamma_{\text{rel}}|$ qui reste constante lors de la rotation autour de ce cercle.

Ensuite, on détermine les deux points extrêmes du cercle sur l'axe réel (les points de réactance nulle). Ces valeurs sont obtenues via la formule de conversion générale $Z = Z_L \frac{1+\Gamma}{1-\Gamma}$, simplifiée ici car on se situe sur l'axe réel :

- Pour l'impédance maximale (r_{\max}), le coefficient de réflexion est positif ($\Gamma = +\rho_L$), ce qui donne la formule : $r_{\max} = Z_L \frac{1+\rho_L}{1-\rho_L}$.
- Pour l'impédance minimale (r_{\min}), le coefficient de réflexion est négatif ($\Gamma = -\rho_L$), ce qui inverse les signes de l'équation : $r_{\min} = Z_L \frac{1+(-\rho_L)}{1-(-\rho_L)} = Z_L \frac{1-\rho_L}{1+\rho_L}$.

Ces deux impédances réelles doivent ensuite être converties dans le plan gamma du système Z_0 en utilisant la formule du coefficient de réflexion, cette fois ci avec Z_0 : $\Gamma_{\text{sys min}} = \frac{r_{\min} - Z_0}{r_{\min} + Z_0}$ et $\Gamma_{\text{sys max}} = \frac{r_{\max} - Z_0}{r_{\max} + Z_0}$.

Le centre du cercle se trouve exactement au milieu de ces deux points : $\text{centre}_x = \frac{\Gamma_{\text{sys min}} + \Gamma_{\text{sys max}}}{2}$ et le rayon vaut $r = \frac{|\Gamma_{\text{sys max}} - \Gamma_{\text{sys min}}|}{2}$. Ce centre est décalé horizontalement par rapport à l'origine, créant un cercle qui n'est plus centré sur l'abaque mais qui représente correctement la transformation d'impédance par la ligne.

Pour les lignes en parallèle c'est bien plus simple. Le cercle est simplement tangent au point $-1, 0$ (qui correspond au court circuit dans le plan des admittances) et au point de départ. Il faut alors trouver le cercle passant par ces deux points. On trouve le centre du cercle en trouvant le point qui est équidistant au court circuit de l'abaque et au Γ actuel

$$\text{centre}_x = \frac{|\Gamma|^2 - 1}{2(1 + \Gamma_{\text{réel}})}$$

où Γ est le coefficient de réflexion du point de départ. Le rayon est simplement la distance entre ce centre et le point $(-1, 0)$: $r = |\text{centre}_x - (-1)|$.

Pour les composants RLC classiques, le comportement est différent selon le type et la position :

- **Résistances** : Elles suivent des cercles de réactance constante (en série) ou de susceptance constante (en parallèle). Le centre en série est $(1, \frac{1}{x})$ avec rayon $|\frac{1}{x}|$, où x est la réactance normalisée. En parallèle, le centre est $(-1, -\frac{1}{b})$ avec rayon $|\frac{1}{b}|$, où b est la susceptance normalisée.
- **Condensateurs et inductances** : Ils suivent des cercles de résistance constante (en série) ou de conductance constante (en parallèle). Le centre en série est $(\frac{r}{r+1}, 0)$ avec rayon $\frac{1}{r+1}$, où r est la résistance normalisée. En parallèle, le centre est $(-\frac{g}{g+1}, 0)$ avec rayon $\frac{1}{g+1}$, où g est la conductance normalisée.

Ces équations sont les équations de base de l'abaque de smith.

Maintenant qu'on peut savoir sur quel cercle, on peut utiliser la fonction `getExpectedDirection(element, previousGamma)` qui calcule la direction (horaire ou anti-horaire) dans laquelle le composant doit se déplacer. C'est très important car par exemple, un condensateur en série tourne dans le sens horaire (réactance négative), une inductance en série dans le sens anti-horaire (réactance positive). Le cercle sur lequel le composant bouge est le même mais la direction change selon le composant.

Finalement il y a la fonction `calculateComponentValue(gamma, ...)` qui convertit une position obtenue de façon graphique (en ajoutant le composant à la souris) en valeur de composant. C'est extrêmement utile car c'est cette valeur qui va ensuite être utilisée pour ajouter le composant au circuit lorsque l'utilisateur va ajouter son composant.

Cette fonction prend en entrée le gamma final (là où la souris est positionnée), l'impédance de départ, le type de composant, sa position (série/parallèle), et d'autres paramètres selon le type d'élément. Elle calcule d'abord l'impédance finale à partir du gamma grâce à la formule $Z = Z_0 \frac{1+\Gamma}{1-\Gamma}$, puis détermine la valeur du composant selon sa nature.

Pour les composants RLC classiques, le calcul diffère selon la position :

- En série, on calcule l'impédance ajoutée $\Delta Z = Z_{\text{finale}} - Z_{\text{départ}}$. Pour une inductance, on utilise $L = \frac{\text{Im}(\Delta Z)}{\omega}$. Pour un condensateur, $C = -\frac{1}{\text{Im}(\Delta Z) \cdot \omega}$ (le signe négatif vient du fait que la réactance capacitive est négative). Pour une résistance, simplement $R = \text{Re}(\Delta Z)$.

- En parallèle, on travaille avec les admittances $Y = \frac{1}{Z}$. On calcule l'admittance ajoutée $\Delta Y = Y_{\text{finale}} - Y_{\text{départ}}$. Pour une inductance, $L = -\frac{1}{\text{Im}(\Delta Y) \cdot \omega}$. Pour un condensateur, $C = \frac{\text{Im}(\Delta Y)}{\omega}$. Pour une résistance, $R = \frac{1}{\text{Re}(\Delta Y)}$.

Pour les lignes de transmission, c'est plus complexe vu qu'il faut calculer la longueur physique de la ligne :

- Pour une ligne série (sans stub), on doit pouvoir savoir combien de fois on a fait le tour du cercle déterminé par `getArcParameters()` sur l'abaque. Initialement on calculait la formule de propagation de la réflexion ($\Gamma(L) = \Gamma(0) \cdot e^{-j2\beta L}$) pour pouvoir trouver l'angle de rotation et ensuite trouver la longueur de la ligne. Mais finalement, la partie graphique de l'application est au courant de l'angle de rotation vu qu'on utilise cette valeur pour pouvoir borner le mouvement de l'utilisateur. Il suffit alors de donner cette valeur à la fonction et de calculer la longueur de la ligne comme $L = \frac{\text{angle}}{2\beta}$. Si cette information n'est pas disponible (par exemple lors d'une modification manuelle), on recalcule l'angle à partir des gammes de départ et d'arrivée transformés dans le référentiel de la ligne (Z_L), puis on applique la même formule.
- Pour les stubs (court-circuit ou circuit ouvert), on travaille avec l'abaque d'admittance. Un stub court-circuité donne $Y_{\text{in}} = -j \frac{Y_0}{\tan(\beta L)}$, donc on résout $\tan(\beta L) = -\frac{Y_0}{B}$ où B est la susceptance cible, ce qui donne $L = \frac{\arctan\left(-\frac{Y_0}{B}\right)}{\beta}$. Pour un stub ouvert, $Y_{\text{in}} = jY_0 \tan(\beta L)$, donc $\tan(\beta L) = \frac{B}{Y_0}$ et $L = \frac{\arctan\left(\frac{B}{Y_0}\right)}{\beta}$. On s'assure ensuite que la longueur est positive en ajoutant des multiples de $\frac{\pi}{\beta}$ si nécessaire.

La fonction renvoie null si le calcul est impossible (division par zéro, valeur négative ou non-finie), garantissant ainsi que seules des valeurs physiquement réalistes sont retournées. Et si une telle valeur est retournée, l'opération d'ajout de composant est ignorée.

5.2.6 Gestion des « undo/redo »

Comme tout bon programme informatique, il est très utile de revenir sur ce qu'on a pu faire. Pour cela, deux systèmes ont été mis en place.

Un système générique `HistoryManager` qui permet d'avoir une gestion de deux piles génériques. Une pile de redo et une pile de undo. Lorsqu'on consomme une action undo, l'état actuel est sauvegardé dans la pile de redo, puis l'état précédent est restauré depuis la pile d'undo. Inversement, lorsqu'on fait un redo, l'état actuel est remis dans la pile d'undo et l'état suivant est récupéré depuis la pile de redo.

Ensuite pour pouvoir utiliser ce gestionnaire, un record `UndoRedoEntry` a été mis en place qui garde comme information l'opération effectuée (ADD, REMOVE, MODIFY), l'index du circuit modifié ainsi qu'une paire d'élément qui lie l'élément du circuit subissant l'opération et son

index dans le circuit choisi. Ensuite selon ce que l'utilisateur fait, chaque action est gardée dans le manager et l'utilisateur peut à choix revenir sur ce qu'il a fait! Le viewModel lui s'occupe de reconstruire le circuit à chaque undo/redo.

5.2.7 Les composants discrets

Dans le monde réel, les composants ne peuvent pas prendre n'importe quelle valeur. Ils existent en séries normalisées (séries E12, E24, E96, etc.) et il est pratique de pouvoir visualiser directement les valeurs des composants qu'on a sous la main, plutôt que des valeurs théoriques idéales.

L'utilisateur peut créer sa propre bibliothèque de composants disponibles via une fenêtre de dialogue (DiscreteComponentConfigDialog). Cette fenêtre permet d'ajouter des composants avec leur valeur exacte et leurs caractéristiques parasites, soit en utilisant le facteur de qualité, soit sous forme d'ESR. Dépendamment du constructeur du composant et de son type, les datasheets mettent à disposition soit l'un, soit l'autre (ou les deux).

Une nouvelle classe ComponentEntry a été mise en place pour encapsuler ces informations et pour pouvoir facilement les utiliser et aussi les exporter. L'utilisateur peut alors exporter ou importer des fichiers CSV contenant les informations nécessaires à cette bibliothèque de composants.

Ensuite, lorsque le mode « composants discrets » est activé par l'utilisateur, le comportement de l'ajout à la souris change. L'utilisateur peut toujours bouger sa souris comme il le souhaite, mais la valeur du composant calculé et son affichage se bloquent sur les valeurs des composants discrets en utilisant la méthode getClosestComponentEntry(). Cette méthode parcourt tous les composants du type sélectionné et trouve celui dont la valeur est la plus proche de la valeur calculée à la position actuelle de la souris. De plus, l'affichage de l'abaque rajoute des points directement sur l'abaque pour que l'utilisateur puisse voir où sont positionnés ces composants discrets.

5.2.8 Le management du projet

Pour pouvoir sauvegarder l'état du projet sur lequel on travaille, il fallait mettre en place un moyen de sérialiser les informations et de les exporter dans un fichier. Le choix a été fait de ne pas garder en mémoire les fichiers S1P importés lors de la sauvegarde, ni les sweeps effectués, ni la configuration des composants discrets. Chacun peut être exporté de son côté.

Les éléments sauvegardés, stockés dans un record agissant comme DTO (Data Transfer Object), sont le nom du projet, la fréquence de travail, l'impédance caractéristique du système (Z_0), l'impédance de charge, et surtout tous les circuits créés par l'utilisateur (stockés sous forme de liste de listes de CircuitElement).

La classe ProjectManager s'occupe de la sérialisation et désérialisation des projets. Elle utilise la bibliothèque Jackson pour convertir les données en format JSON avec le module Jdk8Module pour gérer correctement les types Java modernes (comme Optional).

Lors de la sauvegarde du projet via la méthode `saveProject()`, le logiciel vérifie si le projet est déjà associé à un fichier. Si non, un dialogue de sélection demande à l'utilisateur de choisir un emplacement où stocker le fichier (avec l'extension `.jsmfx`). Ensuite, le chemin du fichier est sauvegardé dans la classe et toutes les sauvegardes ultérieures se feront automatiquement sur ce même fichier. Une option « Save As » existe aussi pour pouvoir sauvegarder dans un nouveau fichier même si le projet est déjà lié à un fichier existant.

Lors du chargement via `loadProject()`, le fichier JSON est désérialisé et toutes les propriétés du ViewModel sont restaurées. Une fois le chargement terminé, les flags `hasBeenSaved` et `isModified` sont mis à jour pour refléter l'état du projet. Ces flags sont mis à jour à chaque interaction que l'utilisateur a avec l'abaque pour indiquer si le projet a eu des changements depuis la dernière sauvegarde.

Finalement, si des changements ont eu lieu mais qu'aucune sauvegarde n'a été effectué lorsque l'utilisateur souhaite quitter l'application, on demande à l'utilisateur s'il est sûr de vouloir quitter. Un comportement classique de ce genre d'application.

5.3 Partie ViewModel

Le ViewModel est la partie qui contient l'état de l'application, c'est la mémoire, mais aussi le cerveau. C'est lui qui s'occupe de mettre à jour les valeurs et de recalculer la chaîne d'impédance lorsqu'on modifie le circuit. La classe SmithChartViewModel agit comme l'unique source de vérité pour l'état de l'application et quasiment toutes les autres classes dépendent d'elle pour le comportement de l'abaque.

5.3.1 Les JavaFX Properties

JavaFX met à disposition des classes spéciales, les **JavaFX Properties**. Elles sont intrinsèquement observables, ce qui veut dire qu'on peut réagir dès qu'elles sont modifiées. Les différentes classes mettent en place des « listeners » qui déclenchent une interaction dès qu'une valeur change. Ou bien on peut aussi mettre en place des méthodes qui lient des éléments d'interfaces à des valeurs dans le viewModel.

Le ViewModel est constitué de plusieurs catégories de propriétés :

Les propriétés physiques du système : Ce sont les paramètres « principaux » de l'abaque de Smith. On trouve l'impédance caractéristique du système (z_0 , généralement 50Ω), l'impédance de charge au départ de la chaîne (`loadImpedance`) et la fréquence du système (`frequency`). Chacune

de ces propriétés est observée, et toute modification déclenche automatiquement un recalculation complet de la chaîne d'impédance.

Les collections de données : Le ViewModel gère plusieurs listes observables (`ObservableList`) qui stockent les différents types de points affichés sur l'abaque. Les `dataPoints` contiennent les points du circuit principal (l'impédance cumulée à chaque étape). Les `s1pDataPoints` stockent les données importées depuis un fichier S1P, et les `transformedS1PPoints` représentent ces mêmes points après transformation par les composants du circuit. Les `sweepDataPoints` contiennent les résultats d'un balayage en fréquence. Finalement, une liste `combinedDataPoints` agrège tous ces points pour faciliter le rendu graphique.

Les propriétés d'interaction utilisateur : Pour afficher les informations en temps réel lors du survol de l'abaque avec la souris, le ViewModel maintient des propriétés comme `mouseGamma`, `mouseImpedanceZ`, `mouseAdmittanceY`, `mouseVSWR`, `mouseReturnLoss` et `mouseQualityFactor`. Ces valeurs sont calculées instantanément lorsque la souris se déplace sur l'abaque et sont affichées dans l'interface via un binding.

Les propriétés d'état de l'application : Le ViewModel garde également la trace de l'état global, tel que le nom du projet (`projectName`), si le projet a été sauvegardé (`hasBeenSaved`), s'il a été modifié (`isModified`), ou encore si l'utilisateur est en train de modifier un composant (`isModifyingComponent`). Un flag `isRedrawing` permet d'éviter les redessins redondants via le mécanisme de debouncing expliqué précédemment.

5.3.2 Le recalculation des éléments de l'abaque

Dès que l'utilisateur modifie un paramètre du système (fréquence, impédance de charge, impédance caractéristique) ou ajoute, retire ou encore modifie un composant du circuit, un recalculation complet de la chaîne d'impédance est nécessaire. La méthode `recalculateImpedanceChain()` est alors appelée sur le ViewModel. Cette méthode délègue le calcul au service `CircuitSimulator`, qui se charge de la simulation proprement dite. J'ai essayé de développer une fonction qui recalcule la chaîne d'impédance de façon intelligente, mais ce calcul est si rapide qu'il n'est pas très utile d'essayer d'optimiser cela et de rajouter de la complexité programmatique. Le `CircuitSimulator` parcourt séquentiellement tous les éléments du circuit en partant de l'impédance de charge. Pour chaque élément, il calcule la transformation d'impédance via la méthode `propagateOne()`. Cette méthode prend la dernière impédance calculée et, selon le type de composant ajouté, calcule l'impédance qui suit en la rajoutant à l'impédance actuelle.

À chaque étape de la propagation, un nouveau `DataPoint` est créé et rajouté à la liste qui sera retournée au ViewModel à la fin de l'exécution de la fonction.

Ce même service `CircuitSimulator` est également utilisé pour la transformation des points S1P à travers le circuit d'adaptation (via `calculateTransformedS1P()`) et les balayages en fréquence (via `performSweep()`). La logique de propagation reste la même, seules les données d'entrée et la manière de les itérer diffèrent.

5.3.3 Gestion de plusieurs circuits

L'application permet de gérer plusieurs circuits pour le même système de base, c'est une fonctionnalité qui a été rajoutée sur la fin du projet. La manière dont elle a été conçue fait en sorte que le système en lui-même ne soit pas au courant qu'il travaille sur plusieurs circuits. Lorsque l'utilisateur le choisit, le ViewModel change simplement de circuit principal.

Le ViewModel gère une liste observable `allCircuits` qui contient plusieurs `ObservableList<CircuitElement>`, chaque liste représentant un circuit distinct. Une propriété `circuitElementIndex` indique quel circuit est actuellement actif. La propriété `circuitElements` est alors liée dynamiquement au circuit actif via un binding JavaFX `circuitElements.bind(Bindings.valueAt(allCircuits, circuitElementIndex))`.

Ce binding va alors changer dynamiquement la référence sur le circuit actif et, vu que le circuit change, le `viewModel` va lancer un calcul complet de tous les éléments. On sépare complètement la notion de plusieurs circuits du calcul et de l'affichage de l'abaque. La vue et tous les éléments liés aux calculs ne voient que le circuit actif.

5.3.4 Utilisation du pattern Singleton

Dans la logique de l'application, il ne doit y avoir qu'un seul ViewModel vu que c'est, comme dit plus tôt, l'unique source de vérité. Pour imposer cet aspect, l'utilisation d'un pattern Singleton était nécessaire. Il était aussi utile, dans certaines classes, de pouvoir simplement accéder à l'état de l'application avec un simple `SmithChartViewModel.getInstance()`, cela évite de devoir passer l'objet en référence directement.

On utilise aussi ce pattern pour la fenêtre du graphe MagLog pour les fichiers S1P.

ARCHITECTURE

6 État de l'application au rapport intermédiaire

Pour le moment, l'application n'est bien entendu pas terminée. Cependant, après avoir mis en place les différentes bibliothèques et le langage, j'ai pu développer une version initiale qui contient les éléments de base du fonctionnement d'un abaque de Smith.

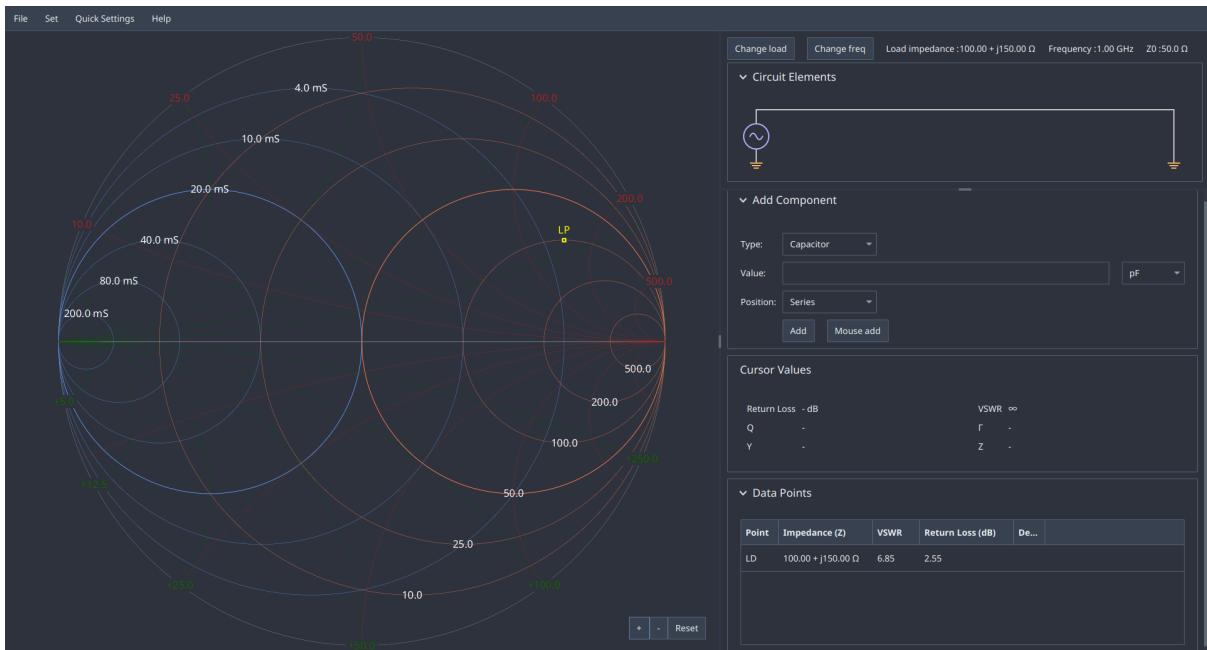


Fig. 12. – Vue d'ensemble de l'application

Actuellement, l'interface ne propose qu'un mode sombre, mais le choix entre mode clair et mode sombre sera intégré dans l'application finale.

Les fonctionnalités déjà mises en place sont :

- Ajout de composants RLC en série et en parallèle.
- Ajout de lignes de transmission et de stubs (ouverts et fermés).
- Visualisation graphique de l'abaque de Smith.
- Visualisation du schéma du circuit créé par l'ajout de composants.
- Calcul en temps réel des valeurs nécessaires à l'utilisation de l'abaque (VSWR, Return Loss, etc.).
- Ajout des composants soit en entrant les valeurs manuellement, soit en utilisant la souris (avec un système d'aimantation là où le curseur doit se trouver).

On obtient alors un logiciel utilisable, même s'il manque encore des fonctionnalités majeures nécessaires telles que le **Sweep** ou le **Tuning**.

6.1 Déploiement

Pour déployer l'application de façon multiplateforme, deux choix s'offrent à nous. Soit on construit une image .jar (un **UberJar** plus précisément, en utilisant l'outil Gradle ShadowJar, qui est tout simplement un jar qui contient toutes les dépendances nécessaires par le programme, obligatoire ici vu l'utilisation de JavaFX) qui permet de lancer l'application n'importe où, là où la JVM est installée. Soit on génère une application propre à chaque plateforme que l'utilisateur télécharge selon son système.

Le point fort de la seconde méthode est qu'on n'oblige pas l'utilisateur à avoir Java installé sur son appareil, mais on perd la portabilité du .jar.

Mon choix se porte alors sur un déploiement utilisant les outils de CI/CD pour proposer les deux solutions, la version portable en .jar (en indiquant le prérequis Java), et des exécutables natifs pour Windows, Linux et macOS.

6.2 Le ViewModel

On l'a vu dans la partie architecture, le ViewModel est le cerveau de l'application. Concrètement, la classe SmithChartViewModel agit comme la « source unique de vérité » (**Single Source of Truth**) pour l'état de l'application. Elle ne se contente pas de stocker des données, elle orchestre la logique de l'application et la synchronisation avec l'interface.

6.2.1 Le binding

Le ViewModel expose toutes les données nécessaires à la vue sous forme de Properties JavaFX (par exemple frequency, zo, circuitElements). Grâce au système d'observateurs de JavaFX, des **Listeners** sont attachés à ces propriétés dans le constructeur.

Dès qu'une valeur critique change (que ce soit la fréquence, l'impédance caractéristique ou l'ajout d'un composant), la méthode centrale `recalculateImpedanceChain()` est automatiquement déclenchée. Cela garantit que l'affichage est toujours cohérent avec les données, sans que la Vue n'ait besoin de demander explicitement une mise à jour.

6.2.2 La chaîne de calcul d'impédance

C'est le cœur de l'application. La méthode `recalculateImpedanceChain()`, comme son nom l'indique, calcule l'état du circuit actuel pour que la vue puisse ensuite afficher les éléments sur l'abaque de façon correcte.

On part de l'impédance de charge et on itère ensuite sur la liste des différents éléments qui constituent notre circuit. Pour chaque composant, on prend l'impédance calculée au composant précédent (ou bien si c'est le premier composant de la chaîne, on regarde l'impédance de la charge) et on calcule la nouvelle impédance en ajoutant le composant qu'on traite actuellement.

On fait cela jusqu'à ce que tous les éléments soient traités. Pour pouvoir avoir un affichage correct de toutes les étapes sur l'abaque, deux listes sont mises à jour, la liste `dataPoints` (pour le tableau de valeurs) et la liste `measuresGamma` (pour le dessin sur l'abaque).

6.2.3 L'interaction souris et l'historique

Le ViewModel gère également les interactions en temps réel comme le calcul des informations du point de l'abaque sur lequel la souris de l'utilisateur pointe grâce à la méthode `calculateMouseInformations`. Elle prend les coordonnées de la souris qui sont envoyées depuis la vue, et calcule l'impédance, l'admittance et toutes les valeurs liées à celle-ci pour que la vue les affiche. C'est une fonctionnalité clé de Smith.exe et il a été décidé de la remettre dans ce programme-ci.

Enfin, la classe intègre une gestion de l'historique (Undo/Redo) basée sur deux piles (Stack). Chaque ajout ou suppression de composant est enregistré comme une `UndoRedoEntry`, permettant à l'utilisateur d'annuler ou de rétablir ses actions. Concrètement, cela fonctionne comme lorsque l'utilisateur ajoute ou enlève un élément du circuit manuellement.

6.3 Dessin de l'abaque

Le premier challenge, après concevoir une interface plus ou moins agréable à utiliser, était de dessiner l'abaque de Smith. Tous les éléments sont disposés sur un Canvas de JavaFX qui me permet ensuite de récupérer le contexte graphique pour ensuite pouvoir dessiner mes traits par dessus.

Le rendu est géré par la classe `SmithChartRenderer` et se décompose en trois étapes successives, réexécutées à chaque redimensionnement de la fenêtre ou modification des données.

6.3.1 L'abaque lui-même

La méthode `drawSmithGrid` dessine les cercles de résistance et de conductance constantes, ainsi que les arcs de réactance et de susceptance. Pour ce faire, elle itère sur une liste de valeurs prédéfinies (0.2, 0.5, 1.0, 2.0, 4.0, 10.0), ce sont les mêmes valeurs prédéfinies que propose Smith.exe par ailleurs.

Une astuce technique utilisée ici est l'utilisation du clipping (`gc.clip()`). Plutôt que de calculer mathématiquement les intersections exactes des arcs avec le cercle extérieur, on dessine des cercles complets, et on demande au contexte graphique de masquer tout ce qui dépasse du rayon principal de l'abaque.

6.3.2 Les points d'impédance

La méthode `drawImpedancePoints` récupère la liste des coefficients de réflexion (Γ) calculés par le ViewModel. Comme le plan complexe de l'abaque de Smith correspond directement aux coordonnées cartésiennes du coefficient de réflexion, la projection sur l'écran est directe, la partie réelle de Γ devient la coordonnée X et la partie imaginaire la coordonnée Y, le tout mis à l'échelle par le rayon du cercle.

6.3.3 Le tracé du circuit d'impédance

Après avoir dessiné les points d'impédance de chaque élément constituant le circuit, il reste la partie la plus complexe, relier ces points entre eux en respectant la physique de l'abaque. Ce chemin est généré par la fonction `drawImpedancePath`.

Contrairement à un graphique classique, on ne relie pas les points par des lignes droites. Le déplacement d'un point à un autre, suite à l'ajout d'un composant, suit toujours une trajectoire courbe spécifique (cercle de résistance constante pour une réactance série par exemple).

Pour dessiner cela, l'algorithme calcule d'abord le centre et le rayon du cercle de mouvement. Ensuite, il détermine l'angle de départ et l'angle d'arrivée. Enfin, il vérifie dans quelle direction tracer l'arc (horaire ou anti-horaire) selon la nature du composant (resistance, condensateur, etc.) et sa position dans le circuit.

7 Conclusion intermédiaire

Arrivé à ce stade, je considère que le projet a bien démarré. Les bases de l'application sont posées et l'architecture actuelle semble solide pour supporter la suite du développement. L'objectif principal de cette première phase, qui était d'avoir un prototype capable d'afficher l'abaque et de gérer les interactions de base, est atteint.

Le choix de Java et de JavaFX m'a permis de rapidement mettre en place un environnement de travail efficace. Comme c'est une technologie que je maîtrise, j'ai pu me concentrer sur la logique de l'abaque. Cela valide également l'approche MVVM choisie et très fortement suggérée par JavaFX, qui sépare bien les calculs mathématiques de l'interface graphique, rendant le code bien clair et structuré.

Pour la suite, je suis confiant car je suis légèrement en avance sur la planification initiale. Le but maintenant est de me concentrer sur les fonctionnalités plus complexes comme le Sweep et le Tuning.

CONCLUSION INTERMÉDIAIRE

Bibliographie

- [1] Ae6ty. « SimSmith (SimNec) ». Disponible sur : https://www.ae6ty.com/smith_charts/
- [2] Coppens J. « linSmith ». Disponible sur : https://jcoppens.com/soft/linsmith/index_en.php
- [3] Frith L. « Phillip Smith: From Amateur Radio Operator to Creator of the Smith Chart ». Disponible sur : <https://www.allaboutcircuits.com/news/phillip-smith-from-amateur-radio-operator-to-creator-of-the-smith-chart/>
- [4] Rhea R. « Smith Chart History ». Disponible sur : <http://smithchart.org/phsmith.shtml>
- [5] TME. « Impédance - comment elle est calculée et pourquoi est-elle importante ? ». Disponible sur : <https://www.tme.eu/ch/fr/news/library-articles/page/57276/impedance-comment-elle-est-calculee-et-pourquoi-est-elle-importante-/>
- [6] Science Sanctuary. « Standing Wave Diagram ». Disponible sur : <https://sciencesanctuary.com/>
- [7] Rohde & Schwarz. « Understanding the Smith Chart ». Disponible sur : <https://youtu.be/rUDMo7hwihS?si=ffJWC7NbdH4x1xPe>
- [8] Sbyrnes321. « Smith Chart Explanation (Image) ». Disponible sur : https://en.wikipedia.org/wiki/Smith_chart
- [9] Rabasté D. *L'abaque de Smith.*
- [10] Dellspurger F. « Smith Chart V4.1 ». Disponible sur : <https://www.fritz.dellsperger.net/smith.html>
- [11] Palexdev. « MaterialFX ». Disponible sur : <https://github.com/palexdev/MaterialFX>
- [12] Paz M. K. « AtlantaFX ». Disponible sur : <https://github.com/mkpaz/atlantafx>
- [13] Pierre Favrat B. H. *Physique de la transmission - Milieux câblés.* 2025.

BIBLIOGRAPHIE

- [14] « Explication de l'abaque de Smith (Vidéo) ». Disponible sur : <https://www.youtube.com/watch?v=EoSJ1M%E2%80%93npg>
- [15] W2AEW. « Basics of the Smith Chart - Intro, impedance, VSWR, transmission lines, matching ». Disponible sur : <https://www.youtube.com/watch?v=TsXd6GktIYQ>
- [16] Tedjini S. *Abaque de Smith*. 2015.

Figures

Fig. 1	L'abaque de Smith tel que présenté dans le journal « Electronics »	18
Fig. 2	On adapte la partie réactive ($+jX$ et $-jX$) et maximise le transfert de puissance .	20
Fig. 3	Phénomène des ondes stationnaires [6]	21
Fig. 4	Explication graphique de l'abaque [8]	22
Fig. 5	Représentation des cercles de résistance constante sur l'abaque de Smith	23
Fig. 6	Représentation des courbes de réactance constante	23
Fig. 7	Impédance de $75 + j50$ Ohm	24
Fig. 8	Impédance avec circuit d'adaptation ⁵	24
Fig. 9	Effet d'ajouts d'inductances et de capacités en série et parallèle [9]	25
Fig. 10	Aperçu de Smith.exe	28
Fig. 11	Schéma simplifié de l'architecture du projet	38
Fig. 12	Vue d'ensemble de l'application	53

⁵Source : Logiciel Smith.exe

FIGURES

Outils utilisés

Lors de la conception de ce travail de Bachelor, différents outils sont utilisés pour la rédaction du rapport et la conception du code.

Programmation

L'IDE utilisé est IntelliJ. L'application étant développée en Java, IntelliJ de JetBrains est l'environnement que je maîtrise le mieux, ce choix était donc logique. Les technologies principales s'articulent autour de JavaFX version 25, qui nécessite JDK 23, pour la gestion de l'interface graphique. C'est le squelette du projet, l'ensemble de l'application repose sur cette bibliothèque, que ce soit pour le dessin de l'abaque ou pour la logique de l'interface utilisateur.

Un outil intéressant, supporté nativement dans IntelliJ, est JavaFX Scene Builder. À la manière de QtCreator, il permet de s'aider d'une interface visuelle pour créer les vues de l'application.

AtlantaFX [12] est une bibliothèque qui permet d'ajouter un thème moderne à l'application JavaFX. Cela évite de devoir créer des composants personnalisés pour les éléments simples et me permet de me focaliser sur l'écriture du code.

Ensuite, pour compiler l'application, j'utilise Gradle. C'est un concurrent à Maven qui est, d'après mon expérience, plus performant et plus stable.

Conception

Le diagramme UML, réalisé en parallèle de l'écriture du code, est conçu avec l'outil PlantUML. Ce choix permet une approche programmatique de la modélisation, facilitant la mise à jour du diagramme au fur et à mesure de l'avancement du développement.

Gestion de versions

Le projet étant individuel, un simple dépôt Git a été mis en place sur GitHub pour jalonner chaque étape. J'ai décidé de ne pas complexifier le processus avec la gestion d'issues pour le moment.

Le projet est organisé en deux dépôts, un pour l'application (le code source), et un second pour la partie académique (le rapport de TB).

Rapport

L'entièreté du rapport est rédigée sur Typst, à l'aide de l'extension Tinymist Typst⁶ sur Visual Studio Code. Ensuite, pour la correction d'éventuelles fautes d'orthographe, le logiciel Antidote⁷ a été utilisé.

Ensuite le template Typst utilisé est un projet GitHub⁸ créé par Christophe Roulin, un étudiant de l'HEIG.

Utilisation de LLMs

Différents LLMs (Gemini et GPT majoritairement) ont été utilisés pour aider à la reformulation de certaines parties du rapport. L'outil GitHub Copilot a également été utilisé comme aide à la conception du code.

⁶Lien de l'extension : <https://marketplace.visualstudio.com/items?itemName=myriad-dreamin.tinymist>

⁷Lien du site : <https://www.antidote.info/fr/>

⁸Lien du repos GitHub : <https://github.com/DACC4/HEIG-VD-typst-template-for-TB>