Étude de la rigidité dynamique spectrale pour les billards planaires analytiques

S8IP

Romain Hoang

15 août 2025

Abstract : Ce rapport traite de la dynamique de billard sur les sous-ensembles $\mathbb{Z}_2$-symétriques, strictement convexes et à bord analytique. L'objectif est d'explorer, à l'aide de simulations numériques, les limites d'ensembles génériques qui sont dynamiquement spectralement rigides. On notera que les domaines ont tendance à sortir du cadre analytique. On donne également des outils permettant d'étudier la rigidité pour des domaines C^r.

Superviseur 1 :
Nom : Jacopo de Simoi
Email : jacopods@math.utoronto.ca
Adresse : Department of Mathematics, University of Toronto
40 St George Street, Toronto, ON
Canada M5S 2E4

Superviseur 2 :
Nom : Philippe Mathieu
Email : philippe.mathieu@centralesupelec.fr
Adresse : 9 Rue Joliot Curie
Bâtiment Bouygues, 91190
Gif-sur-Yvette

Remerciements

Je tiens à remercier Jacopo de Simoi sans qui ce stage n'aurait pas pu voir le jour. J'ai vraiment été guidé tout le long de ce projet, et sans l'aide et le temps de Jacopo je n'aurais pu mener à bien ce stage. Je remercie également University of Toronto qui m'a accepté en tant que visiting research student et qui m'a offert les infrastructures nécessaires. Je remercie également mon école, CentraleSupélec, qui m'a permis de partir faire ce projet. Enfin, je remercie mon superviseur de CentraleSupelec, Philippe Mathieu, qui a pris de son temps pour superviser mon projet, m'appeler, et évaluer mon travail.

Préambule

Dans le cadre de mes études à CentraleSupélec, j'ai pu faire un stage de recherche à l'étranger, en l'occurrence à Toronto au Canada. C'était un stage de 21 semaines, au sein de University of Toronto, Department of Mathematics. Ce travail en constitue le rapport, afin de décrire ce qui a été effectué. Ce document fait donc état des lieux de mon humble introduction à la dynamique de billard : contrairement à un article de recherche, certaines notions sont introduites ici sans être (explicitement, ou non) réutilisées par la suite. Certaines sections peuvent être trop détaillées et d'autres pas assez, néanmoins le document doit être capable de donner les connaissances suffisantes pour comprendre le travail effectué. Ce rapport s'accompagne d'une présentation oral. Une première partie de ce document donne les outils théoriques, la deuxième partie quant à elle explique comment est construite l'étude numérique. Enfin, les codes en annexe n'ont pas forcement vocation à être lus.

Table des matières

Table des figures

Liste des codes

Première partie
Étude Théorique

1 Le sujet d'étude

1.1 Motivations

Dans ce stage de recherche, on s'interesse à la dynamique de billard. Donnons une légère motivation au sujet d'étude. Le problème étudié est motivé par la célèbre question de M. Kac [1] "Peut-on entendre la forme d'un tambour ?" Plus formellement, si on se donne $\Omega \subset \mathbb{R}^2$ un domaine planaire et on note $\mathrm{Sp}(\Omega) = \{0 < \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_k \leq \cdots\}$ le *Spectre de Laplace* de Ω pour certaines conditions au bords (par exemple de Dirichlet). Autrement, $\mathrm{Sp}(\Omega)$ est l'ensemble avec multiplicités des valeurs positives réel λ qui satisfont le problème de valeur propre

$$\Delta u + \lambda^2 u = 0, \qquad u = 0 \text{ sur } \partial\Omega.$$

Soit une classe $\mathcal{M}$ de domaines, et un domaine $\Omega \in \mathcal{M}$, on dit que Ω est *spectralement déterminé dans* $\mathcal{M}$ si il est l'unique élément (modulo isometries) de $\mathcal{M}$ avec son Spectre de Laplace : si $\Omega, \Omega' \in \mathcal{M}$ sont *isospectrales* i.e. $\mathrm{Sp}(\Omega') = \mathrm{Sp}(\Omega)$, alors Ω' est l'image de Ω par une isométrie (i.e. une composition d'une translation et d'une rotation). La question de Kac peut alors être formulée comme suit, supposons que l'on se donne une classe de domaines $\mathcal{M}$:

Problème Inverse du Spectre *Est-ce que tous les* $\Omega \in \mathcal{M}$ *sont spectralement déterminés ?*

Si $\mathcal{M}$ est l'ensemble de tous les domaines planaires, alors la réponse est bien connu pour être négative (par exemple [2]).
Par contre, tous les exemples de domaines qui ne sont pas spectralement déterminés ne sont pas convexes, en plus ils sont bornés par des courbes analytiques par morceaux. D'autre part, Zelditch a prouvé dans [3] que le problème inverse du spectre possède une réponse positive quand $\mathcal{M}$ est une classe générique de domaines analytiques $\mathbb{Z}_2$-symétriques convexes (i.e. symétriques par rapport à une réflexion selon un axe).

Le problème pour les domaines non-analytiques est substantiellement plus dur. Pour les domaines C^∞, Osgood–Phillips–Sarnak [4, 5, 6] ont montré que les ensembles isospectraux sont nécessairement compact dans la topologie C^∞. Sarnak [7] a aussi conjecturé qu'un ensemble de domaines isospectraux est constitué de domaines isolés (toujours au sens de la topologie C^∞). En d'autres mots, C^∞-proche d'un domaine C^∞, il doit n'y avoir aucun domaine isospectral, sauf ceux qui peuvent être obtenus par une isométrie. Une version plus faible de cette conjecture peut être exprimée via la notion de rigidité :

Un domaine Ω est dit *spectralement rigide dans* $\mathcal{M}$ si pour toute famille isospectrale à 1-paramètre C^1-lisse $(\Omega_\tau)_{|\tau|\leq 1} \subset \mathcal{M}$ avec $\Omega_0 = \Omega$ est nécessairement une famille isométrique. On peut alors se demander : "Est-ce que tous les domaines C^∞ sont

spectralement rigides ?"

Le problème de rigidité spectrale est en principe bien plus simple que le problème inverse du spectre, pourtant il s'avère qu'il est extrêmement difficile. Hezari–Zelditch (voir [8]) ont apporté un résultat qui va dans le sens affirmative.

1.2 Le Spectre des Longueurs et sa relation avec le Spectre de Laplace.

Il y a une relation remarquable entre le Spectre de Laplace et un objet défini dynamiquement, que l'on va maintenant définir. Le *Spectre des Longueurs* de Ω est défini comme l'ensemble

$$\mathcal{L}(\Omega) = \mathbb{N}\{\text{longueurs de toutes les géodésiques fermées de } \Omega\} \cup \mathbb{N}\{\ell_{\partial\Omega}\},$$

où $\ell_{\partial\Omega}$ correspond à la longueur du bord $\partial\Omega$ et $\mathbb{N} = \{1, 2, \cdots\}$. Par *géodésiques fermées* Ω, on entend une trajectoire périodique du flot de billard (i.e. flot geodésique à l'intérieur de Ω avec réflexions optiques sur $\partial\Omega$).

Andersson–Melrose (voir [9, Theorem (0.5)]), a montré que pour des domaines C^∞ *strictement convexes* (i.e, un domaine avec un rayon de courbure ρ fini et strictement positif), on a la relation qui suit entre le support singulier de la trace d'onde et le Spectre des Longueurs :

$$\text{supp sing}\left(t \mapsto \sum_{\lambda_j \in \text{Sp}(\Omega)} \exp(i\lambda_j t)\right) \subset \pm\mathcal{L}(\Omega) \cup \{0\}. \tag{1}$$

En fait, la relation marche toujours pour des domaines C^∞ non-convexes (voir [10, Theorem 5.4.6]). De plus, sous des conditions génériques (voir Remarque 2.10 pour plus de détails) on peut montrer que cette inclusion est en réalité une égalité et donc le Spectre de Laplace determine le Spectre des Longueurs. Il est alors naturel de se poser les mêmes questions dans un contexte dynamique.

On dit que Ω est *dynamiquement spectralement déterminé dans* $\mathcal{M}$ s'il est l'unique élément (modulo isometries) de $\mathcal{M}$ avec son Spectre des Longueurs.

Problème inverse dynamique *Est-ce que tous les* $\Omega \in \mathcal{M}$ *sont dynamiquement spectralement déterminés ?*

Tous les contre-exemples du problème inverse du spectre mentionné précédemment constituent également des contre-exemples pour le problème inverse dynamique. De même, à ce jour il n'y a pas de contre-exemples réalisés par des domaines convexes. De plus, le résultat mentionné du desssus par Zelditch (dans [3]) tient toujours dans le contexte dynamique, on va en reparler. Dans le cas d'un domaine convexe suffisament lisse, le problème est ouvert et présente les mêmes défis que le problème du spectre inverse.

Definition 1. *Soit une classe de domaines $\mathcal{M}$. On dit que $(\Omega_\tau)_{|\tau|\leq 1}$ est une famille à 1-paramètre C^1-lisse de domaines dans $\mathcal{M}$ si $\Omega_\tau \in \mathcal{M}$ pour tout $|\tau| \leq 1$ et il existe*

$$\gamma\colon [0,1] \times \mathbb{T} \to \mathbb{R}$$

tel que pour tout $\xi \in \mathbb{T}$, $\gamma(\cdot, \xi) \in C^1([0,1])$ et, pour tout $\tau \in [-1,1]$, $\gamma(\tau, \cdot)$ est un C^{r+1} difféomorphisme de $\mathbb{T}^1$ sur $\partial\Omega$. La fonction γ est appelé paramétrisation de la famille.

Definition 2. *Une famille $(\Omega_\tau)_{|\tau|\leq 1}$ est dite dynamiquement isospectrale si $\mathcal{L}(\Omega_\tau) = \mathcal{L}(\Omega_{\tau\prime})$ pour tout $\tau, \tau\prime \in [-1,1]$*

Définissons maintenant la notion dynamique correspondant à celle de rigidité spectrale.

Definition 3. *On dit qu'un domaine $\Omega \in \mathcal{M}$ est dynamiquement spectralement rigide dans $\mathcal{M}$ si pour toute famille à 1-paramètre C^1-lisse dynamiquement isospectral $(\Omega_\tau)_{|\tau|\leq 1} \subset \mathcal{M}$ telle que $\Omega_0 = \Omega$ est nécessairement une famille isométrique.*

Dans la suite, quand on parle de famille $(\Omega_\tau)_{|\tau|\leq 1}$, on parle de famille à 1-paramètre C^1-lisse.

2 Dans quelle classe de domaines travailler ?

On note $\mathcal{S}^r$ la classe de domaines strictement convexes à bordure C^{r+1}, $\mathbb{Z}_2$-symétriques. De même, on note $\mathcal{S}^\omega$ la classe de domaines strictement convexes à bordure analytique, $\mathbb{Z}_2$-symétriques.

Jacopo de Simoi, Vadim Kaloshin, Qiaoling Wei ont montré dans [11], que si on se donne la classe $\mathcal{S}^8_\delta$ de domaines de $\mathcal{S}^8$ qui sont δ-proches du cercle, alors tout $\Omega_0 \in \mathcal{S}^8_\delta$ est *dynamiquement spectralement rigide dans* $\mathcal{S}^8_\delta$.

On peut alors se demander si ce résultat est toujours vrai lorsque l'on omet l'hypothèse de proximité du cercle. Peut-on calculer numériquement un domaine Ω qui ne soit pas dynamiquement spectralement rigide dans $\mathcal{S}^8$?
Nous verrons qu'en méthodes numériques, le caractère C^9 peut poser problème.

Pour essayer de construire un tel ensemble, on va se baser sur l'approche utilisée dans [11]. L'objectif est de comprendre comment montrer qu'un domaine est dynamiquement spectralement rigide (DSR), et de mettre en défaut cet aspect avec un contre-exemple Ω créé numériquement. En cas d'échec, ceci ne prouve évidemment pas qu'un tel ensemble n'est pas DSR. En effet, il serait possible de montrer que Ω ne soit pas DSR d'une autre manière, encore inconnue.

Comme mentionné plus haut, le résultat mentionné par Zelditch dans [3] tient toujours dans le contexte dynamique. Alors, pour tout Ω **générique** dans $\mathcal{S}^\omega$, Ω est dynamiquement spectralement déterminé dans $\mathcal{M}$ et donc dynamiquement spectralement rigide dans $\mathcal{S}^\omega$.

2.1 La notion de généricité

Dans ce contexte, « générique » signifie que la propriété est vraie sur une intersection dénombrable d'ouverts denses dans l'espace fonctionnel des domaines analytiques considérés.
Pour qu'un domaine soit générique (au sens du théorème), il doit satisfaire deux conditions dynamiques essentielles :

(1) **Longueurs distinctes à période fixée** : Pour chaque période q, toutes les orbites périodiques de période q doivent avoir des *longueurs différentes*. Cela garantit que le spectre des longueurs est « simple » et non dégénéré.

(2) **Non-dégénérescence des orbites périodiques** : Les orbites doivent être *isolées et stables*. Plus précisément, la matrice de monodromie (Poincaré) associée à une orbite périodique ne doit pas avoir la valeur propre 1. Cela empêche l'existence de familles continues d'orbites (ce qui diluerait leur contribution dans la trace des ondes).

Si l'une de ces deux conditions échoue, le domaine est alors dit *non générique*. Le théorème de Zelditch ne s'applique plus, ce qui signifie que la détermination spectrale n'est plus garantie — mais elle peut exister pour d'autres raisons.

2.2 Le cas particulier du cercle

Le cercle est un exemple central pour illustrer la notion de non-généricité :

(1) Il admet une infinité d'orbites périodiques de même période et de même longueur (triangles équilatéraux, carrés, etc.), ce qui viole la condition (1).

(2) Ces orbites peuvent tourner librement : elles forment des familles continues, donc ne sont pas isolées — ce qui viole la condition (2).

Ainsi, bien que le cercle soit analytique, strictement convexe et $\mathbb{Z}_2$-symétrique, il est hors du cadre générique du théorème de [3]. Néanmoins par [11] on sait que le cercle est dynamiquement spectralement rigide.

2.3 La normalisation

Il est naturel d'introduire une notion de normalisation, nos domaines étant $\mathbb{Z}_2$-symétriques, on choisit arbitrairement un axe de symétrie qu'on appelle l'axe de symétrie. Puisque le domaine est convexe, son axe de symétrie intersecte $\partial\Omega$ en deux points. On nomme un de ces deux points le point marqué de $\partial\Omega$, l'autre point sera appelé le point auxiliaire de $\partial\Omega$. À partir de maintenant, à chaque fois que l'on considère un domaine Ω, on assume qu'un choix a été fait pour l'axe de symétrie et pour le point marqué. Observons qu'une fois que ces choix ont été faits pour Ω_0, alors par continuité, ils sont déterminés de manière non-ambigüe pour chaque élément de la famille $(\Omega_\tau)_{|\tau|\leq 1}$. De plus, on suppose aussi que la paramétrisation γ de la famille $(\Omega_\tau)_{|\tau|\leq 1}$ est telle que $\gamma(\tau, 0)$ est le point marqué de $\partial\Omega$.

Definition 4. *Pour une paramétrisation γ d'une famille $(\Omega_\tau)_{|\tau|\leq 1}$, on définit la fonction de déformation infinitésimale :*

$$n_\gamma(\tau, \xi) = \langle \partial_\tau \gamma(\tau, \xi), N_\gamma(\tau, \xi) \rangle$$

où $\langle \cdot, \cdot \rangle$ est le produit scalaire usuel dans $\mathbb{R}^2$ et $N_\gamma(\tau, \xi)$ est le vecteur normal unitaire sortant de $\partial\Omega$. Observons que n est continue en τ et $n(\tau, \cdot) \in C^r(\mathbb{T}^1, \mathbb{R})$ pour tout $\tau \in [-1, 1]$. Par la condition de normalisation de $(\Omega_\tau)_{|\tau|\leq 1}$, on conclut que $n_\gamma(\tau, \cdot)$ est une fonction paire, ie $n_\gamma(\tau, \xi) = n_\gamma(\tau, -\xi)$, et $n_\gamma(\tau, 0) = 0$ pour tout $\tau \in [-1, 1]$

Lemme 1. *Soit $(\Omega_\tau)_{|\tau|\leq 1}$ une famille de domaines dans $\mathcal{S}^r$ et soit $\gamma(\tau, \xi)$ une paramétrisation de la famille. Alors,*

(a) Pour tout autre paramétrisation de la famille $\tilde{\gamma}(\tau, \tilde{\xi})$, et $\tau \in [-1, 1]$. Si $n_\gamma(\tau, \cdot)$ est identiquement nulle, alors $n_{\tilde{\gamma}}(\tau, \cdot)$ est également identiquement nulle.

(b) $n_\gamma(\tau, \xi) = 0$ pour tout $(\tau, \xi) \in [-1, 1] \times \mathbb{T}^1$ si et seulement si $(\Omega_\tau)_{|\tau|\leq 1}$ est une famille constante.

Démonstration. cf [11]. □

3 Formulation fonctionnelle de la rigidité spectrale

Dans cette section, il s'agit de créer un opérateur $\mathscr{T}_\Omega$. On ne va pas prouver ni expliciter les détails de sa création, simplement donner les idées pour comprendre ce qu'est cet opérateur. Le lecteur interessé peut se référer à [11] pour avoir tous les détails.

3.1 La dynamique de billard

Soit $\Omega \in \mathcal{S}^r$, on fixe son périmètre égal à 1. On rappelle que s dénote l'abcisse curviligne et qu'on a par convention $s = 0$ au point marqué, et alors donc $s = 1/2$ au point auxiliaire. On considère une dynamique de billard sur Ω, décrite comme, une particule ponctuelle se déplacant à vitesse constante à l'interieur de Ω. Quand la particule percute $\partial\Omega$, elle rebondit selon les lois de l'optique : angle d'incidence = angle de réflection. Il est de coûtume d'étudier la dynamique de billard en passant à une version en temps discret de celle-ci, i.e. à une fonction sur la section canonique de Poincaré $M = \partial\Omega \times [-1, 1]$. La première coordonnée (paramétrisée par $\gamma(s)$) identifie le point auquel la particule a percuté $\partial\Omega$ et la seconde coordonnée y est égal à $cos(\varphi)$, où φ est l'angle entre la trajectoire sortante en s et la tangente à $\partial\Omega$ orientée positivement. La fonction de la balle de billard sur M est alors définie comme

$$\begin{aligned} f \colon \partial\Omega \times [-1, 1] &\to \partial\Omega \times [-1, 1] \\ (s, y) &\mapsto (s', y') \end{aligned}$$

où s' est la coordonnée du point auquel la trajectoire sortante de s avec angle φ re-percute $\partial\Omega$ et $y' = \cos(\varphi')$, où φ' est l'angle entre la trajectoire entrante en $s\prime$ et la tangente à $\partial\Omega$ orientée négativement.

3.2 Les fonctionnelles $\ell_{\Omega,q}$

Lemme 2. *Soit $(\Omega_\tau)_{|\tau|\leq 1}$ une famille de domaine dans $\mathcal{S}^r$ et soit $\Delta \colon [-1, 1] \to \mathbb{R}$ une fonction de Darboux telle que $\Delta(\tau)) \in \mathcal{L}(\Omega_\tau)$. Si $(\Omega_\tau)_{|\tau|\leq 1}$ est isospectral, alors Δ est constant.*

Démonstration. cf [11]. □

Notons $\Delta_0(\tau)$ le périmètre de Ω_τ, c'est à dire :

$$\Delta_0(\tau) = \int_0^1 ||\partial_s \gamma(\tau, \xi)|| d\xi$$

Par définition, Δ_0 est continue et $\Delta_0(\tau) \in \mathcal{L}(\Omega_\tau)$, on conclut par notre précédent lemme que Δ_0 est constant, donc :

$$0 = \Delta_0'(\tau) = \int_0^1 \langle \partial_{\tau\xi} \gamma(\tau, \xi), T(\tau, \xi) \rangle d\xi$$

où $T(\tau, \xi) = \partial_s \gamma(\tau, \xi)/||\partial_s \gamma(\tau, \xi)||$ est le vecteur tangent à Ω_τ au point $\gamma(\tau, \xi)$, orienté positivement. En intégrant par partie on obtient :

$$0 = -\int_0^1 \langle \partial_\tau \gamma(\tau, \xi), \partial_s T(\tau, \xi) d\xi = \int_0^1 \frac{n(\tau, \xi)}{\rho_{\Omega_\tau}(\xi)} \frac{ds}{d\xi} d\xi$$

où $\frac{ds}{d\xi}$ représente la variation de la variable arc-length par rapport à ξ et, rappelons, ρ_{Ω_τ} est le rayon de curvature de $\partial\Omega_\tau$. Pour tout Ω (paramétrisé par ξ), on définit la fonction linéaire

$$\ell_{\Omega,0}(\nu) = \int_0^1 \frac{\nu(\xi)}{\rho_\Omega(\xi)} \frac{ds}{d\xi} d\xi$$

De part notre discussion, on conclut que si $(\Omega_\tau)_{|\tau|\leq 1}$ est une famille isospectrale, γ une paramétrisation de la famille, alors pour tout $\tau \in [-1, 1]$, on a $\ell_{\Omega,0}(n_\gamma(\tau, \cdot)) = 0$

Considérons une orbite periodique de période q et soit $p \in \mathbb{Z}$ son numéro d'enroulement. Alors on définit le nombre de rotations d'une orbite comme le ration p/q. Le lemme qui suit est une conséquence simple du fait que Ω possède une $\mathbb{Z}_2$-symétrie.

Lemme 3. *Soit $\Omega \in \mathcal{S}^r$, pour tout $q \geq 2$, il existe une orbite periodique de nombre de rotations $1/q$ passant par le point marqué de $\partial\Omega$ et avec une longueur maximale parmis les autres orbites periodiques passant par le point marqué. On appelle de telles orbites les orbites periodiques marquées symétriques maximales et on les note $S^q(\Omega)$.*

Démonstration. cf [11]. □

Définissons $L(\tau, s, s') = ||\gamma(\tau, s) - \gamma(\tau, s')||$. Pour $q \geq 2$, soit

$$L_q(\tau, s) = \begin{cases} 2\sum_{i=0}^{k-1} L(\tau, s_i, s_{i+1}) & \text{si } q = 2k \\ \sum_{i=0}^{k-1} 2L(\tau, s_i, s_{i+1}) - L(\tau, s_k, s_k, -s_k) & \text{si } q = 2k+1 \end{cases}$$

De manière analogue, on note $\Delta_q(\tau)$ la longueur de l'orbite periodique marquée symétrique maximale de nombre de rotations 1/q pour le domaine Ω_τ, donc :

$$\Delta_q(\tau) = \max_s L_q(\tau, s)$$

Observons que, par définition, $\Delta_q(\tau) \in \mathcal{L}(\Omega_\tau)$ et donc par le Corollaire, si $(\Omega_\tau)_{\tau\leq 1}$ est isospectral, alors Δ_q est constant. Il y a un obstacle lorsque l'on essaie de mettre en place la même stratégie que pour Δ_0 : a priori Δ_q n'est pas dérivable. Nous devons considérer alors une approche différente.

Lemme 4. *Pout tout $q \geq 2$, la fonction $\Delta_q \colon [-1,1] \to \mathbb{R}$ est une fonction Lipschitzienne.*

Démonstration. cf [11]. □

On définit le *sous-différentiel inférieur* D^- de la manière suivante : pour une fonction continue Φ, $p \in D^-\Phi(\tau)$ s'il existe $G \in C^1$ tq $G \leq \Phi$ sur un voisinage de τ, $G(\tau) = \Phi(\tau)$ et $G'(\tau) = p$.

Lemme 5. *Si $\tilde{\boldsymbol{s}}$ est un point maximal, i.e.*

$$L_q(\tau, \tilde{\boldsymbol{s}}) = \max_s L_q(\tau, s) = \Delta_q(\tau)$$

alors on a $\partial_\tau L(\tau, \tilde{\boldsymbol{s}}) \in D^-\Delta_q(\tau)$

Démonstration. cf [11]. □

Maintenant soit $\Omega \in \mathcal{M}$ paramétrisé par ξ et on suppose $S^q(\Omega) = (\xi_q^k, \varphi_q^k)_{k=0}^{q-1}$ une orbite périodique marquée symétrique maximale de nombre de rotations 1/q, alors on définit la fonctionnelle $\ell_{\Omega,q}$ comme ceci : pour toute fonction continue $\nu \colon \mathbb{T}^1 \to \mathbb{R}$,

$$\ell_{\Omega,q} := \frac{1}{q} \sum_{k=0}^{q-1} \nu(\xi_q^k) \sin(\varphi_q^k)$$

Proposition 1. *Soit $(\Omega_\tau)_{|\tau|\leq 1}$ une famille isospectrale, γ une paramétrisation de la famille, alors pour tout $\tau \in [-1,1], q \geq 2$, on a $\ell_{\Omega_\tau,q}(n_\gamma(\tau,\cdot)) = 0$.*

Démonstration. cf [11]. □

Enfin, on définit conventionnellement $\ell_{\Omega,1}(\nu)$ comme l'évaluation de la fonction ν au point marqué $s = 0$, donc on a simplement

$$\ell_{\Omega,1}(\nu) = \nu(0)$$

Observons que si $(\Omega_\tau)_{|\tau|\leq 1}$ est une famille normalisée, γ une paramétrisation de la famille, alors le point marqué est fixé à l'origine et alors $\ell_{\Omega_\tau,1}(n_\gamma(\tau,\cdot)) = 0$. On résume donc,

Corollaire 1. *Soit $(\Omega_\tau)_{|\tau|\leq 1}$ une famille isospectrale normalisée, γ une paramétrisation de la famille, alors pour tout $q \geq 0$, pour tout $\tau \in [-1,1]$. on a $\ell_{\Omega_\tau,q}(n_\gamma(\tau,\cdot)) = 0$.*

3.3 L'opérateur $\mathscr{T}_\Omega$

Soit $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ le cercle unité, muni de sa topologie quotient. On considère l'espace des fonctions réelles r-fois continûment dérivables et symétriques sur le cercle :

$$C^r_{\text{sym}}(\mathbb{T}) = \{f \in C^r(\mathbb{T}, \mathbb{R}) \mid f(-x) = f(x)\ \forall x \in \mathbb{T}\}$$

$\mathbb{T}$ étant compact, on munit cet espace de la norme suivante :

$$\|f\|_{C^r} = \max_{0 \le k \le r} \sup_{x \in \mathbb{T}} |f^{(k)}(x)|$$

Alors $C^r_{\text{sym}}(\mathbb{T})$ est un espace vectoriel normé complet, donc un espace de Banach.

De même, on considère l'espace des suites bornées :

$$\ell^\infty = \left\{(a_n)_{n\in\mathbb{N}} \mid \sup_n |a_n| < \infty\right\}$$

muni de la norme :

$$\|(a_n)_{n\in\mathbb{N}}\| = \sup_n |a_n|$$

qui est aussi un espace de Banach.

Pour $\Omega \in \mathbb{R}^2$, on définit maintenant l'application linéaire entre espaces de Banach $\mathscr{T}_\Omega \colon C^r_{sym} \to \ell^\infty$:

$$\mathscr{T}_\Omega(\nu) = (\ell_{\Omega,0}(\nu), \ell_{\Omega,1}(\nu), ..., \ell_{\Omega,q}(\nu), ...)$$

Le fait que $\forall \nu \in C^r_{sym}$, $\mathscr{T}_\Omega(\nu) \in \ell^\infty$ vient de [12].

On peut donc énoncer le théorème le plus important de ce rapport :

theorem 1. *Soit $\Omega \in \mathcal{S}^r$, si $\mathscr{T}_\Omega$ est injectif, alors Ω est DSR dans $\mathcal{S}^r$.*

Démonstration. Soit $(\Omega_\tau)_{|\tau|\le 1}$ une famille à 1-paramètre C^1-lisse, isospectrale, tel que $\Omega_0 = \Omega$. Supposons par contradiction qu'il existe un $\tau \in [-1, 1]$ tel que $n(\tau, \cdot)$ soit non identiquement nulle. Alors par injectivité de $\mathscr{T}_\Omega$, il existe $q \in \mathbb{N}$ tel que $\ell_{\Omega,q}(n(\tau, \cdot))$, ce qui contredit le corollaire 1. □

Pour savoir si un domaine Ω est DSR dans $\mathcal{S}^r$, il suffit alors de vérifier l'injectivité de $\mathscr{T}_\Omega$, un bon moyen de faire cela est de regarder les valeurs propres de $\mathscr{T}_\Omega$. En particulier, un ensemble Ω tel que $0 \in VP(\mathscr{T}_\Omega)$ n'est pas DSR. C'est la méthode que nous allons utiliser ici pour essayer de générer des ensembles non-DSR.

4 Les coordonnées de Lazutkin

Dans cette section, notre objectif est de transformer les coordonnées naturelles (s, φ) du billard en coordonnées adaptées (x, y) — les *coordonnées de Lazutkin* — dans lesquelles la dynamique près du bord devient presque linéaire.

Le but des coordonnées de Lazutkin est de transformer la dynamique du billard près du bord en un système presque intégrable, de la forme

$$x^+ = x + y + O(y^4), \qquad y^+ = y + O(y^4), \qquad \big((x^+, y^+) \text{ est le point d'impact après } (x, y)\big)$$

où y reste quasi constant et x se déplace presque linéairement.
Pour y parvenir, on procède ainsi :

1. Calculer, par développement de Frenet, les accroissements $s^+ - s$ et $\varphi^+ - \varphi$ en puissances de φ.
2. Chercher une nouvelle coordonnée horizontale $x = F(s)$ annulant les termes d'ordre y^2 dans $x^+ - x$, ce qui conduit à la paramétrisation de Lazutkin.
3. Introduire la coordonnée verticale y proportionnelle à $\sin(\varphi/2)$ et *pondérée* par le poids de Lazutkin $\mu(x)$ pour uniformiser le terme principal du déplacement.

On obtient alors la forme normalisée de Lazutkin, point de départ pour l'étude des caustiques proches du bord.

4.1 Calcul des développements en série

On veut exprimer les accroissements « suivant » et « précédent » comme séries :

$$s^\pm - s = \sum_{j\geq 1} b_j^\pm \, \varphi^j, \qquad \varphi^\pm - \varphi = \sum_{j\geq 2} d_j^\pm \, \varphi^j,$$

Cadre et notations
On considère une courbe $\partial\Omega \subset \mathbb{R}^2$ de régularité C^r $(r \geq 3)$ strictement convexe, paramétrée par longueur d'arc

$$\gamma : \mathbb{R} \to \mathbb{R}^2, \qquad s \mapsto \gamma(s), \qquad \|\gamma'(s)\| = 1.$$

On note :

— $T(s) = \gamma'(s)$ le *vecteur tangent unitaire* (vecteur de $\mathbb{R}^2$),
— $N(s)$ le *vecteur normal unitaire intérieur* tel que (T, N) est une base orthonormée directe,
— $\kappa(s) > 0$ la *courbure* (scalaire), et $\rho(s) = 1/\kappa(s)$ le *rayon de courbure* (scalaire).

Toutes ces quantités sont de classe C^{r-1}.

4.1.1 Développement de $s^\pm - s$

En dimension 2 et paramétrisation par longueur d'arc, par les formules de Frenet-Serret on a

$$T'(s) = \kappa(s)\, N(s), \qquad N'(s) = -\kappa(s)\, T(s).$$

En dérivant succesivement γ, on obtient

$$\gamma'(s) = T(s), \qquad \gamma''(s) = \kappa(s)\, N(s), \qquad \gamma^{(3)}(s) = \kappa'(s)N(s) - \kappa(s)^2 T(s).$$

Par développement de Taylor autour de s :

$$\begin{aligned}\gamma(s+\sigma) &= \gamma(s) + \gamma'(s)\,\sigma + \frac{1}{2}\gamma''(s)\,\sigma^2 + \frac{1}{6}\gamma^{(3)}(s)\,\sigma^3 + O(\sigma^4) \\ &= \gamma(s) + T(s)\,\sigma + \frac{\kappa(s)}{2} N(s)\,\sigma^2 + \frac{\kappa'(s)N(s) - \kappa(s)^2 T(s)}{6}\,\sigma^3 + O(\sigma^4).\end{aligned}$$

Application au vecteur $\Delta(\sigma)$

On définit la *corde* : $\Delta(\sigma) := \gamma(s+\sigma) - \gamma(s)$.

En projetant sur la base $(T(s), N(s))$, on obtient les composantes tangentielles et normales :

$$\Delta_T(\sigma) := \Delta(\sigma)\cdot T(s) = \sigma - \frac{\kappa(s)^2}{6}\sigma^3 + O(\sigma^4),$$
$$\Delta_N(\sigma) := \Delta(\sigma)\cdot N(s) = \frac{\kappa(s)}{2}\sigma^2 + \frac{\kappa'(s)}{6}\sigma^3 + O(\sigma^4).$$

Si φ est l'angle (petit) entre $T(s)$ et la direction de $\Delta(\sigma)$ (i.e. la direction du prochain segment de trajectoire), alors

$$\tan\varphi = \frac{\Delta_N(\sigma)}{\Delta_T(\sigma)}.$$

En injectant les séries précédentes,

$$\begin{aligned}
\tan\varphi &= \frac{\frac{\kappa}{2}\sigma^2 + \frac{\kappa'}{6}\sigma^3 + O(\sigma^4)}{\sigma - \frac{\kappa^2}{6}\sigma^3 + O(\sigma^4)} \\
&= \left[\frac{\kappa}{2}\sigma + \frac{\kappa'}{6}\sigma^2 + O(\sigma^3)\right] \times \frac{1}{1 - \sigma^2\left(\frac{\kappa^2}{6} + O(\sigma)\right)} \\
&= \left[\frac{\kappa}{2}\sigma + \frac{\kappa'}{6}\sigma^2 + O(\sigma^3)\right] \times \left[1 + \sigma^2\left(\frac{\kappa^2}{6} + O(\sigma)\right) + O(\sigma^4)\right] \\
&= \left[\frac{\kappa}{2}\sigma + \frac{\kappa'}{6}\sigma^2 + O(\sigma^3)\right] \times \left[1 + \frac{\kappa^2}{6}\sigma^2 + O(\sigma^3)\right] \\
&= \frac{\kappa}{2}\sigma + \frac{\kappa'}{6}\sigma^2 + O(\sigma^3)
\end{aligned}$$

Comme $\varphi = \arctan(\tan\varphi) = \tan\varphi + O(\tan(\varphi)^3) = \tan\varphi + O(\sigma^3)$ car $\tan(\varphi)^3 = O(\sigma^3)$, on a à l'ordre utile

$$\varphi = \frac{\kappa}{2}\sigma + \frac{\kappa'}{6}\sigma^2 + O(\sigma^3).$$

On cherche σ sous la forme

$$\sigma = p\,\varphi + q\,\varphi^2 + O(\varphi^3).$$

En substituant dans $\varphi = A\sigma + B\sigma^2 + O(\sigma^3)$ avec $A = \kappa/2$, $B = \kappa'/6$:

$$\begin{aligned}
\varphi &= A(p\varphi + q\varphi^2) + B(p\varphi)^2 + O(\varphi^3) \\
&= (Ap)\,\varphi + \left(Aq + Bp^2\right)\varphi^2 + O(\varphi^3).
\end{aligned}$$

Identification des coefficients de φ et φ^2 :

$$Ap = 1 \implies p = \frac{1}{A} = \frac{2}{\kappa}, \qquad Aq + Bp^2 = 0 \implies q = -\frac{B}{A^3} = -\frac{\kappa'/6}{(\kappa/2)^3} = -\frac{4}{3}\frac{\kappa'}{\kappa^3}.$$

Donc

$$\sigma = \frac{2}{\kappa}\,\varphi - \frac{4}{3}\frac{\kappa'}{\kappa^3}\,\varphi^2 + O(\varphi^3)$$

Comme $\kappa = 1/\rho$ et $\kappa' = -(\dot\rho)/\rho^2$ avec $\dot\rho = d\rho/ds$, on obtient

$$-\frac{4}{3}\frac{\kappa'}{\kappa^3} = -\frac{4}{3}\cdot\frac{-\dot\rho/\rho^2}{(1/\rho)^3} = \frac{4}{3}\,\rho\,\dot\rho.$$

Ainsi,

$$\sigma = 2\rho\,\varphi + \frac{4}{3}\rho\,\dot\rho\,\varphi^2 + O(\varphi^3)$$

et, comme $s^+ - s = \sigma$, les deux premiers coefficients de la série $s^+ - s = \sum_{j\geq 1} b_j^+\,\varphi^j$ sont

$$\boxed{b_1^+ = 2\rho \qquad b_2^+ = \frac{4}{3}\rho\,\dot\rho}$$

Remarques :

1. La stricte convexité ($\kappa > 0$) et la régularité assurent, via le théorème des fonctions implicites, l'existence d'une solution lisse en φ (avec s fixé initialement) $\sigma = \sigma^+(s,\varphi)$ pour $|\varphi|$ petit.
2. La réversibilité $f^{-1} = \mathcal{R}\circ f\circ\mathcal{R}$, avec $\mathcal{R}(s,\varphi) = (s,-\varphi)$, implique les parités $b_j^+ = (-1)^j b_j^-$.

4.1.2 Développement de $\varphi^\pm - \varphi$

On note φ l'angle de sortie au point s entre la tangente $T(s)$ et la direction de la trajectoire juste après le rebond, et φ^+ l'angle au point suivant $s^+ = s + \sigma$ entre $T(s^+)$ et la direction de la trajectoire juste après ce rebond-là.

Entre s et s^+, la direction de la trajectoire est constante (mouvement rectiligne), mais la tangente au bord change car celui-ci est courbé. La tangente tourne d'un angle

$$\Delta\theta = \int_s^{s^+} \kappa(u)\,du.$$

La loi de la réflexion donne alors la relation géométrique

$$\varphi^+ = \Delta\theta - \varphi,$$

d'où

$$\varphi^+ - \varphi = \Delta\theta - 2\varphi.$$

On développe $\kappa(u)$ autour de s :

$$\kappa(u) = \kappa(s) + \kappa'(s)\,(u-s) + \frac{1}{2}\kappa''(s)\,(u-s)^2 + O((u-s)^3).$$

En intégrant de s à $s+\sigma$:

$$\Delta\theta = \kappa(s)\sigma + \frac{\kappa'(s)}{2}\sigma^2 + \frac{\kappa''(s)}{6}\sigma^3 + O(\sigma^4).$$

On injecte le développement obtenu précédemment

$$\sigma = 2\rho\,\varphi + \frac{4}{3}\rho\,\dot{\rho}\,\varphi^2 + O(\varphi^3)$$

dans $\varphi^+ - \varphi = \Delta\theta - 2\varphi$.

Le terme linéaire en φ dans $\Delta\theta$ est exactement 2φ et s'annule donc avec le -2φ de $\varphi^+ - \varphi$. Le premier terme non nul provient donc de l'ordre φ^2.
Contribution 1 : terme en $\kappa(s)$ multiplié par le terme $\frac{4}{3}\rho\,\dot{\rho}\,\varphi^2$ de σ :

$$\kappa \cdot \frac{4}{3}\rho\,\dot{\rho} = \frac{4}{3}\dot{\rho}$$

Contribution 2 : terme en $\frac{\kappa'(s)}{2}$ multiplié par le carré du terme principal de σ :

$$\frac{\kappa'}{2} \cdot (2\rho\,\varphi)^2 = \frac{\kappa'}{2} \cdot 4\rho^2\,\varphi^2.$$

Or $\kappa' = -\frac{\dot{\rho}}{\rho^2}$, donc cette contribution vaut $-2\dot{\rho}$.
En additionnant les contributions :

$$\varphi^+ - \varphi = \left(\frac{4}{3}\dot{\rho} - 2\dot{\rho}\right)\varphi^2 + O(\varphi^3) = -\frac{2}{3}\dot{\rho}\,\varphi^2 + O(\varphi^3).$$

Ainsi :

$$\boxed{d_2^+ = -\frac{2}{3}\dot{\rho}}$$

Remarque :

1. Comme pour les $b_j^\pm$, la réversibilité du billard $f^{-1} = \mathcal{R}\circ f\circ\mathcal{R}$, implique une relation entre les coefficients $d_j^\pm$. En effet, $\varphi^- - \varphi =$ composante en φ de $f(s,-\varphi) - (-\varphi)$, ce qui conduit, après développement, à $d_j^- = (-1)^{j+1} d_j^+$.

Au final, on a

$$\boxed{\begin{cases} s^+ = s + 2\rho(s)\,\varphi + \frac{4}{3}\rho(s)\,\dot{\rho}(s)\,\varphi^2 + O(\varphi^3) \\ \varphi^+ = \varphi - \frac{2}{3}\dot{\rho}(s)\,\varphi^2 + O(\varphi^3) \end{cases}}$$

Ce n'est **pas** une translation simple : le coefficient devant φ dans $s^+ - s$ dépend de s, donc la dynamique n'est pas homogène.

4.2 Détermination de $F(s^\pm)$

On effectue un développement de Taylor en $s^\pm - s$ (en gardant les termes d'ordres inférieur à 2 en φ) :

$$F(s^\pm) = F(s) + F'(s)\,(s^\pm - s) + \tfrac{1}{2}F''(s)\,(s^\pm - s)^2 + O(\varphi^3).$$

En sommant « suivant » et « précédent » et en soustrayant $2F(s)$, on obtient

$$F(s^+) - 2F(s) + F(s^-) = F'(s)\big[(s^+ - s) + (s^- - s)\big] + \tfrac{1}{2}F''(s)\big[(s^+ - s)^2 + (s^- - s)^2\big] + O(\varphi^3).$$

Avec les séries :

$$s^+ - s = b_1\varphi + b_2\varphi^2 + O(\varphi^3), \qquad s^- - s = -b_1\varphi + b_2\varphi^2 + O(\varphi^3),$$

(j'utilise la parité $b_j^- = (-1)^j b_j^+$). Ainsi :

$$(s^+ - s) + (s^- - s) = 2b_2\,\varphi^2 + O(\varphi^3),$$

$$(s^+ - s)^2 + (s^- - s)^2 = (b_1\varphi)^2 + (-b_1\varphi)^2 + O(\varphi^3) = 2b_1^2\varphi^2 + O(\varphi^3).$$

En réinjectant :

$$F(s^+) - 2F(s) + F(s^-) = \Big(2b_2F'(s) + b_1^2F''(s)\Big)\,\varphi^2 + O(\varphi^3).$$

Avec $b_1 = 2\rho$ et $b_2 = \frac{4}{3}\rho\,\dot\rho$, on trouve le coefficient d'ordre 2 en φ :

$$\frac{4}{3}\left(2\rho\dot\rho\,F'(s) + 3\rho^2F''(s)\right).$$

On veut annuler ce coefficient pour rendre y invariant à cet ordre.

On impose donc :

$$2\rho\dot\rho\,F'(s) + 3\rho^2F''(s) = 0.$$

Posons $G = F'$. L'ODE devient :

$$2\frac{\dot\rho}{\rho}\,G + 3\,G' = 0 \quad\Longrightarrow\quad \frac{G'}{G} = -\frac{2}{3}\frac{\dot\rho}{\rho}.$$

Intégration :

$$\ln G = -\frac{2}{3}\ln\rho + \text{const} \quad\Longrightarrow\quad G(s) = F'(s) = C\,\rho(s)^{-2/3}.$$

Donc :

$$\boxed{F(s) = C\int_0^s \rho(z)^{-2/3}\,dz}$$

(à une constante additive près). C'est le *choix de Lazutkin* à l'ordre principal.

4.3 Forme normale de Lazutkin

On fixe la paramétrisation de Lazutkin

$$x(s) = C_{\mathbb{I}}\int_0^s \rho(s')^{-2/3}\,ds', \qquad C_{\mathbb{I}}^{-1} = \int_{\partial\Omega}\rho(s')^{-2/3}\,ds',$$

et on introduit le *poids de Lazutkin*

$$\mu(x) \;=\; \frac{1}{2\,C_{\mathbb{I}}\,\rho(x)^{1/3}} \;>\; 0.$$

On définit ensuite la variable verticale (normalisée) par

$$y \;=\; \frac{2\,\sin(\varphi/2)}{\mu(x)} \qquad (\text{i.e. } y = 4\,C_{\mathbb{I}}\,\rho(x)^{1/3}\sin(\varphi/2)).$$

Étape 1 : *Linéarisation du déplacement en x.*

Par construction,

$$x^+ - x \;=\; 2C_{\mathbb{I}}\rho^{1/3}\,\varphi \;+\; O(\varphi^3).$$

Or, par définition de y,

$$y \;=\; \frac{2\,\sin(\varphi/2)}{\mu(x)} \;=\; \frac{2}{\mu(x)}\Big(\tfrac{\varphi}{2} + O(\varphi^3)\Big) \;=\; 2C_{\mathbb{I}}\rho^{1/3}\,\varphi \;+\; O(\varphi^3).$$

Ainsi, au premier ordre,

$$x^+ - x \;=\; y + O(\varphi^3) \quad \text{et, en réécrivant } \varphi \text{ en fonction de } y, \quad \boxed{x^+ - x \;=\; y \;+\; O(y^3)}$$

Étape 2 : *Élimination systématique des termes jusqu'à l'ordre* 3. Le choix $x(s)$ ci-dessus supprime *tous* les termes d'ordre 2 dans $x^+ - x$. Pour supprimer également les termes d'ordre 3 dans *les deux* composantes, on raffine la définition de y en ajoutant une correction cubique dépendant de x :

$$y \;=\; \frac{2\,\sin(\varphi/2)}{\mu(x)} \;+\; a_3(x)\,\sin^3(\varphi/2),$$

où $a_3(x)$ est choisi de sorte que les contributions cubiques dans $x^+ - x$ et $y^+ - y$ s'annulent. Un calcul direct (développements de $s^+ - s$ et $\varphi^+ - \varphi$, plus chaîne de dérivation $y = y(x,\varphi)$) donne alors

$$x^+ \;=\; x + y + O(y^4), \qquad y^+ \;=\; y + O(y^4).$$

Autrement dit, *dans les coordonnées (x, y) pondérées par $\mu(x)$*, la dynamique du billard près du bord est une translation quasi-intégrable à l'erreur $O(y^4)$.

Comment obtenir $y^+ = y + O(\varphi^3)$ dans la première version :

Rappelons que $x(s) = F(s)$ et $F'(s) > 0$ donc F est localement inversible en s : $s = F^{-1}(x(s))$. Gardons donc en tête que $\rho(x)$ est un abus de notation, en réalite $\rho(x) = \rho(F^{-1}(x))$
On part de la définition

$$y(x,\varphi) = \frac{2\,\sin(\varphi/2)}{\mu(x)} = 4\,C_{\mathbb{I}}\,\rho^{1/3}(x)\,\sin(\varphi/2)$$

On veut estimer $y^+ - y = y(x^+,\varphi^+) - y(x,\varphi)$ jusqu'à l'ordre 3.

Posons $A(x) := 2C_{\mathbb{I}}\rho^{1/3}(x)$ de sorte que $y(x,\varphi) = 2A(x)\sin(\varphi/2)$, on a

$$\partial_\varphi y(x,\varphi) = A(x)\,\cos(\varphi/2),$$

$$\partial_x y(x,\varphi) = 4C_{\mathbb{I}}\sin(\varphi/2)\cdot\frac{1}{3}\rho(F^{-1}(x))^{-2/3}\cdot(\partial_x F^{-1})(x)\cdot(\partial_{F^{-1}(x)}\rho)(F^{-1}(x))$$

Et donc,

$$\partial_x y(x,\varphi) = 4C_{\mathbb{I}}\sin(\varphi/2)\cdot\frac{1}{3}\rho(F^{-1}(x))^{-2/3}\cdot\frac{1}{C_{\mathbb{I}}\rho(F^{-1}(x))^{-2/3}}\cdot\dot\rho(F^{-1}(x))$$

$$\partial_x y(x,\varphi) = \frac{4}{3}\dot\rho\sin(\varphi/2)$$

On en déduit les développements (pour φ petit) :

$$\partial_\varphi y = A\Big(1+O(\varphi^2)\Big), \qquad \partial_x y = \frac{2}{3}\,\dot\rho\,\varphi + O(\varphi^3).$$

Incréments en x et φ. À l'ordre requis,

$$x^+ - x = A\,\varphi + O(\varphi^3), \qquad \varphi^+ - \varphi = d_2\,\varphi^2 + O(\varphi^3),$$

Développement de $y^+ - y$. Par formule de Taylor multivariée (en ne gardant que les termes d'ordre 2 ; les autres seront $O(\varphi^3)$) :

$$\begin{aligned} y^+ - y =& (\partial_x y)(x^+-x) + (\partial_\varphi y)(\varphi^+-\varphi) \\ &+ \frac{1}{2}(\partial_{xx}y)(x^+-x)^2 + \frac{1}{2}(\partial_{\varphi\varphi}y)(\varphi^+-\varphi)^2 + (\frac{1}{2}\partial_{x\varphi}y)(x^+-x)(\varphi^+-\varphi) \\ &+ O(\|(x^+-x,\varphi^+-\varphi)\|^3) \end{aligned}$$

Des calculs rapident montrent que,

$$\partial_{xx}y = \frac{4}{3C_{\mathbb{I}}}\sin\Big(\frac{\varphi}{2}\Big)\,\rho^{2/3}\,\ddot\rho \;= O(\varphi)$$

$$\partial_{\varphi x}y = \frac{2}{3}\cos\Big(\frac{\varphi}{2}\Big)\,\dot\rho \;= O(1)$$

$$\partial_{\varphi\varphi}y = -\,C_{\mathbb{I}}\sin\Big(\frac{\varphi}{2}\Big)\,\rho^{1/3}\;O(\varphi)$$

Or, avec les ordres ci-dessus,

$$O\big((x^+-x)^2\big) = O(\varphi^2), \quad O\big((\varphi^+-\varphi)^2\big) = O(\varphi^4), \quad O\big((x^+-x)(\varphi^+-\varphi)\big) = O(\varphi^3).$$

Nous ne retenons que les contributions principales en φ^2 :

$$(\partial_x y)(x^+-x) = \Big(\tfrac{2}{3}\dot\rho\,\varphi + O(\varphi^3)\Big)\Big(A\,\varphi + O(\varphi^3)\Big) = \tfrac{2}{3}\,A\,\dot\rho\,\varphi^2 + O(\varphi^3),$$

$$(\partial_\varphi y)(\varphi^+-\varphi) = \Big(A+O(\varphi^2)\Big)\Big(d_2\,\varphi^2+O(\varphi^3)\Big) = A\,d_2\,\varphi^2+O(\varphi^3) = -\tfrac{2}{3}\,A\,\dot\rho\,\varphi^2+O(\varphi^3).$$

Les termes d'ordre φ^2 *s'annulent exactement* :

$$(\partial_x y)(x^+-x) + (\partial_\varphi y)(\varphi^+-\varphi) = O(\varphi^3).$$

Les termes restants (quadratiques croisés et seconds ordres en Taylor) sont tous $O(\varphi^3)$ ou plus petits, d'où $y^+ - y \;=\; O(\varphi^3)$, et comme $y \sim A\,\varphi$ pour φ petit, on peut aussi écrire :

$$\boxed{y^+ - y = O(y^3)}$$

De manière analogue, en poussant les développements aux ordres supérieures, on peut diminuer les non-linéarités. Le choix $x(s)$ ci-dessus supprime *tous* les termes d'ordre y^2 dans $x^+ - x$. Pour supprimer également les termes d'ordre y^3 dans *les deux* composantes, on raffine la définition de y en ajoutant une correction cubique dépendant de x :

$$y \;=\; \frac{2\,\sin(\varphi/2)}{\mu(x)} \;+\; a_3(x)\,\sin^3(\varphi/2),$$

où $a_3(x)$ est choisi de sorte que les contributions cubiques dans $x^+ - x$ et $y^+ - y$ s'annulent. Un calcul nous donne alors

$$x^+ \;=\; x + y + O(y^4), \qquad y^+ \;=\; y + O(y^4).$$

Autrement dit, *dans les coordonnées (x, y) pondérées par $\mu(x)$,* la dynamique du billard près du bord est une translation quasi-intégrable à l'erreur $O(y^4)$.

4.4 $\rho(s)$ ou $\rho(\theta)$?

Faisons une petite aparté ici, afin d'établir qu'il est possible de travailler avec une paramétrisation θ, appeléé l'angle de la tangente, au lieu de travailler avec s la longueur d'arc. Çela nous est utile car toute la partie numérique sera traitée avec la paramétrisation θ.
Soit γ une courbe plane C^2, strictement convexe et orientée positivement, paramétrée par la longueur d'arc $s \mapsto \gamma(s)$ (donc $|\gamma'(s)| = 1$). On note $T(s) = \gamma'(s) = (\cos\theta(s), \sin\theta(s))$ l'angle de la tangente et

$$\kappa(s) = \frac{d\theta}{ds} > 0, \qquad \rho(s) = \frac{1}{\kappa(s)}.$$

La stricte convexité implique que $\theta : \mathbb{S}^1 \to \mathbb{S}^1$ est un difféomorphisme (carte de Gauss). On peut donc inverser θ et considérer $s = s(\theta)$; on définit alors

$$\rho(\theta) := \rho\big(s(\theta)\big) \qquad \text{et} \qquad ds = \rho(\theta)\,d\theta.$$

On voit que définir le rayon de courbure en fonction de s ou en fonction de θ revient au même, les deux n'étant reliés que par un changement de variable.

Toutes les quantités géométriques utilisées ensuite s'écrivent de façon cohérente dans les deux paramètres :

$$\text{Périmètre :} \quad L = \int_{\partial\Omega} ds \;=\; \int_0^{2\pi} \rho(\theta)\,d\theta.$$

$$\text{Coordonnée de Lazutkin :} \quad x(s) = C_L \int_0^s \rho(u)^{-2/3}\,du \;=\; C_L \int_0^\theta \rho(t)^{1/3}\,dt =: x(\theta).$$

$$\text{Constante de Lazutkin :} \quad C_L^{-1} = \int_{\partial\Omega} \rho^{-2/3}\,ds \;=\; \int_0^{2\pi} \rho(\theta)^{1/3}\,d\theta.$$

$$\text{Poids de Lazutkin :} \quad \mu(x) = \frac{1}{2\,C_L\,\big(\rho \circ s(x)\big)^{1/3}}$$

Autrement dit, par composition : $\mu\big(x(s)\big) = \dfrac{1}{2C_L\,\rho(s)^{1/3}}$ et, si l'on paramètre par l'angle de tangente θ, $\mu\big(x(\theta)\big) = \dfrac{1}{2C_L\,\rho(\theta)^{1/3}}$.
Ainsi, que l'on écrive ρ comme $\rho(s)$ ou $\rho(\theta)$, les objets L, x, C_L, μ (et donc toutes les constructions ultérieures qui en dépendent) sont invariants par reparamétrisation.

5 L'opérateur $\tilde{\mathscr{T}}_\Omega$

Il est plus naturel de définir la suite auxiliaire de fonctionnelles

$$\tilde{\ell}_{\Omega,q}(u) := \ell_{\Omega,q}(\mu^{-1}u),$$

et, en conséquence, l'opérateur

$$\tilde{\mathscr{T}}_\Omega u := \big(\tilde{\ell}_{\Omega,0}(u),\, \tilde{\ell}_{\Omega,1}(u),\, \dots,\, \tilde{\ell}_{\Omega,q}(u),\, \dots\big). \tag{5.4}$$

Comme μ ne s'annule pas, l'injectivité de $\tilde{\mathscr{T}}_\Omega$ est équivalente à celle de $\mathscr{T}_\Omega$. Cependant, l'opérateur $\tilde{\mathscr{T}}_\Omega$ est plus commode à étudier.

Calculons $\tilde{\ell}_{\Omega,0}(u) = \ell_{\Omega,0}(\mu^{-1}u)$;
Par définition de $\ell_{\Omega,0}$,

$$\ell_{\Omega,0}(\nu) = \int_0^1 \nu(\xi)\,\frac{1}{\rho(\xi)}\,\frac{ds}{d\xi}\,d\xi = \int \nu(s)\,\frac{ds}{\rho(s)}.$$

Or,

$$\frac{dx}{ds} = C_{\mathbb{I}}\,\rho(s)^{-2/3}, \qquad ds = \frac{1}{C_{\mathbb{I}}}\,\rho(x)^{2/3}\,dx,$$

$$\begin{aligned}
\tilde{\ell}_{\Omega,0}(u) &= \int \mu^{-1}(x)\,u(x)\,\frac{1}{\rho(x)}\,ds \\
&= \int \mu^{-1}(x)\,u(x)\,\frac{1}{\rho(x)}\left(\frac{1}{C_{\mathbb{I}}}\rho(x)^{2/3}\,dx\right) \\
&= \int \mu^{-1}(x)\,u(x)\,\frac{1}{C_{\mathbb{I}}}\,\rho(x)^{-1/3}\,dx.
\end{aligned}$$

Comme $\mu^{-1}(x) = 2C_{\mathbb{I}}\,\rho(x)^{1/3}$, on obtient

$$\tilde{\ell}_{\Omega,0}(u) = 2\int_0^1 u(x)\,dx.$$

c'est-à-dire que $\tilde{\ell}_0$ est proportionnelle à la fonctionnelle de moyenne pour la mesure de Lebesgue. D'autre part,

$$\tilde{\ell}_1(u) \;=\; \mu^{-1}(0)\,u(0),$$

autrement dit, l'évaluation de $\mu^{-1}u$ au point marqué. C'est donc cet opérateur $\tilde{\mathscr{T}}_\Omega$ que nous étudierons.

Deuxième partie

Étude Numérique

6 Création d'un domaine

Pour créer numériquement un domaine $\Omega \subset \mathbb{R}^2$, on a d'abord besoin d'une paramétrisation. Retenons la paramétrisation suivante, pour $\theta \in [0, 2\pi]$

$$\gamma(\theta) = \left(-\int_0^\theta \rho(s)\sin(s)\,ds, \quad \int_0^\theta \rho(s)\cos(s)\,ds \right)$$

Un domaine Ω est alors totalement décrit par son rayon de courbure ρ. La création du rayon de courbure est alors au choix, la plupart du temps nous exprimerons ρ comme troncature de série de Fourier, i.e.

$$\rho(\theta) = \sum_{k=0}^{n} a_k \cos(k\theta) + \sum_{k=0}^{n} b_k \sin(k\theta)$$

. Nous voulons générer un domaine $\mathbb{Z}_2$-symétrique, alors $\rho(\theta) = \rho(-\theta)$ engendre $\rho(\theta) = \sum_{k=0}^{n} a_k \cos(k\theta)$. De plus, nous voulons travailler à perimètre constant L égal à 2π, donc

$$a_0 = \frac{1}{2\pi}\int_0^{2\pi} \rho(\theta)\,d\theta = \frac{L}{2\pi}$$

entraine $a_0 = 1$.
On veut que notre courbe γ se referme sur elle-même, alors l'égalité

$$\gamma(2\pi) - \gamma(0) = \int_0^{2\pi} \rho(s) \begin{pmatrix} -\sin(s) \\ \cos(s) \end{pmatrix} ds = \vec{0}$$

Nous donne $\int_0^{2\pi} \rho(s)\cos(s)ds = 0$ donc $a_1 = \frac{1}{\pi}\int_0^{2\pi} \rho(\theta)\cos(\theta)\,d\theta = 0$

Au final, le rayon de courbure ρ s'exprime comme,

$$\rho(\theta) = 1 + \sum_{k=2}^{n} a_k \cos(k\theta)$$

6.1 Domaines analytiques générés numériquement

On considère une fonction $\rho(\theta)$ sous la forme :

$$\rho_N(\theta) = 1 + \sum_{k=2}^{N} a_k \cos(k\theta),$$

où les coefficients a_k sont choisis de manière à assurer la convexité stricte (i.e. $\rho(\theta) > 0$).

Cette méthode est simple à implémenter numériquement et permet de contrôler directement la régularité : par exemple, en imposant $a_k = O(1/k^9)$, on peut approcher

une fonction C^9.
Or, toute fonction obtenue par somme finie de termes analytiques est elle-même analytique. Donc, même si on approche une régularité C^9, on reste dans le cadre analytique. Le domaine obtenu reste dans le cadre du théorème de Zelditch.

Donc pour résumer, la méthode par troncature de Fourier est efficace et bien adaptée à l'expérimentation numérique, mais reste confinée à l'univers analytique, ce qui limite la portée des observations si l'on veut tester des hypothèses au-delà du cadre de Zelditch. À l'inverse, la méthode de fenêtre de régularisation est plus complexe à mettre en œuvre mais permet d'explorer de manière rigoureuse des domaines strictement C^9 non-analytiques, ouvrant la voie à des expériences sur la rigidité dynamique spectrale dans un cadre plus général. Pour notre étude, on se limitera à la méthode 1 par troncature de série de Fourier.

Génération du domaine

- `rho_coeff(n)` / `rho_coeff_non_normalized(n)` : génèrent des coefficients de Fourier (a_j) (graines reproductibles).
- `rho_function` : évalue $\rho(\theta) = \sum_j a_j \cos(j\theta)$.
- `plot_rho` : trace ρ sur $[0, 2\pi]$ (contrôle de la positivité).
- `gamma` : paramétrise le bord $\partial\omega$ via des intégrales explicites de ρ (retourne $(x(\theta), y(\theta))$).

6.2 Domaines usuelles

Nous utiliserons souvent les mêmes domaines,

$$\textit{le cercle :}\ \mathcal{E} = [1], \quad \mathcal{A} = [1, 0, 0.5, 0.45, 0.3], \quad \mathcal{B} = [1, 0, 0, 0.99]$$

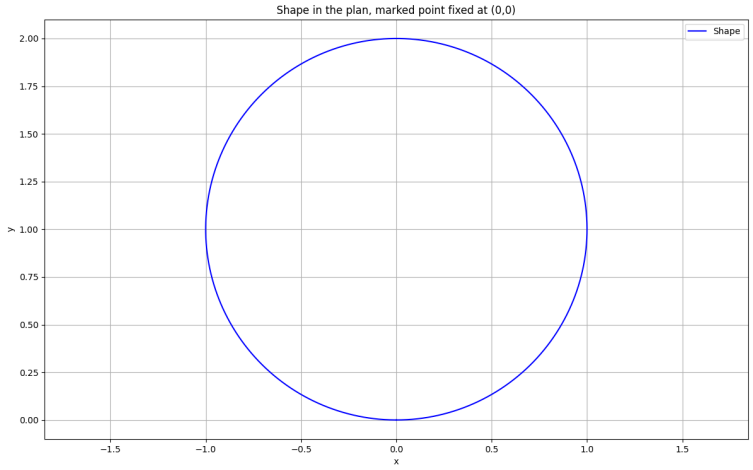

FIGURE 1 – Domaine $\mathcal{E}$

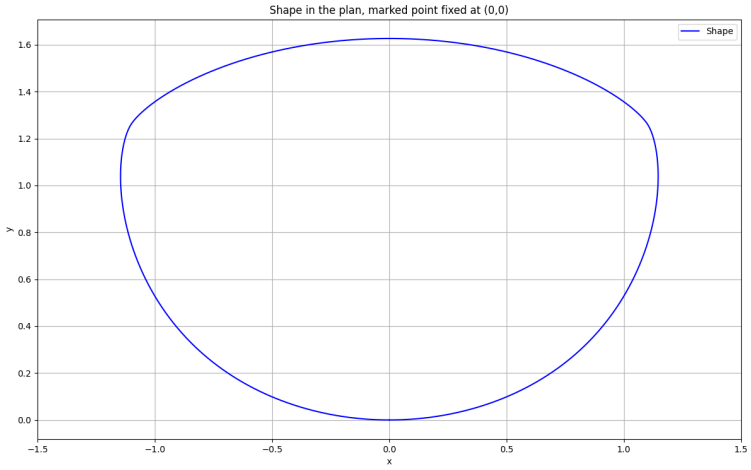

FIGURE 2 – Domaine $\mathcal{A}$

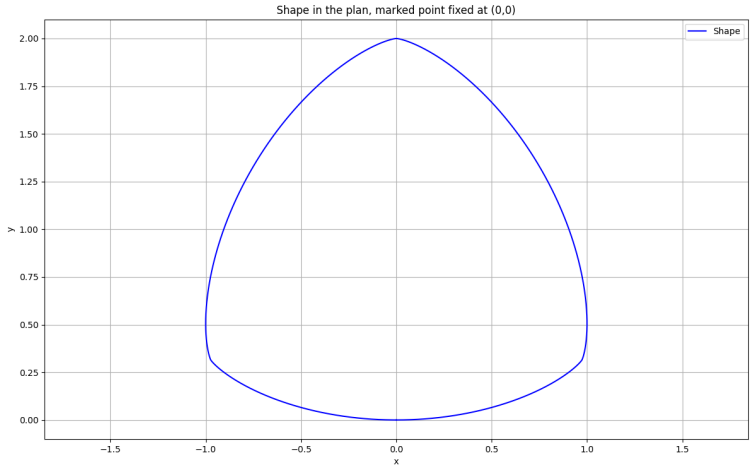

FIGURE 3 – Domaine $\mathcal{B}$

7 Génération des trajectoires périodiques

Nous voilà maintenant munis d'un domaine Ω, on veut générer des trajectoires périodiques de périodes $1/q$, avec $q \geq 2$, émanants de l'origine. Pour faire cela, on cherche les points du bords $(p_q^k)_{k=0}^{q-1} \in \partial\Omega \subset \mathbb{R}^2$ qui maximisent

$$L_q(\xi) = \left(\sum_{k=0}^{q-2} \|p_q^{k+1} - \ p_q^k\| \right) + \|p_q^0 - p_q^{q-1}\|$$

En réalité, on cherche les $(\theta_q^{k=0})_k^{q-1}$ qui maximisent

$$L_q(\theta) = \left(\sum_{k=0}^{q-2} \|\gamma(\theta_q^{k+1}) - \gamma(\theta_q^k)\| \right) + \|\gamma(\theta_q^0) - \gamma(\theta_q^{q-1})\|$$

— `norme2` : norme euclidienne en $\mathbb{R}^2$.

— `minus_L` : $-L(\Theta)$, opposé de la longueur totale du polygone passant par $\gamma(\theta_i)$.

— `L_maximizer` : minimise `minus_L` (SciPy) $\Rightarrow$ angles optimaux θ_i (mod 2π).

— `find_impacts` : mappe les angles optimaux en points d'impact $p_i = \gamma(\theta_i)$.

— `trajectories_visualisation` : trace la frontière et l'orbite $1/q$–périodique.

Pour trouver les impacts, on maximise une fonction L définie sur $[0; 2\pi[^q$, ce qui peut être coûteux à calculer. Un système de coordonnées bien choisies pourrait améliorer ce temps de calcul. En effet, on sait qu'en coordonnées de Lazutkin les points d'impact sont quasiment linéaires. Alors, pour une q orbite périodique, les points d'impact p_q^k sont environ égal à k/q. Ce système bien choisi de conditions initiales permettrait d'améliorer la vitesse de l'algorithme.

— `rho_function_power_1over3` : $\rho^{1/3}$.

— `lazutkin_parameter` : $C_\omega = \left(\int_0^{2\pi} \rho^{1/3}\right)^{-1}$.

— `x_lazutkin`, `d_x_lazutkin` : $x(\theta) = C_\omega \int_0^\theta \rho^{1/3}$ et sa dérivée $C_\omega \rho^{1/3}$.

— `solve_lazutkin_newton` / `solve_lazutkin_root` : inversion de $x(\theta) = y$ (Newton / Brent).

— `find_theta0_newton` / `find_theta0_root` : amorces $\theta_{0,j}$ telles que $x(\theta_{0,j}) = j/q$.

Enfin, afin de pouvoir définir notre opérateur $\tilde{\mathscr{T}}_\Omega$, nous avons besoin des points d'impact $(p_q^k)_{k=0}^{q-1}$ mais également des angles $(\varphi_q^k)_{k=0}^{q-1}$ tels que φ_q^k soit l'angle entre le vecteur tangent en p_q^k et le vecteur vitesse après collision.

— `law_of_cosines` : angle d'un triangle par la loi des cosinus.

— `find_angle` : angle au sommet (puis demi-angle) en p_i pour trois impacts consécutifs.

— `q_list_phi` : angles d'incidence $\varphi_i = \frac{\pi}{2} - \frac{\alpha_i}{2}$ le long de l'orbite.

8 Construction de $\tilde{\mathscr{T}}_{\Omega,N}$

Pour toute fonction $f \in C^r_{\text{sym}}(\mathbb{T})$, la symétrie implique que son développement en série de Fourier ne contient que des termes en cosinus :

$$f(x) = a_0 + \sum_{n=1}^{\infty} a_n \cos(2\pi n x)$$

avec $a_n \in \mathbb{R}$. Ainsi, on peut identifier f à sa suite de coefficients de Fourier $(a_n)_{n\geq 0}$.

Si $f \in C^r$, on a la décroissance classique des coefficients de Fourier :$|a_n| = O\left(\frac{1}{n^r}\right)$ ce qui garantit que la suite (a_n) est dans un sous-espace bien régulier de ℓ^∞.

On peut tronquer la série de Fourier à l'ordre N :

$$f_N(x) = a_0 + \sum_{n=1}^{N} a_n \cos(2\pi n x)$$

Cela correspond à projeter f sur le sous-espace de dimension finie :

$$V_N = \mathrm{Vect}\,\{\cos(2\pi n x) \mid 0 \le n \le N\} \subset C^r_{\mathrm{sym}}(\mathbb{T})$$

On définit alors $\tilde{\mathscr{T}}_{\Omega,N}$ l'opérateur restreint et corestreint :

$$\tilde{\mathscr{T}}_{\Omega,N} := \mathscr{T}_\Omega\big|_{V_N}^{\ell^\infty_N} : V_N \longrightarrow \ell^\infty_N$$

où $\ell^\infty_N = \{(a_n)_{n\in\mathbb{N}} \in \ell^\infty \mid \forall i \ge N, a_i = 0\}$ peut s'identifier à $\mathbb{R}^N$.

Cet opérateur est linéaire, de dimension finie, et permet d'approcher $\tilde{\mathscr{T}}_\Omega$ numériquement. Étudions le spectre de $\tilde{\mathscr{T}}_{\Omega,N}$, qui est une matrice carrée, afin d'avoir des informations sur le spectre de $\tilde{\mathscr{T}}_\Omega$.

On regarde les déformations qui laissent invariant le périmètre du domaine, et qui laissent le point marqué fixe. De ce fait, $\ell_{\Omega,0} = \ell_{\Omega,1} = 0$, uniquement $\ell_{\Omega,q}$ pour $q \ge 2$ nous intéresse.

8.1 Construction de la matrice T, renormalisation

Soit, pour $q \ge 2$, l'orbite périodique marquée et symétrique de période q, de points d'impact $(x_k)_{k=0}^{q-1}$ et d'angles de réflexion $(\varphi_k)_{k=0}^{q-1}$. On évalue ensuite les fonctionnelles $\ell_{\Omega,q}$ sur une base de Fourier paire,

$$e_j(x) := \cos(2\pi j x), \qquad j = 0, 1, 2, \dots,$$

et on définit la matrice (infinie) $T = (T(q,j))_{q\ge0,\,j\ge0}$ par

$$T(q,j) \;:=\; \ell_{\Omega,q}(e_j) = \sum_{k=0}^{q-1} \cos\big(2\pi j\, x_k\big)\, \sin\varphi_k \qquad (q \ge 2). \tag{2}$$

Renormalisation : de T à $\tilde{T}$. L'idée clé consiste à absorber la dépendance principale en q des angles φ_k grâce au poids de Lazutkin. On définit pour toute fonction test u :

$$\tilde{\ell}_q(u) \;:=\; \ell_q\big(\mu^{-1}u\big), \qquad \tilde{T}u \;=\; \big(\tilde{\ell}_0(u), \tilde{\ell}_1(u), \dots\big). \tag{3}$$

Pour des domaines proches du cercle [11] pour q grand, on dispose de l'approximation asymptotique

$$\frac{\sin\varphi_k}{\mu(x_k)} \;=\; \frac{1}{q}\left(1 + \frac{\beta(k/q)}{q^2}\right) + \frac{1}{q}\,S_q\!\left(\frac{k}{q}\right) + \mathcal{O}\big(q^{-4}\big)\,, \tag{4}$$

où β est lisse et S_q est une correction oscillante de moyenne nulle. Insérée dans les équations précédentes, cette relation montre que la matrice $\tilde{T}$ est *presque diagonale* dans la base $\{e_j\}$: plus précisément,

$$\tilde{\ell}_q(e_j) \;=\; \Big(1 + \sigma_{q,0} + \frac{\beta_0}{q^2}\Big)\,\mathbf{1}_{q|j} \;+\; \frac{\tilde{\ell}_\bullet(e_j)}{q^2} \;+\; R_q(e_j), \tag{5}$$

avec $|R_q(e_j)| \lesssim q^{-3}$ et où $\mathbf{1}_{q|j}$ vaut 1 si q divise j et 0 sinon. Autrement dit, le bloc principal sélectionne les fréquences multiples de q.

Opérateur modèle. La matrice renormalisée $\tilde{T} = (\tilde{\ell}_q(e_j))_{q,j}$ est *presque diagonale par blocs* : sur la ligne q, le terme principal ne vit que sur les colonnes j telles que $q \mid j$. On isole ce squelette combinatoire par

$$(T_D)_{qj} \;:=\; \frac{1}{\pi}\,\mathbf{1}_{q|j},$$

où $\mathbf{1}_{q|j} = 1$ si $q \mid j$ et 0 sinon. Le facteur $1/\pi$ fixe une échelle naturelle (cf. le cas du cercle ci-dessous).
On définit

$$T_{\mathrm{norm}} \;:=\; T_D^{-1}\,\tilde{T}.$$

Sur la base de Fourier paire $e_j(x) = \cos(2\pi j x)$, l'asymptotique

$$\frac{\sin\varphi_k}{\mu(x_k)} \;=\; \frac{1}{q}\Big(1 + \frac{\beta(k/q)}{q^2}\Big) \;+\; \frac{1}{q}\,S_q\Big(\frac{k}{q}\Big) \;+\; \mathcal{O}\big(q^{-4}\big)\,,$$

insérée dans la définition de $\tilde{\ell}_q$, et l'orthogonalité discrète $\frac{1}{q}\sum_{k=0}^{q-1} e^{2\pi i j k/q} \;=\; \mathbf{1}_{q|j}$ donnent

$$\tilde{\ell}_q(e_j) \;=\; \Big(1 + \sigma_{q,0} + \tfrac{\beta_0}{q^2}\Big)\,\mathbf{1}_{q|j} \;+\; \frac{\tilde{\ell}_\bullet(e_j)}{q^2} \;+\; R_q(e_j), \quad |R_q(e_j)| \lesssim q^{-3}. \tag{6}$$

Autrement dit,

$$\tilde{T} \;=\; T_D\,(I + K), \qquad K := T_D^{-1}\big(\tilde{T} - T_D\big), \qquad \|K\| \text{ petit.}$$

Par conséquent,

$$T_{\mathrm{norm}} \;=\; T_D^{-1}\tilde{T} \;=\; I + K \quad \text{est } \textit{proche de l'identité.}$$

Les écarts à I capturent précisément les corrections géométriques : α (décalage des impacts), β (biais lisse des angles) et les modes $\sigma_{q,p}$ issus de S_q (partie oscillante). Pour le cercle unité, $\mu \equiv \pi$, $x_k = k/q$ et $\varphi_k = \pi/q$, de sorte que

$$\tilde{\ell}_q(e_j) = \frac{1}{q}\sum_{k=0}^{q-1} \cos\Big(2\pi j\frac{k}{q}\Big)\,\frac{\sin(\pi/q)}{\pi} = \frac{\sin(\pi/q)}{\pi}\,\mathbf{1}_{q|j}.$$

Le choix $(T_D)_{qj} = \frac{1}{\pi}\mathbf{1}_{q|j}$ aligne donc $\tilde{T}$ sur une échelle « circulaire » uniforme (indépendante de q). Avec une calibration alternative par blocs $(T_D)_{qj} = \frac{\sin(\pi/q)}{\pi}\mathbf{1}_{q|j}$, on obtiendrait exactement l'identité pour le cercle, mais on préfère ici une normalisation simple et *uniforme en* q ; l'écart résiduel est absorbé dans K et reste petit dans le régime quasi-circulaire.

Conséquences :

(1) *Analyse.* Écrire $\tilde{T} = T_D(I + K)$ avec $\|K\| \ll 1$ garantit l'inversibilité (série de Neumann), donc l'injectivité de l'opérateur isospectral linéarisé.

(2) *Numérique.* Le spectre de T_{norm} se concentre près de 1 ; la déviation $|\lambda - 1|$ mesure directement les corrections α, β, σ de l'asymptotique dans (6).

Avertissement concernant l'implémentation. Dans ce code, `T_matrix` et `T_tilde` *ne sont pas* les matrices des opérateurs T et $\tilde{T}$ du papier. Plus précisément, on a

$$\texttt{T\_matrix} = D^{-1}\,\tilde{T} \quad \text{(division ligne } q \text{ par } \sin(\pi/q)),$$

$$\texttt{T\_tilde} = T_D^{-1}\,\texttt{T\_matrix} = T_D^{-1} D^{-1}\,\tilde{T},$$

où $(T_D)_{qj} = \frac{1}{\pi}\,\mathbf{1}_{q|j}$ (avec la convention de colonnes e_j) et $D = \mathrm{diag}\big(1, \sin(\pi/2), \sin(\pi/3), \dots\big)$. Ces matrices sont des versions *calibrées/normalisées* utilisées pour améliorer la lecture numérique (proximité de l'identité) ; ne pas les confondre avec les matrices « brutes » de T ou $\tilde{T}$.

— `T_matrix` : pour chaque $q = i{+}1$, construit une orbite maximale, calcule $\xi_s = x(\theta_s)$ et φ_s, puis

$$T_{i,j} = \frac{1}{(q\sin(\pi/q))} \sum_{s=0}^{q-1} \cos\big(2\pi(j{+}1)\xi_s\big)\, \sin\varphi_s\, \mu^{-1}(\theta_s).$$

— `tilde` : forme T_D (diagonale échelle, entrées $1/\pi$ aux positions arithmétiques) et renvoie $\widetilde{T} = T_D^{-1}T$.

— `plot_mjq` : diagnostic $q^2\,T_{q,j}$ sur une colonne donnée.

— `final_spectre` : spectre de $\widetilde{T}$ (valeurs propres dans le plan complexe)

9 La structure du code

9.1 Graphe des dépendances entre fichiers

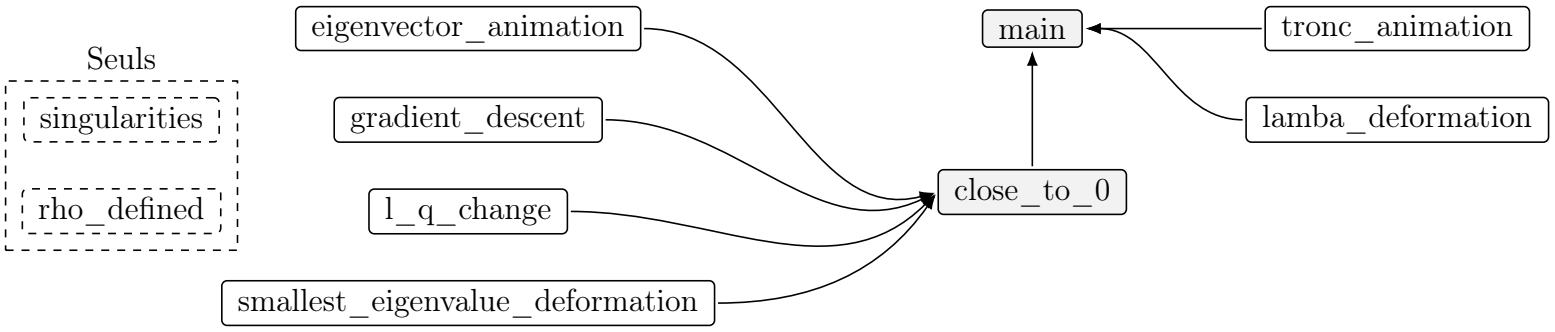

FIGURE 4 – Visualisation des relations d'importation ($A \to B$: A dépend de B).

9.2 main.py

Le code construit des orbites périodiques d'un billard dans un domaine convexe lisse, défini par sa courbure $\rho(\theta)$ (série de Fourier). La logique est : (1) générer/décrire le domaine (ρ, γ), (2) amorcer en coordonnées de Lazutkin puis maximiser la longueur polygonale pour obtenir une orbite $1/q$–périodique, (3) extraire les angles aux impacts et assembler une matrice T puis $\widetilde{T}$, (4) visualiser trajectoires et spectre.

9.3 tronc_animation.py

Ce script construit la matrice T via `main.T_matrix` à partir du vecteur $\Omega = \mathcal{A}$ avec une troncature `tronc`= 50. Il calcule puis anime, dans le plan complexe, l'évolution du spectre des sous-matrices $T_k = T_{1:k,\,1:k}$ pour $k = 1, \ldots,$ `tronc` à l'aide de `matplotlib.animation`. L'animation est exportée en `MP4` (nécessite `ffmpeg`) sous le nom `eigenvalues_animation_up_to_{tronc}.mp4` dans le dossier configuré. Un chronométrage (basé sur `time` et un `start` fourni par `main`) affiche la durée totale avant l'affichage final de la figure.

9.4 lambda_deformation.py

Ce script génère une animation montrant, côte à côte, la déformation d'un domaine Ω_λ et l'évolution du spectre de la matrice associée $\tilde{T}$. À chaque pas $\lambda \in [0, 1]$, il construit un vecteur de forme à partir de $\Omega = (1, 0, 0, 0.99)$ (les deux premiers coefficients fixés, le reste modulé par λ), trace la courbe paramétrée $t \mapsto \gamma(t, \Omega)$ (5000 points), assemble $T =$ `T_matrix`(shape, `tronc`) puis $\tilde{T} =$ `tilde`(T), calcule les valeurs propres de $\tilde{T}$ et met à jour le nuage de points dans le plan complexe. L'ensemble est animé via `matplotlib.animation` et exporté en `.mp4` (avec `ffmpeg`) ; un chronométrage affiche la durée d'exécution. Le paramètre principal est `tronc` (ici 20), qui fixe la troncature pour la construction de T.

9.5 close_to_0.py

La fonction `smallest_eigenvalues` calcule le spectre de $\tilde{T} =$ `main.tilde`(T), sélectionne les valeurs propres de module minimal et renormalise les vecteurs propres (phase fixée pour comparaison). Un vecteur propre v est interprété comme un *mode de déformation* normale du bord paramétré $\gamma(t, \Omega)$:

$$A(t) = \sum_i v_i \cos\big(2\pi(i{+}1)\, x_{\mathrm{Laz}}(t, \Omega)\big),$$

puis γ est déplacée de $\delta\, A(t)\, (-\sin t, \cos t)$ (`eigenvector_deformation` et `gamma_deformation`). L'échelle δ est choisie pour que l'amplitude maximale vaille $\varepsilon = 0{,}03$. Enfin, le script échantillonne t en 5000 points, trace la courbe originale et la courbe déformée (rouge pointillé) avec `Matplotlib`, et affiche le temps d'exécution total.

9.6 eigenvector_animation.py

Ce script anime, pour $k = 1, \ldots, n$, les coordonnées d'un vecteur propre correspondant à la valeur propre de plus petit module de la sous-matrice $\tilde{T}_k = \tilde{T}_{1:k,\,1:k}$: `close_to_0.smallest_eigenvalues` fournit ces valeurs, qui sont tracées dans le plan complexe (parties réelle et imaginaire) et mises à jour via `matplotlib.animation.FuncAnimation`. Les axes sont centrés et bornés sur $[-2, 2]$, une grille est affichée, et une option de sauvegarde en MP4 est préparée (commentée). Un chronométrage affiche le temps total d'exécution (référence `main.start`).

9.7 gradient_descent.py

Ce script cherche à *réduire* la valeur absolue de la plus petite valeur propre de $\tilde{T}_\Omega$, où T_Ω = `main.T_matrix`$(\Omega,$ `tronc`$)$ avec `tronc` $= 10$ la dimension de la troncature de l'opérateur et $\tilde{T}$ = `main.tilde`(T). Un schéma de descente de gradient `schema` basé sur un gradient aux différences finies `grad_omega` (pas $h = 10^{-3}$) met à jour $\Omega[2:] \leftarrow \Omega[2:] - \varepsilon\nabla|\lambda_{\min}|$ avec $\varepsilon = 10^{-3}$, pendant un nombre d'itérations donné. Les courbes paramétrées $t \mapsto \gamma(t, \Omega)$ et le rayon de courbure $\rho(\theta)$ (via `main.gamma` et `main.rho_function`) sont échantillonnés sur 5000 points et tracés avant/après optimisation ; la valeur finale de $|\lambda_{\min}|$ est affichée. Le bloc `__main__` lance `schema(omega,150)` depuis $\Omega = [1, 0, 0.5, -0.45, 0.3] = \mathcal{A}$ et imprime le temps total d'exécution. De plus, ce script permet d'afficher le module de la plus petite valeur propre en fonction des itérations du schéma.

9.8 l_q_change.py

Ce script visualise l'effet d'une déformation normale choisie par un critère de norme ℓ_q sur la courbe paramétrée $\gamma(t, \Omega)$. Après avoir fixé l'échelle `close_to_0.eps` $=$ 0.005, il construit (par `main`) la géométrie à partir de Ω et d'une troncature `tronc`, puis calcule une direction de déformation à l'aide de `l_q_deformation(q, T)` (coefficients dans la base de Lazutkin, pondérés par la norme ℓ_q). Pour une période q donnée, les angles d'impact initiaux sont obtenus par `find_theta0_root` (avec variantes Newton ou uniforme), raffinés par `L_maximizer`, et les points d'impact correspondants sont déduits via `x_lazutkin` et `find_impacts`. La courbe originale $t \mapsto \gamma(t, \Omega)$ et la courbe déformée $t \mapsto$ `gamma_deformation`$(t, \Omega,$ `l_q_deformation`$, \delta)$ sont tracées (axes égaux, grille, légende), ainsi que les impacts avant/après. La figure est sauvegardée en PNG sous un nom incluant Ω, q et `tronc` ; le titre rappelle ϵ, q et le temps d'exécution total.

9.9 smallest_eigenvalue_deformation.py

Ce script génère une animation de la déformation *normale* de la courbe $\gamma(t, \Omega)$ guidée par le vecteur propre associé à la plus petite valeur propre (en module) de $\tilde{T}$. Après avoir fixé $\Omega = \mathcal{A}$ et l'échelle `close_to_0.eps`$= 0.1$, il construit pour $k = 1, \ldots,$ `tronc` la sous-matrice $\tilde{T}_k$, extrait un vecteur propre minimal via `close_to_0.smallest_eigenvalues`, calcule la déformation correspondante `close_to_0.gamma_deformation`, puis met à l'échelle par δ de sorte que l'amplitude maximale vaille ε. La courbe déformée est mise à jour image par image (`matplotlib.animation.FuncAnimation`) avec un titre rappelant ε, Ω et `tronc`. L'animation est exportée en `.mp4` (via `ffmpeg`) dans un dossier local, et un chronométrage global (basé sur `main.start`) affiche le temps total d'exécution.

9.10 singularities.py

Ce script construit une courbe à deux singularités en raccordant deux arcs de cercle reliant $p_1 = (-1, 1)$ et $p_2 = (1, 1)$: l'arc supérieur (centre $(0, 1)$, rayon 1) et l'arc inférieur (centre $(0, 3)$, rayon $\sqrt{5}$). Donnés q_1, q_2, il échelonne $q_1{+}2$ et $q_2{+}2$ angles sur chaque arc pour définir des points d'impact, puis forme la trajectoire polygonale qui alterne entre les deux arcs. Les fonctions `tangent_vector` ($\tau(\theta) = (-\sin\theta, \cos\theta)$) et

`angle_between` permettent de calculer les angles d'incidence φ ; `q_list_phi_right` et `q_list_phi_left` renvoient les suites d'angles "entrant/sortant", `is_periodic` teste la condition de réflexion ($\varphi^- = \varphi^+$), et `l_q` fournit un test scalaire simple. Enfin, le script trace les deux arcs, les impacts et la trajectoire fermée (aspect `equal`, grille, légende), et affiche les pas angulaires théoriques associés à q_1 et q_2.

9.11 rho_defined.py

Ce script définit une fonction rayon de courbure $\rho(\theta)$ à morceaux, valant 1 ou ε selon les intervalles, avec raccords linéaires de largeur η autour de $\theta = \pi/4,\ 3\pi/4,\ 5\pi/4,\ 7\pi/4$ (paramètres par défaut $\varepsilon_0 = 0.002$, $\eta = 0.05$). À partir de ρ, il reconstruit la courbe $\gamma(\theta)$ en découpant l'intégrale aux points singuliers, puis calcule la paramétrisation de Lazutkin $x(\theta)$ avec normalisation $x(2\pi) = 1$. Les angles initiaux d'une orbite $1/q$ périodique sont obtenus par inversion numérique de x (interpolation dense ou Newton), raffinés par maximisation de la longueur polygonale, puis convertis en impacts et angles φ. La matrice T de taille `tronc` est assemblée avec $\mu^{-1}(\theta) = 2C_\omega\rho(\theta)^{1/3}$ et les données (ξ_s, φ_s), et $\tilde{T}$ est déduite via une matrice diagonale T_D explicite. Le script propose des visualisations (tracé de ρ, de $x(\theta)$, trajectoires maximales, spectre de $\tilde{T}$) et deux animations MP4 montrant l'évolution des trajectoires avec q et du spectre quand ε décroît de ε_0 à 0 ; des chronométrages affichent les durées d'exécution.

10 Résultats

10.1 Conséquences numériques et limites

J'ai étudié la rigidité spectrale dynamique en regardant le spectre tronqué de l'opérateur T (relié au spectre des longueurs), et observé des phénomènes de « descente » vers des valeurs propres proches de zéro. Ce comportement était souvent associé à l'apparition de **coins** ou d'irrégularités, indiquant que le domaine devenait non strictement convexe, voire non analytique. Dans ce cas, on sort du cadre du théorème de Zelditch ce qui peut expliquer la perte de rigidité.

10.2 Les tracés

Évolution conjointe du domaine et du spectre En partant du cercle ($\lambda = 0$, Fig. 5) puis en augmentant la déformation ($\lambda = 0.47$ et $\lambda = 1.0$, Fig. 6–7), le panneau gauche montre la dispersion progressive des valeurs propres de $\tilde{T}$ (troncation fixée). Au cercle, la normalisation concentre le spectre près de 1. À mesure que la forme s'écarte de la symétrie axiale parfaite, le nuage s'étale (y compris hors de l'axe réel) et certains modes se rapprochent de 0, ce qui reflète un rapprochement vers une non-injectivité, on observe que des bords plus singuliers commencent à apparaitre.

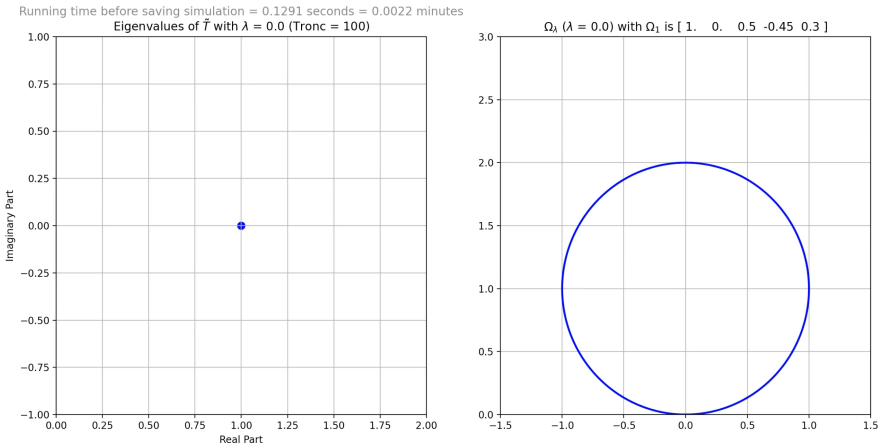

FIGURE 5 – Évolution des valeurs propres et du domaine $/Omega_\lambda$, (1)

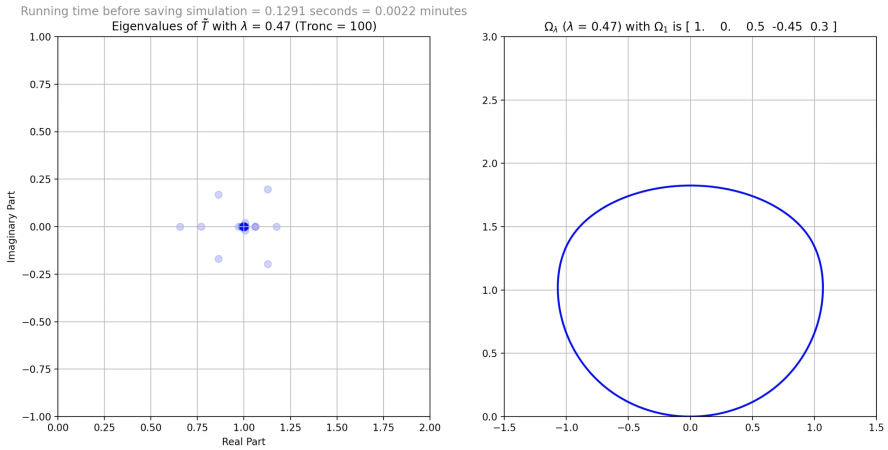

FIGURE 6 – Évolution des valeurs propres et du domaine $/Omega_\lambda$, (2)

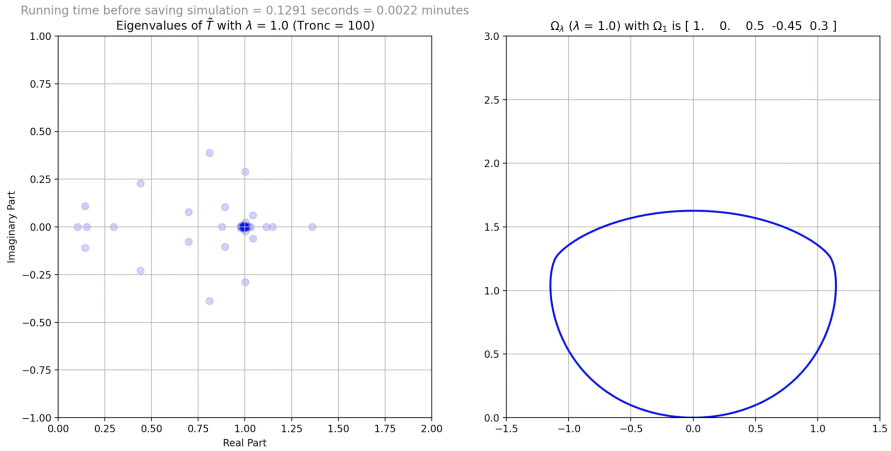

FIGURE 7 – Évolution des valeurs propres et du domaine $/Omega_\lambda$, (3)

Déformation associée à la plus petite valeur propre La courbe pointillée rouge représente la déformation normale du bord obtenue en suivant le vecteur propre correspondant à la plus petite valeur propre (amplitude ε fixée). La modification est fortement localisée dans les zones où la courbure varie le plus. Une amplification de cette déformation tendrait à générer des irrégularités (quasi-coins), en cohérence avec l'observation d'une perte de rigidité lorsque la convexité analytique se dégrade.

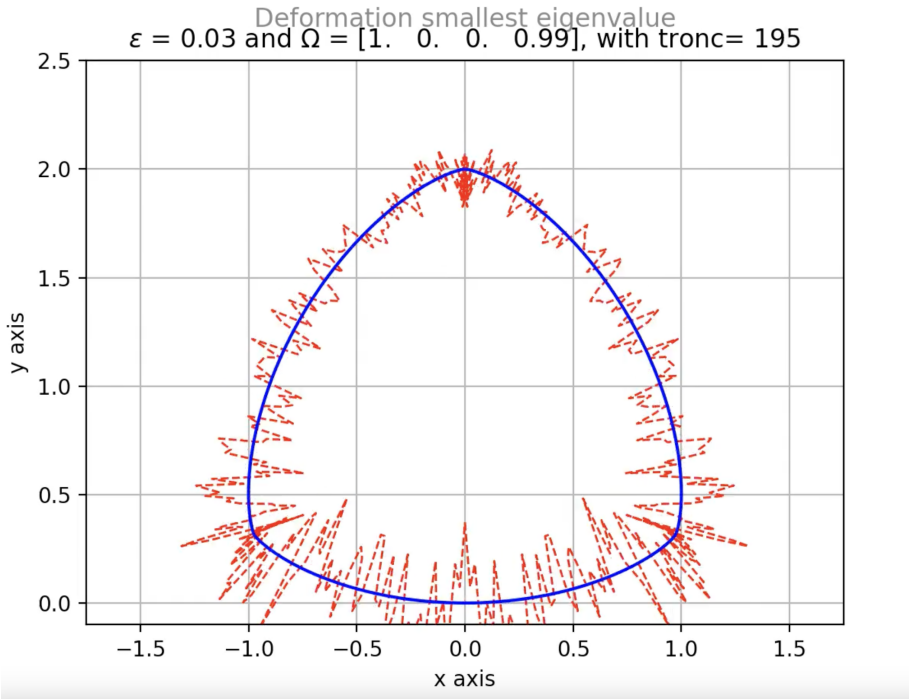

FIGURE 8 – Domaine $\mathcal{B}$, déformation de plus petite valeur propre.

Descente de gradient et redistribution de la courbure Après le schéma de descente minimisant $|\lambda_{\min}(\tilde{T})|$, la Fig. 9 compare la forme initiale et la nouvelle forme : la variation géométrique est visible mais reste modérée, on observe surtout que les "coins" initiaux de notre domaine se sont rapprochés de coins singuliers. La Fig. 10 montre l'évolution du rayon de courbure $\rho(\theta)$: la descente redistribue la courbure (accentuation/déplacement de pics et de creux) tout en conservant $\rho > 0$, confirmant que l'algorithme agit en priorité sur les zones qui ont des rayons de courbures faibles ou élevés.

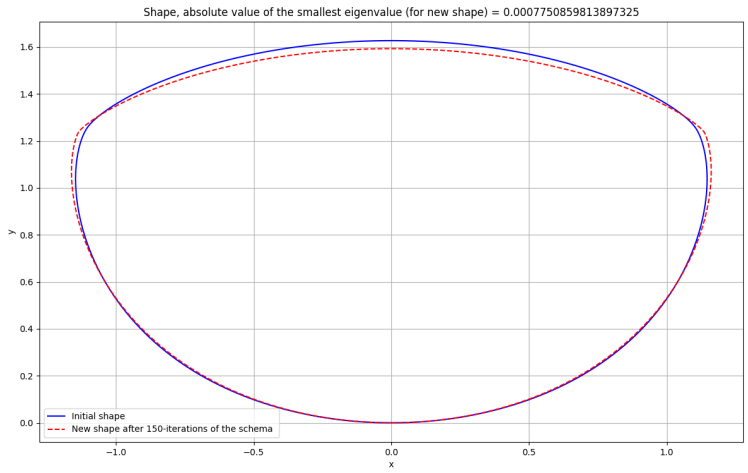

FIGURE 9 – Déformation du domaine $\Omega = \mathcal{A}$ après descente de gradient, 150 itérations.

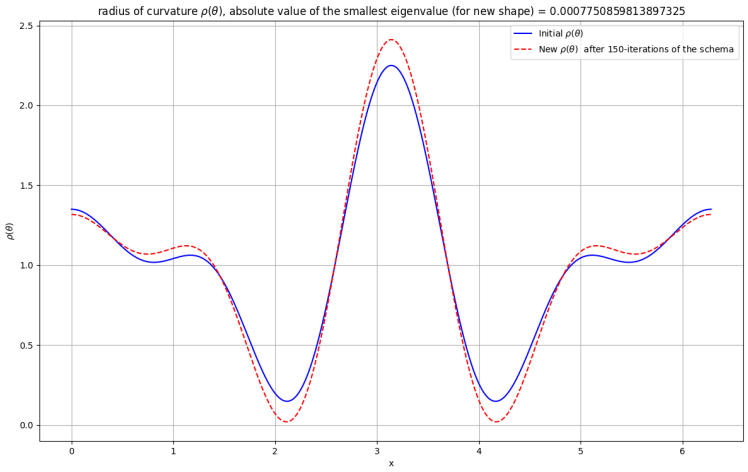

FIGURE 10 – Déformation du rayon de courbure de $\mathcal{A}$ après descente de gradient, 150 itérations.

Spectre de $\tilde{T}$ pour le domaine $\mathcal{A}$ (grande troncation) Pour une matrice 1000×1000, le spectre du domaine $\mathcal{A}$ reste majoritairement concentré près de 1 avec une dispersion limitée. Aucune valeur propre n'est numériquement indiscernable de 0 à cette échelle.

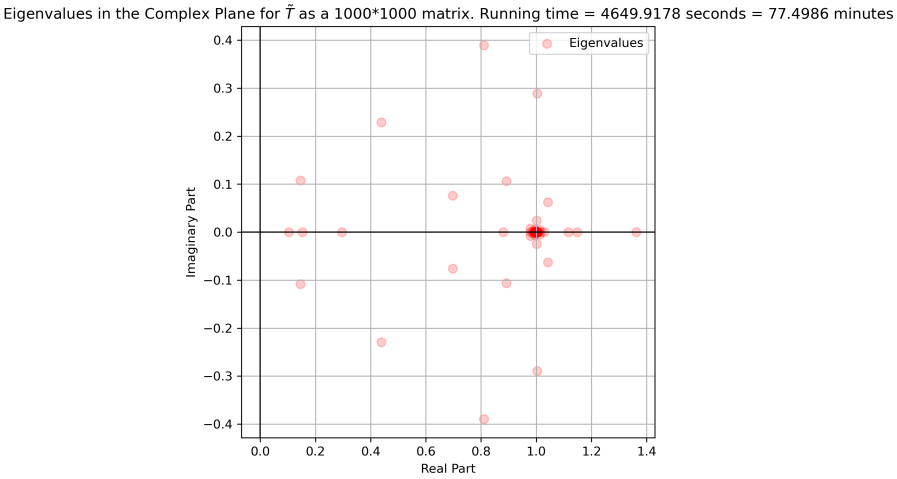

FIGURE 11 – Domaine $\mathcal{E}$

Orbites périodiques marquées $1/50$ **et** $1/200$ Les trajectoires maximales et leurs impacts épousent bien la frontière ; quand la période augmente de 50 à 200, les points d'impact se densifient le long du bord, particulièrement dans des "coins", et les segments polygonaux approchent une caustique proche de la frontière. Ceci est conforme aux coordonnées de Lazutkin (impacts quasi uniformes en x) et à l'approximation $\sin\varphi \sim 1/q$ pour q grand.

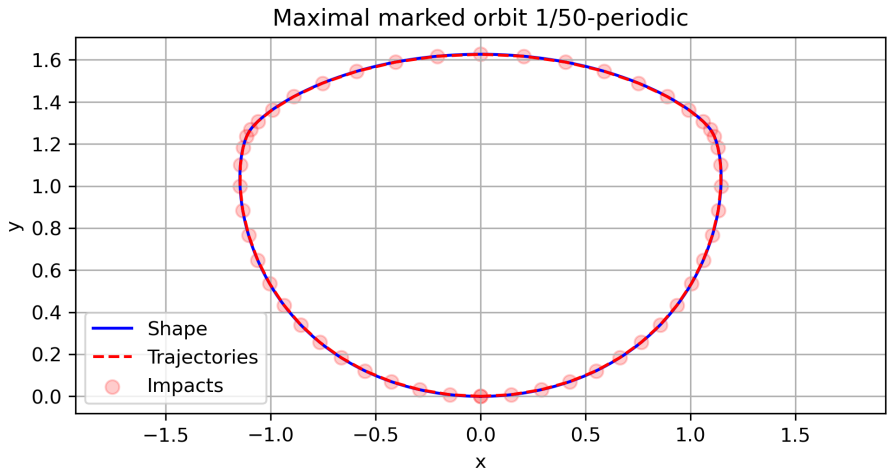

FIGURE 12 – Domaine $\mathcal{A}$, orbite périodique de période 200

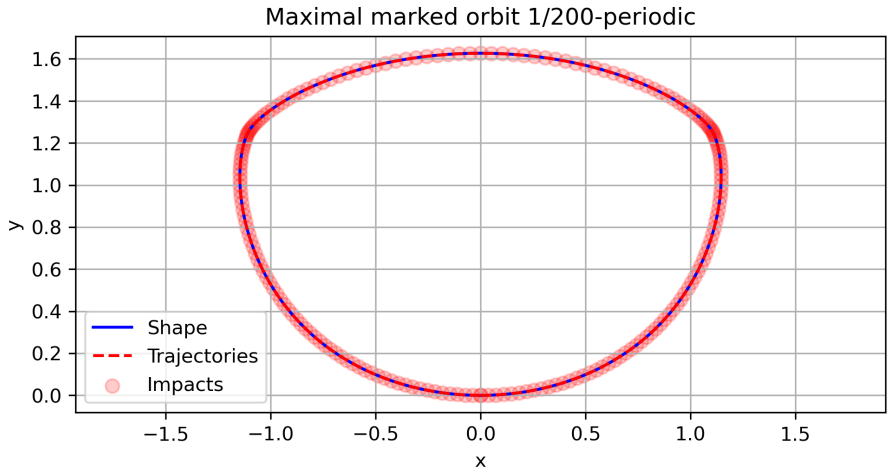

FIGURE 13 – Domaine $\mathcal{A}$, orbite périodique de période 200

10.3 Conclusion

Dans toutes les expériences menées sur des domaines réguliers et strictement convexes, le spectre de $\tilde{T}$ *reste à distance de zéro*. Les tentatives de réduction de $|\lambda_{\min}|$ conduisent à des dégradations géométriques plutôt qu'à de véritables familles isospectrales dans la classe régulière. Ces observations sont *compatibles* avec l'idée que, pour une large classe de domaines réguliers (en particulier analytiques et $\mathbb{Z}_2$-symétriques), la rigidité dynamique est typique. La perte de rigidité perçue numériquement est donc corrélée à une sortie du cadre régulier où s'appliquent les résultats

de type Zelditch. Le vecteur propre associé à la plus petite valeur propre induit une *déformation normale localisée* dans les zones à forte variation de courbure, et une descente de gradient sur $|\lambda_{\min}(\tilde{T})|$ redistribue $\rho(\theta)$ tout en conservant $\rho > 0$, sans « casser » la rigidité dans le régime régulier.

Apports principaux : Un code complet a été construit : génération de domaines par troncature de Fourier (périmètre fixé, symétrie imposée), amorce des orbites maximales $1/q$ en coordonnées de Lazutkin, calcul des angles φ_k, construction de T, renormalisation en $\tilde{T}$, puis étude du spectre (valeurs/vecteurs propres) jusqu'à de grandes troncations. Des codes permettant de visualiser différents phénomènes en jeux ont été fournis.

Néanmoins notre modèle à des limites. Premièrement, la génération par troncature de Fourier qui produit des bords analytiques ne nous permet pas d'explorer des domaines strictement C^r non-analytiques. Une deuxième limite concerne la troncature de l'opérateur $\tilde{\mathscr{T}}_\Omega$, cela nous fournit certes un bon indicateur d'injectivité mais ne constitue pas une preuve. On a enfin une limite naturelle qui survient lorsque l'on traite de problèmes numériquement, l'accumulation des erreurs et temps de calcul qui augmente.

Différentes perspectives possibles

1. Construire/tester des domaines strictement C^r non analytiques (fenêtres de régularisation, splines/B-splines, « bumps » à support compact) avec certification $\rho > 0$.
2. Passer à une analyse *en valeurs singulières* et au *pseudospectre* de $\tilde{T}_N$, avec contrôle de l'asymptotique $N \to \infty$.
3. Quantifier finement l'opérateur de perturbation K dans $\tilde{T} = T_D(I + K)$ via les corrections α, β, S_q.
4. Mettre en place des optimisations contraintes (convexité, bornes C^r) avec différentiation automatique pour rechercher, s'ils existent, des candidats non-DSR *dans la classe régulière.*

En résumé, ce travail met en place un code numérique fiable pour diagnostiquer la rigidité dynamique via $\tilde{\mathscr{T}}_\Omega$. Les simulations indiquent que, tant que l'on reste dans le cadre régulier et convexe, le spectre conserve une marge vis-à-vis de 0 ; les pertes apparentes de rigidité sont surtout liées à la dégradation de la régularité, ce qui renforce l'intuition d'une rigidité dynamique générique dans les classes considérées.

Références

[1] Mark Kac. "Can One Hear the Shape of a Drum ?" In : *The American Mathematical Monthly* 73.4P2 (1966), p. 1-23.

[2] Carolyn Gordon, David L Webb et Scott Wolpert. "One Cannot Hear the Shape of a Drum". In : *Bulletin of the American Mathematical Society* 27.1 (1992), p. 134-138.

[3] Steve Zelditch. "Inverse Spectral Problem for Analytic Domains, II : Z2-Symmetric Domains". In : *Annals of Mathematics* 170.1 (2009), p. 205-269.

[4] Brad Osgood, Ralph Phillips et Peter Sarnak. "Compact isospectral sets of surfaces". In : *Journal of functional analysis* 80.1 (1988), p. 212-234.

[5] Brad Osgood, Ralph Phillips et Peter Sarnak. "Extremals of determinants of Laplacians". In : *Journal of functional analysis* 80.1 (1988), p. 148-211.

[6] Brad Osgood, Ralph Phillips et Peter Sarnak. "Moduli space, heights and isospectral sets of plane domains". In : *Annals of Mathematics* (1989), p. 293-362.

[7] Peter Sarnak. "Determinants of Laplacians ; heights and finiteness". In : *Analysis, et cetera*. Elsevier, 1990, p. 601-622.

[8] Hamid Hezari et Steve Zelditch. "C-infinity spectral rigidity of the ellipse". In : *Anal. PDE* 5.5 (2012), p. 1105-1132.

[9] Karl G Andersson et Richard B Melrose. "The propagation of singularities along gliding rays". In : *Inventiones mathematicae* 41.3 (1977), p. 197-232.

[10] Vesselin M Petkov et Luchezar N Stoyanov. "Geometry of reflecting rays and inverse spectral problems". In : (1992).

[11] Jacopo De Simoi, Vadim Kaloshin et Qiaoling Wei. "Dynamical Spectral Rigidity Among Z2-Symmetric Strictly Convex Domains Close to a Circle". eng. In : *Annals of Mathematics* 186.1 (2017), p. 277-314. issn : 0003-486X.

[12] Artur Avila, Jacopo De Simoi et Vadim Kaloshin. *An integrable deformation of an ellipse of small eccentricity is an ellipse*. 2016. arXiv : 1412.2853 [math.DS]. url : https://arxiv.org/abs/1412.2853.

Annexes

A Scripts Python

A.1 main.py

```
import math
import numpy as np
import matplotlib.pyplot as plt
from random import*
import random
from scipy.optimize import minimize, root_scalar, newton
import scipy.integrate as integrate
import time

# Start the timer
start = time.time()

#### GENERATING A DOMAIN
#### GENERATING A DOMAIN


# Set the seed for random number generation
seed_value = 3

#Generating random sequence of Fourier coefficients normalized,
    such as sum(a2,...)<1 and ai<1/nbr_coeff
def rho_coeff(nbr_coeff):
    seed(seed_value)
    res = [nbr_coeff,0]
    if nbr_coeff <= 2:
        return [1,0]
    else:
        for i in range(nbr_coeff-2):
            n = random.randint(-10**3,10**3)/10**3
            res.append(n)
    res = [x / nbr_coeff for x in res]
    return res

#Generating the same random sequence of Fourier coefficients non
    normalized
def rho_coeff_non_normalized(nbr_coeff):
    seed(seed_value)
    res = [1,0]
    if nbr_coeff <= 2:
        return [1,0]
    else:
        for i in range(nbr_coeff-2):
            n = random.randint(-10**3,10**3)/10**3
            res.append(n)
    return res
```

```
#Defining the function rho as a troncature of a Fourier Serie
def rho_function(theta,shape):
    y = 0
    for j in range(len(shape)):
        y += shape[j]*math.cos(j*theta)
    return y

#To plot rho over [0,2*pi]
def plot_rho(shape):
    X = np.linspace(0,2*math.pi,1000)
    Y = [rho_function(x,shape)  for x in X]
    Z = np.zeros(len(X))
    plt.figure()
    plt.title(r'radius of curvature $\rho(\theta)$')
    plt.ylabel(r'$\rho(\theta)$')
    plt.xlabel('x')
    plt.plot(X,Z,)
    plt.plot(X, Y)
    plt.grid(True)
    plt.show()

#Define gamma(theta), which gives the x,y-coordinates of the
   point on ∂omega identified by the angle theta
def gamma(theta,shape):
    x = math.sin(theta)
    y = 1 - math.cos(theta)
    for i in range(2,len(shape)):
        x += (shape[i]/2)*((1/(1-i))*math.sin((1-i)*theta)
           +(1/(1+i))*math.sin((1+i)*theta))
        y += (shape[i]/2)*((-1/(1+i))*math.cos((1+i)*theta)
           +(1/(1+i))-(1/(1-i))*math.cos((1-i)*theta)+(1/(1-i)))
    return (x,y)


#### FIND THE PERIODIC TRAJECTORIES
#### FIND THE PERIODIC TRAJECTORIES


#Define euclidiean norm of a vector in R2
def norme2(v):
    res = math.sqrt(v[0]**2 + v[1]**2)
    return res

#Define function -L : -(length of the trajectory passing by the
   points ksi[i])
def minus_L(theta_vector,shape):
    res = 0
    for i in range(len(theta_vector)):
        res += norme2(np.subtract(gamma(theta_vector[i],shape),
           gamma(theta_vector[i-1],shape)))
```

```
    return -res

#Optimize function
def L_maximizer(theta_initial,shape):
    #Perform optimization by minimizing minus L
    result = minimize(minus_L,theta_initial, args=(shape,),tol=1
       e-3)
    #Get the maximum value of L by negating the minimum value of
        minus_L
    max_value = -result.fun
    max_point = result.x
    #print(f"Maximum value: {max_value} at point {max_point}")
    return np.mod(max_point, 2 * np.pi)

#Function to obtain ksi from phi
def find_impacts(theta_vector,shape):
    impacts = [gamma(theta_i,shape) for theta_i in theta_vector]
    return impacts

#Print the trajectory of the maximal marked orbit 1/q periodic
def trajectories_visualisation(impacts,shape,q):
    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([gamma(t,shape) for t in theta]).T
    impacts.append(impacts[0])
    x_vals1, y_vals1 = np.array(impacts).T
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    plt.plot(x_vals1, y_vals1, label="Trajectories", color='red'
       ,linestyle='dashed')
    plt.scatter(x_vals1, y_vals1, label="Impacts", color='red',
       marker='o', s=50,alpha=0.2)
    # Add labels and title
    plt.xlabel("x")
    plt.ylabel("y")
    #plt.title("Shape in the plan, marked point fixed at (0,0)")
    #plt.title("Maximal marked orbit 1/"+str(q)+"-periodic")
    # Add grid and legend
    plt.legend()
    plt.axis("equal")  # Ensure correct aspect ratio
    plt.grid(True)
    # Save the plot as a PNG image
    #plt.savefig(str(shape)+'_1_'+str(q)+'_periodic_trajectory.
       png',dpi=300, bbox_inches='tight')
    plt.show()
    del impacts[-1]


#### USE OF LAZUTKIN COORDINATES TO HAVE BETTER INITIAL
   CONDITIONS WHEN OPTIMIZING L
#### USE OF LAZUTKIN COORDINATES TO HAVE BETTER INITIAL
   CONDITIONS WHEN OPTIMIZING L
```

```
#Define rho to the power 1/3
def rho_function_power_1over3(theta,shape):
    return rho_function(theta,shape)**(1/3)

#Compute C_gamma
def lazutkin_parameter(shape):
    res = integrate.quad(lambda theta: rho_function_power_1over3
       (theta,shape), 0, 2*math.pi)[0]
    C_shape = (1/res)
    return C_shape

#Define the Lazutkin parametrization x(theta)
def x_lazutkin(theta,shape):
    res = lazutkin_parameter(shape) * integrate.quad(lambda
       theta_prime: rho_function_power_1over3(theta_prime,shape)
       , 0, theta)[0]
    return res

#Define dx(theta)/dtheta
def d_x_lazutkin(theta,shape):
    res = lazutkin_parameter(shape) * rho_function_power_1over3(
       theta,shape)
    return res

#Solve x(theta) = y using Newton's method with specified
   condition initial
def solve_lazutkin_newton(shape,y,initial,):
    x_solution = newton(lambda theta: x_lazutkin(theta, shape) -
        y, x0=initial, fprime=lambda theta: d_x_lazutkin(theta,
       shape), maxiter=200)
    return x_solution

#Solve x(theta) = y using root_scalar method
def solve_lazutkin_root(shape,y):
    sol = root_scalar(lambda theta: x_lazutkin(theta,shape) - y,
        method='brentq', bracket=[0, 2 * math.pi])
    return sol.root

#Find the vector theta0 by solving each component theta0j = j/q
   with Newton method
def find_theta0_newton(q,shape):
    res = []
    init = math.pi
    for i in range(q):
        y = solve_lazutkin_newton(shape,i/q,init)
        res.append(y)
        init = y
    return res
```

```
#Find the vector theta0 by solving each component theta0j = j/q
    with root_scalar method
def find_theta0_root(q,shape):
    res = []
    for i in range(q):
        y = solve_lazutkin_root(shape,i/q)
        res.append(y)
    return res


#### FIND THE ANGLES IN A TRAJECTORIES
#### FIND THE ANGLES IN A TRAJECTORIES


# Function to calculate the angle of the triangle using the law
    of cosines
def law_of_cosines(a, b, c):
    if a==0 or b==0:
        print('a ou b est nul')
    return math.acos((a**2 + b**2 - c**2) / (2 * a * b))

# Function to find the angles of the triangle given the
    coordinates of the three points
def find_angle(x1,x2,x3):
    # Calculate the lengths of the sides
    a = norme2(np.subtract(x1,x2))  # Length of side x1-x2
    b = norme2(np.subtract(x2,x3))  # Length of side x2-x3
    c = norme2(np.subtract(x3,x1))  # Length of side x3-x1
    # Use the law of cosines to find the angle at point x2
    angle = law_of_cosines(a, b, c)
    return angle/2

#Function to obtain phi from ksi
def q_list_phi(impacts):
    res = np.empty(len(impacts),dtype= object)
    for i in range(len(impacts)):
        if i < len(impacts) - 1:
            phi = math.pi/2 - find_angle(impacts[i-1],impacts[i
                ],impacts[i+1])
            res[i] = phi
        else:
            impacts.append(impacts[0])
            phi = math.pi/2 - find_angle(impacts[i-1],impacts[i
                ],impacts[i+1])
            res[i] = phi
            del impacts[-1]
    return res


### CREATION OF T AND T_TILDE
### CREATION OF T AND T_TILDE
```

```
def mu_inverse(theta,shape):
    return 2*lazutkin_parameter(shape)*rho_function_power_1over3
       (theta,shape)

# Ensure phi is a numpy array in T_matrix function and perform
   element-wise operations
def T_matrix(shape, tronc):
    matrix = np.ones((tronc, tronc))
    matrix[0, :] = mu_inverse(0,shape)
    for i in range(1,tronc):
        theta0 = find_theta0_root(i + 1, shape) # Find theta0
           using Root's method
        #theta0 = find_theta0_newton(i+1,shape) # Find theta0
           using Newton's method
        #theta0 = [(2*math.pi)*(i/q) for i in range(q)] #Uniform
            theta0
        theta_impacts = L_maximizer(theta0, shape)
        ksi = np.array([x_lazutkin(theta, shape) for theta in
           theta_impacts])  # Ensure ksi is a numpy array
        impacts = find_impacts(theta_impacts, shape)
        phi = np.array(q_list_phi(impacts))  # Ensure phi is a
           numpy array
        #trajectories_visualisation(impacts,shape,i+1)
        for j in range(tronc):
            val = 0
            for s in range(i+1):
                val += math.cos(2*math.pi*(j+1)*ksi[s])*math.sin
                   (phi[s])*mu_inverse(theta_impacts[s],shape)
            matrix[i,j] = val*(1/((i+1)*math.sin(math.pi/(i+1)))
               )
    return matrix

# Function to compute tilde(T)
def tilde(T):
    T_D = np.zeros(np.shape(T))
    for i in range(np.shape(T)[1]):
        for j in range(np.shape(T)[1] - i):
            if j % (i + 1) == 0:
                T_D[i, j + i] = 1/(np.pi)
    inverse_T_D = np.linalg.inv(T_D)
    return np.dot(inverse_T_D, T)

#Plot (q**2)*T(q,j) with j fixed
def plot_mjq(T,column):
    # Select a column index to visualize (ensuring it's within
       bounds)
    column_index = min(column, np.shape(T)[0] - 1)  # Ensuring
       the column index does not exceed matrix size
    X = np.array([i for i in range(1,np.shape(T)[0])])
```

```
    Y = [q**2*T[q,column_index,] for q in range(1,np.shape(T)
        [0])]
    plt.figure(figsize=(7, 4))
    plt.plot(X, Y, marker='o', linestyle='-', color='b', label=f
        'Column {column_index} of T')
    plt.xlabel("Row index")
    plt.ylabel("Matrix values")
    plt.title("Sample of T_matrix")
    plt.legend()
    plt.grid(True)
    plt.show()


### FUNCTIONS TO CALL
### FUNCTIONS TO CALL


def final_spectre(shape,tronc):
    T = T_matrix(shape, tronc)
    T_tilde = tilde(T)
    T_tilde_eigenvalues = np.linalg.eigvals(T_tilde)
    real_parts = np.real(T_tilde_eigenvalues)
    imag_parts = np.imag(T_tilde_eigenvalues)
    plt.figure(figsize=(6, 6))
    plt.scatter(real_parts, imag_parts, color='red', marker='o',
        label='Eigenvalues',s=50,alpha=0.2)
    plt.xlabel('Real Part')
    plt.ylabel('Imaginary Part')
    end = time.time()
    print(f'Total time with T of size {tronc} is {end - start:.4
        f} seconds')
    plt.title(f'Eigenvalues in the Complex Plane for '+r'$\tilde
        {T}$'+f' as a {tronc}*{tronc} matrix. Running time = '+f'
        {end - start:.4f} seconds'+f' = {(end - start)/60:.4f}
        minutes')
    plt.axhline(0, color='black', linewidth=1)
    plt.axvline(0, color='black', linewidth=1)
    plt.grid(True)
    plt.legend()
    #plt.savefig(str(omega)+' Eigenvalues_tronc='+str(tronc)+'.
        png',dpi=300, bbox_inches='tight')
    plt.show()

def final_trajectories(shape,q):
    theta0 = find_theta0_root(q, shape) # Find theta0 using Root
        's method
    #theta0 = find_theta0_newton(q,shape) # Find theta0 using
        Newton's method
    #theta0 = [(2*math.pi)*(i/q) for i in range(q)] #Uniform
        theta0
    theta_impacts = L_maximizer(theta0, shape)
```

```
    ksi = np.array([x_lazutkin(theta, shape) for theta in
        theta_impacts])  # Ensure ksi is a numpy array
    impacts = find_impacts(theta_impacts, shape)
    trajectories_visualisation(impacts,omega,q)


if __name__ == '__main__':
    tronc = 50
    omega = np.array([1])
    #omega = np.array([1 ,0, 0.5, -0.45, 0.3])
    #omega = np.array([1 ,0, 0, 0.99])
    #omega = np.array([1 ,0, 0, -0.999])

    #plot_rho(omega)
    final_trajectories(omega,10)
    #final_spectre(omega,tronc)

    #T = T_matrix(omega, tronc)
    #T_tilde1 = tilde(T)
    #T_tilde = np.transpose(T_tilde1)@T_tilde1

    #T_tilde_eigenvalues = np.linalg.eigvals(T_tilde)
    #real_parts = np.real(T_tilde_eigenvalues)
    #imag_parts = np.imag(T_tilde_eigenvalues)
    #plt.figure(figsize=(6, 6))
    #plt.scatter(real_parts, imag_parts, color='red', marker='o
        ', label='Eigenvalues',s=50,alpha=0.2)
    #plt.xlabel('Real Part')
    #plt.ylabel('Imaginary Part')
    #end = time.time()
    #print(f'Total time with T of size {tronc} is {end - start
        :.4f} seconds')
    #plt.title(f'Eigenvalues in the Complex Plane for '+r'$\
        tilde{T}$'+f' as a {tronc}*{tronc} matrix. Running time =
         '+f'{end - start:.4f} seconds'+f' = {(end - start)/60:.4
        f} minutes')
    #plt.axhline(0, color='black', linewidth=1)
    #plt.axvline(0, color='black', linewidth=1)
    #plt.grid(True)
    #plt.legend()
    #plt.savefig(str(omega)+' Eigenvalues_tronc='+str(tronc)+'.
        png',dpi=300, bbox_inches='tight')
    #plt.show()
```

A.2 tronc_animation.py

```
#Import the file that calculates the matrix T_tilde
import main

#Import the needed librairies
```

```
import numpy as np
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os

omega = np.array([1 ,0, 0.5, -0.45, 0.3])
#omega = np.array([1])
tronc = 50

T = main.T_matrix(omega, tronc)
T_tilde = main.tilde(T)

# Create figure
fig, ax = plt.subplots()
ax.set_xlim(-2, 2)  # Adjust limits based on expected
    eigenvalues
ax.set_ylim(-2, 2)
ax.set_title(r"Eigenvalues of $\tilde{T}_k$ (k=0)")
ax.set_xlabel("Real Part")
ax.set_ylabel("Imaginary Part")
ax.grid(True)
scatter = ax.scatter([], [], c='b', alpha=0.2,s=50)

def update(k):
    """Update function for animation"""
    T_tilde_k = T_tilde[:k, :k]  # Extract k × k submatrix
    eigenvalues = np.linalg.eigvals(T_tilde_k)  # Compute
        eigenvalues
    scatter.set_offsets(np.c_[eigenvalues.real, eigenvalues.imag
        ])  # Update scatter plot
    ax.set_title(r"Eigenvalues of $\tilde{T}_{k}$ (k="+str(k)+")
        ")
    return scatter,

# Create animation
ani = animation.FuncAnimation(fig, update, frames=range(1, T.
    shape[0]+1), interval=300)

# Make sure the folder exists
#save_folder = '/Users/..'
#os.makedirs(save_folder, exist_ok=True)

# Define the full save path
#save_path = os.path.join(save_folder, f'
    eigenvalues_animation_up_to_{tronc}.mp4')

# Save the animation as an MP4 video file
#ani.save(save_path, writer='ffmpeg', fps=30)
```

```
end = time.time()
print(f'Total time for the animation up to {tronc} eigenvalues
    is {end - main.start:.4f} seconds'+f' = {(end - main.start)
    /60:.4f} minutes')
plt.show()
```

A.3 lambda_deformation.py

```
#Import the file that calculates the matrix T_tilde
import main

#Import the needed librairies
import numpy as np
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os

tronc = 20
#omega = np.array([1])
omega = np.array([1 ,0, 0, 0.99])
theta = np.linspace(0, 2 * np.pi, 5000)

# Create the figure with 2 subplots (side by side)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))

# Eigenvalues Scatter Plot
ax1.set_xlim(0, 2)
ax1.set_ylim(-1, 1)
ax1.set_title(r"Eigenvalues of $\tilde{T}$ with lambda = 0")
ax1.set_xlabel("Real Part")
ax1.set_ylabel("Imaginary Part")
ax1.grid(True)
scatter = ax1.scatter([], [], c='b', alpha=0.2, s=50)

ax1.set_title("Simple 2D Plane with Curve")
ax1.set_xlabel("Real Part")
ax1.set_ylabel("Imaginary Part")
ax2.set_xlim(-1.5, 1.5)
ax2.set_ylim(0, 3)
ax2.grid(True)

# Initialize the curve
x_vals_c, y_vals_c = np.array([main.gamma(t,[1]) for t in theta
    ]).T
curve_line, = ax2.plot(x_vals_c, y_vals_c, color='blue', label="
    Shape", lw=2)
```

```
def update(k):
    """Update function for animation"""
    c = k/100
    temp_tail = c*omega[2:]
    temp_shape = np.concatenate(([1,0], temp_tail))
    x_vals, y_vals = np.array([main.gamma(t,temp_shape) for t in
        theta]).T
    curve_line.set_data(x_vals, y_vals)  # Show up to `k`-th
       point of the curve

    matrix = main.T_matrix(temp_shape,tronc)
    matrix_tilde = main.tilde(matrix)
    eigenvalues = np.linalg.eigvals(matrix_tilde)  # Compute
       eigenvalues
    scatter.set_offsets(np.c_[eigenvalues.real, eigenvalues.imag
       ])  # Update scatter plot
    ax1.set_title(r"Eigenvalues of $\tilde{T}$ with $\lambda$ =
       "+str(k/100)+" (Tronc = "+str(tronc)+")")
    ax2.set_title(r"$\Omega_{\lambda}$ ($\lambda$ = "+str(k/100)
       +") with $\Omega_{1}$ is "+str(omega))
    return scatter,

mid = time.time()
ax1.text(0.5, 1.05, f'Running time before saving simulation = '+
   f'{mid - main.start:.4f} seconds'+f' = {(mid - main.start)
   /60:.4f} minutes', ha='center', va='bottom', transform=ax1.
   transAxes, fontsize=12, color='gray')
# Create animation
ani = animation.FuncAnimation(fig, update, frames=range(0, 101),
    interval=100)
# Make sure the folder exists
#save_folder = '/Users/..' #to be completed
#os.makedirs(save_folder, exist_ok=True)
# Define the full save path
#save_path = os.path.join(save_folder, str(omega)+f'
   _lambda_deformation_with_tronc={tronc}.mp4')
# Save the animation as an MP4 video file
#ani.save(save_path, writer='ffmpeg', fps=30)

end = time.time()
print(f'Total time for the eigenvalues lambda-animation with
   tronc = {tronc} is {end - main.start:.4f} seconds'+f' = {(end
    - main.start)/60:.4f} minutes')
plt.show()
```

A.4 close_to_0.py

```
#Import the file that calculates the matrix T_tilde
import main
```

```
#Import the needed librairies
import numpy as np
import math
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os

eps = 0.03

def smallest_eigenvalues(A):
    eigenvalues, eigenvectors = np.linalg.eig(A)
    eigenvectors_T = eigenvectors.T
    norms = [abs(z) for z in eigenvalues]
    smallest_eigenvalue = min(norms)
    indices = [i for i, num in enumerate(norms) if num ==
       smallest_eigenvalue]
    small_values = []
    small_vectors = []
    for i in indices:
        small_values.append(eigenvalues[i])
        phase = np.angle(eigenvectors_T[i]/np.linalg.norm(
           eigenvectors_T[i][0]))  # Get angle (theta) of first
           complex entry
        small_vectors.append(np.exp(-1j*phase)*eigenvectors_T[i
           ]/np.linalg.norm(eigenvectors_T[i]))
    return small_values,small_vectors

def print_smallest_eigenvalue(A):
    small_values,  small_vectors = smallest_eigenvalues(A)
    real_parts = np.real(small_values)
    imag_parts = np.imag(small_values)
    print('Number of smallest eigenvalues = '+str(len(
       small_values)))
    print('Eigenvectors associated to each eigenvalues :')
    for i in range(len(small_vectors)):
        print('Eigenvalue '+str(small_values[i])+' with '+str(
           small_vectors[i])+' Eigenvector')
    plt.figure(figsize=(6, 6))
    plt.scatter(real_parts, imag_parts, color='blue', marker='o'
       , label='Smallest eigenvalues',s=50,alpha=0.2)
    plt.xlabel('Real Part')
    plt.ylabel('Imaginary Part')
    plt.title(f'Smallest eigenvalue(s) ('+str(len(small_values))
       +' value(s)) in the Complex Plane for '+r'$\tilde{T}$'+f'
        as a size '+str(np.shape(A)[0])+' matrix.')
    plt.axhline(0, color='black', linewidth=1)
    plt.axvline(0, color='black', linewidth=1)
    plt.grid(True)
```

```
    plt.legend()
    plt.show()

def eigenvector_deformation(eigenvector,theta,shape):
    res = 0
    for i in range(len(eigenvector)):
        res += eigenvector[i]*math.cos(2*math.pi*(i+1)*main.
           x_lazutkin(theta,shape))
    return res

def gamma_deformation(theta,shape,vector,delta):
    add1 = (eigenvector_deformation(vector,theta,shape)*-math.
       sin(theta),eigenvector_deformation(vector,theta,shape)*
       math.cos(theta))
    add = (delta*add1[0],delta*add1[1])
    res = (main.gamma(theta,shape)[0] + add[0],main.gamma(theta,
       shape)[1] + add[1])
    return res


if __name__ == '__main__':
    tronc = 20
    #omega = np.array([1])
    omega = np.array([1 ,0, 0.5, -0.45, 0.3])
    #omega = np.array([1 ,0, 0, 0.050])
    theta = np.linspace(0, 2 * np.pi, 5000)
    T = main.T_matrix(omega,tronc)
    T_tilde = main.tilde(T)
    abs_deformation_values = [abs(eigenvector_deformation(
       smallest_eigenvalues(T_tilde)[1][0],t,omega)) for t in
       theta]
    max_abs_deformation_values = max(abs_deformation_values)
    if max_abs_deformation_values != 0:
        delta = eps/max_abs_deformation_values
    else:
        print("Deformation is 0 !!!")
        delta = eps
    #smallest_eigenvalues(T_tilde)
    x_vals, y_vals = np.array([main.gamma(t,omega) for t in
       theta]).T
    x_vals1, y_vals1 = np.array([gamma_deformation(t,omega,
       smallest_eigenvalues(T_tilde)[1][0],delta) for t in theta
       ]).T
    print(np.size(np.array([gamma_deformation(t,omega,
       smallest_eigenvalues(T_tilde)[1][0],delta) for t in theta
       ]).T))
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    plt.plot(x_vals1, y_vals1, label="Deformed", color='red',
       linestyle='dashed')
    ## Add labels and title
```

```
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Deformed shape")
    # Add grid and legend
    plt.legend()
    plt.axis("equal")  # Ensure correct aspect ratio
    plt.grid(True)
    plt.show()
    end = time.time()
    print(f'Total time with T of size {tronc} is {end - main.
       start:.4f} seconds')
    #print_smallest_eigenvalue(T_tilde)
```

A.5 eigenvector_animation.py

```
import main
import close_to_0

#Import the needed librairies
import numpy as np
import math
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os
import random

omega = np.array([1 ,0, 0.5, -0.45, 0.3])
#omega = np.array([1])
tronc = 40

T = main.T_matrix(omega, tronc)
T_tilde = main.tilde(T)

# Create figure
fig, ax = plt.subplots()
ax.set_xlim(-2, 2)  # Adjust limits based on expected
   eigenvalues
ax.set_ylim(-2, 2)
ax.set_title(r"Eigenvector of smallest eigenvalue of $\tilde{T}
   _k$ (k=0)")
ax.set_xlabel("Real Part")
ax.set_ylabel("Imaginary Part")
ax.grid(True)
scatter = ax.scatter([], [], c='b', alpha=0.2,s=50)

def update(k):
    """Update function for animation"""
    T_tilde_k = T_tilde[:k, :k]  # Extract k × k submatrix
```

```
    smallest_eigenvalue = close_to_0.smallest_eigenvalues(
        T_tilde_k)[1][0]  # Compute eigenvector
    # corresponding to the smallest eigenvalue
    real_parts = [z.real for z in smallest_eigenvalue]
    imag_parts = [z.imag for z in smallest_eigenvalue]
    scatter.set_offsets(np.c_[real_parts, imag_parts])  # Update
        scatter plot
    ax.set_title(r"Eigenvector of smallest eigenvalue of $\tilde
        {T}_{k}$ (k="+str(k)+")")
    return scatter,

# Create animation
ani = animation.FuncAnimation(fig, update, frames=range(1, T.
    shape[0]+1), interval=300)

# Make sure the folder exists
#save_folder = '/Users/..' #to be completed
#os.makedirs(save_folder, exist_ok=True)

# Define the full save path
#save_path = os.path.join(save_folder, f'
    eigenvector_animation_up_to_{tronc}.mp4')

# Save the animation as an MP4 video file
#ani.save(save_path, writer='ffmpeg', fps=30)


end = time.time()
print(f'Total time for the animation up to tronc = {tronc} is {
    end - main.start:.4f} seconds'+f' = {(end - main.start)/60:.4
    f} minutes')
plt.show()
```

A.6 gradient_descent.py

```
import main
import close_to_0

import math
import numpy as np
import matplotlib.pyplot as plt
from random import*
import random
from scipy.optimize import minimize, root_scalar, newton
import scipy.integrate as integrate
import time

# Start the timer
start = time.time()
```

```
tronc = 10

def grad_omega(omega):
    h = 0.001
    res = []
    for i in range(2,len(omega)):
        T = abs(close_to_0.smallest_eigenvalues(main.tilde(main.
           T_matrix(omega,tronc)))[0][0])
        omega[i] += h
        T_h = abs(close_to_0.smallest_eigenvalues(main.tilde(
           main.T_matrix(omega,tronc)))[0][0])
        omega[i] += -h
        res.append(float(((T_h - T)/h)))
    return res

eps = 0.001
def schema(omega,iter):
    val = []
    x_iter = [i for i in range(iter)]
    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([main.gamma(t,omega) for t in
       theta]).T
    Y = [main.rho_function(x,omega)  for x in theta]
    for i in range(iter):
        print('i = '+str(i))
        d_omega = grad_omega(omega)
        omega[2:] = omega[2:] - eps*np.array(d_omega)
        val.append(abs(close_to_0.smallest_eigenvalues(main.
           tilde(main.T_matrix(omega,tronc)))[0][0]))
    x_vals_1, y_vals_1 = np.array([main.gamma(t,omega) for t in
       theta]).T
    Z = [main.rho_function(x,omega)  for x in theta]
    v = abs(close_to_0.smallest_eigenvalues(main.tilde(main.
       T_matrix(omega,tronc)))[0][0])
    print('absolute value of the smallest eigenvalue : '+str(v))
    plt.figure(figsize=(7.5, 3.5))
    plt.scatter(x_iter,val, label="Smallest eigenvalue")
    plt.title("Absolute value of the smallest eigenvalue at each
        step of the scheme")
    plt.xlabel('i')
    plt.ylabel(r"$|\lambda_{\min}|$")
    plt.grid(True)
    plt.legend()
    plt.show()
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Initial shape", color='blue'
       )
    plt.plot(x_vals_1, y_vals_1, label=f"New shape after {iter}-
       iterations of the schema ", color='red',linestyle='dashed
       ')
```

```
    plt.title('Shape, absolute value of the smallest eigenvalue
       (for new shape) = '+str(v))
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid(True)
    #plt.axis('equal')
    plt.legend()
    plt.show()
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(theta, Y, label=r"Initial $\rho(\theta)$ ", color='
       blue')
    plt.plot(theta, Z, label=r'New $\rho(\theta)$ '+f" after {
       iter}-iterations of the schema ", color='red',linestyle='
       dashed')
    plt.title(r'radius of curvature $\rho(\theta)$, absolute
       value of the smallest eigenvalue (for new shape) = '+str(
       v))
    plt.ylabel(r'$\rho(\theta)$')
    plt.xlabel('x')
    plt.grid(True)
    #plt.axis('equal')
    plt.legend()
    plt.show()
    return omega

def plot_shape(shape):
    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([main.gamma(t,shape) for t in
       theta]).T
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    plt.title('Shape')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid(True)
    plt.legend()
    plt.axis('equal')
    plt.show()


if __name__=='__main__':
    omega = np.array([1 ,0, 0.5, -0.45, 0.3])
    #omega = [1, 0, 0.1, -0.2, 0.3, -0.1, 0.1, 0.1, 0.3, 0.1]
    #omega = [1,0,0.2]
    #omega = [1, 0, 0.2, -0.75]
    #omega = [1]
    schema(omega,150)
    end = time.time()
    print('Total running time of the code :'+f'{end - start:.4f}
        seconds = {(end - start)/60:.4f} minutes')
```

A.7 l_q_change.py

```
#Import the file that calculates the matrix T_tilde
import main
import close_to_0

#Import the needed librairies
import numpy as np
import math
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os

close_to_0.eps = 0.005

def l_q_deformation(q,T):
    size = np.shape(T)[0]
    index = q - 1
    elementary_vector = np.eye(1,size,index).T
    deformation_vector = np.dot(np.linalg.inv(T),
       elementary_vector)
    #print(np.dot(T,deformation_vector))
    res = deformation_vector.flatten().tolist()
    return res

if __name__ == '__main__':
    tronc = 20
    q = 4
    #omega = np.array([1])
    omega = np.array([1 ,0, 0.5, -0.45, 0.3])
    #omega = np.array([1 ,0, 0, 0.99])
    theta = np.linspace(0, 2 * np.pi, 500)
    T = main.T_matrix(omega,tronc)
    #print(T)
    T_tilde = main.tilde(T)

    abs_deformation_values = [abs(close_to_0.
       eigenvector_deformation(close_to_0.smallest_eigenvalues(
       T_tilde)[1][0],t,omega)) for t in theta]
    max_abs_deformation_values = max(abs_deformation_values)
    if max_abs_deformation_values != 0:
        delta = close_to_0.eps/max_abs_deformation_values
    else:
        print("Deformation is 0 !!!")
        delta = close_to_0.eps

    theta0 = main.find_theta0_root(q, omega) # Find theta0 using
        Root's method
```

```
    #theta0 = main.find_theta0_newton(q,omega) # Find theta0
       using Newton's method
    #theta0 = [(2*math.pi)*(i/q) for i in range(q)] #Uniform
       theta0
    theta_impacts = main.L_maximizer(theta0, omega)
    ksi = np.array([main.x_lazutkin(theta, omega) for theta in
       theta_impacts])  # Ensure ksi is a numpy array
    impacts = main.find_impacts(theta_impacts, omega)

    impacts.append(impacts[0])
    x_vals2, y_vals2 = np.array(impacts).T
    plt.figure(figsize=(7.5, 3.5))
    x_vals, y_vals = np.array([main.gamma(t,omega) for t in
       theta]).T
    x_vals1, y_vals1 = np.array([close_to_0.gamma_deformation(t,
       omega,l_q_deformation(q,T),delta) for t in theta]).T
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    plt.plot(x_vals1, y_vals1, label="Deformed shape", color='
       red',linestyle='dashed')
    plt.plot(x_vals2, y_vals2, label="1/q Trajectory", color='
       red',linestyle='-', alpha = 0.6)
    plt.scatter(x_vals2, y_vals2, label="1/q Impacts", color='
       red', marker='o', s=50, alpha = 0.6)
    del impacts[-1]

    for j in range(max(2,q-2),q+2):
        if j == q:
            pass
        else:
            theta0_2 = main.find_theta0_root(j, omega) # Find
               theta0 using Root's method
            #theta0_2 = main.find_theta0_newton(j,omega) # Find
               theta0 using Newton's method
            #theta0_2 = [(2*math.pi)*(i/q) for i in range(j)] #
               Uniform theta0
            theta_impacts_2 = main.L_maximizer(theta0_2, omega)
            ksi_2 = np.array([main.x_lazutkin(theta, omega) for
               theta in theta_impacts_2])  # Ensure ksi is a
               numpy array
            impacts_2 = main.find_impacts(theta_impacts_2, omega
               )
            impacts_2.append(impacts_2[0])
            x_valsj, y_valsj = np.array(impacts_2).T
            plt.plot(x_valsj, y_valsj, color='green',linestyle='
               -',alpha = 0.2)
            plt.scatter(x_valsj, y_valsj, color='green', marker=
               'o', s=50,alpha=0.2)
            del impacts_2[-1]

    ## Add labels and title
    plt.xlabel("x")
```

```
    plt.ylabel("y")
    ell = '\u2113'  # ℓ with subscript 1
    end = time.time()
    plt.title("Tronc ="+str(tronc)+r',  $\epsilon$ ='+str(
        close_to_0.eps)+", "+r'$\ell_{q}$'+" deformation for q="+
        str(q)+'. Running time'f' = {(end - main.start)/60:.4f}
        min')
    # Add grid and legend
    plt.legend()
    plt.axis("equal")  # Ensure correct aspect ratio
    plt.grid(True)
    plt.savefig(str(omega)+'_l_q_changes_q='+str(q)+'_tronc='+
        str(tronc)+'.png',dpi=300, bbox_inches='tight')
    plt.show()
    print(f'Total time with T of size {tronc} is {end - main.
        start:.4f} seconds')
```

A.8 smallest_eigenvalue_deformation.py

```
#Import the file that calculates the matrix T_tilde
import main
import close_to_0

#Import the needed librairies
import numpy as np
import matplotlib.pyplot as plt
from random import seed, randint
import time
import matplotlib.animation as animation
import os

close_to_0.eps = 0.1

omega = np.array([1,0,0.5,-0.45,0.3])
#omega = np.array([1,0,0,0.99])
#omega = np.array([1])
tronc = 50

T = main.T_matrix(omega, tronc)
T_tilde = main.tilde(T)
theta = np.linspace(0, 2 * np.pi, 500)

# Create figure
fig, ax = plt.subplots()
ax.set_ylim(-0.1, 2.5)
ax.set_xlim(-1.75, 1.75)
ax.set_title(r'$\epsilon$ = '+str(close_to_0.eps)+r" and $\
    Omega$ = "+str(omega)+", with tronc= 0")
ax.set_xlabel("x axis")
ax.set_ylabel("y axis")
```

```
ax.grid(True)
x_vals, y_vals = np.array([main.gamma(t,omega) for t in theta]).
    T
initial_line, = ax.plot(x_vals, y_vals, 'b-', lw=1, label="
    Original Curve")
deformed_line, = ax.plot(x_vals, y_vals, 'r--', lw=1, label="
    Deformation")
curve_line, = ax.plot(x_vals, y_vals, c='b')

def update(k):
    """Update function for animation"""
    T_tilde_k = T_tilde[:k, :k]  # Extract k × k submatrix
    abs_deformation_values = [abs(close_to_0.
        eigenvector_deformation(close_to_0.smallest_eigenvalues(
        T_tilde_k)[1][0],t,omega)) for t in theta]
    max_abs_deformation_values = max(abs_deformation_values)
    if max_abs_deformation_values != 0:
        delta = close_to_0.eps/max_abs_deformation_values
    else:
        print("Deformation is 0 !!!")
        delta = close_to_0.eps
    x_vals1, y_vals1 = np.array([close_to_0.gamma_deformation(t,
        omega,close_to_0.smallest_eigenvalues(T_tilde_k)[1][0],
        delta) for t in theta]).T
    deformed_line.set_data(x_vals1, y_vals1)  # Update the plot
    ax.set_title(r'$\epsilon$ = '+str(close_to_0.eps)+r" and $\
        Omega$ = "+str(omega)+", with tronc= "+str(k))
    return deformed_line,

ax.text(0.5, 1.05, 'Deformation smallest eigenvalue', ha='center
    ', va='bottom', transform=ax.transAxes, fontsize=12, color='
    gray')
# Create animation
ani = animation.FuncAnimation(fig, update, frames=range(1, T.
    shape[0]+1), interval=1000)

# Make sure the folder exists
#save_folder = '/Users/..' #to be completed
#os.makedirs(save_folder, exist_ok=True)

# Define the full save path
#save_path = os.path.join(save_folder, str(omega)+'
    _Smallest_eigenvalue_animation up to'+str(tronc)+'.mp4')

# Save the animation as an MP4 video file
#ani.save(save_path, writer='ffmpeg', fps=30)


end = time.time()
print(f'Total time for the animation of the deformation up to
    tronc = {tronc} is {end - main.start:.4f} seconds'+f' = {(end
```

```
    - main.start)/60:.4f} minutes')
plt.show()
```

A.9 rho_defined.py

```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
from scipy.optimize import minimize, root_scalar, newton
import matplotlib.animation as animation
import os
import time
from scipy.interpolate import interp1d

# Start the timer
start = time.time()

eps0 = 0.002
eps = eps0
#eps = 0.002
eta = 0.05


def rho_defined(theta):
    if 0 <= theta < math.pi/4 - eta:
        return 1
    elif math.pi/4 - eta <= theta < math.pi/4 + eta:
        y  = math.pi/4
        phi = (y + eta)/(y - eta)
        b = (eps - phi)/(1 - phi)
        a = (1 - b)/(y - eta)
        return a*theta + b
    elif math.pi/4 + eta <= theta < 3*math.pi/4 - eta:
        return eps
    elif 3*math.pi/4 - eta <= theta < 3*math.pi/4 + eta:
        y = 3*math.pi/4
        phi = (y + eta)/(y - eta)
        b = (eps - 1/phi)/(1-(1/phi))
        a = (1-b)/(y + eta)
        return a*theta + b
    elif 3*math.pi/4 + eta <= theta < 5*math.pi/4 - eta:
        return 1
    elif 5*math.pi/4 - eta <= theta < 5*math.pi/4 + eta:
        y  = 5*math.pi/4
        phi = (y + eta)/(y - eta)
        b = (eps - phi)/(1-phi)
        a = (1-b)/(y - eta)
        return a*theta + b
    elif 5*math.pi/4 + eta <= theta < 7*math.pi/4 - eta:
```

```
        return eps
    elif 7*math.pi/4 - eta <= theta < 7*math.pi/4 + eta:
        y  = 7*math.pi/4
        phi = (y + eta)/(y - eta)
        b = (eps - 1/phi)/(1-(1/phi))
        a = (1-b)/(y + eta)
        return a*theta + b
    elif 7*math.pi/4 + eta <= theta <= 2*math.pi:
        return 1
    else:
        return eps

def plot_rho_defined():
    X = np.linspace(0,2*math.pi,10000)
    Y = [rho_defined(theta) for theta in X]
    Y3 = [0 for i in range(len(X))]
    plt.axvline(x=math.pi/4, color='r', linestyle='--', label=r'
       x = $\pi$/4')
    plt.axvline(x=3*math.pi/4, color='r', linestyle='--', label=
       r'x = $3\pi$/4')
    plt.axvline(x=5*math.pi/4, color='r', linestyle='--', label=
       r'x = $5\pi$/4')
    plt.axvline(x=7*math.pi/4, color='r', linestyle='--', label=
       r'x = $7\pi$/4')
    plt.plot(X, Y, label=r'$\rho(\theta)$')
    plt.plot(X, Y3, label=r'0')
    plt.grid(True)
    plt.title(r'$\rho(\theta)$ with '+r'$\epsilon,\eta$ = '+f'{
       eps0},{eta}')
    plt.xlabel(r'$\theta$')
    plt.ylabel(r'$\rho(\theta)$')
    plt.legend()
    plt.show()

def plot_fun(f1,f2,theta_vector):
    print('je suis rentré')
    X = np.linspace(0,2*math.pi,10000)
    Y1 = [f1(theta) for theta in X]
    Y2 = [f2(theta) for theta in X]
    # Create the figure with 2 subplots (side by side)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))
    ax1.set_title(r"function $f_x$")
    ax1.set_xlabel(r"$\theta$")
    ax1.set_ylabel(r"$\rho(\theta)\sin(\theta)$")
    ax1.grid(True)
    ax1.plot(X, Y1, label=r"$f_x(\theta)$", color='blue')
    for i in range(len(theta_vector)):
        ax1.axvline(x=theta_vector[i], color='r', linestyle='--'
           , label=r'x = '+str(theta_vector[i]))
    ax1.legend()
```

```
    ax2.set_title(r"function $f_y$")
    ax2.set_xlabel(r"$\theta$")
    ax2.set_ylabel(r"$\rho(\theta)\cos(\theta$)")
    ax2.grid(True)
    ax2.plot(X, Y2, label=r"$f_y(\theta)$", color='blue')
    for i in range(len(theta_vector)):
        ax2.axvline(x=theta_vector[i], color='r', linestyle='--'
           , label=r'x = '+str(theta_vector[i]))
    ax2.legend()
    plt.show()

def gamma_defined_2(theta):
    # Define your function
    fx = lambda u: rho_defined(u)*math.cos(u)
    fy = lambda u: rho_defined(u)*math.sin(u)
    # Integration limits
    a = 0
    b = theta
     # Discontinuity points within [0, 2pi]
    discontinuities = [
        math.pi/4 - eta, math.pi/4 + eta,
        3*math.pi/4 - eta, 3*math.pi/4 + eta,
        5*math.pi/4 - eta, 5*math.pi/4 + eta,
        7*math.pi/4 - eta, 7*math.pi/4 + eta
    ]

    # Filter points actually within [a, b]
    split_points = [p for p in discontinuities if a < p < b]

    # Compute definite integral
    x, errorx = quad(fx, a, b, points=split_points)
    y, errory = quad(fy, a, b, points=split_points)
    #print(r"Integral of $\rho$ from 0 to "+"{theta}"+" = {
       result}")
    return (x,y)

def gamma_defined(theta):
    # Function to integrate
    fx = lambda u: rho_defined(u) * math.cos(u)
    fy = lambda u: rho_defined(u) * math.sin(u)

    # Define key discontinuity points
    discontinuities = [
        math.pi/4 - eta, math.pi/4 + eta,
        3*math.pi/4 - eta, 3*math.pi/4 + eta,
        5*math.pi/4 - eta, 5*math.pi/4 + eta,
        7*math.pi/4 - eta, 7*math.pi/4 + eta
    ]

    # Include boundaries 0 and theta
```

```
    all_points = [0] + [p for p in discontinuities if 0 < p <=
        theta] + [theta]
    all_points = sorted(set(all_points))

    # Integrate over subintervals
    total_x = 0
    total_y = 0
    for i in range(len(all_points) - 1):
        a = all_points[i]
        b = all_points[i + 1]
        total_x += quad(fx, a, b, limit=500)[0]
        total_y += quad(fy, a, b, limit=500)[0]

    return (total_x, total_y)


def plot_shape():
    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([gamma_defined(t) for t in theta])
        .T
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    # Add labels and title
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Desired shape")
    # Add grid and legend
    plt.legend()
    plt.axis("equal")  # Ensure correct aspect ratio
    plt.grid(True)
    plt.show()

def f_x(theta):
    y = rho_defined(theta)
    res = y*math.cos(theta)
    return res

def f_y(theta):
    y = rho_defined(theta)
    res = y*math.sin(theta)
    return res

#Define euclidiean norm of a vector in R2
def norme2(v):
    res = math.sqrt(v[0]**2 + v[1]**2)
    return res

#Define function -L : -(length of the trajectory passing by the
    points ksi[i])
def minus_L(theta_vector):
    res = 0
```

```
    for i in range(len(theta_vector)):
        res += norme2(np.subtract(gamma_defined(theta_vector[i])
           ,gamma_defined(theta_vector[i-1])))
    return -res


#Optimize function
def L_maximizer(theta_initial):
    #Perform optimization by minimizing minus L
    result = minimize(minus_L,theta_initial,tol=1e-3)
    #Get the maximum value of L by negating the minimum value of
        minus_L
    max_value = -result.fun
    max_point = result.x
    #print(f"Maximum value: {max_value} at point {max_point}")
    return np.mod(max_point, 2 * np.pi)

#Define rho to the power 1/3
def rho_function_power_1over3(theta):
    return rho_defined(theta)**(1/3)

#Compute C_gamma
def lazutkin_parameter():
    # Function to integrate
    f = lambda u: rho_function_power_1over3(u)

    # Define key discontinuity points
    discontinuities = [
        math.pi/4 - eta, math.pi/4 + eta,
        3*math.pi/4 - eta, 3*math.pi/4 + eta,
        5*math.pi/4 - eta, 5*math.pi/4 + eta,
        7*math.pi/4 - eta, 7*math.pi/4 + eta
    ]

    # Include boundaries 0 and theta
    all_points = [0] + [p for p in discontinuities] + [2*math.pi
       ]
    all_points = sorted(all_points)

    # Integrate over subintervals
    total = 0
    for i in range(len(all_points) - 1):
        a = all_points[i]
        b = all_points[i + 1]
        total += quad(f, a, b, limit=500)[0]
    res = (1/total)
    return res

C_omega = lazutkin_parameter()

#Define the Lazutkin parametrization x(theta)
```

```
def x_lazutkin(theta):
    # Function to integrate
    f = lambda u: rho_function_power_1over3(u)
    # Define key discontinuity points
    discontinuities = [
        math.pi/4 - eta, math.pi/4 + eta,
        3*math.pi/4 - eta, 3*math.pi/4 + eta,
        5*math.pi/4 - eta, 5*math.pi/4 + eta,
        7*math.pi/4 - eta, 7*math.pi/4 + eta
    ]

    # Include boundaries 0 and theta
    all_points = [0] + [p for p in discontinuities if 0 < p <=
       theta] + [theta]
    all_points = sorted(all_points)

    # Integrate over subintervals
    total = 0
    for i in range(len(all_points) - 1):
        a = all_points[i]
        b = all_points[i + 1]
        total += quad(f, a, b, limit=500)[0]
    res = C_omega * total
    return res


def plot_lazutkin():
    X = np.linspace(0,2*math.pi,10000)
    Y = [x_lazutkin(theta) for theta in X]
    plt.axvline(x=math.pi/4, color='r', linestyle='--', label=r'
       x = $\pi$/4')
    plt.axvline(x=3*math.pi/4, color='r', linestyle='--', label=
       r'x = $3\pi$/4')
    plt.axvline(x=5*math.pi/4, color='r', linestyle='--', label=
       r'x = $5\pi$/4')
    plt.axvline(x=7*math.pi/4, color='r', linestyle='--', label=
       r'x = $7\pi$/4')
    plt.plot(X, Y, label=r'$\rho(\theta)$')
    plt.grid(True)
    plt.title(r'Lazutkin coordinates $x(\theta)$')
    plt.xlabel(r'$\theta$')
    plt.ylabel(r'$x(\theta)$')
    plt.legend()
    plt.show()

#Define dx(theta)/dtheta
def d_x_lazutkin(theta):
    res = C_omega * rho_function_power_1over3(theta)
    return res

# Generate a dense grid of theta values
```

```
theta_grid = np.linspace(0, 2 * np.pi, 100000)
# Compute x(theta) over the grid
x_grid = np.array([x_lazutkin(t) for t in theta_grid])
# Create interpolator to estimate theta from x
inverse_lazutkin = interp1d(x_grid, theta_grid, bounds_error=
   False, fill_value="extrapolate")

def solve_lazutkin_interpolation(y):
    return float(inverse_lazutkin(y))

def find_theta0_interpolation(q):
    res = []
    for i in range(q):
        y = solve_lazutkin_interpolation(i/q)
        res.append(y)
    return res

#Solve x(theta) = y using Newton's method with specified
   condition initial
def solve_lazutkin_newton(y,initial,):
    #print(f"Solving x_lazutkin(theta) = {y} from initial = {
       initial}")
    x_solution = newton(lambda theta: x_lazutkin(theta) - y, x0=
       initial, fprime=lambda theta: d_x_lazutkin(theta),
       maxiter=200)
    return x_solution

#Find the vector theta0 by solving each component theta0j = j/q
   with Newton method
def find_theta0_newton(q):
    res = []
    init = 0
    for i in range(q):
        y = solve_lazutkin_newton(i/q,init)
        res.append(y)
        init = y
    return res

#Function to obtain ksi from phi
def find_impacts(theta_vector):
    impacts = [gamma_defined(theta_i) for theta_i in
       theta_vector]
    return impacts

#Print the trajectory of the maximal marked orbit 1/q periodic
def trajectory_visualisation(q):
    start_visu = time.time()
    #theta_init = [(2 * math.pi) * (i / q) for i in range(q)]
    #theta_init = find_theta0_newton(q)
    theta_init = find_theta0_interpolation(q)
    theta_vector = L_maximizer(theta_init)
```

```
    impacts = find_impacts(theta_vector)
    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([gamma_defined(t) for t in theta])
        .T
    impacts.append(impacts[0])
    x_vals1, y_vals1 = np.array(impacts).T
    plt.figure(figsize=(7.5, 3.5))
    plt.plot(x_vals, y_vals, label="Shape", color='blue')
    plt.plot(x_vals1, y_vals1, label="Trajectories", color='red'
        ,linestyle='dashed')
    plt.scatter(x_vals1, y_vals1, label="Impacts", color='red',
        marker='o', s=50,alpha=0.2)
    # Add labels and title
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Maximal marked orbit 1/"+str(q)+"-periodic")
    # Add grid and legend
    plt.legend()
    plt.axis("equal")  # Ensure correct aspect ratio
    plt.grid(True)
    # Save the plot as a PNG image
    #plt.savefig(str(shape)+'_1_'+str(q)+'_periodic_trajectory.
        png',dpi=300, bbox_inches='tight')
    mid_visu = time.time()
    print('Running time for plotting trajectory = '+f'{mid_visu
        - start_visu:.4f} seconds'+f' = {(mid_visu - start_visu)
        /60:.4f} minutes'+' (q = '+str(q)+')')
    plt.show()
    del impacts[-1]

def animation_trajectory(q_max):
    start_traj = time.time()
        # Create figure
    fig, ax = plt.subplots()
    ax.set_title(r"Maximal marked orbit 1/"+str(2)+"-periodic")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.grid(True)
    scatter = ax.scatter([], [], c='b', alpha=0.2,s=50)

    theta = np.linspace(0, 2 * np.pi, 5000)
    x_vals, y_vals = np.array([gamma_defined(t) for t in theta])
        .T
    curve_line, = ax.plot(x_vals, y_vals, color='blue', label="
        Shape", lw=2)

    # Initialize red dashed line for trajectories
    trajectory_line, = ax.plot([], [], 'r--', label="
        Trajectories", alpha=0.6)

    def update(q):
```

```
        # Compute trajectory points
        #theta_init = [(2 * math.pi) * (i / q) for i in range(q)
            ]
        #theta_init = find_theta0_newton(q)
        theta_init = find_theta0_interpolation(q)
        theta_vector = L_maximizer(theta_init)
        trajectories = find_impacts(theta_vector)
        trajectories.append(trajectories[0])  # Close the loop

        x_vals1, y_vals1 = np.array(trajectories).T

        # Update shape (still blue)
        curve_line.set_data(x_vals, y_vals)

        # Update red dotted line
        trajectory_line.set_data(x_vals1, y_vals1)
        trajectory_line.set_color('red')
        trajectory_line.set_linestyle('--')

        # Update red scatter points
        scatter.set_offsets(np.c_[x_vals1, y_vals1])
        scatter.set_color('red')  # Points are now red

        ax.set_title(f"Maximal marked orbit 1/{q}-periodic")
        del trajectories[-1]
            # Autoscale based on new trajectory data
        ax.relim()            # Recalculate limits based on
            artists
        ax.autoscale_view()   # Apply the new limits


        return scatter, trajectory_line
    # Create animation
    ani = animation.FuncAnimation(fig, update, frames=range(2,
        q_max+1), interval=400,blit=False)
    # Show the animation in a live window
    #plt.show()
    # Make sure the folder exists
    save_folder = '/Users/romainhoang/Desktop/Research S8/Work/
        Code results'
    os.makedirs(save_folder, exist_ok=True)
    # Define the full save path
    save_path = os.path.join(save_folder, f'
        trajectories_evolution_defined_up_to_{q_max}.mp4')
    # Save the animation as an MP4 video file
    ani.save(save_path, writer='ffmpeg', fps=8)
    end_traj = time.time()
    print('Running time for creating and saving trajectory
        animation = '+f'{end_traj - start_traj:.4f} seconds'+f' =
         {(end_traj - start_traj)/60:.4f} minutes'+' (q_max = '+
        str(q_max)+')')
```

```
def animation_eigenvalues(tronc):
    start_eig = time.time()
    global eps
    # Create a single subplot for eigenvalues
    fig, ax1 = plt.subplots(figsize=(7, 7))

    # Eigenvalues Scatter Plot
    ax1.set_xlim(-0.5, 2)
    ax1.set_ylim(-1, 1)
    ax1.set_title(r"Eigenvalues of $\tilde{T}$ with eps = {eps0}
       ")
    ax1.set_xlabel("Real Part")
    ax1.set_ylabel("Imaginary Part")
    ax1.grid(True)
    scatter = ax1.scatter([], [], c='b', alpha=0.2, s=50)

    def update(k):
        """Update function for animation"""
        global eps, C_omega
        eps = eps0 * (1 - k / 100)
        C_omega = lazutkin_parameter()

        matrix = T_matrix(tronc)
        matrix_tilde = tilde(matrix)
        eigenvalues = np.linalg.eigvals(matrix_tilde)
        scatter.set_offsets(np.c_[eigenvalues.real, eigenvalues.
           imag])
        ax1.set_title(r"Eigenvalues of $\tilde{T}$ with $\
           epsilon$ = "+str(eps)+" (Tronc = "+str(tronc)+")")
        return scatter,
    mid_eig = time.time()
    ax1.text(0.5, 1.05, f'Running time before saving the
       eigenvalues eps-animation = '+f'{mid_eig - start_eig:.4f}
        seconds'+f' = {(mid_eig - start_eig)/60:.4f} minutes',
       ha='center', va='bottom', transform=ax1.transAxes,
       fontsize=12, color='gray')

    # Create animation
    ani = animation.FuncAnimation(fig, update, frames=range(0,
       100), interval=100)

    # Save animation
    save_folder = '/Users/romainhoang/Desktop/Research S8/Work/
       Code results'
    os.makedirs(save_folder, exist_ok=True)
    save_path = os.path.join(save_folder, str(eps)+f'
       _eps_animation_with_tronc={tronc}_eigenvalues_only.mp4')
    ani.save(save_path, writer='ffmpeg', fps=30)
```

```
    end_eig = time.time()
    print(f'Running time for creating and saving the eigenvalues
        eps-animation is {end_eig - start_eig:.4f} seconds = {(
       end_eig - start_eig)/60:.4f} minutes'+' (tronc = '+str(
       tronc)+')')
    eps = eps0
    #plt.show()


def plot_theta_impacts(impacts):
    X = np.linspace(0,2*math.pi,10000)
    Y = [rho_defined(theta) for theta in X]
    for i in range(len(impacts)):
        plt.axvline(x=impacts[i], color='r', linestyle='--',
           label=r'impacts['+str(i)+'] = '+str(impacts[i]))
    plt.plot(X, Y, label=r'$\rho(\theta)$')
    plt.grid(True)
    plt.title(r'$\rho(\theta)$ in rectangular form')
    plt.xlabel(r'$\theta$')
    plt.ylabel(r'$\rho(\theta)$')
    plt.legend()
    plt.show()


#### FIND THE ANGLES IN A TRAJECTORIES
#### FIND THE ANGLES IN A TRAJECTORIES


# Function to calculate the angle of the triangle using the law
   of cosines
def law_of_cosines(a, b, c):
    if a==0 or b==0:
        print('a ou b est nul')
    return math.acos((a**2 + b**2 - c**2) / (2 * a * b))

# Function to find the angles of the triangle given the
   coordinates of the three points
def find_angle(x1,x2,x3):
    # Calculate the lengths of the sides
    a = norme2(np.subtract(x1,x2))  # Length of side x1-x2
    b = norme2(np.subtract(x2,x3))  # Length of side x2-x3
    c = norme2(np.subtract(x3,x1))  # Length of side x3-x1
    # Use the law of cosines to find the angle at point x2
    angle = law_of_cosines(a, b, c)
    return angle/2

#Function to obtain phi from ksi
def q_list_phi(impacts):
    res = np.empty(len(impacts),dtype= object)
    for i in range(len(impacts)):
        if i < len(impacts) - 1:
```

```
            phi = math.pi/2 - find_angle(impacts[i-1],impacts[i
                ],impacts[i+1])
            res[i] = phi
        else:
            impacts.append(impacts[0])
            phi = math.pi/2 - find_angle(impacts[i-1],impacts[i
                ],impacts[i+1])
            res[i] = phi
            del impacts[-1]
    return res

### CREATION OF T AND T_TILDE
### CREATION OF T AND T_TILDE


def mu_inverse(theta):
    return 2*C_omega*rho_function_power_1over3(theta)

# Ensure phi is a numpy array in T_matrix function and perform
   element-wise operations
def T_matrix(tronc):
    matrix = np.ones((tronc, tronc))
    matrix[0, :] = mu_inverse(0)
    for i in range(1,tronc):
        theta0 = find_theta0_interpolation(i + 1) # Find theta0
            using Root's method
        #theta0 = find_theta0_newton(i+1,shape) # Find theta0
            using Newton's method
        #theta0 = [(2*math.pi)*(i/q) for i in range(q)] #Uniform
            theta0
        theta_impacts = L_maximizer(theta0)
        ksi = np.array([x_lazutkin(theta) for theta in
            theta_impacts])  # Ensure ksi is a numpy array
        impacts = find_impacts(theta_impacts)
        phi = np.array(q_list_phi(impacts))  # Ensure phi is a
            numpy array
        #trajectories_visualisation(impacts,shape,i+1)
        for j in range(tronc):
            val = 0
            for s in range(i+1):
                val += math.cos(2*math.pi*(j+1)*ksi[s])*math.sin
                    (phi[s])*mu_inverse(theta_impacts[s])
            matrix[i,j] = val*(1/((i+1)*math.sin(math.pi/(i+1)))
                )
    return matrix

# Function to compute tilde(T)
def tilde(T):
    T_D = np.zeros(np.shape(T))
    for i in range(np.shape(T)[1]):
        for j in range(np.shape(T)[1] - i):
```

```
            if j % (i + 1) == 0:
                T_D[i, j + i] = 1/(np.pi)
    inverse_T_D = np.linalg.inv(T_D)
    return np.dot(inverse_T_D, T)

def final_spectre(tronc):
    start_spec = time.time()
    T = T_matrix(tronc)
    T_tilde = tilde(T)
    T_tilde_eigenvalues = np.linalg.eigvals(T_tilde)
    real_parts = np.real(T_tilde_eigenvalues)
    imag_parts = np.imag(T_tilde_eigenvalues)
    plt.figure(figsize=(6, 6))
    plt.scatter(real_parts, imag_parts, color='red', marker='o',
        label='Eigenvalues',s=50,alpha=0.2)
    plt.xlabel('Real Part')
    plt.ylabel('Imaginary Part')
    end_spec = time.time()
    print(f'Running time for plotting spectrum is {end_spec -
       start_spec:.4f} seconds'+' (tronc = '+str(tronc)+')')
    plt.title(f'Eigenvalues in the Complex Plane for '+r'$\tilde
       {T}$'+f' as a {tronc}*{tronc} matrix. Running time = '+f'
       {end_spec - start_spec:.4f} seconds'+f' = {(end_spec -
       start_spec)/60:.4f} minutes')
    plt.axhline(0, color='black', linewidth=1)
    plt.axvline(0, color='black', linewidth=1)
    plt.grid(True)
    plt.legend()
    #plt.savefig(str(omega)+' Eigenvalues_tronc='+str(tronc)+'.
       png',dpi=300, bbox_inches='tight')
    plt.show()


if __name__ == '__main__':
    tronc = 10
    q = 11
    q_max = 100
    plot_rho_defined()
    plot_lazutkin()
    trajectory_visualisation(q)
    final_spectre(tronc)
    animation_trajectory(q_max)
    animation_eigenvalues(tronc)
    end = time.time()
    print('Total running time of the code :'+f'{end - start:.4f}
        seconds = {(end - start)/60:.4f} minutes')
```

A.10 singularities.py

```
import matplotlib.pyplot as plt
```

```
import numpy as np
import math

# Define singularity points
p1 = np.array([-1, 1])
p2 = np.array([1, 1])

# Top arc: circle of radius 1 centered at (0, 1)
center_top = np.array([0, 1])
r_top = 1
theta1_top = np.arctan2(p1[1] - center_top[1], p1[0] -
    center_top[0])
theta2_top = np.arctan2(p2[1] - center_top[1], p2[0] -
    center_top[0])
theta_top = np.linspace(theta1_top, theta2_top, 300)
x_top = r_top * np.cos(theta_top) + center_top[0]
y_top = r_top * np.sin(theta_top) + center_top[1]

# Bottom arc: circle centered at (0, 3)
center_bottom = np.array([0, 3])
r_bottom = np.sqrt((1 - center_bottom[0])**2 + (1 -
    center_bottom[1])**2)
theta1_bottom = np.arctan2(p1[1] - center_bottom[1], p1[0] -
    center_bottom[0])
theta2_bottom = np.arctan2(p2[1] - center_bottom[1], p2[0] -
    center_bottom[0])
theta_bottom = np.linspace(theta1_bottom, theta2_bottom, 300)
x_bottom = r_bottom * np.cos(theta_bottom) + center_bottom[0]
y_bottom = r_bottom * np.sin(theta_bottom) + center_bottom[1]

# Top impacts
q1 = 3
theta_top_impacts = np.linspace(theta1_top, theta2_top, q1+2)
x_top_impacts = r_top * np.cos(theta_top_impacts) + center_top
    [0]
y_top_impacts = r_top * np.sin(theta_top_impacts) + center_top
    [1]
angle_q1 = math.pi/(2*(q1+1))
print('angle q_1 is')
print(angle_q1)
# Bottom impacts
q2 = 3
theta_bottom_impacts = np.linspace(theta1_bottom, theta2_bottom,
     q2+2)
x_bottom_impacts = r_bottom * np.cos(theta_bottom_impacts) +
    center_bottom[0]
y_bottom_impacts = r_bottom * np.sin(theta_bottom_impacts) +
    center_bottom[1]
angle_q2 = abs(theta1_bottom - theta2_bottom)/(2*(q2+1))
print('angle q_2 is')
print(angle_q2)
```

```
# Trajectory
x_trajectory = np.concatenate((x_top_impacts,x_bottom_impacts
    [::-1]))
y_trajectory = np.concatenate((y_top_impacts,y_bottom_impacts
    [::-1]))
trajectory = list(zip(x_trajectory,y_trajectory))
#print(trajectory)


# Calculating the tangent vector of the circle arc at point
    theta
def tangent_vector(theta):
    tangent = np.array([-np.sin(theta), np.cos(theta)])
    return tangent / np.linalg.norm(tangent)

def angle_between(v1, v2):
    v1 = np.array(v1)
    v2 = np.array(v2)
    dot_product = np.dot(v1, v2)
    norm_product = np.linalg.norm(v1) * np.linalg.norm(v2)
    # Clamp value to avoid numerical issues with arccos
    cos_theta = np.clip(dot_product / norm_product, -1.0, 1.0)
    return np.arccos(cos_theta)

#Function to obtain the angles phi
def q_list_phi_right(impacts,theta_top_imp,theta_bottom_imp):
    theta_bottom_imp_inver = theta_bottom_imp[::-1]
    res = []
    for i in range(len(theta_top_imp)-1):
        v = tangent_vector(theta_top_imp[i])
        diff = (impacts[i+1][0]-impacts[i][0],impacts[i+1][1]-
            impacts[i][1])
        res.append(angle_between(-v,diff))
    for i in range(len(theta_bottom_imp_inver)-1):
        v = tangent_vector(theta_bottom_imp_inver[i])
        diff = (impacts[len(theta_top_imp)+i+1][0]-impacts[len(
            theta_top_imp)+i][0],impacts[len(theta_top_imp)+i
            +1][1]-impacts[len(theta_top_imp)+i][1])
        res.append(angle_between(-v,diff))
    return res

def q_list_phi_left(impacts,theta_top_imp,theta_bottom_imp):
    theta_bottom_imp_inver = theta_bottom_imp[::-1]
    res = []
    v = tangent_vector(theta_bottom_imp_inver[-1])
    diff = (impacts[-2][0]-impacts[-1][0],impacts[-2][1]-impacts
        [-1][1])
    res.append(angle_between(v,diff))
    for i in range(1,len(theta_top_imp)):
        v = tangent_vector(theta_top_imp[i])
```

```
        diff = (impacts[i-1][0]-impacts[i][0],impacts[i-1][1]-
            impacts[i][1])
        res.append(angle_between(v,diff))
    for i in range(1,len(theta_bottom_imp)):
        v = tangent_vector(theta_bottom_imp_inver[i])
        diff = (impacts[len(theta_top_imp)+i-1][0]-impacts[len(
            theta_top_imp)+i][0],impacts[len(theta_top_imp)+i
            -1][1]-impacts[len(theta_top_imp)+i][1])
        res.append(angle_between(v,diff))
    return res

def l_q(theta_top_imp,theta_bottom_imp,traject,q_1,q_2):
    theta_bottom_imp_inver = theta_bottom_imp[::-1]
    phi_right = q_list_phi_right(traject,theta_top_imp,
        theta_bottom_imp)
    s = 0
    for i in range(q_1+2):
        s += theta_top_imp[i]*math.sin(phi_right[i])
    for i in range(q_2+2):
        s+= theta_bottom_imp_inver[i]*math.sin(q1 + phi_right[i
            ])
    print('l_q value (test) is')
    return s

def is_periodic(theta_top_imp,theta_bottom_imp,traject):
    phi_right = q_list_phi_right(traject,theta_top_imp,
        theta_bottom_imp)
    phi_left = q_list_phi_left(traject,theta_top_imp,
        theta_bottom_imp)
    phi_couples = [ (phi_left[i],phi_right[i]) for i in range(
        len(phi_right))]
    print(phi_couples)
    i = 0
    while i<len(phi_couples):
        if phi_couples[i][0] != phi_couples[i][1]:
            return False
    i = i + 1
    return True

#print(l_q(theta_top_impacts,theta_bottom_impacts,trajectory,q1,
    q2))
print(is_periodic(theta_top_impacts,theta_bottom_impacts,
    trajectory))
#u = np.array([1,0])
#v = np.array([1,1])
#print('angle between u and v')
#print(angle_between(-u,v))

# Plotting with circle centers shown
plt.figure(figsize=(6, 6))
```

```
plt.plot(x_top, y_top, label="Top Arc (center at (0,1), radius
    =1)")
plt.plot(x_bottom, y_bottom, label="Bottom Arc (radius=√5 ≈
    2.236)", color='orange')
plt.scatter([p1[0], p2[0]], [p1[1], p2[1]], color='red', label="
    Singularities", zorder=5)

# Marking the centers
#plt.scatter(*center_top, color='blue', s=50, zorder=6, label="
    Center (0,1)")
#plt.scatter(*center_bottom, color='green', s=50, zorder=6,
    label="Center (0,3)")

# Annotating the centers
#plt.text(center_top[0] + 0.1, center_top[1], "(0,1)", color='
    blue', fontsize=9, va='center')
#plt.text(center_bottom[0] + 0.1, center_bottom[1], "(0,3)",
    color='green', fontsize=9, va='center')

# Annotating the radius
#midpoint = (p1 + p2) / 2
#plt.text(midpoint[0], midpoint[1] - 1.4, "radius =√5 ≈ 2.236",
     ha='center', fontsize=10, color='orange')

# Plotting trajectories
#plt.plot(x_top_impacts, y_top_impacts, label="Top Trajectories
    ", color='blue',linestyle='dashed')
plt.scatter(x_top_impacts, y_top_impacts, label="Top Impacts",
    color='blue', marker='o', s=50,alpha=0.2)
#plt.plot(x_bottom_impacts, y_bottom_impacts, label="Bottom
    Trajectories", color='orange',linestyle='dashed')
plt.scatter(x_bottom_impacts, y_bottom_impacts, label="Bottom
    Impacts", color='orange', marker='o', s=50,alpha=0.2)
plt.plot(np.concatenate((x_trajectory,[x_trajectory[0]])), np.
    concatenate((y_trajectory,[y_trajectory[0]])), label="
    Trajectories", color='red',linestyle='dashed')

# Plotting the
plt.title("Curve with 2 singularities, q = "+r"$q_1$ + $q_2$ = "
    +str(q1+q2)+", ("+r"$q_1$="+str(q1)+", "+r"$q_2$="+str(q2)+")
    ")
plt.axis('equal')
plt.grid(True)
plt.legend()
plt.show()
```