Approcher la frontière d'une sous-partie de l'espace ainsi que la distance à cette frontière

Wei Wei

Université Blaise Pascal

UFR SCIENCES ET TECHNOLOGIES
DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE
63 177 AUBIERE CEDEX

Année 2008-2009

Master II : SIAD

RAPPORT DE STAGE

Sujet :

Approcher la frontière d'une sous-partie de l'espace ainsi que la distance à cette frontière

Présenté par : Wei WEI
Responsable de stage : Guillaume DEFFUANT
Tuteur de stage : Raoul MEDINA
Jury de stage : Lhouari NOURINE
Lieu de stage : Cemagref (LISC)

Avril 2009 – Septembre 2009
6 mois
Soutenu le 7 Septembre 2009

Remerciements

Durant ces cinq mois de stage, j'ai bénéficié de soutient moral et technique de plusieurs personnes ce qui a rendu mon séjour très agréable.

Je tiens tout d'abord à remercier Monsieur Guillaume DEFFUANT, responsable de stage, de m'avoir guidé et soutenu tout au long de celui-ci. Il s'est montré dès le premier jour accueillant, chaleureux et disponible. J'ai beaucoup apprécié ses qualités humaines tout à fait hors du commun, en outre, des ingrédients nécessaires au bon déroulement du stage.

Un grand merci à Benoît GANDAR, pour les explications théoriques qu'il m'a fournies, l'aide et les conseils concernant les missions évoquées dans ce rapport, qu'il m'a apportés lors des différents suivis.

Je remercie également Monsieur Raoul MEDINA, tuteur de stage, pour ses conseils précieux m'ont permis de surmonter mes difficultés et de progresser dans mes études.

Je remercie aussi Monsieur Lhouari NOURINE, jury de stage, pour l'intérêt qu'il porte à ce stage.

Je n'oublie pas aussi de remercie Maxime LENORMAND, pour m'avoir aidé dans la correction du rapport.

En dernier, je tiens à remercier tous les membres du LISC, pour la bonne humeur, qui a fait que ces cinq mois ont été si agréables à vivre : Clarisse, Sylvie, Thierry, Claire, Thomas, Nicolas, Nabil, Jean Denis, Sônia et Floriana.

Tables des figures

Résumé

Les "Support Vector Machines" (SVMs) sont des outils d'apprentissage statistique qui ont connu un développement considérable au cours de la dernière décennie et qui sont utilisés dans différentes applications du LISC. Cependant, lorsque l'on cherche à approximer précisément la frontière d'une zone de l'espace, les SVMs ne fournissent pas des résultats globaux corrects tout le temps. Nous avons alors besoin de trouver une méthode différente des SVMs pour assurer de bons résultats. Nous supposons que notre espace d'étude se compose de points disposant d'un label soit positif, soit négatif. Le but de ce stage est de construire, à partir de points positifs et négatifs disposés près de la frontière des zones de changement de label, une fonction qui à tout point de l'espace associe la distance de celui-ci à la frontière ainsi que son label. Ce problème est difficile car la véritable frontière est inconnue : il s'agit bien d'un problème d'apprentissage. Nous avons alors développé un algorithme sur la base des réseaux des neurones. Il se résout facilement en théorie, mais en pratique le nombre de points nécessaire est important et n'est pas compatible avec les outils informatiques actuels, notamment quand la dimension du problème augmente. Nous proposons des méthodes pour choisir correctement les paramètres de cet algorithme.

Mots-clés: SVM, apprentissage, frontière, base, reconstruire, distance.

Abstract

Support vector machines (SVMs) are tools of Machine Learning which have grown considerably over the last decade and are used in various applications of LISC. However, when we attempt to approximate precisely the border of a zone of space, they are not able to provide correct results all the time. We need to find a method different from SVMs for insuring good results. We suppose that our space study is composed by points with a label either positive or negative. The target of this field work is to construct, from these points positive and negative arranged near the border zones of label, a function that at all the point of space assorted the distance of this point to the border and its label. This problem is difficult because the border is unknown: it is about a problem of learning. We developped an algorithm based on the neural networks. It is easily solved in theory, but in practice the number of points necessary is important and doesn't compatible with the tools computer exist, especially when the dimension of the problem increases. We propose methods to choose the correct parameters of this algorithm.

Key word: SVM, Machine Learning, border, basis, rebuild, distance.

Table des matières

Introduction

L'essor de l'informatique a toujours apporté aux scientifiques des outils et des méthodes améliorant leurs recherches. La banalisation des stations de travail et l'évolution perpétuelle de leurs capacités sont à l'origine de méthodes très coûteuses en calcul mais donnant des résultats remarquables.

Cependant, malgré ces avancées, les méthodes numériques permettant de calculer des politiques d'actions viables ou résilientes de systèmes dynamiques se heurtent toujours à "la malédiction de la dimensionnalité": elles requièrent des ressources en mémoire et en temps de calcul qui augmentent exponentiellement avec la dimensionnalité du problème, et qui sont souvent hors de portée. Par conséquence, ces méthodes ne sont applicables qu'à des systèmes dynamiques de dimension faible (Inférieur à 3). Pour aider le calcul de politiques d'action sur des écosystèmes ou des systèmes sociaux qui sont généralement des systèmes de dimension supérieure, il est donc indispensable d'améliorer les performances de ces méthodes.

Le LISC, Laboratoire d'Ingénierie pour les Systèmes Complexes dans lequel j'ai réalisé mon stage, est une unité de recherche appartenant au Cemagref, institut public de recherche pour l'ingénierie de l'agriculture et de l'environnement. Il développe le logiciel KAVIAR qui permet d'approcher des noyaux de viabilité de systèmes et de calculer des politiques d'action permettant au système de rester dans son ensemble de contraintes (voir [6]). Il utilise pour cela l'algorithme de classification des SVMs (Séparateurs à Vastes Marges).

L'algorithme des SVMs, permet de définir des surfaces complexes dans des espaces de dimensions importantes, avec des représentations très concises. Cependant celui-ci peut introduire des erreurs loin des zones proches de la frontière de classification. On peut constater l'apparition de "bulles", i.e. de frontières non contraintes par un vecteur support. Le logiciel KAVIAR, basé sur cet algorithme, délivre alors des résultats érodés, ce que peut avoir des conséquences néfastes sur les calculs de politiques d'actions.

Pour résoudre ce problème, nous nous inspirons des travaux de WALDER & al [1]. Ils ont proposé un algorithme qui se base sur la reconstruction d'une surface à partir d'un ensemble de points. Différentes méthodes ont été proposées, nous avons choisissons la méthode suivant : ensemble de Splines avec des largeurs de bandes différentes.

Mon sujet de stage consiste donc à développer cet algorithme d'apprentissage, en m'appuyant des travaux de thèse de Benoît GANDAR. L'objectif du stage est de régler les paramètres de l'algorithme (choix des Splines, i.e. points de bases, et largeurs de bandes associées) pour intégrer nos résultats dans KAVIAR afin d'éviter les erreurs des SVMs. Pour tester et visualiser les résultats, j'ai commencé par utiliser le logiciel SCILAB, puis, ensuite, en fonction des résultats trouvés, j'ai développé une application Java.

Plan du stage

Dans la première partie du stage, j'ai étudié les documents relatifs au problème et implémenté la première version de l'algorithme en langage Scilab pour tester. Nous avons obtenu des résultats intéressants. Mais à cause d'inconvénients propre au logiciel Scilab, je ne pouvais pas lancer le programme avec beaucoup de points, car la mémoire du logiciel était trop petite et le temps de calcul trop important. Nous avons donc transmis les codes en Java et utilisé les structures du logiciel Kaviar qui est développé par LISC. Avec l'aide de Java, nous avons petit à petit amélioré notre algorithme, notamment le choix de ses paramètres.

Chapitre 1

Connaissance établissement

1.1. Le Cemagref

Le Cemagref ou l'institut de recherche finalisée de référence pour la gestion durable des eaux et des territoires (et originellement Centre national du machinisme agricole, du génie rural, des eaux et des forêts) est un organisme de recherche finalisée sur la gestion des eaux et des territoires. Il a le statut d'établissement public à caractère scientifique et technologique. Ses recherches sont orientées vers la production de connaissances nouvelles et d'innovations techniques utilisées par les gestionnaires, les décideurs et les entreprises pour répondre à des questions concrètes de société dans les domaines de la gestion des ressources, de l'aménagement et de l'utilisation de l'espace.

Ses thèmes de recherche sont centrés sur les ressources en eau de surface, les systèmes écologiques aquatiques et terrestres, les espaces à dominante rurale, les technologies pour l'eau, les agro-systèmes et la sûreté des aliments.

Ses recherches contribuent au développement durable des territoires. Elles aident à protéger et gérer les hydro systèmes et les milieux terrestres, à dynamiser les activités qui les valorisent et à prévenir les risques qui leur sont associés. Ses objets d'études sont donc le plus souvent des systèmes complexes, en relation avec des questions de société et sa démarche est presque toujours interdisciplinaire.

1.1.1 Les départements scientifiques

Les orientations scientifiques du Cemagref sont au cœur des enjeux du développement durable. Les recherches sont organisées en 27 thèmes de recherche regroupés en 9 thématiques :

- La gestion de l'eau et des services publics associés.
- Les risques liés à l'eau.
- Les technologies et procédés de l'eau et des déchets.
- La qualité des systèmes écologiques aquatiques.
- Les systèmes écologiques terrestres.
- L'agriculture multifonctionnelle et les nouvelles ruralités.
- Les technologies pour des systèmes agricoles durables.
- Les technologies et procédés physiques pour la sûreté des aliments.
- Les méthodes pour la recherche sur les systèmes environnementaux.

1.1.2 Les unités

Le Cemagref compte actuellement 21 unités de recherche propres, 6 unités mixtes de recherche (UMR) et une équipe de recherche technologique (ERT).Chaque unité de recherche est placée sous la responsabilité fonctionnelle de l'un des chefs de département ou de la direction scientifique (pour les deux unités de recherche méthodologiques), et sous la responsabilité hiérarchique du directeur régional du lieu d'implantation. Les unités sont des lieux de gestion des compétences et des pôles relationnels locaux. Chacune des UR est constituée d'équipes bien identifiées, chaque équipe assurant la mise en œuvre totale ou partielle d'un thème de recherche.

Unités de recherche propres

- Ouvrages hydrauliques et hydrologie **OHAX** - Aix en Provence
- Hydrosystèmes et bioprocédés **HBAN** - Antony
- Hydrologie – hydraulique **HHLY** - Lyon
- Érosion torrentielle, neige et avalanches **ETGR** - Grenoble
- Hydrobiologie **HYAX** - Aix en Provence
- Écosystèmes estuariens et poissons migrateurs amphihalins **EPBX** - Bordeaux
- Réseaux, épuration et qualité des eaux **REBX** - Bordeaux
- Biologie des écosystèmes aquatiques **BELY** - Lyon
- Qualité des eaux et prévention des pollutions **QELY** - Lyon
- Gestion environnementale et traitement biologique des déchets **GERE** - Rennes
- Écosystèmes méditerranéens et risques **EMAX** - Aix en Provence
- Aménités et dynamiques des espaces ruraux **ADBX** - Bordeaux
- Développement des territoires montagnards **DTGR** - Grenoble
- Écosystèmes montagnards **EMGR** - Grenoble
- Écosystèmes forestiers **EFNO** - Nogent
- Agriculture et espace insulaire **AEMA** - Martinique
- Génie des procédés frigorifiques **GPAN** - Antony
- Technologies pour la sécurité et les performances des agroéquipements **TSAN** - Antony
- Technologies et systèmes d'information pour les agrosystèmes **TSCF** - Clermont-Ferrand
- Technologies des équipements agroalimentaires **TERE** - Rennes
- Ingénierie pour les systèmes complexes **LISC** - Clermont-Ferrand

Unité mixte de recherche

- Gestion des services publics **GSP** – Strasbourg (Groupement de Lyon) Gestion de l'eau, acteurs, usages **G-EAU** – Montpellier
- Mutations des activités, des espaces et des formes d'organisation dans les territoires ruraux - Clermont-Ferrand
- Génie industriel alimentaire **GENIAL** - Antony Information et technologie pour les agro-procédés **ITAP** - Montpellier
- Territoires, environnement, télédétection et information spatiale **TETIS** - Montpellier

Equipe de recherche technologique

- Restauration de la continuité écologique des cours d'eau à poissons migrateurs **ERT** – Toulouse

1.1.3 Le Cemagref en région

Tous les unités de recherche se présentés ci-dessus sont répartir sur 9 implantations régionales comme le graphe dessous (Figure 1).

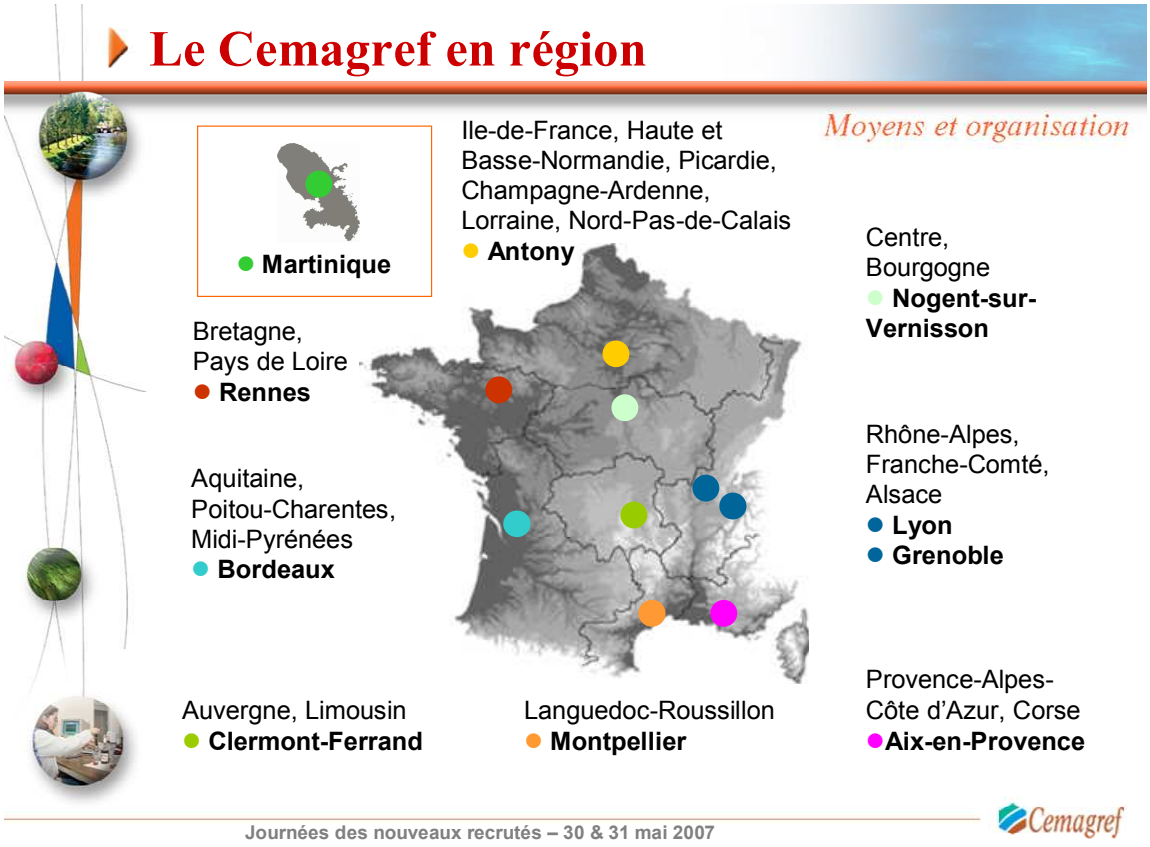


Figure 1 - Le Cemagref en région

1.2. Le LISC

Le LISC (Laboratoire d'Ingénierie pour les Systèmes Complexes) participe à des projets multidisciplinaires, souvent européens, dans lesquels collaborent informaticiens, mathématiciens et spécialistes des écosystèmes ou des sciences sociales. Elle est membre de la Fédération de Recherches CNRS TIMS, Technologies de l'Information, de la Mobilité et de la Sûreté.

Pour comprendre l'évolution des forêts, le comportement de populations d'animaux dans leur milieu, et même les évolutions sociales dans les zones rurales, les chercheurs font appel à des modèles informatiques de plus en plus détaillés. Ils sont ainsi amenés à créer un modèle de chacun des individus d'une population végétale, animale ou humaine, intégré dans un programme informatique qui simule les interactions de chaque individu avec ses paires ou son milieu. Les simulations révèlent des effets collectifs inattendus, sont la théorie mathématique est parfois très difficile à établir. Ce passage de l'individuel au collectif, qui est

au cœur des théories récentes de la complexité, de manifeste dans la plupart des dynamiques environnementales ou sociales.

Les membres du LISC étudient ces comportements globaux, issus d'interactions individuelles et ils les modélisent. Ils développent un outil logiciel pour mieux conduire des expérimentations numériques afin d'observer les comportements globaux lorsque les paramètres varient. Ils construisent ensuite des modèles plus simples et plus facilement utilisables en pratique. Ainsi, ils peuvent calculer sur ces modèles robustes des politiques d'action qui maintiennent la viabilité ou la résilience du système.

Chapitre 2

Les outils utilisés

2.1 Algorithme apprentissage

L'apprentissage artificiel est une discipline scientifique qui recouvre plusieurs domaines d'études mathématiques, statistiques et algorithmiques. La diversité de ces ancrages contribue très certainement au succès de cette récente discipline dont les travaux se caractérisent par : une grande créativité dans la conception de méthodes algorithmiques; la recherche de fondements mathématiques solides pour étayer les méthodes développées; une attention aux mécanismes naturels d'apprentissage et de généralisation; de très nombreux domaines d'applications comme en témoigne l'essor exponentiel de l'apprentissage dans des domaines comme ceux de la fouille de données ou encore de la reconnaissance des formes[2].

Les algorithmes d'apprentissage peuvent se catégoriser selon le type d'apprentissage qu'ils emploient :

- L'apprentissage supervisé : un expert est employé pour étiqueter correctement des exemples. L'apprenant doit alors trouver ou approximer la fonction qui permet d'affecter la bonne étiquette à ces exemples. L'analyse discriminante linéaire ou les SVM sont des exemples typiques.
- L'apprentissage non-supervisé : aucun expert n'est disponible. L'algorithme doit découvrir par lui-même la structure des données. Le clustering et les modèles de mélanges de gaussiennes sont des exemple d'algorithmes d'apprentissage non supervisés.
- L'apprentissage par renforcement : l'algorithme apprend un comportement étant donné une observation. L'action de l'algorithme sur l'environnement produit une valeur de retour qui guide l'algorithme d'apprentissage. L'algorithme de Q-learning est un exemple classique.

Les algorithmes qui l'on rencontre le plus souvent dans ce domaine sont :

- Les machines à vecteur de support (SVM).
- Le boosting.
- Les réseaux de neurones pour un apprentissage supervisé ou non-supervisé.
- La méthode des k plus proche voisins pour un apprentissage supervisé.
- Les arbres de décision.
- Les méthodes statistiques comme par exemples le modèle de mixture gaussienne.
- La régression logistique.
- L'analyse discriminante linéaire.
- La logique floue.
- Les algorithmes génétiques et la programmation génétique.

Ces méthodes sont souvent combinées pour obtenir diverses variantes d'apprentissage. L'utilisation de tel ou tel algorithme dépend fortement de la tâche à résoudre (classification, estimation de valeurs, etc.) L'apprentissage automatique est utilisé dans un spectre très large d'applications : moteur de recherche, aide au diagnostic, bio-informatique, détection de fraudes, analyse des marchés financiers, reconnaissance de la parole, de l'écriture manuscrite, analyse et indexation d'images et de vidéo, robotique…

2.1 Support Vector Machine

2.1.1 Résumé intuitif

Les machines à vecteurs de support ou séparateurs à vaste marge (en anglais Support Vector Machine, SVM) sont un ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de discrimination et de régression.

Les SVMs ont été développés dans les années 1990 à partir des considérations théoriques de *Vladimir Vapnik* sur le développement d'une théorie statistique de l'apprentissage : La *Théorie de Vapnik-Chervonenkis*. Les SVMs ont rapidement étés adoptés pour leur capacité à travailler avec des données de grandes dimensions, le faible nombre d'hyper paramètres, le fait qu'ils soient bien fondés théoriquement, et leurs bons résultats en pratique.

Les séparateurs à vastes marges sont des classifieurs qui reposent sur deux idées clés qui permettent de traiter des problèmes de discrimination non-linéaire, et de reformuler le problème de classement comme un problème d'optimisation quadratique.

La première idée clé est la notion de *marge maximale*. La marge est la distance entre la frontière de séparation et les échantillons les plus proches. Ces derniers sont appelés vecteurs supports. Dans les SVMs, la frontière de séparation est choisie comme celle qui maximise la marge. Ce choix est justifié par la théorie de *Vapnik-Chervonenkis* (ou théorie statistique de l'apprentissage), qui montre que la frontière de séparation de marge maximale possède la plus petite capacité. Le problème est de trouver cette frontière séparatrice optimale, à partir d'un ensemble d'apprentissage. Ceci est fait en formulant le problème comme un problème d'optimisation quadratique, pour lequel il existe des algorithmes connus.

Afin de pouvoir traiter des cas où les données ne sont pas linéairement séparables, la deuxième idée clé des SVMs est de transformer l'espace de représentation des données d'entrées en un espace de plus grande dimension (possiblement de dimension infinie), dans lequel il est probable qu'il existe un séparateur linéaire. Ceci est réalisé grâce à une fonction noyau, qui doit respecter certaines conditions, et qui a l'avantage de ne pas nécessiter la connaissance explicite de la transformation à appliquer pour le changement d'espace. Les fonctions noyau permettent de transformer un produit scalaire dans un espace de grande dimension, ce qui est coûteux, en une simple évaluation ponctuelle d'une fonction. Cette technique est connue sous le nom de *kernel trick*.

2.1.2 Principe général

Les SVMs peuvent être utilisés pour résoudre des problèmes de discrimination, c'est-à-dire décider à quelle classe appartient un échantillon, ou de régression, c'est à dire prédire la valeur numérique d'une variable. La résolution de ces deux problèmes passe par la construction d'une fonction *f* qui à un vecteur d'entrée *x* fait correspondre une sortie *y* : $y = f(x)$

On se limite à un problème de discrimination à deux classes (discrimination binaire), c'est-à-dire $y \in \{-1, 1\}$, le vecteur d'entrée *x* étant dans un espace X muni d'un produit scalaire. On peut prendre par exemple $X = \mathbb{R}^N$.

2.1.3 Discrimination linéaire et hyperplan séparateur

Pour rappel, le cas simple est le cas d'une fonction discriminante linéaire, obtenue par combinaison linéaire du vecteur d'entrée $x = (\mathbf{x}_1, ..., \mathbf{x}_N)^T$:

$$h(x) = w^T x + w_0$$

Il est alors décidé que x est de classe 1 si h(x)≥0 et de classe -1 sinon. C'est un classifieur linéaire.

La frontière de décision h(x) = 0 est un hyperplan, appelé hyperplan séparateur, ou séparatrice. Rappelons que le but d'un algorithme d'apprentissage supervisé est d'apprendre la fonction h(x) par le biais d'un ensemble d'apprentissage :

$$\{(x_1, l_1), (x_2, l_2), \ldots, (x_p, l_p)\} \in \mathbb{R}^N \times \{-1, 1\}$$

où les $\mathbf{L_k}$ sont les labels, p est la taille de l'ensemble d'apprentissage, N la dimension des vecteurs d'entrée. Si le problème est linéairement séparable, on doit alors avoir :

$$l_k h(x_k) \geq 0 \quad 1 \leq k \leq p, \qquad \text{autrement dit} \qquad l_k(w^T x_k + w_0) \geq 0 \quad 1 \leq k \leq p.$$

2.1.4 Marge maximale

On se place désormais dans le cas où le problème est linéairement séparable. Même dans ce cas simple, le choix de l'hyperplan séparateur n'est pas évident. Il existe en effet une infinité d'hyperplans séparateurs, dont les performances en apprentissage sont identiques (le risque empirique est le même), mais dont les performances en généralisation peuvent être très différentes. Pour résoudre ce problème, il a été montré qu'il existe un unique hyperplan optimal, défini comme l'hyperplan qui maximise la marge entre les échantillons et l'hyperplan séparateur.

Il existe des raisons théoriques à ce choix. *Vapnik* a montré que la capacité des classes d'hyperplans séparateurs diminue lorsque leur marge augmente.

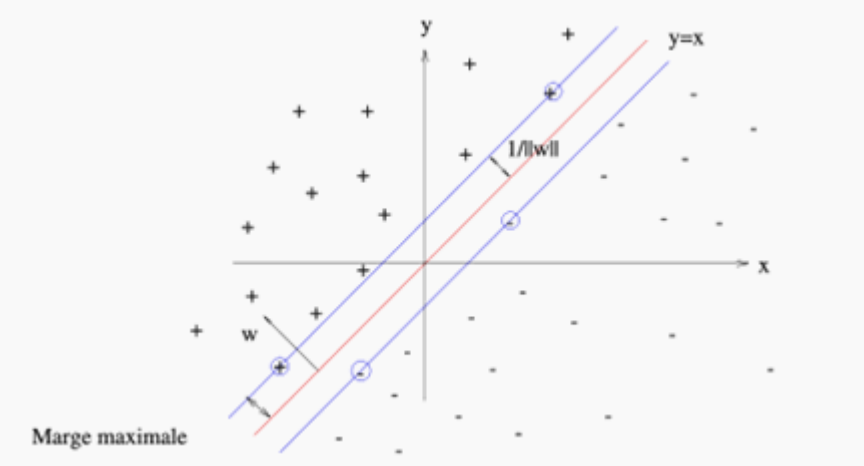


Figure 2 - L'hyperplan optimal (en rouge) avec la marge maximale. Les échantillons entourés sont des *vecteurs supports*.

La marge est la distance entre l'hyperplan et les échantillons le plus proches. Ces derniers sont appelés vecteurs supports. L'hyperplan qui maximise la marge est donné par :

$$\arg\max_{w,w_0} \min_k \{||x - x_k|| : x \in \mathbb{R}^N, w^T x + w_0 = 0\}$$

Il s'agit donc de trouver w et w_0 remplissant ces conditions, afin de déterminer l'équation de l'hyperplan séparateur :

$$h(x) = w^T x + w_0 = 0$$

2.1.5 Choix de la fonction noyau

Plus formellement, on applique aux vecteurs d'entrée x une transformation non-linéaire φ. L'espace d'arrivée φ(*X*) est appelé *espace de redescription*. Dans cet espace, on cherche alors l'hyperplan

$$h(x) = w^T \varphi(x) + w_0$$

qui vérifie $l_k h(x_k) > 0$, pour tous les points x_k de l'ensemble d'apprentissage, c'est-à-dire l'hyperplan séparateur dans l'espace de redescription.

En pratique, on ne connaît pas la transformation φ, on construit plutôt directement une fonction noyau. Celle ci doit respecter certaines conditions, elle doit correspondre à un produit scalaire dans un espace de grande dimension. Le théorème de *Mercer* explicite les conditions que K doit satisfaire pour être une fonction noyau : elle doit être symétrique, semi-définie positive.

L'exemple le plus simple de fonction noyau est le noyau linéaire :

$$K(x_i, x_j) = x_i^T \cdot x_j$$

On se ramène donc au cas d'un classifieur linéaire, sans changement d'espace. L'approche par *Kernel trick* généralise donc l'approche linéaire en en faisant un cas particulier. Le noyau linéaire est parfois employé pour évaluer la difficulté du problème.

Les noyaux usuels employés pour les SVM sont:

- le noyau polynomial

$$K(x_i, x_j) = (x_i^T \cdot x_j)^d$$

- le noyau gaussien

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right)$$

2.3 Scilab

Scilab est un logiciel libre de calcul numérique fournissant un environnement de calcul pour des applications scientifiques. Développé depuis 1990 par des chercheurs de l'INRIA (Institut National de Recherche en Informatique et Automatique), nous pouvons dire que Scilab est un pseudo-clone libre de Matlab.

Scilab est un puissant environnement de programmation interactif pour le calcul numérique. C'est un gros logiciel : environ 13 000 fichiers, plus de 400 000 lignes de code source (en C et Fortran), 70 000 lignes de code Scilab (librairies spécialisées), 80 000 lignes de "man" (help en ligne et doc), le tout géré sous Linux par 18 000 lignes de fichiers de configuration (scripts et Makefiles). Scilab comporte un langage de programmation et un interpréteur qui travaille sur des objets de type numériques mais pouvant être structurés selon les besoins de l'utilisateur. C'est un système complètement ouvert, puisque c'est l'utilisateur qui définit ses propres fonctions à partir des puissantes primitives de calcul qui sont fournies. Scilab est distribué sous forme binaire pour les PC-Linux, Windows et les stations Unix "classiques" (Sun HP, DEC, SGI, ...), mais construire Scilab sous Linux à partir de la version source ne pose aucun problème car les compilateurs *gcc* et g77 sont dans les distributions standards. Un "configure" suivi d'un "make all" et c'est parti pour la compilation : 10 à 15 minutes plus tard (avec un pentium assez récent !) Scilab est prêt à démarrer. Le code exécutable (scilex) se trouve dans le sous répertoire *bin* et fait à peu près 5 Mégas. Evidemment, les sources et les Makefiles étant disponibles, l'utilisateur plus courageux peut même se recompiler un Scilab personnalisé à son goût. L'interface du Scilab voit annexe 1.

Scilab est un environnement agréable pour faire du calcul numérique car on dispose sous la main des méthodes usuelles de cette discipline (c'est aussi la raison pourquoi nous choisissons le Scilab pour tester), par exemple :

- Résolution de systèmes linéaires.
- Calcul de valeurs propres, vecteurs propres.
- Décomposition en valeurs singulières, pseudo-inverse.
- Transformée de Fourier rapide.
- Plusieurs méthodes de résolution d'équations différentielles.
- Plusieurs algorithmes d'optimisation.

- Résolution d'équations non-linéaire.
- Génération de nombres aléatoires.
- De nombreuses primitives d'algèbre linéaire utiles pour l'automatique.

D'autre part, Scilab dispose aussi de toute une batterie d'instructions graphiques, de bas-niveau (comme tracer un polygone, récupérer les coordonnées du pointeur de la souris, etc…) et de plus haut niveau (pour visualiser des courbes, des surfaces) ainsi que d'un langage de programmation assez simple mais puissant et agréable car il intègre les notations matricielles. Quand vous testez un de vos programmes écrit en langage Scilab, la phase de mise au point est généralement assez rapide car vous pouvez examiner facilement vos variables : c'est comme si l'on avait un débogueur. Enfin, si les calculs sont trop longs (le langage est interprété…) vous pouvez écrire les passages fatidiques comme des sous-programmes C ou fortran (77) et les lier à Scilab assez facilement.

Au tout début Scilab peut s'utiliser simplement comme une calculette capable d'effectuer des opérations sur des vecteurs et matrices de réels et/ou complexes (mais aussi sur de simples scalaires) et de visualiser graphiquement des courbes et surfaces. Dans ce cas basique d'utilisation, vous avez uniquement besoin du logiciel Scilab. Cependant, assez rapidement, on est amené à écrire des scripts (suite d'instructions Scilab), puis des fonctions et il est nécessaire de travailler de pair avec un éditeur de texte comme par exemple, emacs (sous Unix et Windows), wordpad (sous Windows), ou encore nedit, vi (sous Unix)…

Pour tout renseignement, consulter la "Scilab home page" : *http ://scilabsoft.inria.fr*

2.4 Java

2.4.1 Java langage

Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton employés de Sun Microsystems avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld.

Le langage Java a la particularité principale que les logiciels écrits avec ce dernier sont très facilement portables sur plusieurs systèmes d'exploitation tels que UNIX, Microsoft Windows, Mac OS ou Linux avec peu ou pas de modifications… C'est la plate-forme qui garantit la portabilité des applications développées en Java.

Le langage reprend en grande partie la syntaxe du langage C++, très utilisé par les informaticiens. Néanmoins, Java a été épurée des concepts les plus subtils du C++ et à la fois les plus déroutants, tels que l'héritage multiple remplacé par l'implémentation des interfaces. Les concepteurs ont privilégié l'approche orientée objet de sorte qu'en Java, tout est objet à l'exception des types primitifs (nombres entiers, nombres à virgule flottante, etc.)

Java permet de développer des applications client-serveur. Côté client, les applets sont à l'origine de la notoriété du langage. C'est surtout côté serveur que Java s'est imposé dans le milieu de l'entreprise grâce aux servlets, le pendant serveur des applets, et plus récemment les JSP (JavaServer Pages) qui peuvent se substituer à PHP, ASP et ASP.NET.

Java a donné naissance à un système d'exploitation (JavaOS), à un environnement de développement (eclipse/JDK), des machines virtuelles (MSJVM, JRE) applicatives multi-plates-formes (JVM), une bibliothèque Java (J2ME) avec interface graphique (AWT/Swing), des applications Java (logiciels, servlet, applet). La portabilité du code Java est assurée par la machine virtuelle. JRE – la machine virtuelle qui effectue la traduction et l'exécution du bytecode en code natif – supporte plusieurs processus de compilation (à la volée/bytecode, natif). La portabilité est dépendante de la qualité de portage des JVM sur chaque OS.

L'utilisation native du langage Java pour des applications sur un poste de travail restait jusqu'à présent relativement rare à cause de leur manque de rapidité. Cependant, avec l'accroissement rapide de la puissance des ordinateurs, les améliorations au cours de la dernière décennie de la machine virtuelle Java et de la qualité des compilateurs, plusieurs technologies ont gagné du terrain comme par exemple *Netbeans* et l'environnement Eclipse. Java est aussi utilisé dans le programme de mathématique *Matlab* au niveau de l'interface homme machine et pour le calcul formel. Les applications Swing apparaissent également comme une alternative à la technologie .NET.

Le langage Java a connu plusieurs évolutions depuis le JDK 1.0 (*Java Development Kit*) avec l'ajout de nombreuses classes et packages à la bibliothèque standard. Depuis le J2SE1.4, l'évolution de Java est dirigée par le JCP (*Java Community Process*) qui utilise les JSR (*Java Specifications Requests*) pour proposer des ajouts et des changements sur la plate-forme Java. Le langage est spécifié par le JLS (*Java Language Specification*). Les modifications du JLS sont gérées sous le code JSR 901. Nous utilisons la librairie *JAMA* dans l'implémentation du code.

2.4.2 JAMA

JAMA est un package de base d'algèbre linéaire pour java. Il est fourni à l'utilisateur pour la construction de la manipulation le différent type des matrices. Il est destiné à fournir suffisamment de fonctionnalités pour des problèmes courants. Il est adopté une manière qui est naturel et compréhensible pour les non-experts. Il est composé six classes java :

Matrix.
CholeskyDecomposition.
LUDecomposition.
QRDecomposition.
SingularValueDecomposition.
EigenvalueDecomposition.

En cours de programmation, il s'agit des calculs de matrices et leurs caractéristiques. Cette librairie *JAMA* est donc une bibliothèque très intéressant pour nous.

Chapitre 3

La problématique

Les machines à vecteurs de support ou séparateurs à vaste marge (en anglais Support Vector Machine, SVM) sont un ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de discrimination et de régression. Une application de SVM est de calculer la fonction de noyau de viabilité d'un système dynamique.

3.1 Le noyau de viabilité

Le problème principal auquel nous nous intéressons dans ce stage est celui de la viabilité d'un système dynamique dans un ensemble de contraintes *K*. Le problème est de trouver une politique d'action sur le système, qui le maintienne indéfiniment à l'intérieur de cet ensemble de contraintes. Ce problème est fréquent en écologie ou robotique, lorsque le système meurt ou se détériore lorsqu'il quitte une certaine région de l'espace. Le but n'est pas alors pas de choisir une solution 'optimale' en fonction d'un certain critère, mais de sélectionner des actions 'viables', dans le sens où elles permettent au système de se maintenir dans un ensemble de contraintes. La figure 3 représente un exemple d'ensemble de contraintes K d'un système dynamique (de forme ovale). Les points de couleur blanche sont les points pour lesquels il n'existe pas de contrôle viables (courbes rouges), et n'appartiennent donc pas au noyau de viabilité du système. Le domaine bleu est le domaine pour lesquels il existe au moins une politique d'action viable (courbe verte), et appartiennent donc au noyau de viabilité.

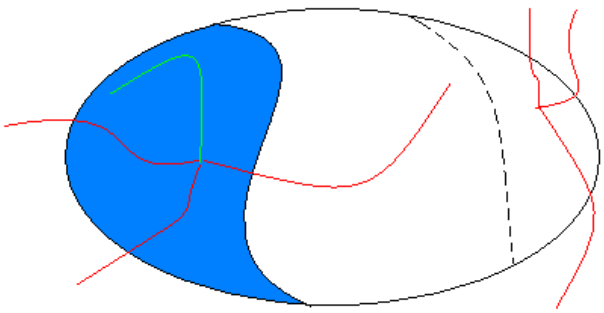


Figure 3 - L'ensemble de contrainte et le noyau de viabilité en bleu.

Un algorithme d'apprentissage visant à approcher le noyau de viabilité d'un système dynamique utilisant des SVMs ont été conçus par Guillaume DEFFUANT & al [3]. Mais malheureusement, selon les paramètres de SVMs et les modèles dynamiques, les résultats obtenus par cet algorithme ne sont pas forcement fiables. Nous rencontrons des bulles dans le domaine de noyau, i.e. des frontières non contraintes par un vecteur support. Ce bug nous influence d'obtenir un noyau correcte. La figure 4 représente un exemple qui peut arriver, à gauche des bulles, le point originairement est viable, mais l'algorithme de classifie comme non viable.

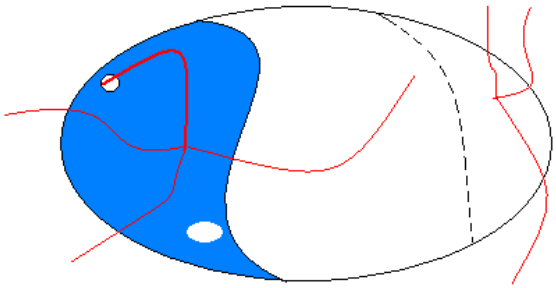
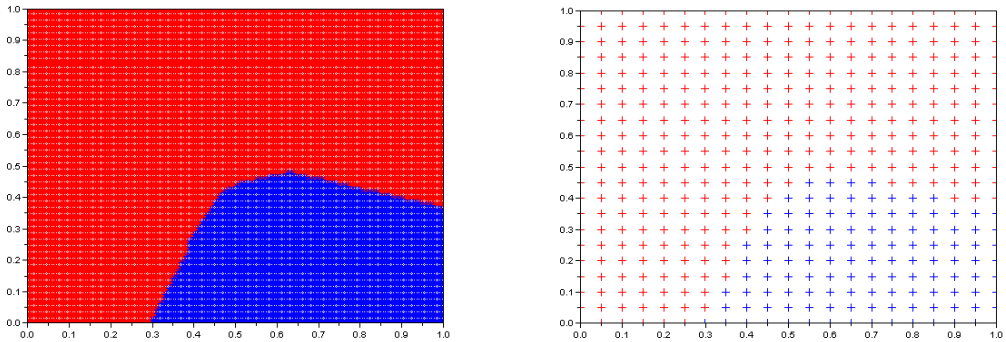


Figure 4 - Un exemple de bulle.

L'algorithme des SVMs ne permet pas de calculer exactement la distance d'un point à la frontière de classification: il permet uniquement de séparer les points en un certain nombre de classes. Afin de résoudre le problème de "bulles", et d'estimer correctement la distance d'un point à la frontière de classification dans les algorithmes d'estimation de noyau de viabilité, nous avons besoin de trouver un algorithme qui soit très différent des SVMs. Le papier de WALDER & al [1] nous propose une méthode pour reconstruire une telle fonction d'approximation de distances et de classification.

3.2 Reconstruction de la fonction

Nous considérons le problème à partir de points positifs ou négatifs disposés près de la frontière pour construire la fonction qui à un point associe la distance à la frontière et le label de celui-ci. Comme la figure 5, nous voulons reconstruire la fonction *(a)* par les points de la grille *(b),* les points rouges sont les points positifs, et les points bleus sont les points négatifs.



(a) (b)

Figure 5 – Reconstruire la fonction à partir de points positif ou négatifs.

3.2.1 La structure des données

Nous supposons avoir un ensemble de points E = $\{(x_i, y_i)\}_{i=1...n}$ correspondant aux points initiaux disposés selon une grille de taille totale n. Dans cet exemple, nous définissons 21 points par dimension. Pour facilité les calculs, nous supposons que toutes les coordonnés des points sont entre 0 et 1 par défaut. Chaque point de E possède une valeur d'étiquette 1 ou -1. Nous pouvons dire que tous les points avec l'étiqueteur 1 sont les points de noyau, -1 sinon. Les x_i sont des valeurs vectorielles, car nous allons étudier le cas en plusieurs dimensions. Les coefficients de x_i sont les coordonnées des points par dimension. Les y_i sont les distances signées des points x_i à la frontière de classification, c'est-à-dire si l'étiquette de points est positive, la distance est aussi positive, sinon la distance est négative. Nous définissons aussi une valeur de pas égale à la distance entre deux points proches (pas de la grille).

L'exemple de la figure 5 est défini en dimension deux, les points bleus correspondent à l'étiquette -1, les rouges correspondent à étiquette 1. Nous supposons avoir une base d'apprentissage : M = $\{(x_1, y_1), ..., (x_m,y_m)\}$, où M sont les points proches de la frontière, m est la taille de M. La fonction de l'estimation de la distance est notée *f*. Nous cherchons la fonction d'estimation *f* dans un espace de Hilbert à noyau reproduisant *H* muni du noyau k(.,.). C'est-à-dire y_i est la valeur initiale, $f(x_i)$ est la valeur approximée que nous voulons chercher. Dans notre étude, nous pouvons calculer la valeur réelle de y_i pour tout point afin d'évaluer la qualité de nos résultats. Mais dans la vraie application, la valeur de yi est souvent inconnue. Nous présenterons dans la partie suivante la construction de la fonction d'approximation *f*. celle-ci se caractérise par des points de base b_k et des largeurs de bande associées S_k: ce sont ces paramètres et leur nombre que je vais chercher à optimiser.

Sur cette même figure 6, nous voyons un ensemble de points de base (les points verts), la difficulté du problème est trouver les bonnes bases pour construire la fonction d'approximation *f*. Nous supposons l'ensemble de points de base B = $\{(b_k,S_k)\}$, où b_k est une valeur vectorielle définie comme x_i, elle correspond à la coordonnée des points de base. S_k est une valeur de distance, elle correspond au rayon du domaine couvert par une base de paramètre b_k.

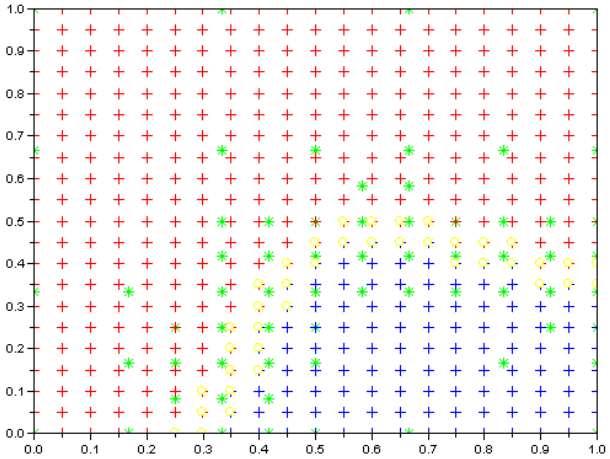


Figure 6 - Structure des points, le graphe obtenu avec Scilab.

3.2.2 Construction des fonctions de base

Nous cherchons ici que la fonction d'approximation de *f* de la fonction sous la forme.

$$f(x) = \sum_{k=1}^{p} \pi_k f_k(x). \qquad \textbf{(1)}$$

Où *p* est le nombre de fonctions de base (donc le nombre de points de base), π_k sont des valeurs nous devons également estimer.

Les fonctions de base de la forme : $f_k(x) = \exp(-\frac{\|x - v_k\|^2}{s_k})$

La fonction de régularisation dans l'équation s'exprime alors : $\|f\|_H^2 = \sum_{k=1}^{p} \frac{\pi_k^2}{s_k}$.

Le problème d'optimisation écrit dans le papier [1] correspondant est alors :

$$\min_{f \in H} \|f\|_H^2 + C \sum_{i=1}^{m} \left(\frac{f(x_i) - y_i}{y_i}\right)^2 \qquad \textbf{(2)}$$

Où C est un terme de pénalisation de la complexité de la fonction. Si C est petit, on tolère les erreurs (sur la base d'apprentissage) mais la fonction sera mathématiquement simple. A l'inverse, si C est grand, nous pénalisons les erreurs d'apprentissage, et la fonction obtenue sera plus complexe.

Donc la formule (2) devient :

$$(P) = \min_{\pi_k} \sum_{k=1}^{p} \frac{\pi_k^2}{s_k} + C \sum_{i=1}^{m} \left(\frac{f(x_i) - y_i}{y_i}\right)^2$$

$$= \min_{\pi_k} \sum_{k=1}^{p} \frac{\pi_k^2}{s_k} + C \sum_{i=1}^{m} \frac{1}{y_i^2} \left(\sum_{k=1}^{p} (\pi_k f_k(x_i) - y_i)\right)^2 = \Delta$$

Ensuite nous faisons la dérivation pour tout j = 1,…, *p* sur formule Δ.

$$\frac{\partial \Delta}{\partial \pi_j} = 2\frac{\pi_j}{s_j} + C \sum_{i=1}^{m} \frac{2}{y_i^2} \left(\sum_{k=1}^{p} (\pi_k f_k(x_i) - y_i)\right) f_j(x_i)$$

Or : $\frac{\partial \Delta}{\partial \pi_j} = 0$ d'où pour tout j = 1,…, p.

$$\frac{\pi_j}{s_j} + C \sum_{i=1}^{m} \frac{1}{y_i^2} \left(\sum_{k=1}^{p} (\pi_k f_k(x_i) - y_i)\right) f_j(x_i) = 0$$

$$\frac{\pi_j}{s_j} + C \sum_{k=1}^{p} \sum_{i=1}^{m} \pi_k \frac{f_k(x_i)}{y_i} \frac{f_j(x_i)}{y_i} - C \sum_{i=1}^{m} \frac{f_j(x_i)}{y_i} = 0$$

$$\frac{\pi_j}{s_j} + C\sum_{k=1}^{p}\sum_{i=1}^{m}\pi_k \frac{f_k(x_i)}{y_i}\frac{f_j(x_i)}{y_i} = C\sum_{i=1}^{m}\frac{f_j(x_i)}{y_i} \qquad \textbf{(3)}$$

Soit nous définissons : $\Psi_j = \left\{\frac{f_j(x_1)}{y_i}, \frac{f_j(x_2)}{y_2}, ..., \frac{f_j(x_p)}{y_p}\right\}$

La formule (3) devient :

$$\frac{\pi_j}{s_j} + C\sum_{k=1}^{p}\pi_k \Psi_k.\Psi_j = C.\Psi_j$$

Soit Π le vecteur défini par $\Pi = [\pi_1, \ldots, \pi_p]$, et pour j = 1,…, p : $f_j(X) = [f_j(x_1),\ldots, f_j(x_m)]$,

ones(1) est une valeur vectorielle possède que le 1.

Nous définissons les matrices M par : $\begin{cases} M_{kj} = C\Psi_k\Psi_j sik \neq j \\ M_{kk} = C\Psi_k^2 + \frac{1}{s_k} sik = j \end{cases}$ et $Z = C.ones(1).\Psi_j$

La formule précédente s'écrit donc matriciellement M Π = Z.

La valeur de matrice M et Z nous pouvons les trouver par calcule car nous avons déjà connu la fonction de base, d'ici nous définissons les fonction f_k sont des fonctions exponentielles de la distance, i.e. $f_k(x) = \exp(-\frac{\|x - v_k\|^2}{s_k})$, donc nous pouvons facilement de trouver la valeur $\Pi = M^{-1}Z$.

Nous faisons une conclusion pour tous les paramètres dans notre structure :

notation	Définition
x_i	Coordonnée des points de la grille initiale.
y_i	Distance de ce point à la frontière.
$f(x_i)$	Valeur d'approximation de la distance signée du point x_i à la frontière.
$f_k(x_i)$	$K^{ième}$ fonctions de base estimée au point x_i.
b_k	Paramètre de la $K^{ième}$ fonction de base: coordonnée du point associé.
S_k	Paramètre de la $K^{ième}$ fonction de base: domaine de couvert de celle-ci.
pas	Distance entre deux points proches.

3.2.3 Choix des points de base

Selon les calculs de la partie précédente, nous avons obtenu la matrice M qui est une matrice carrée de dimension *p*, et la matrice Z est un vecteur de dimension *p*. où *p* est le nombre de points de base. Tous les points de base se caractérisent par deux paramètres

important : b_k et S_k (3.2.1), b_k correspond ses coordonnées de la nouvelle grille de base, S_k correspond à la distance entre les deux points proches.

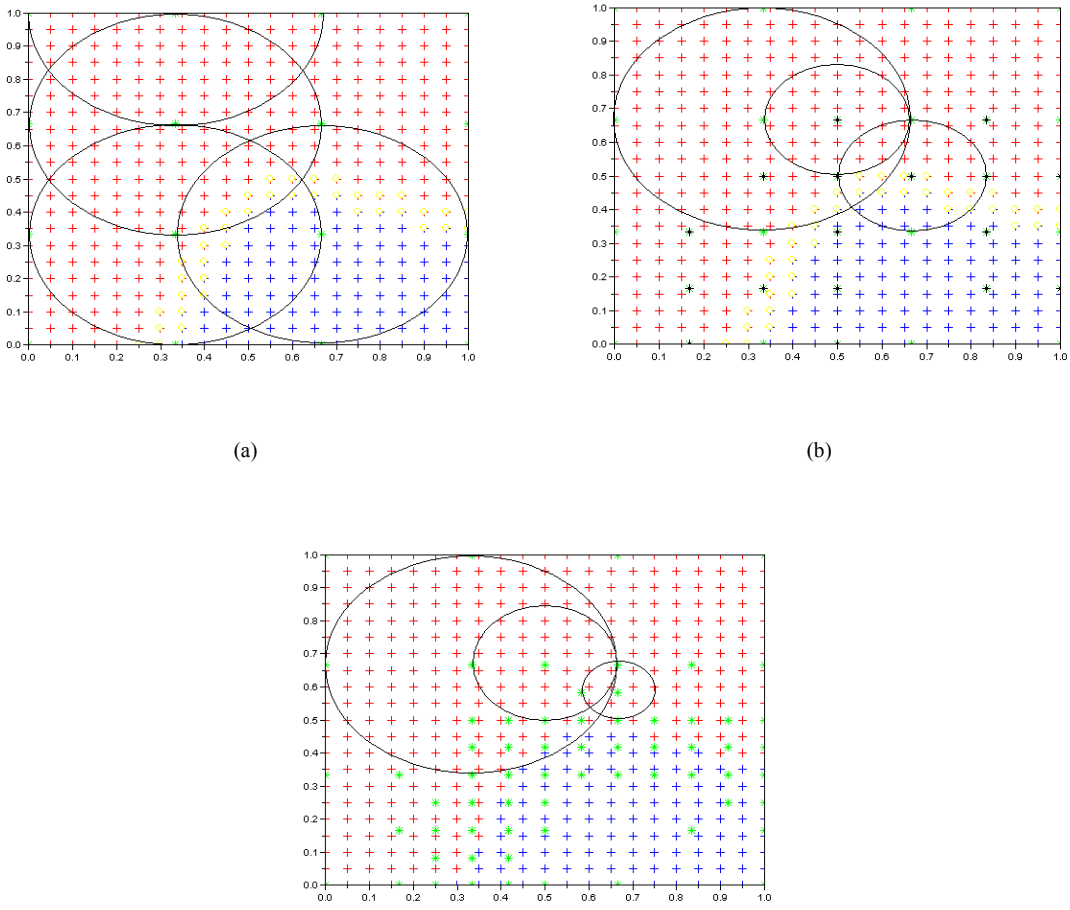
Pour commencer de tester notre méthode, nous proposons une première version de l'algorithme pour choisir les points de base. Le principe de l'algorithme est : à partir de la grille donnée, nous reconstruisons une grille selon les paramètres. Dans cet algorithme, nous avons 3 paramètres :

- Nombre de points par dimension pour la première grille (*nb_points*).
- Nombre d'itération pour chercher les points (*It*).
- Nombre de facteur à recouper les grilles (*facteur*).

Tous les points dans de la nouvelle grille possèdent des caractéristiques b_k et S_k, nous enregistrons tous les points de la première grille. La deuxième itération, nous recoupons l'intervalle de deux points proches de la première grille en *facteur*, nous obtenons la deuxième grille avec le nombre de points *nb_points*facteur-1* par dimension. Pour tous les points de la nouvelle grille, nous calculons la distance avec les points de frontière (la distance plus courte). Nous enregistrons tous les points dont la distance est inférieure à S_k. Nous faisons la même opération pour les itérations suivantes, les listes des points enregistrés sont les points de base. L'algorithme complet est suivant :

```
Algorithme : Choisi_base
Données : M // liste des points de frontière.
          Dimension : // nombre de dimension.
          // une fonction pour générer une grille avec des paramètres donné.
          Generateur_grille(nb_points, dimension)
Constante : Nb_points // nombre de points par dimension par défaut.
            It // nombre de itération.
            Facteur // nombre de facteur.
Variables : bk, // valeur vectorielle.
            Sk. // Réel.
            Valeur. // nombre de points par dimension en courant.
Résultat : B // liste de base.
Début
// construire la première grille de base
        Valeur = nb_points;
        Grille = Generateur_grille(nb_points, Dimension);
        Mise à jour les valeurs de bk et Sk;
        B ← tous les points de Grille;
        Pour i = 2 à It faire
                Valeur = Valeur*Facteur-1;
                Grille =  Generateur_grille(Valeur, Dimension);
                Mise à jour les valeurs de bk et Sk;
                Pour tous les points Pk ∈ Grille faire
                        If (distance(bk, M)<Sk)
                                B ← Pk;
                        Fin if
                Fin pour
                i = i+1;
        Fin pour
Fin
```

L'exemple de la figure 7, le graphe *(a)* montre la première grille de base. Le graphe *(b)* montre la deuxième itération de la grille construire, les points noirs sont les points de base enregistrés de cette itération. Le graphe *(c)* est la solution finale obtenue. Selon les graphes, nous avons peu des points de base dans le domaine loin de frontière et plus de points de base proche de frontière. Les cercles correspondent aux domaines d'influence des points de base, les rayons des cercles correspondent la valeur du S_k.



(a) (b)

(c)

Figure 7 - Choisir les points de base. Les légendes sont la même que la figure 4, les paramètres de l'algorithme est: nb_points égale à 4, It égale à 3, facteur égale à 2.

Chapitre 4

Développement

4.1 Première version (en Scilab)

Scilab est un logiciel pour effectuer des calculs numériques et dispose d'une portabilité totale sous les différents systèmes d'exploitation. Il a été conçu pour faciliter le traitement des matrices, et présente une librairie graphique élaborée et très complète. La programmation est aussi significativement plus rapide que les langages orientés (i.e. C/C++, Java) pour le calcul et pour l'affichage. Tous ces avantages nous permettent de faire des premiers tests rapidement. En effet, nous pouvons programmer et visualiser rapidement et nous n'avons pas difficulté pour traiter des matrices dans les applications.

4.1.1 Programmation

Pour les applications théoriques du chapitre 3, nous avons créé les fonctions suivantes:

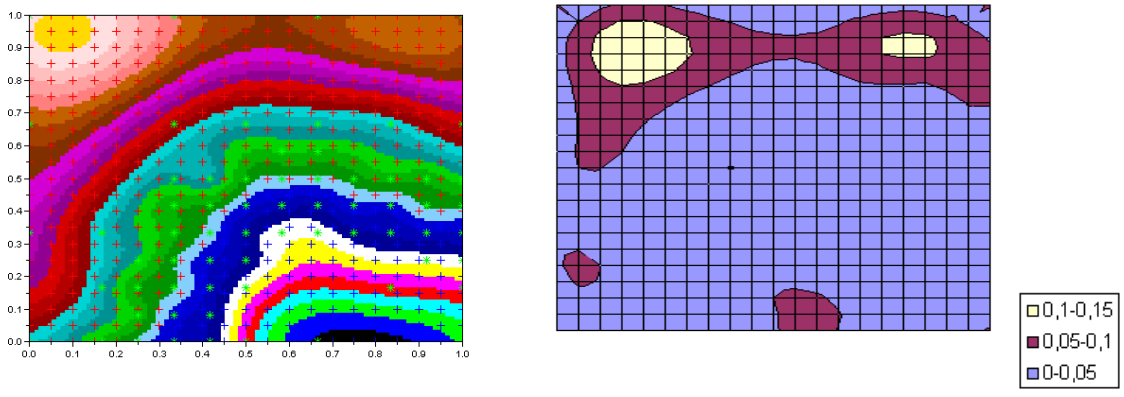
- *Générateur_grille(dimension, nb_points):* pour générer une grille en donnant le nombre de dimension et le nombre de points par dimension. Il retourne une liste des points.
- *Etiqueteur(list_point, k, point_pos, point_neg) :* la fonction va marquer tous les points dans la grille avec une étiqueteur 1 ou -1, elle utilise la méthode de k-plus proche voisin. Elle retourne une liste d'étiquettes. *point_pas* et *point_neg* sont les listes de points pour la méthode k-plus proche voisin.
- *Point_proche(list_points, etiqueteur)* : la fonction cherche tous les points proches de la frontière. On va parcourir tous les points de la grille, et vérifier tous les voisins, nous gardons les points s'ils ont un voisin d'étiquette différente. Elle retourne une liste de points de frontière.
- *Liste_b(list_points, nb_point, it, facteur) :* la fonction implémente l'algorithme *choisi_base (3.2.3)*. Elle retourne une liste de points de base. Nous gardons les valeurs de b_k et S_k.
- *Calcul_distance(list_point, etiqueteur) :* la fonction calcule la distance minimale des points à la frontière. Cette distance est la distance de ce point au point plus proche de la frontière, si ce point frontière est positif, nous faisons plus *pas*/2, si ce point frontière est négatif, nous faisons moins *pas*/2. La distance de points de la frontière égale à *pas*/2.
- *Ajouter_b():* Ajouter liste de base dans la liste d'apprentissage.
- *Choix_f():* choisir la fonction de la base de forme.

- *Matrice_Z() :* la fonction construit la matrice Z (voir paragraphe 3.2.2).
- *Matrice_M()* : la fonction construit la matrice M (voir paragraphe 3.2.2).
- *Approxi_f()* : calculer les valeurs d'approximation des points de la grille. Dans cette fonction nous calculons la matrice pi.
- *Affiche2D* : visualiser les résultats en 2-dimension.
- *Affiche3D* : visualiser les résultats en 3-dimension.
- *Main()*: fonction principale.

La fonction principale est représentée dans l'annexe 2. En utilisant les fonctions dessous, nous avons réalisés une série de testes. Nous avons observé des *erreurs importantes* sur les domaines loin de la frontière. Les grandes erreurs c'est-à-dire les résultats de distance que nous avons obtenus par approximation sont très différents des distances réelles, et nous avons encore des bulles dans les domaines loin de frontière. Pour résoudre ce problème, nous ajouterons aussi des points de base dans la liste d'apprentissage. C'est-à-dire nous remplaçons M par M+B, car les points de base sont couverts tous les points de la grille.

4.1.2 Observation

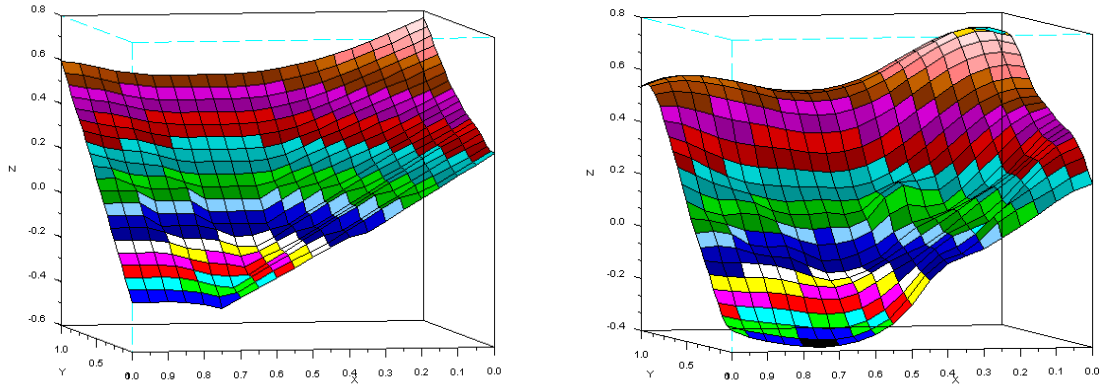
Nous avons obtenu les résultats suivants sur le même exemple que celui de la figure 8, le graphe *(a)* représente un résultat avec les lignes de niveau de la fonction distance approximée, le graphe *(b)* représente le niveau d'erreur de l'approximation. Nous trouvons que la méthode apprend bien les domaines proches de frontière, mais pas les endroits loin de celle-ci. Mais nous ne rencontrerons plus les bulles dans les résultats.



(a) (b)

Figure 8 – (a): Lignes de niveau de la fonction obtenue. (b) Ligne de niveau de la proportion d'erreur sur la distance à la frontière.

Nous affichons aussi un résultat en dimensions trois pour comparer les solutions comme la figure 9. Le graphe (a) est la fonction de distance que nous voulons apprendre, le graphe (b) est la fonction d'approximation trouvée. Nous pouvons faire les mêmes remarques que pour la figure 8.



(a) (b)

Figure 9 – (a): fonction à apprendre. (b): approximation obtenue par la méthode.

La première série de tests, se compose de dix fonctions à apprendre et nous comparons les résultats. Les données de la figure 10 sont les résultats obtenus, toutes les fonctions sont approximées sur une grille de dimension deux avec 31 points par dimension. La première colonne correspond au nombre de points proches de la frontière. La deuxième colonne correspond aux nombres de bases trouvées par l'algorithme *choisi_base (3.2.3)*, les paramètres de l'algorithme sont : *nb_points* égale à 4, *It* égale à 3, *facteur* égale à 2. La troisième colonne correspond aux erreurs des points proches des frontières (résultats en pourcentage). La quatrième colonne correspond aux pourcentages des erreurs sur tous les points de la grille. La cinquième colonne donne le temps de calcul.

Fonction	Points frontières	Points bases	Erreurs points frontière (%)	Erreurs points (%)	Temps de calcul (s)
1	7	19	0,27	0,09	1
2	42	39	0,27	0,13	11
3	42	41	0,18	0,14	18
4	42	43	0,2	0,13	21
5	42	44	0,16	0,1	22
6	43	44	0,25	0,15	22
7	43	46	0,29	0,16	29
8	65	53	0,24	0,18	67
9	80	75	0,25	0,24	249
10	137	113	0,25	0,32	957

Figure 10 – Table des résultats.

Selon la table ci-dessus, nous remarquons que le taux d'erreur des points augmente quand la fonction est compliquée (nombre de points proches de la frontière élevé), les taux d'erreurs sur les points de frontières ne sont pas beaucoup changés. Après manipulation sur

Scilab, nous avons trouvé le temps de calcul pour les fonctions compliqué est très long, donc nous avons besoin de transmettre les codes à un autre langage. D'ici, on a choisi Java.

4.2 Deuxième version (en Java)

4.2.1 Pourquoi Java

A cause des limites suivantes de Scilab, nous ne pouvons pas faire des tests avec un nombre de points assez suffisant.

- L'espace mémoire : Nous avons besoin de tester notre méthode avec des millions de points, l'espace mémoire réservé pour Scilab ne suffit pas.
- Le temps de calcul : la matrice M est une matrice carrée de dimension taille de base, chaque coefficient de M est une calcul de : (nombre de base + nombre de points frontière) * (nombre de base + nombre de points frontière). Si nous supposons que le nombre de base est égale au nombre de points proche de la frontière égale lui-même à 1000, la complexité de calculer matrice M est $1000*1000*(2000*2000) = 4^{12}$, le temps de calcul de Scilab pour cet exemple est plus de 3 heures, mais en Java, il prend seulement 10 seconde. La figure 11 est une représentation du temps de calcul en fonction du nombre de points de base (l'exemple selon la figure 9)). Nous voyons que le temps de calcul croît rapidement quand le nombre de points de base augmente.

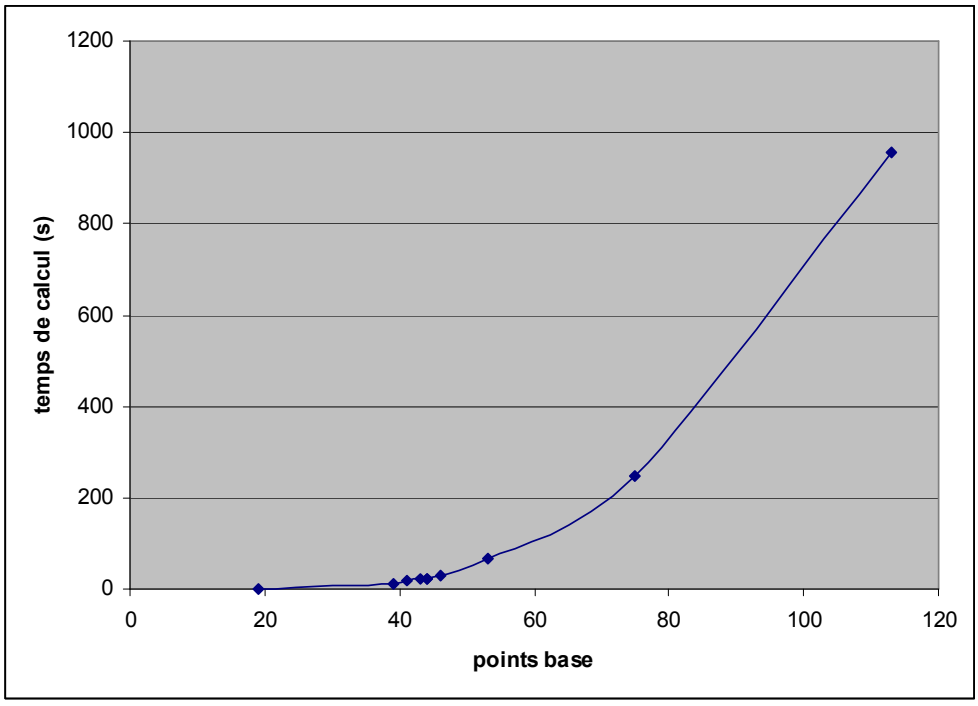


Figure 11 – Le temps de calcul en fonction des points de base.

- Nous remarquons que les résultats ne sont pas intéressants si nous n'avons pas beaucoup de points par dimension, car avec très peu de points par dimension, si nous avons rencontré une fonction assez compliquée, nous pouvons avoir presque tous les points de grille qui sont les points de frontière. Nous avons aussi une raison très important est: Intégration la méthode avec Kaviar. Le LISC a développé une première version d'un outil logiciel nommé KAVIAR qui permet d'approcher des noyaux de viabilité et de calculer des politiques d'action permettant au système de rester dans son ensemble de contraintes. Il permet aussi de calculer des ensembles de points résilients et des politiques d'actions résilientes. Ce logiciel fait appel à des techniques d'apprentissage "Support Vector Machines" (SVMs). La programmation du modèle dans le logiciel est faite en JAVA. Ce logiciel existe deux applications, le mode graphique et le mode batch. La figure 12 représente la fenêtre principale de Kaviar.

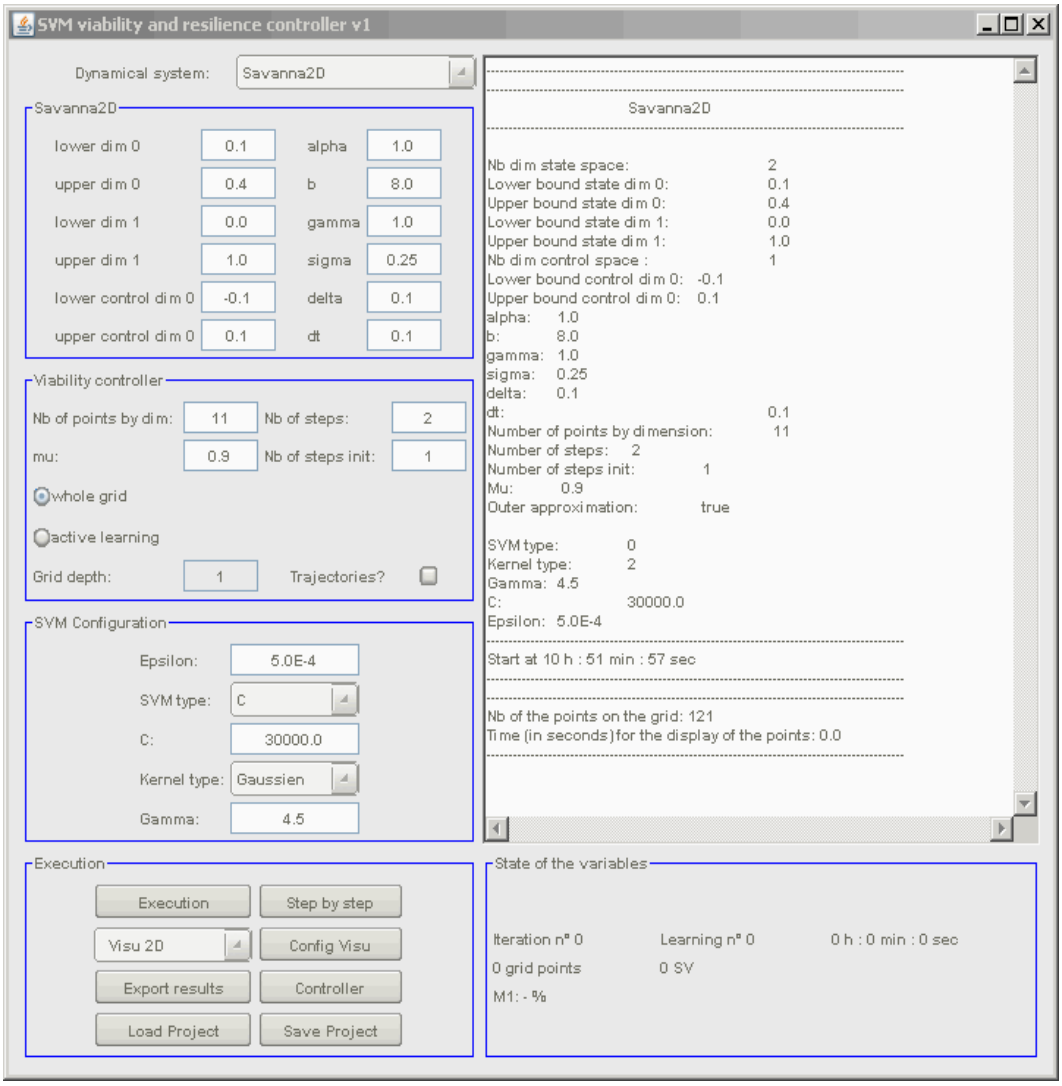


Figure 12 - Fenêtre principale de KAVIAR

4.2.2 Programmation

Pour facilement intégrer notre code dans Kaviar, nous hiérarchisons les classes qui sont déjà existantes dans Kaviar, et ajoutons quelques fonctions dont nous avons besoins pour notre méthode. Nous avons utilisé la classe ***Point*** qui contient trois attributs dans Kaviar:

- *nbDim_:* nombre de dimension.
- *vect_[]_:* valeur de chaque dimension.
- *label_:* l'indice de point.

Le code de classe ***Point*** de Kaviar se voit dans l'annexe 3. Nous avons ajouté des attributs suppléments pour notre cas étude:

- *dist_:* distance par rapport frontière.
- *f_approxi_:* valeur approximation.
- *epsilon_:* valeur S_k pour les points de base.

Et quelques fonctions :

- *public static int distance(Point inA, Point inB)* : calculer la distance entre deux points.
- *public static int distanceToFrontier(Point point, List<Point> Point_frontier)* : calculer la distance d'un point avec la frontière.
- *public static float choice_function(Point A, Point B, float epsilon)* : choisir la fonction de la base (d'ici $f_k(x) = \exp(-\frac{\|x - v_k\|^2}{s_k})$).

Nous avons créé une classe ***Scenario*** qui possède les attributs :

- *points_* : liste des points de la grille.
- *pointsProche_* : liste des points de la frontière.
- *liste_b_* : liste des points de base.
- *liste_add_* : liste de *pointsProche* et *liste_b_*.
- *step_* : pas (la distance entre deux points proches).
- *nb_point_dim_* : nombre de points par dimension.

Nous transmettrons toutes les fonctions de Scilbab (4.1.1) dans la classe Scenario. Ensuite nous avons comparé des résultats obtenus par Scilab et Java, pour s'assurer nous avions les mêmes résultats dans les mêmes exemples.

Nous avons aussi défini une classe ***GeneratorGrille*** pour créer les grilles par défaut, et la grille de points de base. Il contient 4 attributs suivants :

- *m[][]:* les coordonnées des points de la grille.
- *nb_dimension*: dimension de m.
- *nb_col:* nombre de colonne de m.
- *label_points[]:* les étiqueteur des points.

m[][] est une tableau de deux dimensions avec la taille *nombre de points * dimension*. C'est-à-dire chaque colonne de m correspond un point, *nb_dimension* est la dimension des points, *nb_col* est le nombre de points, la table *label_points* de taille égale à *nb_col* contient les étiquettes des points. La grille construit avec la constructeur de classe *public GeneratorGrille(int nb_dim, int nb_point_dim)* et une méthode *public void setLabel_points(float pos[][], float neg[][], int nb_point, int nb_proche)* pour générer les étiquettes.

4.2.3 Développement d'algorithme

A cause du temps de calcul de l'inverse de la matrice M, nous essayons toujours dans les développements algorithmiques de trouver les points de base en nombre suffisant pour construire un estimateur correct de la fonction, mais pas en nombre trop élevé pour limiter les temps de calculs.

Après une série de tests en augmentant le nombre de point par dimension, nous avons obtenu les résultats présentés dans la figure 13. Nous avons construire une fonction avec 201 points par dimension en dimension deux, les points bleus sont les points frontières, les points roses sont les points de base trouvés par notre algorithme et les points jaunes sont les points où il y a des erreurs de signe (toutes les prochaines figures auront la même légende). Les paramètres d'algorithme *choisi_base* (3.2.3) est: 4 *points par dimension par défaut,* 3 *itérations* et 4 *facteurs*. Dans la figure, nous voyons qu'il y a des bulles qui apparaissent dans les endroits où il n'y a pas beaucoup de point de base. Pour évider cette erreur, nous essayons de modifier les paramètres d'algorithme, on donne alors 8 points par dimension à la place de 4. Après le changement, nous arrivons à enlever les bulles dans notre exemple. Selon cette modification, nous supposons que nous avons besoin de suffisamment de points par défaut, pour que les points de base puissent bien couvrir tous les domaines où nous voulons étudier.

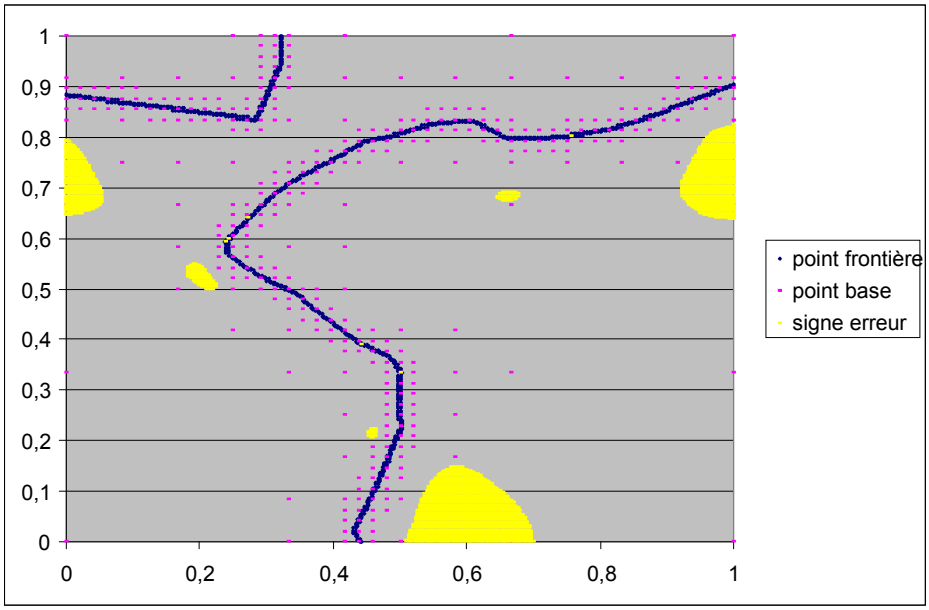


Figure 13 – L'erreur des bulles.

Suivant les modifications des paramètres précédents, nous testons avec des exemples différents et nous ne sommes pas toujours satisfaits des résultats obtenus. On a donc vérifié notre algorithme et le code, on a trouvé les problèmes suivants :

a – Valeur estimation de java

Dans notre structure des points (voir 3.2.1), nous supposons que toutes les coordonnées des points sont entre 0 et 1 pour faciliter les calculs. Cependant les ordinateurs enregistrent les données sous la forme binaire, les opérations mathématiques de *float* et *double* ne donne pas une valeur exacte. Par exemple, la solution de 0.02 – 0.01 nous arrivent à 0.0100000003. Dans notre cas d'études, nous savons que le nombre d'opérations pour calculer la matrice M est énorme (cf 4.2.1). Donc chaque calcul est fourni avec une petite erreur, après des millions d'opérations, nous allons obtenir une grande erreur. Pour éviter ces erreurs, nous proposons de définir les coordonnées de points avec des valeurs entières. Nous commençons par 0 et chaque points suivant ajoutons 2, car la distance de points frontière est égale à la distance entre deux points proches, donc d'ici leur distance est égale à 1.

Nous avons trouvé la manière de coder la valeur de coordonnées des points de la grille. Mais la valeur de la distance entre points ne peut s'écrire sous forme d'entier. Java nous propose une classe ***BigDecimal*** pour résoudre ce problème. Mais la constructeur de ***BigDecimal*** est : *BigDecimal(String val)*. C'est-à-dire que nous somme obligés de passer les paramètres en string, puis le transmettre à *float*. Imaginons, si nous voulons faire une opération d'addition, nous avons besoin de transmettre les deux valeurs flottantes au *String*, puis le construire en classe *BigDecimal*, faire l'addition, ensuite obtenir le résultat *BigDecimal* le transmettre en valeur flottante. Toutes ces opérations sont compliquées et lourdes, donc nous proposons une classe ***Arith*** pour simplifier celles-ci. Il fournit les méthodes statiques suivant qui contient toutes les opérations d'addition, soustraction, multiplication, division et aussi une méthode d'arrondir. La ressource d'***Arith*** est présentée dans l'annexe 4.

public static double add(double v1,double v2)
public static double sub(double v1,double v2)
public static double mul(double v1,double v2)
public static double div(double v1,double v2)
public static double div(double v1,double v2,int scale)
public static double round(double v,int scale)

b – Distribution des points de base

Le deuxième problème que nous avons souvent rencontré est le suivant : il y a plus d'erreurs autour des fonctions compliquées. L'exemple est probant dans la figure 14, dans l'endroit tournant où nous avons souvent des erreurs du signe qui apparaissent.

Nous savons que l'algorithme de *Choisi_base*(3.2.3) nous propose une manière de chercher les points de base uniformément autour des fonctions, il ne distingue pas les fonctions compliquées des fonctions lisses. Mais l'objectif de la méthode est de fournir plus de bases autour du domaine où la fonction est compliquée et moins de bases autour du domaine où la fonction est lisse. Nous proposons une autre méthode pour améliorer notre algorithme de choix des points de base.

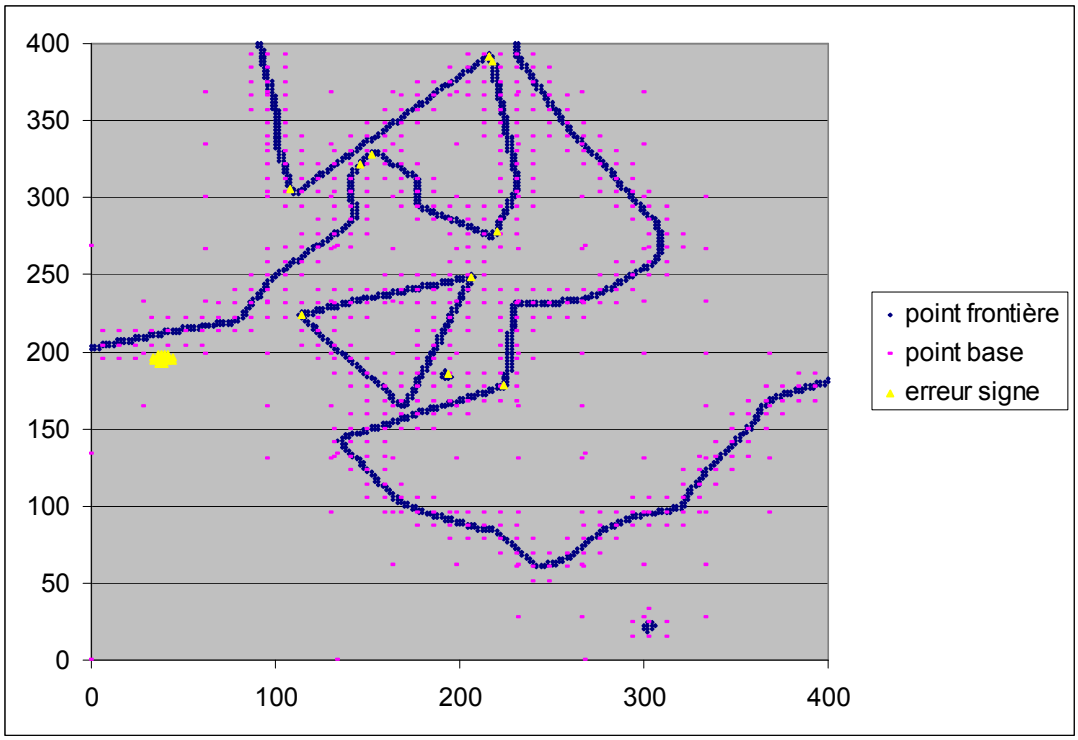


Figure 14 – Distribution de l'erreur de signe, les paramètres pour algorithme de *choisi_base* est: 4 points par dimension, 3 itération et 4 facteur.

La figure 15 représente une grille avec des points positifs et négatifs, les points de la frontière de la fonction compliquée : pour chaque point, le nombre de voisins de label opposé est supérieur à 1. Nous proposons donc une autre méthode pour chercher les points de base : ajouter tous points frontières qui possèdent ses points dans la liste de points de base. Cette méthode nous l'appelons : ***Méthode de sélection des bases sur la frontière***. Maintenant, notre liste de points de base est construite par deux manières, une est la méthode on vient de proposer, l'autre est l'algorithme de *choisi_base(3.2.3)*. Ces deux méthodes se complètent l'une et l'autre, car l'algorithme de *choisi_base(3.2.3)* propose une manière de choisir les points de base pour bien couvrir la surface, mais il ne peut pas distinguer la complexité de la fonction. La nouvelle méthode trouve plus de bases pour les fonctions compliquées. En plus, elle ne prend pas beaucoup de temps, car tous les points de base trouvés selon elle, nous pouvons les trouver en même temps que la recherche des points de frontière, ce qui représente un avantage très intéressant pour nous.

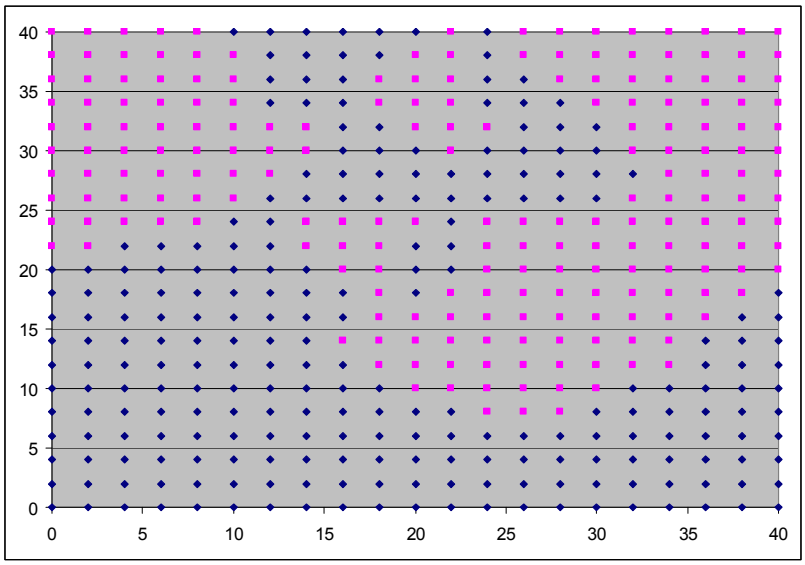


Figure 15 – Un ensemble des points positifs et négatifs. Les points roses sont les points positifs, les points bleus sont les points négatifs.

c – Les erreurs des positions de point base

Notre algorithme *choisi_base(3.2.3)* reconstruit une grille avec des paramètres données. Pour tous les points de base nous avons besoin de calculer leur distance à la frontière. Cette distance nous la calculons avec la méthode suivante : nous calculons la distance de ce point au point frontière plus proche, si ce point frontière est positif, nous faisons plus *pas*/2, si ce point frontière est négatif, nous faisons moins *pas*/2. Mais il est possible que ces points de base soient entre deux points frontières proches. L'exemple de la figure 16 illustre ces propos : dans le domaine dans le cercle, nous avons un point de base (point vert), le point de frontière plus proche est en base à gauche et il est positif. Nous supposons que la distance entre ces deux points égale à 0.4, mais avec notre fonction pour calculer la distance il va faire 0.4 + *pas*/2, après la modification précédente notre *pas* égale à 2, donc la distance va être égale à 1.4.

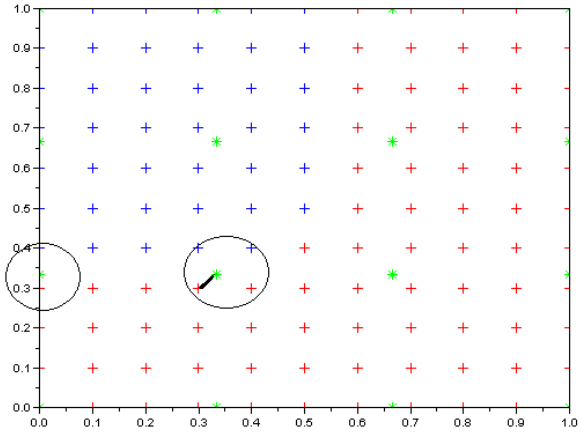


Figure 16– L'erreur des positions de point base.

Pour évider cette erreur, on va modifier l'algorithme de *choisi_base*. Grâce à la modification de la distribution des points de base, nous avons déjà des points de base qui sont proches des la frontières (selon la *méthode de sélection des bases sur la frontière*, tous les points de base trouvés sont les points de la frontière), donc l'algorithme de *choisi_base* n'a pas forcement besoin de chercher les points très proches de la frontière, on va ajouter une condition : on n'enregistre que les points de base si leurs distances à la frontière sont supérieures à un seuil, bien sûr ce seuil doit être supérieur à *pas*/2. Après une série de test, nous avons fixé cette valeur égale à 4. Le nouvel algorithme s'appelle ***choisi_baseII***.

```
Algorithme : Choisi_baseII
Données : M // liste des points de frontière.
          Dimension : // nombre de dimension.
          // une fonction pour générer une grille avec des paramètres donné.
          Generateur_grille(nb_points, dimension)
Constante : Nb_points // nombre de points par dimension par défaut.
            It // nombre de itération.
            Facteur // nombre de facteur.
Variables : bk, // valeur vectorielle.
            Sk. // Réel.
            Valeur. // nombre de points par dimension en courant.
Résultat : B // liste de base.
Début
// construire la première grille de base
        Valeur = nb_points;
        Grille = Generateur_grille(nb_points, Dimension);
        Mise à jour les valeurs de bk et Sk;
        B ← tous les points de Grille;
        Pour i = 2 à It faire
                Valeur = Valeur*Facteur-1;
                Grille =  Generateur_grille(Valeur, Dimension);
                Mise à jour les valeurs de bk et Sk;
                Pour tous les points Pk ∈ Grille faire
                        If ((distance(bk, M)<Sk)&&(distance(bk, M)>4))
                                B ← Pk;
                        Fin if
                Fin pour
                i = i+1;
        Fin pour
Fin
```

4.2.4 Observations

Après les modifications, nous avons lancé le programme sur différentes fonctions. Selon les résultats obtenus, nous pouvons fixer les paramètres d'algorithme *choisi_baseII*. La valeur de *nb_points* (nombre de points par dimension par défaut) est 8, car nous avons besoin d'assez de bases pour bien couvrir toute la surface. La valeur d'*itération* est 2, car nous avons déjà des points de base qui sont proches de la frontière, donc nous n'avons pas besoin d'aller plus finement, nous pouvons aussi gagner du temps de calcul et limiter le nombre de bases

aussi. La valeur de *facteur* égale à 4. La figure 16 représente un résultat obtenu avec l'algorithme modifié. Il n'y a plus d'erreur de signes.

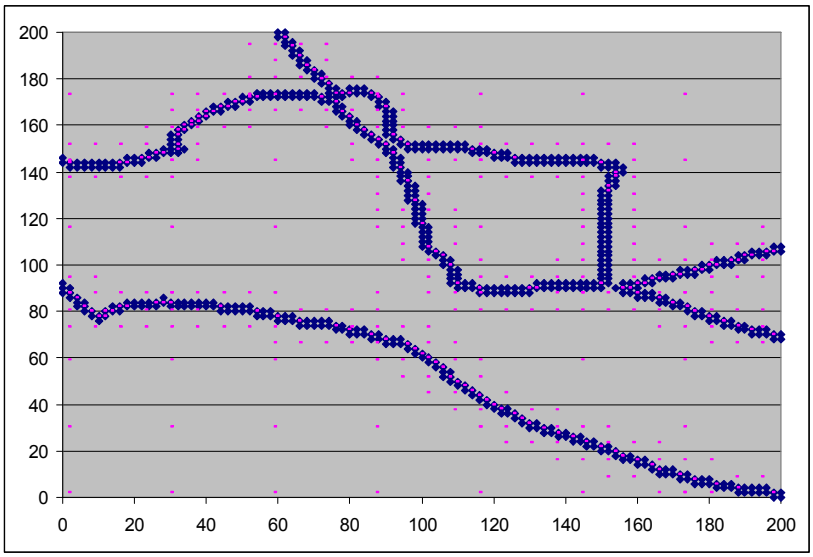


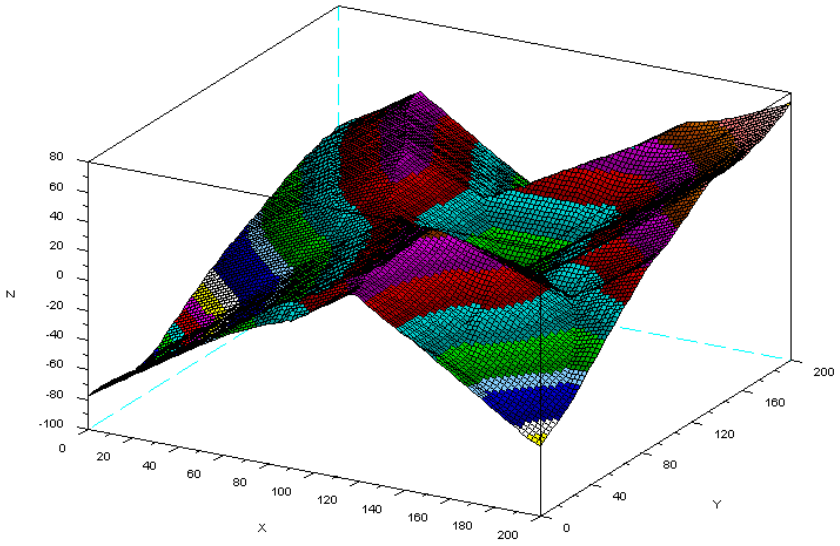
Figure 17– Résultats de la nouvelle méthode.

Dans la figure 17, nous voyons des différences par rapport aux résultats précédents, il y a des bases un peu partout dans la surface, et plus de points de base où la fonction est compliquée. Le tableau suivant nous donne quelques tests sur dix fonctions aléatoires. Tous les exemples sont en dimension deux et 100 points par dimension. La première colonne est le nom de fonction, la deuxième colonne est le nombre de points de la frontière, la troisième colonne est le nombre de points de base, la quatrième colonne est le temps de calcul pour calculer la matrice M, car c'est le temps s'agit le nombre de point de base, cette valeur nous avons obtenu avec l'ordinateur a Processeur double cœur Intel Pentium T4200 (2 GHz, 800MHz, mémoire cache de 1Mo, mémoire de 3Go), la cinquième colonne est le taux d'erreur de la signe, la sixième colonne est le taux d'erreur des points frontière, la septième colonne est le taux d'erreur de tous les points de la surface. Dans le tableau, nous voyons que la moyenne du taux d'erreurs des points frontières et du taux d'erreur de tous les points sont inférieur à 20 pourcent, et le taux d'erreur de la signe est inférieur sont très peu. Donc, généralement nous avons obtenus de bonnes solutions.

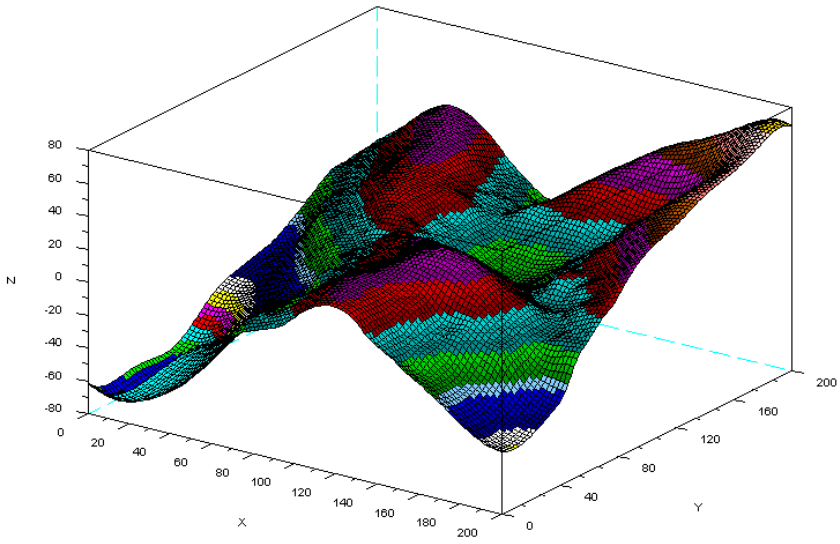
Fonction	Points frontière	Points de base	Temps pour M (ms)	Taux erreur signe (%)	Erreurs des points frontières (%)	Erreur des points (%)
1	475	391	536	0	14.75	10.37
2	751	607	2008	0.17	14.43	17.42
3	751	549	1556	0	15.46	16.89
4	672	443	929	0	18.04	13.86
5	620	483	1085	0	14.56	17.30
6	646	552	1472	0	14.09	15.65
7	642	512	1252	0	13.96	15.39
8	384	300	266	0.06	16.14	14.15
9	444	389	569	0.17	14.43	17.42
10	568	412	1256	0.02	15.01	15.34

Figure 18– Les résultats après modification de l'algorithme.

La figure 19 représente la fonction de la figure 16 en 2D, l'axe X et Y représente la position des points, l'axe est la fonction de la distance à la frontière. Le graphe (a) est la fonction nous voulons obtenir, et le graphe (b) est le graphe approximation.



(a)



(b)

Figure 19– Exemple de fonction à obtenir (a) et son approximation (b).

La figure 20 est un graphique des nuages de points, qui représente le temps de calcul en fonction du nombre de points de base. Nous avons lancé une simulation avec 100 fonctions aléatoires à apprendre, nous voyons que le temps de calcul augmente très vite quand le

nombre de points de base augmente. Nous devons donc bien limiter ce nombre de points de base.

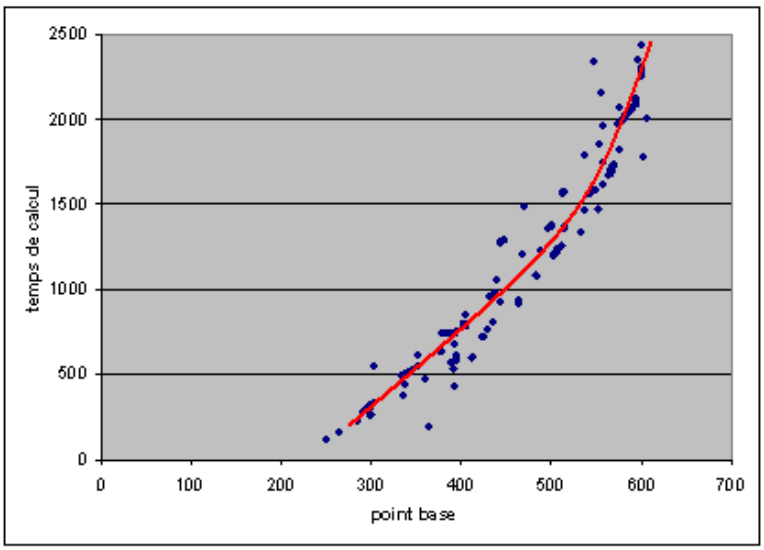


Figure 20 -Nuages de points, le temps de calcul en fonction de points de base.

Conclusion

Bilan

Les objectifs de mon stage ont été atteints. En effet, j'ai conçu et réalisé une application fonctionnelle qui permet de reconstruire une fonction avec l'aide les points de base trouvés par l'algorithme proposé. Cet algorithme se base sur la méthode des ensembles de Splines avec des largeurs de bandes différentes. Le sujet concerne beaucoup de domaines des mathématiques, à cet égard, j'ai rencontré quelques difficultés à comprendre les conceptions de la méthode. Grâce à l'aide de mes encadrants, je suis bien rentré dans le sujet.

Pour tester l'algorithme, j'ai commencé à l'implémenter par logiciel Scilab, car Scilab est un logiciel facile à utiliser, et il propose un outil graphique performant. Grâce à ces avantages nous pouvons facilement et rapidement de visualiser nos résultats puis réaliser des modifications algorithmiques s'il y a des erreurs. Nous avons obtenu une première version d'algorithme après avoir étudié les résultats avec Scilab. Le but de stage est de résoudre les erreurs de l'utilisation des SVMs, le LISC les utilisant pour calculer une approximation du noyau de viabilité d'un système dynamique. Leur logiciel étant basé sur des algorithmes d'apprentissage développé en Java, j'ai alors transmis notre code en java. Pendant cette période, j'ai rencontré des différents problèmes, venant de l'algorithme, ou de problème de codage. Après la résolution des problèmes, j'ai fait une série de teste pour fixé les paramètres de l'algorithme. Enfin, j'ai obtenu des résultats satisfaisant en dimension 2.

Pendant le période de stage, j'ai approfondi mes connaissances de langage orienté et en applications mathématiques. Cette expérience était pour moi, celle qui répond le mieux à mes attentes et mes aspirations.

Perspectives

Les résultats obtenus par notre algorithme proposé peuvent reconstruire en général approximée notre fonction, mais nous avons quand même encore des problèmes. L'exemple de la figure 21, qui représente une fonction assez compliquée, les points de base trouvés par notre algorithme sont bien distribués sur les fonctions vectorielles et horizontales, mais ils sont mal distribués sur les fonctions obliques, il y a trop de points de base dessus. Donc nous devons améliorer notre algorithme pour encore limiter le nombre de points de base, et gagner ainsi du temps de calcul en même temps.

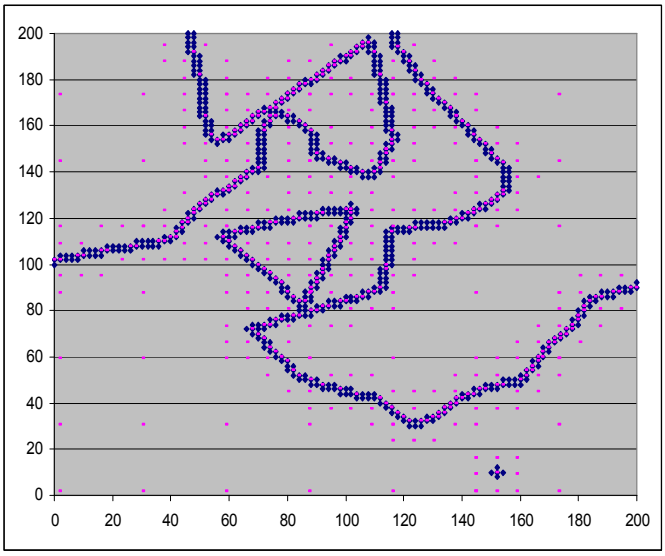


Figure 21– Une fonction compliquée.

La figure 22 est une cartographie des erreurs : les points de couleur bleue sont les points dont les erreurs sont inférieures à 0.3, de couleur jaune dont les erreurs se situent entre 0.3 et 0.6, et de couleur rouge est les erreurs sont supérieures à 0.6. Nous voyons que les erreurs sont rassemblées dans certains domaines, nous devons donc donner plus de points de base dans ces domaines. Nous disposons un autre exemple de cartographie dans l'annexe 5.

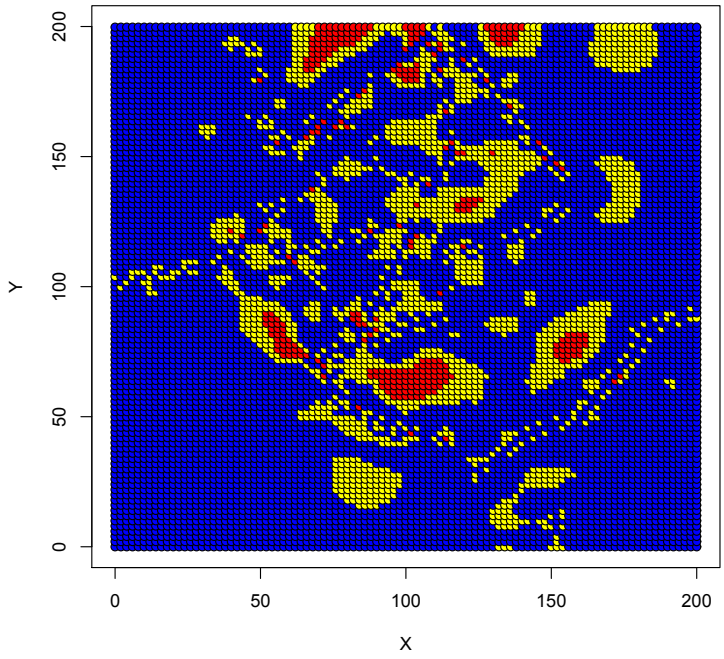


Figure 22– Cartographie de ligne des couleurs, bleu est les erreurs inférieur à 0.3, jaune est les erreurs entre 0.3 et 0.6, rouge est les erreurs supérieur à 0.6.

Par ailleurs, cette méthode doit être opérationnelle en dimension supérieure à 2. Pour l'instant, ce n'est pas le cas. Ainsi, nous donnons un résultat d'un exemple de dimension 3, qui a 51 points par dimension :

Nombre de points :	132651.
Nombre de points proche de la frontière :	7728.
Nombre de points de base :	5729.
Temps de calcul matrice M :	1642911 (ms)
Temps de calcul matrice Z :	325 (ms)
Temps de calcul total :	38 minutes
Taux d'erreurs du signe :	0.08%
Taux d'erreurs des points proche de la frontière :	18.67%
Taux d'erreurs des points :	14.14%

Selon les données ci-dessus, nous remarquons que lorsque le nombre de dimension élevé, le nombre de points, le nombre de points proches de la frontière et le nombre de points de base augmentent trop vite. Nous avons donc des problèmes avec les espaces mémoires et le temps de calcul. Mais nous avons presque les mêmes résultats par rapport ce que nous avons vu en dimension deux, donc la même remarque que nous avons vu en dimension deux : limiter le nombre de points de base pour gagner le temps de calcul et l'espace mémoire. Enfin, si nous arrivons à résoudre tous ces problèmes, nous intégrerons cette méthode à KAVIAR.

Bibliographie

[1] WALDER C., SCHÖLKOPF B. CHAPELLE O. Février 2006: Implicit Surface Modelling with a Globally Regularised Basis of Compact Support.

[2] BENNANI Y., ROUVEIROL C. Mai 2009 : Conférence D'Apprentissage.

[3] DEFFUANT G., MARTIN S., 2004: Approximating Viability Kernels with Support Vector Machines.

[4] GANDAR B., Avril 2009: Eléments de réflexion sur le stage. Rapport interne du LISC, Cemagref.

[5] WEI W., 2008 : Viabilité et résilience de dynamiques environnement et économiques définies à partir de modèles simplifiés.

[6] CHAPEL L., 2007 : Maintenir la viabilité ou la résilience d'un système : les machines à vecteurs de support pour ramper la malédiction de la dimensionnalité? Thèse de doctorat, Blaise Pascal 2007.

[7] CHAPEL L., 2008: SVM viability kernel approximation and résilience computation software version 1.2 User Guide.

[8] GANDAR B., 2008 : Les SVMs une utilisation des techniques d'optimisation quadratique en apprentissage statistique. Cours de seconde année de l'ISIMA.

[9] DEFFUANT, G., CHAPEL, L. et MARTIN, S. 2007: Approximating viability kernels with support vector machines. *IEEE transactions on automatic control,* 52(5):933-937.

[10] TONG, S. et KOLLER, D. 2000: Support vector machine active learning with application to text classification. Dans LANGLEY, P., éditeur : Proceeding of ICML-00, 17th Internationnal Conference on Machine learning, pages 999-1006, Stanford, US. Morgan Kaufmann Publishers, San Francisco, US.

[11] AUBIN, J.-P. 1997: *Dynamic economic theory : a viability approach*. Spring-Verlag.

Site Internet :

http://fr.wikipedia.org

http://users.info.unicaen.fr/~herve/cnrs_services_gratuits.pdf

http://vle.univ-littoral.fr/fr/index.php/Package_sensitivity

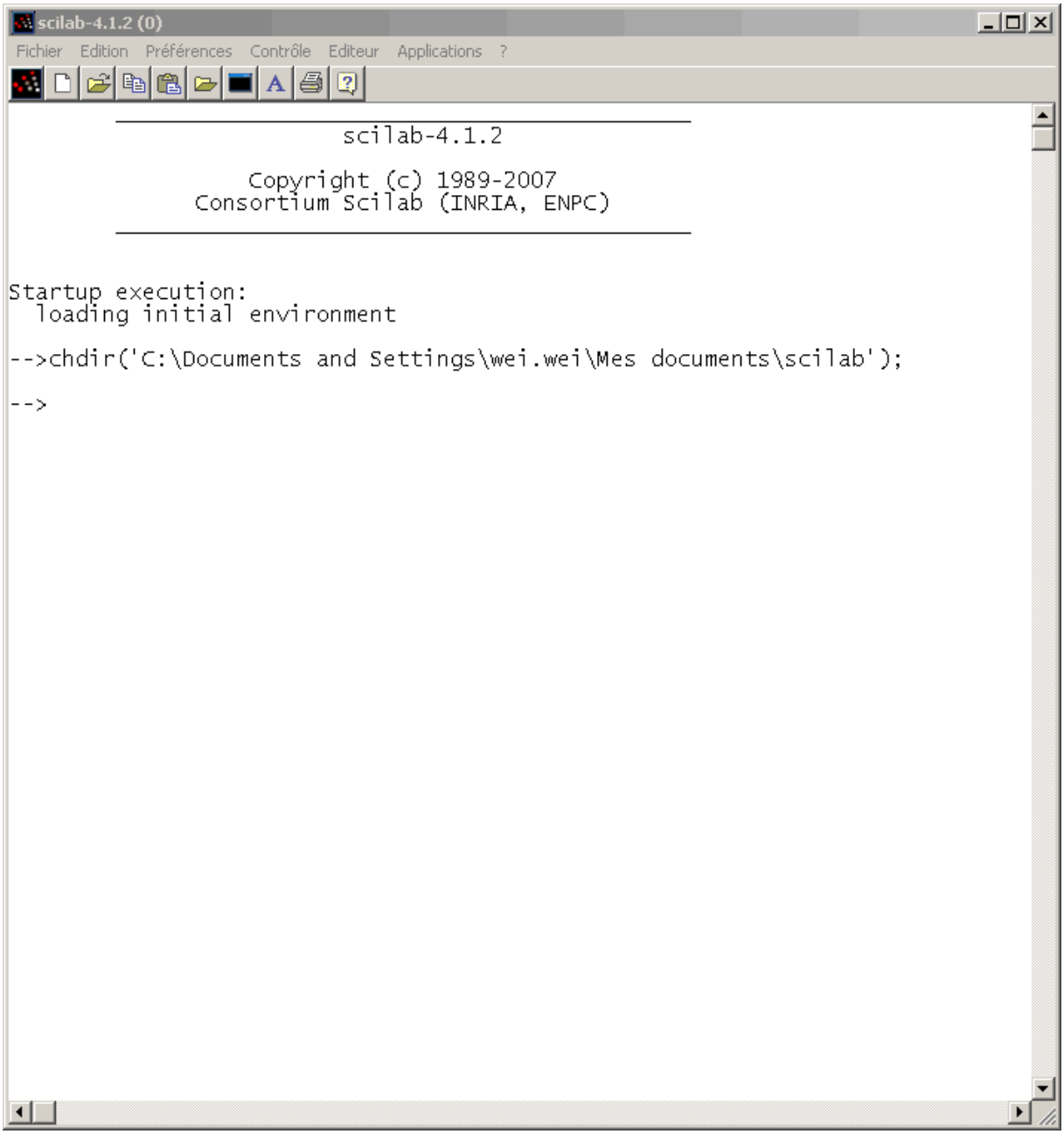
http://intranet.cemagref.fr/Communication/documents/powerpoi/diaporama_institutionnel_2007.ppt

http://www.grappa.univ-lille3.fr/~ppreux/ensg/m1psycho/manip.html

http://wiki.r-project.org/rwiki/doku.php?id=tips:graphics-3d:graphics-3d

Annexes

1. Interface du Scilab



2. Fonction principale de Scilab

```
//function principal avec les distances en ajoutant liste b dans les points
d'apprendissage
//30/4/9 weiWEI
function [lancer, lancer1, t, ajout_p, p, pi, b,
z]=main_bis3(nb_dim,nb_point,pos,neg,c,nb_depart_point,iter,base_fact,k)
  timer();
  pas=1/(nb_point-1);
  exec('generateur_grille.sci');
  g=generateur_grille(nb_dim,nb_point,2);
  exec('etiqueteur_deux.sci');
  t=etiqueteur_deux(g,k,pos,neg);
  exec('point_proche.sci');
  p=point_proche(g,t,nb_point);
  exec('index_point_p.sci');
  index=index_point_p(g,t);
  exec('liste_b.sci');
  b=liste_b(g,nb_depart_point,iter,base_fact,p);
  exec('ajouter_b.sci');
  ajout_p=ajouter_b(b,p);
  exec('calculer_f.sci');
  [c_c,d_d]=calculer_f(ajout_p,b,pas,p);
  exec('matrice_z_dis.sci');
  z=matrice_z_dis(c,ajout_p,b,pas,p,c_c,d_d);
  exec('matrice_m_dis.sci');
  m=matrice_m_dis(c,ajout_p,b,pas,p,c_c,d_d);
  pi=inv(m)*z;
  exec('approxi_f.sci');
  g_bis=generateur_grille(2,101,2);
  f=approxi_f(g,pi,b);
  f_bis=approxi_f(g_bis,pi,b);
  exec('aff_bis.sci');
  lancer=aff_bis(g,f,t,p);
  pourcent=size(find(lancer(:,3)==0),2)/(nb_point*nb_point);
  exec('transformer.sci');
  tran=transformer(lancer(:,2));
  trans_bis=transformer(f_bis);
  x=1:-1/100:0;
  y=0:1/100:1;
  xbasc();
  ttt=trans_bis';
  grayplot(x,y,ttt);
  //plot3d1(x,y,ttt);
  exec('affiche2D_f.sci');
  affiche2D_f(g,t,lancer);
  plot(b(1,:),b(2,:),'*g');
  err=sum(abs((lancer(:,1)-lancer(:,2))/lancer(:,1)))/size(lancer,1);

xsave('graphe3D'+'_err'+string(err)+'_'+string(pourcent)+'_'+string(nb_poin
t)+'_'+string(nb_depart_point)+string(iter)+string(base_fact)+'.scg');
```

```
  printf('result is : %f \n',
size(find(lancer(:,3)==0),2)/(nb_point*nb_point));
  dif=lancer(:,1)-lancer(:,2);
  som=0;
  for i=1:size(dif,1)
    som=som+abs(dif(i,1)/lancer(i,1));
  end
  printf('taux erreur total est : %f \n', som/size(dif,1));
  lancer1=lancer(index,:);
  dif=lancer1(:,1)-lancer1(:,2);
  som=0;
  for i=1:size(dif,1)
    som=som+abs(dif(i,1)/lancer1(i,1));
  end
  printf('taux erreur est sur les points proche: %f \n', som/size(dif,1));
  t=timer();
  printf('le temps de calculer est : %f \n', t);
endfunction;
```

3. Classe Point de Kaviar

```
package viabController;
import java.util.ArrayList;
import java.util.List;
import LibSVM.svm_node;
/**
 * Creates and deals with points
 *
 * @author Florent LOPEZ
 */
public class Point implements Cloneable
{
	/**
	 * Number of dimensions
	 */
	public int nbDim_;

	/**
	 * value for each dimension
	 */
	public float vect_[];

	/**
	 * Label of the point: true if it is viable
	 */
	public int label_;

	/**
	 * Constructs a point in nbDim dimension
	 *
	 * @param inDim dimension of the point
	 */
	public Point(int inDim){
		this.nbDim_ = inDim;
		this.vect_ = new float[ this.nbDim_ ];
	}
	/**
	 * Computes and returns the norm of the point
	 *
	 * @return norm
	 */
	public double getNorme(){
```

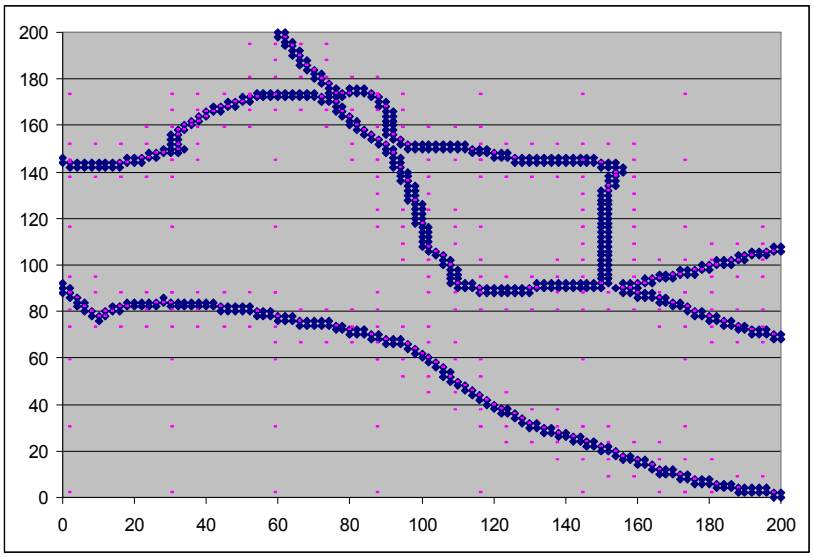
```
                double res = 0;
                for( int i = 0 ; i < this.nbDim_ ; ++i )
                        res += this.vect_[ i ] * this.vect_[ i ];
                return (Math.sqrt( res ));
        }
        /**
         * Computes and returns the distance between 2 points
         *
         * @param inA point 1
         * @param inB point 2
         * @return distance between point 1 and 2
         */
        public static double distance(Point inA, Point inB){
                double res = 0;
                for( int i = 0 ; i < inA.nbDim_ ; i++ )
                        res += (inA.vect_[i] - inB.vect_[i]) * (inA.vect_[i] - inB.vect_[i]);
                return Math.sqrt(res);
        }
        /**
         * Prints the point and its label
         *
         */
        public void print(){
                for( int i = 0 ; i < this.nbDim_ ; ++i )
                        System.out.print(vect_[i]+" ");
                System.out.println("label : "+label_);
                System.out.print("\n");
        }
        /**
         * Prints the point
         *
         */
        public void printPoint(){
                for( int i = 0 ; i < this.nbDim_ ; ++i )
                        System.out.print(vect_[i]+" ");
                System.out.print("\n");
        }
}
```

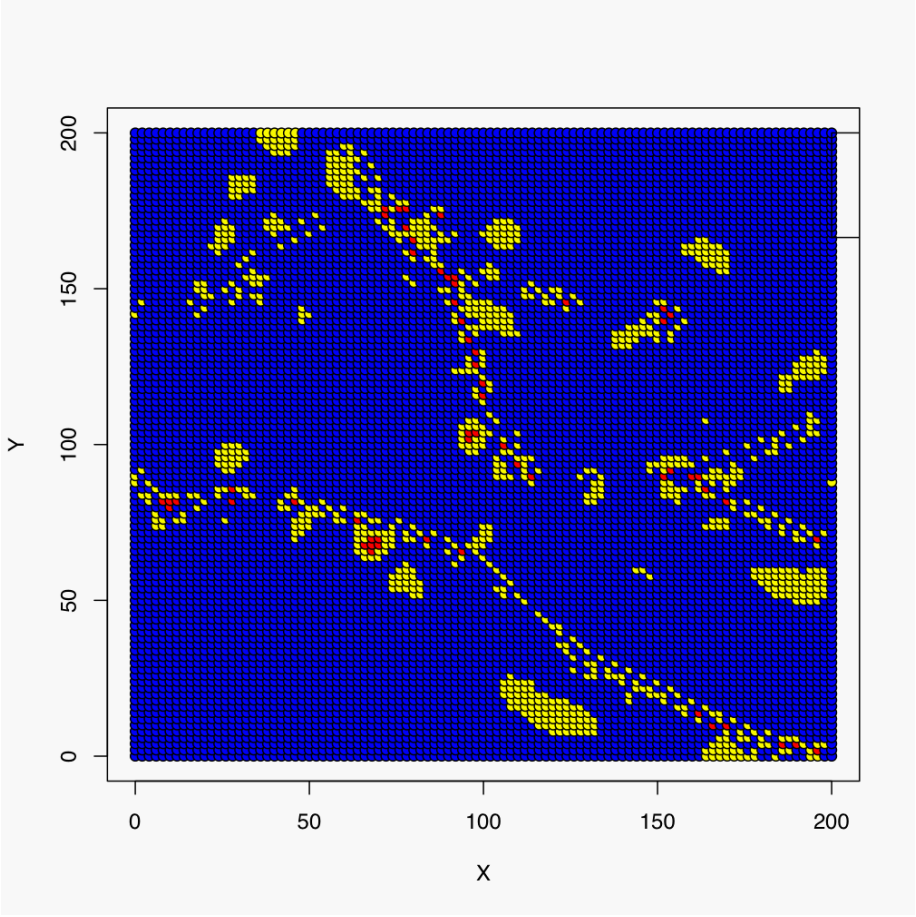
4. Ressource d'Arith

```
import java.math.BigDecimal;

public class Arith{

   //définir le nombre chiffir après virgule pour arrondir
   private static final int DEF_DIV_SCALE = 10;

   private Arith(){
   }

   /**
    * addition.
    * @param v1 cumulateur 1
    * @param v2 cumulateur 2
    * @return la somme
    */
   public static double add(double v1,double v2){
      BigDecimal b1 = new BigDecimal(Double.toString(v1));
      BigDecimal b2 = new BigDecimal(Double.toString(v2));
      return b1.add(b2).doubleValue();
   }

   /**
    * soustraction
    * @param v1: diminuende
    * @param v2: diminuteur
    * @return différence
    */
   public static double sub(double v1,double v2){
      BigDecimal b1 = new BigDecimal(Double.toString(v1));
      BigDecimal b2 = new BigDecimal(Double.toString(v2));
      return b1.subtract(b2).doubleValue();
   }

   /**
    * multiplication
    * @param v1: multiplicande1
    * @param v2: multiplicande1
    * @return le résultat
    */
   public static double mul(double v1,double v2){
      BigDecimal b1 = new BigDecimal(Double.toString(v1));
      BigDecimal b2 = new BigDecimal(Double.toString(v2));
```

```
        return b1.multiply(b2).doubleValue();
    }

    /**
     * division, si le dividende n'est pas divisible par nombre diviseur, garder 10 valeurs après
     * la virgule après avoir arrondir
     * @param v1: dividende
     * @param v2: nombre diviseur
     * @return le résultat
     */
    public static double div(double v1,double v2){
        return div(v1,v2,DEF_DIV_SCALE);
    }

    /**
     * division, si le dividende n'est pas divisible par nombre diviseur, garder scale valeurs
     * après la virgule après avoir arrondir
     * @param v1: dividende
     * @param v2: nombre diviseur
     * @param scale: nombre de valeur garder après la virgule
     * @return résultat
     */
    public static double div(double v1,double v2,int scale){
        if(scale<0){
            throw new IllegalArgumentException(
                "The scale must be a positive integer or zero");
        }
        BigDecimal b1 = new BigDecimal(Double.toString(v1));
        BigDecimal b2 = new BigDecimal(Double.toString(v2));
        return b1.divide(b2,scale,BigDecimal.ROUND_HALF_UP).doubleValue();
    }

    /**
     * arrondir
     * @param v: valeur à arrondir
     * @param scale: nombre de valeur garder après la virgule
     * @return résultat
     */
    public static double round(double v,int scale){
        if(scale<0){
            throw new IllegalArgumentException(
                "The scale must be a positive integer or zero");
        }
        BigDecimal b = new BigDecimal(Double.toString(v));
        BigDecimal one = new BigDecimal("1");
        return b.divide(one,scale,BigDecimal.ROUND_HALF_UP).doubleValue();
    }
};
```

5. Cartographie





Cartographie de ligne des couleurs: bleues sont les erreurs inférieures à 0.3, jaunes sont les erreurs entre 0.3 et 0.6, rouge sont les erreurs supérieures à 0.6.