Faculté des Sciences Appliquées

Implémentation d'une méthode de région de confiance pour le diagnostic industriel

Travail de fin d'études réalisé en vue de l'obtention
du grade de master Ingénieur Civil en Informatique

Damien Gerard damiengerard88@gmail.com

Promoteur
Professeur Quentin Louveaux

Année académique 2010 – 2011

Damien Gerard
Master Ingénieur Civil en Informatique
Année académique 2010-2011

Université de Liège

Travail de fin d'études

Implémentation d'une méthode de région de confiance pour le diagnostic industriel

Résumé

De nombreuses entreprises se spécialisent dans des processus industriels de pointe. Malheureusement, toute l'expertise acquise est parfois insuffisante face à la complexité et au caractère aléatoire des processus. Ce mémoire met de côté le procédé technique lui-même afin de se focaliser sur l'aspect mathématique et apporter des solutions aux entreprises. En particulier, ce mémoire s'intéresse à un problème de galvanisation.

Le but du mémoire est d'utiliser l'optimisation numérique afin d'améliorer un procédé de galvanisation. Ce processus dépend d'un grand nombre de paramètres, et nous souhaitons trouver des conjonctions de ceux-ci afin de maximiser les chances d'obtenir un produit de qualité satisfaisante au final.

À cette fin, nous mettons en œuvre une méthode d'optimisation par région de confiance afin de résoudre un problème non convexe et non linéaire. Chaque fois qu'elle produit une unité, l'entreprise en teste la qualité qu'elle associe aux valeurs des paramètres utilisés pour la production. Ce sont ces données que nous utilisons dans une méthode de région de confiance. Le mémoire traite d'approximation de fonction, de réduction de dimensionnalité afin de rendre l'optimisation applicable, et de recherche très rapide de points dans un espace à hautes dimensions.

Nous découvrirons que nous pouvons effectuer des recherches très rapides de données et qu'il est possible de trouver de bons points de fonctionnement. Cependant, nous mettrons en évidence la faiblesse face aux problèmes à hautes dimensions.

Remerciements

Je tiens à remercier en premier lieu Monsieur Quentin Louveaux, Professeur à l'Université de Liège et promoteur de ce travail de fin d'études. Merci pour ce sujet passionnant et multidisciplinaire. Je garderai un excellent souvenir de cette expérience.

Un grand merci à Louis Sumkay, grand connaisseur du processus de galvanisation. Il m'a permis de mieux cerner le procédé industriel, sa complexité et ses enjeux.

Merci à Sylvain Chêvremont qui a dessiné la plupart des figures de ce mémoire, et à Nicolas Mathieu pour ses conseils divers lors de la rédaction.

Pour terminer, je souhaite remercier mes parents pour la correction orthographique et tout leur soutien.

Table des matières

1 Introduction **7**
- 1.1 Segal . . . 7
- 1.2 Galvanisation . . . 7
- 1.3 Intérêt industriel . . . 7
  - 1.3.1 Défauts de galvanisation . . . 8
- 1.4 Optimisation . . . 9
- 1.5 Méthode de région de confiance . . . 10

2 Région de confiance **12**
- 2.1 Définition du modèle m_k . . . 13
  - 2.1.1 Rayon de recherche des points . . . 13
  - 2.1.2 Complexité du modèle . . . 14
  - 2.1.3 Régression unidimensionnelle . . . 14
  - 2.1.4 Régression multidimensionnelle . . . 15
  - 2.1.5 Résolution des équations normales . . . 16
- 2.2 Quantité de dimensions dans le problème . . . 16
- 2.3 Minimisation du modèle . . . 18
- 2.4 Fiabilité du modèle . . . 19
- 2.5 Minimum local . . . 19
- 2.6 Obtenir des minima intéressants . . . 20
  - 2.6.1 Saut pour sortir d'un minimum . . . 20
  - 2.6.2 Saut adaptable et scores des minima . . . 21
  - 2.6.3 Exemple complet . . . 21

3 Sous-problème de région de confiance **24**
- 3.1 Caractérisation du sous-problème . . . 24
- 3.2 Résolution exacte . . . 25
  - 3.2.1 *Modus operandi* . . . 25
  - 3.2.2 Comportement de la région de confiance selon λ . . . 25
  - 3.2.3 Calculer la racine de $\|s(\lambda)\|_2 - \Delta = 0$ . . . 27
  - 3.2.4 Intervalle d'incertitude . . . 29
  - 3.2.5 Valeurs de départ . . . 31
  - 3.2.6 Cas difficile . . . 32

3.2.7 Terminaison . . . 33
3.2.8 Algorithme . . . 34
3.2.9 Eigensolution . . . 35
3.3 Résolution approximative . . . 36
3.3.1 Gradients conjugués pour un problème convexe . . . 37
3.3.2 Adaptation au cas non convexe . . . 37
3.3.3 Gradients conjugués préconditionnés . . . 37

4 Recherche spatiale 38
4.1 Partitionnement de l'espace . . . 39
4.1.1 kd-tree . . . 39
4.1.2 Quadtree – Octree . . . 40
4.2 Partitionnement des données — M-Tree . . . 41
4.2.1 Structure du M-Tree . . . 41
4.2.2 Split . . . 41
4.2.3 Promote . . . 43
4.2.4 Partition . . . 43
4.2.5 Méthodes complètes de split . . . 43
4.2.6 Aller plus loin . . . 45
4.3 Projection vers un espace linéaire . . . 46
4.3.1 Space-filling curves . . . 46
4.3.2 Terminologie . . . 47
4.3.3 Représentation de la proximité . . . 47
4.3.4 Autres exemples de *space-filling curves* . . . 48
4.3.5 Exemple de construction d'une courbe : N-Curve . . . 48
4.3.6 GetNextZValue . . . 49
4.4 Structure d'indexage à une dimension . . . 54
4.4.1 Zarray . . . 54
4.4.2 UB-Tree . . . 54
4.4.3 Zhash . . . 59

5 Système de gestion de base de données 60
5.1 Propriétés des données . . . 60
5.2 MySQL . . . 61
5.2.1 MyISAM . . . 61
5.2.2 InnoDB . . . 61
5.2.3 Quantité de colonnes . . . 61
5.3 PostgreSQL . . . 62
5.3.1 Un SGBD pour la recherche spatiale . . . 62
5.3.2 Modèle objet-relationnel . . . 63
5.3.3 SQL/MM et recherche spatiale avec PostGIS . . . 63
5.4 SGBD et space-filling curves . . . 66
5.4.1 Intégrer la Z-Value dans un SGBD . . . 66
5.4.2 Intégrer le UB-Tree dans un SGBD . . . 67

6 Résultats **68**
6.1 Région de confiance 68
6.2 Sous-problème de région de confiance 71
6.3 Recherche spatiale 73

7 Conclusion **79**

Chapitre 1

Introduction

1.1 Segal

Segal est une entreprise d'environ 150 personnes, et fête ses 25 ans cette année. Cette entreprise est spécialisée dans la galvanisation haut de gamme pour l'automobile (capot, toit, portière...). Segal traite des tôles planes qui doivent non seulement atteindre la qualité technique exigée par le client, mais également n'afficher aucun défaut esthétique qui pourrait être visible par l'utilisateur final achetant une voiture. Segal ne produit pas des tôles ; l'entreprise se limite à la galvanisation de tôles laminées, des collaborateurs prenant en charge les autres étapes nécessaires dans la production complète d'une tôle.

1.2 Galvanisation

La galvanisation est le recouvrement de tôle par du zinc en fusion afin de la protéger contre la corrosion.

La tôle est cuite dans un four entre 950 ° et 1050 °, puis elle passe dans un bain de zinc en fusion. Quand elle en sort, la tôle est essorée par des jets. Ceux-ci soufflent sur chacune des deux faces afin de retirer une partie du zinc en fusion, contrôlant ainsi l'épaisseur de zinc finale. On peut également plier légèrement la tôle afin de faire passer une face plus près des jets, et ainsi avoir moins de zinc sur une face par rapport à l'autre.

À l'entrée du four, les tôles sont soudées les unes après les autres. Une fois galvanisées et refroidies, elles sont enroulées en bobines de 15 à 20 tonnes.

1.3 Intérêt industriel

C'est le client qui demande une épaisseur de zinc, ce que Segal doit respecter avec une certaine marge d'erreur. Cependant, contrôler l'épaisseur exacte de zinc est techniquement très difficile. La Figure 1.1 présente, par jet, le surplus de zinc en moyenne par rapport à l'exigence du client. (Il y a plusieurs jets en parallèle pour chaque face.) Lorsque le surplus est négatif, l'épaisseur de zinc est en deçà de celle escomptée, et la

qualité est moindre. Dans ce cas, Segal ne peut vendre une tôle de qualité inférieure et doit la déclasser dans une catégorie de qualité moindre, et le prix de vente sera de 10 à 20% moins cher. La tôle sera dès alors utilisée comme partie non visible dans une voiture (intérieur de garde-boue par exemple). Si au contraire, le surplus de zinc est positif, alors trop de zinc a été employé. Le client ne s'en plaindra pas car c'est au détriment de Segal. En effet, le zinc coûte très cher. Si l'épaisseur de zinc pouvait être parfaitement optimisée, Segal estime que le gain devrait être de 160 tonnes par an, soit 280000 € ([Seg11]).

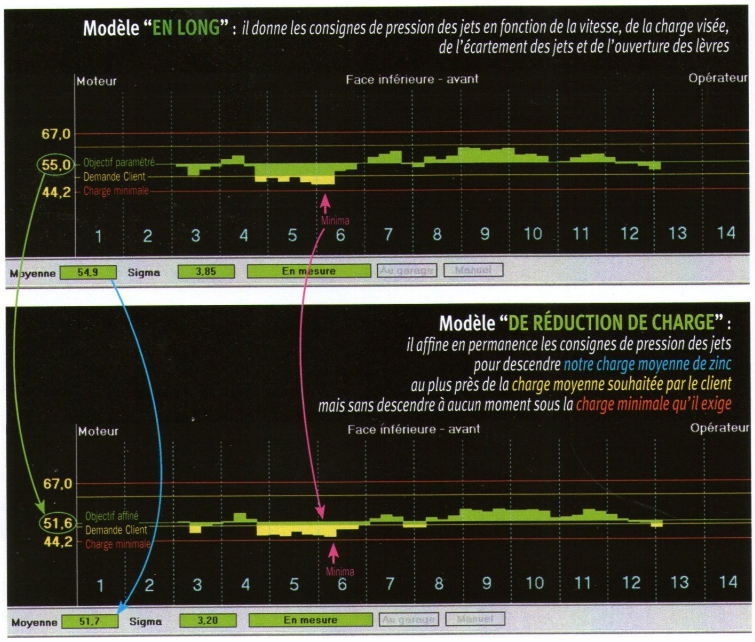


FIGURE 1.1 – Surplus de zinc par jet sur une face de tôle ([Seg11])

1.3.1 Défauts de galvanisation

De plus, Segal doit limiter les défauts de galvanisation. Ils sont la source d'une protection moins satisfaisante face à la corrosion, et peuvent également être visibles par le client, ce qui n'est pas acceptable. La plupart des défauts peuvent être mis en évidence en frottant la bande avec une pierre huilée. Segal utilise également une caméra afin d'en détecter. Celle-ci effectue une mesure de qualité tous les 25 mètres de tôle. De plus, un opérateur inspecte les faces inférieure et supérieure de la bande.

La plupart des défauts détectés sont :

- splashs (points brillants dans la tôle) ;
- griffes provoquées par les étaleurs qui servent à déplacer la bande ;
- points noirs ;
- mattes de zinc ;

– paillettes de zinc ;
– soufflures (trous) issus du laminage. □

Segal ne souhaite pas obtenir un modèle mathématique du processus de galvanisation. En effet, en ayant pratiqué la galvanisation durant 25 ans, l'entreprise a acquis une très grande expertise dans le domaine et a rassemblé de grandes quantités de données. Le souhait de Segal est de pouvoir les interpréter afin de trouver de bonnes conjonctures de paramètres favorables à une galvanisation de qualité optimale. Même si Segal sait détecter quels sont les défauts dans une tôle et connaît les réglages du processus de galvanisation qui ont conduit à ce défaut, l'entreprise ne sait cependant pas choisir les meilleurs paramètres qui permettraient d'éviter autant que possible les défauts tout en économisant le zinc. Cela fait une dizaine d'années que Segal tente de trouver une solution à ce problème, sans succès.

Signalons également que notre approche est mathématique et informatique. Elle n'est pas uniquement destinée à Segal. D'autres entreprises de galvanisation sont potentiellement concernées, et les notions abordées dans ce mémoire pourraient également être applicables à d'autres problèmes industriels.

1.4 Optimisation

La grande difficulté du problème provient de la quantité de paramètres, ce qui ne permet pas à la société de régler le processus de galvanisation aisément. Il a donc été proposé de faire appel à l'optimisation numérique afin de tenter de concevoir un procédé mathématique pour résoudre le problème.

Segal suppose que la qualité de la galvanisation dépend d'au moins 200 paramètres. Tous les 25 mètres de tôle, une caméra mesure automatiquement la qualité (splash, griffes, trous...) en un « point fictif », qui est un point dans un espace multidimensionnel, avec une dimension pour chaque paramètre. Les défauts représentent autant de variables de sorties, auxquelles sont associées les valeurs de tous les paramètres (variables d'entrées). Citons quelques paramètres à titre d'exemple :

– qualité du dégraissage de la tôle en entrée ;
– la tôle doit être extrêmement propre, et le four permet en partie de la nettoyer ;
– pollution du bain ;
– saletés dans les rainures des étaleurs (les étaleurs en ont afin d'éviter l'aquaplaning, mais elles se salissent inévitablement) ;
– température (chauffe, préchauffe, maintien et refroidissement car il y a 4 zones dans le four) ;
– ...

Il est probable que de nombreux paramètres inconnus entrent également en compte. Par exemple, il suffit que l'on ouvre un volet pour faire entrer un camion qui doit décharger, et de la poussière vient se poser sur la bande et peut la déclasser. La poussière n'étant pas mesurée, le déclassement sera attribué à d'autres paramètres. □

Nos données prennent donc la forme de points à 200 dimensions, et quelques variables binaires de sortie. À partir de ces points, nous souhaitons trouver des zones de fonctionnement minimisant les défauts. C'est très différent de la minimisation classique d'une fonction, où nous cherchons un point tel que la fonction est minimale en ce point. Dans notre application, la fonction vaut 0 ou 1 selon la présence d'un défaut ou non. Il y a donc de très nombreux minima. Ce que nous cherchons, ce sont des zones à forte concentration de minima. Ainsi, l'entreprise possèdera des conjonctures de paramètres conduisant à une bonne galvanisation, même si leur réglage n'est pas précis. Il faut également garder à l'esprit qu'il y a une certaine marge aléatoire dans la qualité de la galvanisation. Deux galvanisations effectuées avec le même jeu de paramètres n'auront pas forcément la même qualité.

C'est un problème d'optimisation non convexe et non linéaire, où la fonction objectif f est inconnue puisqu'exprimée au travers d'échantillons seulement. Ce mémoire étudie l'applicabilité d'une optimisation par région de confiance (*trust region method*) afin de tenter de solutionner le problème de Segal.

1.5 Méthode de région de confiance

Afin de pouvoir optimiser une fonction f très difficile (non convexe et non linéaire, dans notre application), la méthode par région de confiance travaille dans une petite sphère que l'on appelle « région de confiance ». À l'intérieur de celle-ci, on y définit un modèle qui représente bien la fontion f à l'intérieur de la sphère. Ce modèle est construit de manière à être facilement minimisable, en tout cas nettement plus que f. Le modèle est donc optimisé à l'intérieur de cette sphère, et puis une nouvelle sphère est dessinée autour de ce nouveau minimum.

Ensuite, la procédure est répétée : création d'un modèle simple, minimisation dans la nouvelle sphère, sphère centrée sur le nouveau minimum etc.

La notion première de la méthode est sa région de confiance. La région de confiance est une sphère dont le rayon traduit la *confiance* qu'on l'on a envers les modèles créés. Plus clairement, si nous parvenons à créer des modèles qui respectent bien f, alors la région de confiance s'agrandit parce que la confiance est plus importante. Inversement, si les modèles créés ne correspondent pas à f, notre confiance diminue et la région de confiance se rétrécit.

Dans le chapitre 2, nous allons commencer par étudier la méthode de région de confiance elle-même. Cela concerne la gestion de sa région de confiance, mais également des problématiques très spécifiques :

- étude de l'applicabilité de la méthode de région de confiance à notre application ;
- adaptation pour fonctionner même sans la connaissance de f (c'est sans précédent dans la littérature scientifique) ;
- plutôt que la recherche d'un minimum classique, nous rechercherons des zones larges de minimisation.

Ensuite, dans le chapitre 3, nous allons un peu plus en profondeur et étudions la minimisation d'un modèle à l'intérieur d'une région de confiance, et implémenterons plusieurs solveurs.

Comme f est inconnu et que nous n'avons que des points comme données, nous devrons utiliser ces derniers pour construire les différents modèles au cours de l'algorithme. Nous devrons sélectionner certains points parmi tout l'ensemble de points. C'est de la recherche à hautes dimensions, que nous devons rendre performante. Le chapitre 4 est entièrement dédié à cette tâche.

Même si ce n'est a priori pas le problème de base, une attention doit être apportée à la gestion des données. Segal parle de 5 millions de points fictifs. En effet, l'entreprise détermine un point fictif tous les 25 mètres de tôle, et cela depuis de nombreuses années. Chaque point ayant 200 dimensions en précision flottante, le tout pèse 8GB. Étant donné que nous devrons accéder à ces points à de très nombreuses reprises durant l'optimisation, il y a lieu de se demander comment nous devrions les gérer. Le chapitre 5 est dédié à cette problématique.

Chapitre 2

Région de confiance

Notre but est de minimiser une fonction objectif $f(x)$ non convexe et non linéaire. Le problème est non contraint. Nous cherchons un minimum local à $f(x)$.

Le procédé par région de confiance consiste à produire une séquence d'itérés $\{x_k\}$ que nous espérons voir converger vers un point critique du premier ordre, voire un minimum (local).

Cependant, une différence majeure entre notre application et toute la littérature scientifique traitant des trust regions est que l'expression de f nous est inconnue. Nous avons seulement à notre disposition un ensemble $P = \{p_i \,|\, i = 1, \ldots, l\}$. p_i est un point dans un espace à n dimensions et est associé à une valeur $f_i = f(p_i) \in \{0, 1\}$, $i = 1 \ldots l$. De plus, les f_i ne sont pas exempts d'erreurs. Ainsi, il y a une quantité (supposée mineure) de points p_i ne vérifiant pas $f_i = f(p_i)$. Dans notre application, nous avons $n = 200$ dimensions, car l'entreprise pense que $f(p_i)$ dépend de 200 paramètres.

f est supposée être une fonction très difficile à minimiser (non convexe et non linéaire). Même si nous pouvions la modéliser précisément, nous ne parviendrions pas à en trouver des minima satisfaisants. Ainsi, à chaque itéré x_k, nous allons plutôt définir un modèle m_k facile à minimiser. Ce modèle est construit de telle manière à approximer au mieux les points de données (et par extension, f) dans le voisinage de x_k, que nous appellerons sa région de confiance (trust region). La région de confiance autour de x_k est l'ensemble des points vérifiant

$$B_k = \{x \in \mathbb{R} \,|\, \|x - x_k\| \leq \Delta_k\}$$

Cette sphère est centrée en x_k et son rayon vaut Δ_k. Nous chercherons à obtenir un pas s_k tel que $x_{k+1} = x_k + s_k$ nous déplace vers un point où le modèle est plus réduit (c'est-à-dire $m_k(x_{k+1}) < m_k(x_k)$), tout en restant dans la région de confiance ($\|s_k\| \leq \Delta_k$). Si $m(x_{k+1})$ ne correspond pas à $f(x_{k+1})$, alors x_{k+1} est rejeté, le rayon de confiance est réduit et nous recalculons un autre modèle à partir de x_k. La région de confiance reflète la zone où l'on pense que le modèle approxime au mieux la fonction. Ces étapes sont reprises dans l'algorithme 2.1.

Algorithm 2.1 Algorithme trust region

repeat
  Définir un modèle m_k dans B_k
  Calculer s_k tel que $x_k + s_k$ minimise suffisamment le modèle m_k et que $x_k + s_k \in B_k$
  Calculer

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}$$

  qui reflète si $m_k(x_k + s_k)$ correspond à $f(x_k + s_k)$
  if $\rho_k \geq \eta_2$ **then**
    {Ce point est accepté et excellent ! On agrandit le rayon}
    $x_{k+1} \leftarrow x_k + s_k$
    Agrandir la région de confiance.
  else if $\rho_k \geq \eta_1$ **then**
    {Ce point est accepté mais peu satisfaisant. On réduit un peu le rayon}
    $x_{k+1} \leftarrow x_k + s_k$
    Un peu réduire la région de confiance.
  else
    {Ce point est rejeté. On réduit plus fortement le rayon}
    $x_{k+1} \leftarrow x_k$
    Réduire la région de confiance.
  end if
until convergence

2.1 Définition du modèle m_k

2.1.1 Rayon de recherche des points

Comment définir le modèle m_k ? Si nous connaissions f, nous utiliserions l'approximation de Taylor autour de x_k. Cependant, nous n'avons à notre disposition que l'ensemble P des points. Afin de construire un modèle à l'itération k, il semble évident qu'il est préférable de sélectionner des points dans le voisinage de x_k, à partir desquels nous appliquerons une régression afin de construire un modèle. Faut-il prendre tous les points $p_i \in B_k$? Ce n'est probablement pas la meilleure idée, car il serait profitable d'avoir également des points au-delà du rayon de la région de confiance, afin de nous renseigner sur le comportement de f légèrement en dehors de B_k. Devrions-nous prendre tous les points contenus dans une boule immense, bien plus grande que B_k ? Ce n'est sûrement pas une idée brillante car un modèle obtenu de la sorte n'aurait pas beaucoup de sens à l'intérieur de B_k, étant donné qu'il reflèterait f à une bien plus grande échelle. C'est donc un compromis qui est très difficile à évaluer.

Soit Q la sphère de recherche des points (*query*) centrée en x_k, et son rayon $r(Q)$ que nous avons défini. Nous devrons déterminer l'ensemble

$$P_k = \{p_i \,|\, \|p_i - x_k\| \leq r(Q); i = 1, \ldots, m; m \leq l\} \subseteq P$$

où P est l'ensemble des l points de données. Comme cette opération est effectuée à chaque itération de la méthode de région de confiance, elle doit être relativement rapide. Cette « recherche spatiale » est nettement moins simple qu'une recherche unidimensionnelle. Le chapitre 4 est entièrement consacré à cette tâche.

2.1.2 Complexité du modèle

Tout l'intérêt de la méthode par région de confiance est d'obtenir, à chaque itération k, un modèle m_k nettement plus facile à optimiser que f. Cependant, le modèle doit évidemment être suffisamment complexe que pour pouvoir refléter f dans la région de confiance B_k. Nous allons construire un modèle quadratique. C'est le modèle le plus complexe et qui reste facile à résoudre.

2.1.3 Régression unidimensionnelle

Nous supposerons désormais avoir obtenu l'ensemble $P_k = \{p_i \,|\, i = 1, \ldots, m\}$ des points contenus dans la sphère de recherche Q. Nous allons construire un modèle à partir de ces points en effectuant une régression.

Bien que nos points soient multidimensionnels, commençons le raisonnement à une seule dimension pour la clarté. La généralisation à plusieurs dimensions sera aisée. Mathématiquement, nous avons m points, où chaque point est dénoté par une paire $(p_i, f_i), i = 1 \ldots m$ et nous tentons de les représenter avec un polynôme $m(p)$ de degré d

$$m(p) = a_0 + a_1 p + a_2 p^2 + \cdots + a_n p^d$$

où $m \geq d + 1$. (Dans notre application, nous choisissons $d = 2$ afin d'obtenir un modèle quadratique.)

De nombreuses techniques d'estimation existent afin d'accomplir la régression. Naturellement, la fonction ne pourra pas passer par tous les points de données. Nous devrons donc définir un modèle qui minimise une certaine fonction de pénalité. Nous avons implémenté la méthode de régression des moindres carrés (least-squares). On dit que l'on effectue une régression au sens des moindres carrés. Elle consiste à minimiser la fonction de pénalité

$$L(a) = \sum_{i=1}^{m} \Big(a_0 + a_1 p_i + a_2 p_i^2 + \cdots + a_d p_i^d - f_i \Big)^2 \tag{2.1}$$

où les coefficients $a_0, a_1, \ldots, a_d$ sont inconnus. Une condition suffisante d'optimalité de $L(a)$ est l'annulation des dérivées partielles. Nous dérivons donc

$$\begin{cases} \frac{\partial}{\partial a_0} &= \sum_{i=1}^{m} 2\left(a_0 + a_1 p_i + a_2 p_i^2 + \cdots + a_d p_i^d - f_i\right) &= 0 \\ \frac{\partial}{\partial a_1} &= \sum_{i=1}^{m} 2\left(a_0 + a_1 p_i + a_2 p_i^2 + \cdots + a_d p_i^d - f_i\right) p_i &= 0 \\ & \vdots & \\ \frac{\partial}{\partial a_d} &= \sum_{i=1}^{m} 2\left(a_0 + a_1 p_i + a_2 p_i^2 + \cdots + a_d p_i^d - f_i\right) p_i^d &= 0 \end{cases} \tag{2.2}$$

que nous pouvons mettre sous forme matricielle

$$\begin{pmatrix} \sum_{i=1}^{m} p_i^{2d} & \sum_{i=1}^{m} p_i^{2d-1} & \cdots & \sum_{i=1}^{m} p_i^{d+1} & \sum_{i=1}^{m} p_i^{d} \\ \sum_{i=1}^{m} p_i^{2d-1} & \sum_{i=1}^{m} p_i^{2d-2} & \cdots & \sum_{i=1}^{m} p_i^{d} & \sum_{i=1}^{m} p_i^{d-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \sum_{i=1}^{m} p_i^{d+1} & \sum_{i=1}^{m} p_i^{d} & \cdots & \sum_{i=1}^{m} p_i^{2} & \sum_{i=1}^{m} p_i \\ \sum_{i=1}^{m} p_i^{d} & \sum_{i=1}^{m} p_i^{d-1} & \cdots & \sum_{i=1}^{m} p_i & d \end{pmatrix} \begin{pmatrix} a_d \\ a_{d-1} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} p_i^d f_i \\ \sum_{i=1}^{m} p_i^{d-1} f_i \\ \vdots \\ \sum_{i=1}^{m} p_i f_i \\ \sum_{i=1}^{m} f_i \end{pmatrix} \tag{2.3}$$

que l'on appelle les « équations normales ». L'abondance de symboles de sommation rend la construction peu pratique. [KKI03] conseille d'utiliser la matrice de Vandermonde

$$V = \begin{pmatrix} 1 & p_1 & p_1^2 & \cdots & p_1^d \\ 1 & p_2 & p_2^2 & \cdots & p_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & p_m & p_m^2 & \cdots & p_m^d \end{pmatrix}$$

qui nous permet de réécrire les équations normales (équations 2.3) sous la forme compacte

$$V^T V a = V^T f \tag{2.4}$$

2.1.4 Régression multidimensionnelle

En régression multidimensionnelle, nous devons évidemment considérer toutes les dimensions, mais également les corrélations entre celles-ci. Heureusement, les modifications sont aisées à réaliser. Afin de rendre les explications claires, nous considérons une régression quadratique tridimensionnelle au sens des moindres carrés. Dans cette section, p dénote la première dimension, q la deuxième et r la troisième.

$$m(p,q,r) = a_0 + a_1 p_i + a_2 q_i + a_3 r_i + a_4 p_i^2 + a_5 q_i^2 + a_6 r_i^2 + a_7 p_i q_i + a_8 p_i r_i + a_9 q_i r_i$$

De même, il suffit de créer de nouvelles colonnes dans la matrice de Vandermonde :

$$V = \begin{pmatrix} 1 & p_1 & q_1 & r_1 & p_1^2 & q_1^2 & r_1^2 & p_1 q_1 & p_1 r_1 & q_1 r_1 \\ 1 & p_2 & q_2 & r_2 & p_2^2 & q_2^2 & r_2^2 & p_2 q_2 & p_2 r_2 & q_2 r_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & p_m & q_m & r_m & p_m^2 & q_m^2 & r_m^2 & p_m q_m & p_m r_m & q_m r_m \end{pmatrix}$$

et de résoudre les équations normales (équations 2.4) avec la nouvelle expression de V.

2.1.5 Résolution des équations normales

Soit la résolution des équations normales. Nous devons tout simplement résoudre un système d'équations linéaires $Ap = b$. Il est à noter que si les polynômes d'interpolation sont orthogonaux, alors A est diagonale et la résolution du système devient triviale. Nous avons choisi les polynômes $\{p^0, p^1, \ldots, p^d\}$ qui ne sont pas orthogonaux. Cependant, il aurait été possible d'utiliser les célèbres polynômes de Chebyshev qui sont, eux, orthogonaux. Malheureusement, nous n'avons pas pu les généraliser au cas multidimensionnel.

Afin de résoudre le problème linéaire $Ap = b$, nous avons mis en œuvre plusieurs solveurs :

- réimplémentation de l'élimination gaussienne (c'est une décomposition LU) ;
- décomposition LU de Lapack [1] ;
- élimination gaussienne avec choix optimal du pivot (pour éviter un pivot proche de 0) de OpenCV [2] ;
- décomposition SVD de OpenCV.

2.2 Quantité de dimensions dans le problème

Grâce à la section précédente, nous sommes capables d'effectuer une régression à n'importe quelle dimension n et degré d. Nous avons déjà choisi $d = 2$ afin d'avoir un modèle quadratique, suffisamment complexe et facile à résoudre. Devons-nous conserver $n = 200$, donc toutes les variables disponibles ? À hautes dimensions, la régression et la minimisation du modèle ne sont pas spécialement plus difficiles. En revanche, nous verrons dans le chapitre 4 que la recherche de points est plus difficile à hautes dimensions, mais ce n'est pas insurmontable.

Malheureusement, les hautes dimensions posent un autre problème de taille, faisant de cette Section la plus importante du mémoire.

Observation 1. Dispersion des données — hypercube
Soit le domaine $\Omega = [0, 1]^n$. Considérons un hypercube de recherche de longueur l dans chaque dimension. La probabilité qu'un point (uniformément distribué) soit dans l'hypercube de recherche vaut

$$P_n(l) = l^n$$

Pour $n = 200$ et $l = 0.95$ (donc on prend 95% dans chaque dimension), la probabilité d'y trouver un certain point est de 0.000035, virtuellement nulle. Il est donc très difficile de trouver des points dans Ω.

1. LAPACK (Linear Algebra PACKage) est une bibliothèque logicielle dédiée à la simulation numérique écrite en Fortran 77.
2. OpenCV (Open Computer Vision) est une bibliothèque graphique spécialisée dans le traitement d'images en temps réel.

Observation 2. Dispersion des données — hypersphère
Soit le domaine Ω. Considérons la plus grande hypersphère `sphere(c, 0.5)` qui est complètement contenue dans Ω. c est le centroïde de l'espace et le rayon vaut 0.5. Intuitivement, nous la voyons comme assez grande par rapport à l'espace, et pensons qu'elle doit contenir beaucoup de points. C'est vrai à basses dimensions, mais cela devient complètement faux à dimensions moyennes.
[WSB98] formule le « volume » multidimensionnel de `sphere(c, 0.5)` et calcule la probabilité qu'un point se trouve dedans :

$$P[\text{point} \in \text{sphere}(c, 0.5)] = \frac{\sqrt{\pi^d}\left(\frac{1}{2}\right)^d}{\left(\frac{d}{2}\right)!}$$

Le volume relatif de la sphère par rapport à l'espace diminue drastiquement avec le nombre de dimensions, et il devient très improbable qu'un seul point soit dans cette sphère (tableau 2.1).

Observation 3. Taille de la base de données
[WSB98] calcule également la quantité de points nécessaires dans l'espace pour qu'en moyenne, au moins un point soit dans la sphère :

$$N(d) = \frac{\left(\frac{d}{2}\right)!}{\sqrt{\pi^d}\left(\frac{1}{2}\right)^d}$$

d	$P[\text{point} \in \text{sphere}(c, 0.5)]$	$N(d)$	taille de la BDD
2	0.785	2	32B
4	0.308	4	128B
10	0.0025	402	32.16KB
20	$2.46\ 10^{-8}$	$4\ 10^{7}$	6.4GB
40	$3.27\ 10^{-21}$	$3.05\ 10^{20}$	9.76^{10}TB
200	$3.46\ 10^{-169}$	$2.89\ 10^{168}$	$4.62\ 10^{159}$TB

TABLE 2.1 – Probabilité qu'un point soit dans sphere$(c, 0.5)$, quantité de points nécessaires et taille attendue de la BDD

Le tableau 2.1 reprend ces résultats. Bien qu'interpellant, il est même extrêmement optimiste car, dans notre application,
- le rayon des régions de confiance sera bien plus réduit que 0.5 ;
- pour effectuer une régression, il ne faut pas un seul point, mais une bonne dizaine d'entre eux.

Nous estimons que même avec 5 millions de points et en imposant de relativement larges sphères de recherche, il est indispensable de réduire le problème à environ 6 dimensions afin d'espérer trouver suffisamment de points pour pouvoir effectuer des régressions.

C'est certainement le résultat le plus important de ce mémoire. Il faut réaliser que ce n'est pas un problème de performances ; c'est un problème de manque d'information. C'est assez contre-intuitif car avec 5 millions de points, nous pourrions penser en avoir bien assez. Cependant, à dimensions moyennes, c'est largement insuffisant. Ce problème de dimensionnalité est tel qu'il porte un nom dans la littérature scientifique : *curse of dimensionality*. On peut considérer que les fonctions de distances perdent leur signification à hautes dimensions.

La dispersion des données met gravement en péril l'applicabilité même de l'optimisation par région de confiance, obligeant l'utilisateur à réduire très drastiquement le nombre de dimensions afin de pouvoir espérer un résultat. Nous avons implémenté le PCA (Principal Component Analysis), qui est une méthode de réduction de dimensionnalité. (En plus de la compression, nous pouvons également l'utiliser pour décorréler les données.) Cependant, peut-on vraiment espérer passer de 200 dimensions à 6 en conservant suffisamment d'information ? Même si nous résolvons le problème réduit à 6 dimensions, la solution optimale aura-t-elle suffisamment de signification dans le problème d'origine ? C'est peu probable. De plus, l'entreprise de galvanisation a une grande expertise et sait à quel point de très nombreux paramètres influencent le résultat de la galvanisation. Il faut ajouter à cela que le problème à 6 dimensions n'est pas pour autant simple à résoudre ; c'est un problème non convexe et non linéaire, et nous n'avons pas d'assurance de trouver un minimum global.

Seule bonne nouvelle, nous pouvons supposer que les points ne sont pas uniformément distribués. En effet, il est probable que l'entreprise a tout de même, par essais/erreurs, focalisé ses galvanisations dans des zones propices. Nous pouvons donc penser que certaines zones sont nettement plus peuplées en points que d'autres, et une optimisation par région de confiance devrait encore pouvoir y être appliquée avec succès.

2.3 Minimisation du modèle

Désormais, nous avons l'expression d'un modèle m_k quadratique multidimensionnel, que nous pouvons réécrire sous la forme

$$m_k(s) = c_k + \langle g_k, s\rangle + \frac{1}{2}\langle s, H_k s\rangle$$

Nous avons besoin de le minimiser dans la région de confiance, c'est-à-dire de résoudre

$$\begin{aligned} \underline{\min} :&\ \langle g_k, s\rangle + \tfrac{1}{2}\langle s, H_k s\rangle \\ \underline{\text{s.t.}} :&\ \|s\| \leq \Delta_k \end{aligned}$$

Ce problème porte le nom de « sous-problème de trust region », car il doit être résolu à chaque itération de l'algorithme de région de confiance. C'est un problème d'optimisation quadratique avec une contrainte quadratique. Cependant, il peut être non convexe ! Dans un premier temps, nous avons utilisé un solveur commercial. Dans un second temps, grâce à [CGT00], nous avons considéré les spécificités de ce problème d'optimisation et avons implémenté 3 solveurs exacts et un solveur approximatif (chapitre 3).

2.4 Fiabilité du modèle

Le problème quadratique est résolu. Un minimum $x_k + s_k$ nous est proposé. Cependant, il est préférable d'être prudent. En effet, le modèle m_k a été conçu autour de x_k, pas autour de $x_k + s_k$. Il est donc possible que m_k approxime mal f au point $x_k + s_k$. [CGT00] nous propose de calculer un coefficient nous indiquant la fiabilité de $x_k + s_k$:

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}$$

qui reflète si $m_k(x_k + s_k)$ correspond à $f(x_k + s_k)$, afin de savoir si nous devrions accepter $x_k + s_k$ ou non. La région de confiance traduit le fait que le modèle m_k approxime plus ou moins bien f au point $x_k + s_k$. Si ce n'est pas le cas, la région de confiance sera réduite dans l'espoir de concevoir un nouveau modèle qui convient mieux. Naturellement, une difficulté supplémentaire vient du fait que f est inconnu dans notre application, empêchant le calcul de ρ_k. Étonnamment, c'est une particularité qui n'est pas encore étudiée dans la littérature scientifique : aucun document ne fait mention de l'algorithme de région de confiance avec f inconnu !

Nous avons implémenté une petite alternative pour pallier la méconnaissance de f. Étant donné que nous avons accès à une régression au sens des moindres carrés, nous allons effectuer une régression $m^*(s)$ autour de $x_k + s_k$ avec un rayon $r(Q^*)$ faible ainsi qu'une régression $m^+(s)$ autour de x_k avec un rayon $r(Q^+)$ faible également. L'indice de confiance devient

$$\overline{\rho_k} = \frac{m^+(x_k) - m^*(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}$$

Il est difficile de quantifier la qualité de cette heuristique, mais nos résultats sont a priori plutôt concluants. Il aurait été intéressant d'essayer les extra trees en mode régression. Malheureusement, il n'y a pas d'implémentation C++, et le temps nous a manqué pour pouvoir le refaire nous-même.

2.5 Minimum local

À ce stade, l'algorithme de région de confiance sait déjà tourner de lui-même : il calcule un modèle, le minimise, en valide le minimum, se déplace vers ce minimum et ainsi de suite... Il est temps de se demander quand l'algorithme va s'arrêter. Le critère de

convergence consiste en la découverte d'un point critique, c'est-à-dire via l'annulation du gradient. Ne possédant évidemment pas le gradient, nous avons utilisé un critère simple qui sera vérifié pour un minimum local : la faible distance entre deux itérés consécutifs. Ce critère, bien que trivial, s'est révélé plutôt satisfaisant.

2.6 Obtenir des minima intéressants

[CGT00] démontre que l'algorithme converge vers un point critique, qui n'est peut-être même pas un minimum local. Nous devrions donc obtenir un point critique, et l'algorithme s'arrêtera. Or, dans notre application, ce n'est pas du tout ce que nous recherchons. Nous avons plutôt besoin de trouver plusieurs minima (éventuellement locaux) « intéressants ». Remarquons que le but n'est pas de trouver le minimum global dans le sens où il faut un point qui minimise le plus f. Ici, f est binaire ; il y a donc un très grand nombre de 0 et de 1. Notre notion de minimum global s'oriente plutôt vers un minimum situé dans un bassin large, afin que si l'on modifie un peu les variables, la galvanisation restera bonne. Nous aimerions donc pouvoir quantifier la grandeur du bassin d'un minimum. Nous avons imaginé une adaptation de l'algorithme de région de confiance afin de permettre la navigation à travers plusieurs minima et l'estimation de leur qualité.

2.6.1 Saut pour sortir d'un minimum

Nous trouvons le $j^{\text{ième}}$ minimum local $min_j = x_k$ et nous calculons le vecteur $\vec{v} = min_j - min_{j-1}$, où min_{j-1} est le précédent minimum local connu. Cf Figure 2.1, nous calculons le nouvel itéré

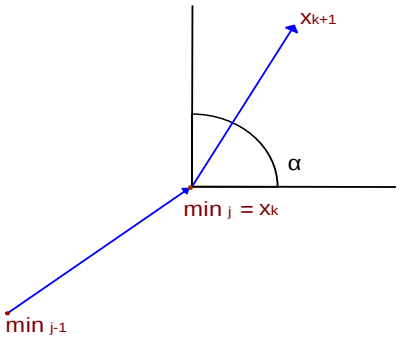


FIGURE 2.1 – Saut de longueur $\|w\|$ avec angle aléatoire

$$x_{k+1} \leftarrow \min_j + wR(\alpha)\frac{\vec{v}}{\|\vec{v}\|}$$

où $R(\alpha)$ est une matrice de rotation et w est le score. La rotation permet de ne pas aller uniquement en ligne droite, mais également vers la gauche ou la droite (avec plus

de dimensions, il faudra des angles supplémentaires). L'angle est choisi aléatoirement. w est la longueur du saut. Deux cas peuvent se présenter :

1. Quelques itérés plus tard, on retrouve un minimum local très proche de min_j. Cela signifie que le saut n'a pas permis de quitter le « bassin », et donc on est revenu vers min_j. Cela signifie que c'est un bon minimum.
2. L'algorithme continue, et le prochain minimum local min_{j+1} n'est pas proche de min_j. Cela signifie que le saut a permis de quitter min_j, donc c'était un minimum local faible. Bien sûr, il se pourrait que les itérés décrivent une boucle dans l'espace et reviennent plus tard vers min_j, mais des minima locaux intermédiaires auront été découverts, et nous saurons qu'il s'agit d'un cycle.

La première question qui se pose est : comment fixer w ? En lui attribuant une valeur, nous pourrons détecter des minima meilleurs que w (c'est-à-dire tels qu'un saut de longueur w ne permet pas de quitter le bassin des minima). Cependant, il est difficile de donner une valeur à w car nous n'avons aucune idée de la qualité espérée des minima. Dans la section suivante, nous allons adapter la méthode pour éviter cette difficulté.

2.6.2 Saut adaptable et scores des minima

Nous nous basons sur la notion de saut afin de développer une méthode avec une longueur de saut adaptée automatiquement, ainsi qu'une mesure de la qualité d'un minimum.

Commençons avec w faible. Si nous rencontrons un minimum, nous l'enregistrons au sommet d'une pile, avec w qui représente le score actuel du minimum. Nous effectuons un saut afin de tenter de sortir du minimum. Si l'itéré revient, alors w est augmenté et est mis à jour, et ainsi de suite jusqu'à ce que l'on quitte définitivement le minimum. Au sommet de la pile, le minimum est accompagné du w qui a été nécessaire pour échapper au bassin du minimum.

Ensuite, il se pourrait que nous rencontrions des minima dont nous sortons du premier coup grâce au saut ; cela veut dire que ces minima étaient inintéressants et nous les oublions. Si nous rencontrons un minimum avec un bassin plus large que ce que nous avons déjà rencontré, alors un saut de longueur w ne devrait pas suffire à en sortir. Quelques itérés plus tard, nous reviendrons vers le minimum et nous pourrons l'enregistrer sur la pile, accompagné de son score.

Cette méthode va permettre d'enregistrer plusieurs minima intéressants et d'avoir une bonne idée de leur qualité. Naturellement, cette technique ne visite pas tous les minima de la fonction, mais seulement ceux qu'elle traverse et qui sont meilleurs que les minima déjà connus.

2.6.3 Exemple complet

La Figure 2.2 présente un problème à deux dimensions, non convexe et non linéaire. (Afin que l'exemple soit plus parlant, nous ne l'avons pas seulement lancé avec des 0 et des 1 car c'est nettement moins parlant.) Il y a un grand minimum, un minimum moyen

et deux petits. Nous aimerions tomber dans un des deux grands, idéalement dans le plus grand. Commençons l'algorithme au point $(5, 5)$, c'est à dire près du plus petit minimum. Sans le saut, l'algorithme termine dans le plus petit minimum et s'arrête. Notre méthode (avec estimation de f et de g) est nettement mieux et nous fournit un tableau de scores (tableau 2.2). Sur la Figure 2.3, les points bleus ont été reconnus comme minima locaux. Nous pouvons distinctement observer les tentatives de l'algorithme pour échapper aux minima.

Minimum	Score
$(62.97; 62.99)$	36.97
$(39.34; 39.34)$	12.38
$(21.79; 21.78)$	7.16
$(7.33; 7.32)$	4.97

TABLE 2.2 – Minima et scores respectifs

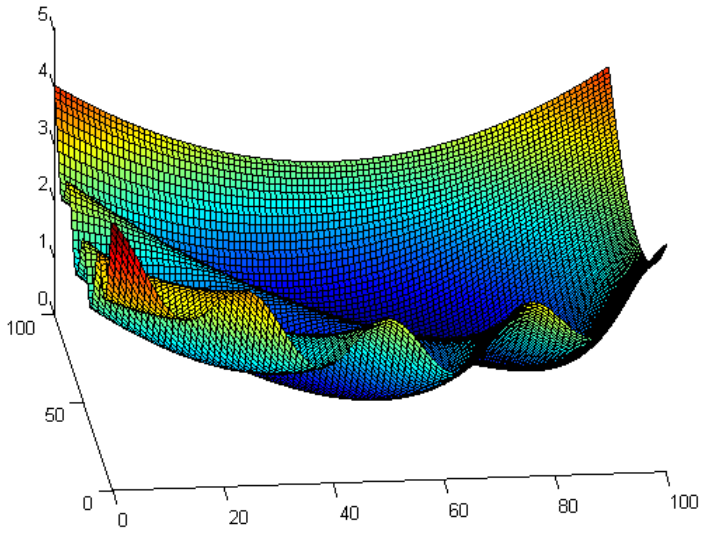


FIGURE 2.2 – Fonction non convexe non linéaire à minimiser

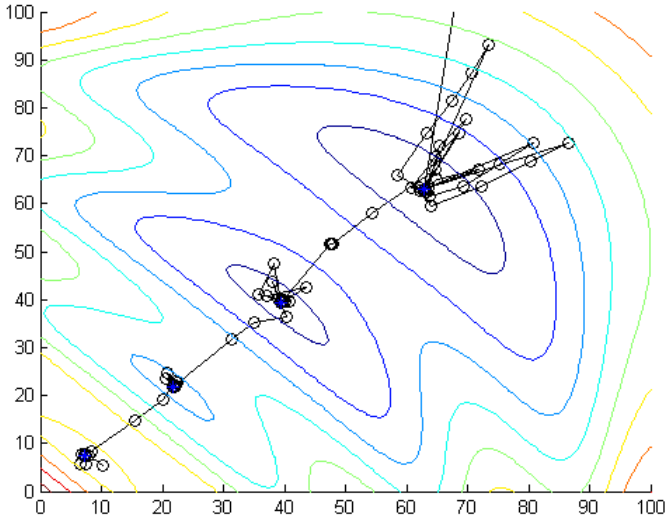


FIGURE 2.3 – Séquence d'itérés en commençant en (5,5) (en bas à gauche)

Chapitre 3

Sous-problème de région de confiance

Le sous-problème de la région de confiance consiste en la minimisation du modèle m_k dans la région de confiance B_k, lors de l'itération k. C'est une opération effectuée à chaque itération de l'algorithme de région de confiance.

3.1 Caractérisation du sous-problème

Nous résoudrons le problème d'optimisation

$$\begin{aligned} \underline{\min} : \ & q(s) = \langle g_k, s\rangle + \tfrac{1}{2}\langle s, H_k s\rangle \\ \underline{\text{s.t.}} : \ & \|s\| \leq \Delta_k \end{aligned}$$

Nous recherchons la solution s^M qui nous donnera le nouveau point $x_{k+1} = x_k + s^M$. Par facilité, dans cette section, nous allons abandonner les indices k qui dénotent le numéro de l'itération.

Nous pouvons déjà caractériser cette solution. Tout d'abord, nous imposons que $q(s)$ soit quadratique. Cependant, il peut être convexe ou non convexe. Nous distinguons trois cas :

- $q(s)$ est convexe et le minimum est à l'intérieur de la région de confiance (le problème est non contraint), donc $\|s^M\| < \Delta$;
- $q(s)$ est convexe et le minimum est à l'extérieur de la région de confiance, donc nous prendrons $\|s^M\| = \Delta$;
- $q(s)$ est non convexe ; dès lors nous opterons pour $\|s^M\| = \Delta$ car c'est le minimum dans la région de confiance (sans région de confiance, la solution aurait été non bornée).

Si H est défini positif, alors $q(s)$ est convexe.

Afin de caractériser plus précisément l'optimum, utilisons le multiplicateur de Lagrange en réécrivant tout d'abord la contrainte par $\frac{1}{2}\|s\|_2^2 \leq \frac{1}{2}\Delta^2$. Nous obtenons

$$q(s,\lambda) = \langle g, s\rangle + \frac{1}{2}\langle s, Hs\rangle - \frac{1}{2}\lambda\left(\Delta^2 - \|s\|_2^2\right)$$

Les contraintes d'optimalité du premier ordre fournissent

$$\frac{\partial q(s^M)}{\partial s}) = g + Hs^M + \lambda^M I s^M = g + H(\lambda^M)s^M = 0 \tag{3.1}$$

$$\lambda^M\left(\Delta - \|s\|_2\right) = 0 \tag{3.2}$$

avec $H(\lambda^M) \triangleq H + \lambda^M I$.

Reformulons l'équation 3.1 :

Lemme 1. *Le minimum de $q(s)$ tel que $\|s\|_2 \leq \Delta$ satisfait*

$$H(\lambda^M)s^M = -g \tag{3.3}$$

3.2 Résolution exacte

3.2.1 *Modus operandi*

Cherchons maintenant la solution à 3.3. Nous pourrions substituer $s(\lambda) = -H(\lambda)^{-1}g$ dans $\lambda^M\left(\Delta - \|s\|_2\right) = 0$ afin d'avoir une équation non linéaire et une seule variable λ. Cependant, comme nous allons le découvrir, la convergence est très mauvaise. Heureusement, nous allons pouvoir manipuler les équations afin d'obtenir ce que nous appelerons l' « équation séculaire », et appliquerons Newton sur cette dernière. De plus, nous utiliserons des bornes inférieure et supérieure, qui seront resserrées progressivement afin d'assurer une convergence *worst-case*. Nous devrons également considérer le cas où H n'est pas défini positif.

3.2.2 Comportement de la région de confiance selon λ

3.2.2.1 Cas convexe

H étant symétrique, sa décomposition SVD se simplifie en

$$H = U^T\Lambda U \quad \text{et} \quad H(\lambda) = U^T(\Lambda + \lambda I)U$$

où $\Lambda = \text{diag}\left(\lambda_1, \ldots, \lambda_n\right)$ est la matrice diagonale des valeurs propres, avec $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Les colonnes de U sont les vecteurs propres correspondants. Utilisons cette décomposition dans l'équation 3.3 :

$$s(\lambda) = -H(\lambda)^{-1}g = -U^T(\Lambda + \lambda I)^{-1}Ug$$

Nous allons étudier l'influence de λ sur $\psi(\lambda) \triangleq \|s(\lambda)\|_2^2$ lorsque les conditions optimales sont remplies (c'est-à-dire si nous avons $s(\lambda) = -H(\lambda)^{-1}g$). Nous pouvons développer

$$\psi(\lambda) = \|s(\lambda)\|_2^2 = \|U^T(\Lambda + \lambda I)^{-1}Ug\|_2^2 = \|(\Lambda + \lambda I)^{-1}Ug\|_2^2 = \sum_{i=1}^{n} \frac{\gamma_i^2}{(\lambda_i + \lambda)^2}$$

où γ_i est la $i^{\text{ème}}$ composante de Ug. Afin de représenter cette fonction, aidons-nous des limites :

$$\begin{aligned} \lim_{\lambda \to \pm\infty} \psi(\lambda) &\sim \tfrac{1}{\lambda^2} \to 0 \\ \lim_{\lambda = -\lambda_i} \psi(\lambda) &\sim \tfrac{\gamma_i^2}{0} \to \infty \end{aligned}$$

Ces limites sont riches d'enseignement. Soit H dont les valeurs propres sont $\{1, 2, 3, 4\}$. Les pôles $-\lambda_i$ valent $\{-1, -2, -3, -4\}$ et sont visibles sur la Figure 3.1 car $\psi(\lambda)$ tend alors vers l'infini.

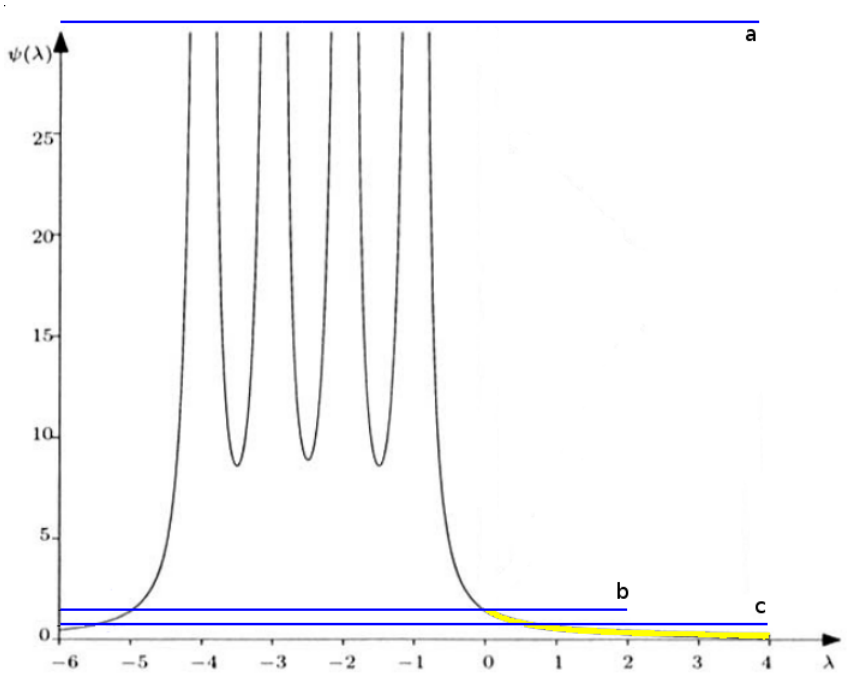


FIGURE 3.1 – Pôles de ψ en fonction de λ (image [CGT00])

Rappelons que tous les points sur la courbe $\psi(\lambda)$ respectent l'optimalité $H(\lambda)s(\lambda) = -g$. Cependant, afin qu'un minimum existe, les conditions 3.4 doivent être respectées :

$$\begin{cases} \lambda \geq 0 & \text{(condition de Lagrange)} \\ \lambda > -\lambda_1 & (H(\lambda) \text{ défini positif car } q(s) \text{ convexe}) \end{cases} \tag{3.4}$$

Même si nous avons développé $\psi(\lambda) \triangleq \|s(\lambda)\|_2^2$, nous n'avons pas encore imposé $\psi(\lambda) \leq \Delta^2$. Sur la Figure 3.1, nous avons tracé trois lignes horizontales : $\Delta^2 = a$, $\Delta^2 = b$ et $\Delta^2 = c$. Rappelons que nous effectuons une minimisation à l'intérieur d'une région de confiance et que nous devons donc toujours respecter $\psi(\lambda) \leq \Delta^2$. Analysons nos trois valeurs de Δ^2 :

- $\Delta^2 = a$, une valeur très grande telle que $\psi(\lambda) < a, \forall\lambda$. On peut donc prendre un λ respectant les conditions 3.4 page 26, et tel que $\psi(\lambda)$ et Δ^2 s'intersectent sur la Figure 3.1. Nous avons une solution intérieure.
- Descendons Δ^2 jusqu'à atteindre $\Delta^2 = b$ qui intersecte $\psi(0)$. En $\lambda = 0$, l'optimum vérifie $\psi(\lambda) = \|s(\lambda)\|_2^2 = \Delta^2$, donc la solution est sur la frontière de la région de confiance.
- $\Delta^2 = c$ intersecte $\psi(\lambda)$ si $\lambda > 0$. Comme il faut toujours $\psi(\lambda) \leq \Delta^2$, nous devons prendre un certain $\lambda > 0$ tel que $\psi(\lambda) \leq c$. L'optimum consiste à prendre λ^M tel que $\psi(\lambda^M) = c$, donc l'optimum est sur la frontière de la région de confiance.
- Si $\Delta^2 > b$, l'intersection avec $\psi(\lambda)$ a lieu pour $\lambda < 0$. Cependant, cela ne respecte pas les conditions 3.4 page 26 (λ doit être positif). Nous devons forcer $\lambda = 0$, entraînant $\psi(0) = b < \Delta^2$, donc le minimum est un point à l'intérieur de la région de confiance.

L'ensemble des solutions est représenté en jaune sur la Figure 3.1.

3.2.2.2 Cas non convexe

Si $q(s)$ est non convexe, nous avons déjà expliqué que le minimum se trouve sur la frontière de la région de confiance. La théorie nous permet de le confirmer. Soit H dont les valeurs propres sont $\{-2, -1, 0, 1\}$ (H est indéfini). Les pôles valent donc $\{2, 1, 0, -1\}$, et les conditions 3.4 page 26 impliquent

$$\begin{aligned} \lambda &> -\lambda_1 \\ \lambda &> 2 \end{aligned}$$

Cf. Figure 3.2, nous prenons uniquement $\lambda > 2$. Dans ce cas, quelle que soit la valeur de Δ^2 (nous pouvons tracer une ligne horizontale à n'importe quelle hauteur sur le schéma), l'arc $\psi(\lambda)$ en jaune sera toujours intersecté en un seul point qui est tel que $\psi(\lambda) = \|s(\lambda)\|_2^2 = \Delta^2$. Donc le minimiseur est toujours sur la frontière de la région de confiance.

3.2.3 Calculer la racine de $\|s(\lambda)\|_2 - \Delta = 0$

Nous nous sommes intéressés sur les caractéristiques du problème quadratique et avons étudié les propriétés de la solution. Désormais, nous allons tenter de résoudre numériquement.

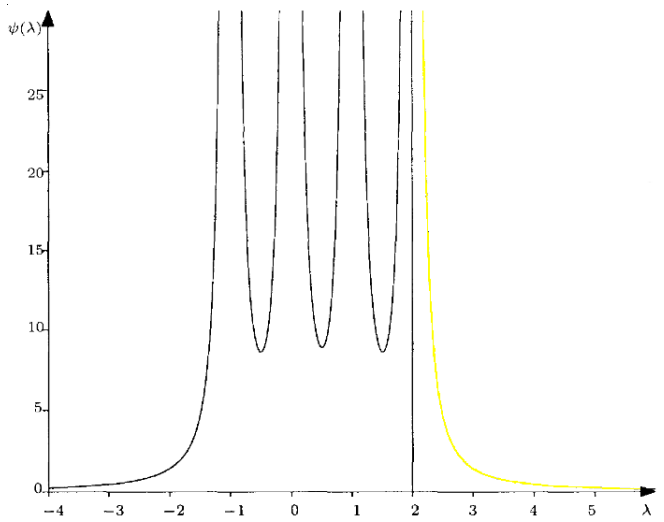


FIGURE 3.2 – Pôles de ψ en fonction de λ, cas non convexe (image [CGT00])

L'approche la plus classique est d'utiliser Newton. Cependant, comme nous l'évoquions en début de chapitre, la convergence est très mauvaise. En effet, elle est intimement liée au fait d'avoir une bonne approximation de la dérivée dans la région d'intérêt. Comme nous l'avons vu sur les figures précédentes, dans le cas convexe, il y a un pôle en $-\lambda_1$ près de l'optimum que nous recherchons, et la dérivée sera donc très instable dans cette région, d'où une convergence très mauvaise.

Heureusement, nous pouvons remanier le problème grâce à l'analyse acquise dans la section précédente. $\psi(\lambda)$ a des pôles mais pas de zéro (fini), ce qui entraîne que $\frac{1}{\psi(\lambda)}$ a des zéros mais pas de pôle (fini). Nous résoudrons donc l' « équation séculaire »

$$\phi(\lambda) \stackrel{\Delta}{=} \frac{1}{\|s(\lambda)\|_2} - \frac{1}{\Delta} = 0 \tag{3.5}$$

avec la méthode de Newton. Ensuite, [CGT00] développe les dérivées et la mise à jour de Newton pour l'équation 3.5. Appliquée à l'équation séculaire, la méthode de Newton est exposée dans l'algorithme 3.1.

Newton fait intervenir la factorisation de Cholesky, que nous avons réimplémentée. D'ailleurs, elle nous est également utile pour une autre raison. En effet, nous imposons $\lambda > -\lambda_1$ afin que $H(\lambda)$ soit défini positif. Cependant, le seul moyen de connaître λ_1 est de calculer les valeurs propres de H, ce qui est très coûteux. Simplement, la factorisation de Cholesky échouera si H n'est pas défini positif et nous saurons si le problème est convexe ou non.

Algorithm 3.1 Une itération de Newton pour $\phi(\lambda) = 0$

Require:
$\lambda > -\lambda_1$
$\Delta > 0$
Ensure:
Mise à jour de λ

1: Factoriser $H(\lambda)$ en $H(\lambda) = LL^T$ (factorisation de Cholesky)
2: Résoudre $LL^T s = -g$ en résolvant d'abord $Lt = -g$ puis $L^T s = t$
3: Résoudre $Lw = s$
4: $\lambda \leftarrow \lambda + \left(\frac{\|s\|_2 - \Delta}{\Delta}\right)\left(\frac{\|s\|_2^2}{\|w\|_2^2}\right)$
5: **return** λ

3.2.4 Intervalle d'incertitude

Il est bien connu que la méthode de Newton peut faire des cycles ou diverger. Afin d'assurer la convergence, nous allons utiliser les propriétés du problème afin d'établir des bornes qui seront resserrées durant l'algorithme, afin de prendre au piège la solution. Afin d'y arriver, nous allons d'abord devoir diviser le domaine de λ en zones.

3.2.4.1 Zones significatives

La Figure 3.3 présente la division du domaine de λ en trois zones. :
– $\mathcal{N}$ (*not feasible*) si λ ne respecte pas les conditions 3.4 page 26 (dans le cas contraire, $\lambda \in \mathcal{F}$ (*feasible*)) ;
– $\mathcal{L}$ (*lower than the model minimizer*) si $\lambda \in \mathcal{F}$ et $\lambda \leq \lambda^M$;
– $\mathcal{G}$ (*greater than the model minimizer*) si $\lambda \in \mathcal{F}$ et $\lambda > \lambda^M$.

Nous ne connaissons ni λ_1 ni λ^M (λ^M est l'optimum que nous recherchons). Dès lors, comment savoir dans quelle zone se trouve un certain λ ? Il est simple de déterminer si $\lambda \in \mathcal{N}$ ou $\lambda \in \mathcal{F}$ (*feasible*) en regardant si la décomposition de Cholesky de $H(\lambda)$ a échoué ou non (algorithme 3.1). Ensuite, si $\lambda \in \mathcal{F}$, alors $\mathcal{L}$ et $\mathcal{G}$ sont différenciés par le fait que $\phi(\lambda) \leq 0$ si $\lambda \in \mathcal{L}$, et $\phi(\lambda) > 0$ si $\lambda \in \mathcal{G}$. Mathématiquement, si $\phi(\lambda) > 0$, nous développons

$$\begin{aligned} \phi(\lambda) &> 0 \\ \frac{1}{\|s(\lambda)\|_2} - \frac{1}{\Delta} &> 0 \\ \frac{1}{\|s(\lambda)\|_2} &> \frac{1}{\Delta} \\ \|s(\lambda)\|_2 &< \Delta \end{aligned}$$

Donc, si $\lambda \in \mathcal{F}$,

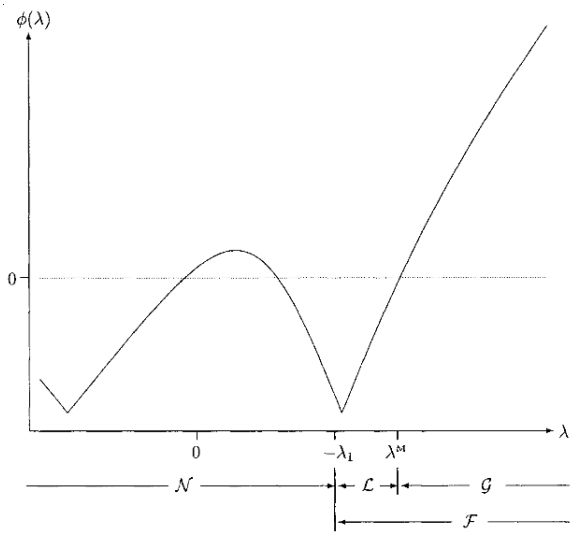


FIGURE 3.3 – Division du domaine en 3 zones (image [CGT00])

$$\begin{cases} \lambda \in \mathcal{G} & \text{si} \quad \|s(\lambda)\|_2 < \Delta \\ \lambda \in \mathcal{L} & \text{si} \quad \|s(\lambda)\|_2 \geq \Delta \end{cases}$$

[CGT00] démontre ce que nous rassemblons dans les lemmes suivants :

Lemme 2. *Si $\lambda \in \mathcal{L}$, alors l'itéré suivant $\lambda^+ \in \mathcal{L}$ (la mise à jour de λ est obtenue avec une itération de Newton). C'est extrêmement intéressant pour la convergence !*

Lemme 3. *Si $\lambda \in \mathcal{G}$, alors l'itéré suivant $\lambda^+ \in \mathcal{L}$ ou $\mathcal{N}$.*

Remarque 1. Si le minimiseur λ^M est à l'intérieur de la région de confiance, alors la région $\mathcal{L}$ contient uniquement $\lambda = 0$. Dans ce cas, le problème est non contraint et l'optimum λ^M vaut 0.

Remarque 2. Si l'on quitte $\mathcal{G}$, on n'y revient plus.

Remarque 3. Si l'on arrive en $\mathcal{N}$, on n'a plus de moyen d'en sortir vu que la factorisation de $H(\lambda)$ a échoué et que l'on n'a plus de quoi continuer l'algorithme.

3.2.4.2 Bornes de l'intervalle d'incertitude

Si l'on arrive dans $\mathcal{N}$, la factorisation de Cholesky échoue et nous sommes bloqués. Comment nous en sortir ? Nous sommes obligés de choisir un $\lambda \in \mathcal{F}$ afin de relancer l'algorithme. C'est maintenant que nous allons introduire les bornes. Tout au long de l'algorithme, nous maintenons un intervalle $[\lambda^L; \lambda^U]$ (« L » pour *lower* et « U » pour

upper) dans lequel nous choisirons le nouveau λ si l'on tombe dans $\mathcal{N}$. Comment mettre à jour les bornes ? Si l'on trouve un $\lambda > \lambda^L$, nous mettrons simplement à jour $\lambda^L \leftarrow \lambda$, et similairement pour λ^U.

3.2.4.3 Améliorer la borne inférieure

Les bornes vont nous permettre de relancer l'algorithme si nous tombons dans la région $\mathcal{N}$. Dans cette section, nous faisons une petite parenthèse afin d'apporter une optimisation. De plus, cette optimisation va également nous être très utile plus tard pour l'ensemble de l'algorithme.

Tentons d'obtenir une meilleure borne inférieure λ^B (« B » pour *better*). Nous mettrons ensuite à jour $\lambda^L \leftarrow \max\{\lambda^L, \lambda^B\}$. Le principe repose sur le quotient de Rayleigh :

$$\underbrace{\lambda_{\min}[H(\lambda)]}_{\lambda_1+\lambda} \leq \frac{\langle u, H(\lambda)u\rangle}{\langle u, u\rangle} \leq \underbrace{\lambda_{\max}[H(\lambda)]}_{\lambda_n+\lambda}$$

avec $u \neq 0$. Reprenons $H(\lambda) = H + \lambda I$, dont les valeurs propres sont $\lambda_1 + \lambda \leq \lambda_2 + \lambda \leq \cdots \leq \lambda_n + \lambda$. $\frac{\langle u,Hu\rangle}{\langle u,u\rangle}$ est plus grand que la plus petite des valeurs propres :

$$\begin{aligned} \lambda_1 + \lambda &\leq \frac{\langle u, H(\lambda)u\rangle}{\langle u, u\rangle} \\ \underbrace{\lambda - \frac{\langle u, H(\lambda)u\rangle}{\langle u, u\rangle}}_{\lambda^B} &\leq -\lambda_1 \end{aligned}$$

Plus $\frac{\langle u,Hu\rangle}{\langle u,u\rangle}$ est faible, meilleure est la borne λ^B. Si le quotient est minimisé, alors $\lambda^B = -\lambda_1$ et u est le vecteur propre correspondant à λ_1, et nous connaîtrons précisément le lieu de séparation entre $\mathcal{N}$ et $\mathcal{L}$! C'est très intéressant car nous pourrons éviter la région $\mathcal{N}$ durant l'algorithme. [CGT00] conseille une méthode astucieuse et efficace pour calculer u. Nous l'avons bien entendu implémentée.

3.2.5 Valeurs de départ

[CGT00] fournit une expression pour l'initialisation des bornes λ^L et λ^U en combinant notamment le quotient de Rayleigh et les bornes de Gershgorin.

Nous avons désormais λ^L et λ^U. Comment choisir un λ initial tel que $\lambda \in [\lambda^L; \lambda^U]$? [CGT00] conseille $\lambda = \frac{\|g\|_2}{\Delta}$. De plus, en essayant l'algorithme complet, il nous est apparu qu'il est très important de choisir $\lambda = 0$ si $0 \in [\lambda^L; \lambda^U]$. Cela permet de trouver directement une solution intérieure ou de l'écarter d'emblée !

3.2.6 Cas difficile

Avec les limites 3.2.6,

$$\begin{aligned}\lim_{\lambda\to\pm\infty}\psi(\lambda) &\sim \tfrac{1}{\lambda^2}\to 0\\ \lim_{\lambda=-\lambda_i}\psi(\lambda) &\sim \tfrac{\gamma_i^2}{0}\to\infty\end{aligned}$$

nous avons négligé $\lim_{\lambda=-\lambda_i}\psi(\lambda)\sim\frac{\gamma_i^2}{0}$ lorsque $\gamma_i\approx 0$. Soit un problème dont les données sont

$$g=\begin{pmatrix}0\\1\\1\\1\end{pmatrix}, H=\begin{pmatrix}-2&0&0&0\\0&-1&0&0\\0&0&0&0\\0&0&0&1\end{pmatrix}$$

(Comme nous allons le montrer, g a également une importance, pas seulement H !) La décomposition SVD de H est

$$H=\underbrace{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}}_{U}\underbrace{\begin{pmatrix}-2&0&0&0\\0&-1&0&0\\0&0&1&0\\0&0&0&0\end{pmatrix}}_{\Lambda}\underbrace{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}}_{U^T}$$

La valeur propre la plus faible est $\lambda_1=-2$. Ainsi, tout comme sur la Figure 3.2, $-\lambda_1=2$ est dans les λ positifs. Soit $\nu_1=(1,0,0,0)^T$ le vecteur propre correspondant à λ_1. Nous calculons $\gamma_1=[Ug]_1=\langle(1,0,0,0),(0,1,1,1)\rangle=0$. Dans ce cas, $\lim_{\lambda=-\lambda_1}\psi(\lambda)\sim\frac{0}{0}\not\to\infty$. Le pôle en $-\lambda_1=2$ peut ne plus tendre vers ∞, voire complètement disparaître, et conduire à la Figure 3.4.

Soit $\Delta^2_{\text{cri}}=\psi(2)$ (« cri » signifiant « critique »). Deux cas se présentent :

- $\Delta^2<\Delta^2_{\text{cri}}$. Cas habituel, nous restons sur la courbe en jaune et avons une solution sur la frontière.
- $\Delta^2>\Delta^2_{\text{cri}}$. Pour rester à l'optimalité, nous devons être sur la ligne bleue, avec $1<\lambda<2$. La seconde condition 3.4 n'est plus respectée car $H(\lambda)$ n'est plus défini positif, et le résultat est incertain. On parle de cas difficile. Quand se produit-il ? Lorsque $\gamma_1\approx 0$, autrement dit lorsque $\langle\nu_1,g\rangle\approx 0$, c'est-à-dire si g est *presque* orthogonal au vecteur propre correspondant à la valeur propre la plus faible.

3.2.6.1 Répercussion sur l'équation séculaire $\phi(\lambda)$

Dans le cas difficile, le pic en $-\lambda_1$ peut disparaître. Avec l'équation séculaire, le pic vers le bas en $-\lambda_1$ n'existera pas, et la fonction $\phi(\lambda)$ pourrait rester positive dans la région de $-\lambda_1$. Il se pourrait donc qu'il n'y ait pas de $\lambda>-\lambda_1$ tel que $\phi(\lambda)<0$, entraînant que $\mathcal{L}$ sera vide, ne permettant pas la convergence.

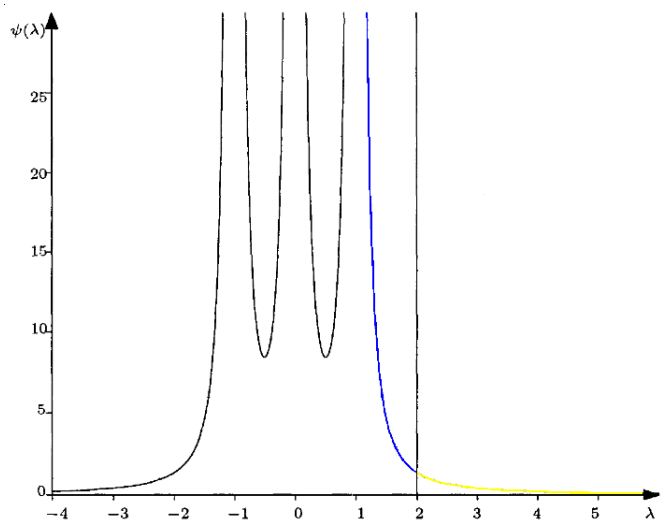


FIGURE 3.4 – Cas difficile : le pôle en $-\lambda_1 = 2$ a disparu (image [CGT00])

3.2.6.2 Résoudre le cas difficile

Lorsque nous rencontrons un blocage, il est préférable de se déplacer vers la frontière de la région de confiance. Ainsi, dans le cas difficile, nous choisissons de suivre la direction du vecteur propre correspondant à la valeur propre la plus faible. À partir du déplacement s déjà effectué au cours des précédentes itérations de l'algorithme, nous suivons le vecteur propre ν jusqu'à atteindre la frontière Δ :

$$\|s + \alpha\nu_1\|_2 = \Delta$$

C'est une équation du deuxième ordre facile à résoudre. Cependant, cela signifie-t-il que nous allons devoir calculer des valeurs propres afin d'obtenir ν_1 ? Heureusement non, car nous avons vu dans la Section 3.2.4.3 que nous pouvons estimer le vecteur propre correspondant à la valeur propre la plus faible grâce au quotient de Rayleigh.

3.2.6.3 Détection du cas difficile

Comment détecter le cas difficile sans devoir calculer les valeurs et vecteurs propres ? Le cas difficile peut résulter en la disparition de la région $\mathcal{L}$, la région $\mathcal{G}$ occupant sa place. Ainsi, si $\lambda \in \mathcal{G}$, nous considérons par sécurité que c'est d'office un cas difficile et le résolvons en tant que tel.

3.2.7 Terminaison

Trois cas sont envisageables.

1. Un itéré tombe dans $\mathcal{L}$, et tous les itérés suivants resteront dans $\mathcal{L}$. La convergence est quadratique.

2. Le problème est un « cas difficile » et $\mathcal{L}$ est vide. Grâce à l'intervalle d'incertitude, la convergence est également assurée.
3. Le minimiseur est à l'intérieur de la région de confiance, et $\mathcal{L}$ contient uniquement $\lambda = 0$. Le point précédent assure également la convergence. De plus, comme nous l'avons dit dans la Section 3.2.5, il est intéressant de commencer l'algorithme avec $\lambda = 0$ afin de considérer directement cette situation.

3.2.8 Algorithme

Nous présentons l'algorithme 3.2 qui assemble tous les algorithmes que nous avons vus dans les sections précédentes.

Algorithm 3.2 Une itération de l'algorithme complet de minimisation

Require:
Borne inférieure λ^L
Borne supérieure λ^U
$\lambda \in [\lambda^L; \lambda^U]$
Ensure:
Mise à jour de λ

$\lambda_{\text{new}} = \text{newton}(\lambda, \Delta)$ {Une itération de Newton (algorithme 3.1)}
if Décomposition de Cholesky échoue **then**
  $\lambda \in \mathcal{N}$
else if $\|s\| < \Delta$ **then**
  $\lambda \in \mathcal{G}$
else
  $\lambda \in \mathcal{L}$
end if
Mise à jour de λ^L et λ^U (cf. Section 3.2.4.2)
if $\lambda \in \mathcal{G}$ **then**
  Améliorer la borne inférieure avec Rayleigh (cf. Section 3.2.4.3)
  C'est peut-être le cas difficile. Résoudre $\|s + \alpha u\|_2 = \Delta$ (cf. Section 3.2.6.2)
end if
Vérifier s'il y a terminaison (cf. Section 3.2.7)
if terminaison **then**
  Arrêter l'algorithme
end if
if $\lambda_{\text{new}} \in \mathcal{N}$ **then**
  $\lambda \leftarrow$ choisir dans $[\lambda^L; \lambda^U]$ (cf. Section 3.2.5)
else
  $\lambda \leftarrow \lambda_{\text{new}}$
end if
return λ

3.2.8.1 Amélioration

Le coût principal réside dans la factorisation de Cholesky de $H(\lambda)$ qui est accomplie en temps $O(\frac{n^3}{6})$. En effet, il faut la calculer à chaque itération de l'algorithme. Nous allons effectuer une opération de décomposition qui ne devra être faite qu'une seule fois, et qui accélérera chaque itération de l'algorithme. Soit une décomposition

$$H = Q^T \overline{H} Q \quad \text{et} \quad H(\lambda) = Q^T(\overline{H} + \lambda I)Q$$

où Q est orthonormale. Moyennant les changements de variables $\overline{s} = Qs$, $\overline{g} = Qg$, $\overline{H} = QHQ^T$ et $\|\overline{s}\|_2 = \|s\|_2$, nous pouvons reformuler le problème d'optimisation en

$$\begin{aligned} \underline{\min} : \ & \langle \overline{g}, \overline{s} \rangle + \tfrac{1}{2} \langle \overline{s}, \overline{H}\overline{s} \rangle \\ \underline{\text{s.t.}} : \ & \|\overline{s}\|_2 \leq \Delta \end{aligned}$$

Une fois le problème résolu, la solution est $s^M = Q^T \overline{s}^M$. Tout l'intérêt est d'avoir une matrice $\overline{H}$ facile à factoriser, car l'opération de factorisation doit être effectuée à chaque itération. La décomposition Householder (que nous avons réimplémentée) a été adaptée pour diagonaliser H en une matrice tridiagonale $\overline{H}$. La décomposition de Cholesky d'une matrice tridiagonale se fait en temps linéaire. Cependant, la décomposition Householder est également 4 fois plus lente que la décomposition de Cholesky d'une matrice complète. Ainsi, cette amélioration apporte un gain au-delà de 4 itérations du sous-problème de région de confiance.

3.2.9 Eigensolution

Nous allons voir un nouvel algorithme dans lequel nous gérons le cas difficile de manière différente. Dans l'algorithme 3.2, nous assimilions $\lambda \in \mathcal{G}$ au cas difficile, ce qui est assez grossier. Ensuite, nous résolvions $\|s+\alpha\nu_1\|_2 = \Delta$. S'il n'y avait pas convergence, l'algorithme continuait.

Jusqu'à présent, nous avons évité de calculer directement $-\lambda_1$, prétextant que c'est trop coûteux. Pourtant, il y a lieu de se demander si c'est vraiment justifié, d'autant plus qu'il existe des méthodes d'approximation des valeurs propres. Nous allons donc estimer λ_1. Si $\phi(-\lambda_1) > 0$, ce qui est équivalent à $\|s(\lambda)\|_2 < \Delta$, alors c'est le cas difficile. Cette détection du cas difficile est nettement plus précise que ce que nous faisions dans l'algorithme 3.2. Avec l'eigensolver, si un cas difficile est détecté, nous résoudrons $\|s + \alpha\nu_1\|_2 = \Delta$ et arrêterons directement l'algorithme.

3.2.9.1 Estimer λ_1

Comment estimer λ_1 efficacement ? Des méthodes itératives permettent de converger vers une valeur propre :

1. [CK07] nous introduit la méthode de Power pour approximer un vecteur propre dominant (correspondant à la valeur propre la plus grande en valeur absolue).
2. [Che] propose l'algorithme Inverse Power afin de calculer le vecteur propre correspondant à la valeur propre la plus faible en valeur absolue.

3. Cependant, la valeur propre la plus faible ne correspond pas forcément à la valeur propre la plus faible en valeur absolue ! Nous pouvons converger vers une valeur propre qui n'est pas celle recherchée. En suivant [Che], nous avons implémenté la méthode Shifted Power, utilisant Power couplé à une méthode de déflation (cf. [BF97]), ceci permettant de scanner itérativement les valeurs propres, jusqu'à trouver celle recherchée. Cependant, cette méthode souffre beaucoup des erreurs d'arrondi et la convergence vers la valeur propre recherchée est incertaine.

Ensuite nous pourrons obtenir la valeur propre correspondante grâce au quotient de Rayleigh. □

L'algorithme procède comme suit. Si H est défini positif, nous prenons toujours initialement $\lambda = 0$ afin de trouver ou écarter de suite un minimum intérieur. Sinon, nous estimons la valeur propre λ_1. Si nous détectons effectivement un cas difficile, alors nous résolvons $\|s + \alpha\nu_1\|_2 = \Delta$ et arrêtons l'algorithme. Sinon, et c'est le cas général, nous itérons avec Newton afin de mettre à jour λ jusqu'à atteindre une convergence. Ces étapes sont reprises dans l'algorithme 3.3.

Algorithm 3.3 Eigensolver

```
1: if H est défini positif then
2:    λ ← 0
3: else
4:    λ ← estimation de −λ1 avec Shifted Power
5: end if
6: newton(λ, Δ)
7: if λ == 0 or ‖s‖ == Δ then
8:    return
9: else if ‖s‖ ≤ Δ then
10:   Résoudre ‖s + αν1‖2 = Δ {Cas difficile}
11:   return
12: end if
13: repeat
14:   λ = newton(λ, Δ) {Une itération de Newton (algorithme 3.1)}
15: until convergence
```

L'évaluation de $-\lambda_1$ a un impact majeur sur la convergence et la capacité à détecter les cas difficiles.

3.3 Résolution approximative

Étant donné que c'est la convergence de l'algorithme de trust-region qui importe, il y a lieu d'envisager une résolution approximative du sous-problème de trust-region. Nous nous tournons vers la méthode des gradients conjugués. Cependant, elle ne peut être appliquée telle quelle. En effet, elle ne fonctionne que pour des problèmes convexes,

alors que le nôtre est peut-être non convexe voire indéfini. Dès lors, [CGT00] adapte la méthode, qui portera désormais le nom de « gradients conjugués tronqués de Seihaug-Toint ».

3.3.1 Gradients conjugués pour un problème convexe

Le principe de base est d'effectuer des déplacements selon un ensemble de directions p_i satisfaisant la H-orthogonalité $\langle p_i, Hp_j \rangle = 0$. [KKI03] démontre que si H est symétrique définie positive et de taille $n \times n$, alors le minimum est atteint après n déplacements dans l'espace à n dimensions. De plus, s'il n'y a que p valeurs propres distinctes, alors le minimum est obtenu en p itérations.

Il est intéressant de noter que le premier itéré correspond au point de Cauchy.

Même si le problème est convexe, il est possible que le minimum soit en dehors de la région de confiance. Un choix évident est donc de s'arrêter à la frontière de la région de confiance lorsqu'elle est atteinte.

3.3.2 Adaptation au cas non convexe

Si H n'est pas convexe et qu'à l'itéré k, nous constatons que $\langle p_k, Hp_k \rangle \leq 0$, alors le choix le plus logique est d'avancer dans la direction p_k jusqu'à atteindre la région de confiance, afin d'effectuer le plus grand pas possible. Nous résoudrons donc $\|s_k + \alpha p_k\| = \Delta$ et arrêterons l'algorithme.

3.3.3 Gradients conjugués préconditionnés

Comme [KKI03] le met en évidence, même si la méthode des gradients conjugués pour les problèmes convexes est très efficace, la pratique ne suit pas exactement la convergence théorique à cause des erreurs d'arrondi. Afin d'accélérer la convergence, nous employons des préconditionneurs transformant la minimisation en

$$\begin{aligned} \underline{\min} : \ & \langle M^{-1}g, s \rangle + \tfrac{1}{2}\langle s, M^{-1}Hs \rangle \\ \underline{\text{s.t.}} : \ & \|s\|_M \leq \Delta \end{aligned}$$

- Préconditionneur de Jacobi : $M = \text{diag}\,(H_{11}, \ldots, H_{nn})$. Cependant, il échoue si tous les termes diagonaux sont égaux.
- Préconditionneur de Cholesky : $M = L$ avec $H = LL^T$
- Préconditionneur incomplet de Cholesky. Un défaut de la décomposition de Cholesky est qu'il est très probable que les zéros dans H disparaissent complètement dans la factorisation. Ainsi, l'idée de la factorisation incomplète de Cholesky est de forcer des zéros là où il y en avait dans H.

Chapitre 4

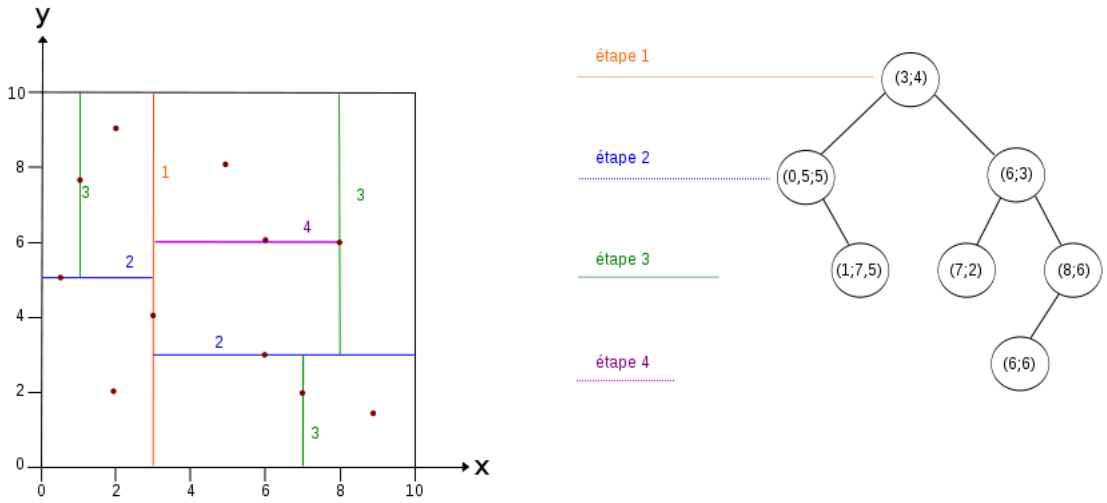
Recherche spatiale

La régression dans une sphère nécessite la connaissance des points qui s'y trouvent. Ces points font partie d'un sous-ensemble dans l'ensemble des points de données. Déterminer ce sous-ensemble est un problème non trivial pour lequel beaucoup de recherches ont été menées au vu du nombre d'applications qui en ont besoin. En effet, récemment, des bases de données multimédia ont vu le jour, dont le but est d'uniformiser la gestion de données de voix, vidéo, image (forme, texture, couleur) et texte. Parmi les divers services que de tels systèmes doivent être à même de fournir, la recherche sur base du contenu a un rôle majeur : un utilisateur doit pouvoir y effectuer des recherches sur base de certains critères. En particulier, il est nécessaire de mettre au point des techniques d'indexage pouvant supporter des recherches de similarité. Par ce terme, nous entendons la mise en évidence de ressemblance d'objets par rapport à leurs variables, puissent-elles exprimer la ressemblance entre deux voix, des modèles communs dans deux images ou, comme dans notre application, la proximité de points.

Dans ce chapitre, nous implémentons et discutons plusieurs méthodes de recherche spatiale nous permettant de trouver tous les points dans une sphère de recherche. Les chapitres 2 et 3 expliquent comment utiliser ces points pour calculer un modèle qui est minimisé dans la région de confiance.

Globalement, les méthodes de recherche spatiale sont basées sur le partitionnement de l'espace ou le partitionnement des données en clusters. Par exemple, dans ce dernier cas, nous regroupons simplement des points proches en clusters afin de faciliter la recherche.

Nous commençons par deux méthodes de recherche triviales (kd-tree et quadtree, Section 4.1) et, étant mécontent des performances, nous passerons ensuite à une méthode plus avancée (M-Tree, Section 4.2) qui permettra de doubler les performances. Cependant, ces 3 méthodes ne sont pas satisfaisantes à hautes dimensions. Nous avons donc implémenté une méthode très originale avec une part d'innovation : technique de projection (Section 4.3) et méthodes d'indexage unidimensionnel équipées de la projection (Section 4.4). Nous obtiendrons des performances 12 fois meilleures à moyennes dimensions et encore bien plus élevées à hautes dimensions.



(a) Décomposition (b) Arbre correspondant

FIGURE 4.1 – Exemple de kd-tree

4.1 Partitionnement de l'espace

4.1.1 kd-tree

Le kd-tree (k-dimensional tree) est une structure de données qui partitionne l'espace afin d'organiser les points dans un espace à k dimensions.

Le kd-tree est un arbre binaire où chaque nœud divise l'espace en deux sous-espaces de k dimensions grâce à un hyperplan. Les points à gauche et sur l'hyperplan iront dans le fils à gauche, et les points à droite de l'hyperplan iront dans le fils à droite.

4.1.1.1 Construction

Même si le choix de l'hyperplan laisse cours à de nombreuses possibilités, le kd-tree impose malgré tout une règle commune. Soit un espace et des données illustrés sur la Figure 4.1 (a). Désignons une dimension de départ, par exemple X. Au niveau 1 (racine), nous divisons selon l'axe X. Au niveau 2 (les deux fils de la racine), nous divisons chaque nœud selon l'axe Y. Au niveau 3, nous divisons selon l'axe X et ainsi de suite, selon des cycles. La procédure s'arrête lorsque chaque feuille ne contient plus qu'un seul point. Il n'est alors plus nécessaire de diviser.

La Figure 4.1 (b) fournit l'arbre correspondant à la décomposition de la Figure 4.1 (a).

Remarquons qu'il n'est pas requis de prendre le point médian pour diviser. Cependant, si on ne le fait pas, nous n'avons aucune garantie que l'arbre soit balancé. En effet, si l'hyperplan est placé n'importe où et laisse 2 points à sa gauche et 10 à sa

droite, l'arbre final ne sera pas du tout balancé. En prenant toujours le point médian, l'arbre sera balancé.

4.1.1.2 Recherche

Nous recherchons les points contenus dans un rectangle prédéfini, dont les bords sont parallèles aux axes. Il suffit simplement de descendre l'arbre. À partir d'un certain nœud, nous explorons les fils dont le sous-espace intersecte le rectangle de recherche, et ainsi de suite jusqu'aux feuilles.

4.1.2 Quadtree – Octree

Tout comme le kd-tree, le quadtree et l'octree (Figure 4.2) divisent l'espace afin de permettre la recherche spatiale. Cependant, le quadtree et l'octree divisent autour d'un point, alors que le kd-tree divise sur une dimension. Le quadtree est simplement la version à 2 dimensions, tandis que l'octree est la généralisation à 3 dimensions, mais le principe reste le même. Désormais, nous utiliserons le terme « quadtree » quelle que soit la dimension : 2, 3 ou plus.

Quelle que soit la distribution des points, le quadtree divise en un point : le centre. À 2 dimensions, il y aura 4 fils (supérieur gauche, supérieur droit, inférieur gauche et inférieur droit). À 3 dimensions, il y aura 8 fils. À n dimensions, il y en aura 2^n. C'est très différent du kd-tree où il y avait toujours deux fils (arbre binaire). Tout comme le kd-tree, le quadtree effectue des divisions jusqu'à ce que chaque sous-espace ne contienne plus qu'un seul point au maximum.

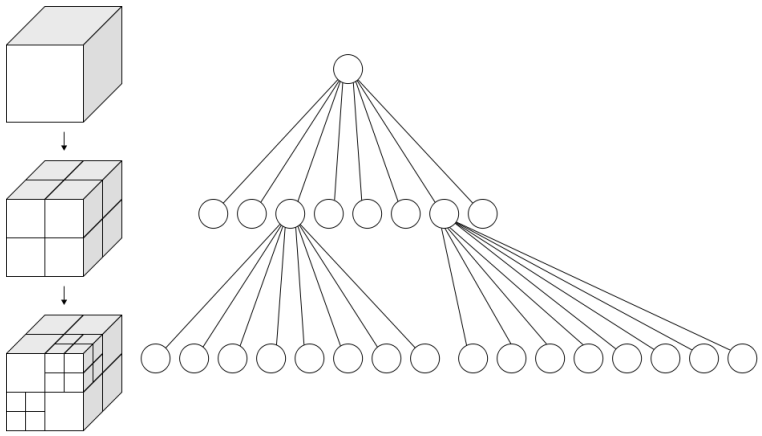


FIGURE 4.2 – Divisions dans un octree et arbre correspondant (image Wikipedia)

4.2 Partitionnement des données — M-Tree

Le M-Tree est une structure de données qui se base également sur un arbre. Une différence majeure par rapport au kd-tree et au quadtree est que le M-Tree partitionne les données au lieu de partitionner l'espace. Les arbres métriques tels que le M-Tree (*metric tree*) ne considèrent que la distance relative entre les objets (plutôt que leur position absolue dans un espace multidimensionnel) afin d'organiser les partitions, et requièrent que la fonction utilisée pour mesurer la distance (dissimilarité) respecte l'inégalité triangulaire.

4.2.1 Structure du M-Tree

Chaque nœud interne de l'arbre sauvegarde un objet de routage O_r. C'est une sphère. Chaque point p de son sous-arbre est contenu dans O_r, autrement dit $d(O_r, p) \leq r(O_r)$ où $r(O_r)$ est la rayon de O_r. Les données (des points, dans notre application) sont sauvegardées dans les feuilles. Le M-Tree est un arbre globalement balancé, ce qui signifie que sa profondeur est $O(\log(m))$ aussi bien en moyenne qu'en *worst-case*.

4.2.2 Split

Afin d'insérer un objet (un point, dans notre application), nous descendons l'arbre jusqu'à une feuille correspondant à une sphère pouvant contenir ce point. Nous attribuons une capacité maximale aux sphères afin de ne pas dégénérer en une recherche linéaire lors d'une recherche ultérieure de points. Si l'on insère un point dans une sphère qui est complète, nous n'avons d'autre choix que de séparer les points en deux sphères. Cette procédure porte le nom de `split`. Elle est le cœur du M-Tree et est capitale pour ses performances.

La Figure 4.3 présente le principe général d'un split. Nous tentons d'insérer un objet (en jaune) dans N. Cependant, la capacité maximale est fixée à 4 et N déborde. Nous créons un nouveau nœud frère N_2, et N devient N_1. Nous répartissons les 5 anciens éléments de N entre N_1 et N_2. Dans notre exemple, N_1 contient *in fine* 2 objets alors que N_2 en contient 3, pour un total de 5 éléments. N_p s'est vu enrichi d'un nouveau fils. Avant, il en avait deux ; désormais, il en a trois, ce qui ne le fait pas déborder. Si jamais N_p avait débordé, il aurait fallu le splitter également. Il se pourrait donc que tous les parents doivent splitter (il y a $\log(m)$ parents). En conclusion, le split est réalisé en temps t, la procédure complète d'insertion est de l'ordre de $O(t \log m)$.

L'algorithme 4.1 présente la forme générale d'une procédure de split. Deux fonctions sont très importantes :

promote À partir de N, `promote` désigne un objet de routage O_1 pour N_1 et un objet de routage O_2 pour la nouvelle partition N_2.

partition Connaissant O_1 et O_2, `partition` prend chaque élément de N et le met soit dans N_1 soit dans N_2 □

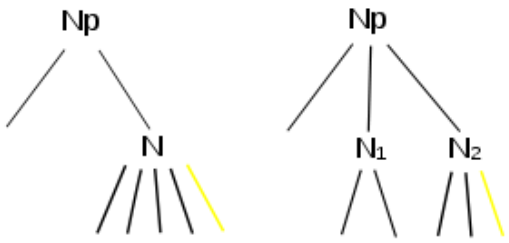


FIGURE 4.3 – Split : avant et après

Algorithm 4.1 split d'un nœud dans un M-Tree

Ensure:
Une nouvelle instance du M-Tree contenant une nouvelle partition
1: ***promote***(N, O_1, O_2)
2: ***partition***(N, N_1, N_2)
3: **if** N est la racine actuelle **then**
4: Créer une nouvelle racine N_p qui sera le parent de N_1 et N_2
5: **end if**

Une implémentation spécifique de `promote` et de `partition` définit une procédure de split (*split policy*). La *split policy* amène à plusieurs considérations :

- **Complexité** Soit B la taille maximale d'un nœud. Nous tentons d'ajouter un nouvel élément, faisant déborder le nœud. Nous devons donc reclasser $B + 1$ éléments. Il y a 2^{B+1} partitions possibles. Si un algorithme doit toutes les considérer, la complexité est exponentielle.
- **Critères de qualité** Qu'est-ce qu'un bon partitionnement ? C'est un partitionnement qui sera plus avantageux pour une recherche rapide. Deux critères tombent sous le sens. Les deux partitions devraient
  1. avoir le rayon le plus réduit possible ;
  2. avoir le volume commun le plus réduit possible, donc en rendant les centres suffisamment éloignés l'un de l'autre.
- **Partitions balancées** Il serait préférable que la taille des partitions soit relativement balancée, afin que le M-Tree (même s'il est globalement balancé) n'ait pas une profondeur démesurée.

[CZP97] et [FTS02] décrivent la plupart des méthodes de `promote` et de `partition` qui ont été implémentées dans ce mémoire, et qui sont le sujet des deux sections qui suivent.

4.2.3 Promote

1. **Random (random)** Deux entrées (parmi les $B+1$ éléments à reclasser) sont sélectionnées aléatoirement pour devenir O_1 et O_2.
2. **Minimum sum of RADii (m_RAD)** Considère toutes les paires entre les $B+1$ éléments, et choisit celle qui minimise $r(O_{p_1}) + r(O_{p_2})$.
3. **Minimize the Maximum RADii (mM_RAD)** Considère toutes les paires entre les $B+1$ éléments, et choisit celle qui minimise $\max\{r(O_{p_1}), r(O_{p_2})\}$.
4. **Maximum Lower Bound on DISTance (M_LB_DIST)** Choisi l'objet de routage O de N pour devenir l'objet de routage O_1, et détermine O_2 comme étant l'objet le plus éloigné de O.
5. **Minimize Overlapping (m_OVERLAPPING)** C'est la méthode la plus coûteuse. Pour chaque paire d'objets de routage, la méthode fait appel à `partition` afin d'obtenir les rayons $r(O_{p_1})$ et $r(O_{p_2})$ correspondants. Parmi toutes ces paires, la méthode choisit celle dont le partitionnement minimise le chevauchement (en norme L_2) entre les deux sphères.

4.2.4 Partition

1. **Hyperplan (hyperplan)** Assigner l'objet $O_j \in N$ à l'objet de routage le plus proche. Si $d(O_j, O_1) \leq d(O_j, O_2)$ alors assigner O_j à N_1 ; sinon, l'assigner à O_2.
2. **Balancé (balanced)** Assigner à N_1 les $\frac{B+1}{2}$ plus proches objets de O_1 ; assigner les autres à N_2.

4.2.5 Méthodes complètes de split

Voici trois autres méthodes qui effectuent le split sans passer par une fonction `promote` :

1. **Déviation standard (stddev)** L'algorithme 4.3 partitionne selon le classement des éléments sur une dimension.

Algorithm 4.2 Partitionnement par déviation standard

1: **for all** $i \in d$ **do**
2: Calculer la déviation selon la dimension i
3: **end for**
4: $s \leftarrow$ la dimension de déviation maximale
5: Classer les éléments selon la dimension s dans l'ordre des éléments les plus proches de la déviation standard
6: Placer les $\frac{B+1}{2}$ dans N_1, et les autres dans N_2.

2. **K-means (k-means)** Le k-means partitionne des éléments dans k clusters. Dans notre application, $k = 2$ car nous voulons deux clusters. Les entrées sont assignées à

l'un des deux clusters dont le centroïde est le plus proche de l'entrée, en recalculant les centroïdes de manière itérative (algorithme 4.3).

Algorithm 4.3 Partitionnement par k-means

```
1: Assigner aléatoirement chaque entrée à un cluster
2: repeat
3:    Calculer les centroïdes
4:    Assigner chaque entrée au cluster de centroïde le plus proche
5: until Les clusters ne bougent plus ou nombre maximum d'itérations atteint
```

3. **Sequential K-means (sk-means)** Le SK-means se différencie du K-means en recalculant les centroïdes dès qu'un élément change de partition (le K-means recalcule les centroïdes après avoir reclassé la totalité des éléments). □

Une fois l'un de ces trois splits effectué (stddev, k-means ou sk-means), nous avons deux partitionnements mais nous ne connaissons pas encore leur centre respectif. Comme illustré sur la Figure 4.4, ce centre ne correspond pas au centre de gravité.

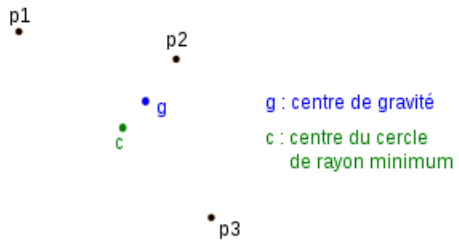


FIGURE 4.4 – Centre du cercle de rayon minimum et centre de gravité

C'est un problème conique, donc très facile, et nous allons utiliser le solveur Knitro pour le résoudre. y est le centre du cercle que nous voulons obtenir, et r son rayon. p_i est un point, la partition contenant $i = 1, \ldots, m$ points.

$$\begin{aligned} \underline{\min} : \ & r \\ \underline{\text{s.t.}} : \ & r \geq \|y - p_i\|_2 \qquad \forall i = 1, \ldots, m \end{aligned}$$

Enfin, nous ne devons pas uniquement englober des points. Une feuille de l'arbre contient des points, mais un nœud interne contient d'autres cercles, qu'il faut pouvoir englober à leur tour (Figure 4.5). Il faut modifier le problème conique afin de prendre en charge les rayons d_i des cercles :

$$\begin{aligned} \underline{\min} : \ & r \\ \underline{\text{s.t.}} : \ & (r - d_i)^2 \geq \|y - p_i\|_2^2 \qquad \forall i = 1, \ldots, m \\ & r \geq d_i \qquad \forall i = 1, \ldots, m \end{aligned}$$

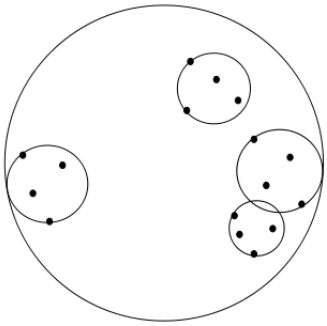


FIGURE 4.5 – Englober des cercles

4.2.6 Aller plus loin

4.2.6.1 Un clustering plus compliqué

La méthode de split la plus avancée consisterait à effectuer le clustering le plus avantageux possible, c'est-à-dire minimisant le rayon des deux cercles et avec le volume commun le plus faible. Chaque point devrait appartenir à un des deux clusterings. Ajouter à cela la contrainte de distance, ce serait donc un problème d'optimisation entière non linéaire, qui est difficile à résoudre. De plus, c'est une optimisation devant être effectuée à chaque split. Clairement, un tel dispositif est exagéré et n'apporterait presque aucun gain car, comme nous allons l'introduire dans la section suivante, les performances du M-Tree (et de façon générale, tout algorithme de partitionnement) sont limitées, quel que soit l'algorithme de clustering !

4.2.6.2 R-Tree : partitions sphériques

Après le M-Tree, l'autre structure majeure en partitionnement des données porte le nom de R-Tree. Le R-Tree est tout à fait similaire au M-Tree, si ce n'est qu'il utilise des rectangles (d'où le nom R-Tree) au lieu de sphères. Naturellement, d'autres méthodes de split sont nécessaires. Cependant, celles-ci sont moins évoluées. Intuitivement, nous pouvons penser que des rectangles doivent offrir un clustering moins approprié que des sphères, si l'on est en norme L_2. [FTS02] confirme que M-Tree est nettement meilleur que R-Tree pour la recherche de similarité à hautes dimensions.

Il est démontré dans [WSB98] que la taille des partitions augmente avec les dimensions, et lors d'une recherche, de plus en plus de partitions devront être visitées, conduisant au comportement linéaire. Avec des partitions rectangulaires, [WSB98] démontre que la recherche exhaustive devient plus efficace vers 15$\sim$20 dimensions.

Les sphères partitionnent mieux les données. [WSB98] montre que le seuil est atteint à 26 dimensions. Au-delà, une recherche exhaustive sera plus rapide, en moyenne.

4.2.6.3 Partitionnement optimal théorique

Enfin, [WSB98] démontre qu'il n'existe aucun partitionnement pouvant être efficace à hautes dimensions. Nous allons utiliser des clusters reliant directement des points, afin de construire les clusters de surfaces les plus réduites possibles. C'est la meilleure option théorique (irréalisable en pratique), et elle définit les performances optimales au-delà desquelles aucune méthode de partitionnement ne pourra aller.

Le seuil est atteint pour 610 dimensions. Aucune méthode de partitionnement ne peut donc aller au-delà. 610 semble relativement grand, mais en pratique, les meilleures méthodes n'atteignent que 10 ou 20 dimensions. Cependant, en deça de 10 dimensions, les méthodes de partitionnement ont de la place pour faire mieux que la recherche exhaustive. Ce résultat est tout de même décevant car quel que soit l'effort fourni afin de développer une méthode avancée de split, nous ne pouvons franchir une certaine limite de performances. Afin de faire mieux et de pouvoir monter dans les dimensions sans crainte de perte de performances, nous allons mettre en œuvre une méthode qui ne repose pas sur un partitionnement. La Section 4.3 introduit une technique de projection, et la Section 4.4 l'utilisera dans des structures d'indexage qui offriront des performances très largement supérieures au M-Tree !

4.3 Projection vers un espace linéaire

Malgré les performances théoriques optimales des précédentes méthodes d'accès, les résultats pratiques sont peu satisfaisants malgré les nombreuses méthodes de split expérimentées. En effet, dans tous les cas, les performances se dégradent par rapport au nombre de dimensions. De plus, même si jusqu'à présent, nous nous inquiétions uniquement des performances, nous aimerions également obtenir une méthode qui s'intègre aisément dans d'autres technologies de recherche et de stockage des données. Dans cette section ainsi que la suivante, nous allons mettre en œuvre de nouveaux outils répondant à ces attentes :

1. meilleures performances à dimensions moyennes (6∼10) ;
2. performances moins dégradées à hautes dimensions ;
3. intégrables dans un SGBD existant en ne demandant que peu d'adaptations.

4.3.1 Space-filling curves

La technologie de base repose sur les *space-filling curves*. Le concept d'une *space-filling curve* est de projeter l'espace multidimensionnel vers un espace à une dimension, afin de s'affranchir du problème de dimensionnalité.

Sur la Figure 4.6, nous pouvons observer que la courbe (commençant dans le coin supérieur gauche) forme des « Z », d'où son nom de Z-Curve. Ses valeurs s'étalent de 0 jusque 63 dans un espace à une dimension. Dans ce cas, c'est donc un mapping d'un espace à deux dimensions vers un espace unidimensionnel. La Z-Curve est un exemple

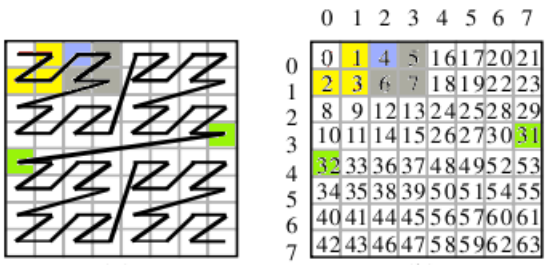


FIGURE 4.6 – *Space-filling curve* de type Z-Curve

de *space-filling curve*, comme il en existe d'autres. La Figure 4.7 illustre une *space-filling curve* à 3 dimensions.

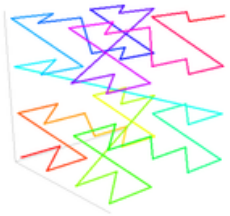


FIGURE 4.7 – *Space-filling curve* à 3 dimensions (image Wikipedia)

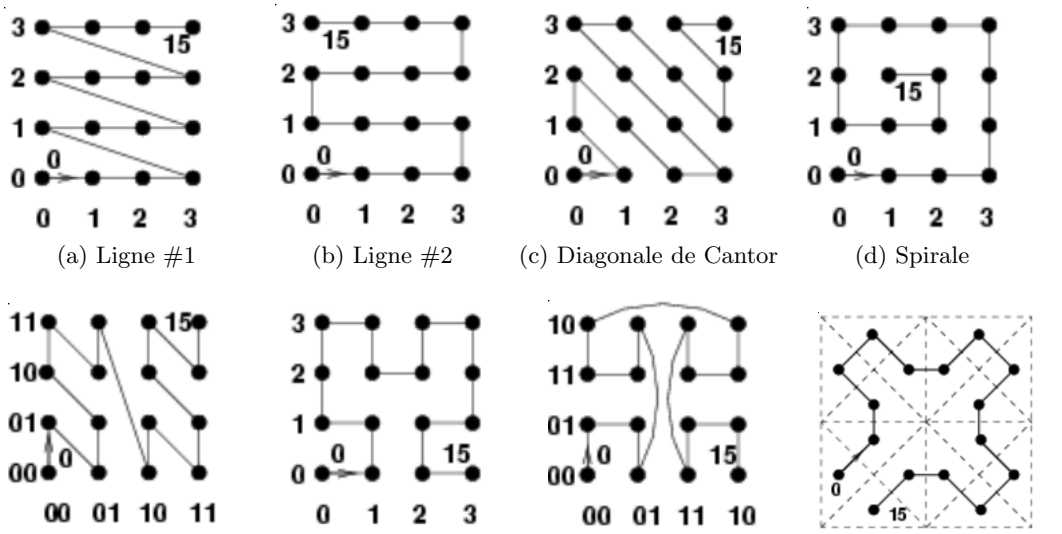
4.3.2 Terminologie

- Une valeur de la Z-Curve en un certain point est une Z-value. Nous utilisons la fonction univoque $\alpha = Z(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{Z}^n$ qui mappe un point de donnée (multidimensionnel) vers une Z-value.
- Une séquence de Z-values consécutives $(\alpha, \alpha + 1, \ldots, \beta)$ est une Z-region qui est dénotée $[\alpha; \beta]$.
- Dans la Z-Region $[\alpha; \beta]$, β est la *region address*.

4.3.3 Représentation de la proximité

Le problème principal reste la recherche de proximité. À partir d'un point (correspondant à une seule Z-value), il faut trouver les points situés en deçà d'une certaine distance. Ces points correspondent à des Z-values proches de la Z-value autour de laquelle est effectuée la recherche. La Z-Curve devrait donc favoriser autant que possible la proximité.

Dans l'exemple de la Figure 4.6, partons de la Z-Value $\alpha = 4$ (en bleu). Les trois Z-values précédentes sont 3, 2 et 1 (en jaune). Les trois suivantes sont 5, 6 et 7 (en gris). Sur la Figure 4.6, les cases $\{1, 2, 3, 5, 6, 7\}$ sont assez proches de la case 4. La Z-Curve a donc traduit la proximité entre ces points. Cependant, la Z-Curve n'est pas parfaite.



(a) Ligne #1 (b) Ligne #2 (c) Diagonale de Cantor (d) Spirale

(e) N-Curve (f) Hilbert (g) Gray (h) Sierpinski (triangle)

FIGURE 4.8 – Exemples de *space-filling curves* (image [LGMR05])

Considérons les Z-values 31 et 32 (en vert). Sur la Z-curve, ces deux Z-Values sont consécutives (donc considérées comme proches) mais elles sont très espacées dans l'espace d'origine (l'espace à deux dimensions). Il est manifestement impossible de concevoir une courbe qui représente parfaitement la proximité dans tout l'espace. Nous ne pouvons dès lors qu'utiliser des courbes qui représentent cette proximité du mieux possible.

4.3.4 Autres exemples de *space-filling curves*

Cf. Figure 4.8, les courbes par lignes, diagonales et spirales sont de mauvaise qualité car elle retranscrivent mal la proximité en de nombreux endroits. Par contre, N-Curve (et Z-Curve), Hilbert, Gray et Sierpinski semblent plus prometteuses. Dans ce travail, nous n'allons utiliser que N-Curve et Z-Curve.

4.3.5 Exemple de construction d'une courbe : N-Curve

La représentation binaire du nombre 6 est 110_b (« b » pour « binaire »). Nous considérons toujours que le bit de poids fort (le plus à gauche) a le numéro 0. Le bit directement à droite a le numéro 1 etc. jusqu'au bit le plus à droite (bit de poids faible).

Sur la Figure 4.9, nous observons que $N((01, 10)) = 6$ (01_b sur l'axe x, 10_b sur l'axe y), avec $N : \mathbb{N}^d \to \mathbb{N}$ qui, à une cellule de l'espace de départ, associe la N-Value correspondante (6 dans cet exemple). Pour obtenir cette valeur 6, il suffit de mettre en commun les bits (bit-interleaving) comme présenté sur la Figure 4.10

La procédure se généralise simplement à d dimensions en itérant sur toutes les dimensions. C'est donc une opération linéaire sur le nombre de bits.

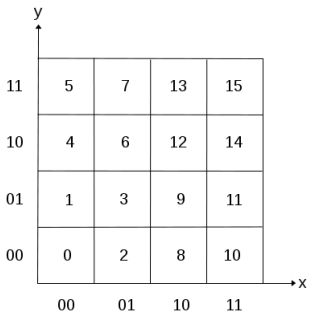


FIGURE 4.9 – N-Curve à 4 divisions par dimension, soit 16 éléments au total

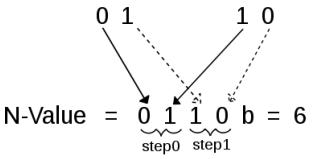


FIGURE 4.10 – Bit-interleaving pour N-Curve

4.3.6 GetNextZValue

L'algorithme `GetNextZValue` est le cœur de la méthode d'indexage, et il va en définir les performances. Soit la recherche de toutes les cellules à l'intérieur d'une zone de recherche Q (en pointillés sur la Figure 4.11). Commençons avec la cellule 12 qui fait partie de Q (il est aisé de vérifier si une cellule appartient à Q ou non). En incrémentant 12, nous obtenons 13 qui fait également partie de Q. Nous obtenons 14 et 15 en incrémentant également. Cependant, en incrémentant de nouveau, nous obtenons 16 qui ne fait pas partie de Q. Nous aimerions pouvoir sauter au prochain point faisant partie de Q, à savoir 36 (c'est le *next intersection point*). Dans un grand espace multidimensionnel avec énormément de cellules, suivre la Z-curve jusqu'à trouver le *next intersection point* peut être très long (temps exponentiel). [RVF+00] présente un algorithme (appelé GetNextZValue) tournant en temps linéaire par rapport au nombre de bits dans la Z-Value (donc très rapide), fonctionnant aussi bien pour la Z-Curve que pour la N-Curve. (Ainsi, par la suite, lorsque nous parlerons de Z-Value, les développements seront conceptuellement similaires pour une N-Value.)

Cependant, nous avons acquis la certitude que l'algorithme `GetNextZValue` présenté par [RVF+00] est faux. Plus précisément, il y a des valeurs à partir desquelles `GetNextZValue` rendra un résultat complètement faux. Heureusement pour lui, [RVF+00] utilise `GetNextZValue` en combinaison avec une autre structure de recherche binaire (un UB-Tree) configurée de telle sorte que les cas critiques ne se produiront pas. Dans la littérature scientifique, un second document aborde `GetNextZValue`. Dans sa thèse

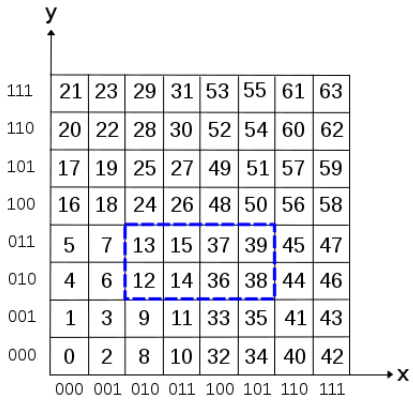


FIGURE 4.11 – Recherche dans une zone de l'espace

[Pru07], Antonín Prukl conclut que « *The function [GetNextZValue] was presented in [RVF+00] however the description was rather vague and contained mistakes* ». Prukl reformule alors l'algorithme GetNextZValue en clarifiant quelques points qui étaient effectivement laissés à l'interprétation. Cependant, selon nous, les deux algorithmes font la même erreur. De fait, Prukl l'utilise en combinaison avec un UB-Tree uniquement, et le cas erroné n'a jamais dû se produire devant ses yeux.

Dans ce mémoire, nous avons corrigé `GetNextZValue` afin qu'il fonctionne en toutes circonstances. Cela nous permet de créer les nouvelles structures d'indexage Zarray et Zhash qui sont encore plus performantes que le UB-Tree.

Il n'y a pas de théorie derrière l'algorithme `GetNextZValue`. C'est un ensemble de règles et de constatations formulées en analysant les propriétés d'une Z-Curve (ou N-Curve, nous ne ferons plus la différence entre les deux pour l'explication) que l'algorithme peut être construit pas à pas. Nous n'allons pas le décrire précisément car l'algorithme est très compliqué. Nous allons seulement en donner les notions de base sur un exemple, Figure 4.12.

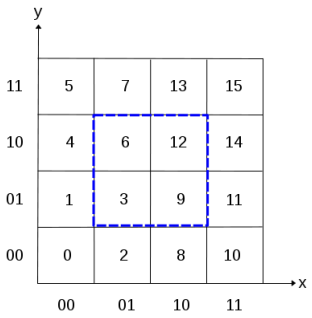


FIGURE 4.12 – Recherche dans une zone de l'espace

Supposons que 3 est connu comme appartenant à Q. Nous l'incrémentons, et voyons que $4 \notin Q$. Nous observons sur la figure qu'en fait, next $= 6$. Le premier objectif de l'algorithme est de déterminer le premier bit (en commençant par le bit de poids le plus fort) qui doit être modifié (bit en magenta), pour passer de `nisp` à `next` :

$$\begin{aligned} \text{cur} = 3 &= 0011 \\ \text{nisp} = \text{cur} + 1 = 4 &= 0100 \\ \text{next} = 6 &= 0110 \end{aligned}$$

Nous nommerons ce bit `changeBP`.

Lemme 4. `changeBP` *est obligatoirement un bit passant de 0 à 1. En ce qui concerne les bits à droite de* `changeBP`*, certains changent, d'autre non, et l'algorithme va devoir déterminer leur mise à jour.*

Comment déterminer `changeBP` ? Il n'y a pas d'explication théorique simple, mais un algorithme conçu en dressant des constatations sur l'allure des courbes :

- La cellule n°4 est à gauche de Q. Vu l'allure de la courbe en N, celle-ci va forcément passer par un point p tel que $\Pi_x(p) = 01_b$, où Π_d est l'opérateur de projection sur l'axe d.
- Si $\Pi_d(p) \subseteq \Pi_d(Q)$, alors `changeBP` ne sera manifestement pas à un bit correspondant à la dimension d.

Ces observations permettent de calculer `changeBP`. Ensuite, il faut mettre à jour les bits à droite de `changeBP`. Nous ne l'expliquerons pas dans les détails car il est nécessaire de dresser de très nombreux exemples afin d'y arriver. Quoiqu'il en soit, cela est réalisé par l'intermédiaire de plusieurs conditions, et cette procédure tourne en temps linéaire par rapport au nombre de bits dans la Z-Value. L'algorithme `GetNextZValue` fonctionne aussi bien pour la Z-Curve que pour la N-Curve. En réalité, tout ce qui change entre les deux, c'est l'implémentation des méthodes

- `BP(dim, step)` renvoie le *bit position* dans la (Z/N)-Value correspondant à la dimension `dim` et au step `step` ;
- `DIM(bp)` renvoie la dimension correspondant au *bit position* `bp` dans la (Z/N)-Value ;
- `STEP(bp)` renvoie le step correspondant au *bit position* `bp` dans la (Z/N)-Value.

L'algorithme 4.4 est basé sur celui proposé dans [RVF$^+$00]. Les lignes surlignées en jaune correspondent aux modifications apportées par rapport à [RVF$^+$00], qui sont donc notre contribution personnelle à l'algorithme afin de le rendre parfaitement correct.

Que faire avec cet algorithme ? À partir d'une Z-Value, il nous donne la prochaine Z-Value appartenant à la zone de recherche Q. Nous pouvons donc l'utiliser récursivement afin de lister toutes les Z-Values appartenant à Q. Tout ce qu'il nous manque, c'est une structure d'indexage unidimensionnel pour obtenir tous les points ayant une certaine Z-Value donnée. Cette problématique est exposée dans la section suivante.

Algorithm 4.4 GetNextZValue

Require:

$nisp$: une Z-Value $\notin Q$

$last$: la plus grande Z-Value connue telle que $last < nisp$ et $last \in Q$

$$flag[i] = \begin{cases} -1 & \text{si nisp est en dessous du minimum de } Q \\ & \text{dans la dimension } i \\ 0 & \text{si nisp est dans } Q \text{ dans la dimension } i \\ 1 & \text{si nisp est au-dessus du maximum de } Q \\ & \text{dans la dimension } i \end{cases}$$

$$outStep[i] = \left| \begin{array}{l} \text{le premier step dans la dimension } i \text{ pour lequel le bit} \\ \text{correspondant diffère entre nisp et le bord le plus proche de } Q, \\ \text{ou } \infty \text{ si nisp est dans } Q \text{ dans la dimension } i \end{array} \right.$$

$$saveMin[i] = \left| \begin{array}{l} \text{le premier step dans la dimension } i \text{ pour lequel le bit} \\ \text{correspondant diffère entre nisp et le bord inférieur de } Q, \\ \text{ou } \infty \text{ si nisp est en dessous de } Q \text{ dans la dimension } i \end{array} \right.$$

$$saveMax[i] = \left| \begin{array}{l} \text{le premier step dans la dimension } i \text{ pour lequel le bit} \\ \text{correspondant diffère entre nisp et le bord supérieur de } Q, \\ \text{ou } \infty \text{ si nisp est au-dessus de } Q \text{ dans la dimension } i \end{array} \right.$$

$outStep = \min(outStep[i])$

d : la dimension correspondant à $outStep$

Ensure:

Calcule le prochain point d'intersection après $nisp$, dans Q

1: $changeBP \leftarrow \text{BP}(d, outStep)$
2: **if** $flag[d] == 1$ **then**
3: {changeBP est incorrect, il faut le corriger}
4: $changeBP \leftarrow \max(\{bp \mid bp < changeBP$ and
saveMax[DIM(bp)] $\neq \infty$ and
$bp \geq \text{BP}(\text{DIM}(bp), \text{saveMax}[\text{DIM}(bp)])$ and
get_bit($nisp, bp$) $== 0\}$)
5: $changeBP \leftarrow bp$
6: $flag[\text{DIM}(changeBP)] \leftarrow 2$
7: **end if**
8: **for** $i = 0 \rightarrow dimensions$ **do**
9: **if** $flag[i] == 2$ **then**
10: Mettre à jour les bits de la dimension i avec $\Pi_i(last) + 1$
11: **else if** $flag[i] == 0$ or $flag[i] == 1$ **then**
12: **if** saveMin[i] $\neq \infty$ and $changeBP >$ BP(i, saveMin[i]) **then**
13: Mettre tous les bits de la dimension i et de position $> changeBP$ à 0
14: **else**
15: Mettre tous les bits de la dimension i et de position $> changeBP$ au minimum de Q dans la dimension i
16: **end if**
17: **else if** $flag[i] == -1$ **then**
18: Mettre tous les bits de la dimension i au minimum de Q dans la dimension i
19: **end if**
20: **end for**

4.4 Structure d'indexage à une dimension

Dans la section précédente, nous avons vu comment projeter des points multidimensionnels sur un espace unidimensionnel discret et fini. Dans cette section, nous implémentons plusieurs algorithmes d'indexage unidimensionnel supportant une space filling curve. Nous commençons par Zarray, très satisfaisant en termes de performances mais désastreux par rapport à la gestion de la mémoire. Ce constat nous orientera vers le UB-Tree, nettement plus économique en mémoire. Cependant, la perte de performances de ce dernier ne nous convient pas et nous finirons avec le Zhash, combinant hautes performances et utilisation parcimonieuse de la mémoire.

4.4.1 Zarray

Nous construisons un vecteur (*array*) avec une Z-Value par élément. Chaque élément est une liste de points ayant cette Z-Value. Ainsi, pour une certaine Z-Value, nous savons obtenir tous les points ayant cette Z-Value. L'algorithme `GetNextZValue` permet ensuite de passer à la Z-Value suivante faisant partie de la zone de recherche. Dans le vecteur, nous allons chercher en temps constant tous les points ayant cette Z-Value, et ainsi de suite. Avec la recherche $O(1)$ du vecteur, c'est la technique la plus rapide que l'on puisse faire. À 6 dimensions et 1 million de points, le Zarray est 12 fois plus performant que le M-Tree, 26 fois plus performant que le kd-tree et 142 fois plus performant que la recherche linéaire, le M-Tree et le kd-tree étant dans leurs configurations optimales respectives.

Avec d dimensions et s divisions par dimension, il y a 2^{sd} valeurs possibles de Z-Values, et autant d'éléments à allouer dans le vecteur. Avec $s = 8$ divisions par dimension et $d = 10$ dimensions, il y a 1 milliard de Z-Values. Nous aurons donc 1 milliard de pointeurs dans notre Zarray. Un pointeur étant sur 4 octets, les 3GB de RAM de notre ordinateur de test ne suffisent pas pour allouer le Zarray. Bien qu'extrêmement performant, Zarray fait face à une explosion de mémoire. L'astuce consiste à remarquer qu'avec 5 millions de points, dans le pire des cas, il y aura 5 millions de Z-Values contenant un seul point. Cela veut dire que l'extrême majorité des éléments dans le Zarray sont vides. Cependant, même pour une Z-Value vide, nous sauvegardions un pointeur. Dans les prochaines sections, nous allons simplement implémenter des structures ne sauvegardant que les Z-Values non vides afin de gagner de l'espace mémoire.

4.4.2 UB-Tree

Nous démarrons avec un arbre binaire (*binary tree*) et allons justifier l'utilisation d'un B-Tree (c'est une généralisation du binary tree). Enfin, nous pourrons passer à un UB-Tree (Universal B-Tree) qui est un B-Tree couplé à une space-filling curve et l'algorithme GetNextZValue.

4.4.2.1 Du binary tree vers le B-Tree

Tentons d'utiliser un arbre binaire afin de permettre la recherche (unidimensionnelle). Chaque nœud possède deux fils. Un fils est choisi sur base d'une comparaison prenant place dans le nœud, et ainsi de suite. Quand la quantité de données est importante, il est courant que l'arbre ne tienne pas sur une seule mémoire. Dans ce cas, certains nœuds seront sauvegardés sur d'autres disques. Lorsqu'un tel nœud devra être consulté, il faudra aller le chercher sur le support approprié, résultant en ce que l'on appelle une *page fault*. Une *page fault* est pénalisante car en général, l'accès à un autre support de sauvegarde est un processus lent. Nous aimerions donc réduire la probabilité qu'une *page fault* survienne.

Soit l'arbre de la Figure 4.13 (c'est un arbre complet, mais il n'a pas été entièrement représenté par souci de clarté). Il est utilisé afin de sauvegarder 64 enregistrements, soit une hauteur d'arbre de $\log_2(64) = 6$. Si nous devons effectuer une descente de l'arbre, nous devrons passer par 6 nœuds, résultant en 6 *page faults* possibles. Une première idée est d'agréger les comparaisons en pages. Sur la Figure 4.13, une page s'étale sur 3 niveaux et englobe 7 nœuds. Tout le contenu d'une page est sauvegardé sur la même mémoire. Ainsi, pour descendre l'arbre, nous devons traverser 2 pages, soit 2 *page faults* possibles, ce qui constitue un résultat plus avantageux.

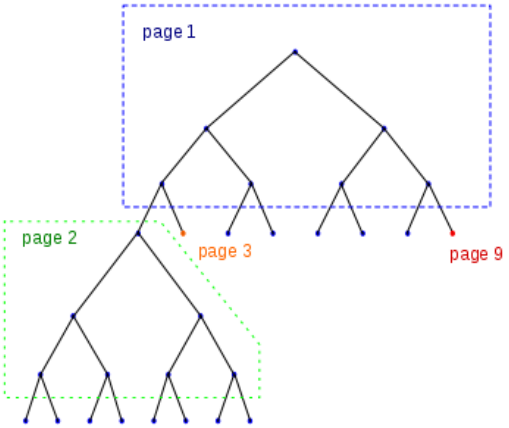


FIGURE 4.13 – Arbre binaire pour 64 éléments

Un autre moyen de minimiser la quantité de *page faults* est d'utiliser un B-Tree plutôt qu'un binary tree. Un B-Tree est la généralisation d'un binary tree où chaque nœud interne, plutôt que de tester une seule clé et avoir deux branchements possibles, va avoir m clés et $m + 1$ branchements possibles. La Figure 4.14 illustre un B-Tree à deux étages et avec maximum 3 éléments par nœud.

Rechercher un élément dans un B-Tree est une opération très aisée. Le procédé est analogue à la recherche dans un binary tree, sauf que nous devons effectuer une recherche à l'intérieur de chaque nœud parcouru.

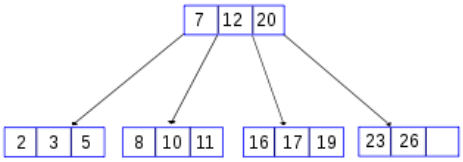


FIGURE 4.14 – Exemple de B-Tree

4.4.2.2 Insertion dans un B-Tree

La procédure d'insertion du B-Tree rend ce dernier globalement balancé. Si nous tentons d'insérer un élément dans un nœud complet, ce dernier doit être divisé en deux (un nouveau nœud apparaît) ; c'est le *page splitting*. Il se peut que le parent doit également être splitté, et ainsi de suite jusqu'à la racine. Il y a $\log_2(n)$ parents dans la hiérarchie et une insertion est donc réalisée en temps $O(\log_2(n))$.

Algorithm 4.5 Insertion dans un B-Tree

Ensure:
Élément inséré et B-Tree globalement balancé

1: **if** nœud n'est pas complet **then**
2: Insérer le nouvel élément dans le nœud, tout en conservant l'ordre des éléments dans le nœud.
3: **else if** nœud est complet **then**
4: L'élément médian est choisi parmi les feuilles et le nouvel élément
5: Les valeurs inférieures au médian sont placées dans le nouveau fils à gauche, tandis que les valeurs supérieures au médian sont placées dans le nouveau fils à droite.
6: Insérer le médian dans le nœud parent, ce qui peut causer un split de ce dernier (s'il était déjà complet). On parle alors de *back-splitting*. S'il n'y a pas de parent (i.e., le nœud était la racine), créer une nouvelle racine au-dessus (ce qui incrémente la hauteur de l'arbre).
7: **end if**

Dans l'algorithme 4.5, nous avons choisi l'élément médian comme séparateur. Il est important de comprendre que ce n'est pas obligatoire de choisir le médian. Si nous avons 9 éléments, il est simplement plus intelligent de choisir le 5^{e}comme séparateur, laissant 4 éléments dans le nouveau fils à gauche et 4 éléments dans le nouveau fils à droite. Cependant, rien ne nous empêche de choisir le 2^{e} ou le 3^{e} élément comme séparateur. Cela constitue le choix du *page separator*. Pour l'instant, nous avons le médian comme *page separator*, mais nous verrons dans la suite de ce travail (Section 4.4.2.4) que nous allons concevoir un autre *page separator* intéressant pour notre application.

Un exemple de construction du B-Tree est présenté Figure 4.15.

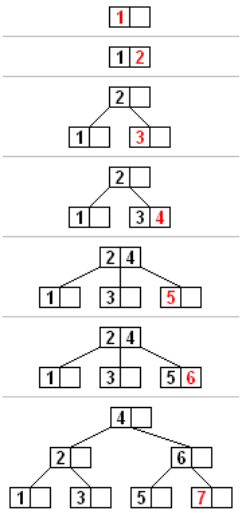


FIGURE 4.15 – Insertion dans un B-Tree

4.4.2.3 UB-Tree : B-Tree et space-filling curve

Le UB-Tree (Universal B-Tree) est une généralisation multidimensionnelle du B-Tree utilisant une *space-filling curve*. Reprenons le tout premier exemple de *space-filling curve* que nous avons vu, Figure 4.16. C'est une Z-Curve. La zone de recherche Q est en jaune. La recherche devra donc nous donner les points

$$\{12, 13, 14, 15, 24, 25, 26, 27, 36, 37, 38, 39, 48, 49, 50, 51\}$$

Nous allons au préalable diviser la Z-Curve en plusieurs Z-Regions. Pour rappel, une séquence de Z-values consécutives $[\alpha; \beta]$ est une Z-Region. Sur la Figure 4.16, nous découpons en quatre Z-Regions selon les lignes bleues, à savoir $[0; 15]$, $[16; 31]$, $[32; 47]$ et $[48; 63]$. Dans un B-Tree, à chaque feuille correspond une Z-Region.

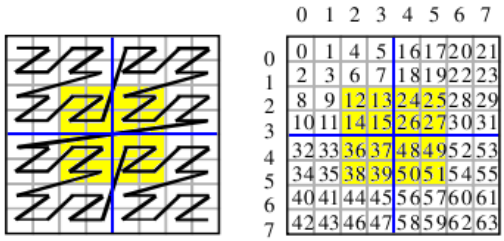


FIGURE 4.16 – Découpages en quatre Z-Regions

Le B-Tree correspondant à l'exemple sera donc celui présenté sur la Figure 4.17.

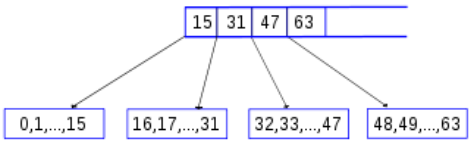


FIGURE 4.17 – B-Tree correspondant

Cet arbre, appelé UB-Tree, permet, à partir d'une Z-Value, de trouver la Z-Region qui la contient.

Dès lors, comment combiner le UB-Tree avec l'algorithme `GetNextZValue` ? `GetNextZValue` nous donne la prochaine Z-Value intersectant Q. Le UB-Tree permet ensuite de trouver la Z-Region contenant la Z-Value donnée. Nous itérons sur tous les points faisant partie de cette Z-Region et ne gardons que ceux intersectant Q. Ensuite, nous prenons la dernière Z-Value de la Z-Region, l'incrémentons, et utilisons `GetNextZValue` pour obtenir la prochaine Z-Value intersectant Q, et ainsi de suite... Formellement, nous implémentons l'algorithme 4.6.

Algorithm 4.6 Range Query dans un UB-Tree

Require:
start : Plus faible Z-Value dans Q (facile à obtenir)
end : Plus grande Z-Value dans Q (facile à obtenir)
Ensure:
Renvoie tous les points intersectant Q

1: $current = start$
2: **while** $current \leq end$ **do**
3: $region$ = getRegionContaining($current$) {region contenant $current$, obtenue grâce au B-Tree, en temps $\log_2(n)$}
4: **for all** Z-Value z in $region$ **do**
5: **if** $z \in Q$ **then**
6: Ajouter aux résultats les points ayant la Z-Value z
7: **end if**
8: **end for**
9: $regionAddress$ = getLastZValueInRegion($region$) {trivial}
10: $current$ = getNextZValue($regionAddress + 1$) {en temps linéaire par rapport au nombre de bits dans la Z-Value}
11: **end while**

4.4.2.4 Page splitting

Comment automatiser le page splitting dans le UB-Tree, c'est-à-dire la délimitation des Z-Regions ? Lorsque nous insérons des Z-Values dans l'arbre, des splits se produiront.

Précédemment, nous avions expliqué que nous pouvions choisir l'élément médian comme *page separator*. Désormais, il semble plus avantageux de splitter de telle façon que nous conservions des régions les plus rectangulaires possible (les tests pratiques confirmeront ce choix).

4.4.3 Zhash

Comme espéré, le UB-Tree a résolu les problèmes d'occupation de mémoire apparus avec Zarray. Cependant, même si le UB-Tree reste nettement meilleur que le M-Tree ou le kd-tree, UB-Tree est au moins 3 fois plus lent que Zarray. En effet, avec le UB-Tree, des descentes d'arbre doivent être effectuées, ce qui rajoute un facteur $\log(n)$ à la recherche. Certes, c'est logarithmique et GetNextZValue est linéaire. Cependant, GetNextZValue est linéaire par rapport au *nombre de bits*. Avec 3 dimensions et 8 divisions par dimension, il n'y a que $3 * 8 = 24$ bits, donc GetNextZValue est extrêmement rapide car il y a très peu de bits ! Nous pouvons même considérer que cet algorithme est presque instantané. Par rapport à ces performances, le facteur logarithmique devient prépondérant. Afin de pallier ce souci, nous allons utiliser une space-filling curve dans un vecteur de hachage, que nous appelons Zhash.

Nous avons simplement réimplémenté un vecteur de hachage auto-extensible, où le *haché* est la Z-Value elle-même. Ainsi, nous n'y sauvegardons que les Z-Values non vides afin d'économiser la mémoire, tout en conservant presque les mêmes performances que Zarray. Zhash est donc notre meilleure méthode d'indexage hyperdimensionnel.

Chapitre 5

Système de gestion de base de données

Dans la première section de ce chapitre, nous allons étudier l'énorme besoin en données afin de pouvoir faire fonctionner l'algorithme de région de confiance. Ensuite, nous considérerons des moyens pour sauvegarder nos 5 millions de points. De plus, nous allons voir que ce choix ne se limite pas uniquement à la sauvegarde de données. En effet, nous envisageons également la possibilité d'intégrer les méthodes de recherche spatiale dans les moyens de sauvegarde de données.

5.1 Propriétés des données

Un système de gestion de base de données est un ensemble de logiciels informatiques permettant de manipuler des bases de données. Notre application a des besoins spécifiques afin d'être prise en main adéquatement.

Grande quantité de lignes et de colonnes Il était sensé y avoir 5 millions de points et 200 variables. Le SGBD doit pouvoir subvenir à ces besoins.

Taille de la base de données Avec un format d'enregistrement *double* sur 8 octets, la quantité de données s'élève à $8 * 200 * 5000000 = 8\text{GB}$

État quasi statique *A priori*, la base de données est statique. Nous devons y insérer d'emblée 5 millions de points, et puis nous ne devons plus en ajouter. S'il fallait quand même en insérer par après, ce serait au compte-gouttes selon les résultats obtenus lors des tests effectués sur l'unité de production. Nous ne devons jamais mettre à jour ou supprimer une ligne.

Relationnel inutile Le modèle relationnel n'est pas nécessaire, car nous avons simplement besoin d'une table pouvant contenir un grand nombre de lignes et de colonnes.

5.2 MySQL

MySQL est l'un des SGBD le plus utilisé au monde (Google, Yahoo !, YouTube, Adobe, Airbus, AFP, Reuters, BBC News...) et il fonctionne sur de nombreux systèmes d'exploitation (AIX, FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Solaris, SunOS, Windows 2000, Windows XP, Windows Vista et Windows 7...). De nombreux langages possèdent une API pour accéder aux données : VB, VB .NET, C/C++, C♯, Java, Delphi, Perl, PHP, Python, Ruby, Tcl... MySQL est un logiciel libre développé sous double licence : GPL (pour les particuliers) et propriétaire (pour un usage commercial).

Avec MySQL, nous devons choisir un moteur de stockage. C'est ce qui sauvegarde, gère et récupère de l'information dans une table. Chaque moteur de stockage a des avantages et des inconvénients, et il faut en choisir un adapté à l'application. Les deux plus connus sont MyISAM et InnoDB, mais il en existe deux douzaines d'autres.

5.2.1 MyISAM

C'est le moteur de stockage par défaut. Les données sont séparées entre trois fichiers sur le disque : un pour le format de la table, un pour les données et le dernier pour les index.

La quantité maximale de lignes dans une table est d'environ $4.295\,10^9$ et il peut y avoir jusque 64 champs d'index.

5.2.2 InnoDB

InnoDB est plus récent. Il protège les transactions, ce qui signifie que l'intégrité des données est assurée durant l'entièreté d'une requête. Par exemple, il peut verrouiller une ligne pendant sa mise à jour, ce qui signifie qu'un autre processus peut mettre à jour d'autres lignes au même moment, offrant une vraie expérience multi-utilisateurs.

Le second point fort d'InnoDB est sa gestion des clés étrangères. Une contrainte de clé étrangère assure l'intégrité référentielle entre une ligne d'une table et une ligne d'une autre table. Par exemple, un vol (en avion) doit référencer le pilote, qui doit exister. Ce mécanisme permet d'assurer une certaine validité de l'ensemble des données dans la base de données.

Cependant, les tables sont plus lourdes et les opérations plus lentes.

5.2.3 Quantité de colonnes

La quantité maximale de colonnes dans une table est de 4096, mais selon plusieurs critères, cette limite pourrait encore être plus faible.

– Chaque table a une taille maximale de 65535 octets par ligne, quel que soit le moteur de stockage. La quantité de colonnes permise dépend donc du type des colonnes. Nous utilisons des *double* (8 octets) et il y a donc $\frac{65535}{8} \approx 8100$ colonnes au maximum. Cela ne permet pas pour autant d'aller au-delà de 4096 colonnes ; quoiqu'il arrive, la limite de 4096 ne peut être dépassée.

– À chaque table est associé un fichier qui en décrit la structure. Ce fichier ne peut dépasser 64KB. Ainsi, si de longs noms de colonnes sont utilisés, la quantité permise de colonnes peut être plus réduite.
– InnoDB ne permet pas plus de 1000 colonnes.

Parmi MyISAM et InnoDB, quel choix est le plus approprié ?

1. Un seul utilisateur accède à la base de données (pas de concurrence).
2. Le mécanisme de clés étrangères nous est inutile, vu que nous avons une seule table.
3. MyISAM est plus léger et plus rapide qu'InnoDB.

Notre choix se porte donc très clairement vers MySQL.

Le *modus operandi* sera donc, au lancement de l'application, d'importer dans l'application la totalité des données à partir de la base de données, et de construire (dans l'application) la structure d'index (kd-tree, quadtree, M-Tree, UB-Tree, Zarray ou Zhash).

5.3 PostgreSQL

PostgreSQL est un SGBD relationnel-objet publié sous licence BSD, permettant de développer des applications commerciales. Cela contraste avec MySQL. En effet, la licence libre GPL de MySQL impose que l'application finale le soit aussi. Si tel n'est pas le cas, une licence payante de MySQL est nécessaire. PostgreSQL est principalement utilisé par Yahoo !, MySpace, Sony Online et Skype.

PostgreSQL présente plusieurs avantages par rapport à MySQL. En effet, PostgreSQL

1. respecte nettement mieux la norme SQL2003 ;
2. est plus lent sur les faibles volumes de données car PostgreSQL gère l'intégrité référentielle par défaut, alors que MySQL ne le fait pas avec le moteur de stockage MyISAM ;
3. possède une plus grande panoplie d'outils : règles, types personnalisés, tableaux, langages procéduraux (Java, Python, PHP...) ;
4. accepte des extensions pour enrichir le système, dont notamment la cartouche PostGIS (PostgreSQL Geographic Information System).

5.3.1 Un SGBD pour la recherche spatiale

Heureusement, nous n'avons pas choisi PostgreSQL comme énième comparateur de performances en lecture. Ce qui nous a attiré, c'est son modèle objet-relationnel et l'utilisation de ce dernier afin de spatialiser notre table. Plus clairement, nous avons besoin de recherches de proximité. Grâce au modèle objet-relationnel, nous allons pouvoir utiliser des « points » au sens orienté objet du terme, et les employer en tant que vrais

types de données dans la table. Cela nous permettra de rechercher tous les points dans une certaine région de recherche. De plus, PostgreSQL implémente certaines méthodes d'indexage spatial que nous avons vues dans la Section 4. Ainsi, ce n'est plus l'application qui doit implémenter l'indexage spatial, c'est le SGBD qui le fait de façon presque transparente !

Ce n'est pas le seul avantage. Si c'est l'application qui doit maintenir l'index et les données, cela signifie qu'elles sont toutes en mémoire vive. Or, il y a énormément de données (un total de 64GB environ) qui ne pourront pas tenir entièrement dans la mémoire vive d'un ordinateur. Grâce à la spatialisation des tables, toutes ces données pourront rester dans la base de données, et nous récupérerons uniquement les données à l'intérieur de la région de recherche. La mémoire de l'ordinateur est donc nettement épargnée, et une partie importante du travail (la recherche de points) est déléguée au SGBD.

Malheureusement, ce modèle ne permet que 3 dimensions au maximum, vu qu'il est à l'origine destiné aux applications spatiales, c'est-à-dire des routes, des ponts, des pays, des lieux etc., soit seulement deux ou trois dimensions. Nous pourrons donc juste nous en servir pour comparer les performances à deux ou trois dimensions.

5.3.2 Modèle objet-relationnel

Dans le cours « Bases de données » de Monsieur Wolper, nous avons étudié l'utilité et les fonctionnalités des bases de données orientées objet, et nous avons entr'apercu leur réalisation technique. Bien évidemment, il est nécessaire de traduire le modèle objet en un modèle purement relationnel.

L'OGC (Open Geospatial Consortium) est une organisation à but non lucratif, fondée en 1994, regroupant de nombreuses compagnies privées actives en géomatique, des agences publiques et représentants du monde académique. Son rôle est de promouvoir l'usage des données spatiales dans les technologies de l'information, en facilitant le travail des professionnels. Elle favorise l'interopérabilité en définissant des standards en matière de données spatiales, de leur traitement et de leur échange. L'OGC a défini des standards pour les propriétés des entités spatiales, et a notamment défini un schéma de spatialisation pour le modèle objet-relationnel ([Ope99]), Figure 5.1. (Seules les entités en bleu sont instanciables, celles en vert sont abstraites.)

PostgreSQL est purement relationnel et ne connaît pas le modèle objet-relationnel. [Sto03] nous précise que c'est son extension PostGIS qui implémente le modèle de l'OGC afin de permettre la spatialisation des tables.

5.3.3 SQL/MM et recherche spatiale avec PostGIS

Aux nouveaux objets disponibles, sont associées des fonctions permettant de les traiter et de les mettre en relation. Quelques exemples des possiblités du modèle-objet et des fonctions spatiales :

- gérer des entités abstraites ;
- conserver une référence vers un objet dans une autre table ;

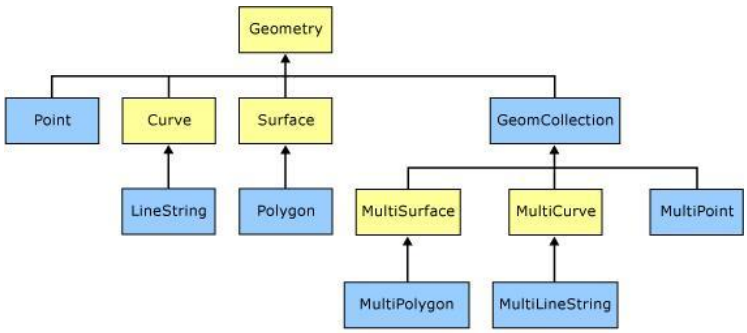


FIGURE 5.1 – Hiérarchie des classes selon l'OGC

- héritage ;
- listes, vecteurs ;
- calculer l'enveloppe convexe d'un polygone ;
- jointures spatiales (intersection, union...) ;
- calculs spatiaux (longueur, aire, centroïde ...).

Ce sont tous des outils que nous devons pouvoir manipuler. Pour ce faire, nous utiliserons une extension du SQL : la norme SQL/MM (Structured Query Language for MultiMedia). SQL/MM est divisé en plusieurs parties ([Sto03]) :

- Partie 1 — Framework : définitions communes aux autres parties.
- Partie 2 — Full-Text : permet d'effectuer des opérations avancées sur du texte.
- Partie 3 — Spatial : manipulation des données spatiales.
- Partie 4 — Images : traitement d'images statiques et dynamiques.
- Partie 5 — Data mining : fouille de données pour la prise de décision.

5.3.3.1 Création de table

Le Listing 5.1 montre la création d'une table.

```
CREATE TABLE mypoints (
  x geometry,
  y double precision,
  CONSTRAINT enforce_dims CHECK (ndims(x) = 2),
  CONSTRAINT enforce_geotype CHECK (geometrytype(x) = 'POINT'::text),
  CONSTRAINT enforce_srid CHECK (srid(x) = 0)
);
CREATE INDEX mypoints_x_gist ON mypoints USING rtree(x);
```

Listing 5.1 – Création d'une table

Modifications remarquables par rapport à une table non spatiale :

- `ndims(x) = 2` impose deux dimensions (nous pouvons choisir deux ou trois dimensions seulement).

– `geometrytype(x) = 'POINT'::text` signale que le type géométrique de x est un point.
– `srid(x) = 0` PostGIS établit une codification des systèmes de références géographiques terrestres. Nos points étant dans un référentiel absolu, nous désactivons la fonctionnalité en attribuant « 0 ».
– Avec `USING rtree(x)`, nous choisissons le R-Tree comme indexage spatial. Nous avons brièvement introduit le R-Tree dans la Section 4.2.6.2 : il utilise un partitionnement à base de rectangles, alors que le M-Tree emploie des sphères. PostGIS ne propose pas le M-Tree.

5.3.3.2 Recherche en table

Nous allons utiliser le SQL/MM pour accomplir la recherche spatiale proprement dite. Admettons que la région de recherche Q est un cercle de centre $c = (5, 6)$ et de rayon $r = 2$. Un point exprimé textuellement est converti en objet avec `ST_GeometryFromText('POINT(5 6)', 0)`, où 0 est le SRID (référentiel) nul. Plusieurs requêtes possibles s'offrent à nous (Listing 5.2) :

```
SELECT ST_AsText(x) AS x2, y FROM mypoints WHERE
    distance(x, ST_GeometryFromText('POINT(5 6)', 0)) < 2;
SELECT ST_AsText(x) AS x2, y FROM mypoints WHERE
    ST_DWithin(x, ST_GeometryFromText('POINT(5 6)', 0), 2);
```

Listing 5.2 – Possibilités de sélection

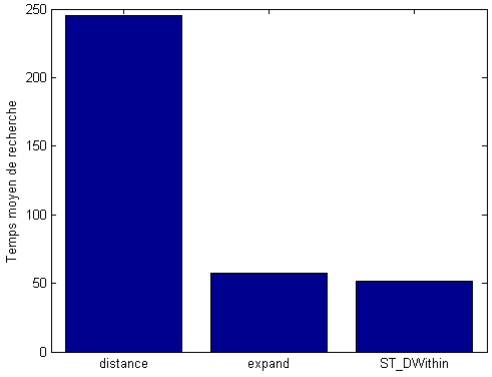


FIGURE 5.2 – Fonctions de recherche de proximité

Les requêtes `distance` et `ST_DWithin` fournissent le même résultat, mais la Figure 5.2 nous montre qu'elles ont des performances très différentes. La fonction `distance` effectue une simple recherche exhaustive sans se servir de l'index. Malheureusement, nous ne savons pas utiliser directement l'index pour cette recherche car le R-Tree utilise des rectangles. Or, nous recherchons des points dans un cercle. Heureusement, l'index peut

tout de même être utilisé. Il suffit de construire la bounding box rectangulaire autour du cercle Q. Cette bounding box a une longueur de côté valant $2r$ et a le même centre que celui du cercle. Nous pouvons dès lors utiliser l'index pour obtenir tous les points contenus dans la bounding box. À partir des points obtenus, une recherche exhaustive est appliquée pour ne garder que les points dans le cercle (Listing 5.3).

```
SELECT ST_AsText(x) AS x2, y FROM mypoints WHERE
    x && expand(ST_GeometryFromText('POINT(5 6)', 0), 2) AND
    distance(x, ST_GeometryFromText('POINT(5 6)', 0)) < 2;
```

Listing 5.3 – Bounding box

`ST_DWithin` est une fonction tout à fait équivalente à ce qui est accompli dans le Listing 5.3, c'est-à-dire qui filtre automatiquement par la bounding box avant d'effectuer les comparaisons de distances (Listing 5.4).

```
SELECT ST_AsText(x) AS x2, y FROM mypoints WHERE
    ST_DWithin(x, ST_GeometryFromText('POINT(5 6)', 0), 2);
```

Listing 5.4 – Filtrage automatique par la bounding box

5.4 SGBD et space-filling curves

Avec PostGIS, nous venons de voir l'utilisation d'une méthode de partitionnement au sein même d'un SGBD. Malheureusement, la spatialisation fournie ne permet d'avoir que des points à 2 ou 3 dimensions seulement. Cependant, nous nous rappelons également que dans la Section 4.3, nous avons développé une méthode de projection vers un espace unidimensionnel, qui peut être prise en charge par un index unidimensionnel (et nous avons implémenté des indexages unidimensionnels dans la Section 4.4). Or, nous savons bien évidemment que les SGBD classiques fournissent de tels index. MySQL et PostgreSQL implémentent tous les deux le B-Tree.

5.4.1 Intégrer la Z-Value dans un SGBD

La façon la plus aisée d'intégrer une Z-Curve dans un SGBD est de créer une colonne « zvalue » associant à chaque point sa Z-Value. Cette colonne doit bien évidemment être indexée afin de rendre la recherche rapide. Dans la pratique, le SGBD utilise un B-Tree. Nous avons donc créé cette colonne dans une table MySQL. L'application utilise son implémentation de GetNextZValue pour calculer le prochain point d'intersection avec la zone de recherche, et puis effectue une requête auprès du SGBD afin d'obtenir tous les points correspondant à cette Z-Value, et ainsi de suite. Les avantages sont

- les données et la structure d'indexage restent dans la base de données ; il n'est plus nécessaire de les charger en mémoire vive (8GB de données) ;
- il est possible d'implémenter `getNextZValue` via une procédure stockée, afin de complètement décharger l'application de la recherche spatiale.

5.4.2 Intégrer le UB-Tree dans un SGBD

[RVF+00] introduit l'idée de l'intégration du UB-Tree dans un SGBD. Dans cette section, nous discutons l'idée bien que nous ne l'avons pas mise en pratique.

Chaque feuille du UB-Tree correspond à une Z-Region. Lorsque nous l'implémentions dans notre application, nous devions contrôler le *page splitting* (Section 4.4.2.4) afin de découper intelligemment les régions. Le UB-Tree était capable de nous renvoyer la Z-Region contenant une Z-Value recherchée. Cependant, avec le B-Tree d'un SGBD,

- nous ne pouvons pas choisir le nombre maximum d'éléments par nœud ;
- nous ne pouvons pas changer l'algorithme de *page splitting* et nous aurons donc des Z-Regions découpées de façon sous-optimale ;
- lorsque nous avons une Z-Value appartenant à la zone de recherche, nous ne pouvons obtenir tous les éléments de la Z-Region contenant la Z-Value recherchée.

Afin de contourner ces problèmes, [RVF+00] propose d'avoir une table avec un tuple par feuille du UB-Tree, donc un tuple par Z-Region souhaitée, ce qui nous oblige à gérer nous-mêmes ces feuilles — en somme, à implémenter le UB-Tree dans l'application pour ensuite convertir chaque feuille en un tuple, qui sera finalement sauvegardé dans la table. Ensuite, l'indexage B-Tree du SGBD pourra nous fournir la Z-Region complète à laquelle appartient une Z-Value recherchée. Cette procédure devrait fonctionner, bien que lourde car nous devons quand même implémenter un UB-Tree dans l'application pour générer les feuilles (Z-Regions) qui deviendront des tuples dans le SGBD. Des problèmes persistent :

- il faudra quand même charger l'ensemble des données (8GB) dans l'application afin de générer les feuilles, ce qui pose un problème de quantité de mémoire vive.
- Les Z-Regions ne seront plus modifiables s'il fallait ajouter *a posteriori* des Z-Values (si nous insérons par après de nouveaux points de données).

Chapitre 6

Résultats

Au cours des précédents chapitres qui ont exposé les notions théoriques implémentées, nous avons, çi et là, avancé quelques résultats afin de justifier nos choix. Dans ce chapitre, nous exposons de façon plus détaillée les résultats, et comparons les paramètres majeurs qui sont riches d'enseignements.

6.1 Région de confiance

Nous avons effectué la plupart de nos tests sur un problème non convexe et non linéaire, illustré sur la Figure 6.1, ses courbes de niveau étant représentées sur la Figure 6.2. Cette fonction est intéressante car il y a un minimum très large, un de taille moyenne, et deux petits. Nous commencerons souvent l'algorithme au point $(5, 5)$ car les deux petits minima (qui sont bien entendu des minima inintéressants) se dressent entre le point de départ et les minima larges. Nous aimerions bien sûr trouver les gros minima, ou du moins l'un d'entre eux.

Durant ce mémoire, nous avons beaucoup parlé du rayon Δ_k de la région de confiance B_k (à l'itération k). Nous avons expliqué la notion de région de confiance, c'est-à-dire une région dont le volume traduit la confiance que nous avons dans le modèle m_k à être capable de bien modéliser la fonction f. La région de confiance est donc mise à jour selon l'évolution de notre confiance. Dans notre exemple, où nous souhaitons converger vers les gros minima, nous serions tentés de penser que la valeur initiale de la région de confiance a de l'importance. De même, l'ensemble P_k des points à l'intérieur de la sphère Q, qui est utilisé pour contruire le modèle m_k, devrait avoir un rôle utile. Nous supposons donc qu'en prenant Δ_k et $r(Q)$ initialement grands, nous aurons une bonne modélisation du comportement lointain de f, afin de ne pas converger vers les petits minima proches de $(5, 5)$. Les tests ont révélé que la valeur initiale de Δ_k et de $r(Q)$ n'est pas vraiment importante, car nous convergeons très souvent vers le plus petit minimum.

Le rayon $r(Q)$ a peu d'influence sur le déroulement global de l'algorithme de région de confiance. Il est intuitivement plus intéressant de prendre $r(Q) = \beta\Delta_k$, où β vaut environ 1.5, ceci afin d'obtenir des points légèrement au-delà de B_k.

Afin d'avoir une certaine assurance d'atteindre de bons minima, nous avons utilisé

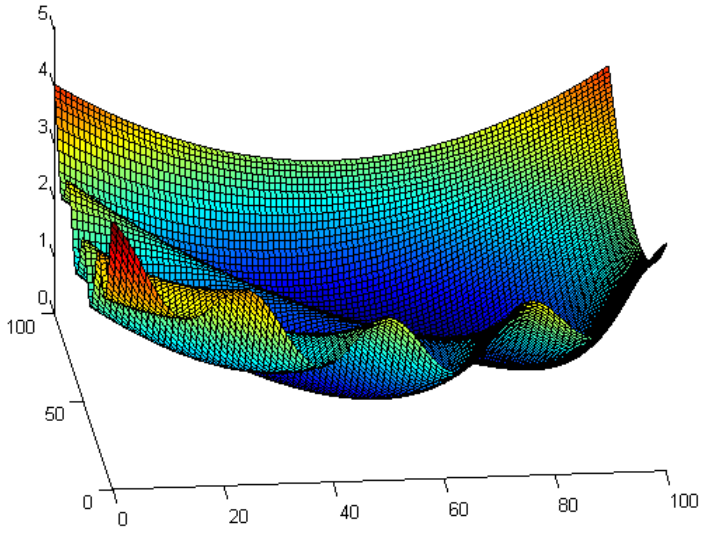


FIGURE 6.1 – Fonction non convexe non linéaire à minimiser

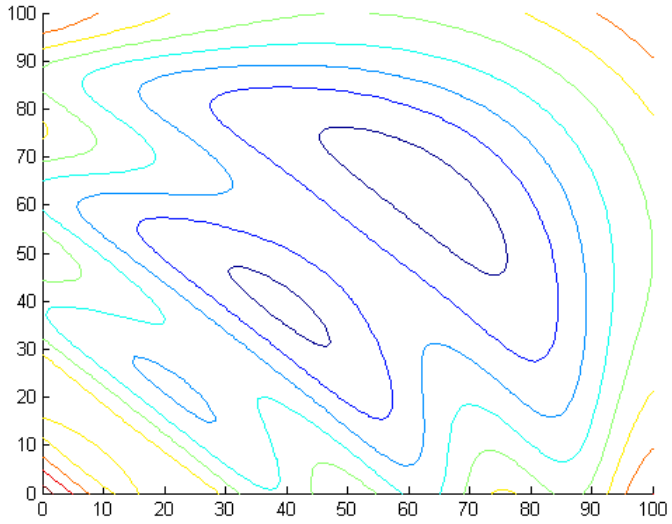
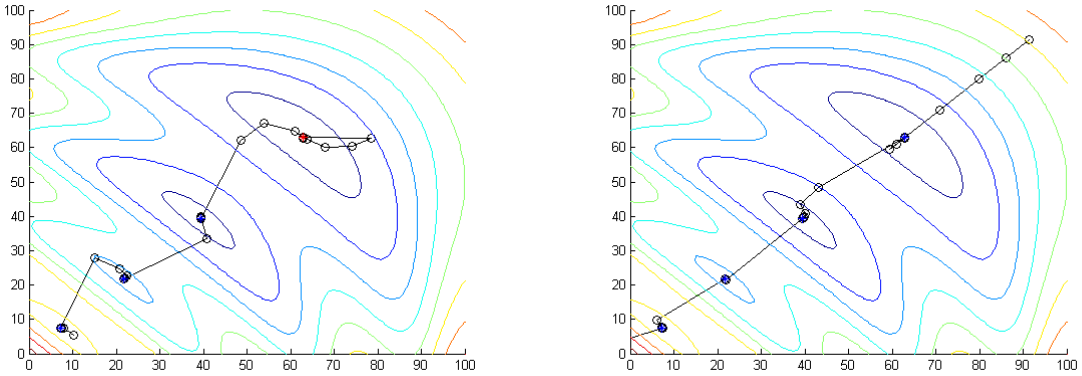


FIGURE 6.2 – Courbes de niveau de la fonction



(a) En partant en bas à gauche

(b) En partant en haut à droite

FIGURE 6.3 – Trust Region avec saut de longueur constante

une technique de saut afin d'échapper à un minimum et ainsi nous donner la possibilité de converger vers d'autres minima. Dans un premier temps, nous avions une longueur de saut constante. Figure 6.2 (a), la technique fonctionne très bien vu que nous convergeons vers le plus gros minimum (les minima reconnus comme locaux sont en bleu). Cependant, Figure 6.2 (b), nous avons démarré l'algorithme en haut à droite (donc près du gros minimum) ce qui semble avantageux. Malheureusement, l'algorithme échoue et va converger vers le petit minimum. Cela est dû au fait que la longueur du saut était un peu trop grande.

Ensuite, nous avons inventé une méthode à longueur de saut automatiquement adaptable (Figure 6.4). Nous ne devons plus nous soucier de la longueur de saut, et l'algorithme nous donne un score par minimum exploré (tableau 6.1), ce qui quantifie la qualité d'un minimum. C'était en effet un objectif majeur de ce mémoire que d'avoir de bons minima accompagnés d'une estimation de leur qualité.

Minimum	Score
$(62.97; 62.99)$	36.97
$(39.34; 39.34)$	12.38
$(21.79; 21.78)$	7.16
$(7.33; 7.32)$	4.97

TABLE 6.1 – Minima et scores respectifs

Une difficulté majeure dans l'application de la méthode à région de confiance résidait dans la méconnaissance de f et de son gradient g. L'expression de f devait permettre d'évaluer la confiance ρ que nous pouvons avoir dans les modèles, et g permet de détecter un minimum local quand nous en rencontrons un. Concernant ρ, nous effectuons une

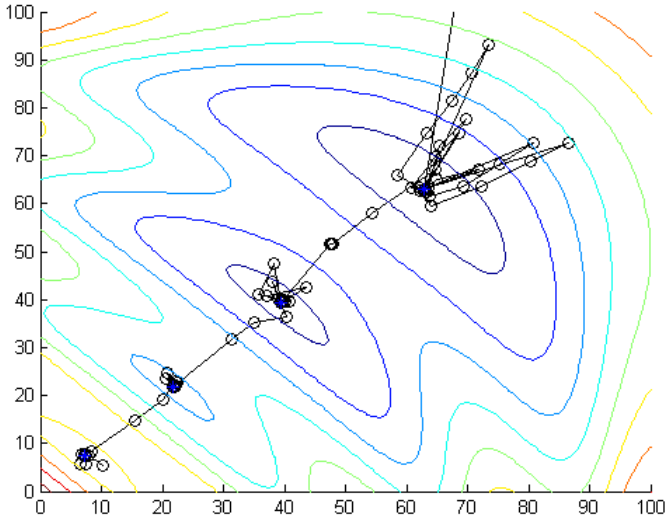


FIGURE 6.4 – Méthode à longueur de saut automatiquement adaptable

régression autour de x_k et une autre autour de x_{k+1}, pour remplacer respectivement $f(x_k)$ et $f(x_{k+1})$. Dans les deux cas, les points sélectionnés pour ces régressions sont très proches afin d'avoir une bonne estimation locale. Cela nous donne un bon résultat sur l'algorithme global. Nous avons bien entendu fait varier le rayon utilisé pour les deux régressions : tant qu'il reste petit, sa valeur précise importe peu.

Par contre, la reconnaissance d'un minimum local est très importante. Nous pourrions penser qu'il n'est pas nécessaire d'avoir un minimum local précis. Avec notre algorithme de longueur de saut automatiquement adaptable, c'est complètement faux. En effet, si nous sommes laxistes sur la reconnaissance d'un minimum local, de nombreux points sont considérés comme minima locaux et comme ils sont éloignés, l'algorithme pensera qu'ils sont distincts. Il n'augmentera donc pas la longueur de saut, et l'algorithme peut ne pas parvenir à quitter un minimum. C'est ce qui se produit sur la Figure 6.5. De plus, il finit même par ressortir du grand minimum et traverser de nouveau d'anciens minima ! Comme les faux minima ne sont pas proches, l'algorithme ne se rend même pas compte qu'il explore des minima déjà rencontrés.

6.2 Sous-problème de région de confiance

Dans cette section, nous nous intéressons aux solveurs du sous-problème de région de confiance. Nous avons implémenté

minimize L'algorithme 3.2 page 34 est le premier que nous avons vu. Il applique Newton récursivement. Il détecte grossièrement les cas difficiles. Si sa résolution du cas

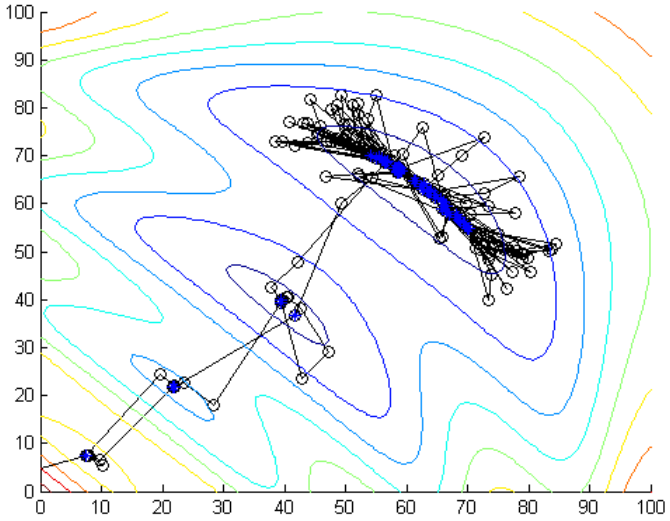


FIGURE 6.5 – Estimation peu précise des minima locaux

difficile conduit à une convergence, il s'arrête ; sinon, l'algorithme continue.

minimizeDecompozed Modification de `minimize` avec décomposition Householder en une matrice tridiagonale, plus rapide à factoriser.

minimizeEigen Approximation de la plus petite valeur propre pour avoir une détection plus précise du cas difficile. Le cas difficile est résolu et l'algorithme s'arrête. Si l'optimum est intérieur, l'algorithme s'arrête après une itération. Dans les autres cas (convergence à la frontière), l'algorithme itère avec Newton.

SteihaugToint Gradients conjugués adaptés au cas non convexe.

SteihaugTointDiag `SteihaugToint` avec préconditionneur de Jacobi.

Il n'est pas directement évident de comparer les performances de ces méthodes, car leurs critères de convergence sont très différents. De plus, ce ne sont que des solveurs de sous-problèmes dans l'algorithme du trust region. Nous pouvons seulement donner une tendance.

Cependant, nous pouvons en comparer certains entre eux. `minimizeDecompozed` est une optimisation de `minimize`. Des tests sur plusieurs centaines de sous-problèmes ont révélé que `minimize` est environ 7% plus rapide que `minimizeDecompozed`. Nous avons expliqué que `minimizeDecompozed` est plus rapide que `minimize` à partir de 4 itérations

du sous-problème. En pratique, nous avons calculé que la moyenne du nombre d'itérations est de 2.19, donc `minimizeDecompozed` est légèrement plus lent.

Avec `minimizeEigen`, nous avons comptabilisé dans le tableau 6.2 le nombre de cas difficiles et de solutions intérieures (dans ces deux cas, arrêt en une itération), et la quantité de solutions à la frontière (qui nécessitent plusieurs itérations avec Newton) :

Scénario	Fréquence
cas difficile	3%
solution intérieure	14%
solution frontière	83%

TABLE 6.2 – Répartition des scénarios avec l'eigensolveur

Nous observons donc que généralement, la solution est à la frontière de la région de confiance, nécessitant plusieurs itérations de Newton afin d'être calculée. Heureusement, le cas difficile n'est pas trop fréquent, mais il n'est pas non plus question de le négliger.

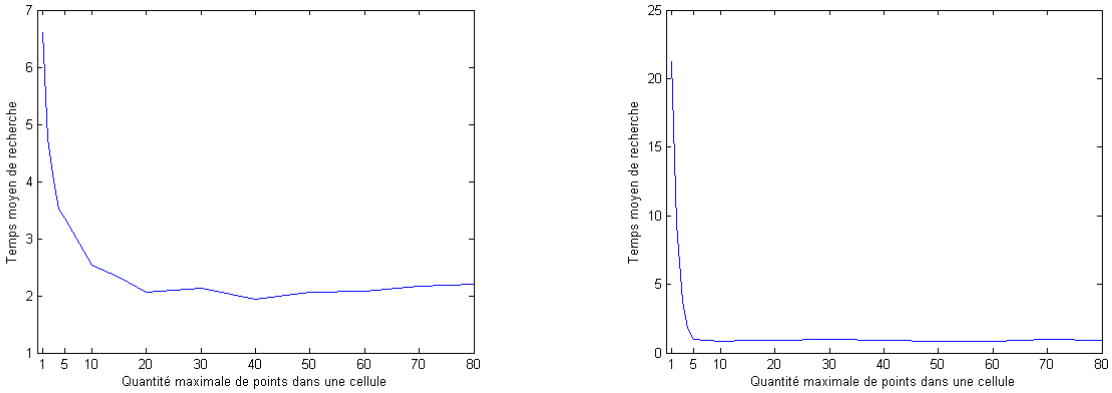
Enfin, nous n'avons pas ressenti un vrai avantage à utiliser la résolution approximative de `SteihaugToint`. En effet, notre méthode à longueur de saut automatiquement adaptable a besoin de minima précis, et une résolution approximative n'est pas particulièrement appropriée. Cependant, nous avons mesuré que `SteihaugTointDiag` (donc avec un préconditionneur de Jacobi) est 30% plus rapide que sans préconditionneur.

6.3 Recherche spatiale

Nous allons comparer quelques paramètres importants des différentes techniques d'indexation spatiale que nous avons implémentées, et nous finirons en comparant ces techniques dans leurs configurations optimales respectives.

Nous avons d'abord implémenté le kd-tree et le quadtree, des techniques assez simples. Ils divisent tous les deux en sous-espaces de manière récursive, jusqu'à ce que chaque sous-espace contienne au maximum un seul point. Nous avons donc placé une capacité plus grande sur les nœuds, afin de ne pas découper jusqu'à ce qu'il ne reste plus qu'un seul nœud. La Figure 6.6 (a) montre que si on accepte quelques points dans un nœud, le gain en performance est très important. De plus, dans l'arbre utilisé pour représenter le kd-tree, nous économisons facilement 40% des nœuds, ce qui constitue également une économie importante de mémoire. Le quadtree, bien que nettement moins performant, affiche le même comportement par rapport à ce paramètre (Figure 6.6 (b)).

Le M-Tree a également un paramètre avec les mêmes effets : c'est la capacité maximale de ses sphères. Nous la fixons à environ 40. Figure 6.7, nous comparons tous les algorithmes de clustering que nous avons implémentés (cf. 43). Parmi les algorithmes les moins compliqués (`mM_RAD`, `m_RAD`, `M_LB_DIST` et `overlapping`), c'est `mM_RAD` qui est le meilleur (*minimize the Maximum RADii*). Les trois algorithmes les plus lourds (`stddev`, `Kmeans` et `SK_means`) sont nettement meilleurs et leurs performances sont similaires.



(a) Temps du kd-tree

(b) Temps du quadtree

FIGURE 6.6 – Partitionnement de l'espace

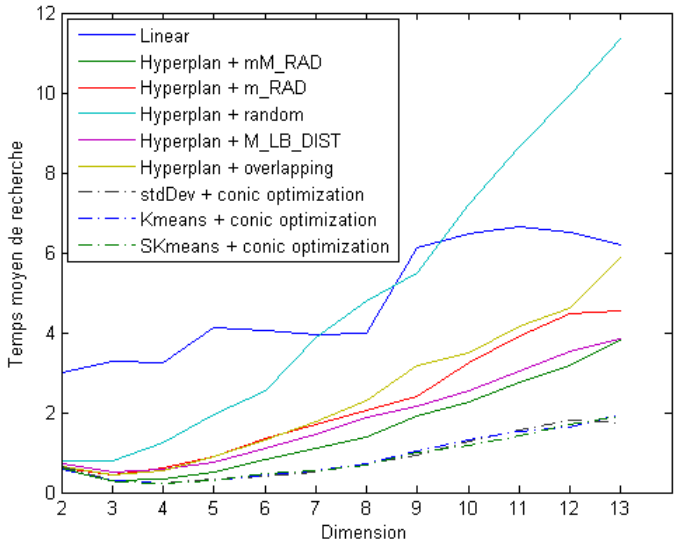


FIGURE 6.7 – Algorithmes de clustering du M-Tree

Ensuite, nous avons implémenté le Zarray, le UB-Tree et le Zhash qui se basent tous les trois sur les space-filling curves. Les trois algorithmes ont donc des propriétés communes. Reprenons la Figure 6.8. Dans cet exemple, il y a 4 divisions par dimension. Avec deux dimensions, nous avons donc $2^4 = 16$ cellules. Le nombre de divisions par dimension est crucial car s'il y en a trop peu, la recherche va dégénérer en une recherche exhaustive. S'il y en a trop, l'espace sera trop morcelé et l'algorithme `GetNextZValue` sera surexploité. La Figure 6.9 illustre ce compromis. Nous découvrons qu'il est préférable d'avoir une quantité faible de divisions par dimension.

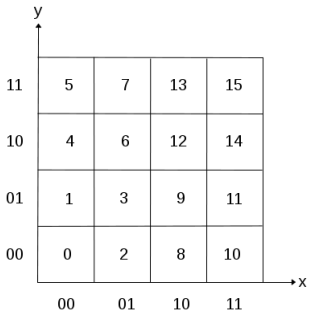


FIGURE 6.8 – N-Curve à 4 divisions par dimension, soit 16 éléments au total

Nous avons expliqué que la mémoire utilisée par Zarray croît de façon exponentielle avec le nombre de divisions et de dimensions. Sur notre ordinateur de test à 3GB, nous ne pouvons pas aller au-delà de 9 dimensions. Afin de régler ce problème, nous avons implémenté le UB-Tree qui se base sur le B-Tree. Lors de la procédure de split d'un nœud de l'arbre, le B-Tree sépare selon l'élément médian. Dans le UB-Tree, nous avions plutôt conseillé de séparer de façon à favoriser des Z-Regions rectangulaires. Un test révèle que le gain en performances s'élève à 40%. Cependant, le UB-Tree était sensiblement moins performant, ce qui nous avait conduit à implémenter le Zarray afin de bénéficier des meilleures performances combinées à une utilisation rationnelle de la mémoire. Désormais, avec Zhash, notre seule limitation fut l'adressage 32 bits de l'ordinateur de test. Avec Zhash, il y a un paramètre intéressant : la taille du vecteur de hachage (*bucket size*). Une taille raisonnablement grande est nécessaire afin de limiter le nombre de collisions dans les valeurs hachées. La Figure 6.10 montre cet effet. Dans cet exemple, avec 1 million de points, nous avons intérêt à prendre un vecteur d'au moins 1050000 éléments.

Finalement, ayant les paramètres optimaux de toutes les méthodes, nous pouvons effectuer le comparatif général (Figure 6.11). Nous remarquons tout d'abord que toutes nos méthodes apportent un vrai gain par rapport à la recherche exhaustive, justifiant notre étude de la recherche spatiale. Le M-Tree est deux fois plus rapide que le kd-tree. Le point le plus important est que les structures basées sur une space-filling curve (Zarray, UB-Tree et Zhash) ont une allure en forme de bol : mauvaises performances à basses et hautes dimensions, et résultats excellents aux alentours d'une certaine dimension. Nous

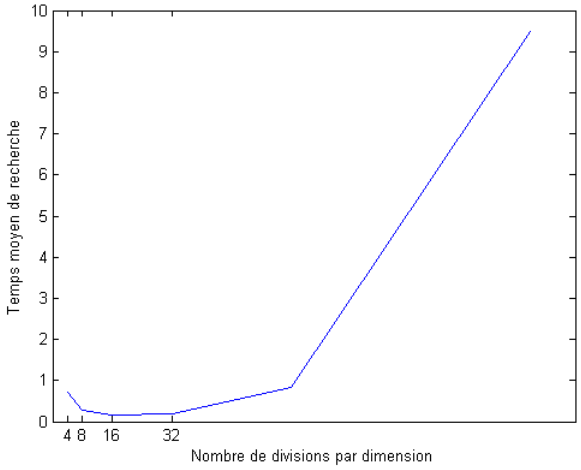


FIGURE 6.9 – Influence du nombre de divisions par dimension, avec 4 dimensions

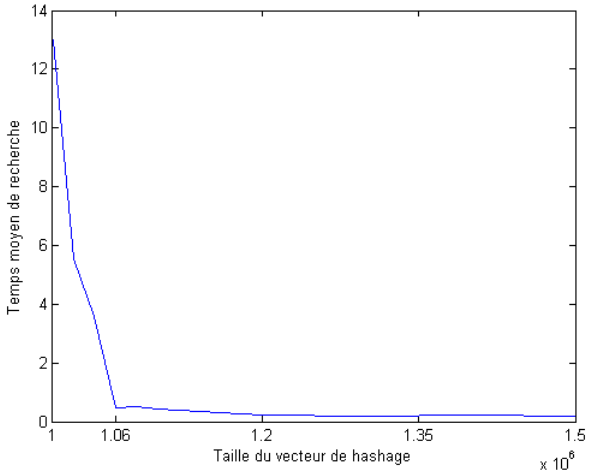


FIGURE 6.10 – Taille du vecteur de hachage, avec 8 dimensions et 1 millions de points

observons sur la Figure 6.12 que l'optimum se situe à 6 dimensions pour le Zarray, UB-Tree et Zhash avec 8 divisions par dimension. Le Zhash à 4 divisions par cellule, lui, a un bol déplacé vers la droite. En fait, nous choisissons le nombre de divisions par dimension selon les dimensions auxquelles nous souhaitons des performances optimisées. Réduire le nombre de divisions par dimension permet de monter dans les dimensions. Ainsi, ces méthodes peuvent fournir des performances très importantes sur une grande gamme de dimensions.

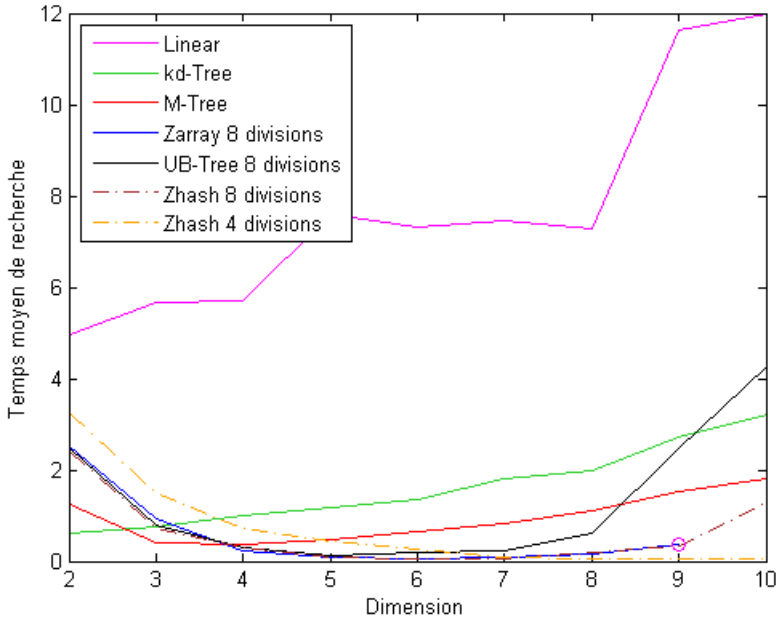


FIGURE 6.11 – Comparaison de toutes les techniques avec 1 millions de points

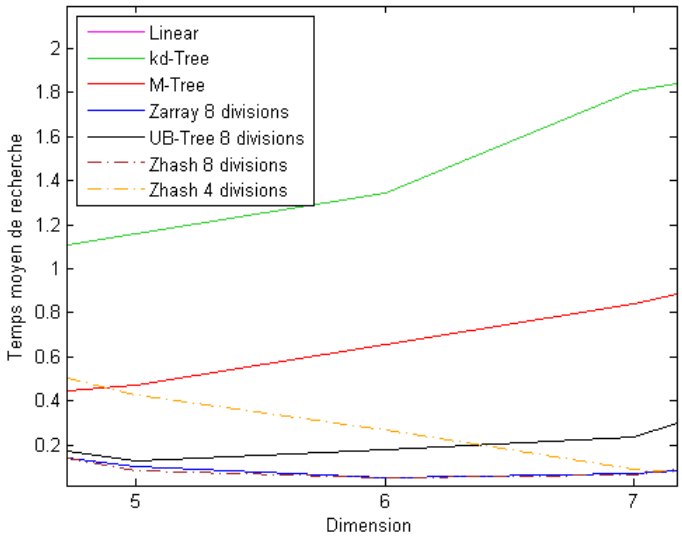


FIGURE 6.12 – Zoom à 6 dimensions

Chapitre 7

Conclusion

L'optimisation de procédés industriels complexes permet de mettre sur le marché des produits de meilleure qualité et d'économiser beaucoup d'argent. Cependant, la résolution mathématique est très difficile.

Dans ce travail, nous avons étudié la recherche de zones de fonctionnement optimal dans une fonction non convexe et non linéaire, connue uniquement au travers de nombreuses expériences. Nous savons concevoir un modèle local de la fonction afin de l'utiliser dans l'algorithme de région de confiance, qui a été adapté pour fonctionner même si la fonction objectif est inconnue.

Les méthodes développées dans ce travail ne s'appliquent pas uniquement au problème de galvanisation. Il concerne bien évidemment les autres entreprises de galvanisation, mais également celles souhaitant optimiser d'autres procédés dans lesquels elles ont une très bonne expertise, mais aimeraient trouver des conjonctions de paramètres optimisant un objectif.

La recherche efficace de points dans un espace à hautes dimensions représentait une difficulté majeure, car la littérature scientifique nous a mis en garde contre la décroissance exponentielle des performances à hautes dimensions. Nous avons inventé le Zhash qui s'est révélé très efficace. Cette technique n'est pas uniquement limitée au problème qui a motivé ce mémoire ; elle peut être utilisée dans de nombreux domaines.

Cependant, la méthode de région de confiance est inadaptée à l'optimisation à hautes dimensions. En effet, nous avons mis en avant la dispersion des points dans l'espace, à mesure que le nombre de dimensions augmente. Cela signifie qu'à hautes dimensions, si nous recherchons tous les points contenus dans une région (même large), nous ne trouverons aucun point, rendant impossible la modélisation locale d'une fonction. Ce n'est pas un souci de performances ; c'est une limite à l'applicabilité de l'optimisation par région de confiance. Afin de pouvoir l'utiliser, il faut réduire les 200 dimensions pour n'en garder que 6, environ. Dès lors, nous imaginons mal comment une solution au problème réduit pourrait être significative dans le problème original.

Ainsi, d'autres techniques devraient être considérées afin de tenter d'optimiser ce procédé industriel.

Bibliographie

[BF97] Richard L. Burden and Douglas Faires. *Numerical Analysis*. Brooks/Cole Publishing, Pacific Grove, CA, USA, 6th edition, 1997.

[CGT00] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust-Region methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Che] Mei-Qin Chen. Inverse Power Method, Shifted Power Method and Deflation.

[CK07] Ward Cheney and David R. Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole Publishing, Pacific Grove, CA, USA, 6th edition, 2007.

[CZP97] Paolo Ciaccia, Pavel Zezula, and Marco Patella. M-tree : An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[FTS02] Yongjian Fu, Jui-Che Teng, and S. R. Subramanya. Node splitting algorithms in tree-structured high-dimensional indexes for similarity search. *Proceedings of the 2002 ACM symposium on Applied computing - SAC '02*, 2002.

[KKI03] George Em Karniadakis and Robert M. Kirby II. *Parallel Scientific Computing in C++ and MPI. A seamless approach to parallel algorithms and their implementation*. Cambridge University Press, 2003.

[LGMR05] Paul A. Longley, Michael F. Goodchild, David J. Maguire, and David W. Rhind. *Geographical Information Systems : Principles, Techniques, Management and Applications*, volume 1. Wiley, 2nd edition, 2005.

[Ope99] OpenGIS Consortium. OpenGIS Simple Features Specification for SQL, Revision 1.1. 1999.

[Pru07] Antonín Prukl. A relational approach to indexing. Master's thesis, Charles University Prague, 2007.

[RVF+00] Frank Ramsak, Markl Volker, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 263–272, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[Seg11] Segal. Antiroulle n°90. July 2011.

[Sto03] Knut Stolze. SQL / MM Spatial : The Standard to Manage Spatial Data in Relational Database Systems. In *BTW*, volume 26 of *LNI*, pages 247–264. GI, 2003.

[WSB98] Roger Weber, Hans-Jörg. Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases*, VLDB '98, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.