UNIVERSITÉ DE MONTRÉAL

UN ENVIRONNEMENT POUR L'OPTIMISATION SANS DÉRIVÉES

DOUNIA LAKHMIRI
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(MATHÉMATIQUES APPLIQUÉES)
JUIN 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

UN ENVIRONNEMENT POUR L'OPTIMISATION SANS DÉRIVÉES

présenté par : LAKHMIRI Dounia
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. LE DIGABEL Sébastien, Ph. D., président
M. ORBAN Dominique, Doctorat, membre et directeur de recherche
M. AUDET Charles, Ph. D., membre

REMERCIEMENTS

Je voudrais commencer par remercier mon directeur de recherche, M. Dominique Orban, pour toute l'aide qu'il m'a apportée, ses précieux conseils et généralement pour toute l'expérience et savoir que j'ai pu acquérir sous sa tutelle durant mes deux années de maîtrise.

Merci ensuite à mes parents pour leur soutien continu malgré les moments difficiles et leurs encouragements à aller de l'avant. À ma sœur Maha pour son aide, ses conseils et son attitude positive. Je vous suis entièrement reconnaissante et vous dédie ce mémoire.

Finalement, à tous les amis que je me suis faits depuis mon arrivée à Montréal, je vous remercie pour avoir fait de ces deux années une expérience unique et mémorable.

RÉSUMÉ

L'optimisation sans dérivées (DFO) est une branche particulière de l'optimisation qui étudie les problèmes pour lesquels les dérivées de l'objectif et/ou des contraintes ne sont pas disponibles. Généralement issues des simulations, les fonctions traitées peuvent être coûteuses à évaluer que ce soit en temps d'exécution ou en mémoire, bruitées, non dérivables ou simplement pas accessibles pour des raisons de confidentialité. La DFO spécifie des algorithmes qui se basent sur divers concepts dont certains ont été spécialement conçus pour traiter ce type de fonctions. On parle aussi parfois de boîtes grises ou noires pour souligner le peu d'information disponible sur la fonction objectif et/ou les contraintes.

Il existe plusieurs solveurs et boîtes à outils qui permettent de traiter les problèmes de DFO. Le but de ce mémoire est de présenter un environnement entièrement développé en Python qui regroupe quelques outils et modules utiles dans un contexte de DFO, ainsi que quelques solveurs. Cet environnement a la particularité d'être écrit de façon modulaire. Cela permet une liberté à l'utilisateur en termes de manipulation, personnalisation et développement d'algorithmes de DFO.

Dans ce mémoire, on présente la structure générale de la bibliothèque fournie, nommée `LIBDFO.py`, ainsi que les détails des solveurs implémentés, en précisant les parties qui peuvent être modifiées et les différentes options disponibles. Une étude comparative est aussi présentée, le cas échéant, afin de mettre en évidence l'effet des choix des options utilisées sur l'efficacité de chaque solveur. Ces comparaisons sont visualisées à l'aide de profils de performance et de données que nous avons aussi implémentés dans un module indépendant nommé `Profiles.py`.

Mots clés : Optimisation sans dérivées, optimisation de boîte noire, Python, profils de performance, profils de données.

ABSTRACT

Derivative free optimization (DFO) is a branch of optimization that aims to study problems for which the derivatives of the objective function and/or constraints are not available. These functions generally come from simulation problems, therefore calling a function to evaluate a certain point can be expensive in terms of execution time or memory. The functions can be noisy, non diffrentiable or simply not accessible. DFO algorithms use different concepts and tools to adjust to these special circumstances in order to provide the best solution possible. Sometimes, the terms grey box or black box optimisation can also be used to emphasize the lack of information given about the objective function.

There is a rich literature of solvers and toolboxes specialized in DFO problems. Our goal is to provide a Python environment called `LIBDFO.py`, that regroups certain tools and modules that can be used in a DFO framework. The modular implementation of this environment is meant to allow a certain freedom in terms of modifying and customizing solvers.

In this document, we present the general structure of `LIBDFO.py` as well as the implementation details of each solver provided. These solvers can be modified by changing certain options that are mentioned in their respective sections. We also provide a benchmark study to compare the results obtained with each version of the same solver. This benchmark is done using performance and data profiles, which are part of an independent module called `Profiles.py` also presented in this document.

Key words: Derivative free optimization, black box optimization, performance profile, data profile, Python.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT v

TABLE DES MATIÈRES vi

LISTE DES TABLEAUX viii

LISTE DES FIGURES ix

LISTE DES SIGLES ET ABRÉVIATIONS x

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.2 Éléments de la problématique 2
1.3 Objectif de recherche 3
1.4 Plan du mémoire 4

CHAPITRE 2 REVUE DE LITTÉRATURE 5
2.1 Méthodes pour l'optimisation sans dérivées 5
2.2 Solveurs 9

CHAPITRE 3 LIBRAIRIE PYTHON 15
3.1 Pourquoi Python ? 15
3.2 Structure de la bibliothèque 15

CHAPITRE 4 MÉTHODES IMPLÉMENTÉES 27
4.1 Nelder et Mead 27
4.1.1 Description générale 27
4.1.2 Pseudo-code 27
4.1.3 Tests numériques 30
4.2 UOBYQA 31
4.2.1 Description générale 31
4.2.2 Initialisation 33

4.2.3 Résolution du pas de région de confiance 37
4.2.4 Résolution du pas de modèle 43
4.2.5 Calcul du paramètre t 48
4.2.6 Mises à jour 49
4.2.7 Tests numériques 50
4.3 MADS 52
4.3.1 Description générale 52
4.3.2 Pseudo-code 53
4.3.3 Tests numériques 58

CHAPITRE 5 CONCLUSION 59
5.1 Synthèse des travaux 59
5.2 Améliorations futures 59

RÉFÉRENCES 61

LISTE DES TABLEAUX

Tableau 2.1 Liste de solveurs en DFO. 10
Tableau 2.2 Liste des boîtes à outils en DFO. 13
Tableau 3.1 Problèmes tests. 25
Tableau 4.1 Validation du solveur Nelder et Mead. 30
Tableau 4.2 Validation de UOBYQA. 51
Tableau 4.3 Validation de MADS. 58

LISTE DES FIGURES

Figure 3.1 Exemple de profils de performance. 24
Figure 3.2 Exemple de profils de données. 24
Figure 4.1 Réflexion et expansion du simplexe. 28
Figure 4.2 Contraction externe et interne du simplexe. 29
Figure 4.3 Réduction du simplexe. 29
Figure 4.4 Organigramme d'UOBYQA. 32
Figure 4.5 G_Q définie positive. 39
Figure 4.6 Cas facile. 41
Figure 4.7 Cas difficile. 42
Figure 4.8 Profils de performance et de données pour $\tau = 10^{-3}$. 47
Figure 4.9 Profils de performance et de données pour $\tau = 10^{-5}$. 47
Figure 4.10 Profils de performance et de données pour $\tau = 10^{-7}$. 48
Figure 4.11 Exemple d'exécution de la recherche par coordonnées. 54
Figure 4.12 Exemple d'exécution de GPS. 55
Figure 4.13 Exemple d'exécution de MADS. 56
Figure 4.14 Comparaison entre les directions de recherche de `MADS.py`. 57

LISTE DES SIGLES ET ABRÉVIATIONS

x_i	i^{e} élément du vecteur x.
A_{ij}	l'élément de la matrice A qui se trouve à la i^{e} ligne et j^{e} colonne.
$\mathcal{B}(x, r)$	la boule ouverte de centre x et de rayon r.
DFO	Derivative Free Optimization.
UOBYQA	Unconstrained Optimization BY Quadratic Approximation.
MADS	Mesh Adaptive Direct Search.

CHAPITRE 1 INTRODUCTION

1.1 Définitions et concepts de base

On parle d'optimisation sans dérivées pour désigner cette branche de l'optimisation numérique qui ne se sert des dérivées ni pour caractériser un optimum local, ni comme outil de recherche de solutions comme c'est le des méthodes de gradient par exemple. On parle aussi d'optimisation de boîtes noires dans le cas où on cherche à résoudre un problème dont la formulation analytique n'est pas fournie explicitement mais plutôt sous forme d'exécutable ou sous forme de modèle basé sur une simulation. Ces deux types de problèmes sont traités par une classe d'algorithmes qui ne se servent pas des dérivées de l'objectif ou celles des contraintes (Audet & Kokkolaras, 2016). Notons que pour alléger les notations par la suite, nous allons désigner ces deux types de problèmes par l'acronyme «DFO».

Il est très important de noter que la DFO n'est pas une concurrente de l'optimisation avec dérivées dont les méthodes doivent normalement être préférées tant que cela est possible, étant donnée leur plus grande efficacité au niveau de leur robustesse, de leur rapidité, de la garantie de l'optimalité des solutions trouvées et de leur capacité à résoudre des problèmes de grande taille. La DFO doit être vue plutôt comme une solution de recours dans le cas où les autres approches ne peuvent être appliquées. L'optimisation sans dérivées est en fait venue répondre à des besoins réels qui ont évolué en fonction de l'avancement scientifique de chaque époque. En effet, dans le livre *Introduction to derivative free optimization* (Conn, Scheinberg et Vicente, 2008, page 1), on explique que pour remédier au problème du calcul des dérivées, principale source d'erreur dans l'optimisation avec dérivées jusqu'à récemment, on pouvait avoir recours aux méthodes de la DFO. Aujourd'hui, la DFO est surtout utilisée dans les cas particuliers où la fonction objectif ne peut pas être traitée par les techniques d'optimisation avec dérivées. C'est le cas, par exemple, des problèmes issus de modélisation informatique basés sur la simulation. On se retrouve aussi confronté à des cas où les dérivées peuvent ne pas exister, être coûteuses à calculer ou biaisées par du bruit. Il est possible que le domaine de définition de la fonction objectif soit tellement fragmenté qu'on ne puisse pas utiliser les méthodes d'optimisation avec dérivées, que l'appel à la fonction soit trop coûteux en temps et en ressources ou encore que cette fonction soit uniquement disponible sous forme d'exécutable, ce qui rend l'étude de ses propriétés difficile. On parle alors d'optimisation de boîte noire.

Ces contraintes peuvent paraître difficiles à surpasser puisqu'elles obligent à se passer de

l'information contenue dans les dérivées et donc de toutes les techniques d'optimisation avec dérivées dont les algorithmes se montrent robustes, rapides et efficaces mais surtout capables de résoudre des problèmes de très grande taille. C'est justement cela qui fait de l'optimisation sans dérivées une des branches les plus ambitieuses de l'optimisation numérique et ce qui explique aussi l'engouement général et l'intérêt nouveau que lui porte la communauté scientifique jusqu'à présent. Une simple recherche sur Google Scholar montre que le nombre d'articles scientifiques publiés sur le sujet au cours des 10 dernières années est de 3510 contre un peu plus de 600 auparavant.

Elle est aussi particulièrement attrayante étant donnés les problèmes réels qu'elle permet de résoudre, issus de domaines tels que la médecine (Marsden et al., 2008), la chimie (Morgan & Deming, 1974), la physique (S. M. Wild, Sarich, & Schunck, 2015), la recherche opérationnelle (A. R. Conn, Scheinberg, & Toint, 1998a) et beaucoup d'autres domaines de l'ingénierie. L'optimisation sans dérivées s'applique à une grande panoplie de problèmes et les algorithmes disponibles deviennent de plus en plus efficaces et capables de résoudre des problèmes dont la taille grandit avec les capacités de calcul des ordinateurs.

1.2 Éléments de la problématique

Depuis l'apparition du domaine de la DFO, plusieurs méthodes et solveurs ont été développés. Ces méthodes se basent sur des principes très variés que nous allons détailler par la suite mais que l'on peut classer dès maintenant en deux familles principales distinctes, à savoir les méthodes de recherche directe et celles se basant sur des modèles.

Les solveurs sont souvent présentés sous forme de logiciels écrits dans des langages de programmation différents. Ce sont souvent des outils tout prêts que l'on utilise directement pour résoudre un certain problème en particulier, sans chercher forcément à rentrer dans les détails de leur implémentation ou des stratégies adoptées pour chaque étape de la résolution. Le fait de manipuler ces outils fermés, dans le sens où il est difficile de modifier une partie de l'algorithme, limite grandement l'exploitation que l'on pourrait en faire si on avait la liberté de créer des variantes de la même méthode. Les choix qu'ont fait leurs auteurs, même si justifiés à une certaine époque pour un certain type de problème en particulier, ne sont souvent pas les seuls choix acceptables possibles, d'autant plus que l'avancement scientifique pourrait aussi justifier la modification de certaines approches et leurs substitutions par des techniques plus modernes.

D'autre part, le fait que ces solveurs aient été développés en utilisant des langages de programmation différents rend la tâche plus difficile lorsqu'on cherche à faire une étude comparative approfondie entre eux. Il existe cependant quelques logiciels destinés à la comparaison d'algorithmes qu'on peut appliquer dans le cadre de la DFO, en utilisant des outils tels que les profils de performance ou de données (J. J. Moré & Wild, 2009).

De manière générale, les solveurs destinés à l'optimisation numérique sont des travaux au code relativement opaque, ce qui limite la recherche dans le domaine, tant par la difficulté éprouvée pour les examiner de près que par l'absence d'une plate-forme commune pour les manipuler et/ou les comparer entre eux.

1.3 Objectif de recherche

Le but de ce travail est tout d'abord de créer une bibliothèque écrite entièrement dans un langage de programmation adéquat, qui pourra réunir quelques méthodes de DFO principales d'une part, mais surtout qui pourra permettre la manipulation des algorithmes de manière plus propice au développement de variantes personnalisées. Cette idée de plate-forme commune devrait se traduire par un outil simple d'utilisation, dans lequel on devrait pouvoir implémenter un algorithme de DFO sans trop de difficulté.

Loin d'être un simple recueil de solveurs, la principale particularité de cette bibliothèque est son idée d'implémentation modulaire. Cela veut dire que chaque algorithme devra être divisé en parties indépendantes (tant que cela est possible) pour que chacune puisse être isolée du reste, ce qui permettra par la suite toute modification ou personnalisation jugée nécessaire. En effet, si l'on veut adopter une nouvelle approche pendant une étape en particulier, il suffira de déconnecter ladite partie et de brancher une fonction implémentant la nouvelle stratégie désirée. Aussi, grâce à cette bibliothèque, on pourra utiliser une technique autrefois spécifique à un seul solveur pour la tester avec d'autres méthodes différentes, tout cela dans le but d'ouvrir encore plus de possibilités de création ou de personnalisation des méthodes pour la DFO.

Cette bibliothèque devra aussi posséder les outils nécessaires de comparaisons tels que les profils de performance et de données (J. J. Moré & Wild, 2009) ainsi qu'une sélection de problèmes test classiques afin de mieux étudier et évaluer la performance de chaque méthode.

1.4 Plan du mémoire

Pour atteindre nos objectifs, ce mémoire est structuré de la manière suivante. Le chapitre 2 donne une revue de littérature concernant les principaux solveurs existants pour l'optimisation sans dérivées et leurs principes de résolution respectifs. Le chapitre 3 explique le choix du langage Python pour ce projet et détaille la structure générale de la bibliothèque présentée ainsi que les techniques d'implémentation utilisées. Dans le chapitre 4, on expose les détails des solveurs déjà implémentés, en expliquant, le cas échéant, les modifications apportées par rapport à leur implémentation usuelle. On s'attardera surtout sur la méthode UOBYQA (M. Powell, 2001) en raison de son importance en DFO et du nombre d'étapes qui constituent l'algorithme et des différentes approches que l'on peut prendre pour résoudre chacune d'entre elles. On présente aussi les méthodes de Nelder & Mead et de Hooke & Jeeves qui peuvent servir de base pour développer des algorithmes plus sophistiqués comme celui de MADS (Audet & Dennis, Jr., 2006). Dans ce chapitre, on présente aussi un module utilisé pour produire des profils de performance et de données qu'on utilise pour comparer l'efficacité de solveurs entre eux. Finalement, on conclut par une discussion sur les limitations de cette bibliothèque et sur les améliorations qui pourront lui être apportées par la suite.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans ce mémoire, on s'intéresse à résoudre le problème :

$$\min_{x \in \mathbb{R}^n} f(x), \tag{2.1}$$

où $f : \mathbb{R}^n \to \mathbb{R}$ est une fonction dont les dérivées ne sont pas accessibles. De plus, on suppose que son évaluation est coûteuse. Il faut donc voir ces évaluations comme une ressource à préserver au maximum. On s'intéresse uniquement aux problèmes sans contraintes.

La littérature concernant ce sujet étant assez riche, il nous est impossible de couvrir ici tous les aspects existants de la DFO. C'est pourquoi nous allons tout d'abord procéder à un bref survol de l'évolution des méthodes existantes avant de nous attaquer à l'angle qui nous intéresse le plus : les solveurs disponibles jusqu'à présent.

2.1 Méthodes pour l'optimisation sans dérivées

Comme on vient de le mentionner, il existe une multitude de méthodes destinées à la DFO, elles peuvent généralement être divisées en deux catégories majeures, à savoir les méthodes de recherche directe et celles qui se basent sur des modèles. Par la suite, nous allons expliquer les caractéristiques de chacune de ces catégories tout en donnant quelques exemples d'algorithmes appartenant à chaque classe.

Les méthodes de la DFO ont beaucoup évolué depuis leur apparition, et si à présent on a des algorithmes élaborés parfois accompagnés d'une théorie de convergence, les premières méthodes, dites « historiques », sont plutôt des heuristiques qui ont l'avantage d'être relativement simples à comprendre et à appliquer. Celles que l'on considère généralement comme les premières sont la méthode de recherche par motifs (Hooke & Jeeves, 1961) et la recherche par coordonnées (Fermi & Metropolis, 1952). Ces méthodes consistent à choisir un point de départ x_0 et un pas $\Delta > 0$. À partir de ce point de départ, on suit à tour de rôle une direction parmi un ensemble prédéfini jusqu'à ce qu'on trouve un nouveau point qui améliore la valeur courante de l'objectif. On se déplace vers ce nouveau point et on reprend le même processus. Si aucune direction ne mène à une amélioration, on diminue la valeur du pas et on recommence la même technique jusqu'à ce que la valeur de Δ soit inférieure à une certaine

tolérance ou qu'on ait atteint le nombre d'évaluations de la fonction.

L'autre méthode emblématique de l'époque est celle de Nelder & Mead (1965) connue aussi sous le nom de « l'autre méthode du simplexe » qui fut très populaire dans plusieurs domaines de recherche (Marsden et al., 2008; Morgan & Deming, 1974) et qui continue d'ailleurs à être très largement appliquée en raison de sa facilité d'utilisation. La méthode de Nelder et Mead a besoin d'un simplexe à l'initialisation, c'est-à-dire un ensemble de $n+1$ points, lorsqu'on se place dans $\mathbb{R}^n$, affinement libres. Cela veut dire qu'aucun point n'appartient au sous espace affine engendré par les n autres points. À chaque étape, on commence par ordonner les points dans l'ordre croissant des valeurs de l'objectif. Ensuite, on calcule le barycentre $x_c = \frac{1}{n}\sum_{i=0}^{n-1} x_i$ des n premiers points, puis on trouve la réflexion x_r du point x_n par rapport au barycentre. Par la suite, on effectue des comparaisons entre les valeurs de $f(x_0), f(x_r)$ et $f(x_n)$ pour savoir si on applique une réflexion, une expansion, une contraction ou une réduction au simplexe. Ces transformations font qu'en pratique, le simplexe suit les courbes de niveaux de la fonction objectif à partir d'un certain nombre d'itérations. Par la suite, beaucoup de travaux sont venus améliorer les résultats de convergence de ce type de méthode. Cela s'est fait notamment en utilisant le principe des bases positives comme c'est expliqué dans les articles de Wen-Ci (1979) et Torczon (1997).

Ces méthodes dites « classiques » ont ensuite été généralisées par une classe portant le nom de méthodes de recherche directe où à chaque itération, on détermine un ensemble de points pour évaluer directement la fonction objectif et garder celui qui a la plus petite valeur. Les méthodes de recherche directe ont connu beaucoup d'alternatives sans que leur principe de base ne change réellement. De manière générale, on se place toujours en un point de départ sur un maillage dont la taille évolue. La recherche se fait en deux étapes : la recherche globale puis la sonde locale. L'étape de recherche sert à explorer des zones éloignées du point courant et ce afin d'échapper aux minima locaux. Si cette étape n'arrive pas à améliorer la valeur courante de la fonction, on procède à la sonde, qui va suivre des directions de recherche issues d'une base positive prédéfinie. C'est cette approche qu'on retrouve dans l'algorithme GPS (Torczon, 1997) par exemple. Dans GPS, la recherche est optionnelle. En ignorant cette étape on retrouve un algorithme qui se rapproche de la méthode de recherche par motifs de Hooke & Jeeves (1961), à la différence des directions de recherche suivies lors de la sonde. On retrouve cependant quelques implémentations qui choisissent de définir une stratégie pour l'étape de recherche comme par exemple l'utilisation des fonctions substitut [1] (Booker et al., 1999).

1. Surrogate

La famille d'algorithmes MADS (Audet & Dennis, Jr., 2006), qui fait aussi partie des méthodes de recherche directe, vient généraliser l'approche de GPS au niveau de l'étape de sonde. En plus du paramètre Δ_m du maillage, MADS a aussi un paramètre $\Delta_p = n\sqrt{\Delta_m}$ qui délimite la zone accessible par l'étape de sonde. Ainsi, plus la taille de Δ_p diminue, plus il y a de directions que l'on peut suivre. L'ensemble des directions de sonde possibles normées devient en réalité dense dans la sphère unité à mesure que le maillage s'affine. L'étape de recherche quant à elle est tout aussi optionnelle et flexible dans MADS que dans GPS.

Parallèlement aux méthodes de recherche directe, on retrouve une autre classe importante qui est celle des méthodes basées sur des modèles. Comme son nom l'indique, cette classe regroupe des algorithmes qui, au lieu d'optimiser directement la fonction objectif, commencent d'abord par construire un modèle qui sera plus facile à manipuler et moins coûteux à évaluer pour tenter de prédire correctement le comportement de la fonction objectif. Cette idée d'utiliser un modèle pour représenter le comportement de la fonction objectif repose sur la supposition que les dérivées de l'objectif existent, alors que cette supposition n'est pas nécessaire pour les méthodes de recherche directe. Le modèle peut être construit de plusieurs manières, même si dans la littérature, beaucoup de méthodes utilisent les modèles quadratiques, qui sont un bon compromis entre une bonne approximation de la fonction objectif et un coût de construction relativement bas. Pour obtenir un modèle, on utilise un ensemble de points de l'espace sur lesquels se base une interpolation ou régression. Afin d'assurer une bonne qualité du modèle construit, il a fallu introduire une théorie concernant l'ensemble des points utilisés. En effet, il faut tout d'abord vérifier que le premier ensemble utilisé, qui peut être généré par une technique comme celle de l'hypercube latin (McKay, Beckman, & Conover, 1979), est bien posé [2]. Ensuite, il faut assurer que la stratégie de recherche et d'acceptation de nouveaux points appliquée par la méthode protège cette propriété en tout temps. Le chapitre 6 du livre (A. R. Conn, Scheinberg, & Vicente, 2008) ainsi que les travaux présentés dans Scheinberg & Toint (2010); A. Conn, Scheinberg, & Vicente (2008, 2005), donnent les outils nécessaires afin de vérifier les bonnes propriétés du modèle quelle que soit la stratégie employée pour le générer.

Cette idée de manipuler un modèle que l'on connaît explicitement permet de réutiliser les techniques de résolution destinées à l'optimisation avec dérivées. L'approche par régions de confiance (A. R. Conn et al. (2008), chapitre 10), par exemple, est une des ces techniques qui ont été adoptées par la DFO. Dans ces méthodes, on minimise le modèle quadratique construit dans la région de confiance en appliquant une technique comme le gradient projeté, la méthode

2. well poised

de J. Moré & Sorensen (1983), ou encore la méthode de Lanczos (Gould, Lucidi, Roma, & Toint, 1999). Nous mentionnerons dans la section suivante, plusieurs solveurs qui se basent sur cette approche de régions de confiance et qui sont spécialement destinés aux problèmes de DFO.

L'utilisation de modèles pour approximer le comportement de la fonction objectif est une approche qui peut aussi être utilisée lors de l'étape de recherche des méthodes directes. En effet, c'est une bonne stratégie pour trouver des points hors de la portée du pas Δ_k où le modèle prédit une amélioration de la valeur courante de la fonction objectif. Cette approche permet donc d'explorer à moindre coût des zones de l'espace un peu plus éloignées contrairement à suivre les directions de recherche de la sonde uniquement. On retrouve des applications de cette utilisation dans les différentes implémentations de l'algorithme de MADS. On peut ainsi comparer les résultats obtenus par la même version de MADS en changeant la stratégie implémentée lors de la recherche, comme l'utilisation de modèles quadratiques (A. Conn & Le Digabel, 2013), les processus gaussiens arborés (TGP, Gramacy & Le Digabel (2011)) ou encore l'utilisation des arbres dynamiques (dynaTree, Talgorn, Le Digabel, & Kokkolaras (2014)).

Il faut néanmoins savoir qu'il existe d'autres méthodes pour la DFO qui ne rentrent dans ni l'une ni l'autre des catégories présentées auparavant. On peut citer principalement les méthodes statistiques et les heuristiques qui n'ont donc pas de garantie de convergence. Parmi les méthodes statistiques, on trouve les méthodes d'échantillonnage telles que l'hypercube latin (McKay et al., 1979) qui, pour générer p points, crée une grille dans l'espace $\mathbb{R}^n$ en divisant chaque dimension en p intervalles. Il affecte ensuite exactement un point par intervalle de la grille. Supposons qu'on affecte un point à une case, ou hypercube. Si on considère tous les hyperplans qui contiennent des faces de cet hypercube, alors aucune autre case dont une face se trouvent sur un de ces hyperplans ne doit contenir de point. La méthode d'échantillonnage normal multivarié MVNS est aussi utilisée pour générer un certain nombre de points dans l'espace. Elle prend en paramètre une moyenne et une matrice de covariance et renvoie des vecteurs aléatoires qui suivent une loi normale. Ces techniques sont par principe, coûteuses en nombre d'évaluations de la fonction objectif. Au lieu de s'en servir comme technique de résolution directement, elles sont plutôt utiles pour générer assez de points pour construire un modèle de la fonction objectif. Certaines méthodes statistiques se basent sur le principe des fonctions substitut. La méthode EGO (Jones, Schonlau, & Welch, 1998) en est un exemple. Dans cet algorithme, on commence par générer des points avec une technique d'échantillonnage afin de construire un modèle de Krigeage dont on se sert pour maximiser

l'espérance d'amélioration [3].

La méthode de recuit simulé (Hwang, 1988) est une heuristique souvent utilisées en DFO. Elle est inspirée d'un processus utilisé en métallurgie. On commence par choisir un point de départ, une température initiale T et une fonction pour la mettre à jour. L'algorithme évolue en acceptant ou non de nouveaux points et en modifiant la température T. Une grande valeur de T permet d'explorer une plus grande partie de l'espace ; c'est l'étape de diversification. Au contraire, une petite valeur de T explore plus intensivement un voisinage proche du point actuel ; c'est l'étape d'intensification.

Finalement, les algorithmes évolutionnaires se basent sur la notion de populations qui s'adaptent à chaque génération afin de trouver la meilleure solution possible. C'est le cas des méthodes de colonies de fourmis (Dorigo, Birattari, & Stützle, 2006) ou d'abeilles (Teodorović & M. Dell'Orco, 2005) qui sélectionnent les meilleurs individus à chaque itération pour générer de nouveaux éléments grâce à des mutations ou croisements. Ces nouveaux individus servent à remplacer les éléments de la colonie qui ont la plus mauvaise performance, c'est-à-dire dans notre contexte, les plus grandes valeurs de l'objectif.

Avec ce bref survol des différentes classes de méthodes de l'optimisation applicables à la DFO, on a une bonne idée de la quantité et la diversité des techniques de résolution utilisées d'un point de vue théorique. On s'intéresse à présent à la manière dont se présentent ces méthodes à l'utilisateur, c'est-à-dire aux solveurs proprement dit. On verra dans les sections suivantes les différents langages de programmation utilisés et la facilité avec laquelle on peut manipuler ces solveurs et les brancher directement à un problème en particulier.

2.2 Solveurs

D'un point de vue pratique, les solveurs sont des « packages » ou boîtes à outils, isolés ou rattachés à un environnement donné, où l'on peut retrouver une implémentation d'une méthode de résolution en particulier. La plupart des solveurs utilisés en DFO sont en libre accès et leur code est mis à jour régulièrement.

Afin d'avoir une vision d'ensemble, le tableau 2.1 liste quelques solveurs utilisés en DFO. Ces solveurs sont séparés en plusieurs classes en fonction de la méthode de résolution sur

3. Expected improvement

laquelle ils reposent. Ce tableau précise aussi le langage de programmation utilisé, ce qui permet de voir la diversité des langages d'implémentation choisis. On ne s'attardera pas à voir si les solveurs proposés gèrent les problèmes avec contraintes ou non puisque nous étudions uniquement le cas sans contraintes.

Tableau 2.1 Liste non exhaustive des solveurs et leur langage de programmation.

Classe	**Nom du solveur**	**Langage**	**Référence**
Recherche directe	DIRECT	Fortran	Hooke & Jeeves (1961)
	NOMAD	C++	Le Digabel (2011)
	MCS	Matlab	Huyer & Neumaier (1999)
	NMSMAX	Matlab	Higham (2002)
	APPSPACK	C++	Hough, Kolda, & Torczon (2001)
	HOPSPACK	C++	Plantenga (2009)
	SID-PSM	Matlab	Custódio & Vicente (2007)
Modèles	BOBYQA	Fortran	M. Powell (2009)
	UOBYQA	Fortran	M. Powell (2001)
	DFO	Fortran	A. R. Conn, Scheinberg, & Toint (1998b)
	CONDOR	Matlab	Berghen & Bersini (2005)
	ORBIT	Matlab	S. Wild, Regis, & Shoemaker (2008)
	LINCOA	Fortran	M. Powell (1994)
	NEWUOA	Fortran	M. Powell (2006)
	WEDGE	Matlab	Marazzi & Nocedal (2002)
Autres	IFFCO	Matlab	Choi et al. (1999)
	PSWARM	Multiple	Vaz & Vicente (2009)
	CMA-ES	Multiple	Hansen (2006)
	IMFIL	Matlab	Patrick (2000)

Parmi les solveurs listés dans le tableau 2.1, on peut voir que la quasi-totalité d'entre eux a été implémentée dans un seul langage de programmation, exception faite de PSWARM et CMA-ES. PSWARM (Vaz & Vicente, 2009) est un algorithme hybride qui combine la recherche par motifs en suivant des directions de recherche pendant la phase de sonde et l'essaim de particules pendant la recherche. PSWARM a été implémenté en Matlab et en C,

avec des interface R, AMPL et Python. CMA-ES (Hansen, 2006) a aussi été implémenté dans plusieurs langages, et utilise une méthode stochastique évolutionnaire dont l'idée principale est d'adapter à chaque itération la matrice de variance/covariance de la distribution multi-normale pour l'étape de mutation. CMA-ES est disponible en C, C++, Java, Matlab, Octave, Python et Scilab.

Le fait de fournir plusieurs implémentations d'un solveur est un très bon moyen de le rendre plus accessible à tous et facilement utilisable. L'utilisateur n'est plus obligé d'écrire une interface afin de faire le lien entre le solveur et sa propre application.

Parmi les autres solveurs, beaucoup d'entre eux ont été développé en Fortran (77 et 90), principalement la famille de solveurs écrite par Mike Powell : UOBYQA, BOBYQA, NEWUOA, LINCOA, etc. Ces solveurs sont de bons exemples de méthodes qui se basent sur des modèles linéaires dans le cas de LINCOA ou quadratiques pour résoudre des problèmes sans contraintes dans le cas de UOBYQA, ou encore avec contraintes de bornes dans le cas de BOBYQA. Le solveur DFO est une implémentation d'un algorithme de régions de confiance pour l'optimisation sans dérivées. DIRECT est l'implémentation d'une méthode de recherche directe qui se base sur la subdivision de l'espace de recherche en rectangles.

Étant donné que Fortran est un langage de programmation compilé, un même solveur implémenté en Fortran sera plus rapide que s'il est implémenté dans un langage interprété comme Python. Ajoutons à cela le fait que le Fortran, en tant que langage spécialisé pour le calcul scientifique, donne accès à une très vaste collection de librairies de résolution numérique, ce qui présente un gros avantage pour n'importe quel solveur.

On retrouve aussi plusieurs solveurs développés en C++, qui est un langage orienté objet, contrairement au Fortran vu précédemment. L'avantage de ce type de langage se fait surtout ressentir au niveau de la conception des programmes. En effet, une conception orientée objet facilite le développement de structures complexes de manière instinctive, même si cela peut ralentir la vitesse d'exécution. D'autant plus qu'elle facilite la manipulation du code et permet de personnaliser les algorithmes plus facilement. Parmi ces solveurs, HOPSPACK, qui est l'évolution de APPSPACK, sont tous deux des solveurs parallèles pour l'optimisation sans dérivées. Ils fournissent une implémentation robuste de l'algorithme GPS et sont capables de prendre en compte des contraintes linéaires. NOMAD est aussi développé en C++. Il repose sur l'algorithme de MADS évoqué dans la section précédente. Ce solveur de recherche

directe permet de se placer sur un maillage adaptatif et évolue en suivant des directions générées suivant une stratégie prédéterminée, qui assurent que tout l'espace disponible peut théoriquement être atteint.

Finalement, l'autre langage qui semble être assez présent est Matlab. Par exemple, le solveur NMSMAX est une implémentation de l'algorithme de Nelder et Mead qui fait partie de la boîte à outils *The matrix computation toolbox*. Les solveurs GPS et NOMAD sont aussi implémentés en MATLAB sous la Global Optimization Toolbox. Le solveur MCS (Huyer & Neumaier, 1999) implémente une approche heuristique de recherche globale par coordonnées. On trouve aussi des solveurs qui implémentent des méthodes de régions de confiance tels que WEDGE (Marazzi & Nocedal, 2002) ou ORBIT (S. Wild et al., 2008) qui construit des modèles d'interpolation en se basant sur les fonctions à bases radiales (RBF). Finalement, CONDOR (Berghen & Bersini, 2005) est une extension d'UOBYQA en Matlab, parallélisable et qui prend en compte les contraintes. Matlab est un environnement spécialement destiné au calcul scientifique. Sa syntaxe et les outils de travail qui lui sont rattachés facilitent beaucoup le développement de solveurs numériques. Il peut aussi s'interfacer avec d'autres programmes en C, C++, Fortran et Java. Il est donc très logique de retrouver un grand nombre de solveurs codés en Matlab. Néanmoins, sachant que la licence n'est pas gratuite, cela peut être considéré comme un désavantage par rapport aux solveurs disponibles en source libre.

Tous les solveurs vus jusqu'à présent font partie d'une grande famille de logiciels destinés à l'optimisation sans dérivées. On a pu voir que chaque solveur a été développé en un ou plusieurs langages de programmation différents. Le choix d'un langage au détriment d'un autre peut être justifié de différentes manières, que ce soit par un soucis de rapidité, facilité de développement ou possibilité de parallélisation. Ces solveurs restent néanmoins des produits isolés dans le sens où ils sont tous présentés sous forme de logiciels à télécharger (gratuitement pour la plupart), que l'on peut ensuite brancher avec le code ou l'exécutable de la fonction objectif. C'est donc à l'utilisateur de faire le travail de recueillir les différents solveurs qui peuvent l'intéresser, écrire les interfaces nécessaires pour faire le lien avec son application. Cet état éparpillé peut être contraignant pour un utilisateur qui désirerait effectuer une étude comparative entre les différents logiciels disponibles afin de décider du meilleur choix à prendre pour son cas particulier.

Il serait intéressant d'avoir une bibliothèque ou boîte à outils à sa disposition qui regrouperait plusieurs algorithmes capables de résoudre le même type de problèmes. Le tableau 2.2 liste quelques unes des boîtes à outils disponibles pour la résolution de problèmes issus de

l'optimisation sans dérivées.

Tableau 2.2 Liste non exhaustive des boîtes à outils et leur langage de programmation

Nom de la boîte à outils	**Payant / Gratuit**	**Langage**	**Référence**
COINOR	Gratuit	Multiple	Lougee-Heimer (2003)
Dakota	Gratuit	C++	Adams et al. (2009)
DFL	Gratuit	Multiple	Liuzzi (2015)
NLOpt	Gratuit	Multiple	Currie & Wilson (2012a)
OpenSolver	Gratuit	Excel	Mason & Dunning (2010)
TOMLAB	Payant	Matlab	K. Holmström (2001)
Opti Toolbox	Gratuit	Matlab	Currie & Wilson (2012b)

Ces outils sont des bibliothèques gratuites ou payantes, comme TOMLAB, qui offrent une grande diversité d'algorithmes destinés à diverses branches de l'optimisation dont l'optimisation sans dérivées. C'est le cas de la boîte à outils Dakota (Adams et al., 2009), développée en C++, par exemple qui s'est voulue spécialisée dans l'optimisation sans dérivées au départ mais qui a ensuite englobé d'autres types de problèmes et applications issus de l'ingénierie. On trouve dans Dakota une implémentation de l'algorithme DIRECT vu précédemment, PATTERN qui regroupe plusieurs algorithmes de recherche pas motifs et EA pour les algorithmes génétiques.

DFL quant à elle, est une librairie qui regroupe une douzaine de solveurs spécialement destinés à l'optimisation sans dérivées. Ils sont généralement tous présents soit en C, en Fortran 90 ou parfois les deux. Les différents solveurs présents se distinguent selon les principes utilisés : recuit simulé, recherche linéaire, etc., ainsi que leur gestion des contraintes de bornes ou des contraintes générales.

TOMLAB est un environnement Matlab qui regroupe aussi plusieurs algorithmes d'optimisation sans dérivées. Par exemple glcCluster est une implémentation d'une version hybride de DIRECT avec des techniques de *clustering*, LGO regroupe un ensemble de méthodes de résolution non linéaire globale et locale ; et beaucoup d'autres.

Les boîtes à outils présentées possèdent des implémentations algorithmiques très sophistiquées et avancées. Notre bibliothèque `LIBDFO.py` ne prétend pas atteindre ces niveaux d'efficacité, le but recherché étant plutôt de fournir des algorithmes implémentés de la manière la plus modulaire possible afin de permettre à l'utilisateur un certain niveau de flexibilité pour qu'il puisse modifier et personnaliser un algorithme autant qu'il le souhaite.

Cette idée de présenter une boîte à outils qui contient déjà plusieurs solveurs faciles à manipuler, tester, comparer et modifier d'une part ; et qui supporte tout autant le développement de nouvelles méthodes est bien tout le but derrière le travail que nous allons présenter dans le chapitre suivant. On commencera par expliquer le choix du langage de programmation utilisé, ensuite on détaillera les idées d'implémentation qui caractérisent la bibliothèque présentée et qui ont pour but de faciliter l'ajout de n'importe quelle nouvelle méthode d'une part et la création de nouvelles variantes de l'autre.

CHAPITRE 3 LIBRAIRIE PYTHON

Dans ce chapitre, on parle de la structure de la bibliothèque fournie. On justifie dans un premier temps le choix du langage Python. Ensuite on explique les structures de données implémentées qu'il faut utiliser pour bien manipuler les algorithmes fournis ou pour implémenter un nouveau solveur.

3.1 Pourquoi Python ?

Python est un langage de programmation en licence libre qui est apparu au début des années 1990. Bien qu'il ne soit pas spécialement conçu pour le calcul scientifique, il présente quelques caractéristiques intéressantes qui justifient le choix de ce langage en particulier pour implémenter la bibliothèque présentée.

Python est orienté objet. Cela nous permet de créer des structures complexes et de les manipuler simplement pour représenter les concepts algorithmiques et mathématiques. Il est multi-plateforme et peut donc être exécuté sur plusieurs systèmes d'exploitation différents sans aucun prérequis. Il n'a pas besoin de compilateur puisque c'est un langage interprété. Il bénéficie aussi d'une gestion automatique de la mémoire ce qui allège le code et facilite la programmation dans le sens où le programmeur a moins d'aspects à gérer.

Comparé aux autres langages généralement utilisés en la programmation mathématique, comme le C, C++ et Fortran, Python est relativement plus lent au niveau du temps d'exécution. Cependant, lorsqu'on se place dans un contexte de DFO, on considère que la différence au niveau du temps d'exécution entre un code en Python et en C est négligeable comparée au temps nécessaire à l'évaluation de la fonction objectif, souvent issue des simulations. D'autre part, les opérations des produit matrices vecteurs, les transformations de Fourier et d'autres opérations communes en programmation scientifique. sont plus rapidement exécutées par des langages compilés de bas niveau. Dans un code Python, on a la possibilité de les déléguer à un langage de plus bas niveau en utilisant des outils comme Cython (Smith, 2015) par exemple.

3.2 Structure de la bibliothèque

Afin de faciliter l'utilisation de la bibliothèque, la section suivante explique le fonctionnement des classes fournies et des structures de données utilisées.

Les solveurs

La classe `Solver` du fichier `solver.py` peut être héritée par les solveurs implémentés. Elle se charge principalement des conditions d'arrêt. De manière générale, un solveur de DFO peut s'arrêter pour une des raisons suivantes :

- le temps d'exécution supérieur à `max_time` ;
- le nombre d'évaluations de la fonction supérieur à `max_feval` ;
- le résidu est inférieur la tolérance `tol`.

Le résidu désigne la taille de la région de confiance, la taille du simplexe, ou la taille du maillage en fonction du type d'algorithme utilisé.

Pour initialiser un solveur, on doit initialiser la classe mère en précisant les paramètres `max_feval, max_time, min_res`. Dans UOBYQA, cela se fait de la manière suivante :

```
1 class UOBYQA(Solver):
2
3     def __init__(self, x0, problem, rhobeg, rhoend, **kwargs):
4         super(UOBYQA, self).__init__(min_res=rhoend, **kwargs)
```

Plus tard, pour vérifier si les conditions d'arrêt de l'algorithme sont satisfaites, on fait appel à la fonction `check_termination()`. Celle ci renvoie un booléen `stop` et le message `flag` qui explique la raison de l'arrêt.

```
1 (stop, flag) = self.check_termination()
```

Les points manipulés

Les solveurs de la bibliothèque manipulent une structure de point particulière. La classe `EvalPoint` est celle qui implémente cette structure. Un `EvalPoint` a pour attribut :

- un numéro d'identification `tag` ;
- un vecteur d'entrée `input` qui représente ses coordonnées ;
- une sortie `output` qui est la valeur de la fonction en ce point. Initialement, cette valeur est `None` par défaut ;
- et l'attribut `direction` qui est la dernière direction de recherche suivie. Cette attribut peut ne pas être utilisé suivant le solveur exécuté.

L'ensemble d'interpolation

Dans le chapitre 2, nous avons parlé d'une classe de méthodes qui se basent sur les modèles. Ces algorithmes sont amenés à manipuler un ensemble de points d'interpolation. Cet ensemble doit vérifier certaines propriétés géométriques afin d'assurer une bonne qualité du modèle construit. La première classe est celle du fichier `set.py`. Une instance de `Set` est initialisée à partir d'un point de départ x_0 et un rayon Δ. Elle contient une liste d'`EvalPoint` qu'on peut fournir directement ou générer par la suite grâce à la méthode de l'hypercube latin (McKay et al., 1979) qui y est implémentée. Cette classe permet aussi de calculer le point central d'un `Set` donné, ainsi que le rayon minimal de la boule qui inclut tous les points de l'ensemble. Dans cette classe, on a aussi une méthode qui donne la représentation matricielle des coordonnées des points de l'ensemble dans la base naturelle de degré 2. De manière générale la base naturelle de degré d peut s'écrire :

$$\Phi = \left\{1, x_1, x_2, .., x_n, \frac{x_1^2}{2}, x_1x_2, .., \frac{x_{n-1}^{d-1}x_n}{(d-1)!}, \frac{x_n^d}{d!}\right\}.$$

Pour un ensemble de points $Y = (y_0, y_1, .., y_p)$ de $\mathbb{R}^n$, la représentation matricielle de ces points dans la base naturelle de degré $d = 2$ est une matrice de la forme :

$$M(\Phi, Y) = \begin{pmatrix} 1 & y_0^0 & \dots & y_n^0 & \frac{1}{2}(y_0^0)^2 & \dots & \frac{(y_{n-1}^0)^{d-1}y_n^0}{(d-1)!} & \frac{(y_n^0)^d}{d!} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots \\ 1 & y_0^p & \dots & y_n^p & \frac{1}{2}(y_0^p)^2 & \dots & \frac{(y_{n-1}^p)^{d-1}y_n^p}{(d-1)!} & \frac{(y_n^p)^d}{d!} \end{pmatrix}.$$

Le théorème 3.14 du livre *Introduction to derivative free optimization* (A. R. Conn et al., 2008) fait le lien entre le conditionnement de la matrice $M(\Phi, Y)$ et le caractère bien posé de l'ensemble Y. Ainsi, si $Y = (y_0, y_1, .., y_p)$ est Λ-posé (définition 3.2.3) sur la boule $\mathcal{B}(0, 1)$, alors $||M^{-1}|| \leq \theta(p+1)^{\frac{1}{2}}\Lambda$, où $\theta > 0$ dépend de n et d mais pas de Y et Λ. Inversement, si M est non singulière telle que $||M^{-1}|| \leq \Lambda$, alors Y est $(\sqrt{(p+1)}\Lambda)$-posé.

Les méthodes qui se basent sur des modèles peuvent construire ces derniers par interpolation, régression ou autre. Notre bibliothèque possède une classe qui implémente les ensembles d'interpolation. Pour implémenter un autre type d'ensemble, il faut créer une sous classe de `Set`, comme c'est le cas de la classe `InterpolationSet`.

La classe `InterpolationSet`, qui hérite de la classe `Set`, regroupe quelques méthodes pour

manipuler un ensemble d'interpolation et maintenir les propriétés géométriques pour faire en sorte qu'il soit bien posé.

Définition 3.2.1. Soit un ensemble $Y = (y_0, y_1, .., y_p)$ de points de $\mathbb{R}^n$. Soit ϕ^* une base polynomiale de l'espace des polynômes de $\mathbb{R}^n$ de degré inférieur à d noté $\mathcal{P}_n^d$. On dit que Y est bien posé si la matrice correspondante $M(\phi^*, Y)$ est non singulière.

Définition 3.2.2. Soit $Y = (y_0, y_1, .., y_p)$ un ensemble d'interpolation où chaque $y_i \in \mathbb{R}^n$. On dit que la base des polynômes $l_j(x), j = 0, .., p$ de $\mathcal{P}_n^d$ est la base des polynômes de Lagrange associée à Y si :

$$l_j(y_i) = \delta_{ij} = \begin{cases} 1 & \text{si} \quad i = j, \\ 0 & \text{sinon.} \end{cases}$$

Pour mieux mesurer ce caractère bien posé, on introduit la notion de Λ-posé.

Définition 3.2.3. Soit $\Lambda > 0$, une boule $\mathcal{B} \in \mathbb{R}^n$ et $\phi^* = \{\phi_0^*(x), ..., \phi_p^*(x)\}$ une base polynomiale de $\mathcal{P}_n^d$. On dit qu'un ensemble $Y = (y_0, y_1, .., y_p)$ est Λ-bien posé lorsqu'une des propriétés suivantes est vérifiée :

- pour la base des polynômes de Lagrange associée à Y on a :

$$\Lambda \geq \max_{0 \leq i \leq p} \max_{x \in \mathcal{B}} |l_i(x)|.$$

- $\forall x \in \mathcal{B}, \exists \lambda(x) \in \mathbb{R}^{p+1}$ tel que :

$$\sum_{i=0}^{p} \lambda_i(x) \phi^*(y_i) = \phi^*(x) \quad \text{avec} \quad ||\lambda(x)||_\infty \leq \Lambda.$$

- Si on remplace n'importe quel point $y \in Y$ par un point $x \in \mathcal{B}$, alors le volume de l'ensemble $\phi^*(Y) = \{\phi^*(y_0), ..., \phi^*(y_p)\}$ augmente au plus d'un facteur Λ.
  Le volume de $\phi^*(Y)$ est

$$v(\phi^*(Y)) = \frac{|\det(M(\phi^*, Y))|}{(p+1)!}.$$

D'après la définition 3.2.3, on voit que pour maintenir le caractère Λ-bien posé d'un ensemble d'interpolation, on peut travailler directement avec les mesures des polynômes de Lagrange.

Dans la classe `InterpolationSet`, on trouve une implémentation des algorithmes 6.1 et 6.2 du livre A. R. Conn et al. (2008). L'algorithme 6.1 permet de calculer les polynômes de Lagrange associés à un ensemble de points $Y = (y_0, y_1, .., y_p)$ à condition que cet ensemble soit bien posé ; dans le cas contraire l'algorithme s'arrête en informant que l'ensemble Y ne satisfait

pas les bonnes propriétés. L'algorithme commence par calculer $\{l_j(y_i) = \phi(y_i), i, j = 0, .., p\}$ où ϕ représente la base naturelle. Elle fait office de première approximation de la base des polynômes de Lagrange désirée. Cet algorithme cherche à chaque itération i un indice $j_i \in \text{argmax}_{i \leq j \leq p} |l_i(y_j)|$. Puis on procède à des normalisations :

$$l_i(x) \leftarrow l_i(x)/l_i(y_{j_i}) \tag{3.1}$$

et orthogonalisations pour $j = 0, .., p \quad j \neq i$:

$$l_i(x) \leftarrow l_j(x) - l(y_i)l_i(x) \tag{3.2}$$

jusqu'à obtention du résultat voulu.

L'algorithme 6.2 a pour objectif de compléter un ensemble de points $Y' = (y_0, y_1, .., y_k)$ où $k < p$ et p est le nombre de points d'interpolation voulu. Cet algorithme fait en sorte que l'ensemble des points d'interpolation renvoyé est bien posé. On commence aussi par une approximation des polynômes de Lagrange en utilisant la base naturelle. À chaque étape $i = 0, .., p$, on trouve l'indice $j_i \in \text{argmax}_{i \leq j \leq k} |l_i(y_j)|$. Si $|l_i(y_j)| > 0$ et $i \leq k$ alors on échange les points y_i et y_{j_i}, sinon on calcule le point $y_i \in \text{argmax}_{x \in \mathcal{B}} |l_i(x)|$. On finit l'itération en appliquant une normalisation (3.1) et une orthogonalisation (3.2).

Ainsi, on peut obtenir un ensemble d'interpolation bien posé et calculer les polynômes de Lagrange associés en initialisant l'`InterpolationSet` avec un point de départ x_0 et un rayon Δ.

Les modèles

L'ensemble d'interpolation vu précédemment est utilisé pour construire un modèle par interpolation.

Définition 3.2.4. Soit une fonction $f \in \mathcal{C}^1$ telle que ∇f est lipschitz.
On dit qu'un modèle m_f de f est FL [1] sur $\mathcal{B}(x, \Delta)$ si $\exists k_f, k_g > 0 / \forall y \in \mathcal{B}(x, \Delta)$

$$\begin{cases} |f(y) - m_f(y)| \leq k_f \Delta^2 \text{ et} \\ ||\nabla f(y) - \nabla m_f(y)|| \leq k_g \Delta. \end{cases}$$

1. Fully linear

Définition 3.2.5. Soit une fonction $f \in \mathcal{C}^2$ telle que ∇f^2 est lipschitz.
On dit qu'un modèle m_f de f est FQ [2] sur $\mathcal{B}(x, \Delta)$ si $\exists k_f, k_g, k_h > 0 / \forall y \in \mathcal{B}(x, \Delta)$

$$\begin{cases} |f(y) - m_f(y)| \leq k_f \Delta^3 \\ ||\nabla f(y) - \nabla m_f(y)|| \leq k_g \Delta^2 \text{ et} \\ ||\nabla^2 f(y) - \nabla^2 m_f(y)|| \leq k_h \Delta. \end{cases}$$

Pour les algorithmes de région de confiance qui suivent les structures décrites par les algorithmes 10.1 et 10.3 du livre A. R. Conn et al. (2008), ces propriétés garantissent respectivement une convergence vers un point critique de premier ou de second ordre.

La classe `Model` est initialisée par un point de départ x et un ensemble d'interpolation. Dans le chapitre 3 du livre A. R. Conn et al. (2008), la définition 3.1 montre que, étant donné un ensemble d'interpolation $Y = (y_0, y_1, .., y_p)$ bien posé, un point $x \in \mathbb{R}^n$ et une base polynomiale ϕ, la matrice $M(\phi, Y)$ est non singulière. On peut exprimer la valeur de $\phi(x) = [\phi_0(x), .., \phi_p(x)]^t$ sous la forme d'une combinaison linéaire unique des vecteurs $\phi(y_i), \quad i = 0, .., p$:

$$\sum_{i=0}^{p} l_i \phi(y_i) = \phi(x), \tag{3.3}$$

ou sous forme matricielle :

$$M(\phi, Y)^t l(x) = \phi(x) \quad l(x) = [l_0(x), .., l_p(x)]^t, \tag{3.4}$$

où $l_i(x)$ est le i^{e} polynôme de Lagrange associé à Y.

Dans la classe `Model`, la fonction `lagrange` calcule la base des polynômes de Lagrange associée à l'ensemble Y en utilisant la base naturelle comme base ϕ :

```
1 def lagrange(self, x):
2     M = self.interset.matrix_natural_basis(self.points)
3     b = self.interset.vector_natural_basis(x.input)
4     l = np.linalg.solve(M, b)
5     self.polynoms = l
```

Dans les méthodes de région de confiance, on a besoin de résoudre un sous problème consistant à minimiser le modèle à l'intérieur de la région de confiance. Dans le cas des modèles quadratiques, le fichier `min_quad.py` regroupe quelques méthodes pour résoudre ce sous problème.

2. Fully quadratic

Les détails de ces techniques seront présentés dans le chapitre suivant, lors de la discussion de la méthode UOBYQA.

On se tourne à présent vers l'autre classe de méthodes de résolution qui est celle des méthodes de recherche directe.

Le maillage

On s'intéresse particulièrement à l'étape de sonde de ces algorithmes. Durant cette étape, on se place à l'itération k en un point x_k du maillage, qui est de taille Δ_k. En fonction de l'algorithme utilisé, on génère un ensemble de directions de recherche afin de trouver potentiellement un point meilleur. Le maillage doit être mis à jour sur base du succès de cette étape.

La classe `Mesh` est la structure qui représente un maillage dans notre environnement. À l'itération k, le maillage M_k s'écrit sous la forme suivante :

$$M_k = \bigcup_{x \in V_k} \{x + \Delta_k D z : z \in \mathbb{N}^{n_D}\},$$

où V_k est l'ensemble de tous les points déjà évalués et D est une matrice de taille $n \times n_D$ dont les colonnes représentent des directions de $\mathbb{R}^n$. Pour initialiser un objet de type `Mesh`, on doit préciser la dimension n du problème, la taille du maillage initial Δ_0, sa taille minimale Δ_{min} et un coefficient de mise à jour $\tau > 1$. À l'itération k, on se trouve sur le maillage de taille Δ_k au point x_k. On sonde un ensemble de points P_k dans le but de trouver un meilleur point qui améliore l'objectif. Ensuite, on met à jour le maillage de la manière suivante :

$$\Delta_{k+1} = \begin{cases} \tau^{\omega^+} \Delta_k & \text{si un meilleur point est trouvé,} \\ \tau^{\omega^-} \Delta_k & \text{sinon.} \end{cases}$$

Les paramètres ω^+ et ω^- dépendent du solveur utilisé. Par exemple, dans la recherche par coordonnées, $\omega^+ = 0$ et $\omega^- = -1$.

Les directions de recherche

Ce qui différencie une méthode de recherche directe d'une autre est principalement le choix des directions de recherche.

Pour implémenter une nouvelle méthode de recherche directe, il faut créer une classe qui

hérite de `Direction` et qui précise comment l'ensemble des directions de recherche D_k est généré à chaque itération k. Par exemple, pour les directions de la recherche par coordonnées, on implémente la classe suivante :

```
class CSDirections(Directions):
    """Coordinate Search with 2n directions."""

    def get_directions(self, mesh):
        poll_size = mesh.poll_size
        n = len(poll_size)
        dirs = []
        for k in xrange(n):
            d = np.zeros(n)
            d[k] = poll_size[k]
            dirs.append(d)
            dirs.append(-d)
        return dirs
```

Les problèmes

Dans notre environnement, on peut implémenter soi-même des problèmes de DFO. Pour cela, il faut utiliser la structure de la classe `DFOProblem`. L'initialisation se fait avec un point de départ x_0, le nombre de variables $nvar$, le nombre de contraintes $ncon$, ainsi que les bornes $Lvar$ et $Uvar$ qui représentent les contraintes de bornes $Lvar \leq x \leq Uvar$. Bien que les solveurs implémentés ne sont pas adaptés aux problèmes avec contraintes, cette structure a été ajoutée dans le but de faciliter la prise en charges des contraintes dans des travaux futurs. Chaque sous classe doit implémenter la fonction `eval(self, x)` qui évalue l'objectif f en x.

On peut aussi utiliser des exécutables directement. La classe `DFOBlackBox`, elle même une sous classe de `DFOProblem`, implémente une interface pour faire appel à un exécutable, évaluer la boîte noire en un point et renvoyer la valeur de l'objectif en ce point et potentiellement la valeur des contraintes. L'exemple suivant montre comment initialiser un objet de type `DFOBlackBox` en utilisant un exécutable fictif nommé `rosenbrock` :

```
executable = './rosenbrock'
nvar = 3
ncon = 0
x0 = np.array([1, 1, 1])
```

```
problem = DFOBlackBox(nvar, ncon, x0, executable)
```

Les profils

On s'intéresse à présent à la comparaison des algorithmes de DFO. J. J. Moré & Wild (2009) proposent une approche qui consiste à tracer des profils de performance et de données. On peut comparer l'efficacité et la rapidité de résolution d'un algorithme par rapport à un autre en interprétant ces profils. Le module `Profiles.py`, qui est indépendant du reste de la bibliothèque, permet de tracer les profils de performance et de données.

On considère un ensemble $\mathcal{S}$ de solveurs qu'on veut comparer sur un ensemble $\mathcal{P}$ de problèmes. Soit un problème $p \in \mathcal{P}$ et un solveur $s \in \mathcal{S}$. La mesure de performance du solveur s sur le problème p est notée $t_{p,s}$. Cette mesure peut indiquer le temps de résolution, le nombre d'évaluations de la fonction avant l'arrêt ou une autre mesure choisie avec la convention que le meilleur $t_{p,s}$ est le plus petit. On note aussi le ratio de performance

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}\}}. \tag{3.5}$$

À partir de cette définition, on voit que le ratio de performance d'un solveur s pour un problème p dépend de la performance des autres solveurs sur ce même problème. Finalement, le profil de performance du solveur s détermine le pourcentage de problèmes dont le ratio de performance est inférieur à un certain seuil α. Lorsqu'on se place sur $\alpha = 1$, on peut lire le nombre de problèmes que le solveur s résout le plus efficacement. Si on se place sur un α assez grand, on lit le pourcentage de problèmes résolus par s.

Les profils de performance de la figure 3.1 compare deux versions de l'algorithme UOBYQA (section 4.2). Lorsque $\alpha = 1$, on voit que la version originale est plus efficace sur 38% des problèmes alors que notre version l'est sur 30% des problèmes. Par contre, la version originale de UOBYQA résout en tout à peu près 92% des problèmes alors que le version modifiée de notre bibliothèque résout un peu plus de 96% des problèmes dans les limites imposées.

Les profils de données permettent de déterminer, pour un certain solveur, le nombre de problèmes qu'il est capable de résoudre en fonction du nombre d'évaluations de gradient de

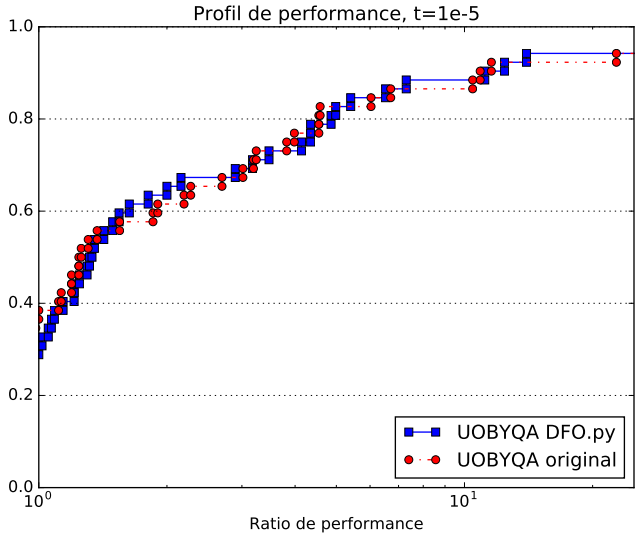

Figure 3.1 Exemple de profils de performance.

simplexe alloué. Le profil de données d'un solveur s est défini de la manière suivante :

$$d_s(\alpha) = \frac{1}{|\mathcal{P}|}\text{card}\{p \in \mathcal{P} : \frac{t_{p,s}}{n_p + 1} < \alpha\} \tag{3.6}$$

où n_p est le nombre de variables du problème p.

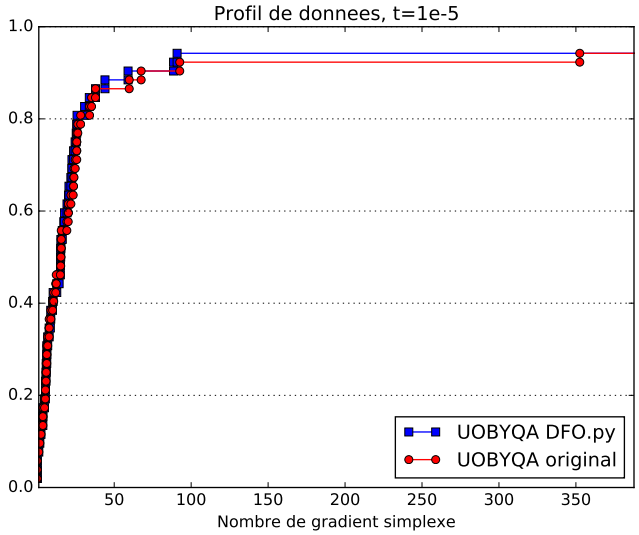

Figure 3.2 Exemple de profils de données.

Les profils de données de la figure 3.2 montre que les deux versions implémentées sont équivalentes en efficacité lorsque le nombre d'évaluations alloué est relativement petit. Par contre, la version originale résout un peu moins de problèmes lorsque le nombre d'évaluations permis est supérieur à 60.

Les problèmes tests

Les profils de performance et de données sont obtenus en exécutant des solveurs sur 52 problèmes tests identiques à ceux présentés dans J. J. Moré & Wild (2009). Ces problèmes sont représentés par les fonctions analytiques du tableau 3.1.

Tableau 3.1 Liste des fonctions utilisées comme problèmes tests.

Nom de la fonction	**Taille du problème**
Fonction linéaire - Rang plein	9
Fonction linéaire - Rang 1	7
Fonction linéaire - Rang 1 avec lignes et colonnes nulles	7
Rosenbrock	2
Helical valley	3
Fonction singulière de Powell	4
Freudenstein et Roth	2
Bard	3
Kowalik et Osborne	4
Meyer	4
Watson	9
Boîte tridimensionnelle	3
Jennrich et Sampson	2
Brown et Dennis	4
Chebyquad	11
Brown quasi-linéaire	10
Osborne 1	5
Osborne 2	11
Bdqrtic	12
Cube	8
Mancino	12
Heart8ls	8

Ces fonctions prennent en argument un point x, le nombre de variables n et construisent un vecteur u de taille m donnée en argument. Ensuite, elles renvoient la valeur de la fonction au

point x selon le mode choisi parmi :

- 'smooth' : $f(x) = \sum_{i=1}^{m} u_i^2$,
- 'nondiff' : $f(x) = \sum_{i=1}^{m} |u_i|$,
- 'wild3' : $f(x) = \sum_{i=1}^{m} (1 + \sigma\phi) u_i^2$,
- 'noisy3' : $f(x) = \sum_{i=1}^{m} (1 + err)^2 u_i^2$.

Où $\sigma = 10^{-2}$, $\phi = 4A^2 - 3$, $A = 0.9 \sin(100||x||_1) * \cos(100||x||_\infty) + 0.1 \cos(||x||_2)$ et $err = 10^-2\epsilon$ où ϵ est une variable aléatoire qui suit une loie uniforme $\in [0, 1]$.

Chacune de ces fonctions est associée à un point de départ usuel x^*. On utilise le paramètre $s \in \{0, 1\}$ pour déterminer le point de départ x_0 du solveur où $x_0 = 10^s x^*$.

CHAPITRE 4 MÉTHODES IMPLÉMENTÉES

Dans ce chapitre, on présente le détail des méthodes implémentées dans la bibliothèque en donnant, quand c'est le cas, les différentes variantes déjà disponibles et en précisant les sections de chaque algorithme qui peuvent être modifiées.

4.1 Nelder et Mead

La méthode de Nelder & Mead (1965) est une méthode «historique» qui manipule un simplexe pour avancer. Un simplexe dans $\mathbb{R}^n$ est un ensemble de $n+1$ points affinement libres. Cela veut dire qu'aucun point n'appartient au sous espace engendré par les n autres points. Cet algorithme, comme mentionné dans le chapitre 2, a été utilisé dans plusieurs domaines et a été cité plus de 22000 fois selon Google Scholar. Cet algorithme ne présente toutefois aucune garantie de convergence.

4.1.1 Description générale

On génère un simplexe initialement. À chaque itération k de l'algorithme, il faut commencer par ordonner les sommets du simplexe de sorte que $f(x_0) \leq f(x_1) \leq ... \leq f(x_n)$. La méthode évolue en effectuant une des quatre opérations suivantes à chaque itération : réflexion, expansion, contraction et réduction. Elles ont pour but d'introduire un ou plusieurs nouveaux points avec de bonnes valeurs de la fonction objectif. Chacune de ces opérations est associée à un coefficient, respectivement $\rho > 0$, $\chi > 1$, $0 < \gamma < 1$ et $0 < \sigma < 1$. De manière générale, on prend les valeurs suivantes :

$$\rho = 1, \quad \chi = 2, \quad \gamma = \tfrac{1}{2}, \quad \sigma = \tfrac{1}{2}.$$

4.1.2 Pseudo-code

Voici le déroulement d'une itération k de l'algorithme de Nelder et Mead :
1. Ordonner les points $f(x_0) \leq f(x_1) \leq ... \leq f(x_n)$
Calculer x_c le barycentre des n meilleurs points : $x_c = \frac{1}{n} \sum_{i=0}^{n-1} x_i$.

2. Réflexion : calculer le point $x_r = x_c + \rho(x_c - x_n)$. Si $f(x_0) \leq f(x_r) < f(x_{n-1})$ alors on accepte le point x_r, c'est à dire $x_n \leftarrow x_r$, et l'itération se termine.

3. Expansion : si $f(x_r) < f(x_0)$ alors on calcule le point $x_e = x_c + \chi(x_r - x_c)$. Si $f(x_e) < f(x_r)$ alors on accepte le point x_e, sinon on accepte le point x_r et l'itération se termine.

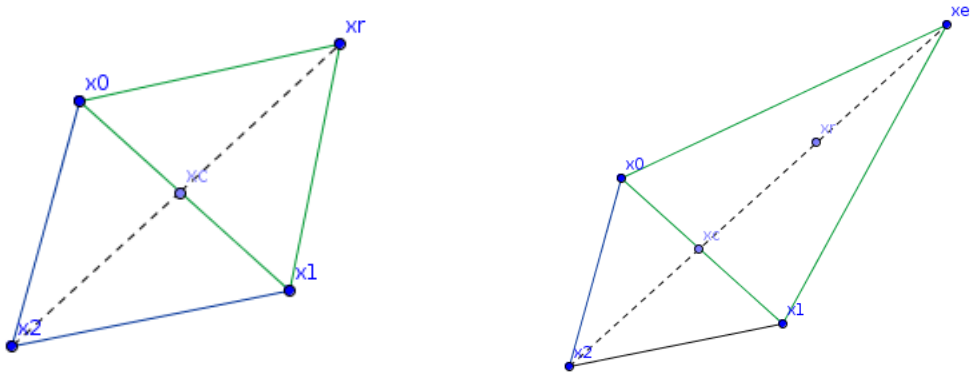

Figure 4.1 À gauche, une réflexion et à droite une expansion.

4. Contraction : si $f(x_r) \geq f(x_{n-1})$
a. si $f(x_{n-1}) \leq f(x_r) < f(x_n)$ alors c'est une *contraction externe*. On calcule le point $x_{ce} = x_c + \gamma(x_r - x_c)$. Si $f(x_{ce}) \leq f(x_r)$ alors on accepte x_{ce} sinon on effectue une réduction.

b. si $f(x_r) \geq f(x_n)$ alors c'est une *contraction interne*. On calcule le point $x_{ci} = x_c + \gamma(x_c - x_n)$. Si $f(x_{ci}) < f(x_n)$ alors on accepte x_{ci} sinon on effectue une réduction.

5. Réduction : On construit n nouveaux points : $y_i = x_0 + \sigma(x_i - x_0)$ pour $i = 1, .., n$. La prochaine itération commencera avec le simplexe formé des points $x_0, y_1, ..., y_n$.

Cet algorithme se termine lorsque la taille du simplexe devient inférieure à une certaine tolérance $\epsilon > 0$ à préciser. Dans cette implémentation, la taille du simplexe est jugée trop petite lorsqu'on vérifie la relation

$$\max_{1 \leq i \leq n} ||x_i - x_0|| \leq \epsilon \max(1, ||x_0||), \tag{4.1}$$

et le meilleur point trouvé est le point x_0.

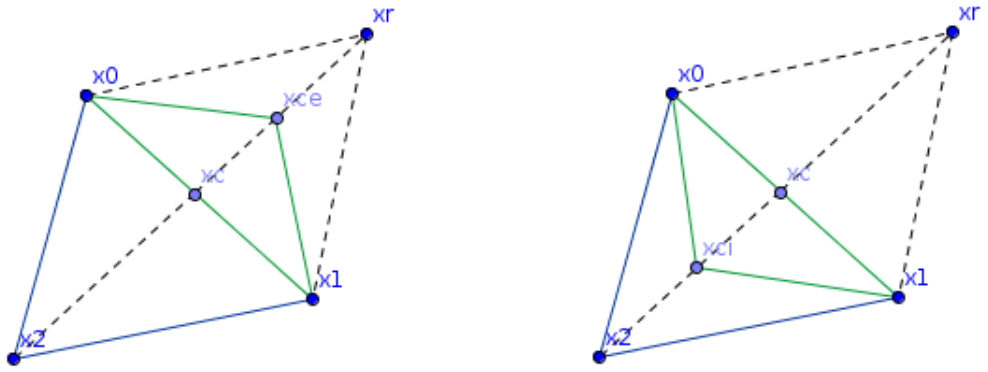

Figure 4.2 À gauche, une contraction externe et à droite une contraction interne.

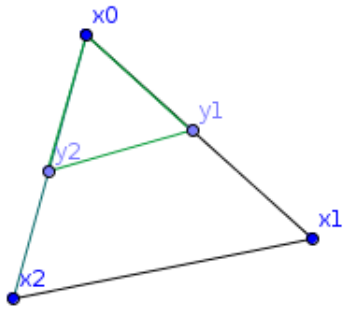

Figure 4.3 Réduction du simplexe.

La méthode de Nelder et Mead est simple à comprendre et à implémenter, et elle présente quelques caractéristiques intéressantes dans le contexte de la DFO. Lorsqu'on n'effectue pas de réduction, on évalue deux nouveaux points au maximum par itération, ce qui permet de ne pas épuiser le nombre d'évaluations maximal rapidement. De plus, ces opérations assurent d'introduire un nouveau point tel que $f(x_{nouv}) < f(x_n)$. Finalement, la construction du nouveau simplexe pendant l'étape de réduction ne fait pas appel à la fonction objectif en soi, mais utilise seulement les coordonnées des points du simplexe courant.

La méthode de Nelder et Mead est flexible au niveau du choix du simplexe de départ. Dans notre bibliothèque, on peut l'initialiser de trois manières différentes :

- définir un simplexe manuellement à l'aide d'une liste de sommets ;
- générer les sommets avec l'hypercube latin ;
- générer les sommets avec la méthode utilisé dans UOBYQA pour construire l'ensemble d'interpolation (voir la section 4.2.2).

La 3ème option sera détaillée dans la section suivante. L'utilisateur est aussi libre d'implémenter une autre technique pour générer le simplexe initial. Il suffit de l'ajouter en tant que sous classe de *Simplex* comme c'est le cas des exemples suivants. Le solveur `NelderMead` est ensuite initialisé en lui donnant comme argument le problème à résoudre, un point de départ x_0 et optionnellement un simplexe initial. Lorsqu'aucun simplexe n'est passé en argument, le solveur `NelderMead` construit un `LHSimplex`.

```
simplex = MYSimplex(points)
simplex = LHSimplex(x0, delta)
simplex = PSimplex(x0, delta, problem)

nel_mead = NelderMead(problem, x0, simplex)
```

4.1.3 Tests numériques

Afin de valider ce solveur, on effectue une comparaison entre notre implémentation de l'algorithme de Nelder et Mead et celle du solveur `NMSMAX`. Le tableau (4.1) résume les résultats obtenus pour deux fonctions objectifs et deux points de départ différents pour chaque objectif.

Tableau 4.1 Validation du solveur Nelder et Mead.

Problème	**Nelder Mead implémenté**	**NMSMAX.m**
Rosenbrock	feval = 195 fmin = 1.04e-17	feval = 200 fmin = -4.47e-18
	feval = 195 fmin = 1.04e-17	feval = 336 fmin = -5.45e-18
Freudenstein et Roth	feval = 163 fmin = 1.73e-17	feval = 143 fmin = -8.48e-16
	feval = 172 fmin = 1.62e-15	feval = 173 feval = -1.15e-18

Chaque ligne du tableau correspond à une exécution d'un problème en partant d'un point

de départ identique. On note `feval` le nombre d'évaluations de la fonction objectif durant l'exécution, et `fmin` la plus petite valeur de l'objectif trouvée.

Globalement, la performance de ce solveur est assez satisfaisante. Toutefois, on ne peut pas conclure qu'une implémentation est plus efficace qu'une autre. Cette fluctuation de performance dépend principalement de la manière dont est généré le simplexe initiale.

4.2 UOBYQA

Avec ou sans dérivées, les méthodes de région de confiance ont pour principe de construire un modèle de l'objectif qui est moins coûteux à optimiser. On fait confiance en la qualité du modèle dans une zone qu'on appelle région de confiance. Lorsqu'on minimise le modèle construit m_f au sein de cette région de confiance, on trouve un nouveau point s. On calcule ensuite un ratio r pour déterminer si le point s est accepté, si le modèle doit être amélioré et si la taille de la région de confiance doit changer. Le chapitre 11 du livre A. R. Conn et al. (2008) donne une structure générale des algorithmes de région de confiances en DFO. Dans Han & Liu (2004) on introduit une variante de UOBYQA qui garantie une convergence globale super-linéaire.

4.2.1 Description générale

UOBYQA (M. Powell, 2001) est une méthode de DFO pour les problèmes sans contraintes qui se base sur la construction d'un modèle quadratique par interpolation. Les détails de cette méthode seront expliqués tout au long de cette section. Pour commencer, l'organigramme 4.4 nous décrit les différentes étapes d'UOBYQA et son cheminement général. On commence par donner la fonction objectif à optimiser, un point de départ $x_b \in \mathbb{R}^n$, des paramètres $\rho_{beg}, \rho_{end} > 0$ qui désignent le rayon de la région de confiance au départ et le rayon minimal qui déclenche l'arrêt de l'algorithme. On note ρ le rayon actuel de la région de confiance tel que $\rho_{beg} \geq \rho \geq \rho_{end} > 0$.

Lors de l'initialisation, on commence par construire un ensemble d'interpolation qui comprend à tout moment $m = \frac{(n+1)(n+2)}{2}$ points. La section 4.2.2 donne explicitement les formules pour construire cet ensemble. De plus, l'algorithme assure que ce dernier est bien posé à tout moment au sens de l'interpolation polynomiale. Cette propriété, en plus des caractéristiques de la fonction objectif, fait que le modèle d'interpolation est toujours *fully quadratic* (FQ) (voir la définition 3.2.5) ce qui assure une convergence globale vers un point critique de second ordre. On définit le paramètre $j = 0$. Ce paramètre détermine au début de chaque itération

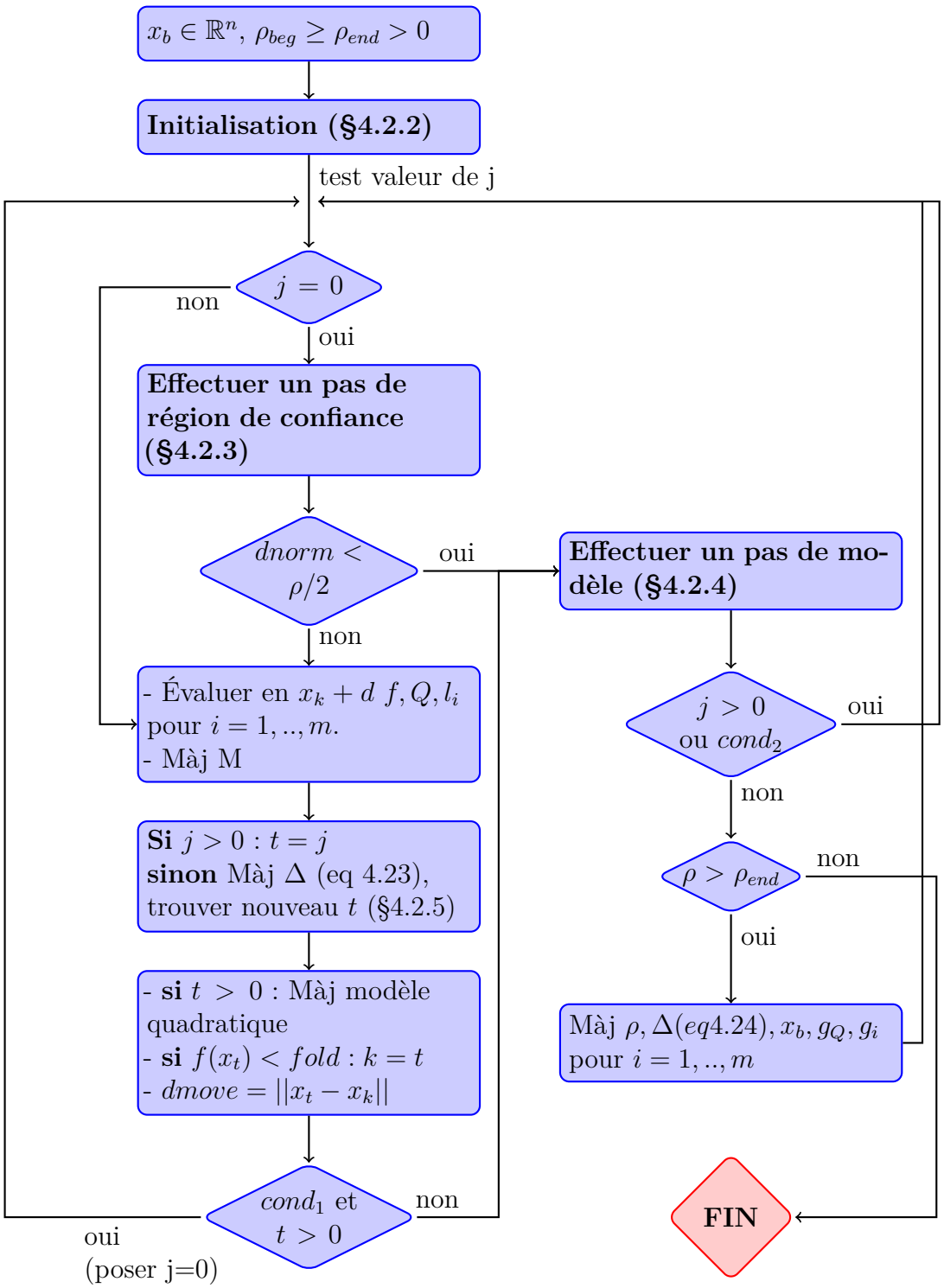

Figure 4.4 Organigramme d'UOBYQA.

si on doit effectuer un pas de région de confiance (voir section 4.2.3) ou si un point a déjà été trouvé pendant l'étape précédente avec un pas de modèle (voir section 4.2.4). Ces deux types de pas introduisent un nouveau point x_{nouv} où on évalue la fonction objectif et le modèle quadratique. On détermine ensuite le paramètre t qui représente l'indice du point de l'ensemble d'interpolation qui sera remplacé par x_{nouv}. L'entier k représente à tout moment l'indice du point qui a la plus petite valeur de l'objectif.

Dans cet algorithme, il y a deux étapes majeures : le pas de région de confiance [1] et le pas de modèle [2]. On résout dans les deux cas un problème de région de confiance. Pour le pas de région de confiance, on cherche à trouver un point qui minimise le modèle quadratique construit dans une région de taille $\Delta \geq \rho$. On teste si le point trouvé est accepté ou non. S'il est accepté, il faut déterminer le point x_t de l'ancien ensemble d'interpolation à remplacer par le nouveau. Le point x_t est celui qui nuit le plus à la qualité du modèle quadratique. Dans le cas contraire, on effectue le pas de modèle. Ce pas aussi résout un problème de région de confiance dont le rayon est ρ, et qui consiste à introduire un nouveau point à l'ensemble d'interpolation afin d'améliorer la qualité du modèle.

Dans l'organigramme 4.4, on a :
$cond_1 = (j > 0)$ ou $(f(x_t) < f_{old})$ ou $(dnorm > 2\rho)$ ou $(dmove > 2\rho)$.
$cond_2 = (j = 0)$ et $(dnorm > \rho)$.

4.2.2 Initialisation

On commence par initialiser le rayon $\Delta = \rho_{beg}$.
À partir du point de départ x_b, on construit un ensemble d'interpolation comprenant $m = \frac{(n+1)(n+2)}{2}$ points de telle sorte que $x_0 = x_b$ et pour $j = 0, .., n-1$

$$x_{2j+1} - x_b = \rho_{beg} e_j \quad \text{et} \quad x_{2j+2} - x_b = \begin{cases} -\rho_{beg} e_j \text{ si } \sigma_j = -1 \\ 2\rho_{beg} e_j \text{ si } \sigma_j = 1, \end{cases} \tag{4.2}$$

où e_j est le j [e] vecteur de la base canonique. On a $\sigma_j = -1$ si $f(x_{2j+1}) > f(x_b)$ ou $\sigma_j = 1$ sinon. On introduit aussi la notation suivante

$$i(p, q) = 2n + p + 1 + \tfrac{1}{2}q(q-1)$$

1. Trust region step
2. Model step

avec $0 \leq p < q \leq n-1$ pour désigner les indices des points entre $2n+1$ et $m-1$. Ainsi, on construit le reste des points d'interpolation de la manière suivante :

$$x_{i(p,q)} = x_b + \rho_{beg}(\sigma_p e_p + \sigma_q e_q). \tag{4.3}$$

Ce choix de points d'interpolation va faciliter le calcul des coefficients du modèle quadratique et des polynômes de Lagrange.

On cherche à construire un modèle quadratique de la forme :

$$Q(x) = c_Q + g_Q^t(x - x_b) + \tfrac{1}{2}(x - x_b)^t G_Q(x - x_b) \tag{4.4}$$

à l'aide des polynômes de Lagrange de degré 2

$$l_j(x) = c_j + g_j^t(x - x_b) + \tfrac{1}{2}(x - x_b)^t G_j(x - x_b). \quad \text{pour } j = 0, ..., m-1. \tag{4.5}$$

Puisque c'est un modèle construit par interpolation, on impose les conditions d'interpolation :

$$Q(x_i) = F(x_i) \quad \text{pour } i = 0, .., m-1. \tag{4.6}$$

Coefficients du modèle quadratique

Les coefficients du modèle quadratique peuvent être déduits des relations précédentes. D'une part,

$$\begin{aligned} F(x_{2j+1}) - F(x_b) &= Q(x_{2j+1}) - Q(x_b) && \text{par (4.6)} \\ &= g_Q^t(x_{2j+1} - x_b) + \tfrac{1}{2}(x_{2j+1} - x_b)^t G_Q(x_{2j+1} - x_b) && \text{par (4.4)} \\ &= \rho_{beg}(g_Q)_j + \tfrac{1}{2}\rho_{beg}^2(G_Q)_{jj} && \text{par (4.2)} \end{aligned}$$

et de la même manière :

$$F(x_{2j+2}) - F(x_b) = \begin{cases} -\rho_{beg}(g_Q)_j + \frac{1}{2}\rho_{beg}^2(G_Q)_{jj} \text{ si } \sigma_j = -1 \\ 2\rho_{beg}(g_Q)_j + 2\rho_{beg}^2(G_Q)_{jj} \text{ si } \sigma_j = 1, \end{cases}$$

ce qui implique que

$$(G_Q)_{jj} = \begin{cases} \dfrac{F(x_{2j}) - F(x_b) + F(x_{2j+1}) - F(x_b)}{\rho_{beg}^2} \text{ si } \sigma_j = -1 \\ \dfrac{2(F(x_{2j}) - F(x_b)) - (F(x_{2j+1}) - F(x_b))}{-\rho_{beg}^2} \text{ si } \sigma_j = 1 \end{cases}$$

$$(g_Q)_j = \frac{F(x_{2j}) - F(x_b) - \frac{\rho_{beg}^2}{2}(G_Q)_{jj}}{\rho_{beg}}.$$

D'après (4.3), (4.4) et (4.6), on a la relation suivante :

$$F(x_{i(p,q)}) = c_Q + \rho_{beg}\left(\sigma_p (g_Q)_p + \sigma_q (g_Q)_q\right) + \tfrac{1}{2}\rho_{beg}^2\left((G_Q)_{pp} + (G_Q)_{qq} + 2\sigma_p\sigma_q (G_Q)_{pq}\right),$$

dont on déduit le terme qui reste $(G_Q)_{pq}$ puisque c'est le seul élément inconnu de l'équation précédente.

Coefficients des polynômes de Lagrange

De la même manière, on calcule les coefficients des polynômes de Lagrange. M. Powell (2001) exploite la structure de l'ensemble des points d'interpolation pour définir explicitement les polynômes de Lagrange. De (4.3) et (4.5), on tire que :

$$l_{i(p,q)}(x) = \tfrac{\sigma_p\sigma_q}{\rho_{beg}^2}(x - x_b)_p (x - x_b)_q \quad \text{pour } 0 \le p < q \le n-1$$

ce qui implique que

$$G_{i(p,q)} = \frac{\sigma_p\sigma_q}{\rho_{beg}^2} \quad \text{et} \quad g_i(p,q) = 0.$$

Soient les polynômes temporaires suivants, pour $i = 0, .., n-1$:

$$\hat{l}_{2i+1}(x) = \frac{(x - x_b)_i (x - x_{2i+1})_i}{(x_{2i} - x_b)_i (x_{2i} - x_{2i+1})_i} \quad \text{et} \quad \hat{l}_{2i+2}(x) = \frac{(x - x_b)_i (x - x_{2i})_i}{(x_{2i+1} - x_b)_i (x_{2i+1} - x_{2i})_i}.$$

D'après (4.2), on peut les réécrire ainsi pour $i = 1, .., n-1$

$$\hat{l}_{2i+2}(x) = \frac{(x - x_b)_i (x - x_{2i})_i}{2\rho_{beg}^2} \quad \text{et} \quad \hat{l}_{2i+1}(x) = \begin{cases} \dfrac{(x - x_b)_i (x - x_{2i+1})_i}{-\rho_{beg}^2} \text{ si } \sigma_i = 1 \\ \dfrac{(x - x_b)_i (x - x_{2i+1})_i}{2\rho_{beg}^2} \text{ si } \sigma_i = -1. \end{cases}$$

Donc

$$(\hat{G}_{2i+2})_{ii} = \tfrac{1}{\rho_{beg}^2}, \quad (\hat{g}_{2i+2})_i = \tfrac{-1}{2\rho_{beg}}, \qquad \begin{cases} (\hat{G}_{2i+1})_{ii} = \tfrac{-2}{\rho_{beg}^2} \text{ et } (\hat{g}_{2i+1})_i = \tfrac{2}{\rho_{beg}} \text{ si } \sigma_i = 1 \\ (\hat{G}_{2i+1})_{ii} = \tfrac{1}{\rho_{beg}^2} \text{ et } (\hat{g}_{2i+1})_i = \tfrac{1}{2\rho_{beg}} \text{ si } \sigma_i = -1. \end{cases} \tag{4.7}$$

Ces polynômes vérifient la propriété $\hat{l}_k(x_i) = \delta_{k,i}$ où $k = 2j+1$ ou $2j+2$ avec $j = 0, .., n-1$, pour tous les entiers $i = 0, .., m-1$ sauf ceux qui s'écrivent sous la forme $i = i(p, j)$ avec $0 \leq p < j$ et $i = i(j, q)$ avec $j < q \leq n-1$.

On considère les polynômes

$$l_k(x) = \hat{l}_k(x) - \sum_{p=0}^{j-1} \hat{l}_k(x_{i(p,j)}) l_{i(p,j)}(x) - \sum_{q=j+1}^{n-1} \hat{l}_k(x_{i(j,q)}) l_{i(j,q)}(x). \tag{4.8}$$

La propriété $l_k(x_i) = \delta_{k,i}$ est maintenant vérifiée pour tous les $i = 0, .., m-1$. D'après (4.7) et (4.8), on déduit les coefficients de polynômes de Lagrange

$$\begin{aligned} G_k &= \hat{G}_k - \sum_{p=0}^{j-1} \hat{l}_k(x_{i(p,j)}) G_{i(p,j)} - \sum_{q=j+1}^{n-1} \hat{l}_k(x_{i(j,q)}) G_{i(j,q)} \text{ et} \\ g_k &= \hat{g}_k - \sum_{p=0}^{j-1} \hat{l}_k(x_{i(p,j)}) g_{i(p,j)} - \sum_{q=j+1}^{n-1} \hat{l}_k(x_{i(j,q)}) g_{i(j,q)} \quad \text{pour } k = 1, .., 2n. \end{aligned}$$

Finalement

$$G_0 = -\sum_{i=1}^{m-1} G_i \text{ and } g_0 = -\sum_{i=1}^{m-1} g_i.$$

Pendant cette étape, on initialise aussi les paramètres :

- $j = 0$ qui détermine à chaque étape si on doit effectuer un pas de région de confiance ou un pas de modèle ;

- $M = 0$ qui est un coefficient lié à l'erreur entre le modèle quadratique et la fonction objectif ;
- $k \in \text{argmin}\{f(x_i), i = 0, .., m-1\}$ qui désigne à tout moment l'indice d'un point d'interpolation qui produit la plus petite valeur de l'objectif.

4.2.3 Résolution du pas de région de confiance

Une fois le modèle quadratique construit et les paramètres bien définis, la première étape est d'effectuer ce pas de région de confiance afin de trouver un point qui minimise le modèle quadratique dans une boule de rayon Δ. Ainsi, le sous-problème qu'on cherche à résoudre est le suivant :

$$(\mathcal{P}) : \begin{cases} \min_{d \in \mathbb{R}^n} Q(x_k + d) \\ \text{s.c } ||d|| \leq \Delta, \end{cases}$$

où Q est de la forme (4.4), i.e.,

$$Q(x_k + d) = Q(x_k) + \nabla Q(x_k)^t d + \frac{1}{2} d^t \nabla^2 Q(x_k) d$$

avec

$$\begin{cases} \nabla Q(x_k) = h_Q = g_Q + G_Q^t (x_k - x_b) \\ \nabla^2 Q(x_k) = G_Q. \end{cases}$$

Theorème 4.2.1. ***Sorensen (1982); Gay (1981)*** d est une solution globale du problème $\mathcal{P}$ si et seulement si $||d|| \leq \Delta$ et il existe $\lambda^* \geq 0$ tel que :

$$(G_Q + \lambda^* I)d = -h_Q \quad \lambda^*(||d|| - \Delta) = 0$$

où $(G_Q + \lambda^* I)$ est semi-définie positive. Si cette matrice est définie positive, alors la solution d est unique.

Soit Λ la matrice diagonale des valeurs propres de G_Q et U une matrice orthonormale de vecteurs propres associés. Ainsi, $G_Q = U \Lambda U^t$.

On applique la procédure de J. Moré & Sorensen (1983) pour trouver la solution $d(\lambda)$ telle que

$$||d(\lambda)|| = ||(G_Q + \lambda I)^{-1} h_Q|| \leq \Delta$$

où λ vérifie les conditions du théorème (4.2.1).

Pour alléger les notations, on introduit la fonction

$$\phi(\lambda) = ||d(\lambda)||_2 = \sqrt{\sum_{i=1}^{n} \frac{\gamma_i^2}{(\lambda_i + \lambda)^2}} \quad \text{où } \gamma_i = (U^t h_Q)_i. \tag{4.9}$$

Le problème est finalement de résoudre l'inéquation

$$\phi(\lambda) \leq \Delta. \tag{4.10}$$

La fonction ϕ est toujours positive. Elle tend vers 0 lorsque λ tend vers $\pm\infty$ et a des pôles lorsque $\lambda = \lambda_i$ où les λ_i sont les valeurs propres de G_Q. Ces pôles sont apparents lorsque $\gamma_i \neq 0$. La figure 4.5 donne un exemple de l'allure de la fonction ϕ.

La solution λ_{sol} va dépendre de la matrice G_Q, et plus précisément de sa plus petite valeur propre notée λ_Q. On fait une étude de cas.

Remarque : Contrairement à M. Powell (2001), on ne traite pas la matrice G_Q pour la rendre tridiagonale avec le procédé de Lanczos. Étant donné que la taille des problèmes résolus ne dépassent pas 30 variables généralement, on se permet d'effectuer une décomposition complète en valeurs / vecteurs propres de G_Q.

G_Q est définie positive (Q est convexe)

Dans ce cas $\lambda_Q > 0$. Soit la valeur $\Delta_{crit} = \phi(0)$ représentée par la ligne rouge de la figure 4.5. Si la relation $\Delta > \Delta_{crit}$ est vérifiée alors

$$\forall \lambda \geq 0 \quad \phi(\lambda) \leq \Delta.$$

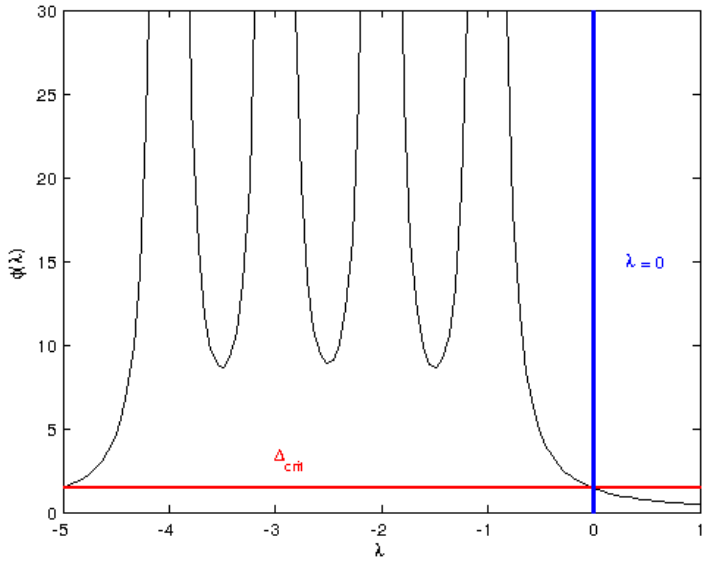

Figure 4.5 Allure de $\phi(\lambda)$ lorsque G_Q est définie positive.

Dans ce cas, on prend $\lambda = 0$ et la solution de $(\mathcal{P})$ est $d = -G_Q^{-1} h_Q$.

Dans le cas $\delta \leq \delta_{crit}$, on peut trouver la valeur de λ qui est solution de l'équation

$$\phi(\lambda) = \Delta$$

par la méthode de Newton.

Soit la fonction

$$\begin{aligned} \psi(\lambda) &= \frac{1}{\phi(\lambda)} \\ &= \frac{1}{\sqrt{\sum_{i=1}^{n} \frac{\gamma_i^2}{(\lambda_i+\lambda)^2}}}. \end{aligned}$$

Son allure plus linéaire que $\phi(\lambda)$ fait que le méthode de Newton converge plus rapidement vers une solution λ_{sol}. Ainsi, l'équation à résoudre devient :

$$\psi(\lambda) = \frac{1}{\Delta} \tag{4.11}$$

avec

$$\psi^{'}(\lambda_k) = \frac{\sum_{i=1}^{n} \frac{\gamma_i^2}{(\lambda_i+\lambda)^3}}{(\sum_{i=1}^{n} \frac{\gamma_i^2}{(\lambda_i+\lambda)^2})^{\frac{3}{2}}}. \tag{4.12}$$

On utilise la méthode de Newton pour trouver la racine de (4.11). On choisit comme point de départ $\lambda_0 = \lambda_{init} + \epsilon$ avec $\lambda_{init} = 0$, $\epsilon = 10^{-3}$ et

$$\lambda_{k+1} = \lambda_k - \frac{\psi(\lambda_k) - \frac{1}{\Delta}}{\psi^{'}(\lambda_k)}.$$

Une fois un λ_{sol} trouvé, la solution du problème de région de confiance $(\mathcal{P})$ est :

$$d = -(G_Q + \lambda_{sol} I)^{-1} h_Q.$$

Cas facile

On parle de cas facile lorsque $\lambda_Q \leq 0$. Cela veut dire que h_Q n'est pas orthogonale à l'espace $\mathcal{E}_Q$ généré par les vecteurs propres associés à la plus petite valeur propre λ_Q. Dans l'expression (4.9), on voit que lorsque $\gamma_Q \neq 0$, la fonction ϕ possède un terme lié à λ_Q. Ce terme est absent de l'expression (4.9) dans le cas difficile.

On cherche ici une solution $\lambda > -\lambda_Q \geq 0$ qui vérifie $\phi(\lambda) = \Delta$. Dans le cas facile, chaque valeur de $\Delta > 0$ a une valeur unique λ_δ associée. Cette valeur est trouvée en appliquant la méthode Newton pour résoudre l'équation (4.11) en commençant de $\lambda_0 = \lambda_{init} + \epsilon$ avec $\lambda_{init} = -\lambda_Q$. Les détails de la résolution sont identiques au cas précédent.

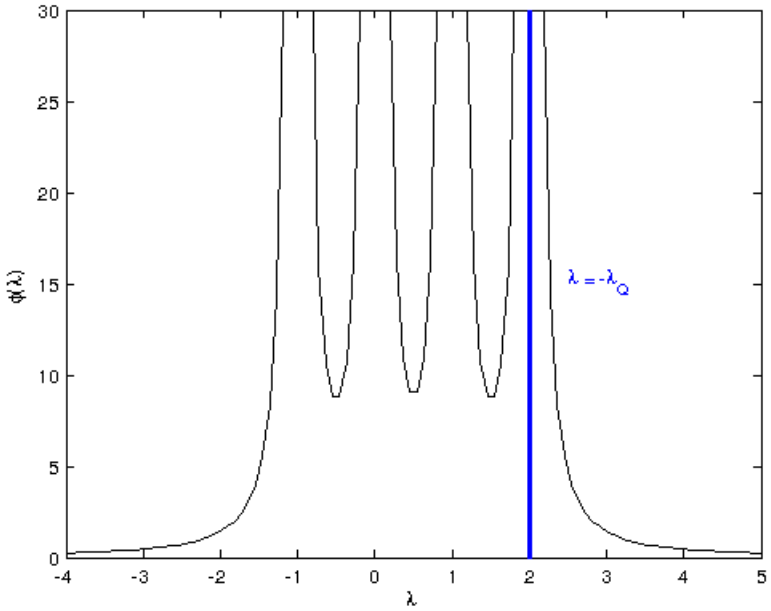

Figure 4.6 Allure de $\phi(\lambda)$ dans le cas facile.

Cas difficile

Le cas difficile est celui où $\lambda_Q \leq 0$ et h_Q est orthogonale à $\mathcal{E}_Q$, i.e. $\gamma_Q = 0$. La figure 4.7 représente un cas difficile où $\lambda_Q = -2$. On voit que pour toute valeur de Δ, on peut trouver un $\lambda > 1$ qui vérifie :

$$\phi(\lambda) = \Delta. \tag{4.13}$$

Cependant, les conditions du théorème 4.2.1 imposent que $\lambda \geq -\lambda_Q = 2$. Le problème se pose lorsque la valeur de Δ est grande. Dans la figure 4.7 c'est le cas pour $\Delta \geq 1.5$.

Soit $\lambda_{sol} = -\lambda_Q$. On considère le vecteur suivant :

$$d = \Delta \frac{u_Q}{||u_Q||},$$

où u_Q est un vecteur propre associé à λ_Q. Puisque $\lambda_Q \leq 0$, u_Q est assurément une direction de descente.

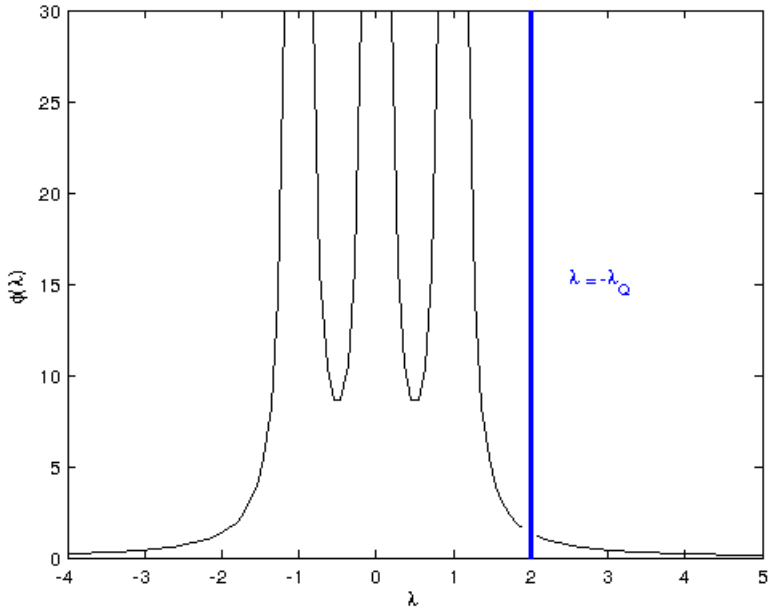

Figure 4.7 Allure de $\phi(\lambda)$ dans le cas difficile.

On a les propriétés suivantes :

- $||d|| \leq \Delta$,
- $\lambda_{sol} > 0$,
- la matrice $(G_Q + \lambda_{sol} I)$ est semi-définie positive,
- $\lambda_{sol}(||d|| - \Delta) = 0$.

Ce vecteur d est une solution acceptable puisqu'elle vérifie les conditions d'optimalité du théorème 4.2.1.

Remarque : il existe plusieurs techniques pour résoudre le problème $\mathcal{P}$. Dans la bibliothèque fournie, le fichier `minimize_quadratic.py` regroupe quelques méthodes pour résoudre ce type problème. La méthode de J. Moré & Sorensen (1983) y est implémenté par exemple ainsi que deux méthodes décrites dans la section suivante. Pour résoudre le problème du pas de région de confiance différemment, il suffit d'implémenter une technique alternative dans ce fichier et modifier la section suivante dans `uobyqa.py` :

```
1 minquad = Min_Quad(hQ, GQ, self.delta, v)
2 d, lbdamin = minquad.more_sorensen()
```

Une fois la solution d trouvée, on sauvegarde la valeur $dnorm = ||d||$, puis on calcule

$$\epsilon = \begin{cases} \frac{1}{2}\rho^2\lambda_Q & \text{si} \quad dnorm < \frac{1}{2}\rho \\ 0 & \text{sinon.} \end{cases} \tag{4.14}$$

4.2.4 Résolution du pas de modèle

Le but de cette étape est d'améliorer la qualité du modèle quadratique. Dans M. Powell (2001), on considère qu'un modèle est adéquat lorsque d'une part, les points d'interpolation se trouvent à une distance raisonnable de la région de confiance. On exprime cette condition par :

$$||x_i - x_k|| \leq 2\rho \quad i = 0, .., m-1.$$

On commence par faire une liste des points qui nuisent à la qualité du modèle par rapport à la propriété précédente

$$\mathcal{J} = \{i : ||x_i - x_k|| > 2\rho\}. \tag{4.15}$$

D'autre part, le modèle est aussi adéquat lorsque l'erreur entre le modèle et la fontion objectif est petite. En supposant que la fonction objectif f possède une dérivée troisième bornée par M, alors le théorème 2 de M. J. D. Powell (2001) pose que l'écart entre l'objectif et le modèle quadratique vérifie :

$$|f(x) - Q(x)| \leq \frac{1}{6}M \sum_{i=0}^{m-1} |l_i(x)|||x - x_i||^3, \quad x \in \mathbb{R}^n. \tag{4.16}$$

Lorsqu'on se place en un point $x \in \mathcal{B}(x_k, \rho)$, le terme lié au point j dans la somme (4.16) est borné par :

$$\frac{1}{6}M \max\{|l_j(x)|||x - x_j||^3 : ||x - x_k|| \leq \rho\}. \tag{4.17}$$

Puisque $x \in \mathcal{B}(x_k, \rho)$, on peut écrire $x = x_k + d$ avec $||d|| \leq \rho$. Par ailleurs, étant donné que x_j est le point le plus éloigné de x_k, on suppose que la distance entre x et x_j est identique à celle entre x_k et x_j. L'expression (4.17) est presque égale à :

$$\frac{1}{6}M||x_j - x_k||^3 \max\{|l_j(x_k + d)| : ||d|| \leq \rho\}. \tag{4.18}$$

Le point x_j est jugé nuisible au modèle si

$$\tfrac{1}{6}M||x_j - x_k||^3 \max\{|l_j(x_k + d)| : ||d|| \leq \rho\} > \epsilon \tag{4.19}$$

où ϵ est la valeur définie en (4.14).

Ainsi, l'étape du pas de modèle consiste d'abord à trouver, le cas échéant, le premier point x_j qui vérifie (4.15) et (4.19) pour le remplacer par un nouveau point qui améliorera la qualité du modèle quadratique.
Pour tester la condition (4.19), il faut résoudre le problème $(\mathcal{L})$ potentiellement plusieurs fois :

$$(\mathcal{L}) : \begin{cases} \max_{d\in\mathbb{R}^n} |l_j(x_k + d)| \\ \text{s.c} \quad ||d|| \leq \rho. \end{cases}$$

Sachant que

$$l_j(x_k) = c_j + g_j^t(x_k - x_b) + \tfrac{1}{2}(x_k - x_b)^t G_j(x_k - x_b),$$

on a $l_j(x_k + d) = l_j(x_k) + h_j^t d + \frac{1}{2}d^t G_j d \quad$ et $h_j = g_j + G_j(x_k - x_b)$.

M. Powell (2001) suggère que pour maximiser $|l_j(x_k + d)|$, on peut, dans un premier temps, maximiser la somme $|h_j^t d| + |d^t G_j d|$. La solution de cette dernière expression est ensuite utilisée pour construire une solution approchée du problème $\mathcal{L}$. On optimise chaque terme à la fois.

Une solution du premier terme est

$$d_1 = \rho \frac{h_j}{||h_j||}. \tag{4.20}$$

Pour la seconde partie, on effectue une décomposition en valeurs / vecteurs propres de G_j. On s'appuie encore une fois sur la petite taille des problèmes traités pour justifier cette opération. Soient λ_{min} et λ_{max}, la plus petite et plus grande valeur propre de G_j et u_{min} et u_{max} des

vecteurs propres associés. On calcule le vecteur

$$d_2 = \begin{cases} \rho \frac{u_{min}}{||u_{min}||} \text{ si } |\lambda_{min}| \geq |\lambda_{max}| \\ \\ \rho \frac{u_{max}}{||u_{max}||} \text{ sinon.} \end{cases}$$

Si d_1 et d_2 sont quasi parallèles, c'est-à-dire $| < d_1, d_2 > | \geq 0.99 \; ||d_1|| \; ||d_2||$, alors une solution approchée d au problème $(\mathcal{L})$ est celle parmi $[d_1, d_2, -d_1, -d_2]$ qui maximise $|h_j^t d + d^t G_j d|$.

Dans le cas contraire, soit le vecteur $v = d_2 - \frac{d_2^t d_1}{||d_1||^2} d_1$. Par construction on a $d_1^t v = 0$. On construit ensuite deux vecteurs u_1 et u_2 qui vérifient les propriétés suivantes :

- $u_1, u_2 \in \text{Vect}(d_1, v)$ qui désigne l'espace vectoriel engendré par d_1 et v,
- $||u_1|| = ||u_2|| = \rho$,
- $u_1^t u_2 = 0$,
- $u_1^t G_j u_2 = 0$.

Si on veut que $u_1, u_2 \in \text{Vect}(d_1, v)$, cela veut dire que

$$\begin{aligned} u_1 &= \alpha d_1 + \beta v \quad \text{et} \\ u_2 &= \gamma d_1 + \delta v. \end{aligned}$$

Le choix des paramètres $\alpha, \beta, \gamma, \delta$ dans le code source du solveur UOBYQA est le suivant

$$\alpha = \frac{\frac{1}{2}(d_1^t G_j d_1 - v^t G_j v) + \sqrt{\frac{1}{4}(d_1^t G_j d_1 - v^t G_j v)^2 + v^t G_j d_1^2}}{(\frac{1}{2}(d_1^t G_j d_1 - v^t G_j v) + \sqrt{\frac{1}{4}(d_1^t G_j d_1 - v^t G_j v)^2 + v^t G_j d_1^2})^2 + v^t G_j d_1^2}$$

$$\beta = \frac{v^t G_j d_1}{(\frac{1}{2}(d_1^t G_j d_1 - v^t G_j v) + \sqrt{\frac{1}{4}(d_1^t G_j d_1 - v^t G_j v)^2 + v^t G_j d_1^2})^2 + v^t G_j d_1^2}$$

$$\begin{aligned} \gamma &= \alpha \\ \delta &= -\beta. \end{aligned}$$

Ces paramètres ont été tirés du code Fortran de UOBYQA et nous ne possédons pas de

justification ou intuition concernant ces choix.

Soit θ_j, la valeur de θ parmi les multiples de $\frac{\pi}{4}$, qui maximise

$$h_j^t u_1 \cos(\theta) + h_j^t u_2 \sin(\theta) + \tfrac{1}{2} u_1^t G_j u_1 \cos(\theta)^2 + \tfrac{1}{2} u_2^t G_j u_2 \sin(\theta)^2.$$

La solution finale choisie est

$$d = \cos(\theta_j) u_1 + \sin(\theta_j) u_2.$$

Remarque : Cette étape est peut aussi être remplacée par une autre méthode de résolution du problème ($\mathcal{L}$). Dans la bibliothèque fournie, on trouve deux implémentations de techniques différentes pour cette étape. La méthode `model_iteration()` exécute les étapes qu'on vient de développer. Une alternative est d'utiliser `model_iteration_original()` qui est une implémentation en Python de la fonction `lag_max.f` du logiciel original d'UOBYQA. Dans cette version, on ne fait pas de décomposition en valeurs/vecteurs propres de G_j. On calcule plutôt une approximation suivant une heuristique détaillée dans M. Powell (2001). Le reste de la procédure de résolution est identique à la version implémentée.

On teste les deux alternatives avec 52 problèmes non différentiables. Les figures 4.8, 4.9 et 4.10 représentent les profils de performance et de données obtenus pour $\tau = 10^{-3}$, $\tau = 10^{-5}$ et $\tau = 10^{-7}$. Globalement, la nouvelle version est légèrement plus efficace que la version originale. Dans la figure 4.8 par exemple, on voit que les deux versions peuvent résoudre 80% des problèmes testés avec moins de 20 évaluations de gradient simplexe.

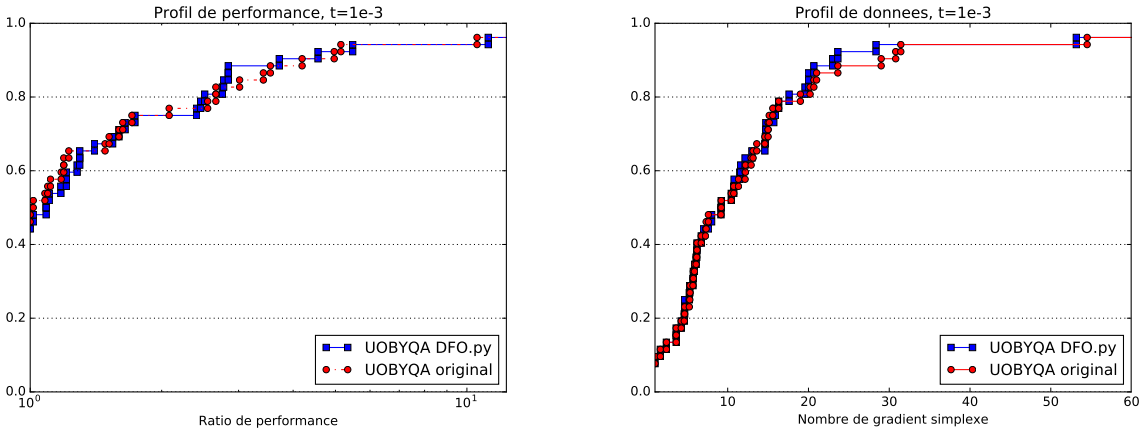

Figure 4.8 Profils de performance et de données pour $\tau = 10^{-3}$.

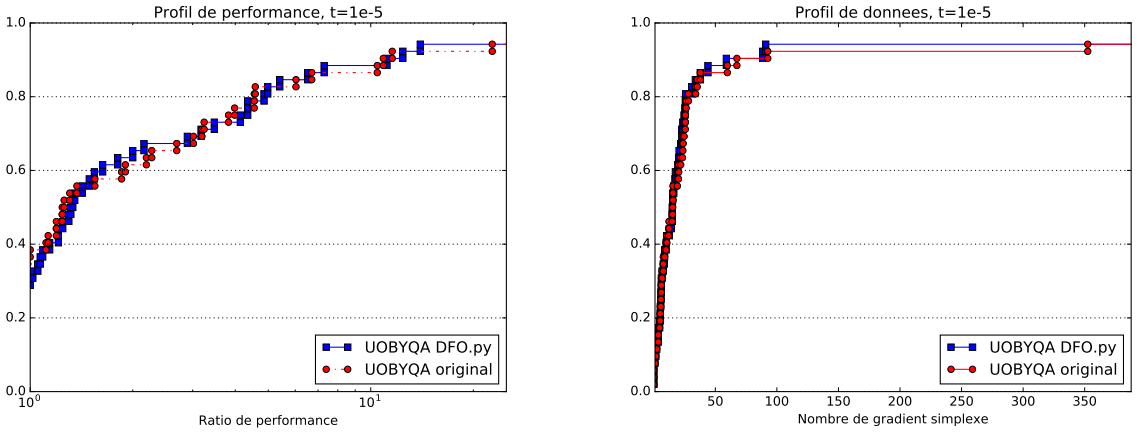

Figure 4.9 Profils de performance et de données pour $\tau = 10^{-5}$.

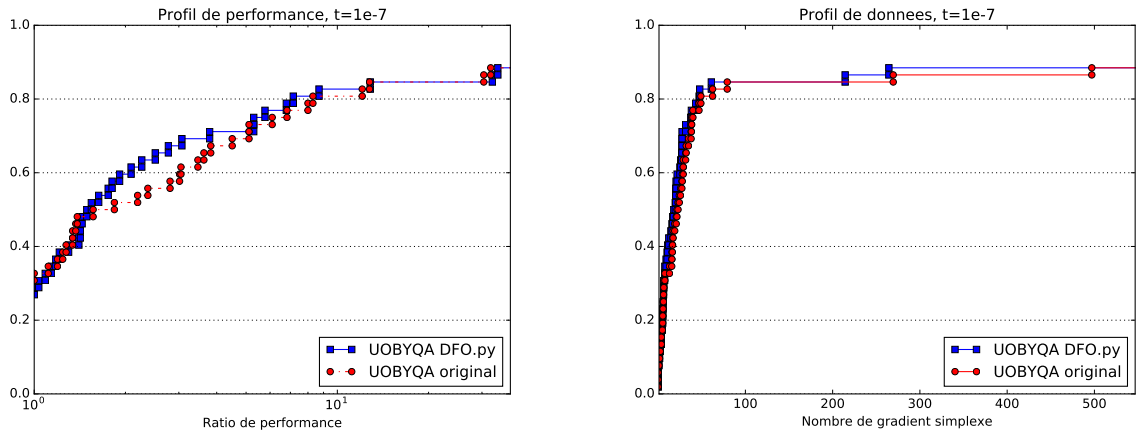

Figure 4.10 Profils de performance et de données pour $\tau = 10^{-7}$.

On pourrait aussi choisir d'exploiter la méthode de résolution du pas de région de confiance pour résoudre les deux problèmes suivants et garder la meilleure des deux solutions

$$\begin{cases} \min\ l_j(x_k + d) \\ \text{s.c} \quad ||d|| \leq \rho \end{cases} \quad \text{et} \quad \begin{cases} \min\ -l_j(x_k + d) \\ \text{s.c} \quad ||d|| \leq \rho. \end{cases}$$

Cette alternative est simple à implémenter et ne nécessite qu'une seule décomposition spectrale.

4.2.5 Calcul du paramètre t

Dans cette section, on explique la procédure pour déterminer l'indice t du point de l'ensemble d'interpolation qui sera remplacé par le nouveau point calculé $x_{nouv} = x_k + d$. Ici d est soit un pas de région de confiance ou un pas de modèle. Arrivé à cette étape, on connaît déjà la valeur de $f(x_{nouv})$. Soient $\mathcal{K}$ et $\hat{x}_k$ tels que $\begin{cases} \hat{x}_k = x_{nouv} \text{ et } \mathcal{K} = \emptyset \text{ si } f(x_{nouv}) < f(x_k) \\ \hat{x}_k = x_k \text{ et } \mathcal{K} = \{k\} \text{ sinon.} \end{cases}$

L'indice t' est celui qui maximise l'expression suivante :

$$|l_i(x_k + d)|\max\{1, ||x_i - \hat{x}_k||^3 \rho^3\}. \tag{4.21}$$

On prend $t = t'$ si $f(x_{nouv}) < f(x_k)$ et si la valeur de (4.21) en t' est strictement supérieure à 1. Dans M. Powell (2001), on explique que ces conditions sont liées aux résultats numériques obtenus.

4.2.6 Mises à jour

Mise à jour du modèle

Les coefficients du modèle quadratique et des polynômes de Lagrange doivent être mis à jour lorsqu'un point d'interpolation x_t est remplacé par un nouveau point $x_{nouv} = x_k + d$.

$$\begin{cases} g_t & \leftarrow g_t/l_t(x_{nouv}) \\ G_t & \leftarrow G_t/l_t(x_{nouv}) \end{cases} \begin{cases} g_i & \leftarrow g_i - l_i(x_{nouv})g_t \\ G_i & \leftarrow G_i - l_i(x_{nouv})G_t \end{cases} \begin{cases} g_Q & \leftarrow g_Q + (f(x_{nouv}) - f(x_k))g_t \\ G_Q & \leftarrow G_Q + (f(x_{nouv}) - f(x_k))G_t. \end{cases}$$

Mise à jour de M

Après avoir évalué les paramètres $f(x_k + d), Q(x_k + d)$ et $l_i(x_k + d)$ pour $i = 1, .., m$, on met à jour le paramètre M qui est noté $\frac{1}{6}M$ dans l'article M. Powell (2001).
M est mis à jour de la manière suivante :

$$M \leftarrow \max\left[M, |Q(x_k + d) - f(x_k + d)| / \sum_{i=0}^{m-1} |l_i(x_k + d)|||x_k + d - x_i||^3\right]. \tag{4.22}$$

Mise à jour de Δ

Le paramètre Δ peut être mis à jour à deux reprises pendant une itération de l'algorithme. L'équation (4.23) donne la formule de mise à jour de Δ lorsqu'on cherche à trouver un nouvel indice t. Soit

$$r = \frac{f(x_k) - f(x_k + d)}{Q(x_k) - Q(x_k + d)},$$

$$\Delta = \begin{cases} \max[\Delta, \frac{5}{4}||d||, \rho + ||d||] \text{ si } r \geq 0,7 \\ \max[\frac{1}{2}\Delta, ||d||] \text{ si } 0,1 < r < 0,7 \\ \frac{1}{2}\Delta \text{ si } r \leq 0,1. \end{cases} \tag{4.23}$$

Δ peut aussi être mis à jour après avoir calculé le nouveau rayon de la région de confiance ρ. On désigne par ρ_{old} l'ancienne valeur du rayon, ainsi la nouvelle valeur de Δ est :

$$\Delta = \max[\tfrac{1}{2}\rho_{old}, \rho]. \tag{4.24}$$

Mise à jour de ρ

On rappelle que ρ_{end} est la valeur minimale que peut atteindre le rayon de la région de confiance. La paramètre ρ est mis à jour de cette manière :

$$\rho = \begin{cases} \rho \text{ si } \rho_{end} < \rho \leq 16\rho_{end} \\ \sqrt{\rho\rho_{end}} \text{ si } 16\rho_{end} < \rho \leq 250\rho_{end} \\ 0.1\rho \text{ si } \rho > 250\rho_{end}. \end{cases} \tag{4.25}$$

Mise à jour de x_b

Après avoir mis à jour ρ et Δ suivant la formule (4.24), x_b doit être mis à jour. Ce changement affecte les gradients du modèle quadratique et des polynômes de Lagrange.
Soit $v = x_k - x_b$, on applique les mises à jour suivantes :

$$\begin{aligned} g_Q &\leftarrow g_Q + G_Q v \\ g_i &\leftarrow g_Q + G_Q v \text{ pour } i = 0, .., m-1. \end{aligned}$$

Puis, on met à jour x_b ainsi :

$$x_b \leftarrow x_k.$$

4.2.7 Tests numériques

Afin de valider ce solveur, on effectue une comparaison entre notre implémentation de l'algorithme de UOBYQA et la version Fortran de M. Powell `UOBYQA.f`. Le tableau (4.2) résume les résultats obtenus pour deux fonctions objectifs et deux points de départ différents pour chaque objectif.

Tableau 4.2 Validation de UOBYQA.

Problème	**Dimension**	**UOBYQA implémenté**	`UOBYQA.f`
Rosenbrock	2	feval = 98 fmin = 1.04e-17	feval = 100 fmin = 7.71e-23
		feval = 503 fmin = 1.04e-17	feval = 192 fmin = 1.28e-24
Hellical	3	feval = 106 fmin = 6.01e-17	feval = 86 fmin = 8.68e-23
		feval = 152 fmin = 2.20e-18	feval = 138 fmin =1.42e-24
Singular	4	feval = 351 fmin = 1.25e-26	feval = 342 fmin = -8.08e-30
		feval = 382 fmin = 5.88e-29	feval = 466 feval = 7.9e-31
Linéaire - rang 1	7	feval = 2000 fmin = 48.65	feval = 2000 fmin = 9.88
		feval = 36 fmin = 54.11	feval = 407 fmin = 9.88
Heart	8	feval = 1094 fmin = 6.6e-03	feval = 348 fmin = 4.72e-17
		feval = 2000 fmin = 12.47	feval = 2000 fmin = 1.8e+09
Linéaire - rang plein	9	feval = 178 fmin = 36.0	feval = 179 fmin = 35.99
		feval = 187 fmin = 36.0	feval = 137 fmin = 35.99
Osbourne -2	11	feval = 2000 fmin = 2.49	feval = 1145 fmin = 4.01e-02
		feval = 78 fmin = 2.06	feval = 1145 fmin = 4.01e-02
Watson	12	feval = 2000 fmin = 2.89e-06	feval = 2000 fmin = 1.3e-08
		feval = 2000 fmin = 2.07e-05	feval = 2000 fmin = 7.36e-07
Brown	15	feval = 625 fmin = 1.67e-19	feval = 358 fmin = 2.04e-19
		feval = 1744 fmin = 4.09e-16	feval = 1907 fmin =1.84e00
Mancino	20	feval = 409 fmin = 1.23e-09	feval = 361 fmin = 7.63e-12
		feval = 635 fmin = 1.72e-09	feval = 628 fmin =5.34e-10

Le tableau (4.2) montre que l'efficacité d'un solveur par rapport à un autre dépend du problème et du point de départ choisi. Cette différence est due principalement aux choix suivis lors des résolutions des deux sous problèmes de régions de confiance.

4.3 MADS

`MADS.py` est une implémentation simplifiée de l'algorithme MADS (Audet & Dennis, Jr., 2006) qui est une méthode de recherche directe. C'est une généralisation de l'algorithme GPS (Torczon, 1997), lui même une généralisation de la recherche par coordonnées : CS (Fermi & Metropolis, 1952). Dans la suite de cette section, on présente les différents aspects théoriques de `MADS.py` et les différences entre les versions disponibles, à savoir, la recherche par coordonnées avec $2n$ et $n+1$ directions et Ortho-MADS. Dans Audet & Dennis, Jr. (2006), on explique que MADS possède une hiérarchie de résultats de convergence dépendamment de la différentiabilité de la fonction objectif et de la structure de son ensemble de définition.

4.3.1 Description générale

`MADS.py` est un algorithme itératif qui peut effectuer deux étapes à chaque itération, la recherche et la sonde. L'étape de recherche est optionnelle et libre à l'utilisateur. Cette liberté vient du fait que la preuve de convergence ne dépend pas de la stratégie employée durant cette étape. La recherche sert à effectuer une exploration globale et génère un ensemble fini de points dans l'espoir de trouver une meilleure valeur de l'objectif. Si la recherche échoue alors on effectue une sonde. La sonde est une exploration locale qui se base sur des directions de recherche générées selon la version de `MADS.py` utilisée. Dans la suite, on s'intéresse uniquement à l'étape de sonde et aux différents types de directions de recherche implémentés.

Les points manipulés par `MADS.py` se trouvent à tout moment sur un maillage dont la taille est déterminée par le paramètre Δ_k^m à l'itération k. Le point courant x_k est celui qui a la meilleure valeur de l'objectif pour l'instant. Au début de chaque étape de sonde, on détermine un ensemble de points où évaluer l'objectif qu'on nomme ensemble de sonde :

$$P_k = \{x_k + \Delta_k^p d \quad d \in D_k\}. \tag{4.26}$$

On évalue la fonction objectif en ces points. Si aucun n'arrive à améliorer l'objectif alors cette étape est considérée comme un *échec*, sinon c'est un *succès* et dans ce cas on se place sur le meilleur point trouvé. Finalement, la taille du maillage est mise à jour en fonction du succès

de l'étape de sonde et de la version de `MADS.py` utilisée. Par exemple, la taille du maillage ne change pas après un *succès* dans la recherche par coordonnées, contrairement à Ortho-MADS qui agrandit la taille maillage dans ce cas.

4.3.2 Pseudo-code

Globalement, on peut résumer la structure de `MADS.py` de la manière qui suit

0. Initialisation Déterminer un, ou un ensemble de points de départ et $\Delta_0^m > 0$.
1. La recherche (facultative) Déterminer un ensemble fini de points selon la stratégie choisie et évaluer la fonction objectif en ces points. Si cette étape est un *succès* alors aller à l'étape **3**. Sinon aller à l'étape **2**.
2. La sonde Calculer l'ensemble P_k en fonction des directions de recherches disponibles et évaluer l'objectif en ces points. Le nombre de points de l'ensemble P_k qu'on évalue effectivement dépend de la stratégie de sonde choisie parmi les suivantes :

- la stratégie par défaut évalue tous les points dans un ordre prescrit puis choisi celui avec la plus petite valeur de l'objectif ;
- la stratégie opportuniste évalue les points dans le même ordre mais s'arrête dès qu'une meilleure valeur est trouvée ;
- la stratégie dynamique sauvegarde la dernière direction de succès et évalue les points dans l'ordre du dernier succès. Elle s'arrête aussi à la première amélioration trouvée. Dans `MADS.py` ; c'est la stratégie par défaut.

3. Mises à jour Mettre à jour la taille du maillage Δ_k^m.
Incrémenter $k = k + 1$ et aller à l'étape **1**.

On peut choisir les directions de recherche parmi les options suivantes

```
directions = CS2nDirections()        # 2n directions de recherche
directions = CSnp1Directions()       # n+1 directions de recherche
directions = ORTHOnp1Directions(n)   # n+1 directions de recherche
```

Notons ici que `MADS.py` ne possède pas de version Ortho-Mads à $2n$ directions bien que c'est la version par défaut dans NOMAD. Cela devrait être ajouté dans les travaux futures. Le choix des directions de recherche, en plus de la mise à jour de la taille du maillage, différencie une version de `MADS.py` d'une autre. On décrit par la suite les différentes manières de générer ces directions.

Recherche par coordonnées

CS à $2n$ directions

La recherche par coordonnées utilise toujours les $2n$ mêmes directions de recherche cartésiennes. Ainsi, on a toujours un ensemble de $2n$ points à sonder qui se trouvent à une distance Δ_k^m de x_k :

$$P_k = \{x_k \pm \Delta_k^m e_i \mid i = 1, .., n\}.$$

Lors d'un *échec*, on réduit la taille du maillage. Généralement, on prend : $\Delta_{k+1} = \frac{1}{2}\Delta_k$.
Par contre, la taille du maillage reste la même après un *succès*.

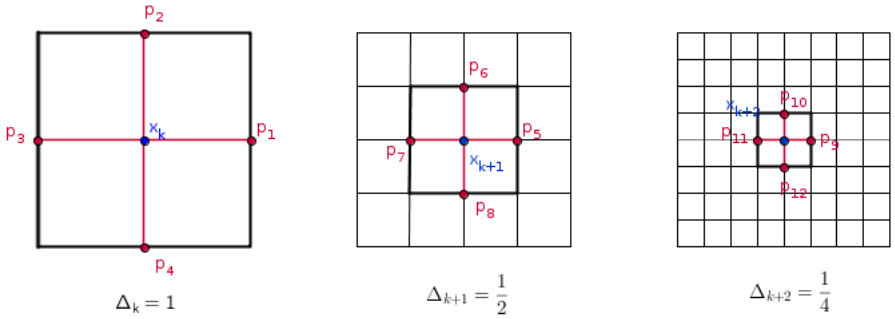

Figure 4.11 Exemple d'exécution de la recherche par coordonnées.

CS à $n+1$ directions

Cette version utilise toujours $n+1$ directions de recherche définies comme tel

$$D = \left\{ e_1, .., e_n, -\sum_{i=1}^{n} e_i \right\}. \tag{4.27}$$

Donc

$$P_k = \{x_k \pm \Delta_k^m d \mid d \in D\}.$$

Le reste de l'algorithme est identique à la version à $2n$ directions.

GPS

À la différence de la recherche par coordonnées, l'ensemble des directions de recherches de GPS peut changer à chaque itération. L'ensemble P_k contient aussi $n+1$ points à sonder au lieu de $2n$. Ces points se trouvent tous à une distance Δ_k^m de x_k :

$$P_k = \{x_k + \Delta_k^m d \mid d \in D_k\},$$

où $D_k \subset D$ est une matrice dont les colonnes sont des directions de recherche qui forment une base positive. Cela veut dire que $\forall x \in \mathbb{R}^n, \exists \lambda_1, .., \lambda_m \in \mathbb{R}^+ \; x = \sum_{j=1}^{m} \lambda_i d_i$. Et D est une matrice qui est définie au début de l'algorithme et qui reste fixe tout au long de l'exécution.

Lorsque la sonde réussit à trouver un meilleur point, la taille du maillage est agrandie. On prend généralement $\Delta_{k+1} = 2\Delta_k$.

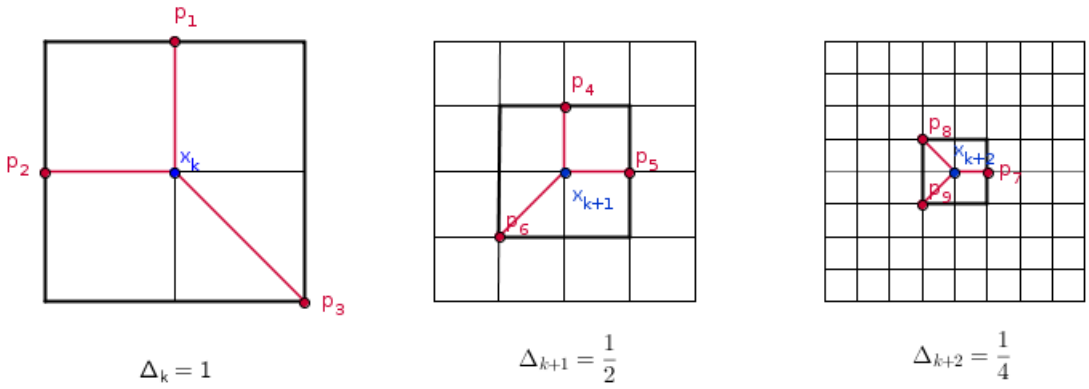

Figure 4.12 Exemple d'exécution de GPS.

MADS

En plus du paramètre de la taille du maillage Δ_k^m, MADS introduit le paramètre de sonde Δ_k^p tel que $\Delta_k^m \leq \Delta_k^p$ à chaque itération, et Δ_k^m est réduit plus rapidement que Δ_k^p. De manière générale, on impose

$$\Delta_k^m = O((\Delta_k^p)^2). \tag{4.28}$$

Cette notation signifie que Δ_k^p est dominé par Δ_k^m. En d'autres termes :

$$\exists N, C > 0 \mid \forall k > N, \Delta_k^m \leq C\sqrt{\Delta_k^p}.$$

L'ensemble des points à sonder s'écrit de la même manière que celui de GPS

$$P_k = \{x_k + \Delta_k^m d \quad | \quad d \in D_k\} \text{ avec } ||\Delta_k^m d_k|| \approx \Delta_k^p.$$

Cette fois D_k n'est pas un sous ensemble de D, mais chaque $d \in D_k$ peut s'écrire sous la forme d'une combinaison entière positive des colonnes de D. Dans la figure 4.13, on voit que plus la taille du maillage s'affine, plus on a de directions de recherche possibles. Plus formellement, Audet & Dennis, Jr. (2006) mettent en place un algorithme qui fait en sorte que l'ensemble des directions normalisées devient asymptotiquement dense sur la sphère unité.

Il existe plusieurs implémentations de MADS en fonction du choix de la matrice D des directions :

- LT-MADS qui génère des directions à l'aide de matrices triangulaires inférieures aléatoires (Audet & Dennis, Jr., 2006),
- Ortho-MADS qui génère des directions orthogonales quasi-aléatoires (Abramson, Audet, Dennis, Jr., & Le Digabel, 2009).

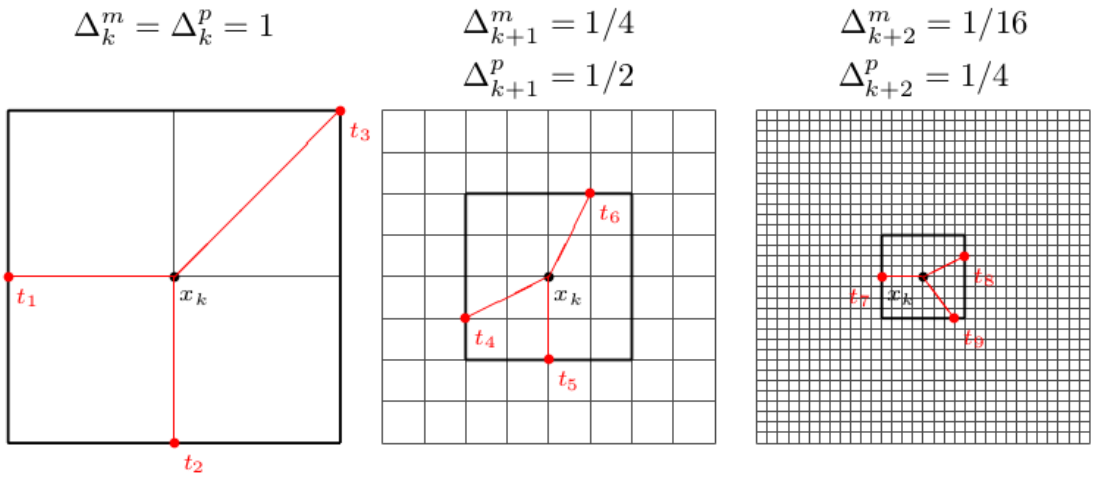

Figure 4.13 Exemple d'exécution de MADS.(© Le Digabel (2014))

Dans `MADS.py`, on implémente une version d'Ortho-MADS.

Dans l'article Abramson et al. (2009), on explique que la génération des directions de recherche d'Ortho-MADS se fait en trois étapes. On commence par utiliser la séquence quasi aléatoire

de Halton pour générer une suite de vecteurs (u_t) dense dans la sphère unité. Cette séquence est ajustée de la manière suivante :

$$q_{t,l} = \frac{2u_t - e}{||2u_t - e||} \text{ où } e \text{ est le vecteur unité.}$$

Finalement, on calcule la matrice de Householder associée à $q_{t,l}$, soit

$$H = ||q_{t,l}||^2 I - 2q_{t,l}q_{t,l}^t.$$

Par construction cette matrice est orthogonale et les directions de recherche sont prises des colonnes de $D_k = [H, -H]$.

Finalement, si l'étape de sonde est un *succès* alors la taille du maillage Δ_k^m est agrandie, sinon elle est rétrécie. Par exemple, on peut suivre la règle suivante

$$\Delta_{k+1}^m = \begin{cases} \frac{1}{4}\Delta_k^m \text{ si } \textit{échec}, \\ 4\Delta_k^m \text{ sinon.} \end{cases} \tag{4.29}$$

Comparaison

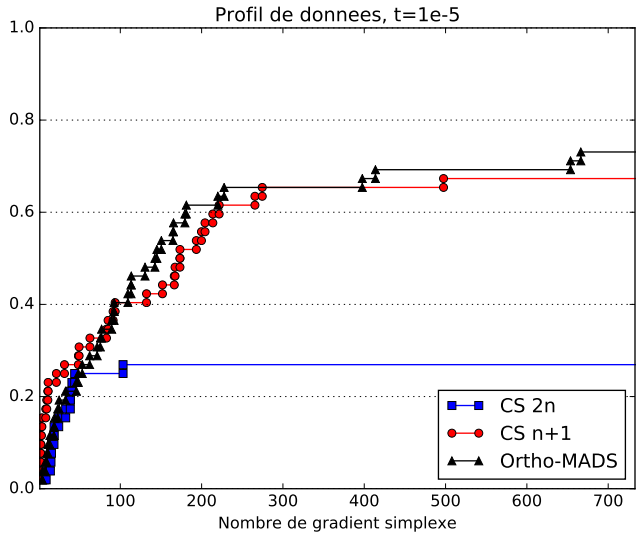

Figure 4.14 Comparaison entre les directions de recherche de `MADS.py`.

La figure 4.14 est un profil de données destiné à comparer `MADS.py` selon le type de directions de recherche employé. On reprend les 52 problèmes utilisés non différentiables utilisés pour la comparaison des versions de UOBYQA. Globalement, la version ortho-MADS est plus efficace que les deux autres alternatives. Elle permet de résoudre jusqu'à 75% des problèmes tandis que la recherche par coordonnées à $n+1$ directions ne dépasse pas 65% des problèmes testés. Finalement, la recherche par coordonnées à $2n$ directions est la moins efficace puisqu'elle résout moins de 30% des problèmes.

4.3.3 Tests numériques

Afin de valider ce solveur, on effectue une comparaison entre notre implémentation de l'algorithme de MADS et celle du solveur `NOMAD`. Le tableau (4.3) résume les résultats obtenus pour deux fonctions objectifs et deux points de départ différents pour chaque objectif.

Tableau 4.3 Validation de MADS.

Problème	**Mads implémenté**	`NOMAD`
Rosenbrock	feval = 40 fmin = 24.19	feval = 260 fmin = -0.87
	feval = 67 fmin = 1	feval = 337 fmin = 3.55
Singular	feval = 119 fmin = 6	feval = 1860 fmin = 0.0292
	feval = 151 fmin = 121	feval = 809 feval = 8.09e-4

Le tableau 4.3 montre que notre implémentation de l'algorithme de MADS a de moins bonnes performances que celle du solveur `NOMAD` puisque les solutions trouvées par ce dernier sont toujours plus optimales, malgré un plus grand nombre d'évaluations de la fonction objectif. Ces résultats sont expliqués par le choix très simplifié des directions de recherche dans notre implémentation.

CHAPITRE 5 CONCLUSION

La bibliothèque finale, nommée `LIBDFO.py`, ainsi que le module `Profiles.py` sont disponibles à l'adresse suivante : `https://github.com/PythonOptimizers/LIBDFO.py`.

5.1 Synthèse des travaux

`LIBDFO.py` est une bibliothèque en libre accès qui regroupe quelques outils et méthodes destinés à l'optimisation sans dérivées. Sa particularité est son implémentation modulaire qui permet de manipuler et modifier simplement les algorithmes de DFO afin de développer plusieurs versions différentes de la même méthode ou d'en créer de nouvelles en branchant les modules voulus.

Cette bibliothèque fournit des outils que l'on utilise dans le cadre des méthodes de recherche directe ou celles à base de modèles quadratiques construits par interpolation. Ces outils, ainsi que les aspects théoriques sur lesquels ils s'appuient sont présentés tout au long de ce mémoire. On présente aussi trois algorithmes de DFO qui sont déjà implémentés dans `LIBDFO.py` : Nelder & Mead (1965), MADS.py — qui inclut une implémentation de l'algorithme MADS (Audet & Dennis, Jr., 2006) — font partie des méthodes de recherche directe ; et UOBYQA (M. Powell, 2001) qui est une méthode qui se base sur des modèles. Chacun de ces algorithmes peut être décliné en plusieurs versions en fonction des options choisies.

Parallèlement, on présente le module `Profiles.py` qui permet de tracer les profils de performance et de données, tels qu'introduits dans J. J. Moré & Wild (2009), afin de comparer l'efficacité de plusieurs solveurs qui ne sont pas uniquement destinés à la DFO.

5.2 Améliorations futures

Le module `LIBDFO.py` présenté est la première version distribuée. Il reste évidemment plusieurs aspects de la DFO qui n'ont pas encore pu être couverts durant cette maîtrise.

Premièrement, la bibliothèque ne traite pas les problèmes avec contraintes ce qui limite les applications pratiques que l'ont peut traiter. Il existe des algorithmes de DFO qui traitent les problèmes avec contraintes. Ces algorithmes doivent être ajoutés au fur et à mesure à notre

bibliothèque. Pour cela, on peut s'aider des différents modules déjà présents. Par exemple, pour certaines méthodes, on peut commencer par implémenter une stratégie de barrière extrême dans un premier temps afin de traiter les contraintes simples.Ensuite, les méthodes de recherche directe n'implémentent pas l'étape de recherche globale. On peut penser à utiliser les outils déjà présents afin de construire des modèles quadratiques de l'objectif durant cette étape, ou implémenter d'autres méthodes qui utilisent des fonctions substituts par exemple. Pour les méthodes qui se basent sur les modèles, une amélioration possible est d'introduire d'autres types de modèles, par régression par exemple, afin de pouvoir comparer l'effet du choix du modèle sur l'efficacité d'un solveur en particulier.

La bibliothèque `LIBDFO.py` n'est pas encore adaptée aux méthodes évolutionnaires qui sont assez populaires dans le domaine de DFO. Au moins un algorithme de cette classe de méthodes doit être ajouté à cette bibliothèque non seulement pour couvrir plus de classes de méthodes, mais aussi pour s'en servir au niveau des comparaisons entre algorithmes. Il existe aussi d'autres types de solveurs de DFO comme les méthodes stochastiques ou des solveurs hybrides qui ne sont pas représentés dans notre bibliothèque. Le traitement des problèmes bi-objectif n'est pas encore possible. Finalement, on peut aussi penser à implémenter des solveurs parallélisables.

Comme évoqué plus tôt, le travail sur ce module est loin d'être fini. Le résultat présenté en ce moment doit être vu comme une première base qui devra être alimentée et maintenue au fur et à mesure.

RÉFÉRENCES

Abramson, M., Audet, C., Couture, G., Dennis, Jr., J., Le Digabel, S., & Tribes, C. (s. d.). *The NOMAD project.* Software available at `https://www.gerad.ca/nomad/`. Consulté sur `https://www.gerad.ca/nomad/`

Abramson, M., Audet, C., Dennis, Jr., J., & Le Digabel, S. (2009). Orthomads : A deterministic mads instance with orthogonal directions. *SIAM Journal on Optimization*, *20*(2), 948–966.

Adams, B., Bohnhoff, W., Dalbey, K., Eddy, J., Eldred, M., Gay, D., ... Swiler, L. (2009). Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis : Version 5.0 user's manual. *Sandia National Laboratories, Tech. Rep. SAND2010-2183*.

Audet, C., & Dennis, Jr., J. (2006). Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, *17*(1), 188–217.

Audet, C., & Kokkolaras, M. (2016). Blackbox and derivative-free optimization : theory, algorithms and applications. *Optimization and Engineering*, *17*(1), 1–2. Consulté sur `http://dx.doi.org/10.1007/s11081-016-9307-4` doi: 10.1007/s11081-016-9307-4

Audet, C., Le Digabel, S., & Tribes, C. (2009). *NOMAD user guide* (Rapport technique N° G-2009-37). Les cahiers du GERAD. Consulté sur `https://www.gerad.ca/nomad/Downloads/user_guide.pdf`

Berghen, F. V., & Bersini, H. (2005, September). CONDOR, a new parallel, constrained extension of powell's UOBYQA algorithm : Experimental results and comparison with the DFO algorithm. *Journal of Computational and Applied Mathematics*, *181*, 157-175.

Booker, A., Dennis, J., Frank, P., Serafini, D., Torczon, V., & Trosset, M. (1999). A rigorous framework for optimization of expensive functions by surrogates. *Structural optimization*, *17*(1), 1–13.

Choi, T. D., Eslinger, O. J., Gilmore, P., Patrick, A., Kelley, C. T., & Gablonsky, J. M. (1999). Iffco : Implicit filtering for constrained optimization, version 2. *Rep. CRSC-TR99*, *23*.

Conejo, P., Karas, E., & Pedroso, L. (2014). A trust-region derivative-free algorithm for constrained optimization. *Optmization methods and software*, 1126-1145.

Conn, A., & Le Digabel, S. (2013). Use of quadratic models with mesh-adaptive direct search for constrained black box optimization. *Optimization Methods and Software*, *28*(1), 139–158.

Conn, A., Scheinberg, K., & Vicente, L. (2005). Geometry of sample sets in derivative free optimization. part ii : polynomial regression and underdetermined interpolation.

Conn, A., Scheinberg, K., & Vicente, L. (2008). Geometry of sample sets in derivative-free optimization : polynomial regression and underdetermined interpolation. *IMA journal of numerical analysis*, *28*(4), 721–748.

Conn, A. R., Scheinberg, K., & Toint, P. L. (1998a). A derivative free optimization algorithm in practice. , *48*, 3.

Conn, A. R., Scheinberg, K., & Toint, P. L. (1998b). A derivative free optimization algorithm in practice. In *Proceedings of 7th aiaa/usaf/nasa/issmo symposium on multidisciplinary analysis and optimization, st. louis, mo* (Vol. 48, p. 3).

Conn, A. R., Scheinberg, K., & Vicente, L. N. (2008). *Introduction to derivative-free optimization*. SIAM, Philadelphia, USA : MPS-SIAM Optimization series.

Currie, J., & Wilson, D. I. (2012a). Opti : lowering the barrier between open source optimizers and the industrial matlab user. *Foundations of computer-aided process operations, Savannah, Georgia, USA*, 8–11.

Currie, J., & Wilson, D. I. (2012b, 8–11 January). OPTI : Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User. In N. Sahinidis & J. Pinto (Eds.), *Foundations of Computer-Aided Process Operations.* Savannah, Georgia, USA.

Custódio, A., & Vicente, L. N. (2007). Using sampling and simplex derivatives in pattern search methods. *SIAM Journal on Optimization*, *18*(2), 537–555.

Dorigo, M., Birattari, M., & Stützle, T. (2006). Ant colony optimization. *Computational Intelligence Magazine, IEEE*, *1*(4), 28–39.

Fermi, E., & Metropolis, N. (1952). Numerical solution of a minimum problem. *LANL Unclassified Report LA–1492*.

Finkel, D., & Kelley, C. (2004). Convergence analysis of the direct algorithm. *North Carolina State University, Center for Research in Scientific Computation and Department of Mathematics*.

Fletcher, R. (1965). Function minimization without evaluating derivatives—a review. *The Computer Journal.*(8), 33-41.

Fletcher, R., & Powell, M. (1965). A rapidly convergent descent method for minimization. *The Computer Journal*(4), 308-313.

Gay, D. M. (1981). Computing optimal locally constrained steps. *SIAM Journal on Scientific and Statistical Computing*, *2*(2), 186–197.

Goldstein, A. A. (1967). *Constructive real analysis*. New York : Harper & Row.

Golub, G., & Loan, C. V. (2012). *Matrix computations - third edition.* Bald. more and London : The Johns Hopkins University Press.

Gould, N., Lucidi, S., Roma, M., & Toint, P. (1999). Solving the trust-region subproblem using the lanczos method. *SIAM Journal on Optimization*, *9*(2), 504–525.

Gramacy, R., & Le Digabel, S. (2011). *The mesh adaptive direct search algorithm with treed gaussian process surrogates.* Groupe d'études et de recherche en analyse des décisions.

Han, L., & Liu, G. (2004). On the convergence of the uobyqa method. *J. Appl. Math and Computing*(1-2), 125-142.

Hansen, N. (2006). The cma evolution strategy : a comparing review. In *Towards a new evolutionary computation* (pp. 75–102). Springer.

Higham, N. (2002). *Nmsmax.* `http://www.applied-mathematics.net/optimization/CONDORdownload.html`.

Hooke, & Jeeves. (1961). "direct search"solution of numerical and statistical problems. *Journal of the ACM (JACM)*, *8*(2), 212–229.

Hough, P. D., Kolda, T. G., & Torczon, V. (2001). Asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Scientific Computing*, *23*(1), 134–156. Consulté sur `http://epubs.siam.org/sam-bin/dbq/article/36582` doi: 10.1137/S1064827599365823

Hunter, J. D. (2007). Matplotlib : A 2d graphics environment. *Computing In Science and Engineering*, *9*(3), 90–95.

Huyer, W., & Neumaier, A. (1999). Global optimization by multilevel coordinate search. *Journal of Global Optimization*, *14*(4), 331–355.

Hwang, C. (1988). Simulated annealing : theory and applications. *Acta Applicandae Mathematicae*, *12*(1), 108–111.

Jones, D. R., Schonlau, M., & Welch, W. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, *13*(4), 455–492.

K. Holmström, K. (2001). Practical optimization with the tomlab environment in matlab..

Kolda, T., Lewis, R., & Torczon, V. (2006). Stationarity results for generating set search for linearly constrained optimization. *SIAM Journal on Optimization*, *17*(4), 943–968.

Le Digabel, S. (2011). Algorithm 909 : NOMAD : Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software*, *37*(4), 1–15.

Le Digabel, S. (2014). *Direct search methods.* University Lecture.

Lewis, R. M., Torczon, V., & Trosset, M. W. (2000). Direct search methods : then and now. *Journal of computational and applied mathematics*(124), 1126-1145.

Liuzzi, G. (2015). *Derivative-free library.* Consulté sur `http://www.dis.uniroma1.it/~lucidi/DFL/`

Lougee-Heimer, R. (2003). The common optimization interface for operations research : Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, *47*(1), 57–66.

Marazzi, M., & Nocedal, J. (2002). Wedge trust region methods for derivative free optimization. *Mathematical programming*, *91*(2), 289–305.

Marsden, Alison, L., Feinstein, A, J., Taylor, & A, C. (2008). A computational framework for derivative-free optimization of cardiovascular geometries. *Computer Methods in Applied Mechanics and Engineering*, *197*(21), 1890–1905.

Mason, A., & Dunning, I. (2010). Opensolver : open source optimisation for excel. In *Proceedings of the 45th annual conference of the orsnz* (pp. 181–190).

McKay, M., Beckman, R., & Conover, W. (1979). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*.

Morgan, S. L., & Deming, S. N. (1974). Simplex optimization of analytical chemical methods. *Analytical chemistry*, *46*(9), 1170–1181.

Moré, J., & Sorensen, D. C. (1983). Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*(3), 553-572.

Moré, J. J., & Wild, S. M. (2009). Benchmarking derivative-free optmization algorithms. *SIAM J. Optimization*(20), 172-191.

Nelder, J., & Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*.

Patrick, A. (2000). Implicit filtering for constrained optimization and applications to problems in the natural gas pipeline industry.

Plantenga, T. D. (2009). Hopspack 2.0 user manual [Manuel de logiciel].

Powell, M. (1994). A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis* (pp. 51–67). Springer.

Powell, M. (2001). Uobyqa : unconstrained optimization by quadratic approximation. *Mathematical Programming*(3), 555-582.

Powell, M. (2006). The newuoa software for unconstrained optimization without derivatives. , 255–297.

Powell, M. (2009). The bobyqa algorithm for bound constrained optimization without derivatives. *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*.

Powell, M. J. D. (2001). On the lagrange functions of quadratic models that are defined by interpolation*. *Optimization Methods and Software*, *16*(1-4), 289–309.

Rios, L. M., & Sahimidis, N. V. (2012). Derivative-free optimization : a review of algorithms and comparison of software implementations. *J Glob Optim*.

Scheinberg, K., & Toint, P. L. (2010). Self-correcting geometry in model-based algorithms for derivative-free unconstrained optimization. *SIAM J. Optim*(20), 3512–3532.

Smith, K. W. (2015). *A guide for python programmers*. O'Reilly Media.

Sorensen, D. C. (1982). Newton's method with a model trust region modification. *SIAM Journal on Numerical Analysis*, *19*(2), 409–426.

Talgorn, B., Le Digabel, S., & Kokkolaras, M. (2014). Problem formulations for simulation-based design optimization using statistical surrogates and direct search. In *Asme 2014 international design engineering technical conferences and computers and information in engineering conference* (pp. V02BT03A023–V02BT03A023).

Teodorović, D., & M. Dell'Orco, M. (2005). Bee colony optimization–a cooperative learning approach to complex transportation problems. In *Advanced or and ai methods in transportation : Proceedings of 16th mini–euro conference and 10th meeting of ewgt (13-16 september 2005).–poznan : Publishing house of the polish operational and system research* (pp. 51–60).

Torczon, V. (1997). On the convergence of pattern search algorithms. *SIAM Journal on optimization*, *7*(1), 1–25.

Vaz, A., & Vicente, L. (2009). Pswarm : a hybrid solver for linearly constrained global derivative-free optimization. *Optimization Methods & Software*, *24*(4-5), 669–685.

Wen-Ci, Y. (1979). Positive basis and a class of direct search techniques. *Scientia Sinica, Special Issue of Mathematics*, 53 - 67.

Wild, S., Regis, R., & Shoemaker, C. (2008). Orbit : Optimization by radial basis function interpolation in trust-regions. *SIAM Journal on Scientific Computing*, *30*(6), 3197–3219.

Wild, S. M., Sarich, J., & Schunck, N. (2015). Derivative-free optimization for parameter estimation in computational nuclear physics. *Journal of Physics G : Nuclear and Particle Physics*, *42*(3), 034031.

Wild, S. M., & Shoemaker, C. (2011). Global convergence of radial basis functions trust region derivative-free algorithms. *SIAM J. Optimization*(3), 761-781.

Wright, M. H. (1995). Direct search methods : Once scorned, now respectable. *Dundee Biennial Conference in Numerical Analysis*, 191-208.