Rapport de Stage de 2ème année

Formation Ingénieur
Option Mathématiques Appliquées

Du 02/04/2024 au 30/08/2024

Etat de l'art des algorithmes d'optimisation

Octave Tougeron

Tuteur école :
Anthony Nouy

Tuteur entreprise :
Gilles Vogt

Remerciements : Je remercie mon tuteur Gilles Vogt pour m'avoir encadré durant ce stage qui représente pour moi un premier contact avec l'ingénierie. Je souhaite également remercier toute l'équipe Analyse Scientifique d'Ingeliance qui m'a chaleureusement accueilli et a pu m'aider dans certaines situations. L'état d'esprit amical de l'équipe m'a grandement aidé à m'intégrer et à trouver mes marques rapidement. Je souhaite également remercier l'équipe du bureau d'étude du site de Mérignac, avec qui j'ai passé d'agréables moments (mon bureau était dans leur open-space). Enfin je remercie Kamel-Walid Maachou, autre stagiaire chez Ingéliance à Lyon, qui m'a fait grandement profiter de son expérience sur le logiciel *salome meca* et m'a permis d'avancer rapidement.

Table des matières

Table des figures

List of Algorithms

Liste des tableaux

1 Introduction

L'optimisation est un domaine fondamental et omniprésent dans les sciences et l'ingénierie, visant à trouver les meilleures solutions possibles à des problèmes complexes. Que ce soit dans la conception de systèmes industriels, l'analyse de données, la planification logistique ou même dans des applications quotidiennes comme les trajets les plus courts, l'optimisation joue un rôle crucial pour améliorer l'efficacité, économiser des ressources et atteindre des performances optimales.

À son essence, l'optimisation implique la recherche et la sélection des conditions idéales parmi un ensemble de possibilités. Cela peut se traduire par la minimisation des coûts, la maximisation des bénéfices, la réduction du temps d'exécution, ou encore la gestion efficace des ressources disponibles. Les problèmes d'optimisation peuvent varier en complexité, allant de simples calculs mathématiques à des systèmes dynamiques impliquant des milliers de variables et de contraintes.

Au regard les enjeux sociétaux et climatiques d'aujourd'hui, l'optimisation a ainsi un rôle à jouer qui peut être non-négligeable. Ingéliance a coutume d'utiliser des méthodes d'optimisation et s'inscrit donc dans ces perspectives.

1.1 Ingeliance

Ingéliance est une société d'ingénierie de l'industrie, qui fournit ses services à divers grands groupes de différents domaines tels que l'aérospatial, la défense, ou le nucléaire. L'entreprise possède 3 divisions majeures : Analyse Scientifique (AS), Développement Système (DS) et Solutions Industrielles (SI). Le groupe possède 20 agences réparties en France et à l'étranger pour environ 700 collaborateurs et un chiffre d'affaire de plus de 50 millions d'euros.

J'ai effectué mon stage dans la première division, au sein d'une équipe de 40 docteurs et ingénieurs, qui sont répartis entre des pôles de mécanique des structures (avec notamment la dynamique rapide ou la thermomécanique) et de mécanique des fluides (avec les études aérodynamiques et hydrodynamiques). Cette division est spécialisée dans le domaine de la simulation numérique. J'ai été encadré par Gilles Vogt, ingénieur docteur en calcul électromagnétique et en dynamique rapide.

1.2 Contexte et objectifs du stage

Ingéliance opère depuis un certain temps des optimisations sous contraintes dans divers domaines et divers projets. Forts de la connaissance d'un grand nombre d'algorithmes d'optimisation, l'arrivée et le développement récents de méthodes

d'optimisation pour les problèmes MINLPs (Mixed Integer Linear Programming) ont ouvert de nouvelles portes vers des optimisations plus rapides, plus précises. Ingéliance souhaitait donc élargir son champ de connaissance dans ce domaine, et c'est là que mon stage intervient.

La problématique principale du stage et de l'optimisation en général en simulation numérique est le nombre d'appels au modèle. En effet, les calculs (2D ou 3D) sont relativement longs (de l'ordre de la minute, de l'heure, voire du jour ou de la semaine), donc on cherche toujours à minimiser ce nombre d'appels.

Les objectifs du stage sont multiples. Dans un premier temps, enrichir les connaissances globales de l'entreprise sur les algorithmes pour les MINLPs. A terme, l'idéal serait de déterminer le ou les algorithmes à mettre en place et à privilégier dans les métiers d'Ingéliance. On explore donc plusieurs voies dans ce stage : les MINLPs, la comparaison entre optimisation sous contrainte et optimisation par pénalisation, et l'optimisation multi-objectif. Les applications industrielles étant très souvent un mélange de MINLP et de problèmes multi-objectifs, à terme les algorithmes recherchés doivent être capable d'aborder les deux problèmes en même temps.

1.3 Déroulement du stage

Le stage a débuté par une étape de bibliographie sur les algorithmes MINLP, puis sur les algorithmes multi-objectif. J'ai ensuite réalisé des tests et des comparaisons entre algorithmes pour essayer de déterminer des catégories pertinentes pour la mécanique des structures en particulier.

Dans un second temps, j'ai réalisé une encapsulation complète d'une chaine de calcul et d'optimisation par simulation numérique via la prise en main d'un logiciel de simulation. J'ai pu réaliser des tests grandeur nature (ou du moins à une échelle plus adaptée que les tests purement mathématiques). Enfin, j'ai réalisé des travaux complémentaires sur divers sujets : parallélisation des algorithmes, reprise de calculs déjà faits, optimisation de composites. Le but étant de permettre à Ingéliance de réutiliser ce que j'avais fait, et d'obtenir une plus large connaissance dans les sujets qui peuvent venir lorsqu'on fait de l'optimisation.

2 Optimisation mono-objectif

2.1 Les MINLPs

2.1.1 Optimisation sous-contrainte

En conception de structures, l'optimisation permet de trouver les valeurs des variables qui rendent un certain objectif minimal (ou maximal). Prenons l'exemple simple de la conception d'un avion :

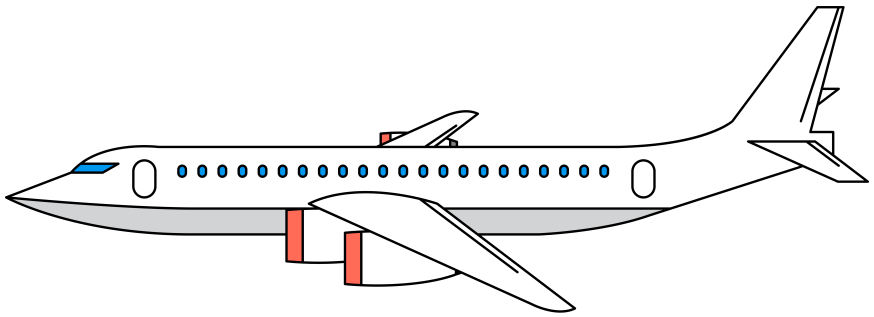


FIGURE 1 – Schéma d'avion

On cherche à :
— Minimiser la masse de l'avion (objectif)
— Avoir au moins 150 places et pouvoir voler (contraintes)
— On peut jouer sur la longueur, la taille des ailes ou des réacteurs... (variables)

Ceci représente donc un problème d'optimisation, qui est sous contraintes car on lui impose des conditions (pour que le projet soit physiquement logique et faisable).

Un tel problème se traduit mathématiquement comme suit :

$$\begin{cases} \min_{x \in X}(f(x)) \\ \text{tel que } g_j(x) \leq 0 \text{ pour } j = 1, \ldots, m \\ \text{et } h_i(x) = 0 \text{ pour } i = 1, \ldots, p \end{cases} \tag{1}$$

Bien souvent, on travaille dans $X \subseteq \mathbb{R}^N$ qui représente l'espace de vie de nos variables. La fonction f représente l'**objectif**, les fonctions g_j les **contraintes d'inégalité** et les fonctions h_i les **contraintes d'égalité**.

Il existe un grand nombre de méthodes pour résoudre les problèmes dits classiques d'optimisation, où les fonctions du problèmes sont "agréables", typiquement lorsqu'elles sont de classe $\mathcal{C}^2$. Parmi elles, on peut citer (voir [15]) :

— La méthode du simplexe ;
— La méthode du point intérieur / extérieur ;
— La méthode du gradient projeté ;
— et bien d'autres...

Néanmoins, nous ne nous focaliserons pas sur ces méthodes, elles sont très documentées et suffisamment connues pour mériter une explication plus approfondie dans ce stage. Le problème de ces méthodes est qu'elles s'appuient bien souvent sur des caractéristiques des fonctions observées (classe, convexité, linéarité...) que nous n'avons pas vraiment lorsque nous réalisons de l'optimisation dans un contexte industriel.

2.1.2 Problèmes MINLPs

Les problèmes MINLPs (Mixed Integer Non Linear Programming) sont des problèmes mathématiques NP-difficiles dû à la complexité de leur résolution. Leur formulation est très similaire à celle d'un problème d'optimisation sous contraintes classique :

$$\begin{cases} \min_{x \in X}(f(x)) \\ \text{tel que } g_j(x) \leq 0 \text{ pour } j = 1, \ldots, m \\ \text{et } h_i(x) = 0 \text{ pour } i = 1, \ldots, p \\ x_k \in \mathbb{Z} \text{ pour } k \in I \ , \ I \subseteq \mathbb{N} \end{cases} \tag{2}$$

On considère donc un certain nombre de variables entières ou binaires parmi nos variables. En plus de cela, on ne fait absolument aucune hypothèse sur les fonctions, en particulier il est très probable qu'elles soient non-linéaires. Classiquement, on classe les MINLPs dans diverses catégories en fonction de leurs caractéristiques mathématiques (convexité de l'objectif, linéarité ou non des contraintes...). C'est ce qui est fait par Martin Schmidt ici [1]. Ces classifications nous intéressent peu car il est généralement très complexe de voir si les objectifs sont convexes dans nos cas industriels. Certaines traductions entre caractères mathématiques et situations physiques seront faites toutefois plus tard.

1. https://www.lamsade.dauphine.fr/poc/sites/default/files/minlp-compact-course.pdf

2.2 État de l'art pour les algorithmes MINLPs

Dans cette section, nous allons nous intéresser aux algorithmes MINLPs principaux que j'ai pu découvrir au cours de mes recherches.

2.2.1 Méthodes mathématiques de base

Les algorithmes de résolution de MINLPs sont multiples et découlent de diverses méthodes inspirées de phénomènes naturels (souvent pour les algorithmes multi-objectifs), ou de méthodes mathématiques. Ici, on décrit 4 procédés qui sont très utilisés dans ces algorithmes et qui permettent d'avoir un aperçu technique du fonctionnement des algorithmes.

- Méthode de relaxation

La relaxation est le procédé qui consiste à élargir le plan d'expérience et le domaine de recherche des solutions. Autrement dit, on supprime un certain nombre de contraintes (ou bien le fait d'avoir de rechercher des variables entières, ou bien on enlève une équation ou on la linéarise...) pour rendre le problème plus simple. Pour les MINLPs, la méthode est très bien décrite dans [25]. Le plus souvent, on utilise une relaxation d'un MINLP vers un NLP classique. Dans ce cas-là, le problème (2) se réécrit simplement :

$$\begin{cases} \min_{x \in X}(f(x))) \\ \text{tel que } g_j(x) \leq 0 \text{ pour } j = 1, \ldots, m \\ \text{et } h_i(x) = 0 \text{ pour } i = 1, \ldots, p \end{cases} \tag{3}$$

Ce procédé est une relaxation continue, car on s'affranchit des contraintes entières. On pourrait aussi relaxer le problème en retirant une des contraintes ; ceci permet de simplifier le problème pour pouvoir le résoudre de manière plus facile. Les trois méthodes suivantes s'inspirent de cela pour trouver les solutions de MINLPs.

- Branch and Bound

Le branch and Bound s'appuie sur des relaxations progressives des MINLPs en NLPs (Non Linear Programming), et sur des disjonction de cas en fonction des diverses solutions trouvées à chaque étapes. [4] explique en détail cette méthode.

Algorithm 1 Algorithme de branch and bound

Data : Un problème d'optimisation P
Result : La solution optimale à P
Step 0. Relaxer le problème initial de manière continue, initialiser la solution optimale à l'infini, et initaliser une liste de noeuds (qui contient juste le problème initial).

```
while la liste n'est pas vide do
    sélectionner et retirer un noeud N de la liste
    if la borne inférieure de N est meilleure que la valeur optimale then
        explorer les enfants de N
        foreach enfant E de N do
            calculer la borne inférieure pour E par relaxation
            if la borne de E est meilleure que la valeur optimale then
                ajouter E à la liste
            end
        end
        mettre à jour la valeur optimale avec la meilleure solution trouvée
    else
        élaguer N
    end
end
```

Un exemple se trouve en Annexe 7.9, pour mieux comprendre les bases de cet algorithme.
Cette méthode est très rapide si le problème est assez peu non-linéaire. La majorité des algorithmes MINLPs se basent sur le Branch and Bound et ses variantes.

- Outer Approximation (OA)

Cette méthode consiste à faire évoluer des bornes supérieures et inférieures des valeurs de l'optimum afin de les faire converger l'une vers l'autre, de manière à se rapprocher de la valeur optimale. Ceci passe par une linéarisation au premier ordre des fonctions qui ne sont pas linéaires, afin de se ramener à un problème MILP (Mixed-Integer Linear Programming). On peut itérer de cette manière jusqu'à ce que les bornes supérieures et inférieures soient égales. Concrètement :

Algorithm 2 Algorithme de l'Outer Approximation

Data : Un problème d'optimisation P
Result : La solution optimale à P
Step 0. On pose $Z_U = +\infty$, $Z_L = -\infty$ et on initialise arbitrairement les variables entières, et on pose k = 1 ;
while $Z_U > Z_L$ **do**

On résout alors le problème relaxé, on obtient une solution
Avec la solution, on actualise Z_U (valeur de l'objectif pour la solution trouvée). On reprend alors le problème initial et on le transforme en MILP par linéarisation des contraintes autour du point trouvé.
On trouve une nouvelle solution, qui nous permet d'actualiser Z_L (valeur de l'objectif pour cette nouvelle solution trouvée).
On actualise $k = k + 1$
Enfin on repart des solutions entières pour de nouveau recommencer la boucle

end

Un exemple se trouve en Annexe (7.8) pour bien comprendre le principe.

- Extended Cutting Plane

L'algorithme de l'Extended Cutting Plane (ECP) est assez similaire à l'OA. Il part des propriétés suivantes :

— Si g est une fonction convexe $\mathcal{C}^1$, alors $g_i(x^*, y^*) + \nabla g_i(x^*, y^*)^T \begin{pmatrix} x - x^* \\ y - y^* \end{pmatrix} \leq g_i(x, y)$
— Si $\max_i\ g_i(x^*, y^*) \leq 0$, alors $\forall i,\ g_i(x^*, y^*) \leq 0$

L'algorithme se décline comme suit :

Algorithm 3 Algorithme ECP

Data : Un problème MINLP
Result : Le minimum global du problème
Étape 0 : Poser $k = 1$. Retirer toutes les contraintes non-linéaires pour arriver à un MILP
Étape 1 : Résoudre le MILP
Étape 2 : Vérifier si la solution respecte toutes les contraintes non-linéaires
if *toutes les contraintes sont respectées* **then**
    Le minimum global est trouvé, arrêter l'algorithme
else
    Étape 3 : Identifier au moins une des contraintes non-linéaires non respectées
    Linéariser la contrainte la moins respectée, l'ajouter au MILP
    Poser $k = k + 1$
    Revenir à l'Étape 1
end

2.2.2 Algorithmes d'optimisation MINLP

A partir de ces méthodes (principalement, il existe d'autres méthodes bien sûr mais qui sont beaucoup moins utilisées), des algorithmes ont été implémentés dans différents langages de programmation pour résoudre les MINLPs.

Dans cette section, j'ai décidé de détailler différents algorithmes que j'ai pu tester, qui sont pour certains basés sur les méthodes de base vues précédemment et qui ont pour pour but de résoudre des problèmes mono-objectifs (pas forcément des MINLPs). Pour d'autres, ce sont des **algorithmes stochastiques** (marqués **S**). Un algorithme stochastique d'optimisation est une méthode qui utilise des éléments de hasard pour trouver des solutions optimales ou quasi-optimales à des problèmes complexes. Contrairement aux algorithmes déterministes, ces algorithmes intègrent des variables aléatoires ou des processus probabilistes dans leur fonctionnement, permettant une exploration plus diversifiée de l'espace de recherche. Cette approche est particulièrement utile pour éviter les minima locaux et pour résoudre des problèmes où les caractéristiques du paysage de recherche sont inconnues ou trop complexes pour être explorées exhaustivement.

Il existe par ailleurs beaucoup d'autres algorithmes que je ne cite pas ici, à retrouver sur le site des ressources MINLPs [2].

2. https://www.minlp.org/resources/index.php

Algorithme	Description	S
Simulated Annealing	Algorithme du recuit simulé, inspiré du recuit en métallurgie. Il est basé sur une exploration empirique de l'espace ([7]).	S
Tabou	Algorithme tabou de base, qui explore simplement les voisinages des points en enregistrant les lieux déjà explorés pour trouver un optimum ([1]).	
B-BB	Issu du solveur Bonmin, implémente une méthode de Branch and Bound ([16]).	
B-OA	Issu du solveur Bonmin, implémente une méthode de Outer Approximation.	
B-Hyb	Issu du solveur Bonmin, implémente une méthode mixte entre B-OA et B-BB.	
B-ECP	Issu du solveur Bonmin, implémente une méthode de Extended Cutting Plane.	
Cobyla	Construit des approximations linéaires successives de la fonction objective et des contraintes via un simplexe de d+1 points, et optimise ces approximations dans une région de confiance à chaque étape (COBYLA).	
IPOPT	Implémente une méthode de recherche locale de points intérieurs satisfaisant (IPOPT).	
GACO	Algorithme des colonies de fourmis ([22]). Voir un exemple de l'algorithme 4	S
DE	Pour évolution différentielle, algorithme évolutionnaire basé sur la génétique ([24]).	S
PSO	Particule Swarm Optimization, utilise un essaim de particules associées à des vitesses qui vont "pointer" vers la meilleure des particules à chaque itération ([19]).	S
TLBO	Teaching Learning Based Optimization, sélectionne dans une population le meilleur individu et le désigne comme enseignant, et les autres individus apprennent suivant deux phases : enseignement et apprentissage ([26]).	S

TABLE 1 – Description des algorithmes et leur caractère stochastique

Ces premiers algorithmes sont pour la plupart open source ou bien peuvent facilement être implémentés à la main (c'est le cas de PSO ou du recuit simulé par exemple). Ci-dessous, on liste avec peu de détails les solveurs qui se trouvent

implémentés dans la bibliothèque spéciale *AMPL*[3].

— **BARON** : algorithme basé sur une méthode de Branch and Reduce (variante du Branch and Bound) (BARON)
— **CPLEX** : solveur développé par la société ILOG pour la résolution de problèmes linéaires.
— **GUROBI** : basé sur du Branch and Bound.
— **SCIP** : englobe un certain nombre de solveurs pour tous types de problèmes.

Autres solveurs d'*AMPL* (très peu testés ou utilisés dans notre étude car [18] montre qu'ils sont moins performants) :

— **LINDO**
— **CBC**
— **HIGHS**

3. https://amplpy.ampl.com/en/latest/

Algorithm 4 Algorithme des Colonies de Fourmis (ACO) pour un problème d'optimisation

Data : Un problème d'optimisation représenté par un graphe de villes et de distances

Result : Le chemin optimal trouvé par la colonie de fourmis

Étape 0 : Initialiser les phéromones $\tau_{ij}(0)$ sur toutes les arêtes (i, j) du graphe

Étape 1 : for *chaque fourmi* k **do**

Placer la fourmi sur un noeud de départ aléatoire

for *chaque ville* i *visitée par la fourmi* k **do**

Calculer la probabilité de déplacement $p_{ij}^k(t)$ vers chaque ville $j \in J_i^k$:

$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in J_i^k} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta} & \text{si } j \in J_i^k \\ 0 & \text{sinon} \end{cases}$$

où $\eta_{ij} = \frac{1}{d_{ij}}$, et d_{ij} est la distance entre les villes i et j

Sélectionner la prochaine ville j selon la probabilité $p_{ij}^k(t)$

Déplacer la fourmi vers la ville j et mettre à jour la liste des déplacements possibles J_i^k

end

Évaluer la qualité du trajet $T^k(t)$ effectué par la fourmi k et calculer la longueur $L^k(t)$

end

Étape 2 : for *chaque arête* (i, j) **do**

Mettre à jour les phéromones :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t)$$

où $\Delta\tau_{ij}(t) = \sum_{k=1}^m \Delta\tau_{ij}^k(t)$, et

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{si } (i,j) \in T^k(t) \\ 0 & \text{sinon} \end{cases}$$

m est le nombre de fourmis, ρ est le paramètre d'évaporation, et Q est un paramètre de réglage

end

Étape 3 : Vérifier les conditions de terminaison (par exemple, nombre maximal d'itérations ou convergence des solutions)

if *la condition de terminaison est satisfaite* **then**

Retourner le meilleur chemin trouvé et arrêter l'algorithme

else

Réinitialiser les phéromones si nécessaire pour éviter une convergence prématurée

Revenir à l'Étape 1

end

2.3 Comparaison des algorithmes mono-objectif

Dans cette section, on décrit le comportement des algorithmes lorsqu'ils sont confrontés à divers cas d'utilisation. Certains d'entre eux n'ont pas été testés car ils ne peuvent pas prendre en charge des MINLPs, ce qui sort un peu du cadre de notre étude dans le cadre où les problèmes réels sont souvent des MINLPs.

2.3.1 Critères mathématiques

Dans [18], une comparaison massive est faite entre différents solveurs MINLPs (appliqués à des problèmes convexes) en introduisant les critères suivants qui permettent de définir des classes de problèmes MINLPs :

— **Continuous relaxation gap** : On cherche à regarder l'écart entre le problème relaxé z^* et le problème d'origine z. $CRG = \frac{|z^* - z|}{\max(|z^*|, 0.001)}$.

— **Degré de non-linéarité du problème** : On fait le rapport entre le nombre de contraintes non linéaires et le nombre de constraintes total. $d = \frac{n_{nonlin}}{n_{tot}}$.

— **Degré de discrétisation du problème** : On fait le rapport entre le nombre de variables discrètes et binaires et le nombre de variables total. Discrete density (Dd) $= \frac{n_{int} + n_{bin}}{n_{tot}}$.

Ces indicateurs sont tous des taux, ce qui permet de classer les problèmes avec comme critère de base $taux \geq 50\%$ (on dira par exemple que le problème est très non-linéaire) ou non. Dans [18], les performances des algorithmes sont regardées en fonction du nombre de problème de la base MINLPlib [4] qui sont résolus en un certain temps donné. Cela permet de tirer pour nos algorithmes mono-objectifs un tableau simplifié de cas d'utilisation (on reprend la notation $\approx$ pour un résultat dans la moyenne, + pour un bon résultat et - pour un mauvais résultat). Nous avons pris l'initiative de tester quelques algorithmes supplémentaires :

4. https://github.com/lanl-ansi/MINLPLib.jl/tree/master/instances/minlp

Algorithm	High CRG	Low CRG	High d	Low d	High Dd	Low Dd
B-ECP	≈	≈	-	+	≈	≈
B-OA	≈	≈	-	+	≈	≈
B-BB	-	+	+	-	+	-
B-Hyb	-	+	-	+	-	+
LINDO	-	+	-	+	-	+
SCIP	+	+	-	+	+	-
BARON	+	+	+	+	+	+
GACO	+	≈	+	≈	+	+
Cobyla	NA	NA	-	+	-	+
Ipopt	NA	NA	+	+	-	+

TABLE 2 – Cas d'utilisation des algorithmes de MINLPs

Grâce aux différentes sources ([9], [10], [5] et [21]), nous avons ensuite pu réaliser le tableau 14 (Annexe 7.6) de manière assez peu précise tout de même, mais qui peut toujours être utile pour se diriger plus vers tel ou tel algorithme. On se concentre sur le domaine de la mécanique des structures dans cette table.

On remarque clairement que ce qui influe majoritairement sur les critères mathématiques sont les caractéristiques des matériaux. Ce sont eux qui font que le milieu est continu ou pas (les matériaux composites par exemple possèdent des caractéristiques discrètes), et ce sont aussi eux qui rendent le problème non-linéaire ou pas (surtout vis-à-vis des contraintes).

2.3.2 Nombre d'appels à l'objectif

La principale difficulté des problèmes auxquels nous sommes confrontés est le nombre d'appels à l'objectif. En effet, les appels sont très couteux en temps (de l'ordre de 5 à 10 minutes), ce qui pousse à vouloir réduire ce nombre d'appels.

On considère donc 3 problèmes issus de la bibliothèque MINLPLib [5] (voir Annexe 7.5) et obtient les résultats table 3.

Bien sûr, ces tests ne sont pas exhaustifs et méritent une analyse attentive. En effet, les problèmes se limitent à 10 variables ici, et le degré de non linéarité reste restreint. Néanmoins, on constate pour la majorité des critères une dominance de l'algorithme BARON (qui pourrait être très utile dans nos cas). Par ailleurs, il est à noter que si on a la valeur NA dans le tableau 3, cela indique une non-terminaison de l'algorithme ou une valeur de l'objectif erronée ou mal estimée. Dans les autres

5. https://github.com/lanl-ansi/MINLPLib.jl/tree/master/instances/minlp

Algo	Pb1	Pb2	Pb3
B-ECP	253	NA	14
B-OA	261	NA	11
B-BB	581	NA	10
B-Hyb	629	NA	10
SCIP	337	11	207
BARON	33	21	12
CPLEX	15	NA	NA
Ipopt	NA	NA	NA
GUROBI	24	20	100
HIGHS	NA	32	852
GACO	200	60	300
PSO	300	NA	300
Recuit simulé	2200	2200	2200

TABLE 3 – Nombre d'appels au modèle en fonction des algorithmes

cas, l'algorithme a convergé vers le bon optimum à 10^{-3} près au moins.

A noter que parmi les algorithmes que nous privilégions se trouvent GACO, car ce n'est pas un solveur commercial. En effet, Ingeliance privilégie pour l'instant solveurs open-source, ce qui élimine d'emblée des algorithmes (BARON, CPLEX...). Néanmoins, je tenais à les mettre pour montrer leurs très bonnes performances. L'algorithme GACO de Pagmo est un des seuls algorithmes non-commerciaux à pouvoir résoudre tous les problèmes posés. Ce qui ressort globalement est que les algorithmes stochastiques convergent dans beaucoup de situations et **ce seront eux que l'on privilégiera**. Il est important de noter que les algorithmes de Bonmin sont également très efficaces, tant que les variables discrètes restent en nombre limité (le problème 2 se résout avec Bonmin si on change quelques variables discrètes en continues).

En conclusion, du à leur structure, je privilégie les algorithmes stochastiques, car ils présentent un bon compromis entre exploration et exploitation. Néanmoins, les algorithmes de Bonmin peuvent se montrer très performants sur des problèmes relativement peu discrets.

3 Optimisation multi-objectif

Dans cette section, on s'applique à comparer les algorithmes multi-objectifs en regardant cette fois-ci uniquement le nombre d'appels aux objectifs. On introduit auparavant 2 méthodes autres que les algorithmes génétiques et purement mutli-objectifs. Les problèmes MINLPs multi-objectifs se formulent comme suit :

$$\begin{cases} \min_{x \in X} (f_1(x), \dots, f_n(x)) \\ \text{tel que } g_j(x) \leq 0 \text{ pour } j = 1, \dots, m \\ \text{et } h_i(x) = 0 \text{ pour } i = 1, \dots, p \\ x_k \in \mathbb{Z} \text{ pour } k \in I \end{cases} \tag{4}$$

3.1 La méthode des poids

Cette méthode consiste simplement à attribuer des poids à chaque objectif puis de faire la somme pondérée et de l'optimiser comme un problème MINLP classique. Cette méthode a pour avantage de tirer partie de la grande efficacité des solveurs classiques et donc d'être très rapides. Formellement, cette méthode se résume comme suit :

$$\begin{cases} \min_{x \in X} (\lambda_1 f_1(x) + \dots + \lambda_n f_n(x)) \\ \lambda_1 + \dots + \lambda_n = 1 \\ + \text{ contraintes et bornes des variables} \end{cases} \tag{5}$$

Comme on se ramène alors à un problème mono-objectif, il est clair que les ordres de grandeurs obtenus en section 3.2 restent valables ici. Les algorithmes à privilégier sont les mêmes. Par ailleurs, en balayant une certaine plage de valeurs pour les poids des objectifs, on arrive à balayer une partie du front de Pareto du problème, même si la zone explorée est souvent restreinte.

3.2 La méthode ϵ-contrainte

Cette méthode consiste à sélectionner un des objectifs et de transformer tous les autres en contraintes en les associant à certains valeurs ϵ qui doivent être choisies empiriquement. Formellement, on obtient :

$$\begin{cases} \min_{x \in X} f_i(x) \\ f_j(x) \leq \epsilon_j \text{ pour } j \neq i \end{cases} \tag{6}$$

De même que pour la méthode précédente, les résultats obtenus sur les algorithmes mono-objectifs sont toujours exploitables ici. Cette méthode reste donc rapide et

précise à condition d'avoir le bon ordre de grandeur pour ϵ. En principe, si on balaie une certaine plage de valeurs pour les ϵ_j, on balaie une certaine zone du front de Pareto, mais dans les cas réels, on ajuste ϵ pour obtenir des solutions physiquement réalisables.

3.3 La méthode du Goal-Programming

Cette méthode consiste à donner une valeur cible à chacun de nos objectifs (notons ces valeurs T_i), puis de minimiser une sorte de fonction de perte qui peut par exemple être la somme des carrés des écarts aux objectifs :

$$\left\{\min_{x\in X} \sum_{i=1}^{i=n} (f_i(x) - T_i)^2 \right. \tag{7}$$

De même que pour les méthodes précédentes, les résultats obtenus sur les algorithmes mono-objectifs sont toujours exploitables ici. Cette technique nécessite tout de même d'avoir un ordre d'idée de ce que pourrait être une solution acceptable du problème, sans quoi les algorithmes pourraient ne pas terminer. Elle reste un bon moyen de vérifier si une solution intuitive est réellement acceptable ou non.

3.4 Algorithmes multi-objectifs

Avant toute chose, on commence par donner des définitions :

- **Point Pareto-optimal :** Point auquel on ne peut pas améliorer un objectif sans en détériorer un autre.
- **Front de Pareto :** Ensemble des points Pareto-optimaux.
- **Domination par une solution :** On dit qu'une solution A domine une solution B si elle fait au moins aussi bien que la solution B sur tous les objectifs, sauf au moins un ou elle fait mieux.
- **Processus de tri non-dominé :** Processus qui consiste à rassembler dans un premier front de Pareto les solutions non-dominées, puis les retirer de l'ensemble des solutions et former un deuxième front sur le même principe, etc.

Dans cette section, on décrit quelques algorithmes que nous avons testés, et qui ont pour but de résoudre les problèmes multi-objectifs MINLPs ou non. De même que pour la section précédente, il en existe beaucoup d'autres qui ne sont pas nécessairement cités, néanmoins quantité d'entre eux peuvent être retrouvés sur la documentation de la bibliothèque Python Mealpy [6].

6. https://mealpy.readthedocs.io/en/latest/pages/models/mealpy.html

— **NSPSO** : Non-dominated Sorting PSO, un PSO pour le multi-objectifs basé sur le critère de Pareto pour itérer vers le minimum.
— **MOTLBO** : TLBO multi-objectifs
— **VEGA** : Cet algorithme considère une population de N individus. Ces individus sont répartis en k sous-populations, chaque valeur de k représentant un objectif à optimiser. À chaque génération, un nombre de sous populations est généré par sélection en fonction de l'objectif k . Ensuite, ces sous-populations sont regroupées pour former une nouvelle population de N individus et les opérateurs de croisement et de mutation sont appliqués. L'avantage de cet algorithme est qu'il est facile à implanter mais son inconvénient majeur est qu'il a tendance à générer des solutions qui excellent dans un seul objectif, sans tenir compte des autres objectifs. Toutes les solutions de moyenne performance, qui peuvent être de très bons compromis, risquent de disparaître avec ce type de sélection. Cet algorithme est l'algorithme de base des algorithmes génétiques, les autres qui suivent sont issus de celui-ci. La figure 2 montre le fonctionnement global.
— **NSGA** : Non-dominated Sorting GA (voir algorithme 5). C'est l'algorithme le plus utilisé en optimisation multi-objectif.
— **MOGA** : Multi-objective GA, les individus sont rangés en fonction du nombre dindividus qui le dominent

$$rang(x_i, t) = 1 + p_i(t)$$

où p_i est le nombre dindividus qui dominent x_i. Donc on calcule le rang de chaque individu, on affecte la fitness par un changement déchelle (scaling) et on suit ensuite le NPGA.
— **MHACO** : Algorithme des fourmis multi-objectifs.
— **MOEAD** : L'algorithme évolutionnaire multi-objectif basé sur la décomposition est un cadre algorithmique polyvalent. Il décompose un problème d'optimisation multi-objectif en plusieurs sous-problèmes d'optimisation mono-objectif puis utilise une heuristique de recherche pour optimiser ces sous-problèmes simultanément et de manière coopérative.

Algorithm 5 NSGA-II (Non-dominated Sorting Genetic Algorithm II)

Input : Taille de la population N, nombre de générations G
Output : Front de Pareto approché
Initialiser la population P_0 de taille N
Évaluer la fitness de chaque individu dans P_0
Appliquer le tri non-dominé pour classer la population en fronts de Pareto F

for $g \leftarrow 1$ **to** G **do**
Sélectionner les parents à partir de P_{g-1} en utilisant le tournoi basé sur la domination et la diversité
Appliquer le croisement et la mutation pour générer une descendance Q_g de taille N
Évaluer la fitness de chaque individu dans Q_g
Combiner les populations P_{g-1} et Q_g en R_g
Appliquer le tri non-dominé pour classer R_g en fronts de Pareto
Construire la nouvelle population P_g en ajoutant les individus des fronts jusqu'à ce que la taille de P_g atteigne N
end
return *Le premier front de Pareto de* P_G

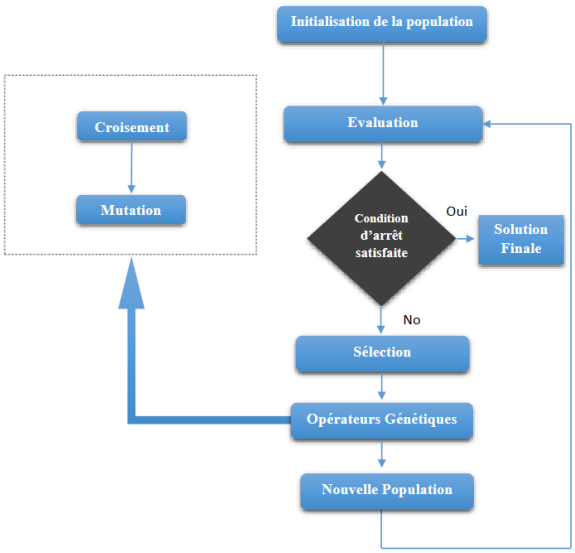


FIGURE 2 – Principe général d'un algorithme génétique [6]

3.5 Comparaison des algorithmes multi-objectifs

Dans cette section, on compare les différents algorithmes multi-objectifs avec les méthodes citées ci-dessus. Pour les 3 méthodes hors algorithmes directement multi-objectif, on utilise le solveur 'SCIP' et on observe ici le nombre d'appels nécessaires au modèle pour résoudre le problème. On regarde également la qualité du front de Pareto obtenu. On s'appuie sur le problème suivant (problème de Srinivas) :

$$\begin{cases} \min\ (9x_1 - (x_2 - 1)^2, (x_1 - 2)^2 + (x_2 - 1)^2 + 2) \\ x_1^2 + x_2^2 \leq 225 \\ x_1 - 3x_2 \leq -10 \\ -20 \leq x_1, x_2 \leq 20 \end{cases} \tag{8}$$

Pour les résultats des algorithmes, on part d'une population de 32 individus. Pour les méthodes des poids, on itère sur un certain nombre de poids différents pour obtenir plusieurs solution ; pour la méthode epsilon, on itère sur un certain nombre de valeurs d'epsilon.

Méthode	Nb d'appels aux objectifs
ϵ-contrainte	1754
Poids	2474
Goal Programming	3000
NSGA2	352
MHACO	352
NSPSO	352

TABLE 4 – Récapitulatif du nombre d'appels aux fonctions par méthode

On voit clairement que les méthodes hors algorithmes multi-objectifs sont gourmandes en coût computationnel, ce qui en fait de mauvaises solutions pour tracer le front de Pareto. De manière générale, obtenir avec ces méthodes un front de Pareto en un nombre d'appels raisonnable n'est pas réalisable ; ici, nous ne sommes qu'à 2 objectifs et 2 variables et on a déjà plus de 300 appels au modèle. On voit d'ailleurs figure 3.5 que le front de Pareto obtenu est très loin d'être global pour ces méthodes, contrairement aux algorithmes multi-objectifs.

Fronts de Pareto de Srinivas

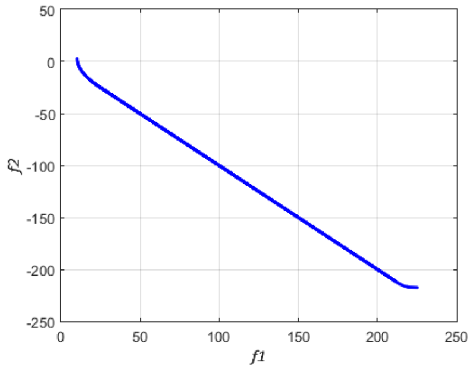


FIGURE 3 – Front réel

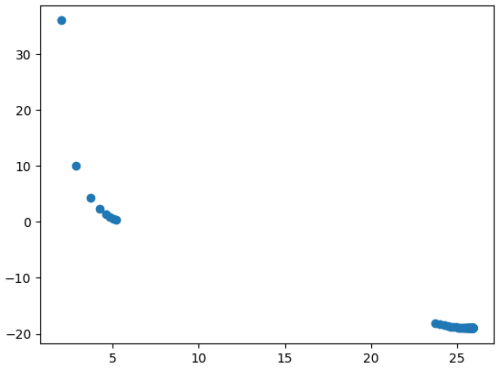


FIGURE 4 – Front méthode des poids

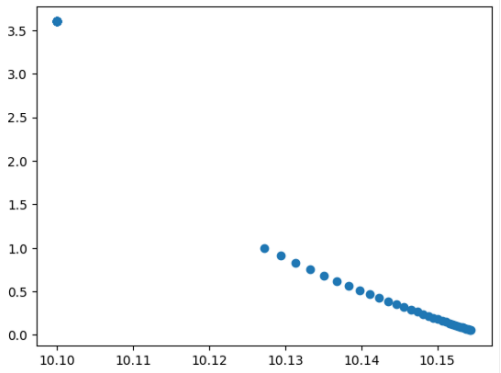


FIGURE 5 – Front par méthode ϵ

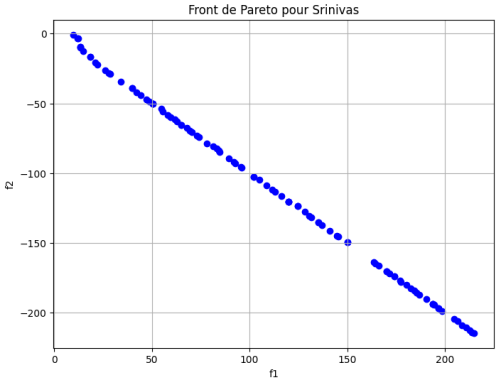


FIGURE 6 – Front par NSGAII

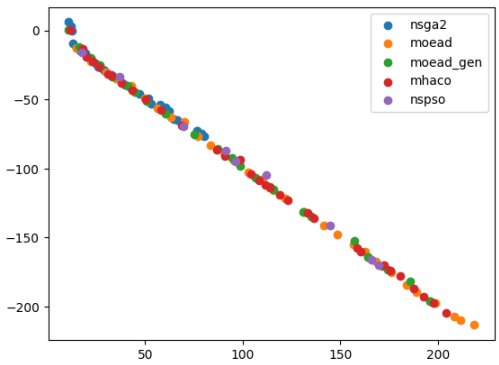


FIGURE 7 – Front par les algorithmes d'OpenTurns

3.6 Une vision nouvelle : l'apport du machine learning

De nos jours, le machine learning et l'intelligence artificielle sont des domaines en plein essort et sur lesquels il faut se positionner pour l'avenir : qui comprendra ces domaines ne sera pas laissé en marge des développements futurs.

Optimisation et apprentissage statistique sont deux domaines extrêmement liés, car la construction de méta-modèles en machine learning revient à vouloir minimiser une certaine erreur. Dans notre cas, nous allons l'utiliser pour développer un nouveau type d'algorithme. L'idée est présentée dans [14] : combiner le NSGAII avec l'algorithme nommé "Efficient Global Optimization" du machine learning. On présente le principe de ce NSGAII-EGO ici.

Algorithm 6 NSGA-II et EGO

Data : Nombre d'appels maximal N_{obj}, taille du plan d'expérience initial N_{pop}
Result : Front de Pareto approché
Initialiser le plan dexpérience avec N_{pop} échantillons
while $N_{appels} < N_{obj}$ **do**
  Construire un méta-modèle pour chaque objectif
  Appliquer NSGA-II sur les méta-modèles pour obtenir un front de Pareto approché
  Diviser l'espace contenant le front de Pareto en d sous-espaces
  for *chaque sous-espace* **do**
    Sélectionner la solution maximisant l'EI
    Ajouter ce point au plan dexpérience
  end
  Appliquer un filtre de redondance pour éviter les sélections redondantes
end

Où l'EI est l'Expected Improvement : on suppose que la meilleure solution que l'on a actuellement est en un point x^*. Dès lors, en un point x on a :

$$EI(x) = \int_{-\infty}^{+\infty} I(x)\phi(z)\,dz \tag{9}$$

où $I(x) = max(f(x) - f(x^*), 0)$ et ϕ est la fonction densité de probabilité d'une loi normale centrée réduite.

L'avantage énorme de cette méthode est qu'au lieu de réaliser l'optimisation sur le modèle originel, on évite beaucoup d'appels car on optimise un méta-modèle. C'est

un gain de temps énorme et précieux à condition bien sûr que les méta-modèles soient bien construits et n'aient pas une trop grande erreur. L'avantage de l'EI est qu'il permet une concentration sur les meilleurs points du front obtenu, ce qui permet d'éviter l'ajout de points réellement inutiles au plan d'expérience. Pour la séparation de l'espace du Front de Pareto, on peut le faire très simplement par exemple en dimension 2 en traçant des droites concourantes à l'origine et inclinées les unes par rapport aux autres d'un même angle.

[14] montre clairement les bons résultats de cet algorithme pour l'optimisation, qui produit un front de Pareto de manière peu coûteuse et précise. Pour nos tests, nous avons pris des modèles de krigeage (processus gaussiens) pour la partie méta-modèle. Sans paramétrage spécial de l'algorithme de krigeage, on obtient la figure suivante en environ 200 appels à la fonction dans le cas du problème de Srinivas :

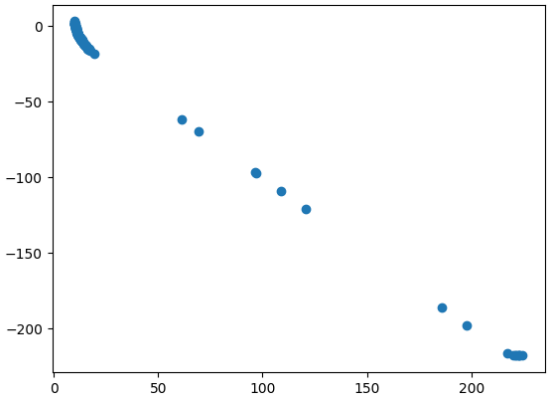


FIGURE 8 – Front de Srinivas obtenu par le couplage entre les algorithmes EGO et NSGAII

Le front est certes moins fourni en nombre de points, mais il est facile d'extrapoler (en tout cas dans ce cas-là) pour déterminer le front complet. Dans tous les cas, même si le front n'est pas complet, on arrive tout de même à obtenir un ensemble de solutions satisfaisant. Bien sûr, les performances dépendent de plusieurs facteurs : le nombre de génération pour le NSGAII, la manière dont on sépare l'espace du front de Pareto, le nombre d'individus dans la population initiale, ou encore les paramètres utilisés pour construire le modèle de krigeage.

3.7 Conclusion sur les algorithmes multi-objectif

Pour conclure sur l'optimisation multi-objectifs des MINLPs, on trouvera ci-dessous un rapide tableau qui décrit les différentes méthodes les plus efficaces et leur cadre mathématique d'application le plus favorable. J'ai choisi de manière

générale, pour les méthodes poids et ϵ-contrainte, l'algorithme BARON qui a les meilleures performances pour les MINLPs (accessible gratuitement pour un maximum de 10 variables, 10 objectifs et 10 contraintes).

Méthode	**Cas d'utilisation**
ϵ-contrainte	Recherche d'une solution unique en peu d'appels
Poids	Recherche d'une solution unique en peu d'appels
NSGA2	Front de Pareto très précis et très riche avec beaucoup d'appels
NSGA2-EGO	Front de Pareto précis et moins riche mais en peu d'appels

TABLE 5 – Cas d'utilisation des méthodes

Il est bon de rappeler dans cette conclusion que tous ces algorithmes sont testés avec des fonctions explicitement définies mathématiquement. Certaines bibliothèques Python présentent donc le désavantage de ne pas pouvoir réaliser l'optimisation des boîtes noires (blackbox), or tous les problèmes rencontrés dans la réalité donnent des boîtes noires. Voici les différentes bibliothèques :

— Pyomo [11]
— JmetalPy [12]
— MealPy [23]
— Gekko [3]
— Openturns [2]
— Pymoo [8]

En particulier, la bibliothèque contenant l'algorithme le plus performant, BARON, est Pyomo, et on ne peut pas optimiser de blackbox via ce module malheureusement (cela explique aussi pourquoi il est très performant). Avec les différents tests réalisés, je me suis concentré sur Openturns qui est assez simple d'utilisation et assez performant, et qui peut bien gérer les blackbox. En particulier, les optimisations réalisées dans la partie suivante sont faites avec Openturns.

4 Tests avec un modèle simple sur *salome meca*

Le problème des tests réalisés auparavant est que tous les problèmes rencontrés sont bien structurés, sans blackbox, et les temps de calcul sont très courts car on résout des modèles mathématiques bien définis explicitement. Nous avons donc réalisé des tests pour réaliser une optimisation sur la chaîne de calcul complète, qui inclut typiquement la création de la géométrie et du maillage, la mise en données, le calcul en lui-même et le post traitement. Certaines de ces étapes peuvent utiliser des logiciels différents.

4.1 Logiciel *Salome meca*

Salome meca est un logiciel de simulation numérique mis en place par EDF. Il est la résultante de l'association de deux logiciels : *Salome*[7], qui permet de faire de la modélisation 3D et de calculer des maillages, et *Code Aster*[8] qui est un solveur utilisant la méthode des éléments finis.

Ce logiciel possède deux gros avantages : le premier est sa gratuité, qui facilite son usage sans poser de problèmes de licences. Le second est sa forte liaison avec Python. En effet, lors de toute modélisation 3D avec un maillage, on peut traduire le fichier obtenu en syntaxe Python automatiquement, via des modules comme GEOM ou SMESH.

En revanche, la pente d'apprentissage sur le logiciel est assez forte, et il m'a fallu un certain temps avant de maîtriser les bases du language *Code Aster*, qui offre beaucoup de possibilités et donc une certaine complexité.

4.2 Modèle général : la table

Le modèle que j'ai utilisé est simple : c'est une modélisation en coque d'une table, qui subit une pression uniforme sur son plan supérieur. Ses 4 pieds sont encastrés et ne peuvent pas bouger pour la modélisation. c'est un modèle très simple par rapport aux cas réels, mais réaliste du point de vue de la chaîne de

7. https://www.salome-platform.org/?lang=fr
8. https://code-aster.org/spip.php?rubrique1

calculs, tout en conservant un temps d'exécution à la fois rapide mais suffisant pour ressentir les difficultés liées aux nombre d'appels à la fonction.

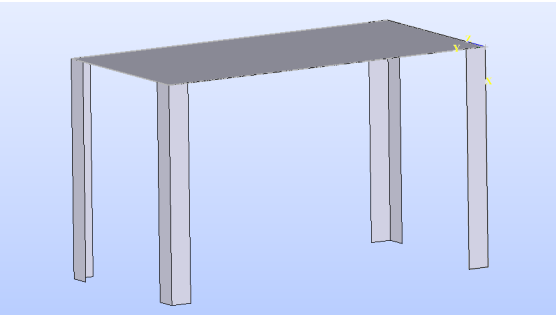


FIGURE 9 – Modèle de la table

L'optimisation a pour but, dans un premier temps, de répondre au problème suivant :

- **Variables :** Longueur , largeur , hauteur , largeur des pieds
- **Objectif :** Masse de la table
- **Contraintes :** Flambement, tenue mécanique (Von Mises)

Exprimé mathématiquement, si on représente la masse par une fonction f, les contraintes par g_1 et g_2 et les variables dans l'ordre respectif L_1, L_2, H et P on obtient :

$$\begin{cases} \min\ f(L_1, L_2, H, P) \\ \text{tel que } g_2(L_1, L_2, H, P) \leq Re \\ \text{et } g_1(L_1, L_2, H, P) > 1 \\ (L_1, L_2, H, P) \in [500, 3000] \times [300, 2000] \times [400, 1500] \times [25, 100] \end{cases} \tag{10}$$

Les valeurs dans les intervalles sont en mm.

Flambement :
Le flambement est une instabilité structurelle qui se produit lorsque la charge critique de compression est atteinte, entraînant un déplacement latéral significatif et potentiellement soudain de la structure. Cela peut se produire même si la charge appliquée est inférieure à la charge qui causerait la rupture du matériau par compression directe.

Code Aster peut, quant à lui, calculer ce qu'on appelle les modes de flambement. Ce sont en fait les différentes valeurs de charges critiques pour lesquelles

on peut théoriquement observer du flambement. En pratique, le logiciel donne les coefficients par lesquels il faudrait multiplier la charge actuelle pour atteindre les charges critiques. Pour éviter le flambement, il faut donc que le plus petit de ces coefficients soit plus grand que 1 (voir [20]).

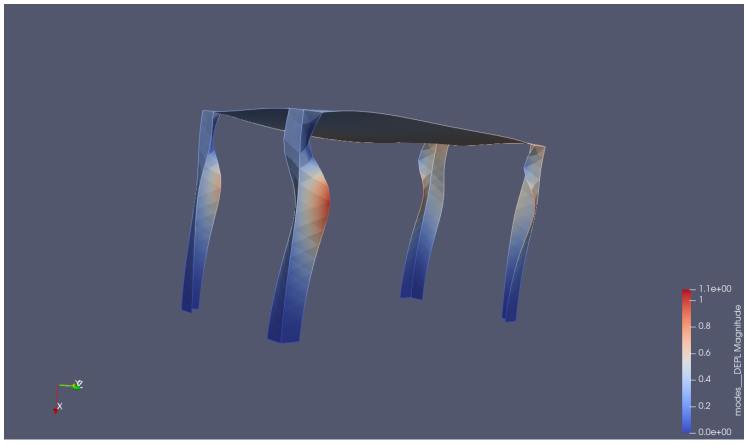


FIGURE 10 – Exemple de flambement

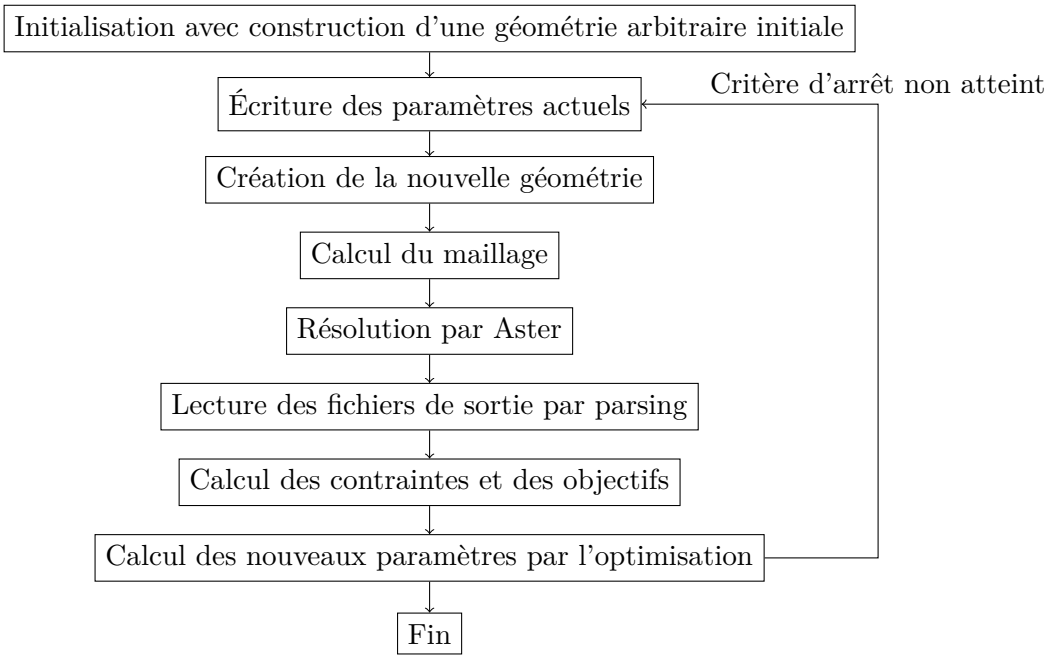
4.3 Description de la chaîne de calcul

Etant donné que nous sommes à présent sur un logiciel de simulation numérique, les évaluations des objectifs et des contraintes se feront exclusivement via ce logiciel. Cela signifie qu'il faut encapsuler la chaîne de calcul dans un code Python de manière de créer un workflow général qui peut être utilisé. Dans cette partie, on décrit la manière dont j'ai codé ce workflow. Commençons avec les fonctions dont nous avons besoin :

- **Fonction de calcul du maillage** : faire appel à salome pour construire le maillage.
- **Fonction de calcul de la simulation** : faire appel à code aster pour réaliser la simulation éléments finis.
- **Fonction(s) de lecture des résultats** : fonction(s) de parsing intelligente(s) pour lire les résultats des calculs et les exploiter.
- **Fonction(s) d'écriture des paramètres** : A chaque boucle d'optimisation, de nouveaux paramètres sont calculés, il faut donc les écrire dans les fichiers salome et code aster pour que le maillage et les calculs soient bien adaptés.
- **Fonction(s) d'évolution** : A chaque boucle d'optimisation, on enregistre quelque part les paramètres et les valeurs des objectifs.

— **Fonction(s) objectif(s) et contrainte(s)** : Encapsulation des calculs, de la lecture des résultats, de la réécriture des paramètres.
— **Boucle d'optimisation** : réalise l'optimisation

Le workflow fonctionne donc comme une optimisation classique, à cela près qu'à chaque appel à la fonction, les opérations suivantes sont réalisées : Écriture des paramètres actuels, lancement des calculs, lecture des résultats, enregistrement des résultats. La partie la plus gourmande en temps est bien entendu le calcul sur les logiciels. Par ailleurs, code aster sort des fichiers assez bien construits et assez simples à parser pour les résultats, ce qui rend l'opération plus simple. Néanmoins, par exemple dans notre cas, code aster sort 4 fichiers de résultats différents (masse, déformations, flambement, contraintes), il faut donc une fonction de parsing pour chaque fichier. Enfin, il est à noter que la syntaxe de code aster lors de la réécriture peut être complexe à prendre en main car il faut bien prendre en compte chaque espace pour que le solveur puisse lire correctement les informations.



4.4 Optimisation mono-objectif : table en acier

Dans un premier temps, j'ai réalisé une optimisation sur une table complètement en acier ($Re = 235$ MPa, $E = 210$ GPa, $\nu = 0.3$, $\rho = 7800$ kg/m^3). L'épaisseur des différents éléments est de 20 mm. La pression appliquée est $P = 50$ MPa. Le problème est alors un problème d'optimisation classique (on n'a pas de

variables entières). On utilise l'algorithme GACO d'OpenTurns pour faire l'optimisation, avec une population initale de 10 individus. On obtient les résultats suivants :

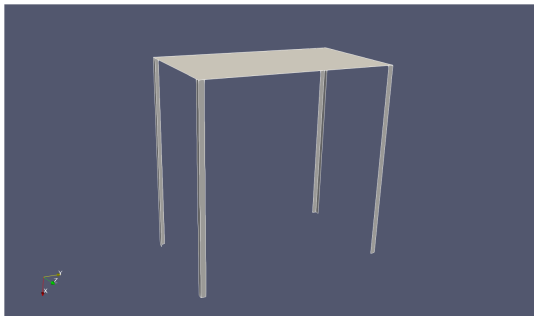


FIGURE 11 – Table en acier optimisée

Valeur des paramètres : [856.001,359.457,547.024,51.3461]
Valeur de l'objectif : ≈ 83.05 kg
Valeur de la contrainte de flambement : 1.31
Valeur de le contrainte de tenue mécanique : 235 MPa

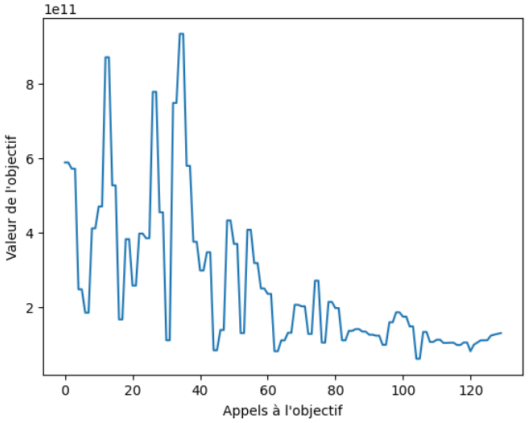


FIGURE 12 – Courbe de l'évolution de la masse en fonction du nombre d'appels

On constate plusieurs choses :
— L'optimisation se déroule bien, et on converge en un nombre d'appels relativement raisonnable.
— La table est de dimension relativement petite : les bornes inférieures sont assez faibles ce qui laisse forcément l'algorithme aller assez bas.

— Etant donnée la masse volumique de l'acier, on obtient une table assez petite mais extrêmement lourde : en terme de conception, le choix du matériau est bien sûr non optimal.
— les contraintes sont presque bien respectées (la tenue mécanique est sur la limite).

4.5 Optimisation multi-objectif : table en bois

Dans ce second test, j'ai réalisé une optimisation sur une table en bois ($Re = 80$ MPa, $E = 20$ GPa, $\rho = 1400$ kg/m^3), avec le plateau en matériau composite (des couches de bois empilées selon certains angles). L'épaisser du plateau dépend du nombre de couche, et chaque couche fait 5 mm.
L'optimisation a pour but, cette fois-ci, de répondre au problème suivant :

— **Variables :** Longueur, largeur, hauteur, largeur des pieds, angle d'orientation en degrés des plis du matériau, nombre de couches
— **Objectif :** Masse de la table, déformations du plateau selon l'axe y
— **Contraintes :** Flambement, tenue mécanique (Von Mises)

On a donc cette fois un MINLP multi-objectif, le type de problème que l'on rencontre le plus souvent. On applique toujours la pression uniforme de 50 MPa, et on ajoute une force axiale de 0.5 MPa selon y comme si on poussait la plateau de la table d'un côté. Le problème se reformule comme suit :

$$\begin{cases} \min\ f(L_1, L_2, H, P, \theta, N) \\ \text{tel que } g_1(L_1, L_2, H, P, \theta, N) \leq Re \\ \text{et } g_2(L_1, L_2, H, P, \theta, N) > 1 \\ (L_1, L_2, H, P, \theta, N) \in [500, 3000] \times [300, 2000] \times [400, 1500] \times [25, 100] \times [0, 90] \times \mathbb{N}^* \end{cases} \tag{11}$$

Nous utilisons dans un premier temps l'algorithme NSGA-II d'OpenTurns pour résoudre ce problème, avec un population initiale de 8 individus. En pratique, la variable N est recherchée dans $\{1, \ldots, 20\}$. On obtient alors les résultats suivants :

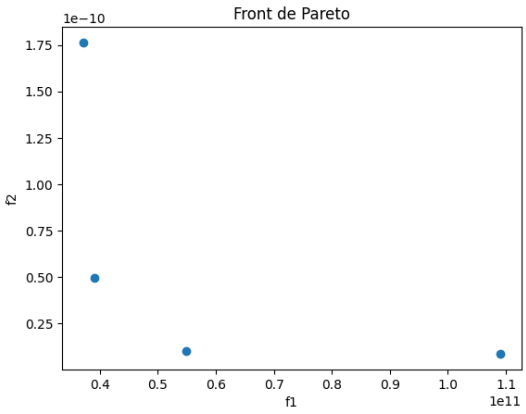


FIGURE 13 – Front de Pareto obtenu

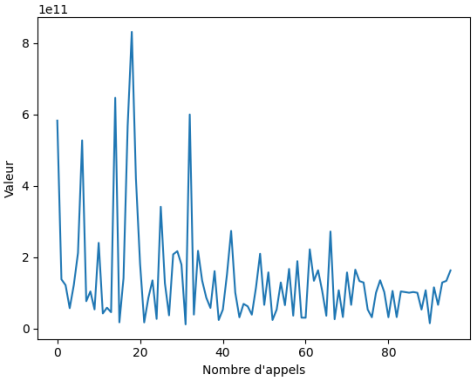


FIGURE 14 – Evolution de la masse

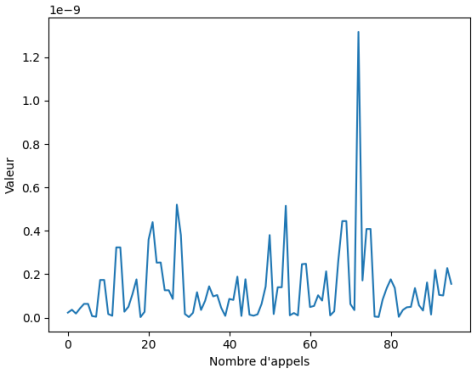


FIGURE 15 – Evolution des déformations

Voici le tableau des valeurs des paramètres optimisés :

N	L1 (mm)	L2 (mm)	H (mm)	P (mm)	θ (deg)	Mode flamb.
13	2923.558	1553.356	1450.221	56.41891	42.16719	1.28
11	1757.6	801.4899	600.243	93.7099	31.0787	1.89
5	889.6625	1514.449	1301.266	40.23595	23.38749	1.73
13	2796.574	775.1974	600.243	93.43379	31.0787	1.02

TABLE 6 – Valeurs des paramètres optimisés

On constate qu'on obtient une bonne allure du front de Pareto, avec des courbes d'évolution des objectifs plutôt cohérentes. Au niveau de la cohérence physique, on trouve tout de même pour la solution la plus légère une table de 37 kg ce qui reste élevé pour une simple table de salon. Par ailleurs, les contraintes sur le flambement sont bien respectées, ce qui montre la validité des valeurs obtenues.

J'ai alors décidé d'expérimenter l'algorithme qui mélange NSGA-II et l'EGO (voir 3.6). Voilà les résultats obtenus :

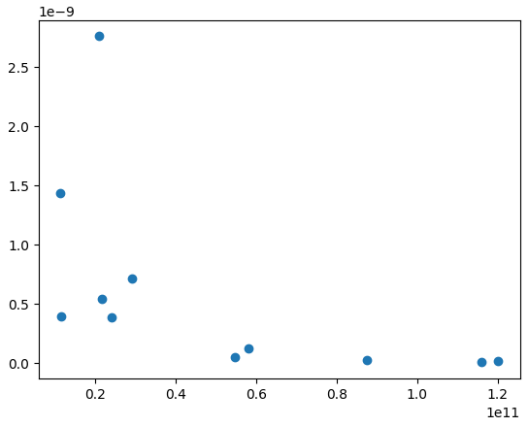


FIGURE 16 – Front de Pareto obtenu

On remarque clairement que le front de Pareto est plus **fourni** : plus de deux fois le nombre de points que pour NSGA-II seul. Cela est dû au nombre d'itérations maximal que j'ai imposé à NSGA-II. En effet, puisque l'optimisation est moins couteuse, on peut se permettre d'aller plus loin dans l'optimisation des méta-modèles. On peut également travailler avec des populations initiales plus grandes que pour le NSGA-II brut. On voit donc finalement que cet algorithme remixé peut être une **solution très intéressante** pour de tels problèmes d'optimisation. Il faudrait bien sûr une étude complète avec d'autres logiciels pour le valider complètement, mais les résultats sont concluants.

4.6 Gestion des crashs logiciels & Capacité à reprendre des calculs déjà faits

4.6.1 Gestion des crashs

Dans cette partie nous abordons la gestion des crashs logiciels. Comment réagissent les algorithmes si le logiciel utilisé plante pour une raison quelconque au

cours d'un calcul ? A priori, dans openturns, si salome meca crash, alors l'optimisation s'arrête et le calcul est perdu. C'est une perte de temps car une partie des calculs s'est bien déroulée, et tout est perdu.

Pour régler ce problème, il suffit d'adopter la structure 'try,...,except,...' de Python. Si les calculs produisent une erreur , alors on retente le calcul 3 fois en repartant du même point. Si le calcul n'aboutit toujours pas, on tente de faire le calcul sur 3 points voisins. Si le calcul marche, on repart de ce point. Par contre, si même les voisins ne donnent pas un résultat satisfaisant, alors on fait retourner à l'objectif la valeur 'inf' ; Autrement dit, on indique à l'algorithme que la zone n'est pas exploitable et ne contient pas un point optimal.

4.6.2 Reprise de calculs d'optimisation déjà faits

Dans Openturns, comme dans d'autres bibliothèques, il est possible d'arrêter une optimisation en cours (en enregistrant ses résultats) puis de réutiliser les calculs déjà faits pour refaire une optimisation.

Pour les algorithmes qui ne se basent pas sur une population, il suffit de faire une première optimisation, d'enregistrer la solution trouvée puis de relancer une optimisation en spécifiant le point de départ comme étant la solution trouvée. Dans le test suivant, (sur un problème donné en Annexe 7.2), nous avons réalisé des optimisations successives en reprenant le résultat de l'optimisation précédente à chaque fois. On obtient une succession d'optimisation qui s'affinent progressivement vers un résultat qui finit par ne plus changer. Voici les résultats qu'on obtient avec *minimize* de *scipy* :

Essai	1	2	3	4	5
Valeur objectif	2×10^{-13}	4×10^{-11}	5×10^{-17}	2×10^{-11}	8×10^{-18}

TABLE 7 – Valeurs des objectifs optimisés avec *minimize*

Pour les algorithmes qui se basent sur une population, il suffit de même de faire une première optimisation, puis d'enregistrer l'état de la population observée à la fin de l'optimisation. On relance ensuite l'optimisation en reprenant cette population. J'ai fait les mêmes tests sous openturns avec **GACO** :

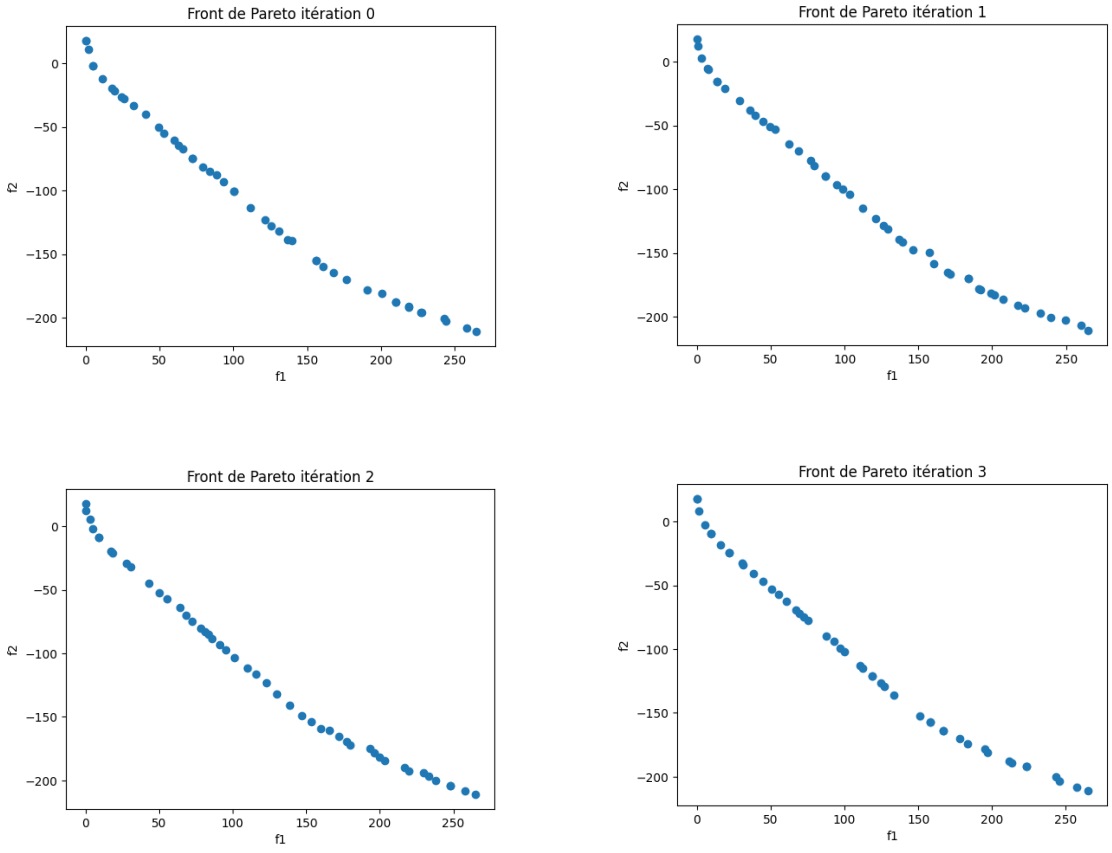
Essai	1	2	3	4	5	6
Valeur objectif	2×10^{-5}	8×10^{-6}	8×10^{-6}	7×10^{-6}	1×10^{-7}	1×10^{-7}

TABLE 8 – Valeurs des objectifs optimisés avec **GACO**

On voit dans les 2 cas que les résultats s'affinent bien. Dans tous les cas, l'optimisation s'améliore ce qui signifie qu'on peut bien reprendre une optimisation en partant de calculs déjà faits.

Pour **NSGAII**, on regarde sur le problème de Srinivas et on observe le Front de Pareto :

Fronts de Pareto de Srinivas à différentes itérations



Le résultat ici est moins évident à discerner, bien que le dessin du front s'affine, on n'observe pas de convergence rapide vers un même front qui serait le bon front (du moins pas avec ce nombre d'itérations, ici 10). En tout cas, on peut tout de même reprendre des calculs déjà faits pour réaliser l'optimisation.

5 Travaux complémentaires

L'objectif principal de ce stage a été de déterminer quelles méthodes étaient les mieux adaptées pour les métiers d'Ingéliance. Nous avons également abordé d'autres problématiques que je vais décrire ici.

5.1 Quelle méthode pour la prise en compte des contraintes ?

Nous allons ici opposer deux façons de résoudre ces problèmes, dans la prise en compte des contraintes d'égalité et inégalité, pour déterminer laquelle est la plus propice à être utilisée dans nos optimisations (notamment en mécanique des structures).

5.1.1 Descriptif des méthodes

Les deux méthodes étudiées ici sont assez intuitives et simples à comprendre. La première, la méthode dite sous contraintes, consiste simplement à considérer le problème (1) en tant que tel et à rejeter certaines solutions trouvées si elles sont hors zone de faisabilité. Le principal problème de cette méthode est qu'elle peut mener à des temps de calcul plus longs si l'algorithme est mal calibré. Par exemple, un algorithme tel que le PSO, vient regarder après calcul de la solution si elle est faisable ou non, ce qui allonge bien sûr les temps de calcul.

La deuxième méthode est celle de la pénalisation des contraintes. On ajoute une fonction pénalisante $p(x)$ à l'objectif à laquelle on attribue un certain poids r (relatif à l'importance de respecter très précisément la contrainte ou pas). Formellement on obtient :

$$\begin{cases} \min\limits_{x \in X} (f(x) + rp(x)) \\ x_k \in \mathbb{Z} \text{ pour } k \in I \end{cases} \tag{12}$$

La fonction de pénalisation prend en compte sous une certaine forme les contraintes de (1) ; une fonction de pénalisation souvent utilisée est la suivante :

$$\min\ f(x) + r \sum_{j=1}^{j=m} max(g_j(x), 0)^2 + r \sum_{i=1}^{i=p} (max(h_i(x), 0)^2 + max(-h_i(x), 0)^2) \tag{13}$$

Nous utliserons pour nos tests cette fonction. La pénalisation ne garantit en rien d'obtenir une solution au problème initial, néanmoins, une bonne prise en compte du poids des contraintes dans la pénalisation peut se révéler très efficace. En ajustant les coefficients de pénalité, les solutions trouvées par l'algorithme d'optimisation tendent à se rapprocher des régions réalisables de l'espace de solution. Avec des coefficients de pénalité suffisamment grands, les solutions non réalisables

sont fortement pénalisées, ce qui oriente l'algorithme vers des solutions réalisables, ce qui permet de considérer cette méthode comme viable et intéressante.

5.1.2 Résultats

Le set de problèmes en Annexe 7.4 permet d'avoir un panel d'éventualités et de contextes, nous avons ici des MINLPs comme des NLPs, avec des contraintes d'égalité et d'inégalité, qui sont peu ou très non linéaires. Pour réaliser les tests nous avons procédé comme suit : l'algorithme utilisé est l'algorithme GACO de Pagmo (utilisé ici via Openturns sur Python), qui peut prendre en compte les contraintes comme la pénalisation. Nous avons ensuite réalisé des boucles de $N = 10000$ tests d'optimisation et nous avons regardé la fréquence à laquelle l'algorithme trouve la bonne solution (ou est dans un intervalle acceptable pour les problèmes NLPs). Nous avons également relevé les coefficients de pénalité. On consigne les résultats dans le tableau 9 (*sc* pour sous contrainte, *péna* pour pénalisation).

Pb	% réussite *sc*	% réussite *péna*	Facteur r
1	81.04	83.59	10
2	81.57	81.35	10^6
3	32.2	31.8	10^3
4	36.8	92.7	10^4
5	97.8	30.4	10^6

TABLE 9 – Résultat des tests avec GACO

Les facteurs de pénalisation ont été déterminés par une boucle de maximisation du pourcentage de réussite (avec des valeurs tests comprises entre 10^{-6} et 10^{10}). Cela rajoute un certain temps (ici très peu car le temps d'appel aux fonctions est négligeable). Néanmoins, avec des applications et appels à la fonction qui peuvent prendre au moins 10 secondes, il peut être très couteux d'avoir à déterminer ce facteur.

5.1.3 Conclusion

On remarque, au regard des tests effectués, que la pénalisation peut être préférée dans des situations où le problème est plutôt simple et lorsqu'on a une idée précise du facteur de pénalisation. Si au contraire, l'appel à l'objectif est couteux (ce qui est le cas lors de la plupart de nos cas industriels), alors passer par une optimisation sous contraintes est plus performant et plus rapide car évite le temps de recherche du facteur de pénalisation. Le risque en se trompant de facteur est de faire trop peser les contraintes ou au contraire pas assez, auquel cas il est tout à fait possible que la solution trouvée soit complètement erronnée.

5.2 Paralléliser les algorithmes d'optimisation

Toujours dans l'optique de gagner du temps de calcul, une possibilité est de paralléliser les calculs d'optimisation afin de distribuer la charge de calcul sur différents coeurs et ainsi diminuer le nombre d'itérations nécessaires. La question est : peut-on facilement paralléliser les calculs d'une optimisation, et si oui comment le faire ?

Il s'avère que la bibliothèque Openturns [2] possède quelques algorithmes parallélisables (autrement dit on peut réaliser des calculs par lots (batchs) sur plusieurs coeurs). On compte parmi eux GACO, NSGA-II et PSO. Néanmoins, il faut que les fonctions passées dans l'algorithme acceptent toutes le fait de pouvoir lancer des instances de calcul en même temps.

5.2.1 Rendre les fonctions OpenTurns capables d'accepter les calculs simultanés

Les fonctions OpenTurns ne sont pas forcément parallélisables. En particulier, lorsqu'on tente d'appliquer un algorithme parallélisable sur notre fonction issue de salome meca, on n'obtient aucune différence notable entre la parallélisation et l'algorithme séquentiel.

J'ai donc recherché un module Python capable de rendre les fonctions parallélisables : otwrapy [9]. Ce module est extrêmement facile à utiliser, rend les fonctions externes parallélisables selon un certain nombre de coeurs à indiquer. Pour les fonctions à expressions mathématiques, la classe d'openturns "PythonFunction" permet d'indiquer directement si on veut rendre la fonction parallélisable via l'opérateur "ncpus".

5.2.2 Essais avec des fonctions mathématiques

J'ai donc d'abord testé la parallélisation sur des fonctions mathématiques, qui permettent un calcul plus rapide pour faire davantage de tests. Les problèmes tests sont en Annexe 7.1, voici les résultats obtenus avec l'algorithme GACO (je remercie d'ailleurs Régis Lebrun, co-fondateur d'OpenTurns, pour m'avoir aidé dans cette partie) :

9. http://openturns.github.io/otwrapy/master/examples.html

Test	Tps *séq* (s)	Tps *para* (s)	Nbre coeurs	Tps calcul objectif (≈) (s)	Facteur
1	212.22	94.86	4	2.5	2.25
2	212.22	209.53	8	2.5	1.01
3	33.74	26.7	8	0.02	1.08
4	0.2657	13.01	4	10^{-5}	0.020

TABLE 10 – Résultat des tests avec GACO

Ici, la colonne facteur représente le rapport entre le temps séquentiel (*séq*) et le temps parallélisé (*para*). Si il est plus grand que 1, on peut dire que la parallélisation est utile (surtout quand c'est plus grand que 2), et sinon, la parallélisation rallonge le temps d'optimisation. Grâce aux résultats, on se rend compte que ce genre de phénomènes apparaît si le temps de calcul de base pour la fonction objectif est trop faible par rapport au temps de lancement de plusieurs calculs en parallèle.

On en déduit, de façon plutôt logique, que la parallélisation est utile pour des fonctions qui mettent un certain temps à s'exécuter (ce qui est exactement notre cas). Ces résultats se vérifient d'ailleurs sur l'algorithme NSGA-II (parallélisé via le module *DEAP* [13]) :

Test	Tps *séq* (s)	Tps *para* (s)	Nbre coeurs	Tps calcul objectif (≈) (s)	Facteur
1	97.8	38.3	4	3	2.25
2	97.8	121.7	6	3	0.79
3	29.42	8.15	4	0.02	3.62
4	29.42	6.25	8	0.02	4.7
5	2.30	2.86	8	10^{-5}	0.8

TABLE 11 – Résultat des tests avec NSGA-II

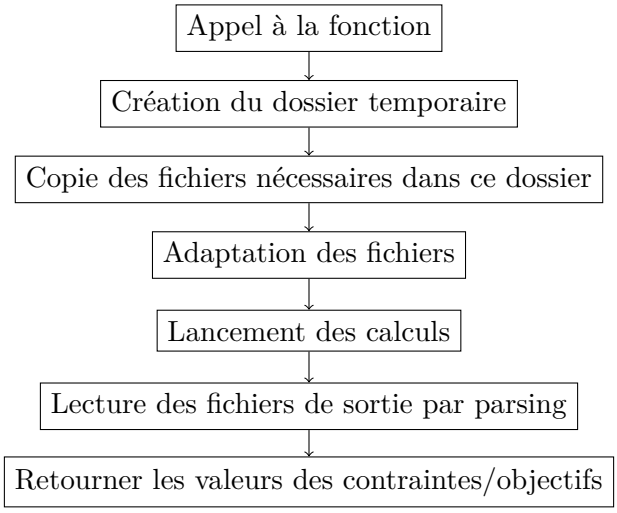
5.2.3 Test avec salome meca

Tester la parallélisation sur le logiciel salome meca est une autre paire de manche. En effet, comme la fonction principale écrit dans des fichiers, y fait appel, les lit, les conflits potentiels et donc les crashs potentiels sont nombreux.

J'ai donc procédé en 2 étapes : la parallélisation à la main des fonctions intermédiaires qui lancent les calculs d'aster et de maillage, puis j'ai rendu les dossiers de travail pour chaque appel uniques, grâce au module Python *tempfile*. Ce module permet de créer des fichiers temporaires uniques dans lesquels on peut mettre tous

les fichiers dont on a besoin ; dans mon cas, j'ai donc réalisé des copies des fichiers dont salome meca avait besoin pour s'exécuter. Il a aussi fallu mettre en place une modification automatique du script de maillage pour que l'export du maillage recalculé se fasse dans le bon dossier.

Il faut avouer que salome meca a montré quelques réticences à se laisser paralléliser, néanmoins après quelques astuces et détails trouvés, j'ai réussi à paralléliser la fonction globale que l'on va chercher à optimiser. La structure globale simplifiée du code de la fonction parallélisable est alors la suivante :



J'ai donc réalisé le même test qu'en partie 5.4. Comme il a fallu faire des modifications au niveau du lancement du calcul de maillage (ajout d'un timer de 20 secondes pour éviter au code de planter), l'expérience est quelque peu différente en terme de temps de calcul. Néanmoins, en terme de nombre d'appels à la fonction, le gain est flagrant puis les appels sont dispatchés sur différents coeurs. A deux coeurs, c'est comme si on avait divisé le nombre d'appels par 2 ce qui est déjà énorme. Au delà de quatre coeurs, la machine et salome meca commencent à moins bien supporter, car cela signifie 4 instances du logiciel lancées en même temps et la mémoire a du mal à gérer cette charge.

5.3 Optimisation des empilements pour un matériau composite

Les matériaux composites sont des matériaux composés d'empilements de couches de différents matériaux (ce sont les plis) qui sont orientés selon une certain direction comme montré figure 17.

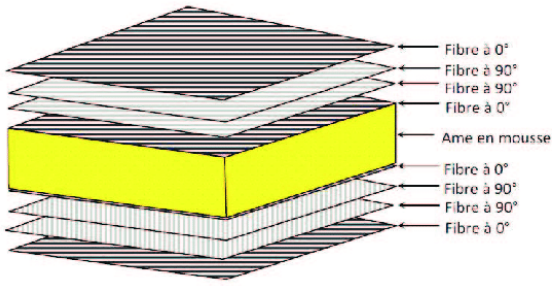


FIGURE 17 – Matériau composite (ref)

Bien souvent, on réalise ce genre de matériau pour obtenir des propriétés particulières (rigidité, masse...). Les composites sont notamment beaucoup utilisés dans l'aéronautique. Ainsi, dans ce cadre, on va tenter de trouver leur ordre et leur orientation qui rendent le matériau optimal selon certains objectifs. La difficulté majeure de l'optimisation de ces matériaux est qu'elle regroupe en quelques sortes un problème combinatoire (du type voyageur de commerce) avec un problème MINLP (nombre de plis discrets, orientation continue ou discrète). J'ai donc procédé en plusieurs étapes. Dans notre cadre, on n'optimisera pas le nombre de plis, on le fixera au préalable.

5.3.1 Le voyageur de commerce

C'est le problème classique d'optimisation combinatoire : un voyageur de commerce doit visiter un nombre N de villes en parcourant la distance la plus petite possible. Par force brute, pour tester toutes les combinaisons possibles, la complexité est en $N!$ ce qui n'est clairement pas acceptable en terme computationnel. Une manière d'implémenter ce problème pour le résoudre est de le coder sous forme de matrice, où les coefficients de la matrice sont les distances entre la ville i et j :

$$D = \begin{pmatrix} 0 & 10 & 15 & 20 & 25 \\ 10 & 0 & 35 & 25 & 30 \\ 15 & 35 & 0 & 30 & 15 \\ 20 & 25 & 30 & 0 & 20 \\ 25 & 30 & 15 & 20 & 0 \end{pmatrix} \tag{14}$$

Cette matrice est symétrique puisque la distance est symétrique. Pour résoudre ce problème, plusieurs méthodes existent ; ici on s'intéresse à 2 d'entre elles qui sont les plus utilisées pour ce problème, le recuit simulé et l'algorithme des colonies de fourmis (déjà décrit par l'algorithme 4). On présente donc le recuit simulé (voir Annexe 7.7).

On fait donc une comparaison entre la force brute, le recuit simulé (AS) et l'algorithme des fourmis (ACO) (voir table 12).

Algo	N	Nb appels	% réussite	Paramètres
Force brute	5	24	100	NA
AS	5	2300	100	$T_{min} = 0.001$, $T_0 = 1000$, $\alpha = 0.995$
ACO	5	1000	100	$n_{fourmis} = 10$, $n_{iter} = 100$
Force brute	7	1008	100	NA
AS	7	2300	100	$T_{min} = 0.001$, $T_0 = 1000$, $\alpha = 0.995$
ACO	7	1000	100	$n_{fourmis} = 10$, $n_{iter} = 100$
Force brute	9	40320	100	NA
AS	9	23022	87	$T_{min} = 0.001$, $T_0 = 1000$, $\alpha = 0.9995$
ACO	9	1000	100	$n_{fourmis} = 10$, $n_{iter} = 100$
Force brute	11	3628800	100	NA
AS	11	57560	73	$T_{min} = 0.001$, $T_0 = 1000$, $\alpha = 0.9998$
ACO	11	1000	100	$n_{fourmis} = 10$, $n_{iter} = 100$

TABLE 12 – Comparaison sur le voyageur de commerce

Il est évident que la force brute présente rapidement des limites dès qu'on dépasse 10 villes à visiter, tandis que les deux autres algorithmes trouvent les bons résultats et plus rapidement, même si on voit clairement que l'ACO est bien plus efficace (les pourcentages ont été déterminés sur 100 tentatives de résolution pour chaque algorithme). En bref, l'algorithme à privilégier est clairement l'ACO dans ce cadre.

5.3.2 Les matériaux composites : 1ère approche

L'idée pour les matériaux composites est de s'inspirer du voyageur de commerce pour mettre en données le problème puis le résoudre de la même manière avec les algorithmes ci-dessus.

Dans cette première approche, je vais dans un premier temps regarder l'optimisation uniquement de l'ordre des plis. En d'autres termes, nous allons considérer un ensemble de plis qui sont déjà orientés les uns par rapport aux autres, et dont le nombre est déjà fixé et qui sont chacun associés à un matériau. Finalement, les plis sont déjà entièrement définis, on ne s'intéresse qu'à leur ordre. De cette manière, le problème se ramène à celui du voyageur de commerce : la matrice contient cette fois-ci les angles entre chaque pli (à noter que cette fois, comme les angles des plis doivent être compris dans [-90,90], la matrice sera antisymétrique). Voici un exemple de matrice que l'on peut avoir :

$$A = \begin{pmatrix} 0 & -\theta_{12} & -\theta_{13} & -\theta_{14} \\ \theta_{12} & 0 & -\theta_{23} & -\theta_{24} \\ \theta_{13} & \theta_{23} & 0 & -\theta_{34} \\ \theta_{14} & \theta_{24} & \theta_{34} & 0 \end{pmatrix} \tag{15}$$

Il ne reste alors qu'à calculer, via n'importe quelle fonction, la rigidité du matériau tel que défini par tel ou tel ordre des plis. Dans notre cas, j'utilise *salome meca*, pour avoir une représentation réaliste. L'avantage ici est qu'on n'aura pas à recalculer le maillage à chaque étape, puisqu'on ne change rien aux dimensions de l'objet.

On modélise donc une plaque en matériau composite de 4 plis avec 2 matériaux en bois qui ont des modules d'Young différents : le premier à 11600 MPa (bois1) et le second à 10800 MPa (bois2). J'ai par ailleurs considéré qu'un pli sur deux serait de chaque matériau (donc le matériau est composé de plis alternés), et la plaque globale du matériau est fixée à une de ses extrèmités, et on vient appliquer une traction de l'autre côté de 100 N/mm^2.

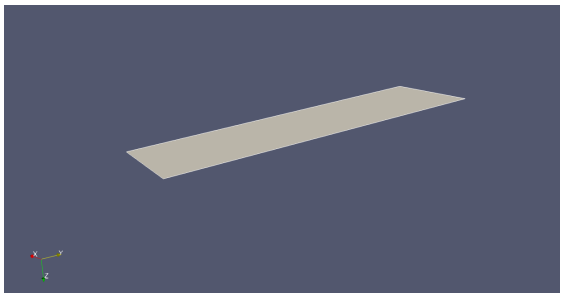


FIGURE 18 – Plaque étudiée

A la fin du calcul *code aster*, on extrait alors les déformations et les contraintes pour calculer le module d'Young global de la structure. On essaie alors avec les algorithmes de maximiser ce module. Pour les tests, nous avons testé sur 4 plis pour des raisons de temps d'exécution.

En bref, les différents résultats obtenus avec l'algorithme des fourmis concernant uniquement l'ordre des plis sont recensés table 13.
Ces résultats concordent à 100% avec la force brute, ce qui montre de nouveau l'efficacité de l'ACO pour ce genre de problèmes.

Néanmoins, l'approche globale décrite ici ne s'arrête pas là. En effet, on part du principe qu'on a déjà toutes les propriétés des plis, or en réalité ce n'est pas le

Matrices d'Angles	Vecteurs d'Ordre	Rigidité (MPa)
$\begin{pmatrix} 0 & -30 & 45 & -60 \\ 30 & 0 & -15 & 75 \\ -45 & 15 & 0 & -90 \\ 60 & -75 & 90 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$	11632.60
$\begin{pmatrix} 0 & -45 & 30 & -15 \\ 45 & 0 & -60 & 90 \\ -30 & 60 & 0 & -45 \\ 15 & -90 & 45 & 0 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$	12234.5
$\begin{pmatrix} 0 & -90 & 15 & -30 \\ 90 & 0 & -45 & 60 \\ -15 & 45 & 0 & -75 \\ 30 & -60 & 75 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 1 \\ 4 \\ 3 \end{pmatrix}$	10923.32

TABLE 13 – Résultats avec l'ACO pour 10 itérations et 10 fourmis

cas. Il faut donc réaliser, en amont, une optimisation du nombre de plis et de leur orientation (voir la partie 4.5). Une fois cette première optimisation faite, on peut alors optimiser l'ordre des plis. Le problème de cette approche est qu'elle est plus gourmande en temps et en appels au modèle, puisqu'on vient faire deux boucles d'optimisation d'affilée.

On peut alors de demander ce qu'il se passe physiquement. Qu'est ce qui fait que la rigidité varie en fonction de l'ordre des plis ? En effet, si on vient simplement faire une traction sur un composite, quel que soit l'ordre des plis, c'est comme si on venait faire une traction sur chaque fibre, et alors le module de Young ne devrait pas être modifié. En effort de traction, ce qui se produit, c'est que si on a deux fibres de matériaux différents sur lesquelles on tire de la même manière, alors le matériau le plus rigide va moins se déformer que l'autre et la plaque globale va se déformer mais hors du plan de la plaque (voir comparaison image 5.3.2). Pour cette raison, le module de Young change en fonction de l'ordre des plis (lorsqu'on vient ajouter l'orientation des plis, évidemment les propriétés intrinsèques du composite changent et le module de Young aussi).

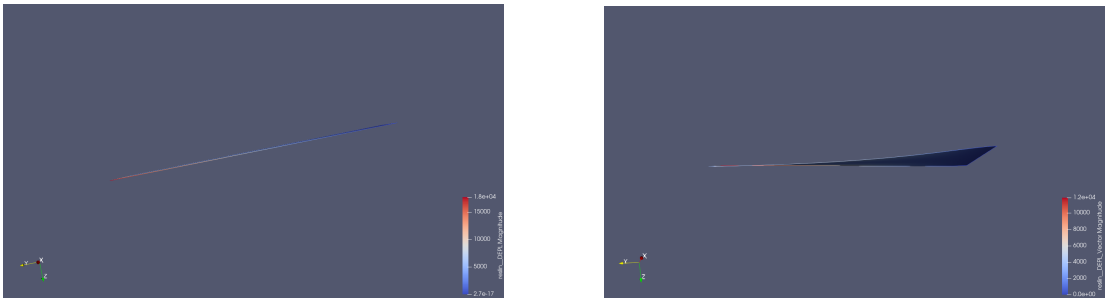


FIGURE 19 – Observation des déformations de la plaque

5.3.3 Les matériaux composites : 2ème approche

Pour cette section, on s'appuie majoritairement sur la thèse [17] de F.X. Irisarri, qui décrit la mise en place d'un algorithme évolutionnaire pour l'optimisation des séquences de plis dans des matériaux composites. Cette thèse s'appuie sur des règles de fabrication de ces matériaux (que je n'ai pas pris en compte jusque-là) prises en compte sous forme de contraintes ou par adaptation des différents opérateurs de l'algorithme (qu'on retrouvera 7.3).

Pour coder cet algorithme, j'ai utilisé le module *DEAP*, qui permet intrinsèquement de redéfinir les opérateurs génétiques de croisement, mutation ou permutation pour les algorithmes proposés. En particulier, j'ai utilisé l'algorithme évolutionnaire simple pour ce code.

Le modèle utilisé est le même que précédemment, et on fixe le nombre de couches à 20 pour le premier test puis 10 pour le deuxième (peut-être vu comme une contrainte sur l'épaisseur de la structure). L'algorithme prendra les valeurs d'orientation des angles dans [0,90,-45,45]. On fixe par ailleurs 10 générations et une population de 10 individus à l'algorithme évolutionnaire, pour rester faisable dans un temps raisonnable sur la machine. Voici les résultats obtenus :

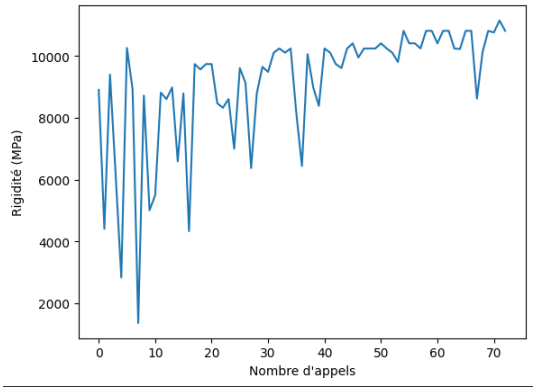


FIGURE 20 – Courbe d'optimisation de la rigidité pour le 1er test

Valeur de rigidité : 11146.96 MPa
Vecteur solution (de longueur 20) : [90, 0, 90, 0, 45, -45, 0, 45, 0, 0, 45, 90, 90, 90, 0, 0, 45, -45, 45, -45]
Matériau pour chaque pli : [bois1, bois2 , bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2, bois1, bois2]

Plusieurs choses sont à remarquer :

— L'algorithme converge bien vers une solution, qui est cohérente en terme d'ordre de grandeur ; la courbe d'optimisation est croissante (puisqu'on veut maximiser, cela reste cohérent).
— Certaines règles de conception des stratifiés ne sont pas respectées car je ne les ai pas implémentées pour les tests.

Pour le deuxième test, on prend en compte cette fois-ci les règles de conception, en les considérant comme des contraintes. On procède ainsi par pénalisation (si un critère n'est pas respecté, on accorde une grande pénalisation au point observé). On se place dans les mêmes conditions que pour le premier test, en fixant le nombre de plis à 10 cette fois.

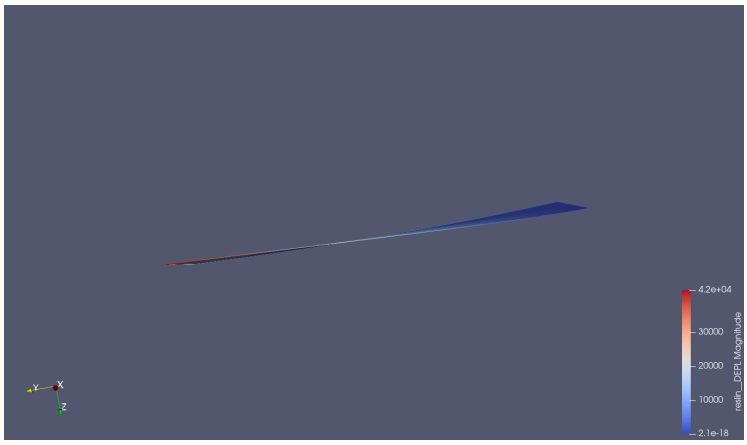


FIGURE 21 – Plaque sous traction peu déformée

Valeur de rigidité : 9961.3 MPa
Vecteur solution (de longueur 10) : [0, 45, 90, -45, -90, -90, -45, 90, 45, 0]
Matériau pour chaque pli : ['bois2', 'bois1', 'bois1', 'bois2', 'bois1', 'bois1', 'bois2', 'bois1', 'bois1', 'bois2']

Plusieurs choses sont de même à remarquer :

— L'algorithme converge bien vers une solution, qui est cohérente en terme d'ordre de grandeur ; la courbe d'optimisation n'est pas informative car on regarde l'objectif pénalisé, qui n'a donc pas de vraies valeurs de rigidité.
— Les règles de conception sont majoriatirement respectées sauf la règle des 45 degrés. Cela s'explique par le fait qu'avec 5 valeurs d'orientation pour 10 plis, garder toutes les contraintes n'est pas possible (trop peu de plis).
— les déformations hors du plan sont très réduites.

6 Conclusion

La volonté d'Ingéliance, au travers de ce stage, était d'acquérir et renforcer ses connaissances en terme d'optimisation pour les problèmes mixtes. Ce stage a permis d'éclaircir plusieurs horizons au sein du domaine de l'optimisation des MINLPs, à commencer par l'exploration de divers algorithmes de résolution. L'état de l'art et les tests ont pu mettre en avant certaines méthodes ou solveurs, par leur précision, leur efficacité et le nombre d'appels nécessaires à la convergence.

Néanmoins, j'ai pu constater dans la littérature que peu d'algorithmes "généraux" existent, et, bien souvent, pour les projets, des algorithmes spécifiques sont développés (du moins les projets de recherche, cf. MOTLBO). A cause de cela, il m'a été compliqué de décider d'un algorithme à privilégier. Dans mes tests, les algorithmes les plus performants et accessibles sont les algorithmes stochastiques (ACO, NSGAII...), et on a vu qu'avec des problématiques comme le nombre d'appels à la fonction, le machine learning couplé à ces algorithmes pouvait se révéler efficace.

Toutefois, je tiens à préconiser ici les 2 algorithmes que j'ai trouvés les plus performants et les plus intéressants à utiliser dans le domaine de la simulation numérique :

- **GACO** : Comme expliqué auparavant, l'algorithme des fourmis est un algorithme stochastique ce qui s'applique bien à l'optimisation des boîtes noires. Il présente l'avantage qu'on doit lui indiquer le nombre de boucles que l'on veut faire autrement dit on fixe le nombre d'appels à la fonction. Enfin, il est performant puisque dans l'ensemble des tests il est arrivé à des résultats concluants.
- **EGO-NSGAII** : L'algorithme NSGAII associé au Machine Learning est un algorithme à explorer car il me semble une très bonne option pour répondre aux exigences. Sous réserve de son efficacité dans des dimensions plus grandes (avec plus d'objectifs), il a été performant et efficace dans nos tests.

Par ailleurs, j'ai été en mesure de développer un workflow d'optimisation sur le logiciel *salome meca*, qui est observé de près par Ingéliance puisqu'un stage se déroule pour le valider. Cela permettra de simplifier la mise en place de telles optimisations si le cas se présente.

Les autres travaux que j'ai réalisés permettent, quant à eux, d'apporter des précisions sur les algorithmes étudiés, et d'explorer leurs capacités : contraintes, parallélisation, que faire en cas de coupure du calcul... Tout cela a permis à la fois de montrer comment gérer les éventuels problèmes qui pouvaient se présenter lors

d'optimisations, mais aussi d'essayer d'optimiser les performances des algorithmes (ce dont Ingéliance a besoin lors de ses simulations numériques).

Enfin, l'étude sur l'optimisation des matériaux composites vient compléter une étude déjà réalisée par un précédent ingénieur d'Ingéliance. Une implémentation avait déjà été réalisée, mais j'ai essayé de développer un autre moyen de répondre à la question. Cette partie a permis d'explorer l'optimisation combinatoire, et a permis de montrer de nouveau l'excellence des algorithmes stochastiques (et notamment celui des fourmis) pour résoudre des problèmes qui a priori sont complexes lorsqu'on passe en grande dimension.

A titre personnel, ce stage m'a ouvert la voie de la simulation numérique. J'ai énormément appris au niveau du code en langage Python, mais aussi au niveau des logiciels de calcul utilisés. J'ai également beaucoup appris sur le plan des connaissances en terme d'optimisation, j'ai à présent une représentation globale de ce domaine qui pourrait bien m'être utile dans d'autres situations. J'ai beaucoup apprécié le fait d'avoir été laissé en totale autonomie pendant mes recherches, cela m'a permis de voir ce qu'était un travail personnel approfondi de recherche sur le long terme.

En outre, l'optimisation est un domaine profond et plein de ressources, utile dans tous les domaines scientifiques. Dans les prochaines années, je pense qu'elle jouera un rôle majeur, notamment pour l'économie des ressources physiques à notre disposition. Bien que les recherches soient plutôt abouties, l'arrivée du machine learning peut clairement redonner de l'élan à l'optimisation.

7 Annexes

7.1 Annexe 1 : Problèmes utilisés pour tester la parallélisation des algorithmes

Pour GACO :

Minimiser :

$$f(x) = \sum_{i=0}^{2} \left(\frac{(x_{2i-2} - 3)^2}{1000} - (x_{2i-2} - x_{2i-1}) + e^{20(x_{2i-2} - x_{2i-1})} \right) \tag{16}$$

Sous les contraintes :

$$c_1(x) = 4(x_0 - x_1)^2 + x_1 - x_2^2 + x_2 - x_3^2 \leq 0$$
$$c_2(x) = 8x_1(x_1^2 - x_0) - 2(1 - x_1) + 4(x_1 - x_2)^2 + x_0^2 + x_2 - x_3^2 + x_3 - x_4^2 \leq 0$$
$$c_3(x) = 8x_2(x_2^2 - x_1) - 2(1 - x_2) + 4(x_2 - x_3)^2 + x_1^2 - x_0 + x_3 - x_4^2 + x_0^2 + x_4 - x_5^2 \leq 0$$
$$c_4(x) = 8x_3(x_3^2 - x_2) - 2(1 - x_3) + 4(x_3 - x_4)^2 + x_2^2 - x_1 + x_4 - x_5^2 + x_1^2 + x_5 - x_0 \leq 0$$
$$c_5(x) = 8x_4(x_4^2 - x_3) - 2(1 - x_4) + 4(x_4 - x_5)^2 + x_3^2 - x_2 + x_5 + x_2^2 - x_1 \leq 0$$
$$c_6(x) = -\left(8x_5(x_5^2 - x_4) - 2(1 - x_5) + x_4^2 - x_3 + x_3^2 - x_4\right) \leq 0$$

Où :
— $x = (x_0, x_1, x_2, x_3, x_4, x_5)$
— Les variables x_0, x_1, x_2, x_3 sont continues.
— Les variables x_4, x_5 sont entières.
— $x \in [-5, 5]^6$

Pour NSGAII :

Minimiser les fonctions objectifs :

$$\text{obj1} = x_0^2 - x_1 + x_2 + 3x_3 + 2x_4 + x_5$$
$$\text{obj2} = 2x_0^2 + x_2^2 - 3x_0 + x_1 - 2x_3 + x_4 - 2x_5$$

sous les contraintes :

$$
\begin{aligned}
e1 &= 3x_0 - x_1 + x_2 + 2x_3 \leq 0 \\
e2 &= 4x_0^2 + 2x_0 + x_1 + x_2 + x_3 + 7x_4 - 40 \leq 0 \\
e3 &= -x_0 - 2x_1 + 3x_2 + 7x_5 \leq 0 \\
e4 &= -x_0 + 12x_3 - 10 \leq 0 \\
e5 &= x_0 - 2x_3 - 5 \leq 0 \\
e6 &= -x_1 + x_4 - 20 \leq 0 \\
e7 &= x_1 - x_4 - 40 \leq 0 \\
e8 &= -x_2 + x_5 - 17 \leq 0 \\
e9 &= x_2 - x_5 - 25 \leq 0
\end{aligned}
$$

avec les variables :

$$
\begin{aligned}
x_0, x_1, x_2, \in \mathbb{R} \quad & \text{(variables continues)} \\
x_3, x_4, x_5 \in \mathbb{Z} \quad & \text{(variables entières)}
\end{aligned}
$$

7.2 Annexe 2 : Problème utilisé pour le test mono-objectif sur la capacité à reprendre des calculs déjà faits

Minimiser :

$$\text{obj1} = x_0^2 + x_1^2$$

7.3 Annexe 3 : Algorithme global de [17]

Algorithm 7 Architecture Générale de l'Algorithme Proposé

Data : Taille de la population μ, taille de l'archive $|\mathcal{A}|$, nombre maximum de générations $G_{\max}$, paramétrage des opérateurs et des règles de conception
Result : Un ensemble de solutions non dominées $\mathcal{P}$
Étape 1 : Initialisation
$\mathcal{P}_0 \leftarrow$ génération d'une population initiale de taille μ
Création de l'archive $\mathcal{A}_0 \leftarrow \mathcal{P}_0$
Étape 2 : Évaluation
Calcul des fonctions-objectif pour chaque individu de $\mathcal{P}_t$
Étape 3 : Calcul de la Fonction de Qualité
Calcul de la fonction de qualité pour chaque individu de $\mathcal{P}_t$
Étape 4 : Sélection Environnementale
Copier les individus non dominés de $\mathcal{P}_t$ dans $\mathcal{A}^+$
if $|\mathcal{A}^+| > |\mathcal{A}|$ **then**
| Réduire $\mathcal{A}^+$ au moyen de l'opérateur de troncature
end
if $|\mathcal{A}^+| < |\mathcal{A}|$ **then**
| Compléter l'archive $\mathcal{A}^+$ avec les meilleurs individus dominés de $\mathcal{P}_t$ (au sens de la fonction de qualité)
end
Étape 5 : Terminaison
if $t \geq G_{\max}$ ***ou*** *un autre critère d'arrêt est satisfait* **then**
| Retourner $\mathcal{P}$, l'ensemble des solutions non dominées contenues dans $\mathcal{A}^+$
end
Étape 6 : Sélection pour la Reproduction
Tirer μ solutions de $\mathcal{A}^+$ par tournoi binaire avec remise
Soit $\mathcal{P}'$ la population temporaire obtenue
Étape 7 : Reproduction
Appliquer les opérateurs de variation à la population $\mathcal{P}'$
Soit $\mathcal{P}_{t+1}$ la population résultante
Incrémenter le compteur de générations ($t \leftarrow t+1$) et retour à l'étape 2

Quelques précisions :

1. Règles de conception des empilements composites

La pratique industrielle de conception des empilements composites se déroule en trois étapes principales :

1. **Détermination du nombre total de plis et proportions pour chaque orientation** :
   - **Symétrie miroir** : les empilements doivent être symétriques par rapport au plan moyen pour éviter les couplages entre flexion et traction.
   - **Équilibrage** : un même nombre de plis orientés à $+\theta$ et $-\theta$ pour éviter les couplages plans.
   - **Règle des 10 %** : chaque orientation doit représenter au moins 10 % des plis pour éviter un comportement dominé par la matrice dans certaines directions.
2. **Choix de l'ordre de séquencement des plis** :
   - **Groupage** : limite le nombre de plis contigus de même orientation pour réduire les phénomènes d'endommagement sensibles à l'épaisseur des groupages de plis.
   - **Désorientation** : impose une désorientation maximale de 45ř entre deux plis adjacents pour minimiser les cisaillements interlaminaires.
   - **Tolérance aux dommages** : vise à améliorer la robustesse face aux défauts et impacts.
3. **Définition des reprises de plis pour des panneaux d'épaisseur variable** :
   - Ajustements nécessaires pour assurer la continuité et l'intégrité structurelle des panneaux tout en respectant les règles ci-dessus.

Ces règles proviennent de l'expérience industrielle et de documents de référence tels que le MIL-HDBK-17-3F.

2. Modifications des opérateurs pour l'algorithme évolutionnaire

Les modifications apportées aux opérateurs pour l'algorithme évolutionnaire incluent :

1. **Codage des solutions** :
   - Solutions encodées sous forme de vecteurs réels correspondant aux orientations des plis. Par exemple, un empilement est représenté par un vecteur où chaque coordonnée correspond à l'orientation d'un pli.
2. **Phase de reproduction** :
   - **Opérateur de croisement** : un croisement bipoint est utilisé pour échanger des séquences de gènes entre deux solutions parentales. Ce croisement est adapté pour imposer des séquences équilibrées.
   - **Opérateur de mutation** : consiste à modifier aléatoirement la valeur d'un gène, contrôlé par un paramètre d'amplitude. Cette mutation est appliquée de manière symétrique si nécessaire.

— **Opérateur de permutation** : implique une permutation circulaire aléatoire des plis dans une séquence choisie aléatoirement. Cette permutation est spécifique aux chromosomes d'empilement et impacte principalement les propriétés de flexion du stratifié.

7.4 Annexe 4 : Instances de test pour comparer les méthodes de prise en compte des contraintes

Pour effectuer les comparaisons, nous avons utilisé 5 exemples de problème MINLPs trouvés dans la bibliothèque des ressources MINLPs.

Pb1

$$\begin{cases} \min\ (-x - 2y) \\ 4x - 3y \leq 2 \\ y - 2x \leq 1 \\ 14y - 6x \leq 35 \\ x \in \mathbb{N}\ ,\ y \in \mathbb{N} \end{cases} \tag{17}$$

Pb2

Minimiser :

$$-i_1 - i_2 - i_3 - i_4 - i_5 - i_6 - i_7 - i_8 - i_9 - i_{10}$$

Sous les contraintes suivantes :

$i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10} \in [-1, 1]$, entiers

$$\begin{aligned} &i_9^2 - 0.987420882906575 \cdot i_9 + i_8^2 - 0.987420882906575 \cdot i_8 + i_7^2 - 0.987420882906575 \cdot i_7 \\ &+ i_6^2 - 0.987420882906575 \cdot i_6 + i_5^2 - 0.987420882906575 \cdot i_5 + i_4^2 - 0.987420882906575 \cdot i_4 \\ &+ i_3^2 - 0.987420882906575 \cdot i_3 + i_2^2 - 0.987420882906575 \cdot i_2 + i_1^2 - 0.987420882906575 \cdot i_1 \\ &+ i_{10}^2 - 0.987420882906575 \cdot i_{10} \leq 0 \end{aligned}$$

Pb3

Minimiser :

$$-0.5\left(100x_1^2 + 100x_2^2 + 100x_3^2 + 100x_4^2 + 100x_5^2\right) + 42x_1 + 44x_2 + 45x_3 + 47x_4 + 47.5x_5$$

Sous les contraintes :

$$\begin{aligned} & x_1, x_2, x_3, x_4, x_5 \in [0, 1] \\ & -20x_1 + 12x_2 + 11x_3 + 7x_4 + 4x_5 \leq 40 \end{aligned}$$

Pb4

$$\begin{cases} \min\ x_2^2 - 7x_2^2 - 12x_1 \\ 2x_1^4 + x_2 = 2 \\ x_1 \in [0, 2]\ ,\ x_2 \in [0, 3] \end{cases} \tag{18}$$

Pb5

$$\begin{cases} \min\ -x_1 - x_2 \\ 8x_1^3 - 2x_1^4 - 8x_1^2 + x_2 \leq 2 \\ 32x_1^3 - 4x_1^4 - 88x_1^2 + 96x_1 + x_2 \leq 36 \\ x_1 \in [0, 3]\ ,\ x_2 \in [0, 4] \end{cases} \tag{19}$$

7.5 Annexe 5 : Instances de test pour la comparaison des algorithmes mono-objectif

Problèmes utilisés pour comparer les algorithmes mono-objectif :

Pb1

Minimiser :

$$\begin{aligned} & x_1 \cdot (4 \cdot x_1 + 3 \cdot x_2 - x_3) \\ & + x_2 \cdot (3 \cdot x_1 + 6 \cdot x_2 + x_3) \\ & + x_3 \cdot (x_2 - x_1 + 10 \cdot x_3) \end{aligned}$$

Sous les contraintes suivantes :

$$\begin{aligned} & x_1, x_2, x_3, x_4 \in \mathbb{R}_{\geq 0} \\ & b_5, b_6, b_7, b_8 \in \{0, 1\} \\ & x_1 + x_2 + x_3 + x_4 = 1 \\ & 8 \cdot x_1 + 9 \cdot x_2 + 12 \cdot x_3 + 7 \cdot x_4 = 10 \\ & x_1 - b_5 \leq 0 \\ & x_2 - b_6 \leq 0 \\ & x_3 - b_7 \leq 0 \\ & x_4 - b_8 \leq 0 \\ & b_5 + b_6 + b_7 + b_8 \leq 3 \end{aligned}$$

Pb2

Minimiser :

$$-i_1 - i_2 - i_3 - i_4 - i_5 - i_6 - i_7 - i_8 - i_9 - i_{10}$$

Sous les contraintes suivantes :

$i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10} \in [-1, 1]$, entiers
$i_9^2 - 0.987420882906575 \cdot i_9 + i_8^2 - 0.987420882906575 \cdot i_8 + i_7^2 - 0.987420882906575 \cdot i_7$
$+ i_6^2 - 0.987420882906575 \cdot i_6 + i_5^2 - 0.987420882906575 \cdot i_5 + i_4^2 - 0.987420882906575 \cdot i_4$
$+ i_3^2 - 0.987420882906575 \cdot i_3 + i_2^2 - 0.987420882906575 \cdot i_2 + i_1^2 - 0.987420882906575 \cdot i_1$
$+ i_{10}^2 - 0.987420882906575 \cdot i_{10} \leq 0$

Pb3

Minimiser :

$$24.55 \cdot x_1 + 26.75 \cdot x_2 + 39 \cdot x_3 + 40.5 \cdot x_4$$

Sous les contraintes suivantes :

$$x_1, x_2, x_3, x_4 \geq 0$$
$$x_1 + x_2 + x_3 + x_4 = 1$$
$$-1.645 \cdot \sqrt{0.28 \cdot x_1^2 + 0.19 \cdot x_2^2 + 20.5 \cdot x_3^2 + 0.62 \cdot x_4^2}$$
$$+ 12 \cdot x_1 + 11.9 \cdot x_2 + 41.8 \cdot x_3 + 52.1 \cdot x_4 \geq 21$$
$$2.3 \cdot x_1 + 5.6 \cdot x_2 + 11.1 \cdot x_3 + 1.3 \cdot x_4 \geq 5$$

7.6 Annexe 6 : Correspondance entre exemples physiques et critères mathématiques

Problème physique	**Critères mathématiques**	**Exemples**
Structures complexes, beaucoup de choix pour la conception, comportements non-linéaires des matériaux	$Dd \geq 50\%$, $d \geq 50\%$, $CRG \leq 50\%$	Toits des stades en treillis, ponts suspendus
Structures complexes, beaucoup de choix pour la conception, comportements linéaires des matériaux	$Dd \leq 50\%$, $d \geq 50\%$, $CRG \geq 50\%$	Batiments en béton de plusieurs étages, ou ponts à travée
Structures avec peu de choix pour la conception, comportements des matériaux complexes à cerner	$Dd \geq 50\%$, $d \leq 50\%$, $CRG \leq 50\%$	Coque de bateau, béton précontraint
Structures avec peu de choix pour la conception, comportements des matériaux linéaires	$Dd \leq 50\%$, $d \leq 50\%$, $CRG \geq 50\%$	Ossatures en acier, ponts en béton armé
Structures avec de nombreux choix pour la conception, comportements des matériaux linéaires	$Dd \geq 50\%$, $d \leq 50\%$, $CRG \leq 50\%$	Antennes de télécommunication, tours de transmission
Structures avec peu de choix pour la conception, comportements des matériaux complexes	$Dd \leq 50\%$, $d \geq 50\%$, $CRG \geq 50\%$	Système amortissement véhicules, plateformes off-shore

TABLE 14 – Contextes physiques des critères mathématiques des problèmes

7.7 Annexe 7 : Algorithme du recuit simulé pour le voyageur de commerce

Algorithm 8 Recuit Simulé pour le Problème du Voyageur de Commerce

Data : Matrice des distances D, température initiale T_0, taux de refroidissement α, température minimale T_{min}

Result : Meilleure route R_{best} et sa distance D_{best}

Initialiser R_{current} comme une route aléatoire
Ajouter la ville de départ à la fin de R_{current}
Calculer D_{current} la distance de R_{current}
Définir $R_{\text{best}} \leftarrow R_{\text{current}}$ et $D_{\text{best}} \leftarrow D_{\text{current}}$
Définir $T \leftarrow T_0$
$k \leftarrow 0$
while $T > T_{min}$ **do**
  $k \leftarrow k + 1$
  Générer les voisins de R_{current}, Sélectionner un voisin aléatoire R_{new}
  Calculer D_{new} comme la distance de R_{new}
  if $D_{new} < D_{current}$ **then**
    $R_{\text{current}} \leftarrow R_{\text{new}}$
    $D_{\text{current}} \leftarrow D_{\text{new}}$
  end
  else
    Calculer la probabilité d'acceptation $p \leftarrow \exp((D_{\text{current}} - D_{\text{new}})/T)$
    if $p > random(0, 1)$ **then**
      $R_{\text{current}} \leftarrow R_{\text{new}}$
      $D_{\text{current}} \leftarrow D_{\text{new}}$
    end
  end
  if $D_{current} < D_{best}$ **then**
    $R_{\text{best}} \leftarrow R_{\text{current}}$
    $D_{\text{best}} \leftarrow D_{\text{current}}$
  end
  $T \leftarrow \alpha T$
end
return R_{best}, D_{best}, k

7.8 Annexe 8 : Exemple Outer Approximation

$$\begin{cases} \min\ y_1 + y_2 + x_1^2 + x_2^2 \\ (x_1 - 2)^2 \leq 0 \\ x_1 - 2y_1 \geq 0 \\ x_1 - x_2 - 3(1 - y_1) \geq 0 \\ x_1 + y_1 - 1 \geq 0 \\ x_2 - y_2 \geq 0 \\ x_1 + x_2 \geq 3y_1 \\ y_1 + y_2 \geq 1 \\ 0 \leq x_1 \leq 4 \\ 0 \leq x_2 \leq 4 \\ y_1, y_2 \in \{0, 1\} \end{cases} \tag{20}$$

Initalisation des bornes et des solutions :

On pose $Z_U = +\infty$, $Z_L = -\infty$ et $y_1 = y_2 = 1$.

Résolution du Problème de Relaxation :

On résout le nouveau problème qui est maintenant NLP (Non-Linear Programming) :

$$\begin{cases} \min\ 2 + x_1^2 + x_2^2 \\ (x_1 - 2)^2 \leq 0 \\ x_1 - 2 \geq 0 \\ x_1 - x_2 \leq 0 \\ x_1 \geq 0 \\ x_2 - 1 \geq 0 \\ x_1 + x_2 \geq 3 \\ 0 \leq x_1 \leq 4 \\ 0 \leq x_2 \leq 4 \end{cases} \tag{21}$$

On trouve $x_1 = 2$, $x_2 = 1$, $Z_U = 7$.

Actualisation des bornes et linéarisation des contraintes

On linéarise l'objectif et les contraintes par une formule de Taylor autour du point que l'on a trouvé juste avant. On trouve en particulier : $f(x) \approx 5 + 4(x_1 -$

$2) + 2(x_2 - 1)$.

On résout alors le problème suivant :

$$\begin{cases} \min \alpha \\ \alpha - x_2 \leq 0 \\ x_1 - 2y_1 \geq 0 \\ x_1 - x_2 - 3(1 - y_1) \geq 0 \\ x_1 + y_1 - 1 \geq 0 \\ x_2 - y_2 \geq 0 \\ x_1 + x_2 \geq 3y_1 \\ y_1 + y_2 \geq 1 \\ 0 \leq x_1 \leq 4 \\ 0 \leq x_2 \leq 4 \\ y_1, y_2 \in \{0, 1\} \end{cases} \tag{22}$$

On trouve la solution : $x_1 = 2$, $x_2 = 1$, $y_1 = 1$, $y_2 = 0$ et $Z_L = 6$.

Itérations Successives

Le processus d'approximation extérieure est itératif. Nous continuons à résoudre des problèmes de relaxation linéaire successifs, jusqu'à ce que les bornes supérieures et inférieures soient égales.

7.9 Annexe 9 : Exemple Branch and Bound

$$\begin{cases} \max x + 2y \\ 4x - 3y - 2 \leq 0 \\ -2x + y - 1 \leq 0 \\ -6x + 14y - 35 \leq 0 \\ (x, y) \in \mathbb{R}^2 \\ x, y \text{ entiers positifs} \end{cases} \tag{23}$$

On le relaxe en :

$$\begin{cases} \max\ x + 2y \\ 4x - 3y - 2 \leq 0 \\ -2x + y - 1 \leq 0 \\ -6x + 14y - 35 \leq 0 \\ (x, y) \in \mathbb{R}^2 \\ x, y \text{ positifs} \end{cases} \tag{24}$$

On trouve alors la solution à ce problème par une résolution avec un algorithme classique d'optimisation sous contraintes et on trouve $(x_{sol}, y_{sol}) = (3.5, 4)$ (cf 23) :

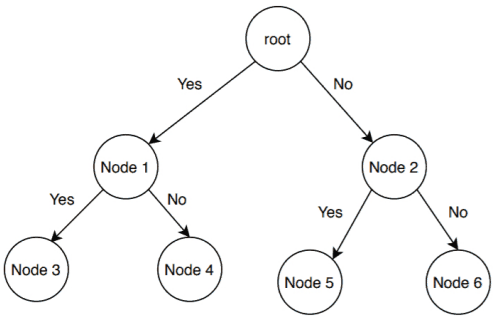


FIGURE 22 – Principe du Branch and Bound

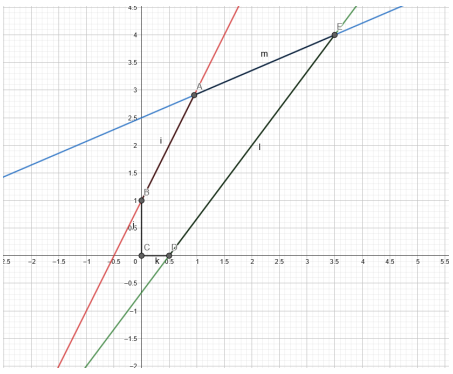


FIGURE 23 – Domaine du problème

Alors, on réalise une disjonction de cas pour obtenir deux nouveaux MINLPs :

P_{01}	P_{02}
$\max\ x + 2y$	$\max\ x + 2y$
$4x - 3y - 2 \leq 0$	$4x - 3y - 2 \leq 0$
$-2x + y - 1 \leq 0$	$-2x + y - 1 \leq 0$
$-6x + 14y - 35 \leq 0$	$-6x + 14y - 35 \leq 0$
x, y entiers positifs	x, y entiers positifs
$x \leq 3$	$x \geq 4$

TABLE 15 – Disjonction des problèmes

On répète alors l'opération jusqu'à trouver des solutions possibles. Bien sûr, les disjonctions de cas permettent d'exclure des branches, ici par exemple il est clair que $x \geq 4$ n'est pas acceptable, on peut donc l'exclure.

7.10 Notations et vocabulaire utiles

Voici quelques notations utilisées dans ce rapport qui peuvent être utiles à la compréhension de certaines parties :

- MINLP : Mixed Integer Non-Linear Programming
- NLP : Non-Liner Programming
- MILP : Mixed Integer Linear Programming
- GA : Genetic Algorithm
- Heuristique : Une heuristique est une méthode ou une technique utilisée pour trouver une solution approximative à un problème lorsque les méthodes classiques sont impraticables en raison de la complexité ou du temps nécessaire pour trouver la solution exacte.

7.11 Codes Python

Tous les codes Python utilisés et réalisés, ainsi que la bibliographie complète de ce travail, se trouve dans mon répertoire github : Répertoire [10]

Références

[1] Joseph Ayas and M André Viau. La recherche tabou. *Notes de cours*, 12, 2004.

[2] Michaël Baudin, Anne Dutfoy, Bertrand Iooss, and Anne-Laure Popelin. *OpenTURNS : An Industrial Software for Uncertainty Quantification in Simulation*, pages 1–38. Springer International Publishing, Cham, 2016.

10. https://github.com/OctaveTougeron/optim_internship

[3] Logan Beal, Daniel Hill, R Martin, and John Hedengren. Gekko optimization suite. *Processes*, 6(8):106, 2018.

[4] Pietro Belotti, Jon Lee, Leo Liberti, François Margot, and Andreas Wächter. Branching and bounds tightening techniques for non-convex minlp. *Optimization Methods and Software*, 24:597–634, 10 2009.

[5] Martin Philip Bendsoe and Ole Sigmund. *Topology optimization : theory, methods, and applications*. Springer Science & Business Media, 2013.

[6] Abdelouhab BENHAMA and Yacine BENKHELOUF. Etude comparative des algorithmes génétiques multi-objectifs. *Mémoire de Master en Automatique, Université Abderrahmane MIRA de Bejaia*, 2015.

[7] Gérard Berthiau. *La méthode du recuit simulé pour la conception des circuits électroniques : adaptation et comparaison avec d'autres méthodes d'optimisation*. PhD thesis, Châtenay-Malabry, Ecole centrale de Paris, 1994.

[8] J. Blank and K. Deb. pymoo : Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[9] Marc Bonnet, Attilio Frangi, and Christian Rey. *The finite element method in solid mechanics*. McGraw-Hill Education New York, 2014.

[10] Radia Bouabdallah. *Optimisation en présence d'incertitudes de la répartition de la charge dans les problèmes elliptiques : Application à la mécanique des structures.* PhD thesis, Université Montpellier, 2021.

[11] Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, David L Woodruff, et al. *Pyomo-optimization modeling in python*, volume 67. Springer, 2021.

[12] Juan J Durillo and Antonio J Nebro. jmetal : A java framework for multi-objective optimization. *Advances in engineering software*, 42(10):760–771, 2011.

[13] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP : Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[14] Jonathan Guerra. *Optimisation multi-objectif sous incertitudes de phénomènes de thermique transitoire*. Theses, INSTITUT SUPERIEUR DE L'AERONAUTIQUE ET DE L'ESPACE (ISAE), October 2016.

[15] G Guillopé. Optimisation sous contrainte. *Laboratoire de mathématiques Jean Leray, Département de mathématiques, Nantes University, Nantes, France*, 2016, 2015.

[16] COIN-OR Home. Bonmin users' manual.

[17] François-Xavier Irisarri. *Stratégies de calcul pour l'optimisation multiobjectif des structures composites*. PhD thesis, Université Paul Sabatier-Toulouse III, 2009.

[18] Jan Kronqvist, David Bernal Neira, Andreas Lundell, and Ignacio Grossmann. A review and comparison of solvers for convex minlp. *Optimization and Engineering*, 20, 06 2019.

[19] Federico Marini and Beata Walczak. Particle swarm optimization (pso). a tutorial. *Chemometrics and Intelligent Laboratory Systems*, 149:153–165, 2015.

[20] Florian Mitjana. *Optimisation topologique de structures sous contraintes de flambage*. PhD thesis, Université Paul Sabatier-Toulouse III, 2018.

[21] Pablo Andrés Muñoz-Rojas. *Optimization of structures and components*. Springer, 2013.

[22] Thomas Stützle, Marco Dorigo, et al. Aco algorithms for the traveling salesman problem. *Evolutionary algorithms in engineering and computer science*, 4:163–183, 1999.

[23] Nguyen Van Thieu and Seyedali Mirjalili. Mealpy : An open-source library for latest meta-heuristic algorithms in python. *Journal of Systems Architecture*, 2023.

[24] Charlie Vanaret, David Gianazza, Jean-Baptiste Gotteland, and Nicolas Durand. Résolution de conflits aériens par un algorithme à évolution différentielle. In *ROADEF 2012, 13ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, pages pp–550, 2012.

[25] Liding Xu. *Relaxation methods for mixed-integer nonlinear programming*. Theses, Institut Polytechnique de Paris, December 2023.

[26] Ru Xue and Zongsheng Wu. A survey of application and classification on teaching-learning-based optimization algorithm. *IEEE Access*, 8:1062–1079, 2019.