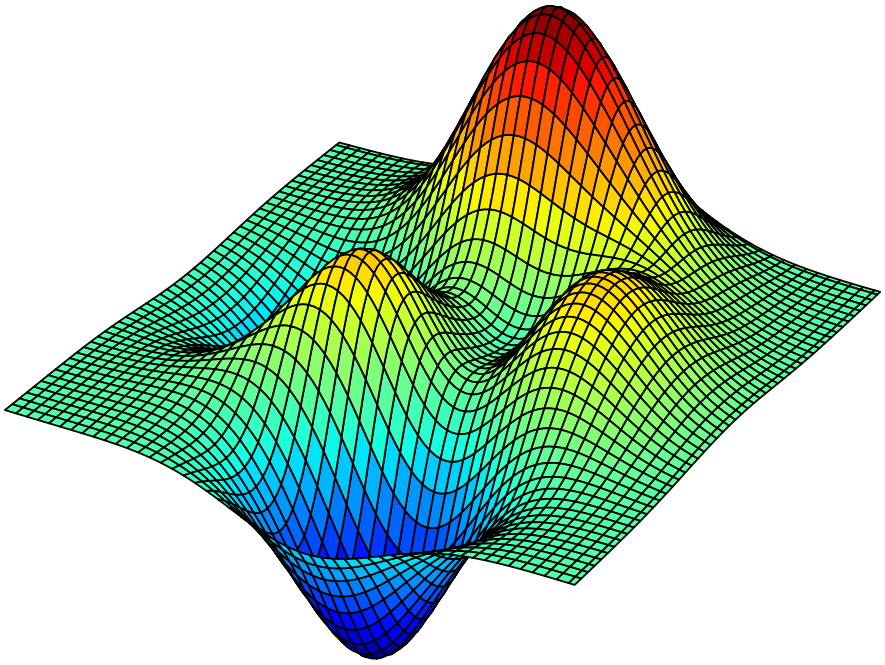
Chapitre 1: Optimisation non linéaire à variables réelles



I Optimisation, minimisation et maximisation de fonctions réelles

1.1 Problèmes d'optimisation mono-critère

L'optimisation est une branche des mathématiques appliquée à de nombreux domaines tels que la conception et la modélisation pour l'ingénierie et l'industrie. La complexité des problèmes posés demande l'utilisation d'algorithmes numériques performants. La qualité des résultats et des prédictions dépend de la pertinence du modèle, du choix des variables d'optimisation et de l'algorithme. L'optimisation mono-critère est un problème mathématique qui consiste à minimiser (ou maximiser) une fonction objectif scalaire $F(\boldsymbol{x})$ sous des fonctions contraintes d'inégalités $g_i(\boldsymbol{x})$ et/ou d'égalité $h_i(\boldsymbol{x})$ avec des variables réelles $\boldsymbol{x}$ bornées ou non.

$$min\ \mathrm{F(x)} \qquad (1\text{-}1)$$

Sujet à :

$$g_i(\boldsymbol{x}) \geq 0\ pour\ i\ \in [0 \dots mc_{ineq}]$$
$$h_i(\boldsymbol{x}) = 0\ pour\ i \in [0 \dots mc_{eq}]$$
$$\boldsymbol{x}\ \in \{x_{j,min}, x_{j,\max}\}_n$$

Si le problème est une maximisation, il suffit de choisir de minimiser -$F(\boldsymbol{x})$. Généralement, la démarche d'optimisation s'attache à trouver les optimums globaux de la fonction objectif. Cette dernière condition est particulièrement difficile pour les problèmes complexes et on ne peut pas toujours s'assurer que la solution trouvée est un minimum global. Explorer l'espace des variables par un maillage représente un coup en calcul trop important et il est nécessaire d'utiliser des algorithmes d'optimisation, au prix de perdre la garantie d'obtenir un optimum global. Dans le cas présent, rien n'affirme que le minimum trouvé par *Golden-Search*.

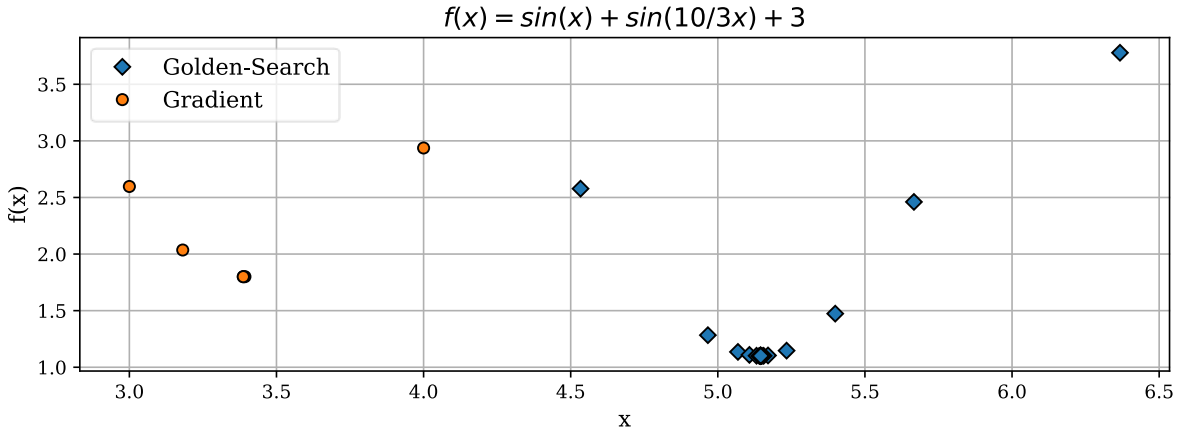


Figure 1 : Minimisation sans représentation

1.2 Classification des algorithmes

Les algorithmes se divisent en plusieurs classes ou familles avec des propriétés différentes. Les méthodes déterministes sont des algorithmes dont le comportement ne varie pas pour une même configuration initiale. Dans ce premier ensemble, on discerne des méthodes directes sans gradient qui utilisent un schéma de recherche de l'optimum sans utiliser les propriétés de dérivabilité de la fonction. La seconde catégorie des méthodes déterministes est l'optimisation analytique souvent basée sur le gradient et les dérivées de la fonction. Ces méthodes sont des algorithmes d'optimisation locale avec ou sans contraintes. La seconde branche concerne les algorithmes stochastiques ou non-déterministes. Ces méthodes s'appuient sur des mécanismes probabilistes et aléatoires pour rechercher l'optimum d'une fonction. L'exécution successive de ces programmes peut mener à des résultats différents. En outre, les méthodes stochastiques ont une bonne capacité à approcher l'optimum global d'un problème.

Tableau 1 : Liste non-exhaustive des algorithmes d'optimisation réelle

Algorithme	Description	Type de recherche
Algorithmes directes (pas de gradient)		
Golden search	Méthode 1D basée sur le nombre d'or	Locale
NelderMead	Méthode N-D basée sur l'évolution d'un simplex	Locale
Powell	Méthode N-D basée sur l'optimisation successive de directions conjuguées	Locale
Algorithmes analytiques avec gradient		
Gradient	Méthode basée sur la direction du gradient	Locale
Gradient conjugué	La direction de descente dépend du gradient et des dernières itérations	Locale
Newton	Cherche à annuler le gradient par la méthode de Newton. Demande la connaissance de la matrice hessienne de la fonction.	Locale
BFGS (quasi-Newton)	Méthode quasi-Newton où la hessienne est approchée par itération successives.	Locale
Région de confiance	Ces méthodes utilisent une approximation quadratique donnée par le gradient et la hessienne pour résoudre un sous-problème quadratique plus simple.	Locale
SLP, SLQP, SQP	Programmation linéaire ou/et quadratique successive. Méthodes basées sur la résolution successive d'un problème linéaire ou quadratique incluant des contraintes par formulation.	Locale et contrainte
Algorithmes stochastiques		
Algorithme évolutionnaire	Inspiré de la théorie de l'évolution, il recherche la meilleure solution par adaptation d'un ensemble de candidats.	Globale
Algorithme génétique	Méthode évolutionnaire utilisant les propriétés de reproduction, mutation et adaptation d'un ensemble de solutions appelé population.	Globale
Recuit simulé	S'inspire de la métallurgie, la solution tend à transiter vers des états minimisant la fonction. Une solution de moins bonne qualité est acceptée avec une probabilité s'appuyant sur la statistique de Maxwell-Boltzmann.	Globale

1.3 La librairie *scipy.optimize* et nouvelles méthodes

La bibliothèque *scipy* possède un module très complet pour la résolution de problèmes d'optimisation monocritère.

scipy.optimize

- 1-D minimisation sans contrainte (**minimize_scalar**) ;
- N-D minimisation locale sans contrainte (**minimize**) :
  - Sans évaluation de gradient :
    - Nelder-Mead Simplex (Nelder-Mead) ;
    - Powell (Powell) ;
  - Avec gradient et/ou hessienne :
    - Gradient conjugué (CG ; Newton-CG)
    - Broyden Fletcher Goldfarb Shanno (BFGS ; L-BFGS-B) ;
    - Région de confiance (trust-ncg ; trust-krylov ; trust-exact) ;
- N-D minimisation locale avec contrainte :
  - Avec gradient et/ou hessienne :
    - Région de confiance (trust-constr) ;
    - Programmation quadratique séquentielle (SLSQP) ;
  - Sans gradient :
    - Optimisation contrainte par approximation linéaire (COBYLA) ;
- N-D minimisation globale sans contrainte :
    - Recuit simulé rapide (dual_annealing) ;
    - Evolution différentielle (differential_evolution) ;
- Programmation linéaire (**linprog**) ;
- Moindre carrés (**least_squares**) ;

Dans le cadre d'une démarche d'ingénierie, les problèmes d'optimisation sont souvent contraints avec des bornes définies et des propriétés de dérivabilité inconnues. Afin de concevoir des algorithmes modifiés, le problème mathématique est donné par l'équation (1-1).

Les améliorations recherchées sont :

- Minimisation par gradient ou BFGS bornée et contrainte ;
- Minimisation sans gradient bornée et contrainte ;
- Minimisation scalaire bornée et contrainte ;
- Minimisation globale bornée et contrainte ;
- Inclure l'appel d'une pré-routine de l'utilisateur avant d'évaluer la fonction et les contraintes. Par exemple, l'utilisation d'une routine qui résout un modèle, puis l'objectif et les contraintes utilisent ce résultat sans relancer les calculs.

1.4 Sensibilité, robustesse et convergence des méthodes d'optimisation

Un algorithme d'optimisation est conditionné par divers paramètres de contrôle et par les conditions initiales imposées. La sensibilité d'une méthode est définie par l'effet d'un changement de conditions initiales sur les résultats finaux. La robustesse est la capacité de l'algorithme, dans une configuration donnée, à trouver l'optimum de fonctions très différentes. L'algorithme parfait donnerait quel que soit le problème posé, un optimum global pour n'importe quelle configuration de contrôle.

Ces méthodes sont itératives et il est nécessaire de s'assurer de la convergence vers un état optimal. Un aspect important des algorithmes d'optimisation est le choix du critère de convergence et l'observation de la dynamique d'évolution de la solution.

1.5 Poser correctement un problème d'optimisation

Quelques règles permettent de simplifier la résolution des problèmes d'optimisation :

- Les fonctions (objectif et contraintes) sont linéaires : → programmation linéaire ;
- Essayer un changement ou un ajout de variables et de contraintes pour rendre le problème totalement linéaire : → programmation linéaire ;
- La fonction est continue et dérivable : → méthode analytique avec gradient approché ;
- La dérivée est connue : → méthode analytique avec gradient exacte ;
- Pas de connaissance de dérivabilité : → méthode analytique sans gradient ;
- Les contraintes sont fortes (égalités) et non-linéaires, mais dérivables → SLQP, région de confiance ou Lagrangien ;
- La fonction possède de nombreux minima locaux : → recherche globale ;
- Améliorer la recherche locale en substituant des contraintes et des variables. Diminué le nombre de variables augmente la complexité de la fonction mais n'affecte pas les algorithmes de recherche globale ;

1.6 Gestion des contraintes

1.6.1 Programmation linéaire

Le module *linprog* de *scipy* permet de résoudre un problème d'optimisation linéaire. Les fonctions contraintes et objectifs sont linéaires et la résolution est réalisée par différentes méthodes (*interior-point, simplex …*). Le problème est formulé de la manière suivante, *scipy* reconditionne le problème de façon canonique :

$$\min \boldsymbol{c}^T \boldsymbol{x}$$

$$\text{s.t } [\boldsymbol{A}_{eq}] \cdot \boldsymbol{x} = \boldsymbol{b}_{eq}$$

$$\left[\boldsymbol{A}_{ineq}\right] \cdot \boldsymbol{x} \leq \boldsymbol{b}_{ineq}$$

$$\boldsymbol{x}_{min} < \boldsymbol{x} < \boldsymbol{x}_{max}$$

1.6.2 Multiplicateur de Lagrange et contraintes d'égalité

La méthode du multiplicateur de Lagrange permet de transformer un problème d'optimisation différentiable à n variables avec m contraintes d'égalité, en un système de $n + m$ équations. L'opérateur Lagrangien est défini comme suivant avec $\boldsymbol{\lambda}$ le vecteur des multiplicateurs de Lagrange :

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \cdot \boldsymbol{c}_{eq}(\boldsymbol{x})$$

On peut minimiser f sous contrainte :

$$\min f(\boldsymbol{x})$$

$$\text{s.t. } \boldsymbol{c}_{eq}(\boldsymbol{x}) = \boldsymbol{0}$$

Une solution x_0 implique qu'il existe $\boldsymbol{\lambda}_0$ tel que (théorème) :

$$\boldsymbol{\nabla}\,\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\lambda}_0) = \boldsymbol{\nabla} f(\boldsymbol{x}_0) + \boldsymbol{\lambda}_0{}^T \cdot \boldsymbol{\nabla} \boldsymbol{c}_{eq}(\boldsymbol{x}_0) = \boldsymbol{0}$$

Alors, le problème se résume à résoudre le système de $m + n$ équations :

$$\begin{cases} \boldsymbol{\nabla} f(\boldsymbol{x}_0) + \boldsymbol{\lambda}_0{}^T \cdot \boldsymbol{\nabla} \boldsymbol{c}_{eq}(\boldsymbol{x}_0) = 0 \\ \boldsymbol{c}_{eq}(\boldsymbol{x}_0) = 0 \end{cases}$$

Idée de démonstration :

L'ensemble des contraintes forme un sous-espace $\boldsymbol{S}$ de dimension $n - m$. Pour trois variables et une contrainte, $\boldsymbol{S}$ est une surface, pour deux contraintes $\boldsymbol{S}$ est une courbe. Pour un point $\boldsymbol{M} \epsilon\, \boldsymbol{S}$ les différentielles de f et $\boldsymbol{c}$ s'écrivent : $df = \boldsymbol{\nabla} f \cdot d\boldsymbol{M}$ et $dc_i = \boldsymbol{\nabla} c_i \cdot d\boldsymbol{M}$

Si on restreint $d\boldsymbol{M} \epsilon\, \boldsymbol{S}$, où par définition $\boldsymbol{c} = \boldsymbol{0}$, alors $dc_i = 0 = \boldsymbol{\nabla} c_i \cdot d\boldsymbol{M}$, donc les gradients des contraintes sont orthogonaux à $\boldsymbol{S}$.

Maintenant si ce point $\boldsymbol{M} = \boldsymbol{x}_0$, alors c'est un extrema de f restreint sur $\boldsymbol{S}$, donc quelques soit le déplacement $d\boldsymbol{M} \epsilon\, \boldsymbol{S}$ autour de $\boldsymbol{M}$, la fonction f ne peut diminuer ou s'accroitre davantage par définition d'un extremum. On comprend ici que la différentielle de f est nulle en ce point : $df = \boldsymbol{\nabla} f \cdot d\boldsymbol{M} = 0$. Le gradient de f est orthogonal à $\boldsymbol{S}$.

On en conclut qu'une combinaison linéaire des gradients des contraintes et le gradient de f sont colinéaires au point $\boldsymbol{x}_0$ solution du problème.

$$d\mathcal{L} = \boldsymbol{\nabla}\mathcal{L} \cdot d\boldsymbol{M} = \boldsymbol{\nabla} f \cdot d\boldsymbol{M} + \boldsymbol{\lambda}^T \cdot \boldsymbol{\nabla} \boldsymbol{c} \cdot d\boldsymbol{M} = df + \sum \lambda_i dc_i = 0$$

Donc il existe $\boldsymbol{\lambda}_0$ tel que :

$$\boldsymbol{\nabla}\,\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\lambda}_0) = \boldsymbol{\nabla} f(\boldsymbol{x}_0) + \boldsymbol{\lambda}_0{}^T \cdot \boldsymbol{\nabla} \boldsymbol{c}_{eq}(\boldsymbol{x}_0) = \boldsymbol{0}$$

Une extension permet de formuler le Lagrangien avec des contraintes d'inégalité. Néanmoins, cette méthode impose de résoudre un système d'équations basé sur les gradients. Si les propriétés de dérivabilité ne sont pas vérifiées et que les expressions des dérivées premières et secondes ne sont pas connues alors l'algorithme semble très compliqué à mettre en œuvre avec des approximations successives.

1.6.3 Pénalisation des contraintes

Cette méthode consiste à transformer une minimisation contrainte en un problème sans contrainte, sans modifier les propriétés de dérivabilité et de continuité. Il existe plusieurs formes de pénalisation avec comme distinction deux propriétés : exacte/inexacte et intérieure/extérieure. Une méthode polyvalente est la pénalisation quadratique qui est exacte extérieure. Le problème (1-1) devient :

$$\min f_{penal}(\boldsymbol{x}) = f(\boldsymbol{x}) + r \cdot \sum_i \min(g_i(\boldsymbol{x}), 0)^2 + r \cdot \sum_i h_i(\boldsymbol{x})^2$$

$$\boldsymbol{x} \in \{x_{j,min}, x_{j,\max}\}_n$$

Le facteur r permet de modifier la pénalisation des contraintes. Ce paramètre doit être ajusté en fonction de la méthode de résolution et des amplitudes des contraintes et des objectifs. Pour cela, il faut essayer de connaître les variations des fonctions, normer les contraintes et ajuster la valeur du facteur de pénalisation. Cette méthode très simple à mettre en œuvre peut engendrer un mauvais conditionnement du problème. Une alternative est d'augmenter le facteur r pendant les itérations de résolution. Une

dernière solution est de remplacer la fonction objectif dans l'équation précédente par l'opérateur Lagrangien. Les multiplicateurs sont estimés à chaque itération. Cette méthode s'appelle le Lagrangien augmenté [1].

1.7 Approximation des dérivées et des gradients

Pour approcher numériquement la dérivée d'une fonction f, la première solution est d'utiliser les différences finies. La première approximation est moins précise que la seconde, mais elle permet d'optimiser le nombre d'appels de la fonction en utilisant la valeur de $f(x)$ souvent déjà connue.

- $\frac{df}{dx} \approx \frac{f(x+h)-f(x)}{h}$ ordre 1 *(forward finite-difference)*;
- $\frac{df}{dx} \approx \frac{f(x+h)-f(x-h)}{2h}$ ordre 2 *(central finite-difference)* ;

Une autre méthode utilise une valeur complexe. L'approximation est d'ordre 2 et permet d'optimisation le nombre d'appels de la fonction f.

- $\frac{df}{dx} = \Im[f(x+jh)]$

Avec $\Im$ la partie imaginaire et $h \approx 1 \cdot 10^{-9}$ le pas.

Il faut faire attention lors de l'approximation des dérivées d'une fonction pénalisée et ne pas approcher les gradients des contraintes. L'erreur d'approximation est transformée au carré et multipliée par le facteur de pénalisation. Il est préférable d'approchée la dérivée directement sur la fonction pénalisée.

II Algorithmes analytiques d'optimisation locale

1 Optimisation scalaire

1.1.1 Méthode du nombre d'or

La méthode du nombre d'or ressemble à la méthode de la dichotomie, mais le point sonde n'est pas le centre d'un segment, mais utilise le ratio du nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$. L'algorithme est restreint aux fonctions unimodales, c'est-à-dire qui ne possèdent qu'un extremum local dans l'intervalle $[x_{min}, x_{max}]$. Dans le cas d'une fonction multimodale, l'algorithme retourne un extremum local sans garantir que ce soit l'extremum global sur l'intervalle.

```
def goldenSearch(f,xmin,xmax):

    xa,xb = xmin,xmax
    alpha_1 = (3-sqrt(5))/2
    alpha_2 = 1-alpha_1

    x1 = xa + alpha1*(xb-xa)
    x2 = xa + alpha2*(xb-xa)
    f1 = f(x1)
    f2 = f(x2)

    deltax = (xb-xa)
```

```
    while condition arret :
        if f1 < f2:
            xb = x2
            x2 = x1
            f2 = f1
            deltax = alpha2 * deltax
            x1 = xa + alpha1 * deltax
            f1 = f(x1)

        else:
            xa = x1
            x1 = x2
            f1 = f2
            deltax = alpha2 * deltax
            x2 = xa + alpha2 * deltax
            f2 = f(x2)

    if f1 < f2:
        xmin = (xa + x2)/2
    else:
        xmin = (xb + x1)/2
    return xmin
```

1.1.2 Méthode du gradient

Pour préparer la méthode du gradient en dimensions supérieures, il est utile de l'introduire en dimension scalaire. Cet algorithme utilise la valeur de la dérivée df de f pour désigner une direction dans laquelle rechercher une solution. En dimension 1D, cette direction est donnée par $p_k = -df_k$. A chaque itération on effectue une recherche linéaire pour évaluer la nouvelle valeur de x_{k+1}.

```
def gradient1D(f,x0,xmin,xmax):
    xk = x0

    while condition arret :
        fk = f(xk)
        dfk = df(xk)
        pk = -dfk
        tk = recherche_lineaire(f,xk,fk,dfk)
        xknew = xk + tk*pk

        xk = xknew
```

Comme cette version est contrainte entre x_{min} et x_{max} le nouveau itéré et la recherche linéaire doivent être bornés. Aussi, la recherche peut admettre un gradient non nul si le minimum se trouve sur une frontière. L'algorithme est très similaire à celui appliqué en dimensions supérieures.

1.1.3 Recherche linéaire, condition d'Armijo et critère de Wolfe

Un algorithme de recherche linéaire est nécessaire pour les méthodes de gradient, gradient conjugué et BFGS. La recherche linéaire est un sous-problème 1D posé de la manière suivante :

$$\min \varphi_k(t) = f(x_k + t \cdot p_k)$$

La résolution exacte de ce problème est trop couteuse pour être envisageable. Le critère de Wolfe est un ensemble d'inégalités qui permet d'optimiser la valeur du pas t_k lors d'une recherche linéaire. La première inégalité est désignée comme condition d'Armijo et l'autre par condition de courbure.

$$f(\boldsymbol{x}_k + t \cdot \boldsymbol{p}_k) \leq f_k + t\, c_1\, \boldsymbol{p}_k^T \cdot \boldsymbol{\nabla} f_k$$

$$p_k^T \cdot \boldsymbol{\nabla} f(\boldsymbol{x}_k + t \cdot \boldsymbol{p}_k) \geq c_2\, \boldsymbol{p}_k^T \cdot \boldsymbol{\nabla} f_k$$

Avec $0 < c_1 < c_2 < 0$.

La condition d'Armijo assure que le pas sélectionné permet de diminuer f en dessous de la valeur obtenue par interpolation linéaire (gradient affecté d'un facteur c_1). La seconde équation permet d'assurer que le taux d'accroissement de f est plus grand que précédemment (approche d'un minimum local). La réalisation de la recherche n'est pas intuitive. Le codage d'un algorithme de recherche est possible, mais il préférable d'utiliser la routine de *scipy.optimize.line_search*. Dans *scipy* les valeurs par défaut des paramètres sont $c_1 = 0.001$ et $c_2 = 0.9$.

On note :

- $gfk = \nabla f(x_k)$ le gradient au point k ;
- $\varphi_0 = \varphi(0) = f_k = f(x_k)$ l'évaluation au point k ;
- $d\varphi_0 = \ p_k^T \cdot \nabla f_k$ la pente au point k suivant la direction p_k ;
- $\varphi(t) = f(x_k + tp_k)$ et $d\varphi(x_k + tp_k) = \ p_k^T \cdot \nabla f(x_k + tp_k)$.

Recherche linéaire 1 : backtracking armijo

La recherche linéaire prend un pas de départ puis le diminue jusqu'à valider la condition d'Armijo. Cet algorithme a l'avantage d'être très simple mais inutilisable dans la pratique car il fournit des valeurs très faibles de progression. On parle de *backtracking line search*.

```
def backtracking_linesearch(f,xk,fk,gfk,pk):
    t = 1.0
    dphi = pk.dot(gfk)
    while f(xk+t*pk) <= fk + c1*t*dphi : #Armijo condition
        t = t/2
```

Recherche linéaire 2 : interpolation quadratique et cubique

Une version améliorer de la recherche utilise des interpolations paraboliques et cubiques pour s'approcher d'un minimum. Ces algorithmes sont légèrement modifiés pour contraindre la recherche linéaire entre les bornes limites des variables.

```
def linesearch(f,xk,gfk,pk):
    t0 = 1.0
    #Etape 1
    phi_0 = fk
    phi_t0 = f(xk+t0*pk)
    dphi_0 = pk.dot(gfk)
    if phi_t0 <= (phi_0+t0*c1*dphi_0) : #Armijo sur t0
        return t0
    #Etape 2 : Interpolation quadratique ax**2 + b*x + c
    a = ( phi_t0-t0*dphi_0-phi_0)/t0**2
    b = dphi_0
    t1 =  -b/(2*a)
    phi_t1 = f(xk+t1*pk)
    if  phi_t1 <= (phi_0+t1*c1*dphi_0) : #Armijo sur t1
        return t1
```

```
    #Etape 3 : interpolation cubique a*x**3 + b*x**2 + c*x + d
    iter = 0
    while t1 > 0 and iter < maxIter :
        determinant = t0**2 * t1**2 * (t1-t0)
        a = t0**2 * (phi_t1 - phi_0 - dphi_0*t1) - \
            t1**2 * (phi_t0 - phi_0 - dphi_0*t0)
        a = a / determinant
        b = -t0**3 * (phi_t1 - phi_0 - dphi_0*t1) + \
            t1**3 * (phi_t0 - phi_0 - dphi_0*t0)
        b = b / determinant
        t2 = (-b + np.sqrt(abs(b**2 - 3 * a * dphi_0))) / (3.0*a)
        phi_t2 = f(xk+t2*pk)

        if phi_t2 <= (phi_0+t2*c1*dphi_0):
            return t2
        t1 = t2 ; phi_t1 = phi_t2
    return t1
```

Recherche linéaire 3 : bissection avec critère de Wolfe

La condition d'Armijo n'est pas assez forte pour garantir la convergence des algorithmes de gradient ou quasi-Newton. On peut utiliser un algorithme de bissection entre $t = 0$ et $t = t_0$ afin de garantir les conditions du critère de Wolfe.

```
def wolfeLinesearch(f,gf,xk,gfk,pk):
    t0 = 1.0
    #Etape 1
    phi_0 = fk
    phi_t0 = f(xk+t0*pk)
    dphi_0 = pk.dot(gfk)
    dphi_t0 = pk.dot(gf(xk+t0*pk))
    #Wolfe sur t0
    if phi_t0 <= (phi_0+t0*c1*dphi_0) and dphi_t0 >= c2*dphi_0 :
        return t0
    #Etape 2 : Interpolation quadratique ax**2 + b*x + c
    a = ( phi_t0-t0*dphi_0-phi_0)/t0**2
    b = dphi_0
    t1 =  -b/(2*a)
    phi_t1 = f(xk+t1*pk)
    dphi_t1 = pk.dot(gf(xk+t1*pk))
    #Wolfe sur t1
    if  phi_t1 <= (phi_0+t1*c1*dphi_0) and dphi_t1 >= c2*dphi_0 :
        return t1
    #Etape 3 : Bissection
    iter = 0
    ta = 0
    tb = t0
    while iter < maxIter :
        if phi_t1 > (phi_0+t1*c1*dphi_0) :
            tb = t1
            t1 = (ta+tb)/2
            phi_t1 = f(xk+t1*pk)
            dphi_t1 = pk.dot(gf(xk+t1*pk))
        else  :
            ta = t1
            t1 = (ta+tb)/2
            phi_t1 = f(xk+t1*pk)
```

```
            dphi_t1 = pk.dot(gf(xk+t1*pk))

        if phi_t1 <= (phi_0+t1*c1*dphi_0) and dphi_t1 >= c2*dphi_0 :
            return t1
        iter += 1
    return t1
```

Recherche linéaire 4 : Scipy

Dans la pratique ces méthodes de recherche dépendent beaucoup de la valeur initiale de t_0. Pour assurer le bon comportement des algorithmes, il est préférable d'utiliser la fonction directement implémenter dans *Scipy*. Quelques modifications sont apportées avant l'appel de cette fonction pour respecter les contraintes ajoutées à nos nouveaux algorithmes de minimisation.

Le choix initial de t est très impactant. Il y a deux méthodes [2]:

$$t_0 = 1.0$$

et

$$t_0 = \min\left(1.0\,, 1.01 \cdot \frac{2(f_k - f_{k-1})}{\boldsymbol{p}_k^T \cdot \boldsymbol{\nabla} f_k}\right)$$

Pour contraindre la recherche dans les bornes on peut limiter la valeur t.

1.1.4 Problèmes mathématiques tests 1D

Problème	Bornes	Minimum
Pb n°1 $\min x\sin(x) + \sin(10/3x)$	$x \in [2.7,7.5]$	Golden search : $x_{min} = 5.094$ $f_{min} = -5.683$ Gradient : $x_{min} = 5.094$ $f_{min} = -5.683$
Pb n°2 $\min -x^{\frac{2}{3}} - (1-x^2)^{\frac{1}{3}}$	$x \in [0.01,0.99]$	Golden search : $x_{min} = 0.707$ $f_{min} = -1.587$ Gradient : $x_{min} = 0.707$ $f_{min} = -1.587$
Pb n°3 $\min \sin(1.2x)/(\sin^2(1.5x)+1)$	$x \in [-3.5,2]$	Golden search : $x_{min} = -1.845$ $f_{min} = -0.706$ Gradient : $x_{min} = -1.845$ $f_{min} = -0.706$
Pb n°4 $\min \sin\left(\frac{10}{3}x - \frac{2\pi}{3}\right) - \sin\left(\frac{x^{1.2}}{4} + x\right) \cdot x$ s.t. $c(x) = (4.5-x)\cdot(x-5.8) \geq 0$	$x \in [3,7]$	Golden search : $x_{min} = 5.800$ $f_{min} = -1.999$ $c(x_{min}) = -0.0004$ Gradient : $x_{min} = 5.799$

		$f_{min} = -1.999$ $c(x_{min}) = 0.0$
Pb n°5 $\min \sin\left(\frac{10}{3}x - \frac{2\pi}{3}\right) - \sin\left(\frac{x^{1.2}}{4} + x\right) \cdot x$ s.t. $c(x) = (x - 4.5) \cdot (x - 5.8) \geq 0$	$x \in [3,7]$	Golden search : (minimum local) $x_{min} = 3.999$ $f_{min} = -0.149$ $c(x_{min}) = 0.0$ Gradient : (minimum local) $x_{init} = 5.0$ $x_{min} = 3.999$ $f_{min} = -0.149$ $c(x_{min}) = 0.0$ Gradient : (minimum global) $x_{init} = 6.5$ $x_{min} = 5.809$ $f_{min} = -1.999$ $c(x_{min}) = 0.0$

Les tolérances sont fixées à 10^{-7} et l'algorithme du gradient utilise une recherche linéaire avec critère d'Armijo et interpolation. Les dérivées sont approchées par différences finies d'ordre 1 avec un pas relatif de 10^{-9}. Dans le dernier problème les algorithmes ne parviennent pas à trouver le minimum global de la fonction. Modifier le point initial du gradient permet de sortir de ce minimum local et de se diriger vers le minimum global.

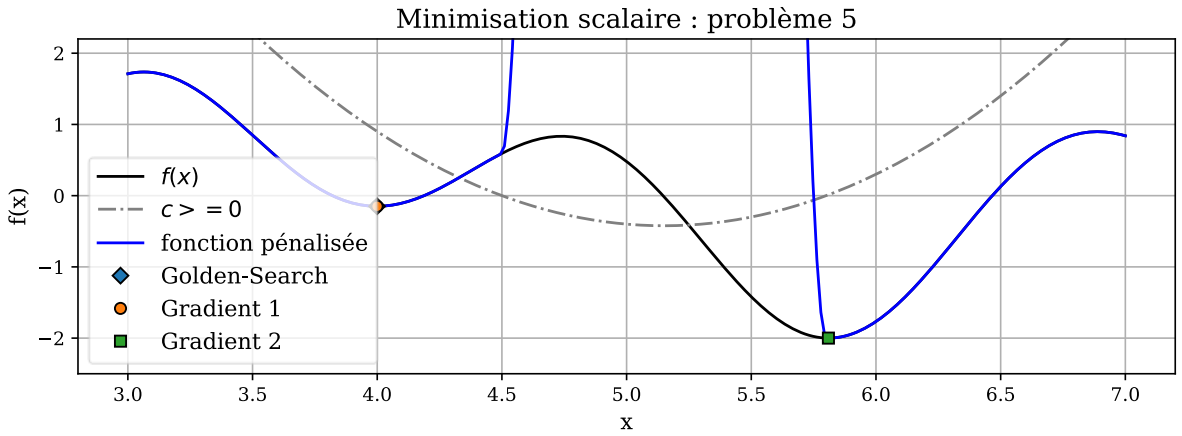


Figure 2 : Minimisation scalaire avec contrainte : pb n°5

1.1.5 Ajustement paramétrique d'une simulation volume finis 1D

Un système de stockage thermique par MCP (matériau à changement de phase) utilise la chaleur latente solide-liquide pour maintenir un corps, ici un fluide en circulation, à une température pseudo-constante. Les MCP sont capables de stocker une très grande quantité d'énergie thermique dans un volume restreint à une température voisine du changement d'état. Ici un système stockeur-échangeur vise à maintenir une eau de chauffage à une température pseudo-constante lors des phases de faible consommation afin de découpler le réseau de chaleur local du réseau urbain. Pour estimer la température de sortie de fluide, on utilise une modélisation 1D le long du tube de l'échangeur. La modélisation dépend de plusieurs paramètres ; une résistance thermique conductive entre le MCP et la paroi du tube R_{cond}, et le coefficient de transfert convectif de l'eau avec la paroi h_{conv}.

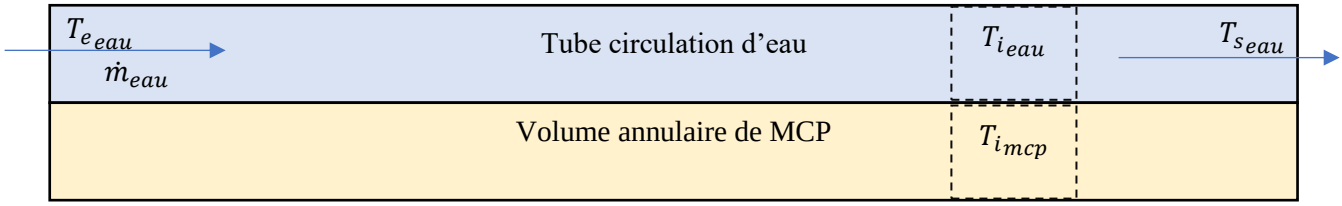


Figure 3 : Schéma du stockeur-échangeur à chaleur latente

Le flux de chaleur échangé entre deux volumes $\Delta\Omega_{eau}$ et $\Delta\Omega_{mcp}$ au travers de la surface δS_i s'exprime par :

$$\Phi_{i_{eau\rightarrow mcp}} = \frac{1}{R_{cond} + \frac{1}{h_{conv}\delta S_i}} \times \left(T_{i_{eau}} - T_{i_{mcp}}\right)$$

Une modélisation 1D dynamique non linéaire du système est implémentée sur Python. Cette simulation est très simple et tourne en quelques fractions de seconde. D'autre part, une simulation en 2D axisymétrique de performance en déstockage complet du système est réalisée. Cette simulation *CFD* prend en compte l'écoulement, un modèle de turbulence ainsi que l'équation de l'énergie avec un couplage solide et fluide. Cette simulation éléments finis demande plusieurs heures de calcul. La température de l'eau en entrée est constante et le système fonctionne jusqu'à l'équilibre thermique. On mesure avec un échantillonnage de 10s la température de l'eau en sortie du système. Ces résultats peuvent être comparés à la simulation 1D pour une valeur du paramètre h_{conv} et on peut calculer l'erreur quadratique moyenne du modèle. La référence est la simulation éléments finis en 2D.

$$\min\left\{ err(h_{conv}) = \frac{1}{N_{pts}} \sum_i \left(T_{ref}(t_i) - T_{sim1D}(t_i)\right)^2 \right\}$$

On souhaite minimiser la fonction $err(h_{conv})$ pour déterminer la meilleure valeur de h_{conv} pour modéliser l'échangeur. A noter que la valeur « réelle » du coefficient d'échange thermique n'est pas forcément celle donnée par la résolution de ce problème. La valeur obtenue par résolution du problème d'optimisation sera la meilleure valeur vis-à-vis de la comparaison des deux modèles.

Ce problème impose plusieurs restrictions :

- La discrétisation ne pourra pas évoluer ;
- La valeur prédite n'est valable que pour un seul débit d'entrée. Il faut répéter l'opération pour différents débits ;

- La fonction $err(h_{conv})$ n'est pas dérivable, mais peut être différenciée avec précaution. Le pas de différentiation doit rester assez grand pour obtenir une variation lissée. La méthode de différenciation complexe ne peut pas être utilisée. Pour les mêmes raison les critères de convergence sur la valeur du gradient doivent être ignorés ;
- La simulation demande plusieurs appels de routines, il est donc utile d'introduire une pré-routine dans les algorithmes de minimisation ;
- Une méthode brute sera coûteuse en temps de calcul ;

Données initiales du problème :

Débit massique d'eau	$0.0204\ kg/s$
Masse de MCP	$36.90\ kg$
Volume d'eau dans l'échangeur	$0.0353\ L$
Température d'entrée de l'eau	$305.00\ K$
Longueur d'échange	$2.9\ m$
Surface d'échange	$0.2278\ m^2$
Nombre d'éléments discrets	15
Température initiale du système	$373.15\ K$
Durée simulation	$20000\ s$

La valeur du coefficient d'échange est bornée entre $50\ W.m^{-2}$ et $300\ W.m^{-2}$. Les tolérances sont assez grandes car la précision du coefficient n'est pas exigée.

	Gradient	*Golden search*
Tolérance	0.005	0.005
Limite itération	50	50
Solution h_{conv}	$155.83\ W.m^2$	$155.89\ W.m^2$
Valeur de l'erreur	$0.9534\ K^2$	$0.9534\ K^2$
Résidu	$3.45 \cdot 10^{-3}$	$3.11 \cdot 10^{-3}$
Nombre d'itération	3	12
Evaluation de la fonction	11	15

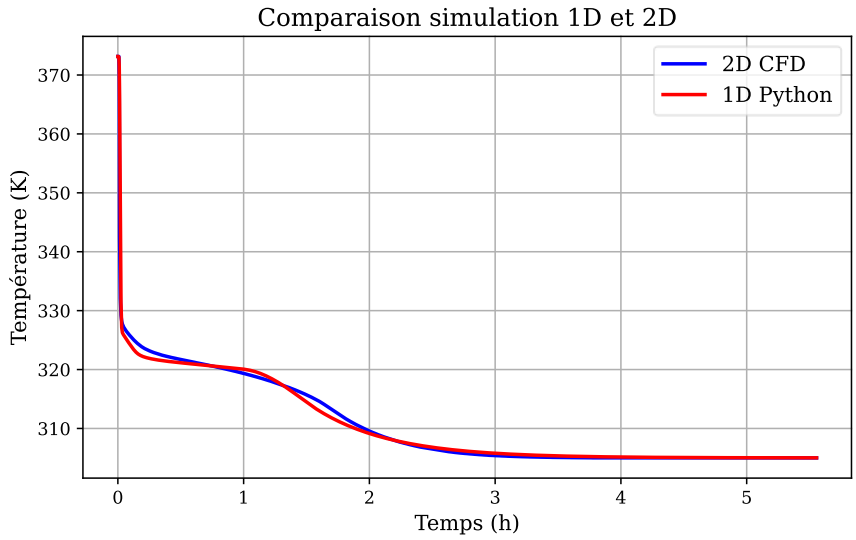


Figure 4 : comparaison simulation 1D et 2D du stockeur-échangeur

2 Optimisation vectorielle

2.1 Le problème test : la fonction de Rosenbrock

La fonction de Rosenbrock est un problème de minimisation utilisée pour confronter les algorithmes d'optimisation. Cette fonction présente une vallée en forme de banane. Les algorithmes analytiques doivent être capables de suivre cette vallée pour trouver le minimum de la fonction.

$$R(x,y) = (x-1)^2 + p(x^2-y)^2$$

Le paramètre p est fixé ici à 10 mais usuellement la valeur 100 permet d'augmenter la difficulté à suivre la vallée.

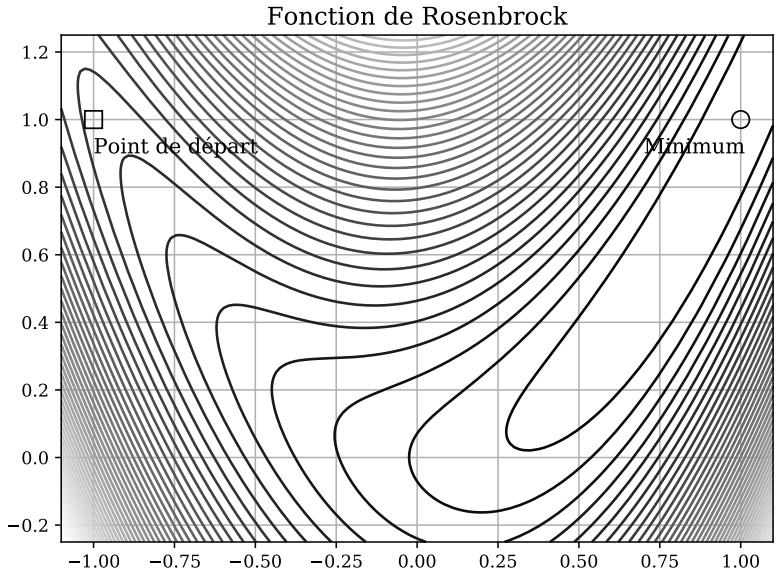


Figure 5 : Isovaleurs de la fonction de Rosenbrock

2.2 Le gradient non linéaire

Cette méthode vectorielle utilise la valeur du gradient comme direction de recherche. On comprend le principal désavantage de cette méthode dans l'exemple de la fonction de Rosenbrock. Les gradients sont orthogonaux aux isovaleurs et on voit que la zone ou cette direction est correctement orientée pour suivre la vallée est très restreinte. Le gradient a du mal à capturer la trajectoire optimale pour arriver au minimum de la fonction et la solution comporte des oscillations.

La recherche linéaire doit être assurée par une méthode de Wolfe, ici la routine de *scipy*.

```
from scipy.optimize import line_search
def minimize_gradient(f,gradf,x0):
    x0 = xk
    while condition_arret :
        fk = f(xk)
        gfk = gradf(xk)
        pk = -gfk
        tk = line_search(f,gradf,xk,fk,gfk,c1=1e-4,c2=0.4)
        xknew = xk + tk*pk
        xk = xknew
```

Les constantes $c1 = 0.0001$ et $c2 = 0.4$ sont utilisées pour la recherche linéaire et le critère de Wolfe.

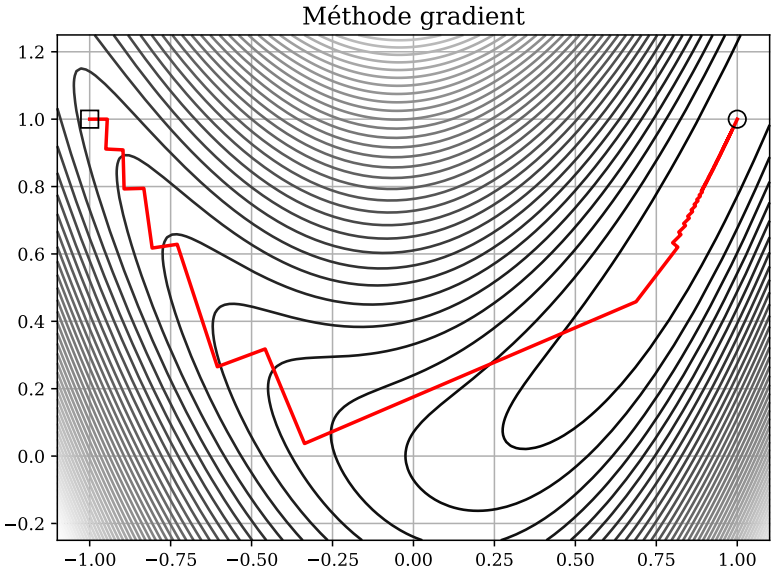


Figure 6 : Rosenbrock algorithme gradient avec pas de Wolfe

L'algorithme converge en 1068 itérations et 8162 évaluations de la fonction (approximations du gradient comprises). Les mauvaises performances de l'algorithme peuvent s'expliquer par le manque d'efficacité dans le choix de la direction de recherche. Cette méthode est souvent restreinte au cadre académique. Le gradient conjugué non linéaire permet de corriger ce défaut.

2.3 Le gradient conjugué non-linéaire

Cet algorithme proche du gradient, utilise une direction de descente qui garde en mémoire l'itération précédente. La première itération utilise la direction opposée au gradient comme pour la version non conjuguée. Ensuite, la direction opposée au gradient est modifiée :

$$\boldsymbol{p}_k = -\boldsymbol{\nabla} f_k + \beta_k \cdot \boldsymbol{p}_{k-1}$$

Le scalaire β_k peut être calculé par plusieurs méthodes, les plus communes sont Fletcher-Reeves, Polack-Ribière et Hesteness-Stiefel. Celle utilisée par *scipy* et par notre algorithme est la méthode de Polack-Ribière contrainte à des valeurs positives. Cette variante assure un redémarrage de la mémoire des directions de recherche.

$$\beta_k = \max\left(0\,, \frac{\boldsymbol{\nabla} f_k^T(\boldsymbol{\nabla} f_k - \boldsymbol{\nabla} f_{k-1})}{\boldsymbol{\nabla} f_k^T \boldsymbol{\nabla} f_k}\right)$$

La recherche linéaire doit être assurée par une méthode de Wolfe, avec les constantes $c1 = 0.0001$ et $c2 = 0.4$.

```
from scipy.optimize import line_search
def minimize_gradient(f,gradf,x0):
    x0 = xk
    fk = f(xk)
    gfk = gradf(xk)
    pk = -gfk

    while condition_arret :
        tk = line_search(f,gradf,xk,fk,gfk,c1=1e-4,c2=0.4)
```

```
        xknew = xk + tk*pk

        xk = xknew
        gfk_old = gfk
        fk = f(xk)
        gfk = gradf(xk)
        yk = gfk-gfk_old
        beta_k = max(0.0, np.dot(gfk,yk) / np.dot(gfk_old,gfk_old) )
        pk = -gfk + beta_k*pk
```

Les performances de la résolution sont améliorées avec une convergence assurée en 13 itérations et 171 évaluations de la fonction.

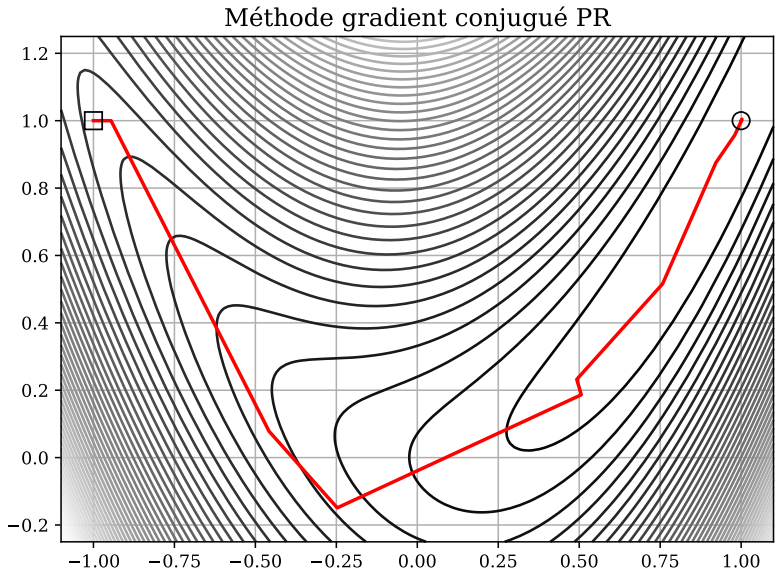


Figure 7: Rosenbrock algorithme gradient conjugué de Polack-Ribière à pas de Wolfe

2.4 Les méthodes quasi-Newton et BFGS

La méthode de Newton est un algorithme qui permet de résoudre un système d'équations non-linéaires de plusieurs variables. Dans le cas d'une minimisation on cherche à annuler le vecteur gradient de la fonction. Les itérations de la résolution par l'algorithme de Newton s'écrivent :

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{\mathcal{H}}(f_k)^{-1} \cdot \boldsymbol{\nabla} f_k$$

Où $\boldsymbol{\mathcal{H}}(f_k)$ est la matrice hessienne de la fonction f. Cet algorithme possède une convergence super-linéaire mais a quelques désavantages :

- L'algorithme converge vers un point de gradient nul pas forcément un minimum ou un maximum ;
- Nécessite la connaissance exacte ou approchée de la hessienne ;
- Nécessite le calcul de l'inverse de la hessienne. Des versions (Newton-CG) utilisent un algorithme itératif de résolution de système linéaire et l'applique partiellement (souvent un gradient conjugué linéaire) puis utilisent une recherche linéaire de Wolfe ;

Une autre idée est d'approcher la hessienne $\mathcal{H}(f_k)$ par une matrice $\boldsymbol{B}_k$ avec certaines propriétés comme ; inversion rapide et simple, stockage mémoire faible ou encore produit matrice-vecteur rapide. Le calcul de la matrice hessienne, ou directement de son inverse, donne une nouvelle direction de descente pour une recherche linéaire. On parle de méthode quasi-Newton. La recherche linéaire doit être assurée par une méthode de Wolfe, avec les constantes $c1 = 0.0001$ et $c2 = 0.9$.

$$p_{k+1} = -\boldsymbol{B}_k \cdot \boldsymbol{\nabla} f_k$$

La matrice à l'itération $k+1$ est calculé à partir de la matrice précédente. La première itération utilise la matrice identité comme approximation de la hessienne $\boldsymbol{B}_0$, cela revient à utiliser la direction opposée au gradient. La méthode BFGS, pour Broyden, Fletcher, Goldfarb et Shanno, est une possibilité pour calculer l'approximation de la hessienne ou de son inverse. Il existe d'autres méthodes quasi-Newton : SR1, DFP, Broyden, mais BFGS est communément désignée comme étant la plus performante.

On pose :

$\boldsymbol{y}_k = \boldsymbol{\nabla} f_{k+1} - \boldsymbol{\nabla} f_k$; $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$; $\rho_k = \boldsymbol{y}_k^T \cdot \boldsymbol{s}_k$

Approximation de la hessienne $\boldsymbol{B}_{k+1}$

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \left(\boldsymbol{y}_k \boldsymbol{y}_k^T\right)\rho_k^{-1} - \left(\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k\right)/\left(\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k\right)$$

Approximation de la hessienne inverse $\boldsymbol{B}_{k+1}^{-1} = \boldsymbol{H}_{k+1}$

$$\boldsymbol{H}_{k+1} = \boldsymbol{B}_{k+1}^{-1} = \left(\boldsymbol{I} - \rho_k^{-1} \boldsymbol{s}_k \boldsymbol{y}_k^T\right) \cdot \boldsymbol{H}_k \cdot \left(\boldsymbol{I} - \rho_k^{-1} \boldsymbol{y}_k \boldsymbol{s}_k^T\right) + \rho_k^{-1} \boldsymbol{s}_k \boldsymbol{s}_k^T$$

L'algorithme : pour pouvoir ajuster les dimensions des vecteurs $\boldsymbol{s}_k$ et $\boldsymbol{y}_k$ il faut utiliser la fonction de *numpy.newaxis*.

```
from scipy.optimize import line_search
def minimize_BFGS(f,gradf,x0):
    dim = len(x0)
    x0 = xk
    fk = f(xk)
    gfk = gradf(xk)
    pk = -gfk
    I = np.identity(dim)
    Hk = np.identity(dim)

    while condition_arret :
        tk = line_search(f,gradf,xk,fk,gfk,c1=1e-4,c2=0.9)
        xk = xk + tk*pk
        gfk_old = gfk
        gfk = gf(xk)
        fk = f(xk)

        sk = (tk*pk)[np.newaxis].T
        yk = (gfk-gfk_old)[np.newaxis].T

        A1 = np.dot(sk,yk.T)/rhok
        A2 = np.dot(yk,sk.T)/rhok
        A3 = np.dot(sk,sk.T)/rhok
        Hk = np.dot((I-A1),np.dot(Hk,(I-A2))) + A3
        pk = -Hk.dot(gfk)
```

Des modifications sont possibles pour remplacer les matrices *numpy.array* en matrices creuses *scipy.sparse* pour gagner de la mémoire si les dimensions du problème sont grandes. Aussi, il existe des versions pour l'optimisation de la mémoire et des calculs vectoriels : L-BFGS et L-BFGS-B.

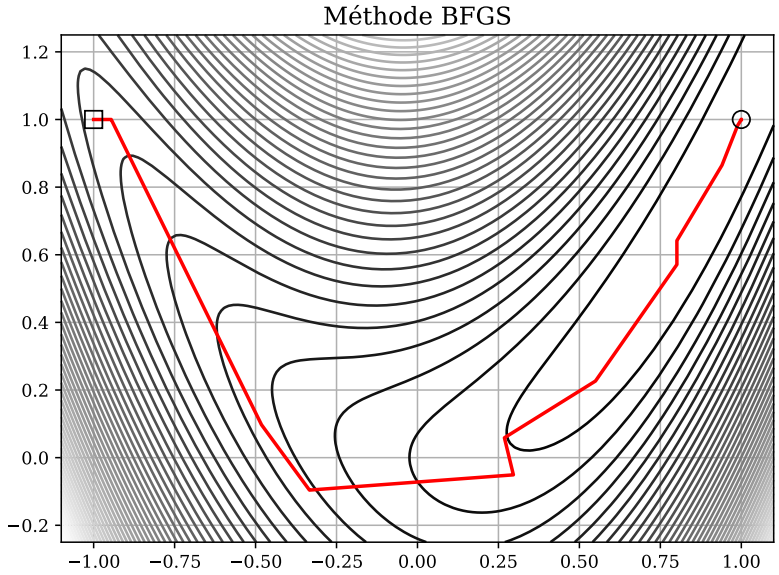


Figure 8 : Rosenbrock algorithme quasi Newton, BFGS, à pas de Wolfe

Les performances de la résolution sont similaires à celles du gradient conjugué avec 15 itérations et 194 évaluations de la fonction. En dimension supérieure et pour des problèmes plus complexes, BFGS devient en général meilleur que le gradient conjugué.

2.5 Recherche directionnelle et méthode de Powell

Cette famille de méthode utilise une base de vecteurs unitaires de recherche pour résoudre un problème d'optimisation sans utiliser la dérivabilité et la valeur du gradient. La base de recherche est souvent le repère orthonormé de l'espace des variables. A chaque itération on fait une boucle de recherches linéaires à partir du point courant suivant les directions de la base vectorielle. Garder cette base pendant toutes les itérations engendre un comportement oscillant comme pour l'exemple du gradient. La méthode de Powell remplace le vecteur ayant la plus grande contribution au déplacement par la direction globale de déplacement. Cette méthode a l'avantage d'être très simple à implémenter et n'utilise pas le gradient.

Pour chaque direction explorée, il est possible d'utiliser des méthodes de minimisation scalaire sans gradient comme l'algorithme de Brent ou la méthode du nombre d'or. Pour des raison de simplicité, il est préférable d'utiliser la fonction *minimize_scalar* de la bibliothèque *scipy.optimize*. Afin de garantir le cahier des charges imposé, il est possible de bornée la recherche linéaire et d'introduire une pénalisation pour les contraintes. L'algorithme de *scipy* ajoute une étape avant de réitérer la boucle *while* pour augmenter la rapidité de convergence. Un sous-problème est résolue sur une approximation de la fonction.

```
from scipy.optimize import minimize_scalar
def Powell(f,x0) :
    dim = len(x0)
    vecBase = np.identity(dim)
    xk = x0
    while condition_arret :
        #point intermédiaire
```

```
        zj = xk
        diff_index = 0
        diff = 0.0
        for pj in vecBase :
            fj0 = fzj
            #fonction scalaire phi
            def phi(t):
                return f(zj+t*pj)
            #minimisation le long de pj
            tj = minimize_scalar(f(zj+t*pj))
            #nouveau point intermédiaire
            zj = zj + tj*pj
            fzj = f(zj)
            #Calcul de la plus grande différence
            if (fj0-fj) > diff :
                diff = (fj0-fj)
                diff_index = j
            fj0 = fj

        #mise à jour
        xk_old = xk
        xk = zj
        dxnorm = norm(xk-xk_old)
        #La direction avec la plus long distance est remplacé par
        #la direction xk_old --> xk
        vecBase[diff_index] = vecBase[-1]
        vecBase[-1] = xk-xk_old
```

Pour le problème de minimisation de la fonction de Rosenbrock, la méthode de Powell converge en 15 itérations et 433 appels de la fonction. La convergence est plus longue que pour les méthodes analytiques différentielles, mais lorsque le gradient est difficilement approximable cet algorithme donne de meilleurs résultats.

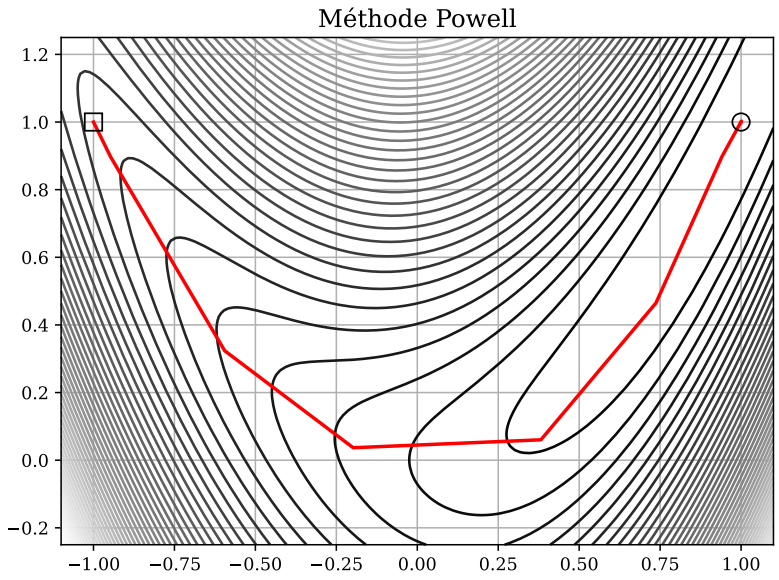


Figure 9 : Rosenbrock, méthode de Powell

2.6 Méthode du simplex de Nelder Mead

La méthode de Nelder Mead est une méthode dite directe. Apprécié pour sa robustesse, cet algorithme est léger car peu de stockage mémoire et tolérant aux bruits dans la fonction objective. Cette dernière n'a pas besoin d'être calculée de manière exacte et peut être approchée. La méthode démarre en formant un polytope (ou simplexe) autour d'un point initial. La comparaison des $n+1$ valeurs du simplexe permet de modifier sa géométrie par réflexions, expansion, contraction et homothétie.

Etapes de transformation du simplexe :

1- Trier les valeurs telles que : $f(x_1) \leq \cdots \leq f(x_{n+1})$
2- Calculer le barycentre dernier point exclu : $\bar{x} = \sum_{i=1}^{n} x_i$
3- Calculer la réflexions $x_r = (1+\rho)\bar{x} + \rho x_{n+1}$
4- Si $f(x_r) < f(x_1)$ alors on essaie d'étendre la réflexion par une expansion :
   a. $x_e = (1+\rho\chi)\bar{x} + \rho\chi x_{n+1}$
   b. Si $f(x_e) < f(x_r)$ alors $x_{n+1} = x_e$
   c. Sinon $x_{n+1} = x_r$
   d. Retour à l'étape 1
5- Si $f(x_r) < f(x_n)$ c'est-à-dire que x_r est un candidat moyen
   a. $x_{n+1} = x_r$
   b. Retour à l'étape 1
6- Si $f(x_r) < f(x_{n+1})$ on peut tenter une contraction externe pour améliorer le candidat
   a. $x_c = (1+\psi\rho)\bar{x} + \psi\rho x_{n+1}$
   b. Si $f(x_c) < f(x_{n+1})$ on remplace $x_{n+1} = x_c$ et retour à l'étape 1
   c. Sinon étape 8
7- Si $f(x_r) \geq f(x_{n+1})$ on peut tenter une contraction interne pour améliorer le candidat
   a. $x_c = (1-\psi)\bar{x} + \psi x_{n+1}$
   b. Si $f(x_c) < f(x_{n+1})$ on remplace $x_{n+1} = x_c$ et retour à l'étape 1
   c. Sinon étape 8
8- Homothétie $x_j = x_1 + \sigma(x_j - x_1)$ pour $j \in [2, n+1]$ et retour à l'étape 1

Cet algorithme est proposé par [2] et *scipy*. Il est utile et facile de contraindre les déplacements dans une région comprise entre deux bornes. Les valeurs usuelles des paramètres sont :

$$\rho = 1\,;\chi = 2\,;\psi = 0.5;\sigma = 0.5$$

```
def NelderMead(f,x0) :
    rho = 1 ;chi = 2; psi = 0.5; sigma = 0.5
    dim = len(x0)
    simplex,fsimplex = inisialiser_simplex(x0,f)
    while condition_arret :
        index = numpy.argsort(fsimplex)
        simplex,fsimplex = simplex[index],fsimplex[index]
        xbar = numpy.sum(simplex[:-1],axis=0)/dim
        #reflexion
        xr = (1 + rho) * xbar - rho * simplex[-1]
        fr = f(xr)
        if fr < fsimplex[0]:
            #expansion
            xe = (1 + rho * chi) * xbar - rho * chi * simplex[-1]
            fe = f(xe)
            if fe < fr:
                simplex[-1] = xe
```

```
                fsimplex[-1] = fe
            else:
                simplex[-1] = xr
                fsimplex[-1] = fr
        else : #fr>=f0
            homothetie = False
            if fr < fsimplex[-2]:
                simplex[-1] = xr
                fsimplex[-1] = fr
            else:  # fr >= fn
                if fr < fsimplex[-1] :
                    # Contraction externe
                    xc = (1 + psi * rho) * xbar - psi * rho * simplex[-1]
                    fc = f(xc)
                    if fc < fsimplex[-1]:
                        simplex[-1] = xc
                        fsimplex[-1] = fc
                    else :
                        homothetie = True
                else:
                    # Contraction interne
                    xcc = (1 - psi) * xbar + psi * simplex[-1]
                    fcc = f(xcc)
                    if fcc < fsimplex[-1]:
                        simplex[-1] = xcc
                        fsimplex[-1] = fcc
                    else:
                        homothetie = True
            #homothetie
            if homothetie :
                x1 = simplex[0]
                for j,xjp1 in enumerate(simplex[1:]):
                    simplex[j+1] = x1 + sigma * (xjp1 - x1)
                    fsimplex[j+1] = f(simplex[j+1])
```

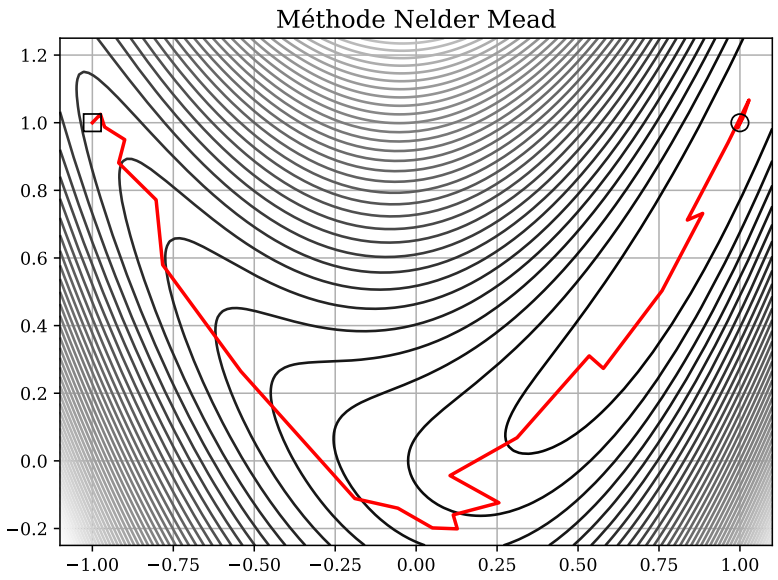


Figure 10 : Rosenbrock, méthode de Nelder Mead

Sur l'exemple de la fonction de Rosenbrock la méthode de Nelder Mead atteint le minimum en 37 itérations et 69 évaluations de la fonction, ce qui est le meilleur résultat.

Les tests de convergence sont spécifiques à cette méthode :

$$\max\left(\|x_j - x_1\|_{j\epsilon[2,n+1]}\right) \leq tol$$

$$\max\left(|f_j - f_1|_{j\epsilon[2,n+1]}\right) \leq ftol$$

2.7 Problèmes mathématiques tests

Problème	Bornes / initialisation	Minimum
Pb n°1 $\min(x^2 + y^2 - r_0)^2 - x$	$x, y \in [-2r_0; 2r_0]$ $x_0 = \{0.5, 0.02\}$	CG / BFGS / Powell / NM : $x_{min} = \{1.107, 0\}$ $f_{min} = -1.056$
Pb n°2 $\min rosenbrock\,(\boldsymbol{x})$ Sujet à : $-1.2x_1 + x_2 + 0.3 \geq 0$ $2 - x_1 - x_2 \geq 0$ $0.4x_1 + x_2 \geq 0$	$x_{min} = \{-1.1, -0.25\}$ $x_{max} = \{1.1, 1.25\}$ $x_0 = \{1, -1\}$	CG / BFGS / Powell / NM : $x_{min} = \{1, 1\}$ $f_{min} = 0$
Pb n°3 $\min rosenbrock\,(\boldsymbol{x})$ Sujet à : $x_1^2 - x_2 = 0$	$x_{min} = \{-1.1, -0.25\}$ $x_{max} = \{1.1, 1.25\}$ $x_0 = \{1, -1\}$	CG / BFGS / Powell / NM : $x_{min} = \{1, 1\}$ $f_{min} = 0$
Pb n°4 : grandes dimensions $\min f(\boldsymbol{x}) = rosenbrock(\boldsymbol{x})$ Taille du problème $n = 100$	$x_{min} = \{-2, \dots, -2\}$ $x_{max} = \{2, \dots, 2\}$ $x_0 = \{-1, \dots, -1\}$	CG / BFGS : $x_{min} = \{1, \dots, 1\}$ $f_{min} = 0$

Il est intéressant de voir que sur le problème d'optimisation avec contraintes d'inégalité, les méthodes de Powell et surtout Nelder-Mead dépasse la zone de faisabilité. Sur ce type de contrainte il serait utile de définir une pénalisation interne pour éviter ce phénomène.

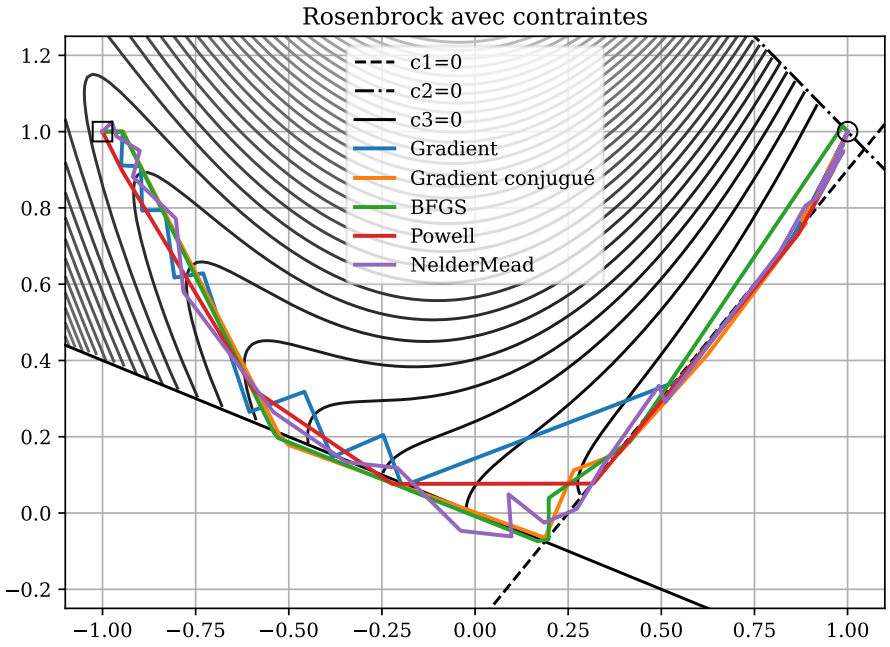


Figure 11 : Minimisation contrainte de la fonction de Rosenbrock

2.8 Problème d'ingénierie

2.8.1 Optimisation technico-économique d'une pompe à chaleur

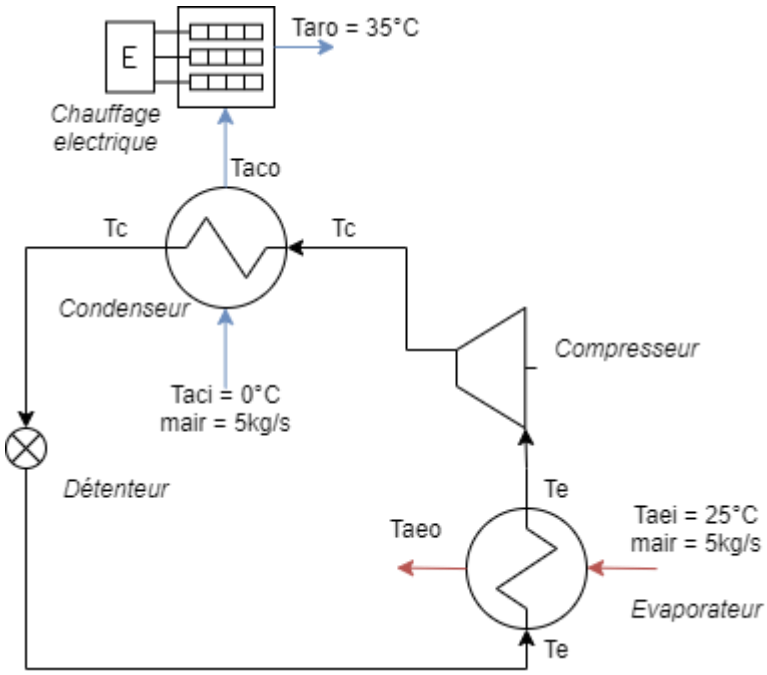


Figure 12 : Schéma d'une pompe à chaleur

L'objectif de cette étude est d'optimiser le coût d'une pompe à chaleur. Le cas de fonctionnement étudié demande de chauffer un débit d'air m_{air} à la température T_{aro}. Ce flux d'air rentre à $T_{aci} = 0°C$ et ressort de la PAC à la température T_{aco}. L'air termine d'être chauffé par une résistance de puissance $\dot{Q}_{elec}$. Ensuite l'air revient dans la PAC à la température T_{aei} et ressort à l'extérieur à la température T_{aeo}. Les puissances du condenseur, évaporateur et compresseur sont $\dot{Q}_{cond}$, $\dot{Q}_{eva}$ et $\dot{W}_{comp}$, et les températures du fluide caloporteur sont notées T_c et T_e.

Dans la simulation, un équilibre thermodynamique est appliqué avant un bilan économique. Les valeurs fournis sont indicatives. [3]

Bilan thermodynamique :

$COP = COP_{ideal} \times \eta_{cop} = \frac{Tc}{Tc-Te}\eta_{cop}$;

$\dot{Q}_{eva} = m_{air} cp_{air}(T_{aco} - T_{aci})$;

$\dot{W}_{comp} = \dot{Q}_{eva}/COP$;

$\dot{Q}_{cond} = \dot{Q}_{eva} + \dot{W}_{comp}$;

$T_{aco} = \frac{\dot{Q}_{cond}}{cp_{air} m_{air}} + T_{aci}$;

$\dot{Q}_{elec} = m_{air} cp_{air}(T_{aro} - T_{aco})$;

$\Delta T_c = \frac{(\mathrm{Tc-Taci})-(\mathrm{Tc-Taco})}{\ln\left(\frac{\mathrm{Tc-Taci}}{\mathrm{Tc-Taco}}\right)}$;

$\Delta T_e = \frac{(\mathrm{Taei-Te})-(\mathrm{Taeo-Te})}{\ln\left(\frac{\mathrm{Taei-Te}}{\mathrm{Taeo-Te}}\right)}$;

$A_{cond} = \frac{\dot{Q}_{cond}}{U\Delta T_c}$;

$A_{eva} = \frac{\dot{Q}_{eva}}{U\Delta T_e}$;

Données

Performances de la PAC :
$\eta_{cop} = 0.31$ efficacité réelle PAC ;

Flux d'air :
$cp_{air} = 2000\ \mathrm{J/(K.kg)}$;
$m_{air} = 5\ \mathrm{kg/s}$;

Températures :
$T_{aci} = 273\ \mathrm{K}$;
$T_{aro} = 308\ \mathrm{K}$;
$T_{aei} = 298\ \mathrm{K}$;

Performances échangeurs :
$U = 25\ \mathrm{W/(m^2K)}$ coefficient d'échange ;

Bilan économique :

$AC_{elec} = c_{elec}(\dot{W}_{comp} + \dot{Q}_{elec})Nh/1000$;
$IC_{eva} = c_{ech} * A_{eva}$;
$IC_{cond} = c_{ech} * A_{cond}$;
$IC_{comp} = c_{comp} * \dot{W}_{comp}/1000$;
$IC_{res} = c_{res} * \dot{Q}_{elec}/1000$;

$I_0 = (IC_{eva} + IC_{cond} + IC_{comp} + IC_{res})(1 + c_{inst})$;
$I_n = 0 \; \forall \; n > 1$;

$M_n = I_0 \times c_{maintenance}$;
$R_n = AC_{elec}$;

$$C_{tot} = \sum_{n=0}^{Na} \frac{I_n + R_n + M_n}{(1+r)^{n+1}}$$

Données
Coûts des équipements :
$c_{elec} =$ 0.06 \$/kWh ;
$c_{ech} = 100$ \$/m^2 ;
$c_{comp} = 200$ \$/kW ;
$c_{res} = 80$ \$/kW ;

Installation et maintenance :
$c_{inst} = 0.2$;
$c_{maintenance} = 0.1$;

Scénario de vie :
$Nh = 4000$ h utilisation sur une année ;
$Na = 10\ ans$ durée de vie ;
$r = 0.1$ taux actualisation et intérêts ;

Optimisation :

Minimiser :

$$C_{tot}(T_{aeo}, T_c, T_e)$$

Sujet à :

$$c1 = T_{aeo} - T_e \geq 4 \text{ K}$$
$$c2 = T_{aei} - T_e \geq 4 \text{ K}$$
$$c3 = T_c - T_{aci} \geq 4 \text{ K}$$
$$c4 = T_c - T_{aco} \geq 4 \text{ K}$$
$$c5 = COP \geq 1$$
$$c6 = \dot{Q}_{elec} \geq 0\ W$$

Pour :

$T_{aeo} \in$ [273 ; 295]K , $T_e \in$ [243 ; 271]K et $T_c \in$ [303 ; 333]K

La fonction coût n'est pas dérivable et une approximation du gradient ne donnerait pas des résultats satisfaisants. Il est préférable de se tourner vers une méthode directe comme Powell ou Nelder-Mead. Il reste une dernière difficulté : initialiser un point de départ de la recherche. La température T_{aeo} conditionne la puissance de la PAC, c'est-à-dire la puissance du compresseur et de la résistance électrique. Plus cette température est élevée moins la puissance récupérée est grande et plus les composants seront coûteux. Les températures T_c et T_e doivent être très différentes des températures de l'air pour améliorer l'échange thermique. Plus T_c est grand, meilleur est l'échange Plus T_e est petit, meilleur est l'échange. Cependant, ces deux températures conditionnent aussi le COP de la PAC. Si la différence de T_c et T_e est trop grande, le COP sera faible et entrainera un surcoût de puissance sur le compresseur. On choisit de partir du point $T_{aeo} =$ 273 K, $T_c = 333$ K et $T_e = 243$ K. D'autres points initiaux ont été testés pour assurer le résultat de l'optimisation. Cette optimisation est confrontée aux méthodes globales stochastiques qui confirment les résultats.

Powell	**Nelder-Mead**	**Algorithmes stochastiques**
$T_{aeo} = 277\ K$; $T_e = 254.50\ K$; $T_c = 333\ K$; $C_{tot} = 197727\ \$$; Itération : 25 Appels fonction : 2777 Résidu x : $4.9E-10$ Résidu f : $2.44E-10$ Violation des contraintes 0	$T_{aeo} = 277\ K$; $T_e = 256.15\ K$; $T_c = 333\ K$; $C_{tot} = 197661\ \$$; Itération : 37 Appels fonction : 69 Résidu x : 0.0925 Résidu f : $5.35E-10$ Violation des contraintes 0	En moyenne pour 10000 itérations : $T_{aeo} = 277\ K$; $T_e = 256\ K$; $T_c = 333\ K$; $C_{tot} = 197680\ \$$; Violation des contraintes 0

2.8.2 Optimisation technico-économique d'un réseau de distribution d'eau ramifié

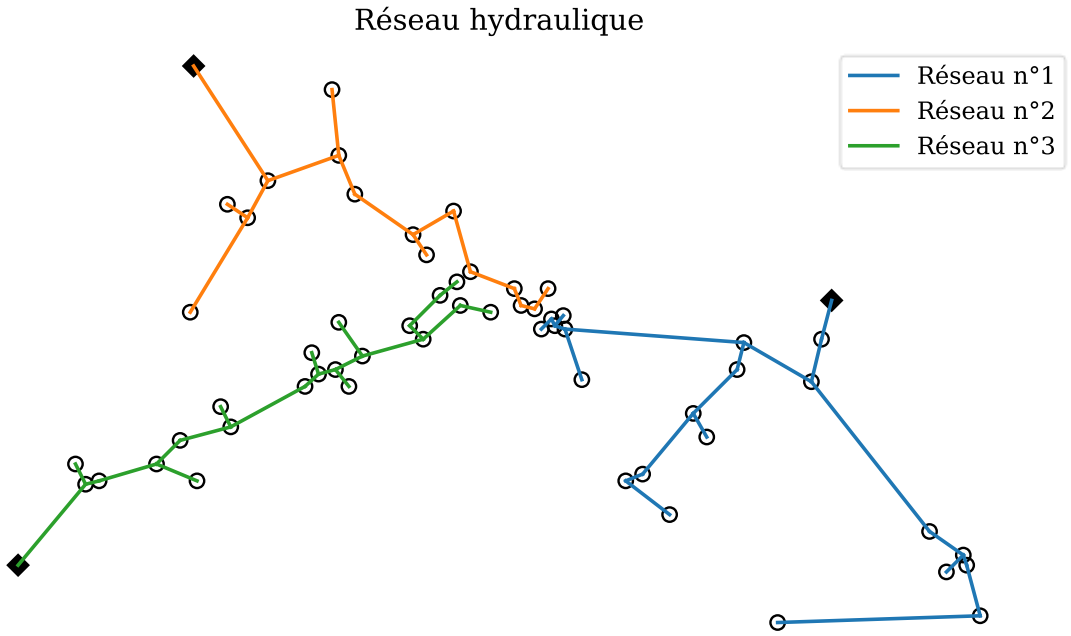


Figure 13 : Topologie d'un réseau hydraulique

On souhaite optimiser la re-conception de ce réseau hydraulique ramifié en diminuant les pertes charges et le coût d'investissement dans la rénovation des canalisations. Les variables d'optimisation seront les diamètres des canalisations.

Modélisation hydraulique

Un réseau hydraulique est modélisé comme un ensemble de branches reliant des nœuds. Les grandeurs hydrauliques principales sont les débits dans chacune des branches et les pressions à chaque nœud. Un réseau ramifié ne possède pas de boucle interne et les débits sont connus dans les branches du réseau. Dans le cas étudié il y a trois réseaux principaux non reliés entre eux. Dans cette configuration la perte charge sur une branche ij est calculée par la relation suivante :

$$p_j - p_i = \delta p_{ij} = \frac{1}{2}\rho\left(\frac{f_{ij}L_{ij}}{D_{ij}} + \beta_{ij}\right)v_{ij}^2$$

Avec L_{ij} et D_{ij} la longueur et le diamètre de la canalisation ij, f_{ij} et β_{ij} les coefficients de pertes charges régulières (frictions) et singulières. On remplace la vitesse v_{ij} par le débit massique $\dot{m}_{ij}$ et on introduit la section $S_{ij} = \frac{\pi}{4} D_{ij}^2$. Ces données du réseau hydraulique et les débits sont fournis.

$$\delta p_{ij} = \frac{1}{2}\left(\frac{f_{ij} L_{ij}}{D_{ij}} + \beta_{ij}\right) / (\rho S_{ij}^2)\ |m_{ij}|\ m_{ij}$$

Il est utile d'exprimer la puissance hydraulique des pertes charges grâce une formulation matricielle où $\boldsymbol{ab}$ est un produit termes à termes (*numpy*) et $\boldsymbol{a} \cdot \boldsymbol{b}$ est le produit matriciel.

$$\dot{W}_{press} = \dot{\boldsymbol{m}}^T \cdot \boldsymbol{\delta p}$$

Ainsi que le gradient par rapport aux diamètres :

$$\frac{\partial \dot{W}_{press}}{\partial D_{ij}} = \dot{\boldsymbol{m}}^T \cdot \frac{\partial \boldsymbol{\delta p}}{\partial D_{ij}}$$

Modélisation économique

Par simplicité on peut écrire l'image de l'investissement à fournir en fonction des diamètres, ainsi que le gradient :

$$I_{eco} = \sum_{ij}^{N} L_{ij} D_{ij}^{\alpha_{eco}}$$

$$\frac{\partial I_{eco}}{\partial D_{ij}} = \alpha_{eco} L_{ij} D_{ij}^{\alpha_{eco}-1}$$

Avec un paramètre économique fixé ici à $\alpha_{eco} = 1.2$.

Bornes du problème

Les vitesses d'écoulement minimum et maximum admissibles permettent de connaître le diamètre minimum D_{min} et maximum D_{max} avec $v_{max} = 2.5\ m/s$ et $v_{min} = 0.1\ m/s$. Ensuite, il est possible de déduire les valeurs maximales et minimales des performances hydrauliques et de l'investissement économique. On peut ainsi construire deux critères normalisés :

$$\Phi_{press} = \frac{\dot{W}_{press} - \dot{W}_{min}}{\dot{W}_{max} - \dot{W}_{min}}$$

$$\Phi_{eco} = \frac{I_{eco} - I_{min}}{I_{max} - I_{min}}$$

Optimisation bi-critère

Sans rentrer dans les détails de la méthode *epsilon-contrainte* pour la recherche de front de Pareto, il est possible d'optimiser les deux critères précédents en résolvant plusieurs problèmes d''optimisation. On sait que Φ_{eco} varie entre deux bornes connues. La minimisation de ce critère dégrade les performances hydrauliques Φ_{press}. Pour un ensemble de contrainte d'investissement à ne pas dépasser C_{eco}, il faut minimiser le critère hydraulique Φ_{press}. Répéter cette minimisation sur l'ensemble de la plage d'investissement permet de construire le front de Pareto qui résume les meilleurs compromis $\Phi_{press}; \Phi_{eco}$. On peut refaire la démarche en inversant critère à minimiser et contrainte à ne pas dépasser.

Minimiser

$$\Phi_{press}(\boldsymbol{D_{ij}})$$

Sujet à :

$$\Phi_{eco}(\boldsymbol{D_{ij}}) \leq C_{eco}$$

$$\boldsymbol{D_{min}} \leq \boldsymbol{D_{ij}} \leq \boldsymbol{D_{max}}$$

Résolution

Le problème est dérivable et les gradients des fonctions sont connus. Le problème possède 57 variables. Les méthodes analytiques avec gradient sont à privilégiées. Ce problème est résolu pour 150 contraintes d'investissement avec les méthodes BFGS, gradient conjugué pénalisés et la routine *minimize* de *scipy* avec l'algorithme de région de confiance contraint et *SLSQP*. Le résultat d'un problème permet d'initialiser le suivant. Les fronts de Pareto sont comparés aux résultats de l'algorithme multicritères NSGA-II pour un nombre d'évaluations de la fonction avoisinant 100000 itérations.

Résultats

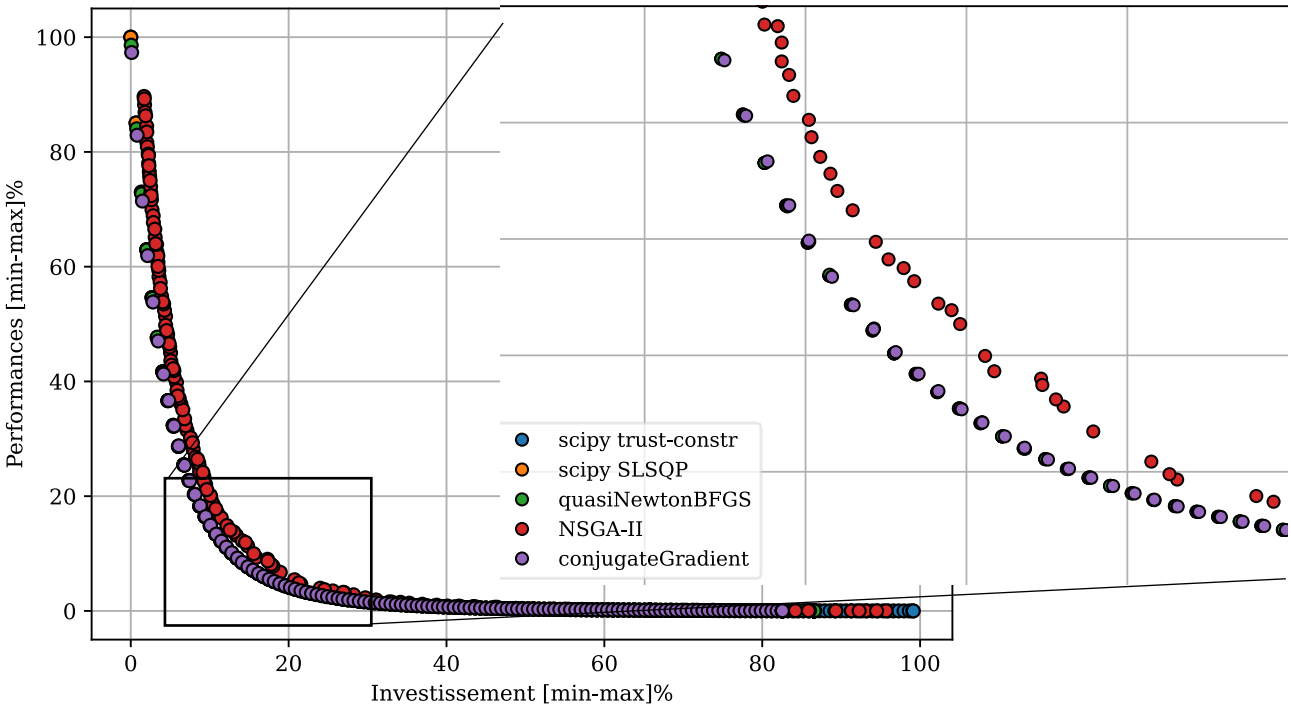


Figure 14 : Optimisation réseau hydraulique ramifié

Algorithme	Appels	Plage optimum énergie	Plage optimum coût	Temps relatif
Scipy trust-constr	6588	0.001 - 99.99 %	0 – 99.13 %	100%
Scipy SLSQP	953	0.14 – 100 %	0 – 65.77 %	3.72%
BFGS	4133	0.03 – 97.32%	0.11 – 83.89 %	2.70%
CG	4139	0.04 – 97.32%	0.11 – 82.55 %	1.79%
NSGA-II	100000	0.01 – 97.32%	1.69 – 95.69 %	21.82%

L'algorithme le plus rapide est l'implémentation de *SLSQP* de *Scipy* avec 953 appels de fonction mais avec une plage d'optimum économique explorée à 65%. En termes de performances, la méthode par région de confiance de *Scipy* possède le meilleur rapport entre nombre d'appels et critère d'optimalité mais dans un temps d'exécution très long. Les méthodes *BFGS* et gradient conjugué *CG* conçues pour nos besoins ont des performances acceptables. Avec une plage d'optimalité supérieure à *SLSQP*, un nombre d'appels inférieur à *trust-constr* et un temps de calcul faible, ils représentent une alternative sérieuse. Enfin, l'algorithme génétique *NSGA-II* fourni un front de Pareto de moins bonne qualité mais permettrait d'initialiser d'autres algorithmes analytiques. Cet exemple ne donne pas une comparaison exhaustive des algorithmes mais démontre le potentiel des méthodes conçues ici. La courbe de Pareto permet de se concentrer sur des zones d'optimisation à fortes plus-value. Dans une démarche ingénieure, la tendance observée est suffisante pour poursuivre l'analyse et réduire les choix de conceptions

Conclusion optimisation locale analytique

Cette partie montre qu'il est possible de reproduire des routines simples de minimisation et de s'approcher des performances de *scipy,* ou d'outils de hautes performances, sur des problèmes complexes avec ou sans contraintes, bornés, dérivables ou non.

Ces algorithmes comportent la possibilité d'appel d'une pré-routine. Cet ajout est utile pour les simulations comportant des calculs intermédiaires. La formulation des problèmes en ingénierie demande souvent un calcul de l'état du « système » avant les coûts objectifs, les contraintes et les dérivées. La maîtrise de la séquence d'évaluation fonction/contraintes/dérivées permet de diminuer l'appel de cette pré-routine avec une mise en mémoire de l'état du système. Si l'on souhaite obtenir toutes les contraintes, la valeur de la fonction et les dérivées pour un point, il n'est pas souhaitable de recalculer l'état du « système ». Soit pour 1 contrainte, 4 évaluations pour un même point $\boldsymbol{x}$, (1 fonctions, 1 contraintes, 2 gradients). Pour 10 contraintes ce chiffre passe à 22 évaluations. On comprend ici la nécessité de l'ajout de la pré-routine.

Scipy reste très supérieur sur la prise en compte des contraintes. Les trois algorithmes *COBYLA*, *SLSQP* et *trust-constr* sont très performants mais utilisent des formulations complexes qui demandent la maitrise des théories de l'optimisation contrainte. Ces approches restent à privilégier le plus souvent. Si *scipy* fournit un éventail d'outils très bons pour l'optimisation locale, les algorithmes globaux sont moins complets. Dans la partie suivante, il sera question des algorithmes stochastiques, qui sont des outils très puissants, pouvant gérer des problèmes réels, entiers ou combinatoires. Très simples à coder, l'ajout de contraintes par pénalisation permet d'obtenir des méthodes avec une plus-value certaine par rapport à *scipy*.

III Algorithmes stochastiques d'optimisation globale

1 Présentation des méthodes stochastiques

1.1 Généralités et intérêts des algorithmes de minimisation globale

Les méthodes d'optimisation stochastiques s'appuient sur des mécanismes de recherche de solution et de transition aléatoires et probabilistes. Ces méthodes s'affranchissent des barrières de potentiel et des minimas locaux. La recherche se concentre sur des zones à fort potentiel d'optimisation et permet de sortir d'optimums locaux en conservant les meilleures solutions enregistrées. Ces méthodes ne nécessitent pas de point de départ ni la connaissance des propriétés analytiques de la fonction objectif. Cependant ils peuvent fournir des résultats différents pour une même configuration. Les méthodes globales demandent aussi un grand nombre d'évaluations de la fonction objectif et plusieurs résolutions pour conforter leurs résultats.

Dans le cas de problèmes très complexes, ces approches globales permettent d'obtenir une solution satisfaisante pouvant être consolidée via une méthode locale. Ces méthodes sont très polyvalentes ; elles peuvent résoudre des problèmes continus, discrets, combinatoires ou même hybrides. Cette polyvalence permet d'aborder beaucoup de problèmes de l'ingénierie technique. En outre, cela peut engendrer un effet boîte noire où l'on s'imagine qu'il suffit de paramétrer la fonction objectif et de lancer l'algorithme sans préparation. Comme les algorithmes analytiques locaux, les recherches globales doivent comporter une étude de la fonction objectif, des contraintes et des variables en jeu. Les paramètres d'optimisation sont souvent assez complexes à estimés et là aussi des réglages ou réajustement sont souvent nécessaires.

Les résultats doivent aussi être regardés avec précaution. La solution doit être homogène sur plusieurs exécutions de l'algorithme. Il est important d'observer la convergence de la méthode afin de régler les paramètres de l'algorithme.

Caractéristiques des méthodes globales :

- Pas besoin de connaître les propriétés de la fonction (continuité, dérivabilité) ;
- Efficace sur des fonctions fortement non linéaires, non-convexes, non-dérivables ;
- Pas d'évaluation de gradient ou de matrice hessienne ;
- Fonctionnent sur des problèmes continus, discrets, combinatoires ou hybrides ;
- Souvent simple à coder.

Mais possèdent certaines faiblesses :

- Pas d'extremum global garanti ;
- Peut se révéler inefficace sur certains problèmes avec un très grand nombre de variables. Typiquement pour de l'optimisation topologique les méthodes basés sur le gradient sont préférées ;
- Devient complexe à coder avec des contraintes importantes ou avec des opérateurs élaborés ;
- Peut induire un effet « boîte noire ».

Les algorithmes stochastiques sont à favorisées quand le problème est fortement non linéaire, sans possibilité de calculer un gradient, pas d'extremum à priori connu. Il faut aussi essayer de réduire le nombre de variables quand cela est possible. Pour les contraintes d'égalités, la substitution ou la pénalisation sont des solutions à privilégier.

1.2 Problème de minimisation globale : fonction *eggholder*

La fonction de *eggholder* est une fonction réelle de deux variables scalaires. Elle comporte un grand nombre de minimas locaux et représente un cas d'usage pour les routines d'optimisation globale. [4]

$$eggholder(x,y) = -(y+47)\cdot\sin\left(\sqrt{\left|y+\frac{x}{2}+47\right|}\right) - y\cdot\sin\left(\sqrt{|x-(y+47)|}\right)$$

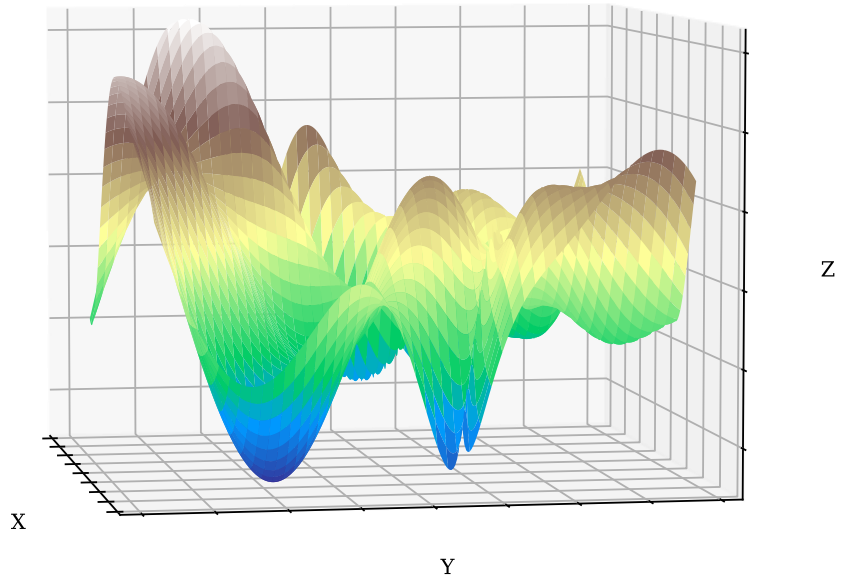


Figure 15 : Fonction eggholder

2 Recuit simulé à variables réelles (SA)

2.1 Analogie avec la métallurgie

Le recuit simulé (*simulated annealing* ou SA) est un algorithme métaheuristique qui vise à imiter le processus de recuit issu de la métallurgie. Dans la fabrication d'une pièce mécanique, les étapes de formation (moulage), mise en forme (usinage, pliage …) et traitements thermiques (trempes superficielles), induisent des contraintes internes qui peuvent la fragiliser. Afin de réduire ces contraintes on réalise une chauffe puis un refroidissement contrôlé. Ce processus appelé recuit permet de retrouver un état d'équilibre de plus faible énergie. Le chauffage aide le système à s'extraire d'un état stable qui représente un minimum local vers un minimum global.

La loi de Boltzmann donne la probabilité qu'une particule à la température T change d'état énergétique de E_1 vers E_2. Cette probabilité s'exprime par $p = \exp\left(-\frac{E_2-E_1}{kT}\right)$. On remarque que si le second état énergétique est supérieur au premier, il y a une probabilité non nulle que le système transite vers celui-ci. C'est le phénomène qui permet au cristal de sortir d'un minimum local.

L'algorithme de recuit simulé ré-utilise et généralise le concept d'énergie à la fonction objectif que l'on vise à minimiser. La constante de Boltzmann disparait et une température fictive remplace l'aspect thermique. Cette méthode a été mis au point en 1983 par S. Kirkpatrick, C.D. Gelatt et M.P. Vecchi de la société IBM et indépendamment par V. Černy en 1985. [5] [6] [7]

2.2 Description de l'algorithme

L'algorithme vise à résoudre le problème suivant :

$$\min f(\boldsymbol{x})$$

$$\boldsymbol{x} \in [\boldsymbol{x}_{min}\,;\boldsymbol{x}_{max}]$$

Les données initiales :

Un état initial est donné par la température T_0, un vecteur $\boldsymbol{x}_0$ aléatoire et une énergie $E_0 = f(\boldsymbol{x}_0)$.

La boucle itérative :

- Perturbation du système vers (1) : $E_1 = f(\boldsymbol{x}_1)$ et $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \delta\boldsymbol{x}$ avec $\delta\boldsymbol{x}$ un vecteur de déplacement aléatoire ;
- La solution $\boldsymbol{x}_1$ est accepté avec une probabilité $p = \exp\left(-\frac{E_1 - E_0}{T}\right)$;
  - Si la solution est acceptée, alors l'état (0) est remplacé par l'état (1) ;
- La température T décroit.

L'algorithme comprend des étapes supplémentaires pour vérifier les bornes du vecteur de recherche $\boldsymbol{x}$. Les contraintes peuvent être ajoutée avec une méthode de pénalisation quadratique.

Décroissance de la température :

Il existe plusieurs techniques de décroissance de la température pour le recuit simulé. Une méthode très simple et relativement efficace est une décroissance géométrique de rapport r :

$$T_{k+1} = T_k \times r$$

D'autres approches utilisent des lois complexes et intégrent des paliers sans décroissance de la température. La méthode précédente a l'avantage d'intégrer une technique de détermination des paramètres de l'algorithme présentée ci-dessous.

Détermination des paramètres de l'algorithme :

Un défaut important du recuit simulé est la difficulté à déterminer les paramètres de décroissance de la température. On peut ajouter une procédure de réglage de l'algorithme :

- Faire n (100 environ) permutations aléatoires et évaluer la différence d'énergie moyenne δE ;
- Choisir un taux d'acception initial θ_0 :
  - Très haute température $\theta_0 = 0.85$;
  - Haute température $\theta_0 = 0.60$;
  - Basse température $\theta_0 = 0.20$;
- Déduire la température initiale :
  - $\theta_0 = \exp\left(-\frac{\delta E}{T_0}\right) \quad \leftrightarrow \quad T_0 = -\delta E / \ln(\theta_0)$
- Déduire le taux de décroissance :
  - Nombre d'itérations de l'algorithme N ;
  - Taux d'acception final $\theta_n = 0.001$;
  - Température finale $T_n = -\delta E / \ln(\theta_f)$;
  - Facteur de décroissance $r = \left(\frac{T_N}{T_0}\right)^{1/N}$;

Lors des permutations nécessaires à cette pré-routine, il est possible de mémoriser la meilleure solution rencontrée et de l'utiliser comme point de démarrage de l'algorithme. Avec cet ajout, il est possible de faire une initialisation à basse température et améliorer la convergence de l'algorithme.

Condition d'arrêt et de convergence :

La meilleure solution rencontrée doit être gardée en mémoire. Si cette solution n'évolue pas pendant un certain nombre d'itération, on peut choisir d'interrompre le programme. Une interruption prématurer peut être suivie de l'exécution d'un algorithme de recherche locale. Dans ce cas de figure il est préférable d'utiliser une méthode sans gradient comme Nelder-Mead ou Powell.

Vérification lors des premières exécutions :

Lors de plusieurs exécutions la qualité des résultats obtenus est variable mais doit converger vers des solutions proches. L'utilisation d'une routine de renforcement de la convergence avec un algorithme d'optimisation local permet de garantir les résultats. Il est important d'observer l'allure de la convergence de la résolution afin d'augmenter ou diminuer le nombre d'itérations.

Code Python : le recuit simulé

```
import numpy as np
import numpy.random as rd

def recuit_simule(func,xmin,xmax,maxiter=1000):

  # parametres
  perturbRatio = 0.5 #Ratio de déplacement maximale

  # solution initiale
  ndof = len(xmin)
  s0 = rd.sample(ndof)
  x0 = s0*(xmax-xmin) + xmin
  E0 = func(x0)

  # initialisation de la température et du taux de décroissance
  initialTemp,decreaseRatio = autoSetUp(func,ndof,maxiter)
  T = initialTemp

  # solution optimale
  xopt = x0[:]
  Eopt = E0

  for iter in range(maxiter):

    # perturbation de l'etat du systeme
    x1 = x0 + perturbRatio*2*(rd.sample(ndof)-0.5)
    x1 = np.minimum(xmax,np.maximum(xmin,x1))
    E1 = func(x1)
    deltaE = E1-E0

    # acceptation de la solution
    if deltaE < 0.0 :
      x0 = x1[:]
      E0 = E1

      # solution optimale
      if E0 < Eopt :
```

```
        xopt = x0[:]
        Eopt = E0

    # acceptation d'une solution dégradée
    elif rd.random() < np.exp(-deltaE/T) :
      x0 = x1[:]
      E0 = E1

    # réduction de la température
    T = T*decreaseRatio
  return xopt
```

Code Python : routine de paramétrisation de la température

```
def autoSetUp(func,ndof,maxiter):
  # 100 permutations aléatoires
  rd_xarray = np.array([rd.sample(ndof)*(xmax-xmin) + xmin \
                    for i in     range(100)])
  farray = np.array([func(xi) for xi in rd_xarray])

  # Evaluation de la différence d'énergie
  deltaE = farray[1:]-farray[:-1]
  # Valeur moyenne des valeurs positives
  deltaE_avg = np.mean(deltaE[deltaE>0])

  #Démarrage haute température
  initialTemp = -deltaE_avg/np.log(0.60)
  finalTemp = -deltaE_avg/np.log(0.01)
  decreaseRatio = (initialTemp/finalTemp)**(1/maxiter)

  return initialTemp,decreaseRatio
```

2.3 Test du recuit simulé : *eggholder*

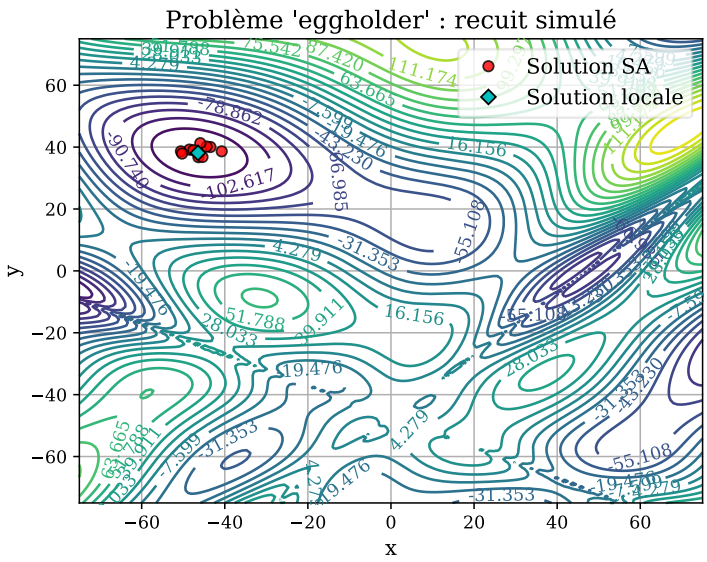


Figure 16 : résolution eggholder, recuit simulé

On se propose de résoudre le problème suivant à 10 reprises :

$$\min eggholder(x, y)$$

$$-75 \leq x, y \leq 75$$

Le nombre d'itération est limité à 500. Il y a 100 évaluations supplémentaires effectuées pour la détermination des paramètres de température. Une résolution locale est appliquée pour assurer la convergence et vérifier les résultats intermédiaires données par l'algorithme de recuit simulé. Les solutions observées sont uniformes et les méthodes locales convergent vers le même minimum.

2.4 Limitations et amélioration du recuit simulé

Une limitation de l'algorithme est sa vitesse de convergence. L'opérateur le plus important est la perturbation de la solution courante dans son voisinage. Un algorithme naïf utilise une loi de probabilité uniforme ou normale. Si le déplacement du vecteur de recherche est trop restreint, la recherche risque de stagner dans un minimum local. Si la recherche est trop large, le processus aléatoire risque de ne pas converger avec une précision suffisante. Des approches performantes ont été développées pour les problèmes à variables réelles.

Tsallis C, Stariolo DA proposent en 1996 une version généralisée du recuit simulé (GSA : *Generalised Simulated Annealing*). Aussi H. Szu et R. Hartley publient une méthode rapide (FSA : *Fast Simulated Annealing*) en 1987. Ces approches améliorent l'opérateur de perturbation avec des recherches localisées. En 2002, Xiang Y, Gubian S, Suomela B, et Hoeng J combinent ces deux méthodes et améliorent la convergence en utilisant une distribution de Cauchy-Lorentz qui se modifie dynamiquement. La méthode de décroissance de la température est elle aussi modifiée. Ces dernières améliorations sont exploitées pour des problèmes à variables réelles. *Scipy* utilise cette dernière approche pour sa routine : *dual_annealing*.

Il possible d'étendre l'algorithme de recuit simulé aux problèmes discrets, combinatoires ou mixte. Il suffit de modifier l'opérateur de perturbation de la solution. Cela permet à l'algorithme d'aborder une grande quantité de problème en gardant une architecture très simple. De plus, la méthode est particulièrement adaptée à la programmation orientée objet.

3 L'évolution différentielle (DE)

3.1 Origines de la méthode

L'évolution différentielle est une méthode issue des algorithmes évolutionnistes qui s'inspirent des dynamiques d'adaptation du monde vivant. Il en existe une grande variété : l'algorithme génétique, les essaims particulaires, les colonies de fourmis, les essaims d'abeilles… Ces méthodes sont basées sur l'évolution d'un ensemble de solution candidates appelé population. Le vocabulaire utilisé est similaire aux domaines d'études d'origine : phéromones, dynamique particulaire, stratégie d'évolution, gênes, sélection ou encore reproduction. Ces méthodes se démocratisent dans les années 80 et 90 et n'ont alors de cesse d'être améliorées.

L'évolution différentielle est inspirée de l'algorithme génétique combiné à une interprétation géométrique. L'autoadaptation de la population est réalisée via un vecteur calculé à partir de plusieurs individus. Le nom « différentiel » désigne la partie géométrique de la méthode, alors que « l'évolution » fait référence aux dynamiques du monde vivant. L'opérateur d'évolution proposé par K. Price et R. Storn [8] en 1995 est simple dans sa construction mais se révèle être très puissant.

Les extensions actuelles peuvent gérer des problèmes à contraintes non-linéaires. Dans le cas présent, une pénalisation est suffisante. J. Lampinen [9] propose une amélioration de la prise en compte des contraintes en 2002. D'autres extensions moins intuitives peuvent aborder des problèmes à variables mixtes.

3.2 Principes de l'algorithme

3.2.1 Structure et paramètres

L'algorithme d'évolution différentielle se structure autour de trois opérateurs ; la mutation, la recombinaison (ou croisement) et la sélection. Les stratégies d'évolution sont déclinées suivant une

nomenclature simple : *DE/X/Y/Z*. Le terme *DE* désigne *differential evolution*, *X* est le point à modifié, *Y* le nombre de différenciation et *Z* la méthode de croisement. Ainsi, la stratégie la plus commune est *DEbest1bin*, c'est-à-dire que le meilleur individu (*best*) est mis à jour avec une différentielle et un croisement binomial. Cette nomenclature peut paraître abstraite mais elle est bien documentée et les subtilités sont rapides à prendre en main. Deux paramètres permettent d'ajuster la recherche : F le facteur de recombinaison et *CR* le taux de croisement. L'algorithme est apprécié pour sa simplicité de paramétrage et les paramètres sont communément choisis avec $F \in [0.5,2.0]$ aléatoire et $CR = 0.9$.

Tableau 2 : paramètre DE

Paramètre	Description	Valeur conseillée
popsize	Taille de la population	5 à 15 fois la taille du problème
maxiter	Nombre d'itérations	1000 à 5000
atol, tol	Tolérance de convergence	0.0 et 0.0001
CR	Taux de croisement	0.9
F	Facteur de recombinaison	Aléatoire uniforme entre [0.5,2.0]

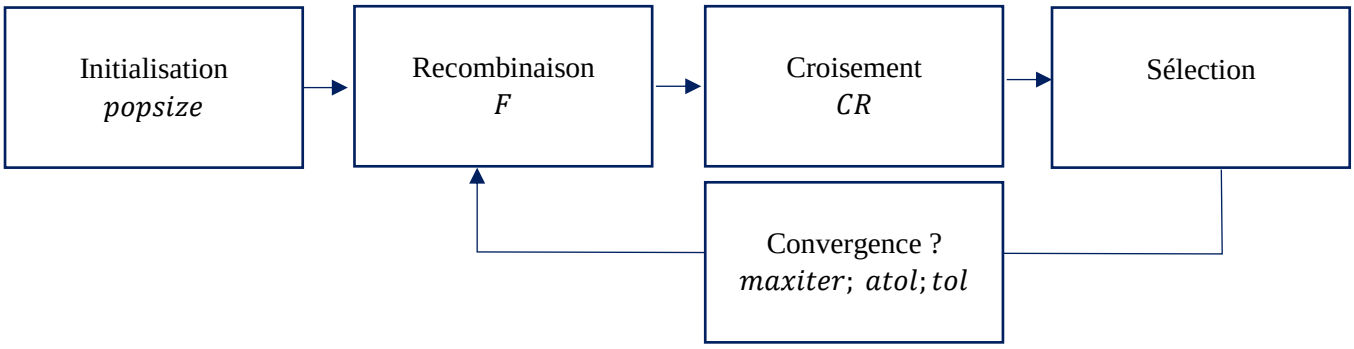


Figure 17 : Structure de la méthode DE

3.2.2 Initialisation de l'algorithme et *Latin hypercube sampling*

La population initiale est choisie aléatoirement dans l'espace des variables. Il existe plusieurs méthodes de génération d'une population. Une loi uniforme sur l'ensemble de l'espace peut générer des *clusters*, c'est-à-dire des groupes de point proches. On parle de *random sampling*. Une méthode plus robuste est le *Latin hypercube sampling (LHS)* introduit dans les années 80. [10]

```
import numpy as np
import numpy.random as rd
def LHS(shape):
    popsize = shape[0]
    ndof = shape[1]
    dsize  = 1.0/popsize

    sample_array = dsize*rd.sample(size=shape)
    sample_array[:] += np.linspace(0.,1.,popsize)[:,np.newaxis]
    init_pop = np.zeros(shape)
    for dim in range(ndof) :
        rdm_order = rd.permutation(popsize)
        init_pop[:,dim] = sample_array[rdm_order,dim]

    return init_pop
```

3.2.3 Recombinaison des vecteurs

Pour chaque individu $\boldsymbol{x}_i$ la méthode produit un vecteur mutant $\boldsymbol{v}_i$ duquel il pourra hériter d'une partie. Il existe une grande variété de méthode de recombinaison. Les six schémas les plus populaires sont présentés ici :

« DE / rand / 1 » $$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F \cdot (\boldsymbol{x}_{r2} - \boldsymbol{x}_{r3})$$

« DE / best / 1 » $$\boldsymbol{v}_i = \boldsymbol{x}_{best} + F \cdot (\boldsymbol{x}_{r2} - \boldsymbol{x}_{r3})$$

« DE / rand / 2 » $$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F \cdot (\boldsymbol{x}_{r2} + \boldsymbol{x}_{r3} - \boldsymbol{x}_{r4} - \boldsymbol{x}_{r5})$$

« DE / best / 2 » $$\boldsymbol{v}_i = \boldsymbol{x}_{best} + F \cdot (\boldsymbol{x}_{r2} + \boldsymbol{x}_{r3} - \boldsymbol{x}_{r4} - \boldsymbol{x}_{r5})$$

« DE / rand / 2 » $$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F \cdot (\boldsymbol{x}_{r2} + \boldsymbol{x}_{r3} - \boldsymbol{x}_{r4} - \boldsymbol{x}_{r5})$$

« DE / best / 2 » $$\boldsymbol{v}_i = \boldsymbol{x}_{best} + F \cdot (\boldsymbol{x}_{r2} + \boldsymbol{x}_{r3} - \boldsymbol{x}_{r4} - \boldsymbol{x}_{r5})$$

« DE / currenttobest / 1 » $$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F \cdot (\boldsymbol{x}_{best} + \boldsymbol{x}_{r2} - \boldsymbol{x}_i - \boldsymbol{x}_{r3})$$

« DE / currenttorand / 1 » $$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F \cdot (\boldsymbol{x}_{r1} + \boldsymbol{x}_{r2} - \boldsymbol{x}_i - \boldsymbol{x}_{r3})$$

Avec $r1\,;r2\,;r3\,;r4\,;r5$ des rangs aléatoires distincts. Ces six stratégies permettent de contrôler la diversité et la vitesse de convergence de la méthode. *Best1* est la méthode qui converge le plus vite. *Rand2* ou *best2* assurent la diversité des solutions.

3.2.4 Croisement, sélection et convergence

Il existe communément deux méthodes de croissement : le croisement binomial où chaque degré de liberté possède une probabilité de croisement indépendante et le croisement exponentiel ou les évènements ne sont plus indépendants. Un point de croisement est choisi dans le candidat puis la probabilité de croisement diminue dans le point suivant par permutation circulaire. Pour des raisons de simplicité les croisements binomiaux seront préférés. Si la composante *j* du candidat a une probabilité de croisement positive, la composante est remplacée par le mutant $\boldsymbol{x}'$, sinon le candidat $\boldsymbol{x}_{orig}$ reste inchangé.

Le croisement génère un candidat $\boldsymbol{x}_{trial}$ qui est conservé si la fonction est inférieure à la valeur du candidat d'origine. Si celle-ci est aussi inférieure à la meilleure solution enregistrée, alors les vecteurs de population et d'évaluation sont ajustés immédiatement pour gagner en vitesse de convergence. C'est une mise à jour immédiate. Pour un comportement similaire de K.Price et R.Storn, le meilleure individu sera considéré à la fin de chaque génération (itération). La méthode de sélection est ajustable pour prendre en compte les contraintes comme le suggère Lampinen [9].

La méthode convergence si le nombre d'itération atteint le maximum prescrit ou si la condition suivante est validée :

$$\mathrm{std}\left(\boldsymbol{f}_{population}\right) \leq atol + tol \cdot \mathrm{abs}\left(\mathrm{mean}\left(\boldsymbol{f}_{population}\right)\right)$$

Avec $\boldsymbol{f}_{population}$ le vecteur d'évaluation des individus, $std\,;abs\,;mean$ les fonction écart-type, valeur absolue et moyenne. Cette condition est proposée dans la routine *differential_evolution* de *Scipy*. Il est aussi judicieux de rajouter un nombre maximal d'itérations amélioration de la meilleure solution.

3.2.5 Algorithme d'évolution différentielle

```
def differential_evolution(func,xmin,xmax,popsize=20,maxiter=1000):
    atol,tol = 0.0,0.001
    CR = 0.9
    F = [0.5,2.0]

    ndof = len(xmin)
    population_x = LHS((popsize,ndof)) * (xmax-xmin) + xmin
    population_f = np.array([func(xi) for xi in population_x])

    bestarg = np.argmin(population_f)
    population_x[[0,bestarg],:] = population_x[[bestarg,0],:]
    population_f[[0,bestarg]] = population_f[[bestarg,0]]
    best_f = population_f[0]

    for iter in range(maxiter):

        scale_factor = rd.sample(ndof)*(F[1]-F[0]) + F[0]
        for k,xk in enumerate(population_x) :
            #recombinaison
            x_prime = recombinaison(k,population_x,scale_factor)
            x_prime = np.maximum(xmin,np.minimum(x_prime,xmax))

            #croisement
            croisement = rd.sample(size=ndof) <= CR
            x_trial = np.where(croisement,x_prime,xk)

            #selection
            fk = population_f[k]
            f_trial = func(x_trial)
            if f_trial < fk :
                population_x[k] = x_trial
                population_f[k] = f_trial

                if f_trial < best_f :
                    population_x[[0,k],:] = population_x[[k,0],:]
                    population_f[[0,k]] = population_f[[k,0]]
                    best_f = population_f[0]

        #convergence
        if convergence(population_f,atol,tol) :
            print("SOLUTION CONVERGED : iter %i"%iter)
            break

    return population_x[0]
```

3.3 Test évolution différentielle : *eggholder*

Comme pour le recuit simulé on se propose de résoudre le problème suivant à 10 reprises :

$$\min eggholder(x, y)$$

$$-75 \leq x, y \leq 75$$

Le nombre d'évaluation de la fonction étant limité à 500, donc le nombre d'itération maximum est de 12 et la population sera de 40 individus. La solution convergence avant le nombre maximum d'itération et l'erreur par rapport à la recherche de minimum local est très inférieure à celle observée sur le recuit simulé. Cette observation est attendue car le recuit simulé en variable réelle converge difficilement sans amélioration. De plus l'évolution différentielle *best1bin* a tendance à converger rapidement par rapport aux autres méthodes stochastiques. Cette tendance peut devenir préjudiciable pour les problèmes contraints et en plus grande dimension. Les stratégies *best2bin, rand2bin* et le croisement exponentiel (*best2exp, rand2exp*) améliorent la diversité de la recherche.

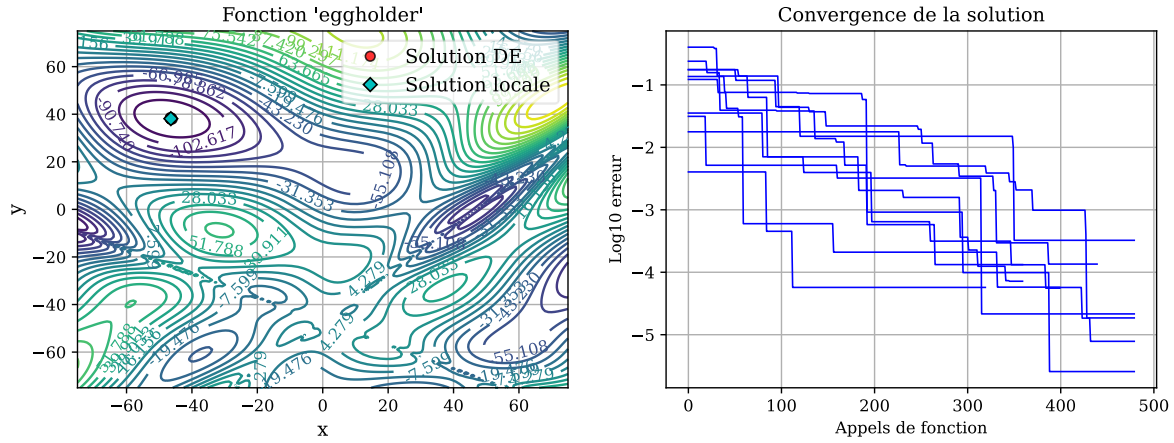


Figure 18 : Résolution eggholder évolution différentielle

Références

[1] P. Armand, «Algorithme d'optimisation par pénalisation mixte : Lagrangien augmenté et barrière logarithmique,» 2015.

[2] T. V. Mikosch, S. I. Resnick et S. M. Robinson, «Numerical optimization,» Springer , New York, 2006.

[3] A. R. Parkinson, R. J. Balling et J. D. Hedengren, Optimization methods for engineering design : applications and theory, Brigham Young University, 2013.

[4] Global Optimization Test Functions Index., Retrieved, June 2013.

[5] C.Tsallis et D.A.Stariolo, Generalized Simulated Annealing, Physica A, 1996.

[6] «Recuit simulé,» 06 2021. [En ligne]. Available: http://fr.wikipedia.org/w/index.php?title=Recuit_simul%C3%A9&oldid=183985537.

[7] L.Uhl, 1001 codes Python pour la modélisation, Ellipses, 2014.

[8] K. Price et R.Storn, Differential Evolution - a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, 1997.

[9] J.Lampinen, A constraint handling approach for the differential evolution algorithm., Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600). Vol. 2. IEEE, 2002.

[10] N. Durand, «Algorithmes Génétiques et autres méthodes d'optimisation appliqués à la gestion de trafic aérien,» 2004.

[11] B. Amine, «Le recuit simulé,» 2011.

[12] «Golden-section search,» 2021. [En ligne]. Available: https://en.wikipedia.org/wiki/Golden-section_search.

[13] «Multiplicateur de Lagrange,» [En ligne]. Available: https://fr.wikipedia.org/wiki/Multiplicateur_de_Lagrange.

[14] G. Tushar, «Elitist Non-dominated Sorting Genetic Algorithm: NSGA-II».

[15] R.Hauser, «Line Search Methods for unconstrained optimization,» Lecture 8, Numerical Linear Algebra and Optimisation Oxford University Computing Laboratory, MT, 2007.

[16] P. W, S.A.Teukolsky, W.T.Vetterling et B.P.Flannery, «Numerical Recipes,» Cambridge University Press.

[17] K. Deb, A. Pratap, S. Agarwal et T. Meyarivan, «A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II,» IEEE, 2002.

41 Optimisation non linéaire à variables réelles