Synthèse Intégrale : Optimisation Numérique (TP1-TP4)

Révision Examen

Janvier 2026

1 TP 1 : Minimisation sans contrainte - Fondements

L'objectif est de trouver $x^* = \arg\min f(x)$ par des méthodes de descente. La validation du code (gradient/Hessienne) est la première étape indispensable.

1.1 Validation des calculs : Le Test de Taylor

Pour vérifier que le gradient analytique est correct, on compare la variation réelle de la fonction avec l'approximation linéaire de Taylor :

$$f(x + \epsilon d) \approx f(x) + \epsilon \langle \nabla f(x), d \rangle$$

On observe le ratio $\frac{|f(x+\epsilon d) - f(x)|}{\epsilon \langle \nabla f(x), d \rangle}$. Si le gradient est juste, ce ratio doit tendre vers 1 quand $\epsilon \to 0$ (jusqu'aux limites de précision machine).

1.2 Direction de Newton et Sécurisation

La direction de Newton $d_k = -[\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$ est extrêmement rapide près du minimum (convergence quadratique) mais peut être instable ailleurs (si la Hessienne n'est pas définie positive).

Stratégie de Sécurisation

On vérifie l'angle entre la direction proposée d et le gradient. Si $\cos(\theta) = \frac{\langle d, -\nabla f(x) \rangle}{\|d\| \|\nabla f(x)\|} < 0.1$, la direction est jugée "mauvaise" et on bascule sur une descente de gradient classique ($d = -\nabla f(x)$).

Listing 1 – Implémentation de Newton sécurisée

```
def dc_Newton(x, function, df):
    H = function.Hess(x)
    try:
        # Résolution du système linéaire (plus stable que l'inversion)
        d = np.linalg.solve(H, -df)
    except np.linalg.LinAlgError:
        return -df # Repli sur le gradient si H est singulière

    # Test de la condition de descente
    cos_theta = np.dot(d, -df) / (np.linalg.norm(d) * np.linalg.norm(df))
    if cos_theta > 0.1:
        return d
    return -df
```

1.3 Recherche Linéaire : Stabilité et Wolfe

Le choix du pas s est le second levier de performance.

- **Pas fixe (`ls_constant`) :** Simple mais diverge souvent sur des fonctions raides comme Rosenbrock.
- **Backtracking :** Divise le pas par 2 tant que $f(x + sd) \geq f(x)$. Assure la décroissance mais pas une avance optimale.

— **Conditions de Wolfe :** Le standard industriel.

— **Armijo (Admissibilité) :** $f(x+sd) \leq f(x) + c_1 s\langle \nabla f(x), d\rangle$. Évite les pas trop grands.

— **Courbure :** $\langle \nabla f(x+sd), d\rangle \geq c_2 \langle \nabla f(x), d\rangle$. Évite les pas trop petits.

Listing 2 – Recherche de pas de Wolfe (version robuste)

```
def ls_wolfe(x, function, step, descent, f, df):
    step_min, step_max = 0., np.inf
    scal = np.dot(df, descent)
    step2 = step
    eps1, eps2 = 0.1, 0.9

    while True:
        x2 = x + step2 * descent
        f2 = function.value(x2)
        if f2 > f + eps1 * step2 * scal: # Armijo échoue
            step_max = step2
            step2 = 0.5 * (step_min + step_max)
        else:
            df2 = function.grad(x2)
            if np.dot(df2, descent) < eps2 * scal: # Courbure échoue
                step_min = step2
                step2 = min(0.5 * (step_min + step_max), 2 * step_min)
            else:
                return x2, f2, df2, step2
```

2 TP 2 : Recherche linéaire et méthodes quasi-Newton

Ce chapitre traite de l'optimisation en grande dimension où le calcul de la Hessienne est prohibitif. On combine une recherche de pas robuste (Wolfe) avec une approximation de la courbure à mémoire limitée (L-BFGS).

2.1 Recherche linéaire sous conditions de Wolfe

Le choix du pas s est crucial. L'algorithme `ls_wolfe` cherche un compromis entre une décroissance suffisante et un progrès réel.

— **Condition d'Armijo (Admissibilité) :** $f(x+sd) \leq f(x) + c_1 s\langle \nabla f(x), d\rangle$. Elle évite les pas trop grands qui dépasseraient le minimum.

— **Condition de Courbure :** $\langle \nabla f(x+sd), d\rangle \geq c_2 \langle \nabla f(x), d\rangle$. Elle évite les pas trop petits qui ralentiraient la convergence.

Listing 3 – Implémentation de la recherche de pas de Wolfe

```
def ls_wolfe(x, function, step, descent, f, df):
    step_min, step_max = 0., np.inf
    scal = np.dot(df, descent)
    step2 = step
    eps1, eps2 = 0.1, 0.9 # Paramètres c1 et c2

    while True:
        x2 = x + step2 * descent
        f2 = function.value(x2)
        if f2 > f + eps1 * step2 * scal: # Armijo échoue
            step_max = step2
            step2 = 0.5 * (step_min + step_max)
        else:
            df2 = function.grad(x2)
            if np.dot(df2, descent) < eps2 * scal: # Courbure échoue
                step_min = step2
```

```
                step2 = min(0.5 * (step_min + step_max), 2 * step_min)
            else:
                return x2, f2, df2, step2
```

2.2 L'algorithme L-BFGS (Limited-memory BFGS)

L'algorithme L-BFGS estime l'inverse de la Hessienne sans jamais stocker de matrice $n \times n$. Il utilise uniquement les m derniers couples de vecteurs :

— $s_i = x_i - x_{i-1}$ (variation de position)

— $y_i = \nabla f(x_i) - \nabla f(x_{i-1})$ (variation de gradient)

— $\rho_i = 1/\langle s_i, y_i \rangle$ (doit être > 0 pour garantir la convexité locale)

2.2.1 Algorithme "Two-Loop Recursion"

La direction de descente $r = -H_k \nabla f(x_k)$ est obtenue par deux passes successives sur l'historique :

Algorithme de calcul de direction

1. $q = \nabla f(x_k)$
2. **Passe arrière :** Pour i de k à $k-m$, $\alpha_i = \rho_i(s_i \cdot q)$, puis $q = q - \alpha_i y_i$
3. **Scaling :** $r = \frac{s_k \cdot y_k}{y_k \cdot y_k} q$
4. **Passe avant :** Pour i de $k-m$ à k, $\beta_i = \rho_i(y_i \cdot r)$, puis $r = r + (\alpha_i - \beta_i)s_i$
5. **Résultat :** r est la direction Quasi-Newton.

Listing 4 – La classe BFGS et la gestion du stockage

```
class BFGS():
    def __init__(self, nb_stock_max=8):
        self.nb_stock_max = nb_stock_max
        self.stock = []       # Stockage des triplets [s, y, rho]
        self.last_iter = []   # Sauvegarde [x_prec, g_prec]

    def push(self, x, g):
        if self.last_iter != []:
            s, y = x - self.last_iter[0], g - self.last_iter[1]
            rho_inv = np.dot(y, s)
            if rho_inv > 0:
                if len(self.stock) == self.nb_stock_max:
                    self.stock.pop(0)
                self.stock.append([s, y, 1.0 / rho_inv])
            else:
                self.stock = [] # Réinitialisation si instabilité
        self.last_iter = [x, g]

    def get(self, grad):
        q = -grad
        if len(self.stock) > 0:
            L_a = []
            for s, y, rho in reversed(self.stock):
                alpha = rho * np.dot(s, q)
                L_a.append(alpha)
                q -= alpha * y
            L_a.reverse()

            # Scaling initial (Gamma)
            gamma = np.dot(self.stock[-1][0], self.stock[-1][1]) / np.dot(self.
                stock[-1][1], self.stock[-1][1])
            q *= gamma

            for (s, y, rho), alpha in zip(self.stock, L_a):
```

```
            beta = rho * np.dot(y, q)
            q += s * (alpha - beta)
    return q
```

2.3 Comparaison et Performance

Sur la fonction de Rosenbrock, L-BFGS avec recherche de Wolfe permet d'atteindre la convergence en une cinquantaine d'itérations sans jamais calculer de dérivée seconde.

Comparaison Scipy : L'utilisation de `scipy.optimize.minimize` avec l'option `jac=f.grad` montre que nos implémentations manuelles atteignent des performances comparables aux solveurs industriels (L-BFGS-B), validant ainsi la robustesse de notre recursion et de notre gestion du pas de Wolfe.

3 TP 3 : Méthodes de Point Intérieur

Cette méthode permet de résoudre des problèmes de minimisation sous contraintes d'inégalité : $\min f(x)$ s.t. $g_i(x) \leq 0$ en transformant les contraintes "dures" en pénalités "souples".

3.1 La Barrière Logarithmique

On définit la fonction barrière f_t pour un paramètre de pénalité $t > 0$:

$$f_t(x) = f(x) - \frac{1}{t} \sum_{i=1}^{p} \ln(-g_i(x))$$

Plus x s'approche de la frontière ($g_i(x) \to 0^-$), plus le terme $-\ln(-g_i)$ tend vers $+\infty$, créant une "barrière" infranchissable.

3.1.1 Calcul des dérivées

Pour appliquer Newton, on utilise les expressions suivantes :

— **Gradient :** $\nabla f_t(x) = \nabla f(x) - \frac{1}{t} \sum \frac{1}{g_i(x)} \nabla g_i(x)$

— **Hessienne :** $Hf_t(x) = Hf(x) - \frac{1}{t} \sum \left(\frac{1}{g_i(x)} Hg_i(x) - \frac{1}{g_i(x)^2} \nabla g_i(x) \nabla g_i(x)^T \right)$

3.2 Méthode du Chemin Central (Homotopie)

L'idée est de suivre la trajectoire des minima $x^*(t)$ lorsque $t \to \infty$.

Algorithme du Chemin Central

1. **Initialisation :** $t = t_{start}$, $x = x_{admissible}$ strictement ($g(x) < 0$).
2. **Boucle externe :** Tant que $p/t > \epsilon$ (où p est le nb de contraintes) :
   — **Boucle interne :** Minimiser $f_t(x)$ via Newton (Warm Start).
   — **Mise à jour :** $t \leftarrow \mu \cdot t$ (avec $\mu \approx 10$ à 100).

3.3 Application au Lasso (Reformulation Lisse)

Le Lasso $\min \frac{1}{2}\|Bw - y\|^2 + \lambda \|w\|_1$ n'est pas dérivable en zéro. On le reformule en variables positives $w = x^+ - x^-$ avec $x^+, x^- \geq 0$:

$$\min \frac{1}{2}\|B(x^+ - x^-) - y\|^2 + \lambda \sum (x_i^+ + x_i^-) \quad \text{s.t.} \quad -x_i^+ \leq 0, -x_i^- \leq 0$$

Cela permet d'utiliser les méthodes de point intérieur sur un problème quadratique à contraintes linéaires.

On transforme $\min f(x)$ s.t. $g(x) \leq 0$ via une barrière logarithmique.

3.4 Définition et Dérivées

$$f_t(x) = f(x) - \frac{1}{t}\ln(-g(x))$$

Justification de la convergence linéaire

L'erreur théorique est bornée par $\frac{p}{t}$ (où p est le nombre de contraintes). Comme on multiplie t par μ à chaque itération externe ($t_k = t_0\mu^k$), l'erreur décroit géométriquement : $\ln(\text{erreur}) \leq \ln(p/t_0) - k\ln(\mu)$. C'est une **convergence linéaire**.

4 TP 4 : Échantillonnage Compressif et FISTA

4.1 Du théorème de Shannon à l'échantillonnage compressif

Le théorème de Shannon impose une fréquence d'échantillonnage $f_e \geq 2f_{\max}$. L'échantillonnage compressif (Compressed Sensing, CS) permet de s'affranchir de cette contrainte si le signal $x \in \mathbb{R}^n$ est **parcimonieux** dans un dictionnaire $\Psi \in \mathbb{R}^{n\times m}$:

$$x = \Psi\alpha, \qquad \|\alpha\|_0 = k \ll n.$$

On réalise $p < n$ mesures linéaires aléatoires via une matrice $A \in \mathbb{R}^{p\times n}$:

$$y = Ax = A\Psi\alpha.$$

4.2 Formulation du problème d'optimisation (Lasso)

La reconstruction du signal repose sur la résolution d'un problème convexe de type Lasso, combinant une fidélité aux données (L_2) et une pénalité de parcimonie (L_1) :

$$\min_{\alpha\in\mathbb{R}^m} F(\alpha) = \underbrace{\frac{\sigma}{2}\|A\Psi\alpha - y\|_2^2}_{J(\alpha)} + \underbrace{\|\alpha\|_1}_{h(\alpha)}.$$

4.3 Préliminaires mathématiques

Gradient de la partie lisse. La fonction $J(\alpha)$ est différentiable et son gradient s'écrit :

$$\nabla J(\alpha) = \sigma(A\Psi)^\top(A\Psi\alpha - y) = \sigma\Psi^\top A^\top(A\Psi\alpha - y).$$

Constante de Lipschitz. Le gradient ∇J est Lipschitzien de constante L. Une borne supérieure pratique utilisée en implémentation est :

$$L \;=\; 2\sigma\,\mathrm{Tr}(A^\top A) \;\approx\; 2\sigma\sum_{i,j} A_{i,j}^2.$$

4.4 Opérateur proximal et seuillage doux

La norme L_1 n'étant pas dérivable en 0, on utilise son opérateur proximal :

$$\mathrm{prox}_{s\|\cdot\|_1}(v) = \mathrm{sign}(v)\ \max(|v| - s, 0),$$

appelé *seuillage doux*. Il favorise naturellement les solutions parcimonieuses.

4.5 Algorithme proximal simple (ISTA / Forward–Backward)

L'algorithme ISTA combine un pas de gradient sur la partie lisse J et un pas proximal sur h :

$$\alpha_{k+1} = \mathrm{prox}_{\frac{1}{L}\|\cdot\|_1}\left(\alpha_k - \frac{1}{L}\nabla J(\alpha_k)\right).$$

Listing 5 – Proximal Gradient Descent (ISTA)

```
def RestoreX(A, y, sigma, nit):
    n = A.shape[1]
    alpha = np.zeros(2*n)
    L = np.sum(A*A) * 2 * sigma

    for k in range(nit):
        r = A @ Psi(alpha) - y
        grad = sigma * PsiT(A.T @ r)   # Gradient via opérateurs adjoints
        alpha = prox(alpha - (1/L)*grad, 1/L)
    return alpha
```

4.6 Accélération de Nesterov : FISTA

FISTA (Fast Iterative Shrinkage-Thresholding Algorithm) améliore la vitesse de convergence de ISTA, passant de $O(1/k)$ à $O(1/k^2)$, grâce à un terme d'inertie :

$$z_k = \alpha_k + \frac{k-1}{k+2}(\alpha_k - \alpha_{k-1}),$$

$$\alpha_{k+1} = \text{prox}_{\frac{1}{L}\|\cdot\|_1}\left(z_k - \frac{1}{L}\nabla J(z_k)\right).$$

Listing 6 – Algorithme FISTA / Nesterov

```
def Nesterov(A, y, sigma, nit):
    alpha = np.zeros(2*n)
    alpha_old = np.zeros(2*n)
    L = np.sum(A*A) * 2 * sigma

    for k in range(nit):
        beta = (k-1) / (k+2)
        yk = alpha + beta * (alpha - alpha_old)
        alpha_old = alpha.copy()

        r = A @ Psi(yk) - y
        grad = sigma * PsiT(A.T @ r)

        alpha = prox(yk - (1/L)*grad, 1/L)
    return alpha
```

4.7 Points clés à retenir

— **Parcimonie :** un signal k-parcimonieux peut être reconstruit avec environ $p \approx 2k$ mesures aléatoires.

— **Adjoints :** l'utilisation de $\Psi^\top$ (`PsiT`) permet d'éviter la construction explicite de matrices de grande taille.

— **Performance :** FISTA converge bien plus rapidement qu'ISTA, au prix d'une décroissance du coût non strictement monotone.

— **Convergence :** la stabilité dépend du choix correct de la constante de Lipschitz L.