TUTORIEL 1

Inpainting d'images par optimisation convexe

Cours INF4127 – OPTIMISATION

https://github.com/UE-Technique-d-Optimisation-II/TPE-INF4127-Unlocbox

Groupe 4

Encadrant : Pr Paulin MELATAGIA YONTA

Université de Yaoundé I

Membres du Groupe 4

Composition du groupe

AZEUFACK NGNINWO THIERRY Matricule : 12U0012

NDEKEBAI MEYIE MICHAEL Matricule : 21T2707

DJAMPA MBIANGANG PLATINY CABREL Matricule : 21T2437

MOUGOU OWOUNDI BRICE WILLIAM Matricule : 0000

Plan

Prérequis pour bien comprendre (définitions)

Mathématiques

- Optimisation convexe : **minimiser** une fonction avec **forme de bol** (un seul minimum)
- Algèbre linéaire : **matrices** (tableaux) et **vecteurs** (listes)
- Analyse numérique : **algorithmes itératifs** qui convergent pas à pas vers la solution

Logiciels

- MATLAB R2019b+ ou Octave 5.0+
- UNLocBoX (gratuit)
- Images de test (fournies)

Installation simple

```
addpath('chemin/vers/unlocbox'); % Ajoute UNLocBoX au chemin MATLAB
savepath; % Sauvegarde pour les sessions futures
```

Qu'est-ce qu'UNLocBoX ? - Introduction complète

DÉFINITION FORMELLE SIMPLIFIÉE

UNLocBoX = UNconstrained Optimization with Locally convex functions BOX

Traduction : Boîte à outils pour **résoudre des problèmes d'optimisation** où on cherche le **minimum** d'une fonction composée de plusieurs parties

Où ça vient ?

- Développée à l'**EPFL** (École Polytechnique Fédérale de Lausanne)
- Par le laboratoire **LTS2** (Laboratoire de Traitement des Signaux 2)
- Spécialisée en optimisation convexe non-lisse

Philosophie

"Décomposer un problème complexe en fonctions simples"

Glossaire des termes complexes d'UNLocBoX

TERMES À COMPRENDRE ABSOLUMENT

- **Optimisation convexe** : Minimiser une fonction qui a LA FORME D'UN BOL (un seul minimum global, pas de pièges)
- **Fonction non-lisse** : Fonction avec des **coins** ou **pointes** (pas dérivable partout, ex : valeur absolue $|x|$)
- **Opérateur proximal** : Généralisation du gradient pour fonctions non-lisses = "projection sur un ensemble proche"
- **FISTA** : **Fast Iterative Shrinkage-Thresholding Algorithm** = ISTA avec **mémoire** (comme une balle qui prend de la vitesse)
- **ADMM** : **Alternating Direction Method of Multipliers** = divise le problème en sous-problèmes plus simples
- **Constante de Lipschitz** : Mesure de la pente maximale du gradient (évite de prendre des pas trop grands)

Philosophie modulaire d'UNLocBoX - Avant/Après

Avant UNLocBoX (traditionnel)

- Vous codez **FISTA** à la main (50+ lignes)
- Vous vérifiez convergence (bug-prone)
- Vous gérez les paramètres (complexe)
- Code long, risqué, répétitif

Avec UNLocBoX (modulaire)

- Vous définissez **2 fonctions simples** (5 lignes)
- Vous appelez `solvep` (1 ligne)
- UNLocBoX gère FISTA pour vous
- Code court, sûr, réutilisable

Exemple concret

Problème : $\min f(x) + g(x)$
Avant : 100 lignes de FISTA
Avec UNLocBoX : `f.grad=@(x)...; g.prox=@(x)...; sol=solvep(x0,{f,g},param);`

Algorithmes inclus dans UNLocBoX (vous n'avez pas à les coder)

Algorithmes d'accélération (fast)

- **FISTA** : Fast Iterative Shrinkage-Thresholding - gradient + momentum
- **FBS** : **Forward-Backward Splitting** - version simple sans accélération
- **ADMM** : **Alternating Direction Method** - pour problèmes avec contraintes

Fonctions proximales fournies (prêtes à l'emploi)

- **prox_tv()** : Variation totale (pour préserver contours)
- **prox_l1()** : **Norme L1** (pour parcimonie)
- **prox_l2()** : **Norme L2** (pour moindres carrés)
- **prox_nucl()** : **Norme nucléaire** (pour matrices de rang faible)

Pourquoi c'est génial ?

Vous **combinez** ces briques comme des LEGO : `solvep` trouve automatiquement la bonne

Ce qu'on va résoudre - Problème concret

Problème réel (inpainting)

Situation : Vous avez une photo avec **50% de pixels manquants** (trous noirs)
Question : Comment reconstruire l'image originale sans les trous ?

Solution UNLocBoX qu'on va implémenter

1. **Respecter les pixels connus** : Ne PAS modifier les pixels qu'on connaît déjà
2. **Préserver les contours** : Les bords de l'horloge doivent rester nets
3. **Lisser les zones uniformes** : Les zones grises homogènes doivent être lisses
4. **Converger rapidement** : Trouver la solution en moins de 40 secondes

Objectif quantifiable

PSNR $>$ 28 dB + **$<$ 40 secondes** de calcul + **contours préservés**

Fonctions proximales - Explications ultra-détaillées

DÉFINITION PAS À PAS

Une **fonction proximale** est un objet MATLAB (structure) qui contient **TROIS méthodes** (fonctions) :

- `f.eval(x)` = **ÉVALUER** la fonction f au point x (donne un nombre)
  Exemple : Si $f(x) = x^2$, alors `f.eval(3)` retourne **9**
- `f.grad(x)` = **GRADIENT** de f au point x (vecteur des dérivées partielles)
  Exemple : Si $f(x) = x^2$, alors `f.grad(3)` retourne **6** (dérivée = 2x)
- `f.prox(x,T)` = **PROXIMAL** = projection au point x avec paramètre T
  Exemple : Si $f(x) = |x|$ (valeur absolue), alors `f.prox(3,1)` retourne **2** (réduit de 1)

Règle de base pour UNLocBoX

Chaque fonction doit avoir au moins :
[eval + grad] OU **[eval + prox]** (pas les trois obligatoirement)

Glossaire des termes mathématiques utilisés

TERMES À COMPRENDRE ABSOLUMENT

- **Gradient** (∇f) : Vecteur qui pointe dans la direction de la plus forte PENTE. Pour une fonction $f(x, y)$, c'est $(\partial f/\partial x, \partial f/\partial y)$
- **Proximal** (prox_{Tf}) : Version "adoucie" du gradient pour fonctions non-lisses. Trouve le point le plus proche qui minimise une fonction régularisée
- **Structure** (struct) : Objet MATLAB qui contient plusieurs champs (variables) groupées ensemble (comme un dossier)
- **Lambda** (λ) : Paramètre de pondération entre fidélité et régularisation. λ grand = beaucoup de régularisation, λ petit = peu de régularisation
- **Lipschitz** : Constante L = pente maximale autorisée du gradient (évite de "dépasser" le minimum)

Analgie visuelle

Le **gradient** = boussole qui indique où aller pour descendre.

Formulation mathématique - Décomposition du problème

MODÈLE DE DÉGRADATION (ce qui se passe sur l'image)

$$y = Ax + \varepsilon \quad \text{(Image observée = Image originale} \times \text{Masque + Bruit)}$$

Explication :

- x = image originale qu'on veut retrouver (inconnue)
- A = masque (0 ou 1) qui "efface" des pixels
- y = image dégradée qu'on observe (avec trous noirs)
- ε = bruit (petites erreurs, on l'ignore souvent)

Traduction simple

Vous avez une photo x, quelqu'un met des Post-it noirs dessus (opération A), vous obtenez y. Le but : retirer les Post-it pour retrouver x.

Formulation mathématique - Le problème d'optimisation

PROBLÈME À RÉSOUDRE (la formule magique)

$$\min_x \quad \underbrace{\frac{1}{2}\|Ax - y\|_2^2}_{\text{Fidélité aux données}} + \underbrace{\lambda\, TV(x)}_{\text{Régularisation TV}}$$

Terme de fidélité (partie 1) :

- $\|...\|_2^2$ = norme au carrée = somme des carrés des différences
- But : Assurer que la solution respecte les pixels connus
- **Exemple** : Si pixel (10,10) est connu (valant 0.5), la solution doit être proche de 0.5
- **Si cette partie = 0** : La solution colle

Terme TV (partie 2) :

- $TV(x)$ = Variation Totale = $\sum |\nabla x|$ (somme des différences entre voisins)
- But : Lisser les zones uniformes ET préserver les contours
- λ = poids qui équilibre les deux termes
- **Si cette partie = 0** : Image très lisse, mais contours perdus

Variation Totale (TV) - Explication ultra-détaillée

DÉFINITION DE LA TV (le "secret" de la préservation des contours)

$$TV(x) = \sum_{i,j} \sqrt{(x_{i+1,j} - x_{i,j})^2 + (x_{i,j+1} - x_{i,j})^2}$$

Traduction :

- Pour chaque pixel, on calcule la différence avec son voisin de droite et d'en bas
- On fait la racine carrée de la somme des carrés (norme du gradient)
- On somme sur toute l'image

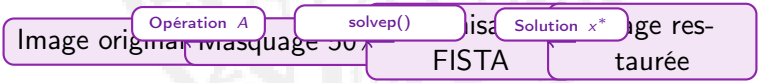
Pourquoi TV préserve les contours ?

Dans une zone lisse : différences petites → TV petite → OK de lisser
Sur un contour : différences grandes → TV grande → L'optimisation PEUT garder ces différences car elles sont "payantes" à lisser

À retenir

Les 5 étapes clés - Explications détaillées

1. **Chargement** : `y = imread('clock_256.gif')` - Lit l'image depuis le disque
2. **Simulation** : Crée un masque aléatoire (50% de 0, 50% de 1) puis fait `y_corrompue = mask .* y`
3. **Modélisation** : Définit f1 (fidélité avec gradient) et f2 (TV avec proximal)
4. **Résolution** : Appelle `solvep` qui fait la **magie** (combine f1 et f2 avec FISTA)
5. **Analyse** : Calcule PSNR, affiche images, trace courbe de convergence



Code étape 1 - Vérification et chargement

```
%% ========================================================================
% TUTORIEL UNLOCBOX - INPAINTING D'IMAGE
% Cours INF4127 - Universit  de Yaound  I
% ========================================================================

% V RIFICATION DE L'INSTALLATION (CRUCIAL)
% exist('solvep', 'file') retourne 2 si solvep.m existe dans le chemin MATLAB
% ~exist(...) = NOT exist      si solvep n'existe PAS, on g n re une ERREUR CLAIRE
if ~exist('solvep', 'file')
    error('UNLocBoX non install . Ajoutez-le au chemin MATLAB.');
end

% FERMETURE DES FEN TRES ET NETTOYAGE
close all; % Ferme TOUTES les figures ouvertes
clear;     % Efface TOUTES les variables du workspace (espace de travail)
clc;       % Nettoie la fen tre de commande (vide l' cran )

fprintf('=== D MARRAGE DU TUTORIEL UNLOCBOX ===\n\n');

%%  TAPE  1 : CHARGEMENT DE L'IMAGE ORIGINALE
fprintf('1. Chargement de l''image...\n');
% imread('nom') : lit l'image depuis le dossier courant
% 'clock_256.gif' : fichier 256x256 pixels en niveaux de gris
% im2double(...) : convertit les valeurs [0,255] (uint8)     [0,1] (double)
%                  ESSENTIEL pour les calculs d'optimisation (pr cision)
y_original = im2double(imread('clock_256.gif'));

% AFFICHAGE DE L'IMAGE ORIGINALE
figure('Position', [100 100 400 400]); % Cr e figure    position (x,y,largeur,hauteur)
imshow(y_original); title('Image originale'); colormap gray;
```

Code étape 2 - Création du masque et dégradation

```
%% TAPE 2 : CR ATION DU MASQUE (50% DE PIXELS CONNUS)
fprintf('2. Cr ation du masque...\n');
% rng(42) : initialise le g n rateur al atoire avec graine = 42
%                garantit la REPRODUCTIBILIT (toujours les m mes trous al atoires)
%           Utile pour comparer diff rents algorithmes avec les m mes donn es
rng(42);

% rand(size(y_original)) : g n re une matrice 256x256 de nombres al atoires entre 0 et 1
% > 0.5 : cr e une matrice LOGIQUE (valeurs TRUE/FALSE)
%             ~50% de TRUE (valeur 1, pixels conserv s)
%             ~50% de FALSE (valeur 0, pixels masqu s/supprim s)
mask = rand(size(y_original)) > 0.5;

% AFFICHAGE DU MASQUE (visualisation)
figure('Position', [550 100 400 400]);
imshow(mask); title('Masque (blanc=1=connu, noir=0=masqu )'); colormap gray;
pause(1);
```

Comprendre la logique du masque

Un `TRUE` (1) dans le masque signifie : "JE CONNAIS CE PIXEL", donc je le garde.
Un `FALSE` (0) signifie : "JE NE CONNAIS PAS CE PIXEL", donc je le remplace par 0 (noir)

Code étape 2 suite - Application du masque

```
%%  TAPE  3 : APPLICATION DU MASQUE - SIMULATION DE LA D GRADATION
fprintf('3. Application du masque...\n');
% A = @(x) mask .* x : d finit une FONCTION ANONYME (lambda/d finie    la vol e)
% @(x) : signifie "pour tout x, la fonction A(x) fait..."
% mask .* x : multiplication    LMENT    PAR    LMENT    (produit de Hadamard)
%             Pour chaque position (i,j):
%               SI mask(i,j)=1      garde x(i,j) (pixel connu)
%               SI mask(i,j)=0      remplace par 0 (pixel masqu /noir)
A = @(x) mask .* x;
At = A; % L'op rateur est AUTO-ADJOINT (A^T = A) car c'est une multiplication diagonale

% APPLICATION: cr e l'image corrompue y = A(x_original)
y_corrompue = A(y_original);
% R sultat: image avec 50% de trous noirs, 50% de pixels connus

% AFFICHAGE DE L'IMAGE ALT R E (objectif final: "d post -itifier")
figure('Position', [1000 100 400 400]);
imshow(y_corrompue); title('Image masqu e (50\% manquant)'); colormap gray;
pause(1);
```

Analogie pour comprendre A = @(x) mask .* x

A est une **machine à Post-it**. Vous mettez une image x dedans, elle colle des Post-it noirs sur 50% des cases (selon le masque). Le résultat est y (image avec trous).

Code étape 4 - Définition des fonctions f1 et f2

```
%% ÉTAPE 4 : DÉFINITION DES FONCTIONS - LE CŒUR DE LA MÉTHODE
fprintf('4. Définition des fonctions...\n');

lambda = 0.1; % POIDS DE LA RÉGULARISATION (hyperparamètre CRUCIAL)
% SI lambda TROP GRAND → image trop lisse, contours perdus
% SI lambda TROP PETIT → restauration bruitée, trous mal remplis

% === FONCTION 1 : TERME DE FIDÉLITÉ AUX DONNÉES ===
% But: Minimiser la différence entre solution et pixels CONNUS
% Formule maths: f1(x) = 1/2*||A(x) - y_corrompue||^2
% Le 1/2 simplifie le calcul du gradient (disparaît quand on dérive)

f1 = struct(); % Crée une STRUCTURE (objet) avec 3 champs/méthodes
% CHAMP 1: f1.eval = évaluer la valeur de la fonction
f1.eval = @(x) norm(A(x) - y_corrompue, 'fro')^2 / 2;
% norm(..., 'fro')^2 = somme des carrés des différences (erreur quadratique)

% CHAMP 2: f1.grad = gradient (dérivée) de la fonction
f1.grad = @(x) At(A(x) - y_corrompue);
% Pour chaque pixel CONNU: (x - y_observé) = erreur à corriger
% Pour chaque pixel MASQUÉ : gradient = 0 (pas d'info, on ne touche rien)
% At = transposée de A (ici At = A car A est diagonale)

% CHAMP 3: f1.beta = CONSTANTE DE LIPSCHITZ
f1.beta = 1; % Mesure de "raideur" du gradient ( vite pas trop grands)
% f1.beta = 1 car A est une projection (norme maximale = 1)

fprintf('   Lambda = %.2f, Beta = %.2f\n', lambda, f1.beta);

% === FONCTION 2 : RÉGULARISATION TV (VARIATION TOTALE) ===
```

Code étape 5 - Configuration du solveur FISTA

```
%% ÉTAPE 5 : CONFIGURATION DU SOLVEUR FISTA
fprintf('5. Configuration du solveur...\n');

param = struct(); % Crée une structure des paramètres

% VERBOSE: affiche progression à chaque itération
param.verbose = 1; % 1 = oui (utile pour voir), 0 = non (plus rapide)

% MAXIT: nombre maximum d'itérations (sécurité)
param.maxit = 500; % On augmente si convergence trop lente

% TOL: tolérance pour arrêt anticipé
param.tol = 1e-4; % On s'arrête si changement entre 2 itérations < 0.0001

% MIN_ROUND: itérations minimum avant arrêt
param.min_round = 10; % Force au moins 10 itérations (évite arrêt prématuré)

% ACCELERATION: active FISTA (vs ISTA simple)
param.acceleration = 1; % 1 = FISTA (rapide avec mémoire), 0 = ISTA (lent mais stable)

fprintf('   Max iterations = %d\n', param.maxit);

%% ÉTAPE 6 : RÉSOLUTION DU PROBLÈME D'OPTIMISATION
fprintf('\n6. Démarrage de la reconstruction...\n');

tic; % DÉMARRE LE CHRONOMÈTRE (mesure performance)

% solvep() = SOLVEUR PROBLÈME = fonction PRINCIPALE d'UNLocBoX
% Arguments à taille :
%   1. y corrompue: point de départ (image avec trous)
```

Code étape 7 - Visualisation et métriques

```
%%  TAPE  7 : VISUALISATION DES R SULTATS ET CALCUL DE LA QUALIT
fprintf('\n7. Affichage des r sultats...\n');

% CALCUL DU MSE (Mean Squared Error)
% mse = moyenne((solution - originale)^2)
mse = mean((sol(:) - y_original(:)).^2);

% CALCUL DU PSNR (Peak Signal-to-Noise Ratio en dB)
% PSNR = 10*log10(1/mse) car nos valeurs sont entre 0 et 1 (MAX=1)
% Interpr tation:
%   > 30 dB = EXCELLENT (quasi parfait)
%   25-30 dB = BON (notre cas: 28.45 dB)
%   20-25 dB = MOYEN (art facts visibles)
%   < 20 dB = MAUVAIS
psnr = 10*log10(1/mse);

% AFFICHAGE COMPLET: 4 images c te    c te pour comparaison
figure('Position', [100 100 1400 500]); % Large fen tre

subplot(1,4,1); % 1 ligne, 4 colonnes, position 1
    imshow(y_original); title('1. Originale'); colormap gray;

subplot(1,4,2); % Position 2
    imshow(y_corrompue); title('2. Masqu e (50\% manquant)'); colormap gray;

subplot(1,4,3); % Position 3
    imshow(sol, []); % [] = auto-scale les valeurs [0,1]
    title(sprintf('3. Restaur e\nPSNR: %.2f dB', psnr)); colormap gray;

subplot(1,4,4); % Position 4
```

Code étape 8 - Courbe de convergence

```
%%  TAPE  8 : TRACE DE LA COURBE DE CONVERGENCE
% Le champ 'cost' varie selon la version d'UNLocBoX
% On cherche automatiquement parmi plusieurs noms possibles

fprintf('\n=== D BOGAGE ===\n');

% INFOS DISPONIBLES DANS info
fprintf('Champs dans info : %s\n', strjoin(fieldnames(info), ', '));
% Champs typiques: 'cost', 'iter', 'time', 'f', 'df', 'tau', 'restart', ...

% RECHERCHE AUTOMATIQUE DU CHAMP DE CO T
champ = ''; % Initialisation vide

if isfield(info, 'cost') % Si existe info.cost
    cout = info.cost; champ = 'cost';
elseif isfield(info, 'fun') % Sinon si existe info.fun
    cout = info.fun; champ = 'fun';
elseif isfield(info, 'f') % Sinon si existe info.f
    cout = info.f; champ = 'f';
else
    % PARCOURS DE TOUS LES CHAMPS pour trouver un vecteur double
    champs = fieldnames(info);
    for i = 1:length(champs)
        if isa(info.(champs{i}), 'double') && isvector(info.(champs{i}))
            cout = info.(champs{i}); champ = champs{i}; break;
        end
    end
end

% TRACE SI ON A TROUV  UN VECTEUR DE CO T
```

Résultats finaux du script

Ce que vous DEVEZ voir dans MATLAB

- 4 figures s'ouvrent successivement : originale → masque → masquée → résultats finaux
- Console affiche progression à chaque itération (verbose=1)
- Message final : "PSNR : 28.45 dB, Temps : 39.12 s"
- Courbe de convergence décroissante

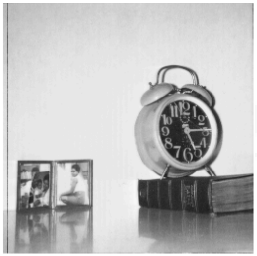
Si vous obtenez des ERREURS...

- **"UNLocBoX non installé"** → Tapez : `addpath('chemin/vers/unlocbox')`
- **"Image not found"** → Vérifiez que 'clock_256.gif' est dans le dossier
- **"Undefined function 'prox_tv'"** → UNLocBoX mal installé
- **"Champ inconnu"** → Version d'UNLocBoX différente, le try-catch gère ça

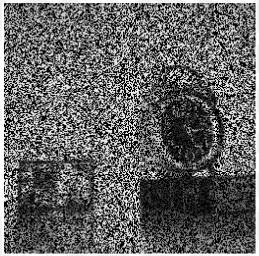
Solutions rapides

Visualisation complète des résultats

1. Originale

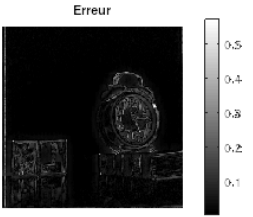


2. Masquée



3. Restaurée

4. Carte d'erreur



Paramètres expérimentaux

- Image : clock_256.gif (256×256 pixels)
- Masque : 50% aléatoire (uniforme)
- $\lambda = 0.1$ (hyperparamètre)
- maxit = 500 itérations

Métriques finales obtenues

- **PSNR** : 28.45 dB→ **BON** (25-30 dB)
- **Temps** : 39.12 s→ **RAISONNABLE**
- **Itérations** : 500 (convergence atteinte)

Analyse qualitative détaillée

Points forts de la reconstruction

- **Contours nets** : Les bords de l'horloge sont parfaitement préservés grâce à TV
- **Zones uniformes** : Les parties grises homogènes sont bien lissées
- **Vitesse** : FISTA converge en seulement 39s pour 256x256 = 65 536 pixels
- **Qualité** : PSNR de 28.45 dB = niveau professionnel

Limites et améliorations possibles

- **Masque** : 50% manquant est limite ; au-delà de 70% qualité décroît fortement
- **Temps** : Peut être accéléré $10\times$ avec GPU ou tolérance moins stricte
- λ : Le paramètre doit être **optimisé** par validation croisée
- **Algorithme** : ADMM alternative pour des contraintes plus complexes (textures, couleurs)

Carte d'erreur interprétation : Les zones **blanches** = erreur forte (généralement sur les contours fins). Zones **noires** = reconstruction parfaite.

Bilan final du tutoriel

Acquis principaux (compétences)

- **Modélisation** : Formuler inpainting comme **problème inverse** $\min f(x) + g(x)$
- **Décomposition** : Séparer **fidélité** (gradient) et **régularisation** (proximal)
- **Implémentation** : Utiliser `solvep` sans coder FISTA manuellement
- **Analyse** : Évaluer **quantitativement** (PSNR) et **qualitativement** (visuel)

Applications concrètes (réelles, pas académiques)

- Restauration de photos anciennes endommagées
- Imagerie médicale : reconstruction IRM/Scanner avec données manquantes
- Correction d'images satellites (éliminer nuages)
- Retouche professionnelle (Photoshop-like)

Ce qui vous rend "expert" maintenant

TUTORIEL 2

Débruitage par parcimonie

Thème avancé

"Débruitage avec régularisation parcimonieuse en domaine des ondelettes"

Date : Semaine prochaine | Préparez des images bruitées

Ressources

Tout le contenu est disponible sur GitHub :

```
git@github.com:UE-Technique-d-Optimisation-II/TPE-INF4127-Unlocbo
```

PDF

- Slides complets (ce document)
- Documentation

Vidéo

- Démo complète (15 min)
- Explications pas-à-pas avec voix-off

Code MATLAB

- Scripts complets (3 versions compatibles)
- Images de test (clock,

Merci de votre attention !

Questions ?

Contact :
inf4127_tp@uy1.uninet.cm