Structures de données pour le recalage d'images

D. Legland

23 mars 2011

Description des classes définies et implémentées pour tester les algorithmes de recalage d'images en 2D et 3D sous Matlab. La bibliothèque développée propose des classes pour représenter les images, différents modèles de transformation, l'interpolation et le ré-échantillonnage des images, des métriques pour comparer les images, et plusieurs algorithmes d'optimisation.

Table des matières

1 Introduction

Inspiration principale : ITK

idée générale : encapsuler les information, et manipuler les objets en se souciant le moins possible de leur implémentation.

Certaines classes sont indépendantes de la dimension de travail. Quand ce n'est pas le cas, la dimension est spécifiée en suffixe du nom.

2 Classe Image

Les différents outils reposent sur une classe « Image », qui a pour but de pouvoir gérer :
- différentes dimensions spatiales d'images : 2D ou 3D, voire 1D (profils) ou 0D (une valeur scalaire)
- différents types de données : des images en niveau de gris, mais aussi des images couleur, spectrales, labels, des images de gradients...
- des images en fonction du temps
- la calibration spatiale des images, et la conservation de cette calibration quand on extrait une coupe 2D d'une image 3D par exemple
- des informations de calibration de l'affichage (pour les images en niveaux de gris)

Cela fait donc 5 dimensions à gérer, dont 3 dimensions spatiales.

La classe est utilisable pour des cas classiques (images en niveaux de gris en 2D et 3D, images de gradient). Il reste cependant encore beaucoup de méthodes non implémentées ou incomplètes, elle reste donc à utiliser avec prudence (2010.11.24).

2.1 Création

Pour la création des images, l'idée est de passer par des constructeurs statiques. Cela permettra à terme de fournir des classes images qui stockent les données de manière optimisée.

On peut créer un nouvel objet grâce à la méthode **create** :

```
1 data = imread('cameraman.tif'); % charge un tableau de donnees
2 img = Image.create(data);       % convertit en objet image
```

La méthode statique **read** permet de charger les images de manière générique :

```
1 img = Image.read('cameraman.tif');
2 rgb = Image.read('peppers.png');
```

2.2 Visualisation

Pour afficher une image 2D, on a une méthode show :

```
1 img = Image.read('cameraman.tif');
2 img.show();
```

Pour représenter une image 3D, on a deux méthodes :

showOrthoSlices affiche 3 coupes orthogonales dans un même repère

showOrthoPlanes affiche 3 coupes orthogonales chacune dans un axe

3 Interpolation et re-échantillonnage

On a aussi plusieurs classes pour gérer le ré-échantillonnage et l'interpolation des images après transformation. En particulier, on dispose d'une classe abstraite ImageFunction, qui permet de manipuler une image interpolée ou une image après transformation de manière transparente.

3.1 Interfaces et classes abstraites

3.1.1 ImageFunction

Interface générale pour les classes d'interpolation. Elle déclare une méthode d'évaluation de la fonction :

evaluate accepte une position (2D ou 3D) en entrée et renvoie une valeur scalaire.

La classe implémente aussi deux méthodes utilitaires (en accès protégé) :

mergeCoordinates pour transformer plusieurs tableaux de coordonnées en un seul tableau de points et un paramètre de taille de tablean

splitCoordinates qui fait l'opération inverse de la précédente : le tableau de coordonnées est transformé en plusieurs tableaux contenant les coordonnées x, y...

3.1.2 ImageInterpolator

Classe abstraite, qui encapsule une image, et sert de base aux autres implémentations d'interpolations.

La classe ImageInterpolator fournit aussi un constructeur statique, « create », qui prend en argument une image et une chaîne de caractères correspondant au type d'interpolation désiré, et qui renvoie un objet d'une classe dérivée. Exemple :

```
1 img = Image2D.read('cameraman.tif');
2 interpolator = ImageInterpolator.create(img, 'linear');
3 class(interpolator)
4 ans =
5       LinearInterpolator2D
```

3.1.3 ImageInterpolator2D, ImageInterpolator3D

Interfaces permettant de typer les classes dérivées.

3.2 Implémentations

Je suis parti au début sur une implémentation par couple type d'interpolation - dimension. Avoir des interpolateurs génériques selon la dimension requiert plus de travail. Seule l'interpolation au plus proche voisins est générique pour le moment.

3.2.1 NearestNeighborInterpolator

Interpolation au plus proche voisin. Marche en 2D et en 3D, pour des images scalaires et des images de vecteurs (chaque composante vectorielle est interpolée individuellement, par une approche marginale).

3.2.2 LinearInterpolator2D

Interpolation linéaire pour des images 2D, marche sur des images scalaires.

3.2.3 LinearInterpolator3D

Interpolation linéaire pour des images 3D, marche sur des images scalaires.

3.2.4 NearestNeighborGradientEvaluator

Il s'agit d'une classe d'évaluation du gradient d'une image, qui stocke l'image de base, et calcule une interpolation du gradient à chaque appel de la fonction evaluate. Cette manière de faire permet de travailler sur des images gradient de la même manière que si on stockait le gradient interpolé, mais avec un coût mémoire moindre. Par contre, le temps d'éxécution est plus important.

3.3 Classes utilitaires

Ces classes sont destinées à faciliter la création et la manipulation des images transformées ou ré-échantillonnées.

3.3.1 ImageResampler

Cette classe permet de créer une nouvelle image à partir :

1. d'une base spatiale (origine + espacement + taille),
2. et d'un objet de classe « ImageFunction »

3.3.2 BackwardTransformedImage

Cette classe qui implémente l'interface « ImageFunction » encapsule une image et une transformation. Quand on évalue la valeur en un point, les coordonnées du point après transformation sont calculées, et utilisées pour évaluer la valeur dans l'image stockée.

3.4 Implémentation obsolètes

Ces implémentations étaient utilisées dans des versions précédentes, mais des classes plus génériques ont été créées depuis.

3.4.1 NearestNeighborInterpolator2D

Interpolation au plus proche voisin pour des images scalaires 2D. Obsolète, remplacée par NearestNeighborInterpolator.

3.4.2 NearestNeighborInterpolator3D

Interpolation au plus proche voisin pour des images scalaires 3D. Obsolète, remplacée par NearestNeighborInterpolator.

4 Comparaison des images

Pour le recalage on a besoin de définir des métriques entre des images, ou plus exactement entre des images interpolées. Comme on veut calculer des gradients par rapport à une transformation paramétrique, on prévoit aussi quelques fonctions adéquates.

Les métriques sont définie par rapport à un ensemble de points de test. Un ensemble de points de tests classique est constitué par l'ensemble des positions des pixels d'une des deux images. Comme on essaie de travailler sur des images interpolées, et donc qui ne définissent pas forcément leur base spatiale, l'ensemble de points tests est spécifié séparément. De plus, cela permet de ne donner qu'un sous-ensemble réduit des points (de l'ordre de quelques milliers), et ainsi d'accélérer les temps d'éxécution.

4.1 Classes abstraites

4.1.1 ImageToImageMetric

Cette classe stocke trois champs :

img1 une instance de ImageFunction qui représente traditionnellement l'image fixe

img2 une instance de ImageFunction qui représente traditionnellement l'image mobile

points un ensemble de points de test

Elle déclare la méthode suivante, qui sera implémentée dans les classes dérivées :

computeValue calcule la valeur de la métrique

Certaines classes devraient aussi implémenter une méthode pour fournir le vecteur jacobien par rapport aux paramètres d'une transformation. Pour le moment (2010.08.24) il n'y a pas d'interface pour le spécifier.

computeValueAndGradient cette méthode prend en entrée une transformation paramétrée ainsi qu'une image gradient.

4.2 Implémentation

Pour le moment une seule implémentation, basée sur la moyenne (ou la somme) des carrés des différences, mais avec des noms différents...

4.2.1 MeanSquaredDifferencesMetric

Correspond à une discrétisation de l'intégrale

$$MSD = \int (I_1(x) - I_2(x))^2 \, dx$$

En pratique, on passe par une moyenne des carrés des différences.

Afin de pouvoir calculer des dérivées, elle déclare les champs suivants :

transform le modèle de transformation utilisé, qui doit implémenter la méthode getJacobian.

gradientImage une image vectorielle représentant typiquement le gradient de l'image mobile

4.2.1.1 Méthodes implémentées

computeValue Cette méthode itère sur tous les points tests, évalue la valeur correspondante dans chacune des deux images, et calcule la moyenne des carrés des différences. La moyenne est calculée pour les points qui sont dans les deux images.

computeValueAndGradient cette méthode prend en entrée une transformation paramétrée ainsi que les composantes d'une image gradient, et renvoie la valeur du vecteur gradient calculée en moyennant, pour les points de test, le produit du jacobien de la transformation par l'image gradient. L'image gradient est interpolée selon les plus proches voisins.

4.2.2 SumOfSquaredDifferencesMetric

Il s'agit principalement d'une ré-implémentation de la précédente. Elle renvoie une somme au lieu d'une moyenne.

4.3 Autres fonctions de coût

On a d'autres fonctions de coût, principalement les fonctions de régularisation des transformations.

4.3.1 MotionRegularizationFunction

Calcule une valeur qui augmente au fur et à mesure que la transformation s'éloigne de l'identité. Le coût se décompose en un coût de translation (norme du vecteur) et un coût de rotation (norme de l'angle de rotation, en degrés).

5 Transformations géométriques

En parallèle des classes image, on définit aussi une hiérarchie de classes pour les transformations géométriques. On a aussi une classe spéciale qui représente une transformation paramétrée. Certaines opérations sont définies en fonctions des spécialisations.

La classe de base est la classe Transform. La classe ParametricTransform ajoute la manipulation des paramètres à optimiser. On trouve aussi des classes, comme AffineTransform, dont le but est de limiter la ré-écriture des transformations spécialisées.

5.1 Interfaces et classes abstraites

5.1.1 Transform

Il s'agit de la classe la plus générique. Elle définit principalement la méthode transformPoint, ainsi que quelques méthodes utilitaires.

transformPoint transforme un point en un autre

transformVector transforme un vecteur en un autre (prend en entrée le vecteur en entrée et la position du vecteur)

getJacobian renvoie la matrice jacobienne (matrice des dérivées premières en fonction des coordonnées) pour une position donnée

Toutes ces méthodes sont abstraites, et nécessitent d'être implémentées dans les classes dérivées. Il est possible aussi de composer les transformations, via la méthode compose.

compose renvoie un nouvelle transformation, résultat de l'application de la transformation passée en paramètre suivie de cette transformation

5.1.2 ParametricTransform

Cette classe (abstraite) ajoute la gestion des paramètres. Elle définit un champs « params », sous la forme d'un vecteur ligne, qui peut être modifié via les méthodes appropriées.

Pour distinguer les transformations paramétrées des transformations non paramétrées, j'ajoute parfois le suffixe « Model » au nom de la classe. Exemple : « TranslationModel ».

Méthodes implémentées

getParameters renvoie le vecteur de paramètres

setParameters modifie le vecteur de paramètres

getParameterLength renvoie la taille du vecteur de paramètres

On a aussi quelques méthodes pour faciliter l'interprétation

getParameterNames renvoie un tableau de chaînes contenant le noms de chaque paramètre

getParameterName renvoie le nom du i-ème paramètre

Enfin, on a aussi une méthode pour calculer les dérivées mais en fonction des paramètres :

getParametricJacobian calcule la matrice jacobienne, qui a autant de lignes que le nombre de dimensions spatiales, et autant de colonnes que le nombre de paramètres

5.1.3 AffineTransform

Cette classe sert de base abstraite aux autres transformations affines. Elle implémente plusieurs méthodes qui se basent sur la matrice de transformation associée. Notons que la matrice de transformation n'est pas définie dans cette classe, mais dans les classes dérivées.

Méthodes abstraites

getAffineMatrix renvoie la matrice affine associées à cette transformation

Il s'agit de la seule méthode abstraite de cette classe, et donc la seule qui nécessite une implémentation par les classes dérivées.

Méthodes implémentées

Les méthodes suivantes définies par les interfaces sont implémentées par la classe AffineTransform :

transformPoint transforme un point en un autre

transformVector transforme un vecteur en un autre

getJacobian renvoie la matrice jacobienne (des dérivées premières en fonction des coordonnées) pour une position donnée

Note : pour une transformation affine, la matrice jacobienne est obtenue en isolant la partie linéaire de la matrice de transformation.

compose renvoie un nouvelle transformation, résultat de l'application de la transformation passée en paramètre suivie de cette transformation. Essaie de renvoyer une transformation affine si possible.

On a aussi une méthode spécifique aux transformations affines

getInverse renvoie la transformation affine inverse

5.2 Modèles de transformation

Les classes décrites ici permettent de représenter un modèle de transformation dont on peut optimiser les paramètres. Ces transformations héritent donc à la fois de Transform et de ParametricTransform.

Conventions (par toujours) utilisées :
- on utilise le suffixe « Model » ou « TransformModel »
- on utilise le préfixe « Centered » pour préciser que l'on peut modifier le centre de la transformation.
- on ajoute le suffixe 2D ou 3D quand la transformation n'est pas générique.

5.2.1 TranslationModel

Une translation en dimension arbitaire, définie par un vecteur de translation.

5.2.2 CenteredMotionTransform2D

Il s'agit d'une transformation définie par une rotation autour d'un centre donné (par défaut l'origine), suivie d'une translation. On a donc 3 paramètres. L'angle de rotation est stocké en degrés, et converti en radians pour les calculs. Le centre de la rotation est un paramètre non optimisable.

5.2.3 EulerTransformModel3D (CenteredEulerTransform3D)

Cette classe représente une transformation rigide obtenue en combinant trois rotations successives suivies d'une translation. On a donc 6 paramètres.

Il y a de multiples manières de choisir les angles d'Euler. Pour cette classe, les rotations sont choisies ainsi, dans cet ordre :

1. une rotation autour de l'angle O_x selon un angle φ
2. une rotation autour de l'angle O_y selon un angle θ
3. une rotation autour de l'angle O_z selon un angle ψ

Les angles de rotation sont stockés en degrés et convertis en radian pour les calculs. Le centre de la rotation est modifiable, mais ne fait pas partie des paramètres optimisables.

5.2.4 CenteredAffineTransformModel3D

Représente une transformation affine paramérable. Les paramètres sont les 12 coefficients de la matrice de transformation (d'abord les coefficients de la première ligne, puis la deuxième...).

5.2.5 CenteredQuadTransformModel3D

Modèle de transformation dans lequel les coordonnées du point transformé sont calculées à partir de fonctions quadratiques des coordonnées du point d'entrée.

$$\begin{aligned} x' &= p_1 + p_4x + p_7y + p_{10}z + p_{13}x^2 + p_{16}y^2 + p_{19}z^2 + p_{22}xy + p_{25}xz + p_{28}yz \\ y' &= p_2 + p_5x + p_8y + p_{11}z + p_{14}x^2 + p_{17}y^2 + p_{20}z^2 + p_{23}xy + p_{26}xz + p_{29}yz \\ z' &= p_3 + p_6x + p_9y + p_{12}z + p_{15}x^2 + p_{18}y^2 + p_{21}z^2 + p_{24}xy + p_{27}xz + p_{30}yz \end{aligned}$$

Le centre de la transformation est initialisé par défaut à (0,0,0).

5.3 Implémentations utilitaires

Les transformations présentées ici servent soit de base à l'implémentation de transformations utilisables directement, soit permettent de combiner les transformations entre elles.

5.3.1 MatrixAffineTransform

Implémentation classique d'une transformation affine, sous la forme d'une matrice. Utilisées pour des tests, ou pour implémenter des transformations classiques.

5.3.2 CenteredTransformAbstract

Squelette d'implémentation d'une transformation centrée autour d'un point. La classe définit un champ « center », ainsi qu'un accesseur et un modifieur.

5.3.3 ComposedTransform

Cette classe stocke un tableau de transformations. Lors de l'appel à une des méthodes de calcul (par exemple « transformPoint »), elle appelle la méthode correspondante pour chaque transformation stockée, et combine les résultats.

Aucune hypothèse n'est faite sur les transformation stockées, elles doivent juste hériter de « Transform », et être définies pour les mêmes dimensions.

5.3.4 ComposedTransformModel

La même que précédemment, mais elle hérite en plus de ParametricTransform. Les méthodes de manipulation des paramètres ont été surchargées pour tavailler sur les paramètres de la dernière transformation.

5.3.5 TransformGradientResampler

Cette classe est initialisée avec une image. Quand on appelle la méthode « resample » sur une transformation, on obtient une nouvelle image en niveaux de gris de la même taille que l'image d'initialisation, et qui contient pour chaque pixel ou voxel la valeur locale du déterminant du jacobien de la transformation.

6 Optimisation

La boîte à outil optimisation de Matlab fournit plusieurs algorithmes d'optimisation. Cependant, on peut avoir besoin d'algorithmes non implémentés dans la BAO, et l'utilisation des fonctions n'est pas toujours pratique (besoin de beaucoup de code et de pointeurs de fonctions). On a donc créé des classes dédiées à l'optimisation, qui permettent une meilleure modularité.

Note : les noms des classes, champs et méthodes s'inspirent principalement d'ITK et de la BAO Optimisation de Matlab.

6.1 Interfaces pour l'optimisation

On définit plusieurs classes d'interfaces, qui spécifient les méthodes à implémenter. On a principalement une interface pour les algorithmes d'optimisation proprement dits, et des interfaces pour manipuler et combiner les fonctions de coût, qui peuvent être paramétriques ou non.

6.1.1 Optimizer

La classe Optimizer est une classe abstraite, parente de tous les classes implémentant un algorithme d'optimisation.

Champs de classe

La classe Optimizer contient deux champs principaux :

costFunction représente la fonction de coût à optimiser, sous la forme d'un pointeur de fonction dont le premier argument est le vecteur de paramètres

params réprésente l'état courant des paramètres

D'autres champs sont présents pour faciliter la personnalisation :

outputFunction un pointeur de fonction qui est censé être appelé à chaque itération

displayMode une chaîne de caractères qui spécifie la quantité d'information à afficher. Peut être une valeur à choisir parmi « off » (n'affiche rien - dangereux) ; « iter » (affiche le résultat de chaque itération), « notify » (affiche le résultat si l'algorithme ne converge pas, valeur par défaut), ou « final » (affiche juste le résultat final).

Méthode abstraite

startOptimization() démarre l'algorithme d'optimisation jusqu'à ce qu'une condition d'arrêt soit atteinte. La gestion des conditions d'arrêt (nombre d'itérations, déplacement inférieur à un seuil...) est laissée aux classes dérivées.

6.1.2 Optimisation via un pointeur de fonction

La fonction à optimiser est passée sous la forme d'un pointeur de fonction. La fonction doit accepter en premier argument le vecteur de paramètres, et renvoyer la valeur de la fonction de coût en premier argument de sortie.

Pour les méthodes d'optimisation utilisant le gradient ou la matrice Hessienne, Matlab suppose que la fonction renvoie ces paramètres respectivement en deuxième et troisième arguments de sortie.

Dans un cadre de recalage, on a deux actions à accomplir :

1. mettre à jour le vecteur de paramètres de l'objet (une transformation, ou un groupe de transformations) à optimiser
2. calculer la valeur d'une métrique qui dépend de l'objet à optimiser.

On peut utiliser un des pointeurs de fonction suivants :

evaluateParametricMetric(params, transfo, metric) cette fonction met à jour les paramètres de la transformation, et appelle la méthode « computeValue » de la métrique.

evaluateParametricMetricValueAndGradient la même, mais on évalue aussi le gradient. Les paramètres additionnels (typiquement, l'image de gradient) sont transmis à la méthode « computeValue » de l'objet métrique.

6.1.3 CostFunction

Il s'agit d'une classe abstraite (interface) qui devrait servir de base à la totalité des fonctions de coût. La méthode évaluate accepte en entrée un vecteur de paramètres. Selon le nombre d'arguments en sortie, cette méthode renvoie une valeur scalaire, le gradient, ainsi que la matrice Hessienne.

(abstract) evaluate(params) évalue la fonction de coût en fonction du vecteur de paramètres.

On l'utilise via un optimiseur de la manière suivante :

```
1 evaluator = ... % cree une instance d'une sous-classe de CostFunction.
2 % associe la fonction de cout a l'optimiseur
3 fun = @evaluator.evaluate;
4 optimizer.setCostFunction(fun);
```

6.2 Objets « optimisables »

Le passage par pointeur de fonction n'est pas des plus pratiques, surtout quand on travaille avec des fonctions de coût qui combinent plusieurs éléments (métrique entre les images, termes de régularisation...). L'idée de cette section est de présenter quelques classes qui simplifient la création de la fonction de coût.

6.2.1 MetricEvaluator

Il s'agit d'un objet qui encapsule (1) la transformation paramétrique à optimiser, et (2) une métrique qui dépend de la transformation.

```
% Cree un optimiseur (ici par descente de gradient)
optimizer = GradientDescentOptimizer();

% Cree un evaluateur, a partir de la transfo et de la metrique
evaluator = MetricEvaluator(transfo2, metric);

% associe la fonction de cout a l'optimiseur
fun = @evaluator.evaluate;
optimizer.setCostFunction(fun);
```

6.2.2 ParametricFunction

Classe abstraite qui déclare deux méthodes, et implémente la méthode evaluate.

(abstract) setParameters(params) modifie les paramètres internes

(abstract) computeValue() calcule la valeur avec les paramètres courants

evaluate(params) appelle les deux fonctions précédentes, et renvoie la valeur courante, ainsi qu'éventuellement le gradient.

6.2.3 ParametricObject

Classe de base (interface) pour les objets paramétriques. Les classes dérivées sont les transformations paramétriques, ainsi que les agglomérations de transfos. Dans ce dernier cas, la modification d'un paramètre est dispatché à la transformation concernée.

6.3 Monitoring de l'optimisation

On dispose aussi d'un système de gestion d'évènements, qui permet d'actualiser l'affichage graphique au cours de la procédure d'optimisation. Quelques classes utilitaires pour afficher l'évolution de l'algorithme ont aussi été développées.

6.3.1 OptimizationListener

Il s'agit d'une classe abstraite, dont le but est de fournir une implémentation minimale pour les méthodes suivantes :

optimizationStarted appelée quand la procédure d'optimisation démarre

optimizationIterated appelée à la fin de chaque itération de la procédure d'optimisation

optimizationTerminated appelée lorsque la procédure d'optimisation est terminée

Les classes dérivées ont ainsi juste à re-implémenter l'une ou la totalité de ces méthodes pour avoir un comportement spécifique. On ajoute un écouteur à un optimiseur de la façon suivante :

```
opt = NelderMeadSimplexOptimizer();
listener = ...
opt.addOptimizationListener(listener);
```

6.3.2 ParametersEvolutionDisplay

Affiche l'évolution de plusieurs paramètres dans une fenêtre donnée en fonction du nombre d'itérations.

6.3.3 OptimizedValueListener

Affiche l'évolution de la valeur actuelle de la fonction d'optimisation en fonction du nombre d'itérations. Affiche dans un objet axes.

6.3.4 ParametricFunctionEvolutionDisplay

Affiche l'évolution de la valeur courante d'une fonction paramétrique quelconque en fonction du nombre d'itérations. Affiche dans un objet axes.

7 Algorithmes d'optimisation

Plusieurs algorithmes d'optimisation ont été implémentés en se basant sur l'interface « Optimizer ». Il s'agit soit de réécritures complètes, soit d'encapsulation de fonctions de la boîte à outils optimization de Matlab.

7.1 Optimisation basée sur la valeur de la fonction

Les algorithmes de cette catégorie ne nécessitent de connaître que la fonction de coût. Leur vitesse de convergence est en général faible. La classe la plus pratique pour le moment est GaussianLinearSearchOptimizer.

7.1.1 MultiLinearSearchOptimizer

Cet algorithme recherche le minimum d'une fonction en utilisant plusieurs recherches linéaires. Implémenté en utilisant la recherche de Brent. On peut spécifier l'ensemble des directions de recherche (par défaut : recherche dans la direction de chaque paramètre individuel, et dans les directions de couples de paramètres).

7.1.2 BoundedMultiLinearOptimizer

Algorithme basique « maison ». Le principe est similaire à la recherche linéaire multiple, mais selon un principe différent. On définit (1) les bornes de variations pour chaque paramètre et (2) le nombre de points de test entre ces bornes. On itère ensuite en générant des valeurs différentes pour un paramètre à la fois, et en gardant la valeur qui minimise la fonction de coût.

Avantage simple à mettre en oeuvre

Inconvénients assez lente, et ne permet pas de réduire la précision de manière incrémentale

7.1.3 GaussianLinearSearchOptimizer

Un autre algorithme « maison » basé sur la recherche linéaire. Cette fois ci, les valeurs de rechrche pour chaque paramètre sont générées en utilisant une distribution gaussienne centrée sur la dernière meilleure valeur trouvée, et avec une variance que l'on peut préciser.

Avantages assez rapide, et se prête bien à une recherche multi-précision, par exemple en divisant la variance de chaque paramètre par un facteur constant au bout d'un nombre suffisant d'itérations.

Inconvénients nécessite encore beaucoup d'évaluation de la fonction de coût, et risque de minima non trouvés si on cherche paramètre après paramètre

7.1.4 NelderMeadSimplexOptimizer

Implémentation de l'algorithme du simplexe basée sur le livre Numerical Recipes (3ème édition). Permet de spécifier le delta initial dans chaque direction.

Rappel : on utilise un tableau de taille $(n+1)*n$ pour stocker les coordonnées des sommets du simplexe. La méthode est donc plus appropriée pour des problèmes présentant un petit nombre de paramètres.

7.1.5 MatlabSimplexOptimizer

Cet algorithme recherche le minimum d'une fonction (supposée convexe) par la méthode du simplexe de Nelder-Mead. En pratique, il s'agit d'une encapsulation de la fonction « fminsearch » de Matlab.

Un défaut de la fonction fminsearch est qu'on ne peut pas spécifier la taille initiale du simplexe, ce qui fait que la recherche se fait dans un domaine trop étroit. A l'usage, on se retrouve assez vite dans des minima locaux. Il vaut donc mieux lui préférer la classe NelderMeadSimplexOptimizer, qui permet de spécifier la taille, voire la position du simplexe initial.

7.2 Optimisation utilisant le gradient

Plusieurs algorithmes utilisent le gradient des fonctions de coût pour accélerer la recherche de l'optimum. Pour le moment, seul l'algorithme de descente de gradient est implémenté, mais Matlab en propose d'autres.

7.2.1 GradientDescentOptimizer

L'idée de cet algorithme est de se déplacer à chaque itération dans la direction inverse du gradient. La gradient impose la direction, mais le pas est fixé par l'algorithme, en général une fonction décroissante du nombre d'itérations.

Champs à renseigner :

nIter le nombre max d'itération

step0 l'amplitude du pas de déplacement à la première itération

tau la taux de décroissance du pas de déplacement

On peut ensuite avoir des variantes de la descente de gradient, par exemple en utilisant une descente stochastique de gradient (on retire les points d'évaluation à chaque itération).

8 Perspectives

La bibliothèque est encore largement en développement, mais est déjà fonctionnelle.
Parmi les points à développer ultérieurement :
- la création de paquetages, afin d'organiser un peu mieux les classes
- la gestion des images couleurs
- gérer l'information mutuelle sous forme de classe