OOLIP : bibliothèque de classes pour le recalage d'images sous Matlab

D. Legland

3 novembre 2010

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 Structure de donnée Image

Pour le moment plusieurs classes image, organisées en hiérarchie :

Image interface pour les images, qui définit les méthodes abstraites et fournit quelques utilitaires pour gérer les informations de calibration et les meta-données

ScalarImage interface pour les images dont les pixels sont des scalaires, par opposition aux images vectorielles (images spectrales ou couleur)

Image2D implémentation pour les images scalaires 2D

Image3D implémentation pour les images scalaires 3D

VectorImage2D implémentation pour les images vectorielles 2D

On trouve quelques méthodes statiques pour créer les images en utilisant la classe appropriée :

```
1 img = Image2D.read('cameraman.tif');
2 img.show();
```

À terme, il serait souhaitable d'avoir une classe unique. Une adaptation est en cours (2010.08.24).

3 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.

3.1 Interfaces et classes abstraites

3.1.1 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

3.1.2 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 colonne que le nombre de paramètres

3.1.3 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

3.2 Modèles de transformation

Les classes décrites ici permettent de représenter un modèle de transformation dont on peur optimiser les paramètres. Ces transformations héritent donc à la fois de Transform (et même de AffineTransform pour le moment) et de ParametricTransform.

3.2.1 TranslationModel

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

3.2.2 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.

3.2.3 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 degré et convertis en radian pour les calculs. Le centre de la rotation est modifiable, mais ne fait pas partie des paramètres optimisables.

3.3 Implémentations utilitaires

3.3.1 MatrixAffineTransform

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

3.3.2 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.

4 Interpolation et re-échantillonnage

On a aussi plusieurs classes pour gérer le ré-échantillonnage et l'interpolation des images après transformation.

4.1 Interfaces et classes abstraites

ImageFunction interface générale pour les classes d'interpolation. Elle définit une méthode accepte une position (2D ou 3D) en entrée et renvoie une valeur scalaire

ImageInterpolator interface d'interpolation, qui se base sur une image.

ImageInterpolator2D, ImageInterpolator3D interfaces permettant de typer les classes dérivées

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 inter = ImageInterpolator.create(img, 'linear');
3 class(inter)
4 ans =
5       LinearInterpolator2D
```

4.2 Implémentations

Pour le moment on a une implémentation par couple type d'interpolation - dimension. Il n'est pas forcément plus efficace d'avoir des interpolateurs génériques.

NearestNeighborInterpolator2D

NearestNeighborInterpolator3D

LinearInterpolator2D

LinearInterpolator3D

4.3 Classes utilitaires

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

4.3.1 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 »

4.3.2 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.

5 Comparaison des images

Pour le recalage on a besoin de définir des métriques entre des images, ou plus exactement entre des images interpolées.

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.

5.1 Classes abstraites

5.1.1 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.

5.2 Implémentation

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

5.2.1 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

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.

5.2.2 SumOfSquaredDifferencesMetric

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

6 Optimisation de la fonction de coût

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 dédiée 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

6.1.1 Optimizer

La classe Optimizer définit un algorithme d'optimisation. Elle contient deux champs principaux :

costFunction qui représente la fonction de coût à optimiser

params qui 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 à chaque étape

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'itération, déplacement inférieur à un seuil...) est laissée aux classes dérivées.

6.1.2 Fonction à optimiser

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 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 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.

6.2.1 NelderMeadSimplexOptimizer

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.

A l'usage, il semble que l'on se retrouve assez vite dans des minima locaux.

6.2.2 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).

6.3 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.

6.3.1 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 focntion décroissante du nombre d'itérations.

Champs à renseigner :

nIter le nombre max d'itération

step0 l'amplitude du pas de déplacement

tau la taux de décroissance du pas de déplacement

6.4 Monitoring de l'optimization

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.

6.4.1 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 listener à un optimiseur de la façon suivante :

```
1 opt = NelderMeadSimplexOptimizer();
2 listener = ...
3 opt.addOptimizationListener(listener);
```

6.4.2 ParametersEvolutionDisplay

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

6.4.3 OptimizedValueListener

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

6.4.4 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 Perspectives

La bibliothèque encore largement en développement.
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