TD #2 – CETS8AH

20 février 2024
Erwan Kerrien

Seule la section 2 est notée. Vous rendrez ce TD, à faire seul, pour le 12 mars sous format électronique : envoyez par mail à `Erwan.Kerrien@inria.fr` les M-fichiers que vous aurez écrits, ainsi qu'un texte de réponse (format Word, OpenOffice ou Latex, ou autre si pas trop exotique). N'hésitez pas me poser des questions par mail.

La section 1 n'est pas à proprement parler un exercice. Elle a pour but de vous guider dans la découverte des possibilités de Matlab en termes de transformations d'images et d'ensembles de points. Ce faisant, je vous fais écrire un ensemble de fonctions qui vous serviront pour les exercices de la section 2. On ne s'intéresse ici, comme dans le cours, qu'au recalage rigide.

Le but général du TD est de vous faire expérimenter le recalage d'images à base de points détectés à travers l'implantation de la formule d'inversion de Procuste vue en cours, et en découvrant les limites d'utilisation de cette méthode : distribution des points dans l'image, comportement face au bruit de détection des points, ou en présence de points aberrants.

1 Transformations d'images sous Matlab

Cette première section a pout but d'introduire les fonctions disponibles sous Matlab pour définir une transformation et l'appliquer à des points (dits *d'intérêt*) ou à des images. À nouveau, dans ce qui suit, seules les lignes qui commencent par `>>` fonctionneront telles quelles quand vous les taperez. Les autres commandes sont données à titre d'exemple seulement, et il vous revient de les adapter pour les tester sur un exemple concret. Ces fonctions sont les suivantes (complétez ces descriptions brèves avec l'aide en ligne de Matlab) :

1. `maketform` : cette fonction permet de générer une transformation spatiale. Le premier paramètre de cette fonction est le type de transformation désiré : nous ne nous intéressons ici qu'aux transformations rigides, qui sont une sous-classe des transformations affines (option=`'affine'`). Comme c'est un besoin très répandu, les versions les plus récentes de Matlab ont défini des fonctions accessoires comme `affine2d`, ou `rigid2d`. Cette dernière est celle qui va nous intéresser dans ce TP.
2. `rigid2d` : définit une transformation affine à partir d'une matrice T de taille 3×3 ou bien d'une matrice de rotation `rot` et d'un vecteur de translation `trans`. Nous utiliserons cette dernière possibilité. Pour plus ample information, voir la page sur les transformations géométriques dans Matlab.
3. `transformPointsForward` : permet de transformer un ensemble de points en lui appliquant une transformation définie par un appel à `maketform` ou un de ses dérivés. La fonction `transformPointsInverse` permet d'appliquer la transformation inverse. Les N points sont définis par une matrice à N lignes mais on peut aussi passer les vecteurs d'abscisses (u) et d'ordonnées (v) séparément, et récupérer en sortie ces coordonnées de manière séparée également.
4. `imwarp` : permet de transformer une image en lui appliquant une transformation créée par `ridig2d` ou plus généralement par `maketform`.

Apèrs cette brève description, nous allons voir par un exemple, comment utiliser chacune de ces fonctions. Ce faisant, vous écrirez certaines fonctions utiles pour la suite du TP. **Cette première partie de TP n'est pas notée.** Ces descriptions ne sont pas exhaustives et je ne donne que les paramétrages les plus couramment utilisés pour ces fonctions.

1.1 Création de la transformation

Considérons une transformation rigide, c'est-à-dire la composition d'une rotation $\mathbf{R}$ (matrice carrée unitaire) et d'une translation T (vecteur). Matlab utilise la multiplication transposée (cf l' aide sur les transformations affines). Un point est donc vu comme un vecteur 1×2.

La formule générale qui permet d'exprimer la transformation d'un point P en un point Q est donc : $Q = P\mathbf{R} + T$ où $\mathbf{R}$ (matrice 2×2) et T (vecteur 2×1) sont respectivement les paramètres `rot` et `trans` que l'on souhaite passer à `rigid2d`.

Cette formule générale n'est pas aisée à manipuler car elle suppose que la rotation se fait par rapport à l'origine du repère. Dans le traitement d'image en général, cette origine correspond au coin supérieur gauche de l'image. Si on imagine donc qu'on veut réaliser une rotation pure d'une image, on se retrouvera avec une image qui tourne par rapport à son coin supérieur gauche, ce qui est peu intuitif. On préfère donc indiquer un centre de rotation C comme paramètre

supplémentaire, ce centre étant en pratique le centre de la zone d'intérêt dans l'image (ou de l'image entière si aucune zone d'intérêt n'a été définie). On réécrit donc la formule ci-dessus, mais avec pour origine C :

$$Q - C = (P - C)\mathbf{R} + T$$

ce qui nous permet d'obtenir la formule de transformation de P en Q :

$$Q = P\mathbf{R} + C(\mathbb{1} - \mathbf{R}) + T$$

où $\mathbb{1}$ est la matrice identité (commande `eye` sous Matlab).

On remarque que la formule conserve sa forme avec une partie rotation $\mathbf{R}$ et une partie translation $C(\mathbb{1} - \mathbf{R}) + T$.

Dans le cas d'une transformation rigide en 2D, la rotation est définie au moyen d'un seul angle a (en radians) par (pour ceux et celles que la position du signe pertuberait, n'oubliez pas qu'on doit transposer la matrice) :

$$\mathbf{R} = \begin{pmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{pmatrix}$$

Le code suivant implante le calcul d'une transformation rigide en 2D. Il suppose que l'angle est donné en degrés (d'où la petite ligne de transformation en radians). Écrivez cette fonction sous Matlab car vous en aurez besoin plus tard.

```
function A=makeRigidTransform(a, C, T)
  % test d'erreur
  assert(length(C) == length(T), 'Input vectors must have the same dimension');

  % dimension de l'espace à considérer: doit être 2
  dim=length(C);
  assert(dim == 2, 'This function only works in 2D');

  % traduit l'angle en radians
  arad=a*pi/180;

  % met T sous forme de vecteur ligne (permet de le donner à la fois sous
  % forme de vecteur ligne et sous forme de vecteur colonne
  Tc=reshape(T,1,dim);

  % met C sous forme de vecteur ligne (permet de le donner à la fois sous
  % forme de vecteur ligne et sous forme de vecteur colonne
  Cc=reshape(C,1,dim);

  % calcule la matrice de transformation
  R=[cos(arad) sin(arad); -sin(arad) cos(arad)];
  Tl = Cc*(eye(dim)-R) + Tc;

  % génère la transformation à renvoyer
  A=rigid2d(R, Tl);
end
```

1.2 Transformation d'un ensemble de points

La fonction `transformPointsForward` permet de transformer un ensemble de points, par exemple des points d'intérêt détectés dans une image. Il y a plusieurs formes d'appel de cette fonction mais la principale est la suivante :

```
Q=transformPointsForward(A,P);
```

Cet appel considère la transformation `A`, obtenue par `ridig2d` ou un de ses dérivés comme `makeRigidTransform`, et l'applique à l'ensemble de points `P` pour obtenir l'ensemble de points `Q`. Les points en entrée et en sortie sont rangés dans les **lignes** de `P` et `Q`. Ainsi ces deux matrices ont le même nombre de lignes (= nombre de points à transformer) et le nombre de colonnes de `P` est égal à la dimension en entrée de la transformation `A` (2 dans le plan) alors que le nombre

de colonnes de `Q` est égale à la dimension en sortie de `A` (2 également dans le plan). On a ainsi la relation suivante : `Q(k,:) == A(P(k,:))` pour tout indice `k` de ligne.

La fonction `transformPointsInverse` fait la même chose hormis qu'elle applique l'inverse de la transformation `A` donnée en premier argument.

1.3 Transformation d'une image

La fonction `imwarp` permet de transformer une image. La forme minimale est `J=imwarp(I,A)` (attention à l'inversion de paramètres par rapport à `transformPointsForward` : ici c'est d'abord l'image à transformer **puis** la transformation !) où `A` est la transformation à appliquer à l'image `I` pour obtenir l'image `J`. Un troisime paramètre peut être donné sous forme de chaîne de caractères qui indique le type d'interpolation à utiliser. Par défaut, il s'agit de l'interpolation bilinéaire (c'est-à-dire linaire en 2D).

Cependant, cette forme est peu utilisée dans le cas où vous faites un recalage d'images. La raison tient au comportement par défaut de cette fonction. Pour le comprendre, reprenons le petit exemple donné en cours : la rotation de 15 degrés par rapport au centre. Entrez les commandes suivantes (après vous être placés dans le répertoire adéquat) :

```
>> I=imread('Lena1024.jpg');
>> A=makeRigidTransform(15,(size(I,1:2)-1)/2, [0 0]);
>> J=imwarp(I,A);
>> imshow(I);
>> figure, imshow(J);
```

La première commande charge une image à partir du fichier `Lena1024.jpg`. La deuxième génère la rotation d'angle 15 degrés et de centre, le centre de l'image, donné par `(size(I,1:2)-1)/2` [1]. La troisième commande applique la rotation à l'image `I` pour obtenir l'image `J`. Qu'y a-t-il de surprenant dans l'image résultat ?

D'une part la rotation est horaire alors qu'elle devrait être dans le sens trigonométrique. Ceci est dû au fait que l'axe des X dans l'image est selon les lignes et l'axe des Y selon les colonnes, avec pour origine le coin supérieur gauche : le repère est indirect, d'où l'inversion du sens de rotation. Ceci n'est pas très grave quand il s'agit de recalage (on ne fait que changer le signe de l'angle), mais il est toujours bon de le savoir quand on cherche interpréter les résultats.

D'autre part, L'image `J` est plus grande que l'image `I` (vous pouvez le vérifier avec la commande `size(J)`). Quand vous visualisez l'image, vous voyez en effet tous les coins transformés. Normal me direz-vous ? Oui et non. Considérons le coin supérieur gauche de coordonnées `[1 1]`. Il est transformé en le point de coordonnées `[61.7880 -33.3167]` (ce que vous pouvez vérifier par un appel à `transformPointsForward(A,[1 1])`). Vous remarquez que la deuxième coordonnée est négative : elle sort de l'espace pixel habituellement réservé à une image. Le comportement par défaut de `imwarp` est de stocker le résultat de tous les pixels de l'image originale (un peu comme avec l'option `full` des fonctions de filtrage). Comme les indices ne peuvent être négatifs, Matlab applique une translation à l'image en sortie. Cette translation peut être récupérée si on passe un deuxième argument de sortie à imwarp :

```
>> [J,RJ] = imwarp(I,A)
```

Ceci est gênant car les critères de similariét supposent que les images se correspondent pixel à pixel. De plus, vous ne pourrez pas superposer à votre image ainsi transformée `J` les points transformés `Q` à cause de la translation qui, elle, n'est pas appliquée à `Q`. Il faut donc soit prendre en compte la translation dans le calcul de la transformation, soit éviter que cette translation soit appliquée, quitte à perdre quelques pixels. C'est cette deuxième option que nous allons choisir ici.

La structure `RJ` est une `imref2d` et permet de définir un repère spatial lié à l'image. Pour plus de détails, voyez l'aide de Matlab. Ce qui nous intéresse ici, est qu'on peut aussi imposer cette structure en entrée, via le paramètre `'OutputView'`. Afin de conserver en sortie la même taille d'image et plus généralement le même espace image qu'en entrée, on peut donc faire :

```
>> J=imwarp(I,A,'OutputView',imref2d(size(I)))
```

Enfin, un troisième paramètre est intéressant : c'est `FillValues`. En effet, vous remarquez dans l'image transformée `J` que certains pixels sont mis à noir car ils ne correspondent à aucun pixel de l'image originale. `FillValues` indique la (ou les, dans le cas d'une image couleur) valeur(s) à utiliser pour ces pixels de fond. Par défaut elle vaut 0, mais dans le cadre du recalage d'image, il est intéressant de lui donner une valeur qui ne se rencontre pas dans l'image à

1. Le `1:2` permet de ne prendre que les deux premières composantes de la taille, ce qui est pratique en cas d'image couleur

transformer, par exemple -1 (si vous traitez des images dont les valeurs peuvent être négatives, un bon choix générique est `min(min(I))-1`). Voici l'exemple final complet pour faire une bonne transformation d'image dans le cadre du recalage d'images :

```
>> J=imwarp(double(I),A,'OutputView',imref2d(size(I)), 'FillValues', -1);
>> imshow(uint8(J));
```

Notez ici que j'ai casté l'image originale en `double` pour permettre le stockage effectif de valeurs négatives dans `J`. Je recaste `J` en `uint8` pour l'affichage, mais gardez à l'esprit que la valeur 0 sera affichée en lieu et place des -1 (mais `J` contient bien toujours les valeurs négatives).

Cette manière de faire permet de sélectionner les pixels pertinents pour le recalage de la manière suivante : `idx=find(J>=0)` (c'est un exemple. On peut aussi avoir `idx=find(J ~= -1)`, ou `idx=find(J>-1)`...), et gérer ainsi de manière efficace et élégante, par exemple le problème de l'overlapping.

Afin de pouvoir utiliser une commande plus sympathique pour le recalage d'images, je vous propose de coder une petite fonction :

```
% I est l'image en entree
% A est la transformation a appliquer
% fv est la valeur pour les pixels de fond (-1 par defaut)
% J est l'image transformee
function [J,RJ]=transformImage(I,A,fv)
  % met fv a -1 si pas de 3e parametre
  if nargin < 3
    fv=-1;
  end
  [J,RJ]=imwarp(I,A,'OutputView', imref2d(size(I)), 'FillValues', fv);
end
```

2 Recalage rigide basé points (*Exercice noté*)

Le but de cet exercice est d'implanter et d'étudier le recalage basé sur des points appariés. Ces points peuvent par exemple être des points d'intérêt pouvant être extraits grâce aux méthodes que vous avez vues par ailleurs.

2.1 Procuste

Téléchargez l'archive `http://members.loria.fr/EKerrien/files/data/CETS8AH_images3.zip`. Elle contient une image `varan.jpg`, associée à des points d'intérêt `varan.pts`, ainsi qu'une version transformée de l'image (par une transformation rigide inconnue de vous) `varan_t.jpg`, associée à des points d'intérêt `varan_t.pts`.

Vous chargerez les images avec la fonction `imread` et les points avec la fonction `load`.

Dans la suite les variables suivantes seront utilisées :

- `I` pour l'image `varan.jpg`
- `J` pour l'image `varan_t.jpg`
- `P` pour les points `varan.pts`
- `Q` pour les points `varan_t.pts`

Les points chargés sont tels qu'ils sont en correspondance : pour tout `i`, le point `P(i,:)`=P_i correspond au point `Q(i,:)`=Q_i.

Vous pouvez visualiser ces images et points grâce aux commandes suivantes :

```
>> imshow(I);
>> hold on
>> plot(P(:,1),P(:,2),'+y');
>> hold off
>> figure
>> imshow(J);
>> hold on
```

```
>> plot(Q(:,1),Q(:,2),'+y');
>> hold off
```

Questions notées Je reprends ici la même notation qu'en cours pour les expressions algébriques, à savoir que les vecteurs sont des vecteurs colonnes (alors que Matlab considère des vecteurs ligne pour les transformations).

1. Donnez un exemple de couples de points appariés (petit nombre si possible... par exemple 3 ou 4 couples de points) pour lesquels la méthode des axes principaux échoue alors que la méthode de Procuste fonctionne bien.
2. Implantez une fonction matlab `A=find_trans_Procuste(P,Q)` qui prend en paramètres deux ensembles de N points `P` et `Q`, chacun sous la forme d'une matrice $N \times 2$, et renvoie le recalage rigide `A`, sous forme de transformation affine Matlab, estimé par la méthode de Procuste (voir la fonction Matlab `svd`) de telle manière que la relation suivante est au mieux respectée : `Q=transformPointsForward(A,P)`.
3. Comme il peut y avoir du bruit sur la position des points `P` et `Q`, la relation précédente n'est pas exactement respectée. Il existe une erreur résiduelle que l'on peut mesurer pour chaque couple d'indice i, par :

   $$e_i = Q_i - (\mathbf{R}P_i + \mathbf{T})$$

   Écrivez une fonction `E=erreur_trans(P,Q,A)` qui prend deux ensembles de points `P` et `Q`, ainsi qu'une transformation `A` et renvoie l'erreur quadratique moyenne (vue en cours, que Procuste minimise) :

   $$E = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|Q_i - (\mathbf{R}P_i + \mathbf{T})\|^2}$$

   où N est le nombre de points `P` (et `Q`), et $(\mathbf{R}, \mathbf{T})$ sont la matrice de rotation et le vecteur translation définissant la transformation `A`. On a donc l'équivalence entre formule mathématique et code :

   $$\mathbf{R}P_i + \mathbf{T} = \texttt{transformPointsForward(A,P(i,:))'}$$

   Note : *Si vous avec un vecteur colonne D à N dimensions, dont chaque composante se note d_i, alors la formule suivante est vérifiée :*

   $$\sum_{i=1}^{N} d_i^2 = D^t D$$

4. vérifiez que votre implantation fonctionne avec les images et points chargés ci-dessus. **Donnez-moi la valeur de `E`, ainsi que les coordonnées des 3 vecteurs `X`, `Y` et `Z` suivants :**
   - `E=erreur_trans(P,Q,A)`
   - `X=transformPointsForward(A,[0 0])`
   - `Y=transformPointsForward(A,[1 0])`
   - `Z=transformPointsForward(A,[0 1])`

3 Recalage rigide par ICP (*Exercice noté*)

L'ICP, ou *Iterated Closest Points*, permet de recaler non plus des points mais des courbes, décrivant par exemple le contour de formes dans l'image.

Le principe en est le suivant :

1. Soient deux courbes, décrites chacune par un ensemble de points, notées `P` et `Q` (les courbes sont donc polygonales)
2. recopier les points `P` dans `Pa`
3. initialiser la transformation recherchée `A` au déplacement nul : `A=makeRigidTransform(0,[0,0],[0,0])`
4. Pour chaque point de `Pa`, rechercher le point le plus proche dans `Q`. Former un ensemble `Qa` avec ces points les plus proches. (On pourra utiliser la fonction `dsearchn`).
5. recaler `Pa` sur `Qa` en utilisant la méthode de Procruste : on trouve `Aa`
6. mettre à jour la transformation `A` en la combinant avec `Aa`.

   La composition de transformations n'est pas codée dans Matlab. Pour mettre à jour `A` à partir de `Aa`, vous pourrez utiliser le bout de code suivant :

   ```
   >> A = rigid2d(A.Rotation*Aa.Rotation,A.Translation*Aa.Rotation+Aa.Translation)
   ```
7. Si l'erreur de recalage est inférieure à un seuil `t`, arrêter et retourner `A`.
8. Sinon, mettre à jour `Pa` en le remplaçant par `transformPointsForward(A,P)` et recommencer au point 4

h

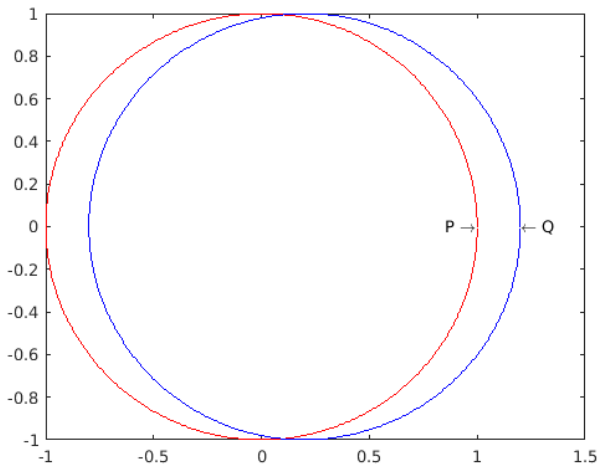

FIGURE 1 – Deux ensembles `P` et `Q` sur lesquels tester votre algorithme de recalage de courbes

Question : Écrire une fonction `A=recaleICP(P,Q,t)` qui renvoie la transformation recalant `P` sur `Q` avec un seuil d'erreur `t`. Vous pourrez par exemple tester votre algorithme sur les points créés par les commandes Matlab suivantes :

```
>> a=[0:0.01:2*pi]
>> P=[cos(a);sin(a)]
>> Q=P+repmat([0.2;0],[1,size(P,2)])
```

La figure 1 en montre un affichage réalisé par les commandes

```
>> plot(P(1,:),P(2,:),'r-',Q(1,:),Q(2,:),'b-')
>> text(0.85,0,'P \rightarrow')
>> text(1.2,0,'\leftarrow Q')
```

Note : *Cette question est volontairement moins guidée que les autres. J'attends que vous me montriez des résultats sur des formes pertinentes qui me permettront d'appréhender visuellement la qualité du résultat. Je testerai votre code sur mes formes secrètes... donc soyez imaginatifs pour essayer des formes qui peuvent faire échouer la méthode (car elle n'est pas infaillible) !*