MT51 - TP1
Étude des groupes, sous-groupes et tranformations de motifs

Laurent Couvidou – rendu le 1 mai 2006

1. Outils de base

1. *Rotation affine*

On utilise la matrice de rotation pour calculer le roté de m par r_θ, soit le code Matlab suivant :

```
function m2=rote(theta,m1)
R=[cos(theta),-sin(theta);sin(theta),cos(theta)];
m2=R*m1;
end
```

Voir le fichier `rote.m`.

On code une deuxième version effectuant des vérifications sur les paramètres d'entrée (cf. `rote_verif.m`). Cependant, on gardera la version la plus simple par soucis de légèreté (principe que l'on appliquera à tout le TP).

2. *Symétrie affine*

De la même manière, on utilise la matrice de symétrie par rapport à $(O, \vec{i}, \vec{j})$:

```
function m2=symetrique(m1)
S=[1,0;0,-1];
m2=S*m1;
end
```

Voir les fichiers `symetrique.m` et `symetrique_verif.m`.

2. Étude du groupe diédral $(D_{2n}, \circ)$ pour $(n=4)$

1. *Conversions décimal – quadratique*

On accepte un entier k entre 0 et 7 ; k s'écrit : $k = \alpha.4^1 + \beta^0 = 4\alpha + \beta$. α est la division entière de k par 4, et β le reste. On obtient donc le code Matlab suivant pour la conversion quadratique – décimal, adapté aux matrices (cf. `trad_qu_dec.m` et `trad_qu_dec_verif.m`) :

```
function k=trad_qu_dec(alpha,beta)
k=alpha.*4+beta;
end
```

Et pour la conversion décimal-quadratique (`trad_dec_quad.m` et `trad_dec_qu_verif.m`) :

```
function [alpha,beta]=trad_dec_qu(k)
alpha=floor(k./4);
beta=mod(k,4);
end
```

2. *Table de la loi ∘ sur D_8*

La fonction `compose` prends les indices i et j de deux transformations de D_8, et renvoie l'indice k de leur composition. On utilise les conversions ci-dessus et les formules vues en TD (démonstration en quatre étapes) :

$$e_i \circ e_j = e_k$$
$$i \leftrightarrow (\alpha_i, \beta_i) \quad j \leftrightarrow (\alpha_j, \beta_j)$$
$$k \leftrightarrow (\alpha, \beta) \begin{cases} \alpha \equiv (\alpha_i + \alpha_j) (mod\ 2) \\ \beta \equiv ((-1)^{(\alpha_j)} \beta_i + \beta_j) (mod\ 4) \end{cases}$$

Soit sous Matlab (cf. `compose.m` et `compose_verif.m`), en prenant en compte les matrices :

```
function k=compose(i,j)
[alphai,betai]=trad_dec_qu(i);
[alphaj,betaj]=trad_dec_qu(j);
alpha = mod(alphai+alphaj,2);
beta = mod(betai.*(-1).^alphaj+betaj,n);
k=trad_qu_dec(alpha,beta);
end
```

On peut désormais écrire la table de composition en associant chaque transformation une à une (cf. `tab_compo.m`) :

```
function tab=tab_compo()
for i=1:8
    for j=1:8
        tab(i,j)=compose(i-1,j-1);
    end
end
end
```

Et on obtient :

```
>> tabcompo()
ans =
     0     1     2     3     4     5     6     7
     1     2     3     0     7     4     5     6
     2     3     0     1     6     7     4     5
     3     0     1     2     5     6     7     4
     4     5     6     7     0     1     2     3
     5     6     7     4     3     0     1     2
     6     7     4     5     2     3     0     1
     7     4     5     6     1     2     3     0
```

3. Preuve : $(D_8, \circ)$ abélien

$(D_8, \circ)$ est abélien si et seulement si :
1. $\circ$ est interne : *vérifié en TD, et utilisé ci-dessus* ;
2. $\circ$ est associative : *la composition est associative* ;
3. Il existe un élément neutre e_0 pour $\circ$: *c'est l'identité (transformation d'indice 0 dans D_8)* ;
4. Tout élément de D_8 admet un inverse dans D_8 : **à vérifier**.

L'inverse de l'élément e_i de D_8 est e_i^{-1} tel que $e_i \circ e_i^{-1} = e_0$. On constate dans la table de composition qu'il y a un zéro par ligne. On peut trouver un e_j pour chaque e_i tel que $e_i \circ e_j = e_0$: le point 4. est vérifié.

Pour obtenir un inverse sous Matlab, on parcoure la ligne i, et on cherche la colonne j dans laquelle se situe le zéro (cf. `inverse_rech.m`) :

```
function j=inverse_rech(i)
j=0;
tab=tab_compo();
while tab(i+1,j+1) ~= 0
    j = j+1;
end
end
```

Ou, plus simplement, on constate en observant la table :

$Pour\ i \in \{0,3\}\ l'inverse\ de\ e_i \leftrightarrow (0,i)\ est\ e_j \leftrightarrow (0,(4-i)(mod\ 4))$
$Pour\ i \in \{4,7\}\ l'inverse\ de\ e_i\ est\ e_i$

Soit sous Matlab (cf. `inverse.m`) :

```
function j=inverse(i)
if i>=0 && i<=3
    j = mod(4-i,4);
else
    j = i;
end
end
```

4. Stockage de la table de composition

La fonction `tab_compo()` permet d'y accéder à tout moment, pour la stocker en statique :

```
tab = tab_compo()
```

5. Sous-groupes de D_8

On utilise le théorème de caractérisation (théorème 7 du cours, p.2) : pour générer un sous-groupe à partir d'un certain nombre d'éléments, il suffit de composer chaque élément avec l'inverse de lui-même et de tous les autres, de supprimer les doublons et de recommencer jusqu'à ce que le résultat ne change plus.

C'est l'objet du code `sous_groupe_engendre_th7.m`.

Comme vu en TD, on peut simplifier en composant simplement chaque élément avec lui-même et ceux qui le suivent, ce qui nous donne (cf. `sous_groupe_engendre.m`) :

```
function sg=sous_groupe_engendre(e)
taille = size(e,2);
% Théorème de Lagrange : on a dépassé la taille n le sous-
% groupe est donc le groupe complet
if taille > 4
    sg = 0:7;
    return;
end
sg = e; % Le sous-groupe comprends les éléments de départ
changement = 0;
% Pour chaque élément fourni
for (cpt1=1:taille)
    % On prends le même, puis les suivants
    for (cpt2=cpt1:taille)
        % Et on effectue une composition
        nouveau = compose(e(1,cpt1),e(1,cpt2),n);
        % Si le résultat n'est pas déjà dans le sous-groupe
        % en cours de construction, on l'y ajoute et on
        % note le changement
        if ~any(nouveau == sg)
            % Théorème de Lagrange : on va dépasser la
            % taille 4, le sous-groupe est donc le groupe
            % au complet
            if size(sg,2) == n
                sg = 0:7;
                return;
            end
            sg = [sg,nouveau];
            changement = 1;
        end
    end
end
% En cas de changement, on réeffectue toutes les
% compositions sur le sous-groupe obtenu (lourd, mais
% efficace)
if (changement)
    sg = sous_groupe_engendre(sg);
end
end
```

On pourrait considérer qu'une version matricielle serait plus efficace. Cependant, même si telle qu'elle est codée la fonction `compose` accepte les matrices, la necessité de « croiser » les compositions entre les éléments ne permet pas d'obtenir un code simple (cf. tentative dans `sous_groupe_engendre_mat.h`).

De plus, coder cette fonction en itératif permet de placer des conditions d'arrêt efficaces (utilisant le théorème de Lagrange), et d'arrêter ainsi les calculs dès que nécessaire. Les conditions d'arrêt étant atteintes très vite, on gardera donc cette version itérative.

Pour les sous-groupes engendrés à partir d'un élément, on obtient :

```
>> for e=0:7, sg=sous_groupe_engendre(e), end
sg =
     0                                  engendré par  id
sg =
     1     2     3     0                engendré par  r
sg =
     2     0                            engendré par  r²
sg =
     3     2     1     0                engendré par  r^3
sg =
     4     0                            engendré par  s
sg =
     5     0                            engendré par  s∘r
sg =
     6     0                            engendré par  s∘r²
sg =
     7     0                            engendré par  s∘r^3
```

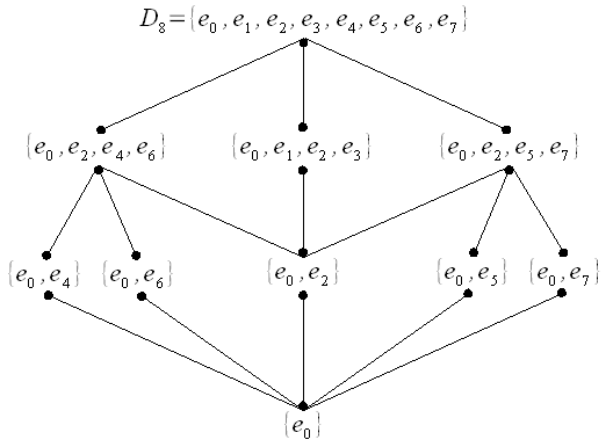
Croiser les rotations r avec n'importe r^3 quelle autre transformations différente de l'identité donne directement D_8.

r^2 semble plus intéressante, croisons-la avec les symétries :

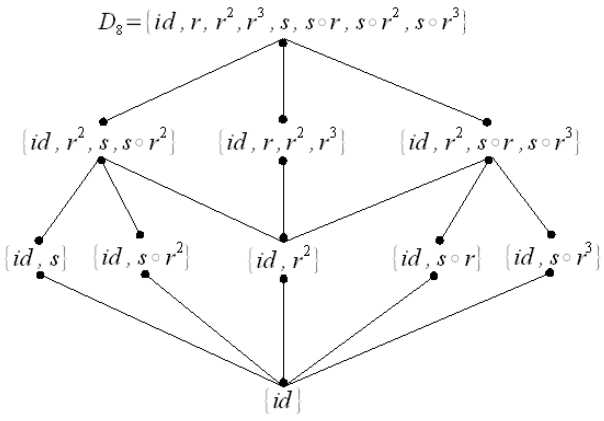
```
>> for e=4:7, sg=sous_groupe_engendre([2,e]), end
sg =
     2     4     0     6
sg =
     2     5     0     7
sg =
     2     6     0     4
sg =
     2     7     0     5
```

On obtient deux sous-groupes non encore rencontrés. On n'en trouve aucun autre, même en croisant le symétries entre elles. On a donc, en tout, 9 sous-groupes de D_8 distincts, que l'on peut trier dans un diagramme de Hasse selon la relation d'inclusion :

Notations e_i :



Avec les transformations :



3. Étude du groupe diédral $(D_{2n}, \circ)$ pour n quelquonque

On réécrit les fonctions précédente, en ajoutant un paramètre n optionnel. Ceci est réalisé en utilisant la variable nargin de Matlab, qui permet de savoir combien de paramètres sont passés à l'appel de fonction. Si n n'est pas spécifié, on lui passe la valeur 4 par défaut.

Il suffit ensuite de généraliser le code de la fonction. Par exemple, pour `trad_dec_qu` :

```
function [alpha,beta]=trad_dec_qu(k,n)
% n est optionnel
if nargin == 1, n = 4; end % Travail dans D8 par défaut
alpha=floor(k./n);
beta=mod(k,n);
end
```

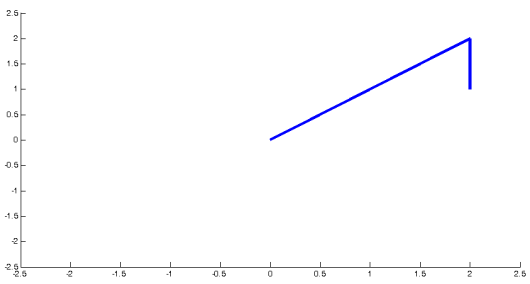
4. Transfert de motif

1. *Représentation visuelle des sous-groupes*

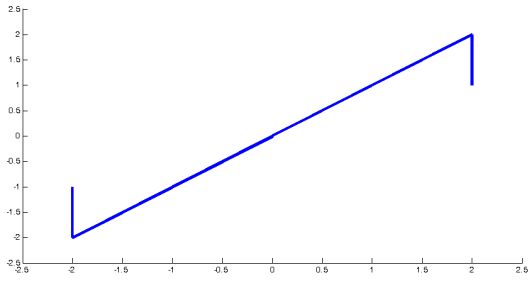
On utilise le motif M_0 proposé dans le sujet, et on affiche ses transformations pour chaque sous-groupe, en utilisant `creat_motif.m`.

Voici les motids obtenus pour chaque sous-groupe :

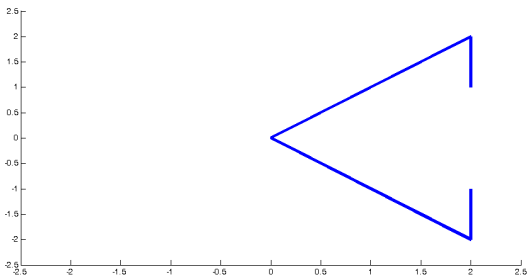
`creat_motif(sous_groupe_engendre(0))`
$soit\ M_H\ pour\ H=\{id\}$



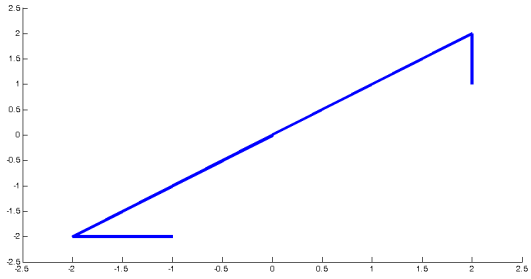
`creat_motif(sous_groupe_engendre(2))`
$soit\ M_H\ pour\ H=\{id, r^2\}$



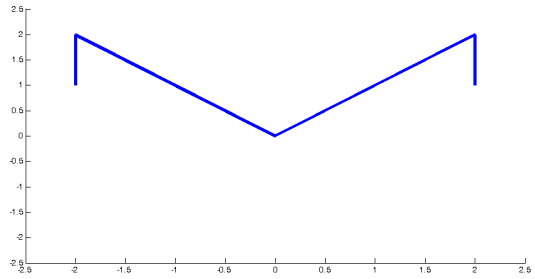
`creat_motif(sous_groupe_engendre(4))`
$soit\ M_H\ pour\ H=\{id, s\}$



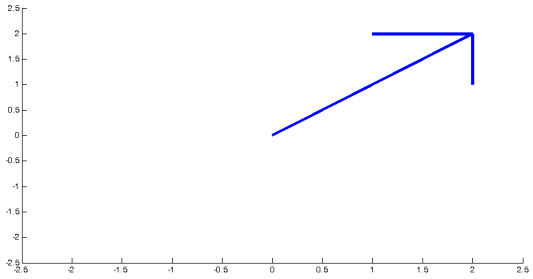
`creat_motif(sous_groupe_engendre(5))`
$soit\ M_H\ pour\ H=\{id, s\circ r\}$



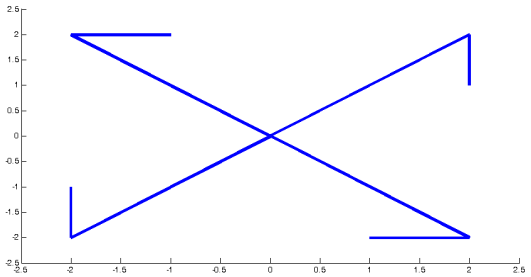
`creat_motif(sous_groupe_engendre(6))`
soit M_H pour $H=\{id, s\circ r^2\}$



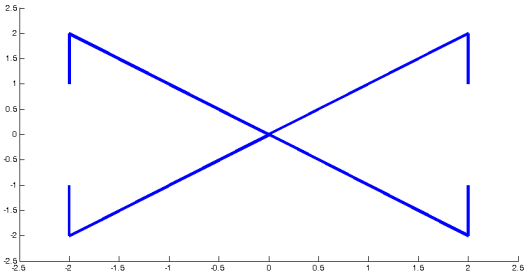
`creat_motif(sous_groupe_engendre(7))`
soit M_H pour $H=\{id, s\circ r^3\}$



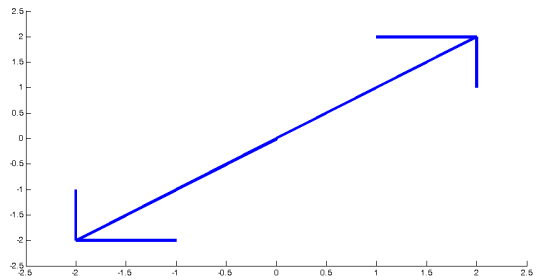
`creat_motif(sous_groupe_engendre(1))`
soit M_H pour $H=\{id, r, r^2, r^3\}$



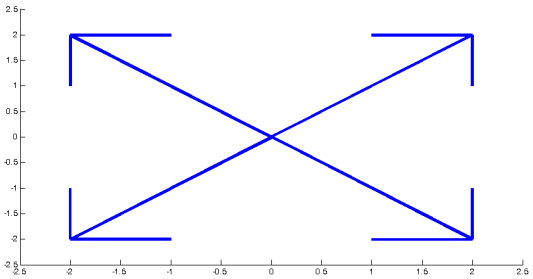
`creat_motif(sous_groupe_engendre(2,4))`
soit M_H pour $H=\{id, r^2, s, s\circ r^2\}$



`creat_motif(sous_groupe_engendre(2,5))`
soit M_H pour $H=\{id, r^2, s\circ r, s\circ r^3\}$



`creat_motif(sous_groupe_engendre(0:7))`
soit M_H pour $H=D_8$



2. *Tranfert de forme de base*

La rotation et la symétrie du plan affine sont des isométries : les longueurs ne changent pas. Pour transférer une forme de base, il suffira donc de transférer quelques points et de garder les caractéristiques de la forme.

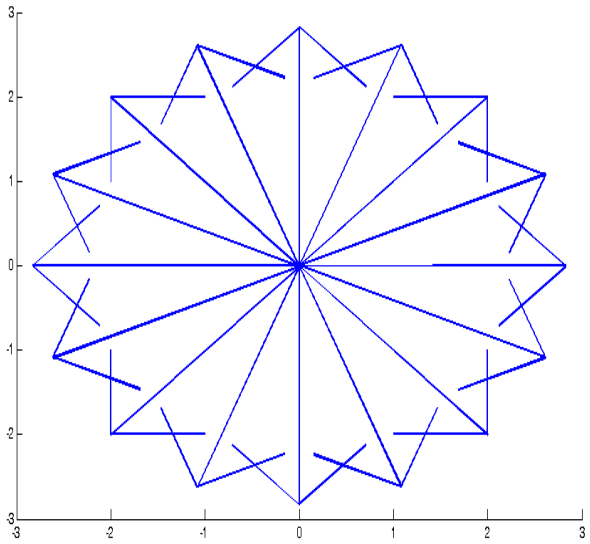
Exemple : pour un cercle, on transfère le centre, et on redessine avec le même rayon.

3. *Prolongements*

On généralise la fonction `create_motif`, de manière à pouvoir représenter les transformations d'un motif par n'importe quel groupe ou sous-groupe diédral D_{2n} .

Exemple : n=16

```
>> creat_motif(0:31,16)
```



Enfin on l'étends pour prendre en compte des listes de points quelquonques, formant ainsi les motifs souhaités.

Exemple :

```
>> X=[0,1,2,1,0];
>> Y=[0,1,1,0,0];
>> creat_motif(0:7,4,X,Y);
```