# Fractales

Jean-Baptiste Rouquier

Voici un TP sur des opérations géométriques. Nous allons tracer des fractales. Il y a plusieurs morceaux de code dans ce TP, pour les recopier dans votre fichier depuis xpdf :

- sélectionnez le texte à recopier en cliquant-glissant avec le bouton gauche (sous Adobe Reader, tapez «v» d'abord pour prendre l'outil «sélection») ;
- collez le texte dans votre éditeur : sous Linux, faîtes un clic du milieu (le bouton du milieu est souvent une roulette, s'il n'y a que deux boutons le troisième est souvent émulé par un clic sur les deux à la fois) à l'endroit où vous voulez insérer le code.

Pour utiliser les fonctions graphiques, il faut ajouter le code suivant en tête du fichier :

```
#load "graphics.cma";;
open Graphics
```

La seconde ligne n'est pas indispensable mais permet de taper par exemple `read_key` au lieu de `Graphics.read_key`.

Nous travaillons en deux dimensions :

```
type point = {x:float; y:float}
```

Le code suivant vous permet d'écrire `(f % g % h) x` pour calculer $f \circ g \circ h\,(x)$ :

```
let (%) f g x = f (g x)
```

Cela peut s'avérer pratique lorsqu'il s'agit de composer des transformations géométriques.

## 1 Transformations géométriques

**Question 1.1.** Écrire les fonctions suivantes

```
translation : point -> point -> point
rotation_origine : float -> point -> point
oppose : point -> point
rotation : point -> float -> point -> point
homothetie : float -> point -> point
distance : point -> point -> float
```

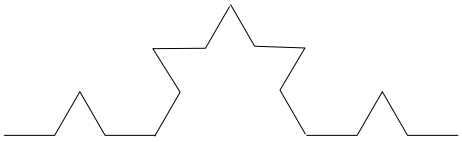
- `oppose` est la symétrie par rapport à l'origine,
- `rotation` prend en argument le centre et l'angle,
- `homothetie` se fait par rapport à l'origine,
- `distance` est la distance euclidienne.

**Question 1.2.** Définir `pi` (de type `float`) à l'aide d'une des fonctions `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`.

## 2 Courbes

Nous allons tracer la courbe de Von Koch. On part d'un segment de droite et on le remplace par quatre segments de la façon suivante :

On obtient le niveau 1. Pour passer du niveau $i$ au niveau $i+1$, on applique la même transformation à chacun des segments du niveau $i$. Ainsi le niveau 2 est



**Question 2.1.** Écrire `draw_line : point -> point -> unit`. On pourra utiliser `Graphics.moveto : int -> int -> unit` qui place le «point courant» sur le pixel dont on a donné les coordonnées, suivi de `Graphics.lineto : int -> int -> unit` qui trace une ligne depuis le point courant jusqu'au pixel donné.

**Question 2.2.** Écrire la fonction `decoupe : point -> point -> point array` qui prend les deux extrémités d'un segment $AB$ et renvoie un tableau de cinq points, qui sont les extrémités des quatre segments par lesquels on va remplacer $AB$.

**Question 2.3.** Écrire la fonction récursive `draw_koch : int -> point -> point -> unit` telle que `draw_koch p a b` dessine le niveau `p` entre les points `a` et `b`. Le niveau 0 est un simple segment.

Testez votre fonction par le code suivant, en appuyant sur une touche (après avoir sélectionné la fenêtre graphique) pour passer à l'échelle suivante.

```
let () =
  open_graph "";
  let width = size_x () in
  let a,b = {x = 0.; y = 0.}, {x=float width; y = 0.} in
  for profondeur = 1 to 8 do
    draw_koch profondeur a b;
    ignore (Graphics.read_key ());
    clear_graph ();
  done;
  close_graph ()
```

**Question 2.4 (bonus).** Écrire

```
draw :
    (point -> point -> point array * bool array) ->
    int -> point -> point -> unit
```

analogue à `draw_koch`, mais qui prend en paramètre la fonction de découpage au lieu de toujours utiliser `decoupe`. Cette fonction de découpage est de type `point -> point -> point array * bool array` : elle renvoie toujours les extrémités des segments à tracer (dans un `point array`), mais aussi une variable supplémentaire `reversed`, de type `bool array`. Si `reversed.(i)` est `true`, alors il faut dessiner le $(i+1)^{\text{e}}$ segment $AB$ en partant de $B$ et en allant vers $A$ (au lieu de le tracer de $A$ à $B$).

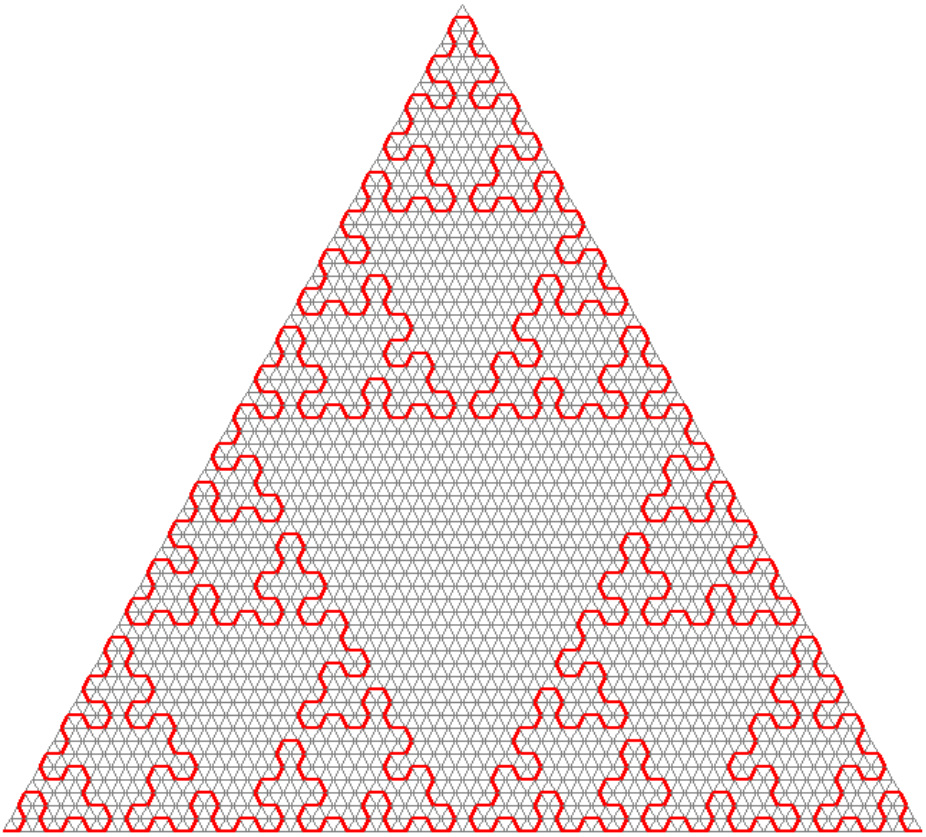
Testez votre fonction draw avec les fonctions de découpe suivantes :

```
let decoupe_dragon a b =
  let c = (translation a % rotation_origine (pi/.4.)
           % homothetie (sqrt 2. /.2.) % translation (oppose a)) b in
  [|a; c; b|],
  [|false; true|]

let decoupe_3 a b =
  let m = translation (oppose a) b in
```

```
let c = (translation a % homothetie 0.47) m in
let milieu = homothetie 0.5 m in
let d = (translation a % translation milieu % rotation_origine (pi/.2.)
         % homothetie 0.95) milieu in
let e = (translation a % homothetie 0.53) m in
[|a; c; d; e; b|], [|false; false; false; false|]
```

**Question 2.5 (bonus).** Écrire la fonction de découpe qui permet de tracer la courbe suivante :



# 3 Iterated Functions System

Comme leur nom l'indique, les fractales de type IFS sont construites en itérant un ensemble de fonctions.

Précisément on choisit un ensemble $F$ de transformations contractantes du plan. On tire au hasard un point $p$ (ou bien on part du centre, cela a peu d'importance). Puis on répète un grand nombre de fois les opérations

- tirer au sort $f \in F$ ;
- remplacer $p$ par $f(p)$ (c'est-à-dire «$p := f(p)$») ;
- colorier en noir le pixel le plus proche de $p$[1].

Cette idée a été proposée par Barnsley. Elle a ensuite été utilisée pour compresser des images (l'idée est de trouver l'ensemble $F$ qui va recréer l'image de départ par l'algorithme précédent ; on peut traiter les couleurs).

**Question 3.1.** Écrire `etape : point -> (point -> point) array -> point`.

**Question 3.2.** Écrire `main : (point -> point) array -> int -> unit` qui prend l'ensemble $F$ et le nombre d'itérations, et applique l'algorithme précédent.

[1] Vous pouvez utiliser `Graphics.plot : int -> int -> unit` qui prend les coordonnées du pixel à colorier

Testez votre fonction avec le code suivant :

```
let () =
  open_graph "";
  let width = float (size_x ()) and height = float (size_y ()) in
  let f = homothetie 0.45 in
  let cote = 0.55 *. height in
  main
    [| f;
       translation {x=cote;y=0.} % f;
       translation {x=cote;y=cote} % f;
       translation {x=0.;y=cote} % f|]
    100_000;
  let f = homothetie 0.5 in
  main (* Triangle de Sierpinski *)
    [| f;
       translation {x=0.5 *. width;  y=0.} % f;
       translation {x=0.25 *. width;  y= width *. sqrt 3. /. 4.} % f|]
    100_000;
  let f = homothetie (1./.3.) in
  let cote = width /. 3. in
  main (* courbe de Von Koch *)
  [| f;
     translation {x=cote; y=0.} % rotation_origine (pi /. 3.) % f;
     translation {x=cote *. 1.5; y=cote*.sqrt 3. /. 2.}
       % rotation_origine (-. pi /. 3.) % f;
     translation {x=2.*.cote; y=0.} % f
  |]
  100_000;
  close_graph ()
```

**Question 3.3 (bonus).** Ajouter les affinités qui transforment un carré en un rectangle (c'est une «homothétie» appliquée seulement à l'ordonnée) et le «cisaillement» qui transforme un carré en un parallèlogramme (en anglais *shear*). Ou mieux, généraliser en traitant tout par des matrices et vecteurs à deux dimensions. Interfacer avec la souris pour que l'on puisse modifier facilement les rectangles (ou quadrilatères généraux), ajouter la possibilité d'enregistrer les paramètres numériques et l'image bitmap, faire des menus et boîtes de dialogue conviviaux grâce à la GTK (annulation, raccourcis clavier, couleurs, etc.), écrire une documentation complète, bref, récrire en OCaml la «machine à copie multiple»[2] ☺.

[2] http://jlpfractware.free.fr/