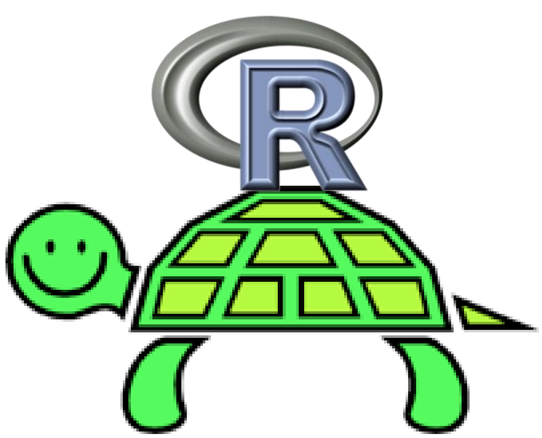
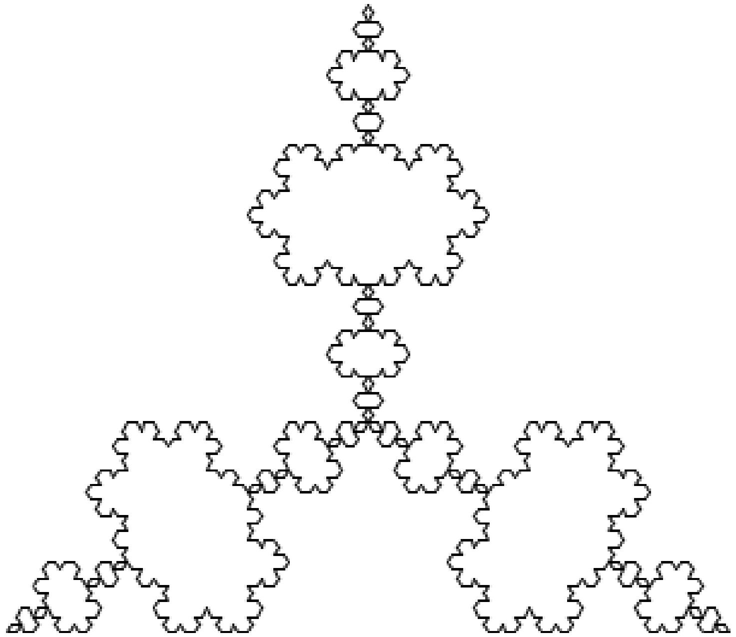
Université Nice Sophia Antipolis
Faculté des Sciences

http://www.i3s.unice.fr/~malapert/org/teaching/introR.html

La Tortue



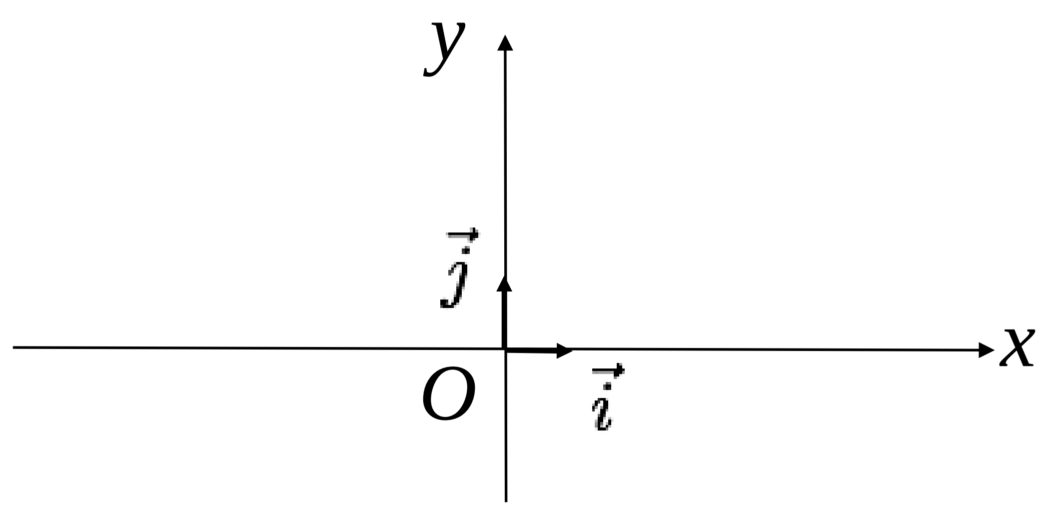


Les deux types de graphisme dans le plan

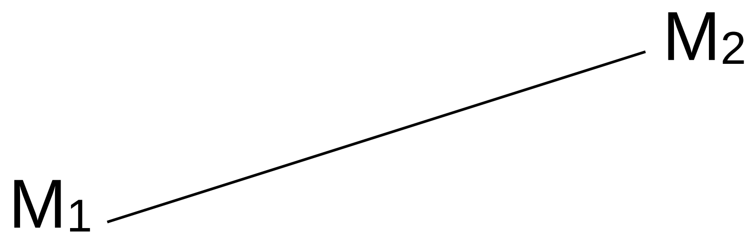
- Il y a deux types de graphisme 2D, mathématiquement parlant :

1. Le graphisme CARTESIEN (global)

- Le plan est rapporté à un repère orthonormé direct $(O,\vec{i},\vec{j})$.

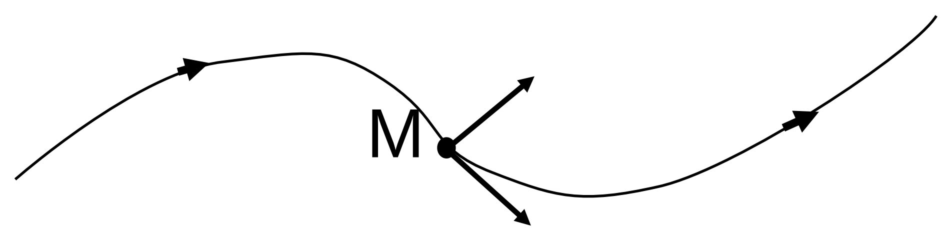


- Une seule opération essentielle :
  tracer un segment du point $M_1(x_1;y_1)$ au point $M_2(x_2;y_2)$.



2. Le graphisme POLAIRE (local)

• Aucune notion de coordonnées. L'animal traceur porte un repère mobile orthonormé avec une notion de droite et de gauche.



Au point M, la tortue va commencer à tourner sur sa gauche !

• Deux opérations essentielles :
- tourner à droite ou à gauche sur place d'un angle a
- avancer dans la direction courante d'une distance d

• Opérateurs de ***translation*** et de ***rotation*** plane, qui engendrent le groupe des ***déplacements***. La tortue se déplace dans le plan !

• Graphisme moins matheux, plus intuitif. Inutile de calculer les coordonnées des points...

• Une trajectoire qui semble *lisse* sera en fait un polygone !

Le module TurtleGraphics de R

- Le ***graphisme de la tortue*** a été inventé au Laboratoire d'Intelligence Artificielle du MIT vers 1968 avec le langage LOGO.

- Il est disponible dans quasiment tous les langages de programmation qui offrent des facilités graphiques.

- Et en particulier en R avec le module TurtleGraphics.

- Ce module n'est pas livré avec la distribution R standard.

```
install.packages("TurtleGraphics")
```

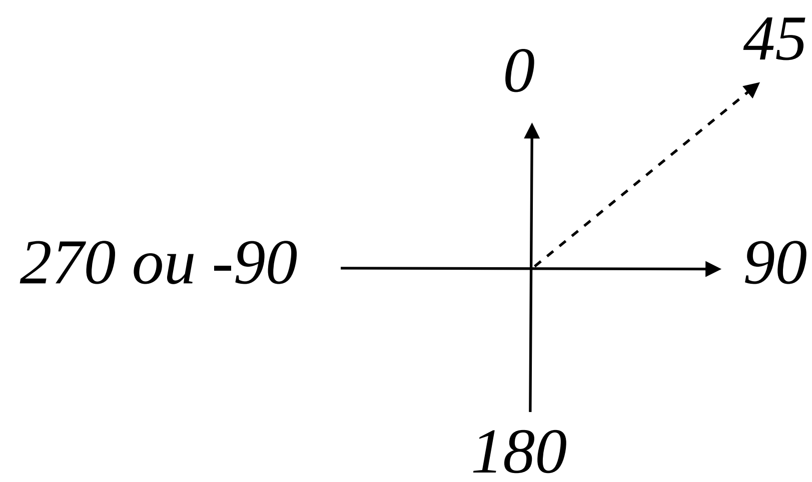
- Il faut en importer les noms pour pouvoir les utiliser :

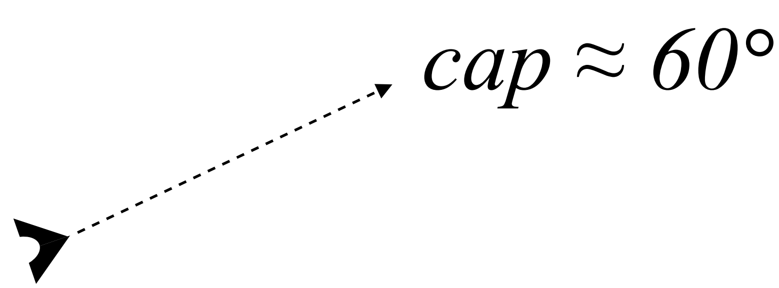
```
library(TurtleGraphics)
```

Le graphisme cartésien

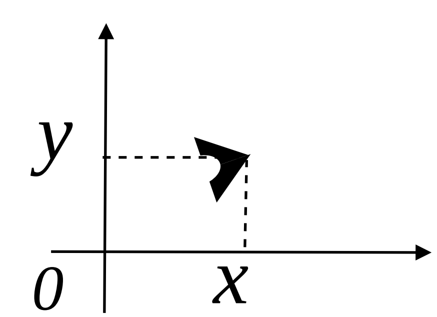
• C'est celui des matheux dans la mesure où il faut calculer les coordonnées des points à relier.

• Une *tortue* est représentée par une flèche qui indique son cap en **degrés** :





• Une *tortue* a une position : une abscisse et une ordonnée.



• Une *tortue* a un crayon (*pen*) qui peut être baissé (*down*) ou levé (*up*). Si le crayon est baissé, la tortue laisse une trace en se déplaçant. On peut choisir la couleur du crayon ainsi que le type et l'épaisseur de la ligne.

• Une tortue a donc un ETAT représenté mathématiquement par trois données : *position*, *cap*, *crayon*.

La position

turtle_getpos()
turtle_setpos(x,y)
turtle_goto(x,y)

Le cap

turtle_getangle ()
turtle_setangle(a)

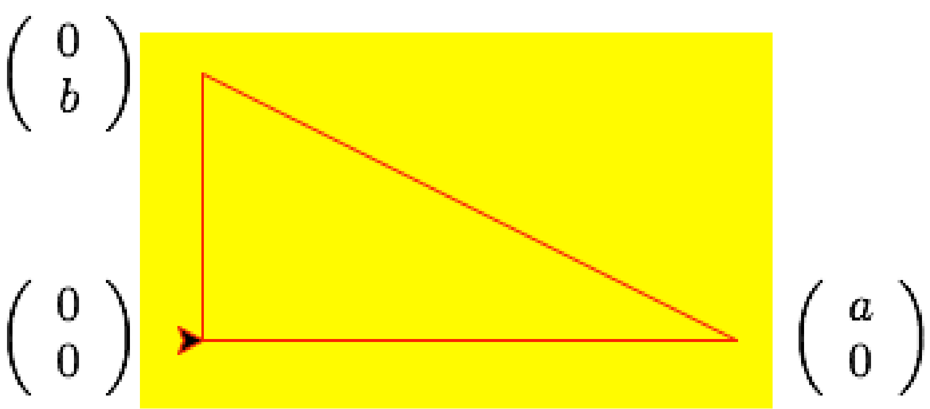
Le crayon

turtle_down()
turtle_up()
turtle_param(col, lwd ,lty)
turtle_col(col)
turtle_lwd(lwd)
turtle_lty(lty)

• Agir sur le canvas : turtle_init et turtle_reset().

• Exemple : dessin d'un triangle rectangle de côtés a et b.

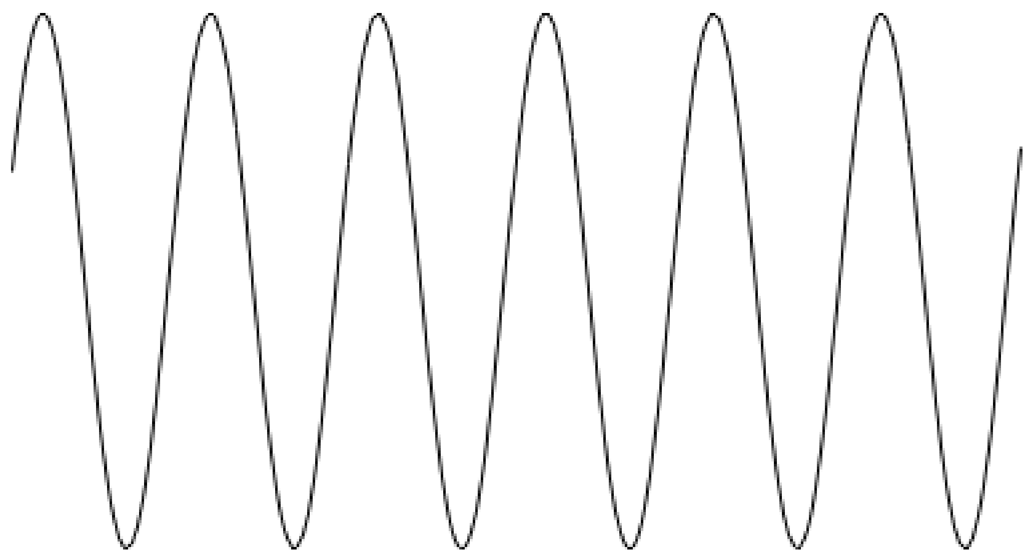
```
trirect <- function(a, b) {
    turtle_up()
    turtle_goto(0, 0);
    turtle_down()
    turtle_goto(a, 0)
    turtle_goto(0, b)
    turtle_goto(0, 0)
  }
```




```
turtle_init()
trirect(100, 50)
```

- Exemple : tracé de la courbe du cosinus

```
trace_function <- function(f, a, b, n=100) {
   turtle_up()
   turtle_goto(a, f(a))
   turtle_down()
   for(x in seq(a,b, length.out=n)) {
     turtle_goto(x, f(x))
   }
}
```



```
> turtle_reset()          # effacement du canvas, réinitialisation
> turtle_init(width=20, height=22)
>  turtle_do(trace_function(function(x) {10*(cos(x)+1)}, 0, 20))
```

- Comme goto ou trace_fonction, la plupart des fonctions de dessin n'ont pas de résultat, seulement des effets.

- ATTENTION, les points du canevas ont des coordonnées positives.

Le vecteur nommé

• Vous avez noté la présence d'ún vecteur, ici un couple (x,y).

```
> turtle_getpos()
 x  y
50 50
```

• Le résultat de la fonction turtle_getpos() est un couple dont les composantes se notent p[1], p[2], p[3]...

• De plus, les composantes sont ici nommées x et y.

```
> p <- turtle_getpos()
> p[1]
 x
50
> p[2]
 y
50
> p['x']
 x
50
```

• La très intéressante affectation entre vecteurs :

```
> p <- turtle_getpos()
 x  y
50 50
> p[c('x', 'y')] = c(1,2)
x y
1 2
> p[1:2] <- c(3,4)
x y
3 4
> p <- c(3,4)
[1] 3 4
```

p['x', 'y'] = c(a,b) $\not\Leftrightarrow$ p = c(a,b)

si a *et* b *sont deux expressions quelconques...*

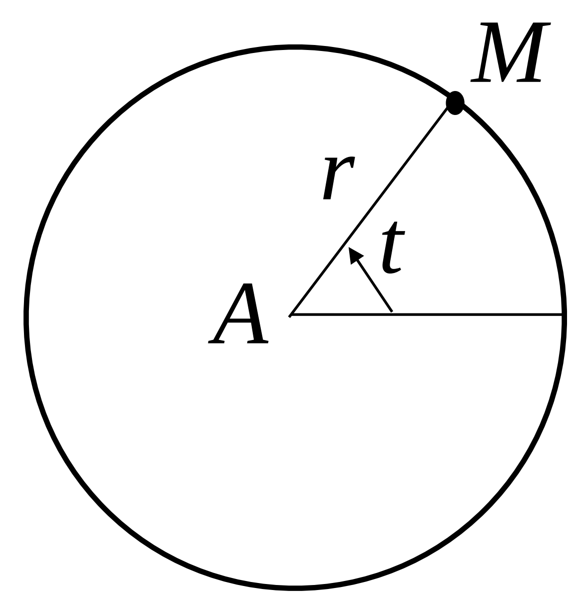
Courbes en coordonnées paramétriques

• La cinématique (étude du mouvement) s'intéresse à la trajectoire d'un corps dont les coordonnées (x,y) sont fonction d'un ***paramètre*** t.
Autrement dit : $x = x(t)$ et $y = y(t)$

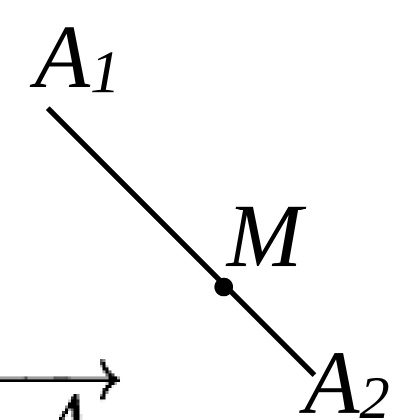
• Ces courbes englobent les courbes $y = f(x)$ mais sont plus générales !

• Exemple : le cercle de centre $A(a\,;\,b)$ et de rayon r n'est autre que la trajectoire d'un mobile dont les coordonnées sont données par :

$$M \quad \boxed{\begin{array}{l} x = a + r\cos(t) \\ y = b + r\sin(t) \end{array}}$$



• Exemple : le **segment** A_1A_2 joignant le point $A_1(a_1\,;\,b_1)$ au point $A_2(a_2\,;\,b_2)$ est la trajectoire paramétrée par :



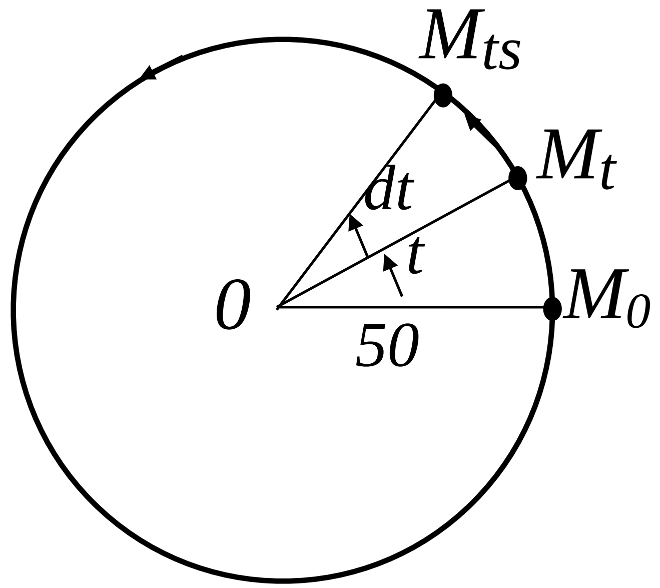
$$M \quad \boxed{\begin{array}{l} x = t\,a_1 + (1 - t)\,a_2 \\ y = t\,b_1 + (1 - t)\,b_2 \end{array}}$$

$$\overrightarrow{MA_2} = t\overrightarrow{A_1A_2}$$
$$t \in [0, 1]$$

• Animation de la tortue parcourant un cercle de centre 0 et de rayon 50. Le caractère *continu* du mouvement est une illusion d'optique. En fait il est discrétisé : le paramètre t avance chaque fois de dt.

• Le choix de n peut être empirique, guidé par l'esthétique de la simulation. Mais si l'on approche un cercle par un polygone à 40 côtés, on est conduit à prendre dt = 2*pi/40 ≈ 0.16

```
anim_cercle <- function(r, n=50) {
   turtle_up()
   turtle_goto(2*r,r)
   turtle_down()
   for(x in seq(0,6*pi, length.out=n)) {
     turtle_goto(r + r*cos(x), r + r*sin(x))
   }
 }
```




```
> turtle_init()
> anim_cercle(50)
```

Le graphisme polaire

- Il s'agit du *vrai* graphisme tortue pour les puristes...
- Nous ignorons la valeur du cap et de la position dans le graphisme polaire pur.

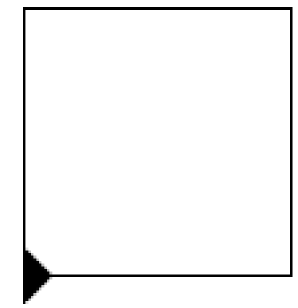
Le cap	La position
turtle_left(a)	turtle_forward(d)
turtle_right(a)	turtle_backward(d)
turtle_turn(a,dir)	turtle_move(d, dir)

- Notez que : turtle_right(a) ⇔ turtle_left(-a) et
turtle_backward(d) ⇔ turtle_forward(-d)

- Une suite d'appels à ces fonctions turtle_ left(...) et turtle_ forward(...) permet donc de décrire une courbe d'un seul tenant. En levant le crayon, on peut tracer plusieurs courbes non reliées entre elles.

- Exemple, dessin d'un carré de côté c.

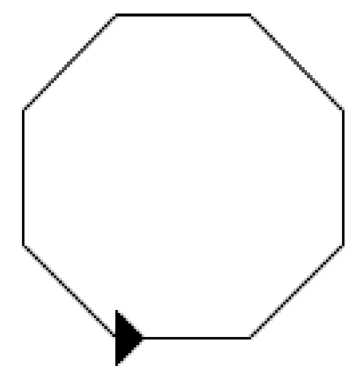
```
carre <- function(c) {
   for(i in 1:4) {
      turtle_forward(c)
      turtle_left(90)
   }
}
```



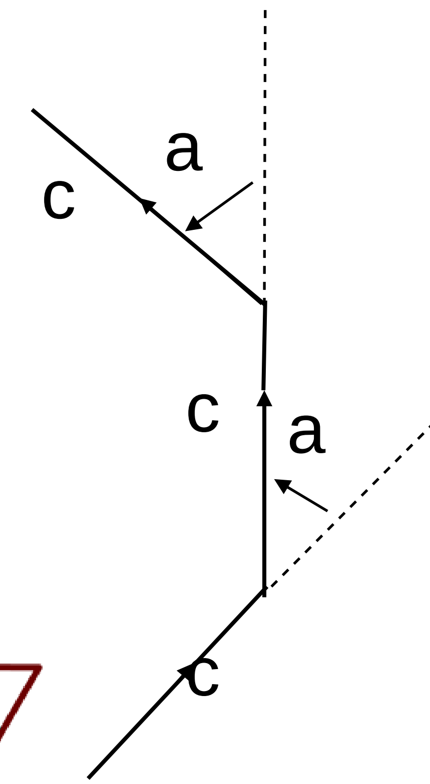
carre(100)

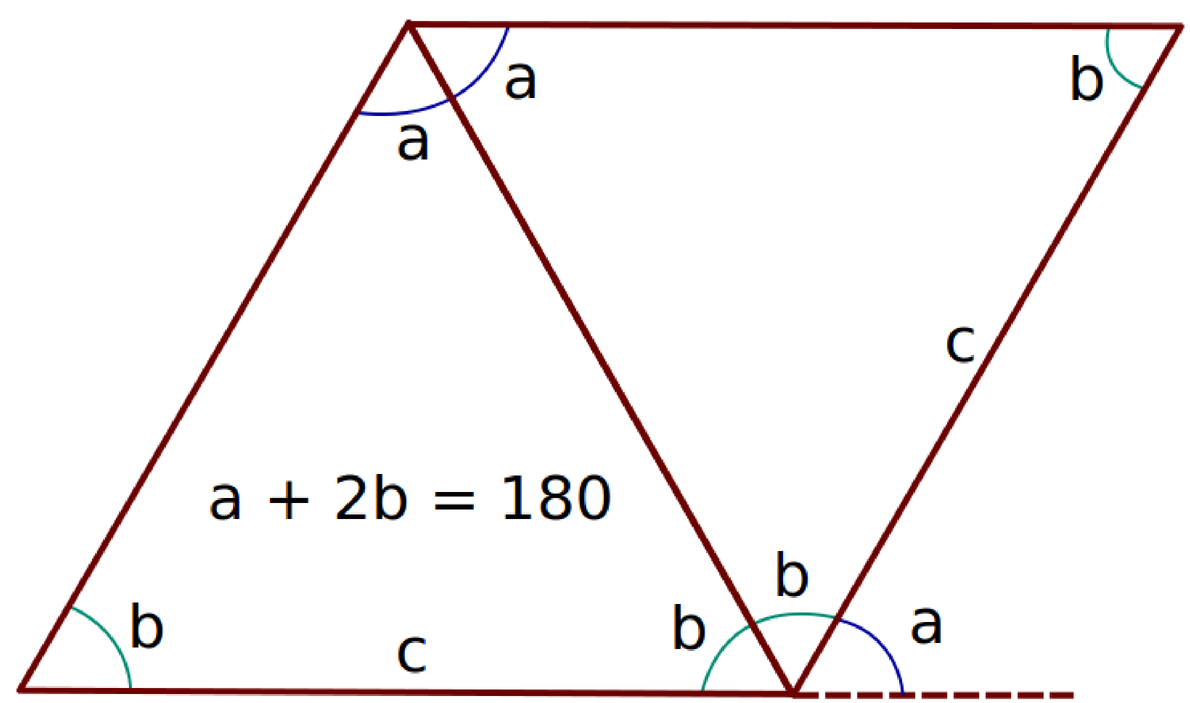
- Généralisation : dessin d'un polygone régulier à n côtés.

```
polygone <- function(n, c) {
  a <- 360 / n
  for(i in seq(n)) {
    turtle_forward(c)
    turtle_left(a)
  }
}
```



polygone(8,100)





La boucle for

- Elle est bien pratique lorsque l'on connaît à l'avance le nombre d'itérations :

```
carre <- function(c) {
    for(i in 1:4) {          # i = 1,2,3,4
          turtle_forward(c)
          turtle_left(90)
     }
}
```

```
polygone <- function(n,c) {
    a <- 360 / n
    for (i in seq(n)) {          # i = 1,2,…,n-1,n
        turtle_forward(c)
        turtle_left(a)
}
```

- i:j pour parcourir [i,j], et seq(n) pour parcourir [1,n]

- *Grosso modo*, si i est une variable inutilisée par ailleurs :

```
for (i in a:b) {
    <instr>
}
```

$\Leftrightarrow$

```
i <- a
while (i <= b) {
  <instr>
  i <- i + 1
}
```

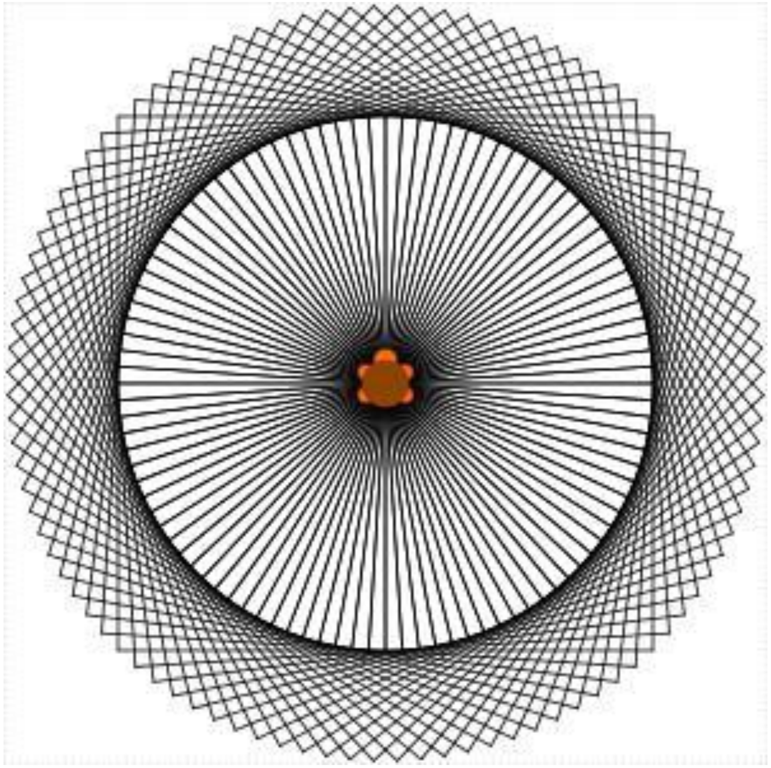
- *Attention, la variable i n'est pas locale à la boucle :*

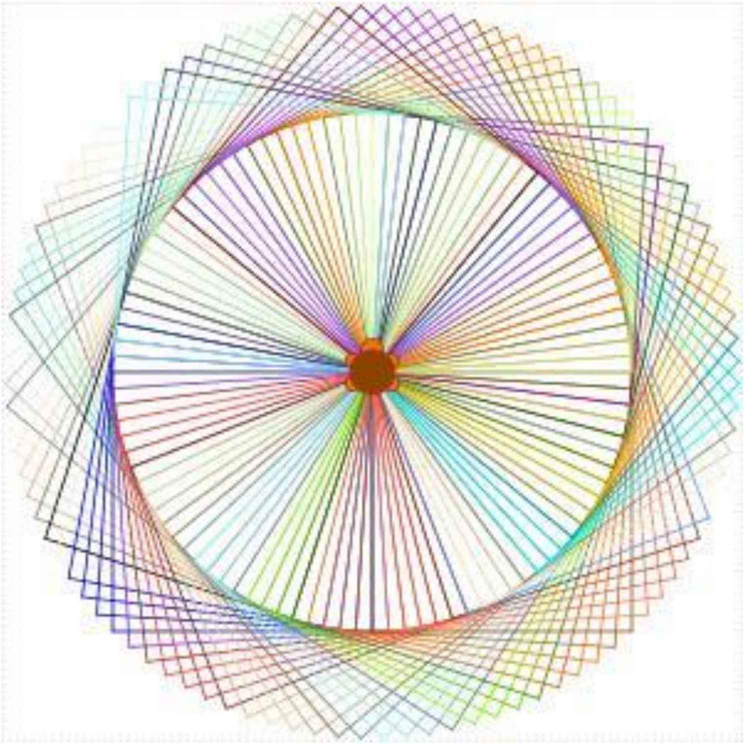
```
> i <- 10
> for(i in seq(4) print(i)
Erreur : symbole inattendu(e) in "for(i in seq(4) print"
> for(i in seq(4)) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
> i
[1] 4
```

• Exemple de dessin obtenu par des carrés en rotation.

```
carre <- function(c) polygone(4,c)
```

```
fleur <- function(n, c) {
   a <- 360 / n
   for(i in seq(n)) {
     carre(c)
     turtle_left(a)
   }
}
```





```
turtle_init()
 turtle_do(fleur(100, 35))
```

• Il est possible mais non obligatoire de **localiser la fonction auxiliaire** carre . Elle ne sera plus utilisable par ailleurs !

Une fonction locale

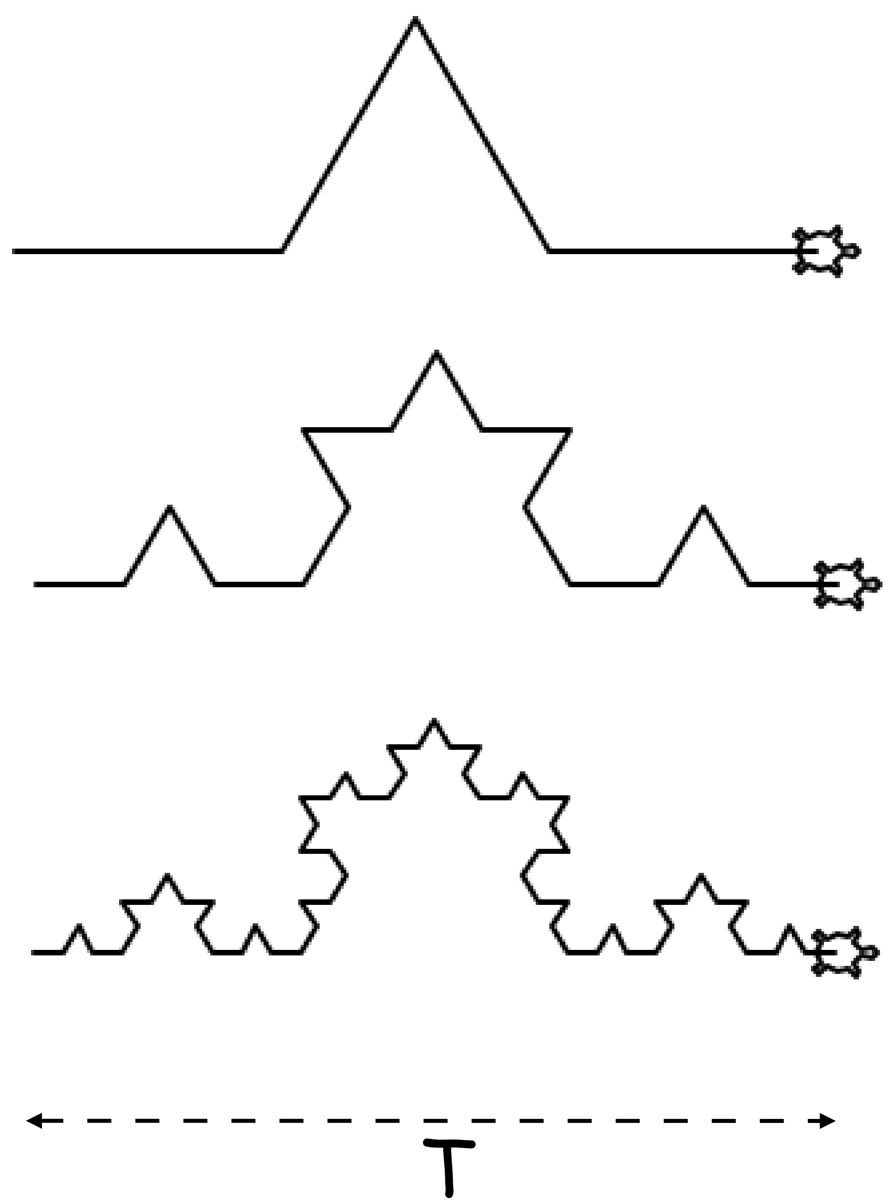
```
fleur <- function(n, c) {
  carre <- function() {polygone(4,c)}
  a <- 360 / n
  cols <- rep_len(colors(TRUE), n)
  for(cl in cols) {
    turtle_col(cl)
    carre()
    turtle_left(a)
  }
}
```

La courbe fractale de Von Koch

• Petite incursion dans la récurrence graphique. La suite (VK_n) des courbes de Von Koch de base T est construite de proche en proche :

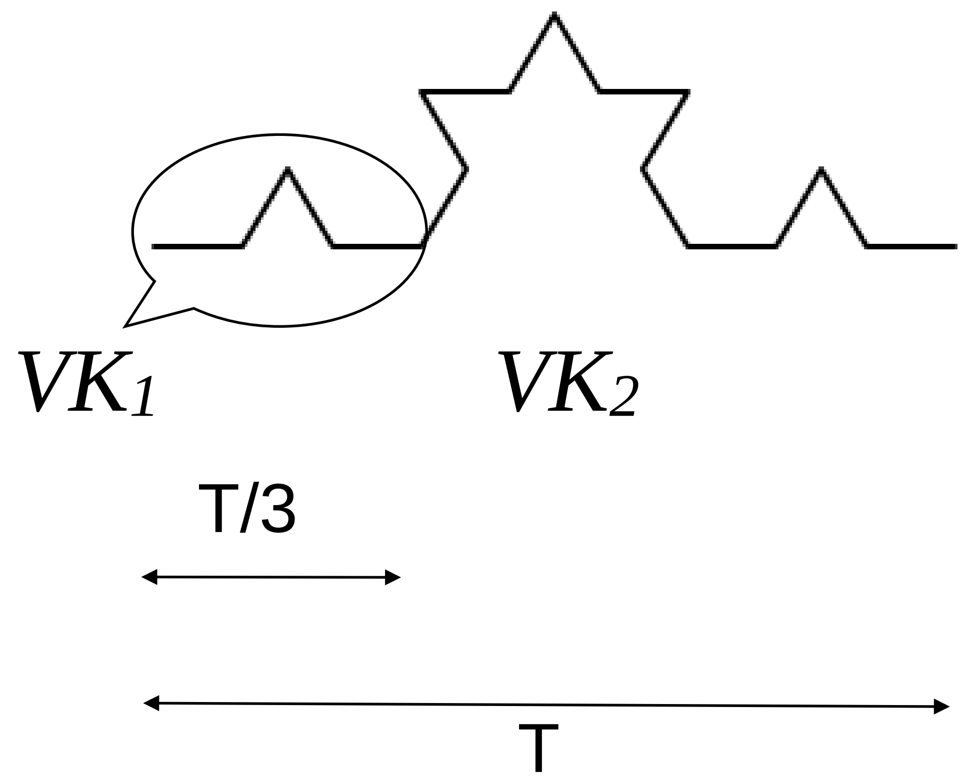
- VK_0 est un segment de longueur T
- VK_1 s'obtient par chirurgie sur VK_0
- VK_2 s'obtient par la même chirurgie sur chaque côté de VK_1
- VK_3 s'obtient par la même chirurgie sur chaque côté de VK_2

etc.



• Mathématiquement, la courbe VK_n s'obtient donc comme assemblage de **quatre** courbes VK_{n-1} . Il s'agit donc d'une RECURRENCE sur n :

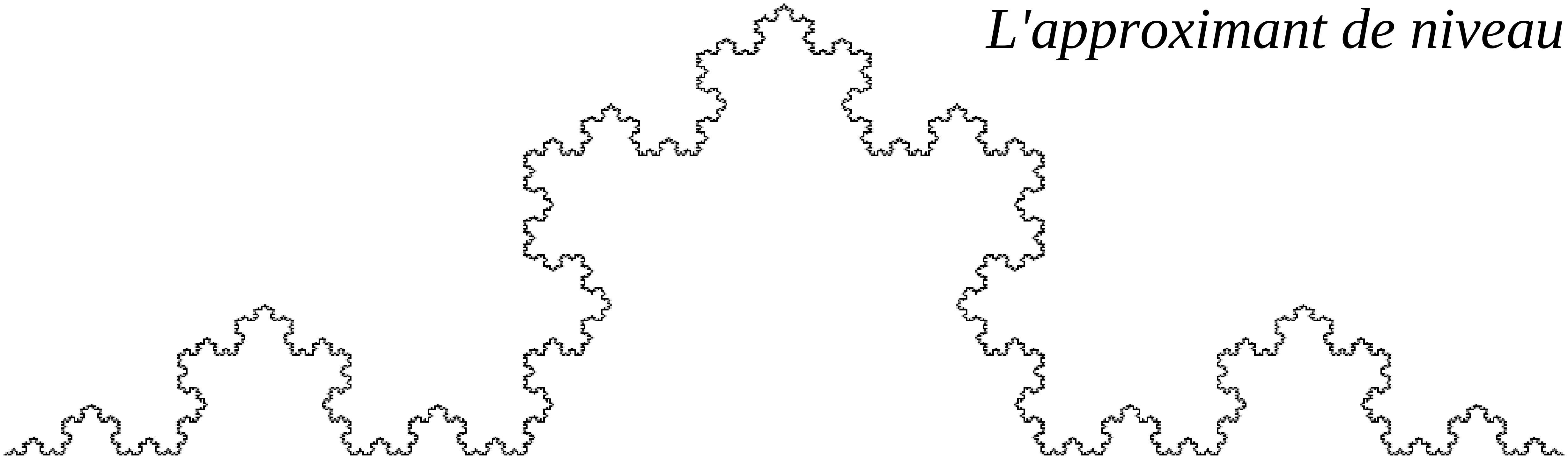
```
VK <- function(n,T) {
   ## approximant de niveau n et de base T
   if (n == 0) turtle_forward(T)
   else {
     VK(n-1,T/3)
     turtle_left(60)
     VK(n-1,T/3)
     turtle_right(120)
     VK(n-1,T/3)
     turtle_left(60)
     VK(n-1,T/3)
   }
 }
```

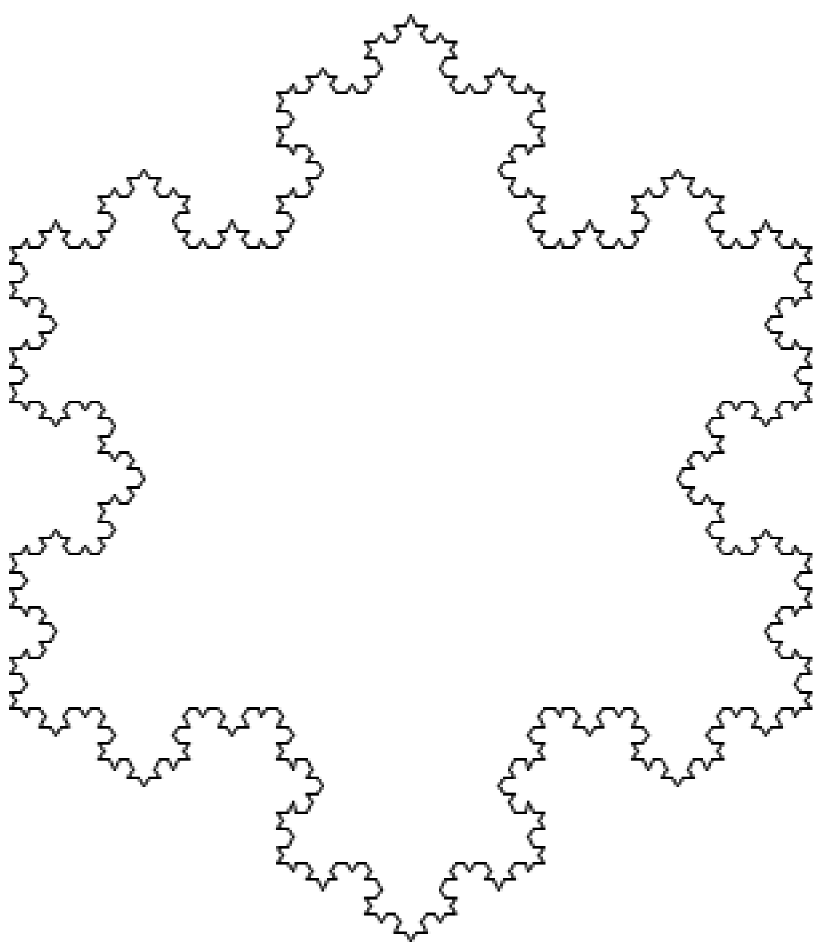


• La courbe de Von Koch VK est la "limite" de la suite : $VK = \lim_{n \to +\infty} VK_n$

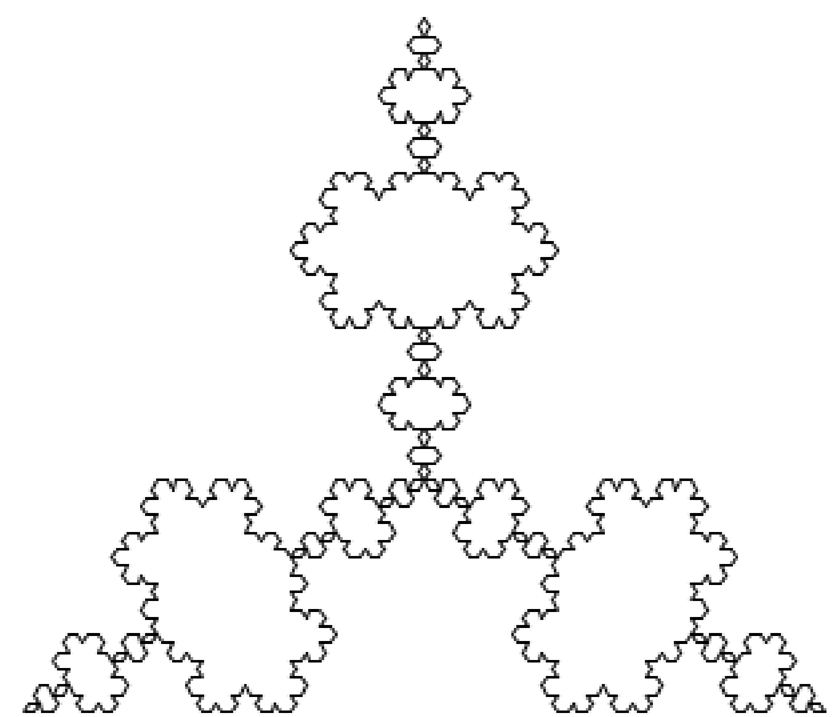
• Découverte en 1906, VK possède d'étranges propriétés. Par exemple, elle est continue mais n'admet de tangente en aucun point !!

L'approximant de niveau 6





Le flocon de Von Koch



L'antiflocon

Les variables locales...

• Jusqu'à présent, dans plusieurs fonctions, nous avons introduit des variables qui n'étaient pas des paramètres de la fonction. Par exemple, dans la fonction fleur ci-contre, la variable i .

```
fleur <- function(n, c) {
   a <- 360 / n
   `...
}
```

• Une telle variable est dite **locale à la fonction**. Elle n'a rien à voir avec une variable de même nom i existant en-dehors de cette fonction !

```
> i = 42
> foo <- function() {i=10;print(i)}
> foo()
[1] 10
> i
[1] 42
```

locale !

...et les variables <u>globales</u>

• Une variable définie en-dehors de toute fonction est **globale**. Pour y faire référence dans une fonction, il faut le déclarer explicitement !

```
> i <- 42                         globale
> foo <- function() {i<-10;print(i)}
> foo()
[1] 10                            locale
> i
[1] 42                            globale
```

• **Les modifications apportées à une variable globale sont locales !**

• Conclusion : **par défaut, les variables introduites dans une fonction sont locales !**

• Pourquoi R a-t-il fait ce choix ? Pour **décourager autant que possible l'utilisation de variables globales** ! Dont acte...