# *Le module Kandinsky*

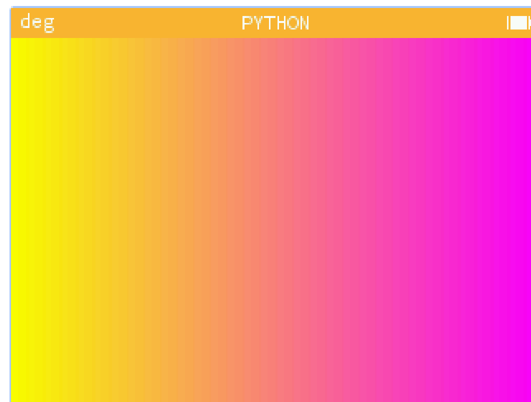
*Correction des exercices :*

J'ai directement écrit et testé ces programmes dans [le Workshop de Numworks] (onglet « Mes scripts »), l'éditeur IDLE de Python que j'utilise habituellement sur mon ordinateur ne pouvant charger le module Kandinsky (celui-ci est sans doute disponible puisque le code-source de Numworks est mis à disposition, mais je vais m'en passer pour l'instant).

1) Écrire un programme qui réalise un dégradé horizontal entre deux couleurs quelconques.

L'idée ici est de couvrir l'écran avec des pixels dont la couleur est identique sur une même colonne verticale (x constant) mais change sur les lignes horizontales (y constant), passant progressivement de c1 à gauche à c2 à droite. Pour obtenir le dégradé, on calcule les différences dr, dg et db entre les composantes rgb de c2 et c1 et on ajoute aux composantes de c1 la part de ces différences correspondant à l'avancement dans la ligne (donc au numéro de la colonne).

```
from kandinsky import *
def couleur():
  dr=c2[0]-c1[0]
  dg=c2[1]-c1[1]
  db=c2[2]-c1[2]
  return color(c1[0]+i*dr//320,
  c1[1]+i*dg//320,
  c1[2]+i*db//320)

c1=(255,255,0)
c2=(255,0,255)
for i in range(320):
  col=couleur()
  for j in range(222):
    set_pixel(i,j,col)
```



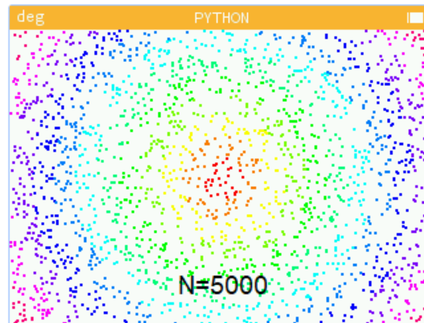

Voici le résultat pour un dégradé entre le jaune et le magenta. Pour prolonger cette idée, puisque la hauteur de l'écran n'est pas utilisée ici (sauf pour étaler la couleur), je vais dégrader verticalement la couleur d'une colonne entre sa valeur actuelle qui sera appliquée sur la ligne du haut et une troisième couleur (pour compléter mon exemple, je prendrai du cyan) qui ne sera atteinte que sur la ligne du bas. J'applique exactement le même principe que précédemment, calculant dans un premier temps la couleur du haut (dégradé entre c1 et c2) dans les variables r, g et b et, dans un second temps, dégradant la couleur obtenue avec c3.

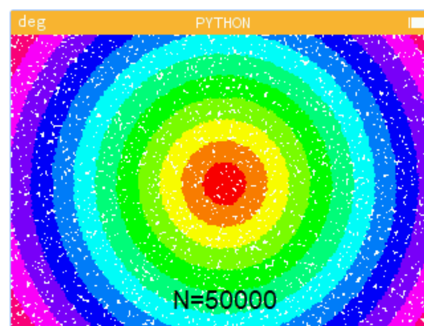
```
1  from kandinsky import *
2  def couleur():
3    r=c1[0]+i*(c2[0]-c1[0])//320
4    g=c1[1]+i*(c2[1]-c1[1])//320
5    b=c1[2]+i*(c2[2]-c1[2])//320
6    r+=j*(c3[0]-r)//320
7    g+=j*(c3[1]-g)//320
8    b+=j*(c3[2]-b)//320
9    return color(r,g,b)
10
11 c1=(255,255,0)
12 c2=(255,0,255)
13 c3=(0,255,255)
14 for i in range(320):
15   for j in range(222):
16     col=couleur()
17     set_pixel(i,j,col)
```

2) Écrire un programme qui place des points aléatoirement dans l'écran, la couleur de ces points étant fonction de la distance au centre de l'écran, le point de coordonnées (160,111).

Voilà un programme simple à mettre en œuvre qui promet un résultat graphiquement intéressant mais dont j'ai du mal à évaluer s'il pourra où non s'exécuter progressivement (les points arrivant les uns après les autres et non tout d'un coup après un long temps d'attente).

```
from kandinsky import *
from random import *
from math import *

def couleur(x,y):
  d=int(sqrt((x-160)**2+(y-111)**2)/16.25)
  return color(c[d][0],c[d][1],c[d][2])

def tirage():
  for i in range(n):
    x=randint(0,320)
    y=randint(0,222)
    c=couleur(x,y)
    set_pixel(x,y,c)
    set_pixel(x+1,y,c)
    set_pixel(x,y+1,c)
    set_pixel(x+1,y+1,c)

n=5000
c=[[255,0,0],[255,127,0],[255,255,0],
[127,255,0],[0,255,0],[0,255,127],
[0,255,255],[0,127,255],[0,0,255],
[127,0,255],[255,0,255],[255,0,127]]
while True:
  tirage()
  y=input()
  if y=='0':break
```



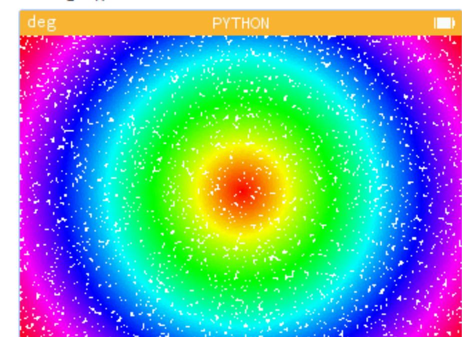

Le programme détermine si la couleur du point appartient à l'une des douze couleurs de notre cercle chromatique. La distance maximum au centre de l'écran étant 195 environ, la division par 16.25 permet de ramener les coordonnées aléatoires à l'une de ces douze couleurs. J'ai dessiné des points plus gros, formés de quatre pixels, pour qu'ils soient plus visibles.

La boucle while de la partie principale est problématique : sur le Workshop elle conduit à un plantage de la machine virtuelle qui ne permet pas la correction (on n'a jamais la main pour remplacer les quatre dernières lignes par l'instruction tirage() qui résoudrait pourtant le problème). Par contre, j'ai envoyé ce programme sur ma calculatrice et il s'exécute correctement : lorsque les 5000 points ont été placés, je tape sur les touches 1 EXE et obtiens 5000 nouveaux points. Je peux alors recommencer ou bien taper 0 EXE, ce qui arrête le programme. Je pense que c'est l'instruction input() qui ne fonctionne pas bien sur la machine virtuelle et comme on n'a jamais accès à la console (ce n'est pas le cas sur la calculatrice), l'erreur n'est pas identifiable.

Pour prolonger ce travail, on peut obtenir un passage progressif d'une couleur à l'autre, un dégradé sans frontière : éliminons les six couleurs intermédiaires qui sont seulement les milieux des dégradés successifs et adaptons le module qui permettait d'obtenir un dégradé dans l'exercice précédent. Le résultat est alors un peu différent.

```
from kandinsky import *
from random import *
from math import *

def degrade(c1,c2,k):
  dr=c2[0]-c1[0]
  dg=c2[1]-c1[1]
  db=c2[2]-c1[2]
  return color(c1[0]+int(k*dr),
  c1[1]+int(k*dg),c1[2]+int(k*db))

def tirage():
  for i in range(n):
    x=randint(0,320)
    y=randint(0,222)
    c=couleur(x,y)
    set_pixel(x,y,c)
    set_pixel(x+1,y,c)
    set_pixel(x,y+1,c)
    set_pixel(x+1,y+1,c)
```

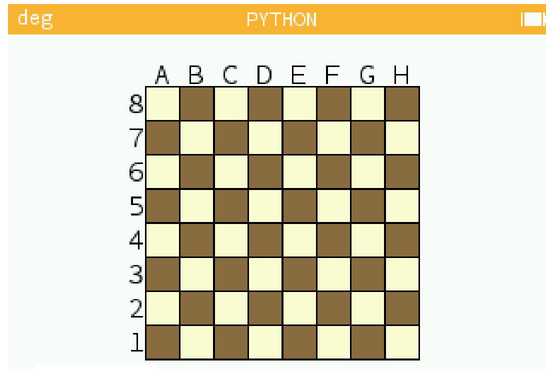
```
def couleur(x,y):
  d=sqrt((x-160)**2+(y-111)**2)/32.5
  return degrade(c[int(d)],
  c[(int(d)+1)%6],d-int(d))

c=[[255,0,0],[255,255,0],[0,255,0],
[0,255,255],[0,0,255],[255,0,255]]
n=50000
tirage()
```



3) Écrire un programme qui trace un échiquier (un quadrillage de 8 lignes et 8 colonnes), les cases étant alternativement blanches et noires, sur fond vert.

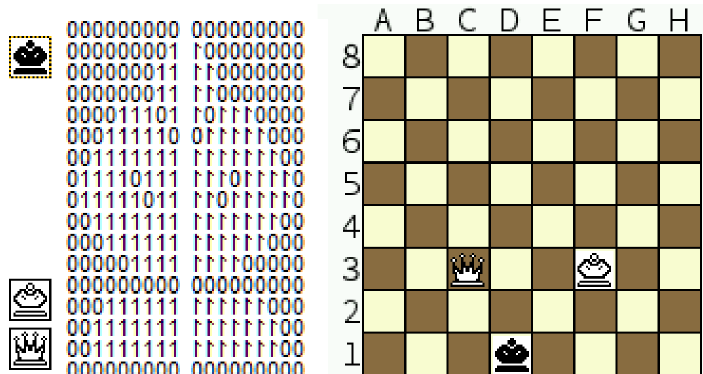
Pas de difficulté majeure ici. Il est bien pratique de définir les courtes fonctions segmentH, segmentV et carre qui permettent de tracer facilement des segments horizontaux, verticaux et des carrés. Les arguments que nécessitent ces fonctions varient selon l'usage spécifique que l'on en fait, mais fondamentalement, elles se ressemblent.

```
from kandinsky import *
def segmentH(x1,x2,y):
  for i in range(x2-x1):
    set_pixel(x1+i,y,noir)
def segmentV(x,y1,y2):
  for i in range(y2-y1):
    set_pixel(x,y1+i,noir)
def carre(x,y,c):
  for i in range(20):
    for j in range(20):
      set_pixel(x+i,y+j,c)

blanc=color(255,255,212)#ivoire
brun=color(139,108,66)#chatain
noir=color(0,0,0)
L=['A','B','C','D','E','F','G','H']
for i in range(8):
  draw_string(L[i],85+i*20,15)
  draw_string(str(8-i),70,32+i*20)
  for j in range(8):
    if i%2==j%2:col=blanc
    else:col=brun
    carre(80+i*20,30+j*20,col)
for i in range(9):
  segmentH(80,240,30+i*20)
  segmentV(80+i*20,30,190)
set_pixel(240,190,noir)#dernier pt
```




J'ai ajouté à l'échiquier les repères qui sont habituellement utilisés pour noter les cases. Maintenant, il ne reste plus qu'à animer ce jeu, par exemple en reconstituant des parties. Pour commencer, on pourrait essayer de représenter la situation d'une finale simple : roi noir contre roi et dame blancs. L'utilisateur joue le roi, la calculatrice joue le roi et la dame. Le roi noir cherche à éviter d'être mat en étant pat (il n'est pas en échec mais ne peut bouger sans s'y mettre comme dans l'exemple ci-dessous). C'est aux blancs de commencer...

Pour vous aider, j'ai essayé de dessiner, dans un carré de 20 sur 20 (ce serait mieux de le faire dans un carré de 19 sur 19), les trois figures engagées dans cette situation de fin de jeu. Il faut maintenant écrire une fonction qui affiche ces dessins sur l'écran (mon image est un montage). Comme on peut se servir de l'instruction input, on doit pouvoir saisir le déplacement du roi noir (par exemple en utilisant les touches 4, 8, 6 et 2 de la calculatrice pour aller vers la la gauche, le haut, la droite et le bas). Quant au déplacement des pièces blanches, un peu d'intelligence artificielle est nécessaire pour coincer ce roi fuyard...

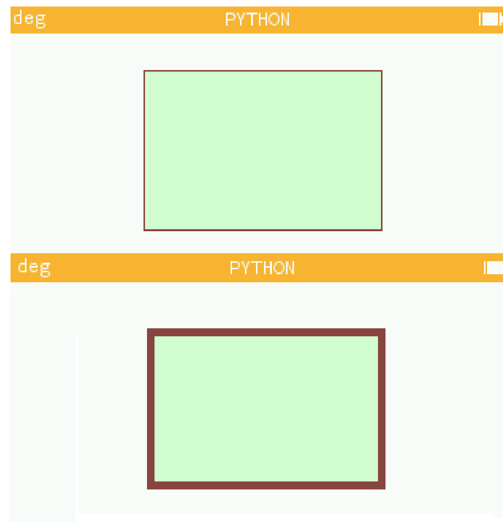



4) Écrire un programme qui trace un rectangle de largeur L et de hauteur h, au centre de l'écran, sa bordure étant d'une couleur c1 et d'une épaisseur e et son intérieur étant d'une couleur c2.

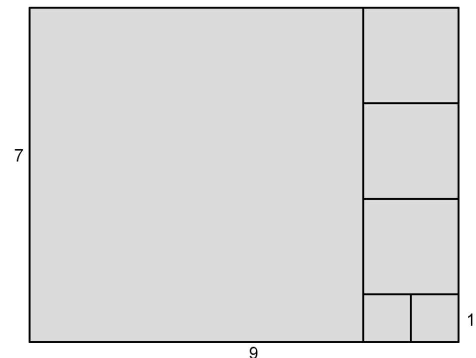
Tracer un rectangle n'est pas difficile, même s'il faut penser à chacun des pixels du bord et de l'intérieur. Comme dans le module Tkinter de Python, on pourrait créer deux fonctions : d'une part

draw_rectangle qui trace juste le bord du rectangle (l'intérieur, transparent, n'écrase pas ce qui y était déjà dessiné) et d'autre part fill-rectangle qui le trace en le remplissant. J'ai opté pour une seule fonction, la demande de remplissage de l'intérieur avec la couleur c2 étant commandée par c2 elle-même.

```
from kandinsky import *
def rectangle(l,h,c1,e,c2):
  x0=160-l//2
  y0=111-h//2
  for i in range(l):
    for j in range(h):
      set_pixel(x0+i,y0+j,c2)
  for i in range (e):
    for j in range(l-2*i):
      set_pixel(x0+j+i,y0+i,c1)
      set_pixel(x0+j+i,y0+h-i,c1)
    for j in range(h-2*i):
      set_pixel(x0+i,y0+j+i,c1)
      set_pixel(x0-i+l,y0+j+i,c1)
    set_pixel(x0+l-i,y0+h-i,c1)

cint=color(210,255,212)#ivoire
cbor=color(139,68,66) #chatain
rectangle(150,100,cbor,5,cint)
```




On peut se demander à quoi peut servir d'afficher un rectangle centré sur une calculatrice. Il y a pourtant des applications, je pense par exemple à un programme qui tracerait, pour deux entiers a et b, un rectangle mesurant a sur b, centré dans l'écran et découpé en carrés, de façon à illustrer la recherche du PGCD par l'algorithme d'Euclide. Sur notre illustration, le rectangle de côtés 9 et 7 est découpé en six carrés : le premier montre que 9=**1**×7+2, les trois suivants montrent que 7=**3**×2+1 et les deux derniers montrent que 2=**2**×1+0, et donc que PGCD(9,7)=1.

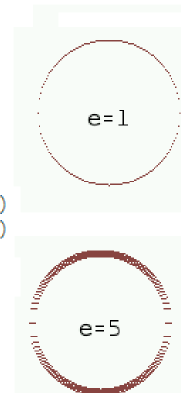



Tracer ce découpage du rectangle en carrés ne serait pas difficile mais un programme n'ayant qu'une application ne présente pas grand intérêt. Il faut un algorithme qui réalise cela quels que soient a et b, avec une mise à l'échelle de façon que le rectangle entre dans l'écran.

5) Écrire un programme qui trace un cercle de centre O(x,y) et de rayon R.

De même que pour le rectangle de l'exercice précédent, on va créer une fonction cercle permettant optionnellement de remplir ou non l'intérieur avec une couleur. D'un point de vue mathématique, un cercle de centre $O(x_O, y_O)$ et de rayon $R$ est l'ensemble des points $M(x,y)$ tels que $(x-x_O)^2+(y-y_O)^2=R^2$, ce qui conduit à $y=y_O\pm\sqrt{R^2-(x-x_O)^2}$.
Transformée en programme, cette formule trace un cercle avec des trous sur les bords, car il y a trop peu de pixels utilisés lorsque $x$ s'approche de sa valeur minimale $x_O-R$ ou de sa valeur maximale $x_O+R$.
Ce n'est pas satisfaisant, nous voulons un cercle sans trou. Une idée simple à mettre en œuvre est de tracer de cette façon les deux quarts de cercle supérieurs et inférieurs (en termes géographiques NO-NE et SE-SO et, en termes trigonométrique, de $\frac{\pi}{4}rad$ à $\frac{3\pi}{4}rad$ et de de $\frac{-3\pi}{4}rad$ à $\frac{-\pi}{4}rad$ ) et, pour les deux autres quarts de cercle, de faire subir aux deux premiers un quart de tour direct. Pour cela, il suffit de se rappeler des formules de trigonométrie :

```
from kandinsky import *
from math import *
def cercle(x0,y0,r,c1,e,c2):
  for i in range(e):
    x1=x0-r+i
    x2=x0+r-i
    for x in range(x1,x2+1):
      y1=int(y0+sqrt((r-i)**2-(x-x0)**2))
      y2=int(y0-sqrt((r-i)**2-(x-x0)**2))
      set_pixel(x,y1,c1)
      set_pixel(x,y2,c1)
  draw_string('e='+str(e),x0-15,y0-5)

cint=color(210,255,212)#ivoire
cbor=color(139,68,66) #chatain
cercle(160,111,100,cbor,5,cint)
```

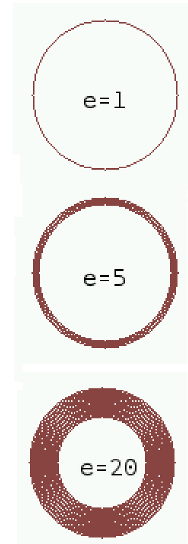


$\cos(\alpha+\frac{\pi}{2})=-\sin(\alpha)$ et $\sin(\alpha+\frac{\pi}{2})=\cos(\alpha)$.

Cela donne de bien meilleurs cercles, sans trou, même s'il subsiste quelques pixels blancs dans le bord. Ce défaut ne se remarque que lorsque le bord prend de l'épaisseur, je suppose que les arrondis effectués par la fonction int() conduisent les pixels de deux cercles voisins à se superposer alors qu'ils ne devraient pas.

La solution n'est toujours pas satisfaisante car je souhaite un cercle sans trou dans son bord. L'amélioration ultime vient de cet aménagement : je double le nombre e de cercles qui crée l'épaisseur et je diviser par deux la diminution du rayon à chaque étape. De cette façon, le problème des pixels qui se superposent, laissant des trous, ne se retrouve plus.

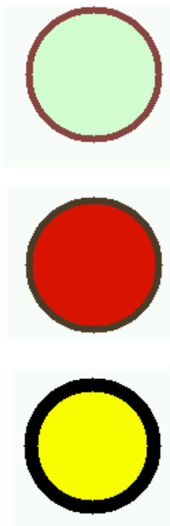
```
from kandinsky import *
from math import *
def cercle(x0,y0,r,c1,e,c2):
  for i in range(e):
    xd=x0-int((r-i)/sqrt(2))
    xf=x0+int((r-i)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i)**2-(x-x0)**2))
      set_pixel(x,y1,c1)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c1)
        x1,y1=x2,y2
  draw_string('e='+str(e),x0-15,y0-5)

cint=color(210,255,212)#ivoire
cbor=color(139,68,66) #chatain
cercle(160,111,50,cbor,1,cint)
```

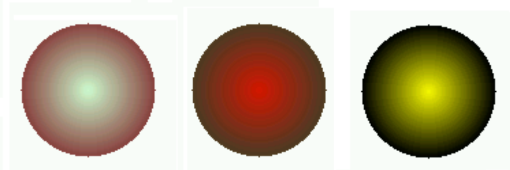



Je profite de cette forme d'achèvement pour implémenter le remplissage de l'intérieur : il suffit de créer la fonction cercle_plein qui envoie deux exécutions de la fonction cercle, la première pour tracer le bord et la seconde pour remplir l'espace restant qui est considéré comme le bord d'un cercle dont l'épaisseur est égale au rayon.

J'ai envie de terminer sur la construction d'un dégradé entre la couleur du bord et celle de l'intérieur. Utilisant le principe du dégradé entre deux couleurs mis au point un peu plus haut, la fonction cercle_grade réalise cela.

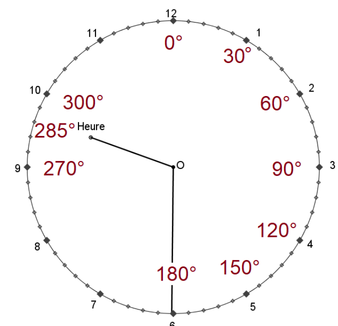
```
from kandinsky import *
from math import *
def cercle(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def cercle_plein(x0,y0,r,c1,e,c2):
  cercle(x0,y0,r,c1,e)
  cercle(x0,y0,r-e,c2,r-e)
  #draw_string('e='+str(e),x0-15,y0-7)

cint=color(219,23,2) #cinabre
cbor=color(78,61,40) #bitume
cercle_plein(160,111,50,cbor,5,cint)
```





```
def cercle_grade(x0,y0,R,c1,e,c2):
  for i in range(R):
    r=c1[0]+i*(c2[0]-c1[0])//R
    g=c1[1]+i*(c2[1]-c1[1])//R
    b=c1[2]+i*(c2[2]-c1[2])//R
    cercle(x0,y0,i,color(r,g,b),1)

cExt=[219,23,2]
cInt=[78,61,40]
cercle_grade(160,111,50,cExt,10,cInt)
```

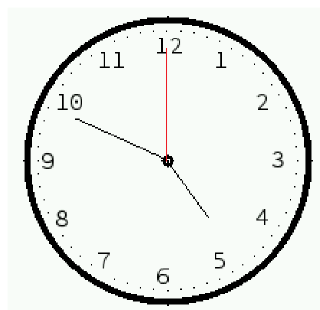
6) Écrire un programme qui trace une horloge dont les deux (ou trois) aiguilles indiquent une heure donnée à la minute près (ou à la seconde près).




L'horloge est dessinée par un cercle, les douze points marqués des nombres de 1 à 12, le point central et les aiguilles – des segments issus du centre – qui indiquent l'heure. Pour se faire, une fonction heure transforme l'heure donnée en deux angles (on étudiera le cas des secondes plus tard). Par exemple, l'heure 9h30 doit être transformée en deux angles : 285° (pour les heures) et 180° pour les minutes. Mais ces angles doivent être transformés en coordonnées cartésiennes des points H et M pour tracer les segments [OH] et [OM] qui matérialisent les aiguilles.

Il faut confier les angles à une fonction coordonnees qui calcule les coordonnées de H et M et ensuite envoyer tout cela à une fonction segment qui trace les aiguilles.
Le programme ci-dessous donne une solution qui fonctionne. Bien sûr, elle peut être améliorée. J'ai essayé d'obtenir un mouvement des aiguille à peu près correct : les segments qui matérialisent les aiguilles doivent être effacés avant d'être retracé un peu plus loin, ou bien sur eux même dans le cas où ils auraient été effacés par le passage d'une autre aiguille. Ce programme est le plus long de la série du fait de cette animation. Le problème est que l'on ne peut pas tester ce genre de programme dans le Workshop (il exécute tout avant de donner le résultat final). Il faut le télécharger dans sa machine (avant d'effacer d'autres programmes si il y a un manque de place, enregistrer vos programmes dans le Workshop !). Pour la figure de l'illustration, j'ai ajouté la trotteuse à la main (dans le Workshop elle n'est pas visible à la fin du processus).

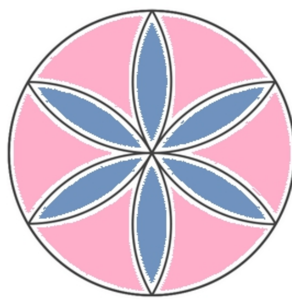
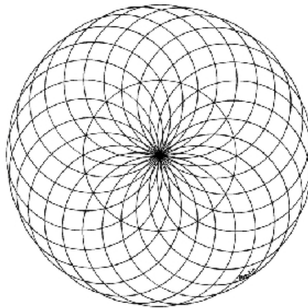
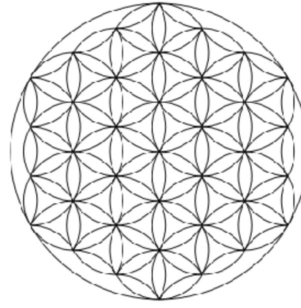
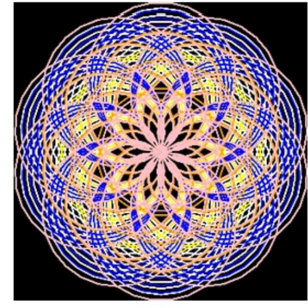
```
from kandinsky import *
from math import *
def cercle(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def seg(xa,ya,xb,yb,c):
  if abs(yb-ya)<abs(xb-xa):
    if xb<xa:
      xa,xb=xb,xa
      ya,yb=yb,ya
    m=(yb-ya)/(xb-xa)
    p=ya-m*xa
    for i in range(xb-xa):
      set_pixel(int(xa+i),
      int(m*(xa+i)+p),c)
  else:
    if yb<ya:
      ya,yb=yb,ya
      xa,xb=xb,xa
    m=(xb-xa)/(yb-ya)
    p=xa-m*ya
    for i in range(yb-ya):
      set_pixel(int(m*(ya+i)+p),
      int(ya+i),c)

def coord(h,r):
  t=pi*(h/6+1.5)
  return [int(r*cos(t)+160),
  int(r*sin(t)+111)]

def heure(hh,mm,r):
  hm=(hh*60+mm)/60
  return coord(hm,r)
```




horloge(16,53,180)

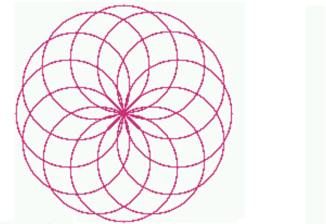
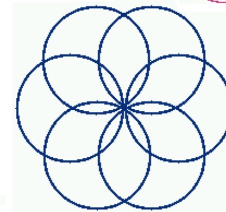
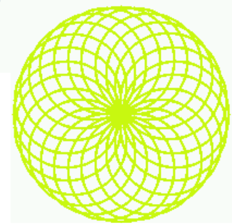
```
def cadran(x0,y0,r,c1,e,c2):
  cercle(x0,y0,r,c1,e)
  cercle(x0,y0,4,c1,2)
  for j in range(60):
    t=pi/30*j
    set_pixel(int((90)*cos(t)+160),
  int((90)*sin(t)+111),c1)
  for i in range(12):
    h=heure(i+1,0,80)
    draw_string(str(i+1),h[0]-8,h[1]-8)

def temporise(c):
  for i in range(7800):
    c=(c**2)%12345

def aiguilles(t,c):
  h=heure(t[0],t[1],50)
  seg(160,111,h[0],h[1],c)
  h=heure(t[1]/5,0,70)
  seg(160,111,h[0],h[1],c)

def horloge(h,m,t=0):
  time=[h,m]
  c1=color(0,0,0)       #noir
  c2=color(240,11,20)   #trotteuse
  c3=color(255,255,255)#blanc
  cadran(160,111,100,c1,4,c2)
  aiguilles(time,c1)
  s=0
  while s<t:
    h=heure((s)%60/5,0,80)#trotteuse
    seg(160,111,h[0],h[1],c2)#trace
    temporise(98765)
    s=s+1
    seg(160,111,h[0],h[1],c3)
    aiguilles(time,c3)#efface aiguilles
    if s%60==0:
      time[1]+=1
      if time[1]==60:
        time[1]=0
        time[0]=(time[0]+1)%12
    h=heure(time[0],time[1],50)
    aiguilles(time,c1)#trace aiguilles
    cadran(160,111,100,c1,4,c2)
```

7) Écrire un programme qui trace une de ces rosaces qui ne sont constituées que de cercles (les deux premières sont classiques, la 3ème est appelée « fleur de vie » et la dernière, trouvée sur internet, y est appelée spirale de Georgia).

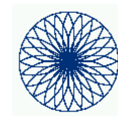
Commençons par le 2ème de ces rosaces qui me semble plus simple que les autres. Je vais mettre le nombre de cercles tracés en paramètres : l'idée est de tracer n cercles de rayon r, les centres de ces cercles étant régulièrement disposés sur un cercle de centre O(160,111) – le centre de l'écran – et de rayon r. On va réutiliser une nouvelle fois notre fonction cercle et calculer les coordonnées de chaque centre en appliquant une rotation de centre O et d'angle $\frac{2\pi}{n}$ . Le résultat est tout de suite satisfaisant.

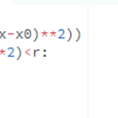
```
from kandinsky import *
from math import *
def cercle(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def rosace(n,r,c,e):
  x,y=160+r,111
  for i in range(n):
    x1=int(160+r*cos(i*2*pi/n))
    y1=int(111+r*sin(i*2*pi/n))
    cercle(x1,y1,r,c,e)

cint=color(205,50,123)
cbor=color(0,0,0)
rosace(12,50,cint,1)
```

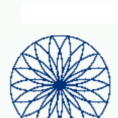




Pour obtenir la 1ère rosace, on peut se contenter de glisser dans la fonction rosace un test qui examine si le point à tracer est à l'intérieur du cercle de centre O et de rayon r (le cercle sur lequel sont placés les centres des arcs que l'on souhaite garder). En implémentant cette méthode rosace1, je m'aperçoit que la rosace obtenue à des pétales entiers pour n=3, 6, 9, 12, …, 18, 24, 30 soit les multiples de 3, et les pétales sont tronqués pour les autres valeurs de n (sur l'illustration de la ligne du bas, n=4, 5, 7, 8).

```
from kandinsky import *
from math import *
def cercle1(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      if sqrt((160-x1)**2+(111-y1)**2)<r:
        set_pixel(x1,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        if sqrt((160-x2)**2+(111-y2)**2)<r:
          set_pixel(x2,y2,c)
        x1,y1=x2,y2

def rosace1(n,r,c,e):
  x,y=160+r,111
  for i in range(n):
```



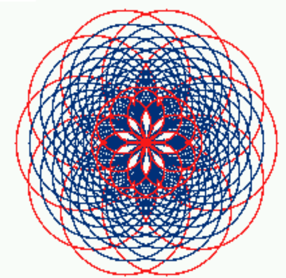
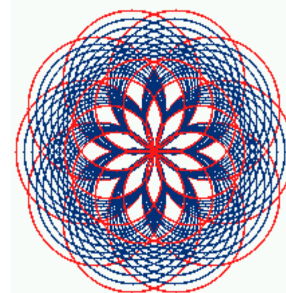




La rosace n°3 me paraît relativement proche, sur le principe, des rosaces déjà tracées. Cela ne devrait pas poser trop de difficultés, mais une forme de paresse me retient de l'implémenter. Peut-être qu'un d'entre vous aura ce courage...

La dernière est plus complexe encore mais devrait poser

```
from kandinsky import *
from math import *
def cercle2(x0,y0,r,c,e):
  for i in range(2*e):
    xd=x0-int((r-i*0.5)/sqrt(2))
    xf=x0+int((r-i*0.5)/sqrt(2))
    for x in range(xd,xf+1):
      x1=x
      y1=y0+int(sqrt((r-i*0.5)**2-(x-x0)**2))
      set_pixel(x,y1,c)
      for j in range(3):
        x2=x0+y1-y0
        y2=y0+x0-x1
        set_pixel(x2,y2,c)
        x1,y1=x2,y2

def rosace3(n,r,c1,c2,e):
  rj=r
  for j in range(n-2):
    rj=int(rj-rj/n)
    for i in range(n):
      x1=int(160+rj*cos(i*2*pi/n))
      y1=int(111+rj*sin(i*2*pi/n))
      if j==0 or j>n-4:col=c2
      else:col=c1
      cercle2(x1,y1,rj,col,e)

col1=color(5,50,120)
col2=color(255,25,25)
rosace3(10,55,col1,col2,1)
```

```
def rosace3(n,r,c1,c2,e):
  rj=r
  for j in range(n-2):
    rj=int(rj-rj/(2*n))
    x,y=160+rj,111
```

moins de difficultés que la précédente car les cercles sont tracés en entier. Je ne suis pas très satisfait du résultat obtenu mais, le nombre de pixels étant assez limité, je ne pense pas arriver à autant de détails que sur l'image de l'énoncé.
J'ai essayé plusieurs solutions pour déterminer les rayons progressivement décroissant des cercles, mais je n'ai pas encore obtenu un effet semblable à la spirale de Georgia qui contient plus de deux couleurs et qui est composée par davantage de cercles. Le principe reste cependant assez voisin de celui-ci.