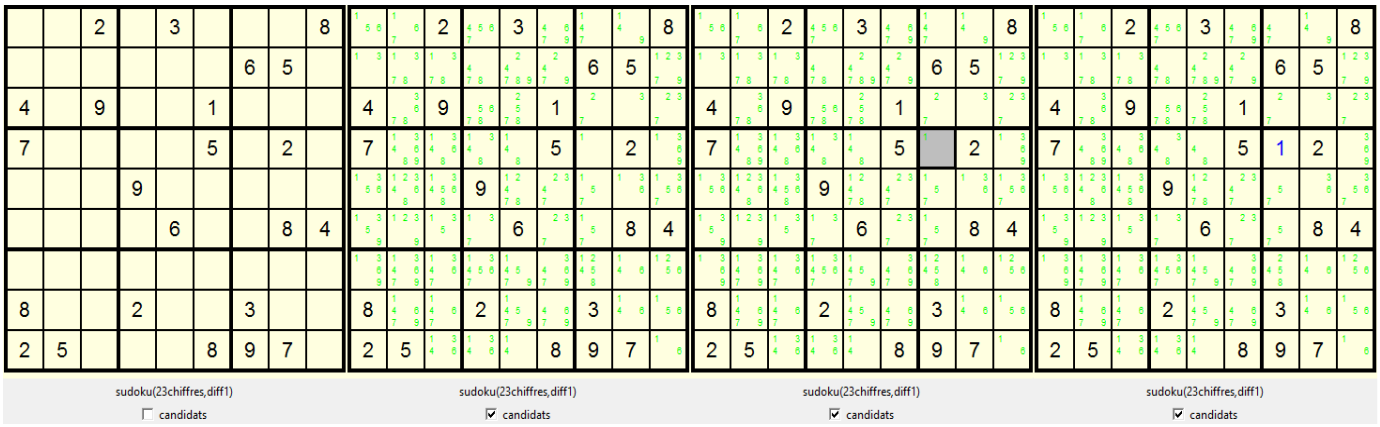
Partie n°6 : Interactivité

Nous avons déjà rencontré quelques situations interactives – où l'utilisateur peut converser avec le programme – quand nous avons utilisé le module *tkinter* et découvert quelques uns de ses nombreux dispositifs graphiques. Mais l'interactivité est également présente dans un simple programme qui demanderait à l'opérateur d'entrer son nom jusqu'à ce que le nom entré ne contienne aucun chiffre. Cela pour dire que la notion d'interactivité recouvre des fonctionnalités très diverses, pas forcément compliquées, qui permettent de moduler les actions du programme.

a) Pour continuer avec les sudokus, nous aimerions disposer d'un programme qui affiche une grille initiale et offre la possibilité de la compléter avec des chiffres entrés au clavier. La fenêtre d'affichage doit être rendue sensible (la méthode *bind*() de la classe *Tk* réalise cela) et réagir aux entrées du clavier en lançant un évènement (*KeyPress*) qui permet à la commande *ajoute*() de se servir du chiffre entré. Le programme doit reconnaître une grille complète et valide en affichant le message « Bravo ! La solution a été trouvée en … secondes »). On peut prévoir aussi une case à cocher *Checkbutton*() qui, lorsqu'elle est cochée, déclenche l'affichage des chiffres « candidats » dans les cases vides.

Pour commencer, nous reprenons notre classe *Grille* avec ses deux attributs *grille* et *possible* ainsi que la procédure de lecture d'un fichier *sudoku.txt*. La classe *Grille* doit s'enrichir de quelques méthodes qui permettront de placer les nouveaux chiffres dans les cases. Placer un chiffre dans une case se fera par *setCase*() qui appelle *placeElement*() si le chiffre appartient à {1,2,3,4,5,6,7,8,9} et *placeZero*() si le chiffre est 0 – ce qui équivaut à un effacement du chiffre de la case. La méthode *affiche*() doit être enrichie pour tenir compte des options d'affichage : lorsqu'on clique dans une case, on peut montrer que la case a été sélectionnée en la colorant en gris ; lorsque le chiffre entré est invalide, on peut l'afficher en rouge alors qu'il sera affiché en bleu s'il est valide et en noir s'il fait partie de la grille initiale ; lorsque l'option « candidats » a été cochée, on doit afficher les chiffres candidats en petit (selon une grille 3×3).



L'interactivité principale est l'entrée des chiffres dans la grille. Elle est déclenchée par l'instruction fen.*bind*("<KeyPress>", *modifie*) où *ajoute*() est la fonction qui est lancée lorsqu'on presse une touche du clavier. La fonction *ajoute*() prend l'évènement *event* en argument, et l'utilise par l'instruction touche=*event.keysym* qui a pour effet d'affecter à la variable *touche* le caractère entré (sous la forme d'une chaîne, par exemple '3' si on presse la touche 3 et 's' si on presse la touche s). On pourrait effectuer un contrôle sur le caractère entré (pour savoir s'il s'agit bien d'un chiffre) mais nous considérons que l'utilisateur sait ce qu'il fait... Avant d'entrer le chiffre, l'utilisateur doit sélectionner la case de la grille qu'il veut compléter : cela se fait par l'ajout de cette instruction qui sensibilise le programme aux clics de souris cadre.*bind*("<*Button-1*>", *surligne*) où *surligne*() est la fonction qui est lancée lorsqu'on clique avec le bouton gauche de la souris (*Button-1*). La fonction *surligne*() prend l'évènement *event* en argument, et en utilise les attributs de coordonnées par les instructions xcurseur=*event.x* et ycurseur=*event.y*. Les coordonnées servent à déterminer quelle case de la grille a été sélectionnée (pour éviter les erreurs d'exécution si on clique en dehors de la grille, nous avons disposée les instructions dans un bloc *try*) et à lancer la méthode *surligneCase*() de la classe *Grille* (si la case était bien vide dans la grille initiale) puis le réaffichage de la grille. L'interaction optionnelle d'affichage des chiffres candidats, assurée par le cochement de la case, est réalisée par les instructions choix=*IntVar*() et aide=*Checkbutton*(*fen*, *text="candidats", variable=choix,command=affiche*) où *affiche*() est la fonction qui est lancée lorsqu'on coche ou décoche la case *aide*. Cette fonction se contente d'effacer la grille et de lancer la méthode *affiche*() de la classe *Grille*, où on a pris soin d'utiliser la valeur du paramètre *choix* qui vaut 0 quand la

case est désélectionnée et 1 lorsqu'elle est sélectionnée. La grille est donc ré-affichée avec la bonne valeur du paramètre. Voilà pour l'essentiel ! Il suffit, pour maintenir l'affichage des candidats à jour, de relancer cet affichage à chaque modification de la grille.

```
from tkinter import *
class Grille : # Classe pour Sudoku. basée sur sudoku3
    global solutions,choix, label
    def __init__(self,g):
        self.nChiffres=0
        self.grille=g
        self.surligned=81
        self.invalide=81
        self.possible=[]#contient la liste des possibilités pour une case
        for i in range(81):
            if self.grille[i]==0:self.possible.append(9*[True])
            else:self.possible.append(9*[False])
        for i in range(81):
            if self.grille[i]!=0:
                self.placeElement(i,self.grille[i])#print(self.possible)
    def surligneCase(self,c):
        self.surligned=c
    def isValide(self,g):
        for i in range(81):
            for j in range(i+1,81):
                if self.meme(i,j) and g[i]!=0 and g[i]==g[j]: return False
        return True
    def setInvalide(self,n):
        self.invalide=n
    def setCase(self,i,n):
        if n==0: self.placeZero(i)
        else: self.placeElement(i,n)
        self.surligned=81
    def meme(self,i,j):
        return (i-j)%9==0 or i//9==j//9 \
               or 3*(i//27)+(i%9)//3==3*(j//27)+(j%9)//3
    def placeZero(self,c):
        self.possible=[]
        self.grille[c]=0
        self.nChiffres=0
        for i in range(81):
            if self.grille[i]==0:self.possible.append(9*[True])
            else:self.possible.append(9*[False])
        for i in range(81):
            if self.grille[i]!=0:
                self.placeElement(i,self.grille[i])
    def placeElement(self,c,n):
        self.grille[c]=n
        self.nChiffres+=1
        for i in range(81):#mise à jour des possibles dans
            if i%9==c%9: self.possible[i][n-1]=False # la même colonne
            if i//9==c//9: self.possible[i][n-1]=False # la même ligne
            if 3*(i//27)+(i%9)//3==3*(c//27)+(c%9)//3: self.possible[i][n-1]=False # le même bloc
        for i in range(1,10):self.possible[c][i-1]=False #mise à jour dans la case
    def affiche(self):
        for i in range(1,9): #affiche les grosses lignes de la grille
            cadre.create_line(10,10+i*40,370,10+i*40,width=2,fill='black')
            cadre.create_line(10+i*40,10,10+i*40,370,width=2,fill='black')
        for i in range(4):   #affiche les grosses lignes de la grille
            cadre.create_line(8,10+i*120,372,10+i*120,width=4,fill='black')
            cadre.create_line(10+i*120,10,10+i*120,370,width=4,fill='black')
        if self.surligned!=81 :
            c=self.surligned
            cadre.create_rectangle(11+c%9*40,11+c//9*40,48+c%9*40,48+c//9*40,width=1,fill='grey')
        for i in range(81):
            couleur="black"
            if self.grille[i]!=grille[i]: couleur="blue"
            if self.invalide==i: couleur="red"
            if self.grille[i]!=0: #affiche les chiffres dans la grille
                cadre.create_text(30+i%9*40,30+i//9*40,text=str(self.grille[i]),\
                                  font="Arial 16",fill=couleur)
        if choix.get()==1:
            for i in range(81):
                for j in range(9):
                    if self.possible[i][j]:
                        cadre.create_text(17+i%9*40+j%3*12,19+i//9*40+j//3*12,text=str(j+1),font="Arial 7",fill="green")
```

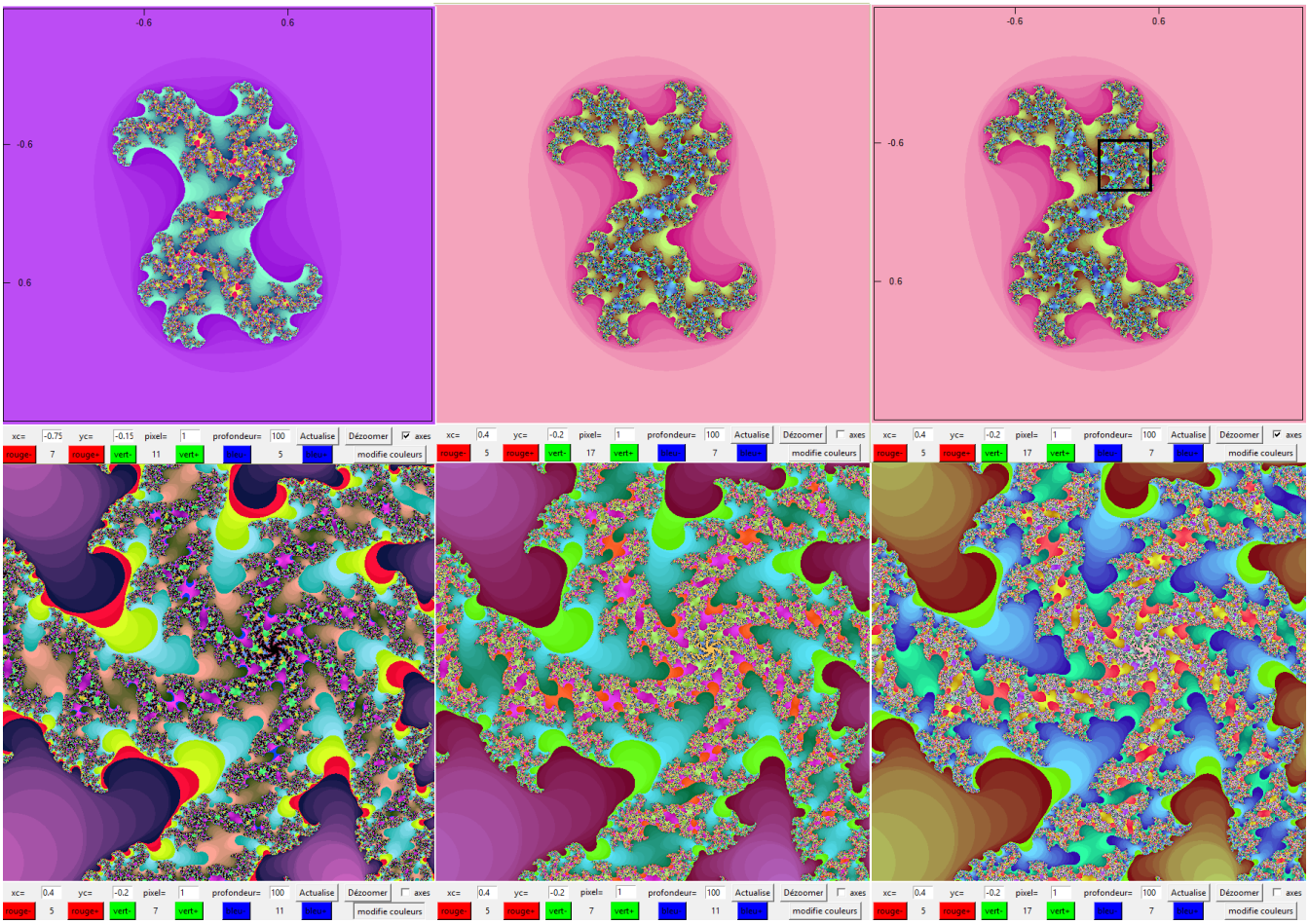
```
from time import clock
from os import getcwd,chdir
def affiche():
    cadre.delete(ALL)
    probleme.affiche()
def surligne(event):
    global probleme,case
    xcurseur=event.x
    ycurseur=event.y
    case=(xcurseur-10)//40+9*((ycurseur-10)//4
    if case in range(81) and grille[case]==0:
        probleme.surligneCase(case)
        cadre.delete(ALL)
        probleme.affiche()
def ajoute(event):
    global probleme,case
    t=event.keysym
    try:
        touche=int(t)
        if touche in range(10) and grille[case]==0:
            probleme.setCase(case,touche)
            cadre.delete(ALL)
            controle(case)
            probleme.affiche()
    except:None
def controle(c=81):
    global probleme,t,label
    if probleme.grille.count(0)==0:
        if probleme.isValide(probleme.grille) :
            t-=clock()
            print('Bravo! La solution a été trouvée en {} secondes.'.format(-t))
            label.configure(text="Bravo! Problème résolu en {} s!".format(int(10
        else:print('Il y a un petit problème, la solution trouvée est incorecte!
    else:
        if probleme.isValide(probleme.grille) :
            print('Il reste {} chiffres à trouver.'.format(81-probleme.nChiffres
            probleme.setInvalide(81)
        else:
            print('La grille n\'est pas valide.')
            probleme.setInvalide(c)


chdir(getcwd())
fen=Tk()
nomSudoku='sudoku(23chiffres,diff1)'
cadre=Canvas(fen,width=376,height=376,bg='light yellow')
cadre.pack()
label=Label(fen,text=nomSudoku)
label.pack()
choix=IntVar()
aide=Checkbutton(fen,text="candidats",variable=choix,command=affiche)
aide.pack()
grille,solutions,case=[],[],81
fichierSudoku=open(nomSudoku+'.txt','r')
for ligne in fichierSudoku.readlines() :
    if ligne[0]==' ': break
    grille+=ligne.split(' ')
    grille[-1]=grille[-1][:1]#pour enlever le \n de fin de ligne
fichierSudoku.close()
grille=[int(a) for a in grille]
probleme=Grille([int(a) for a in grille])
controle()
probleme.affiche()
t=clock()
cadre.bind("<Button-1>",surligne)
fen.bind("<KeyPress>",ajoute)
fen.mainloop()
```

Ce programme fonctionne à peu près comme nous le souhaitions. Ce n'est sans doute pas le mieux que l'on puisse faire. Il faudrait songer à implémenter d'autres fonctionnalités interactives telles que : proposer de résoudre d'autres grilles (en cherchant parmi les grilles présentes dans un dossier), afficher un chronomètre, proposer la solution, ou une option qui corrige les erreurs (si un chiffre entré ne correspond pas à la solution), etc. Ce faisant, nous avons un peu progressé en utilisant une case à cocher et en rendant le programme sensible à deux types d'évènements différents (un clic gauche de la souris et une saisie au clavier).

Un autre dispositif de programmation a été utilisé pour la 1ère fois : le bloc *try* qui a une syntaxe simple constitué des deux lignes consécutives : *try* :< instructions> et *except* :< instructions>. On essaie de faire quelque chose dans le bloc *try*, et si ce quelque chose conduit à une erreur, on n'arrête pas le programme

pour autant, mais on exécute ce qui suit la commande *except*. Ici, la conversion de la touche pressée avec *int*() peut échouer (si la touche est une lettre, ou bien la touche *Enter*, ou *Alt Gr* ou encore une autre touche) et conduire à une erreur. Dans notre cas, on ne fait rien si une erreur arrive, mais dans d'autres situations, on pourrait intercepter le type d'erreur et l'afficher ou lancer d'autres traitements spécifiques de l'erreur. La commande complète serait *except type_de_l_exception as* type : qui conduirait à mettre le type de l'erreur (NameError, TypeError, ZeroDivisionError, ValueError, etc.) dans la variable *type*. En l'absence du bloc *try*, notre programme génèrerait une erreur de type *ValueError* en cas de saisie de la touce *Enter* avec le message suivant : *invalid literal for int() with base 10: 'Return'*.

b) Changeons de sujet et faisons un pas de plus dans le monde merveilleux des fractales ! Les ensembles de Julia sont des images fractales construites selon un procédé assez simple. À chaque pixel P de coordonnées $(x;y)$ nous allons associer une couleur $coul(r,b,v)$ selon la 1ère valeur de l'entier n pour laquelle l'image P_n du point P s'écarte de l'origine $O(0;0)$ du repère d'une différence supérieure ou égale à 2. Pour trouver les coordonnées $(x';y')$ de l'image P_n, à partir de celles $(x;y)$ de P_{n-1} on applique les relations : $x'=x^2-y^2+x_I$ et $y'=2xy+y_I$ où $(x_I\,;y_I)$ sont les coordonnées d'un point I du plan. Lorsque le point P_n ne sort pas (en essayant les valeurs de n jusqu'à une certaine valeur maximum appelée *prof* pour profondeur) du disque de rayon 2 centré sur O, la couleur du point est noire, sinon la couleur est définie par un choix arbitraire, par exemple le dégradé de vert $coul(0,\frac{(prof-n)\times 255}{prof},0)$. En procédant ainsi, on obtient l'image de l'ensemble de Julia associé au point I. Programmer l'affichage d'une telle image pour x et y compris entre -2 et 2, avec la possibilité de modifier les coordonnées de I et la possibilité de zoomer sur une partie de cette image (par exemple en cliquant dans le cadre deux extrémités de la nouvelle fenêtre en diagonale).

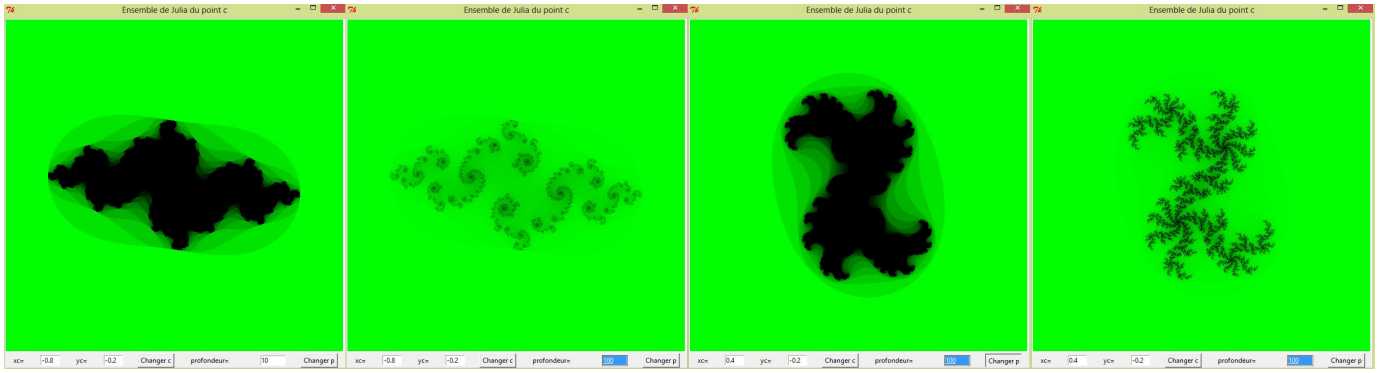


Disons le tout de suite, nous avons délibérément choisi de ne pas parler de nombres complexes pour rester abordable au niveau de la classe de 1ère, mais l'opération sur les coordonnées définie par l'énoncé revient à appliquer au complexe $z=x+iy$, la transformation $z'=z^2+z_I$ avec $z_I=x_I+iy_I$. L'utilisation des complexes n'apporte pas grand chose de plus à cette situation, tant que l'on se satisfait d'une exploration du phénomène, sans rechercher les explications qui sont, elles, liées aux propriétés de ce type de suites complexes qui ont été étudiées par le mathématicien Gaston Julia au début du XXème siècle.
Avant de mettre en place l'interactivité recherchée, on doit déjà construire cette image de l'ensemble de

Julia associé à un point. Il nous faut un *Canvas* (cadre) pour y dessiner des points, c'est-à-dire des rectangles de 1 pixel de côté (*cadre.create_rectangle*(*x*,*y*,*x*+1,*y*+1,*width*=0,*fill*=*coul*)). Tout se joue sur la couleur à donner au pixel. Cette couleur peut être choisie dans un dégradé dès lors que l'on a déterminé la valeur de l'entier *n* (entre 1 et un maximum appelé *profondeur*) pour lequel l'image P_n du point sort du disque de rayon 2. La fonction *calcule*() effectue cette détermination. Il faut, au préalable, avoir déterminé les coordonnées réelles (x_P;y_P) du point *P* auquel elle s'applique. Comme on balaye la fenêtre pixel par pixel, on applique des formules de changement de repère aux coordonnées (*x*;*y*) de *P* dans la fenêtre : $x_P=x\times(xMax-xMin)/taille+xMin$, où *xMax* et *xMin* sont les maximum et minimum de l'abscisse réelle *x* (au départ, on a dit que c'était 2 et –2) et où *taille* est la dimension de la fenêtre (on a pris 600 pixels, mais c'est beaucoup, les temps de calcul seront longs, 400 aurait pu faire l'affaire).

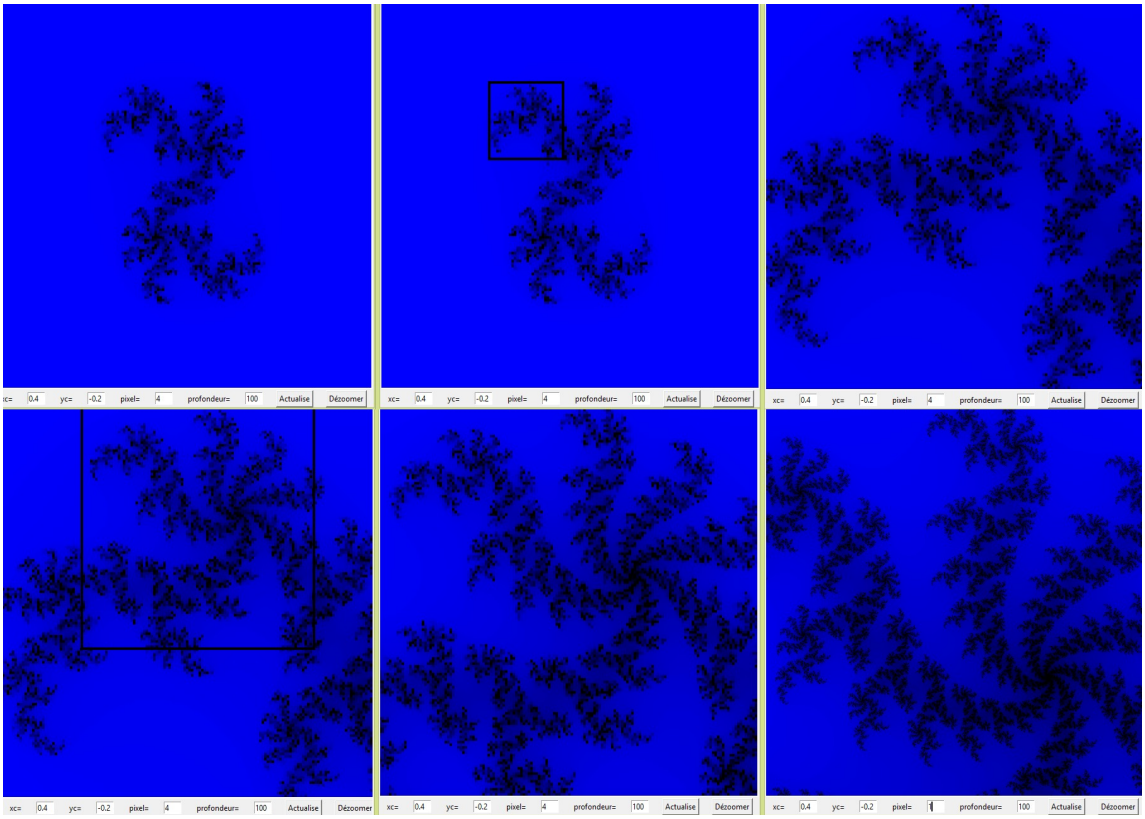
```
from tkinter import *
class Point :
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def suivant(self):
        return Point(self.x**2-self.y**2+iPoint.x,self.x*self.y*2+iPoint.y)
    def distance(self):
        return self.x**2+self.y**2

def modifie():
    xI=float(entree1.get())
    yI=float(entree2.get())
    iPoint=Point(xI,yI)
    trace()
def profondeur():
    global prof
    index=coeffProf.curselection()
    prof=int(coeffProf.get(index))
    trace()
def calcule(p):
    p1=p
    for i in range(prof):
        p1=p1.suivant()
        if p1.distance()>=2: return i
    return -1
def trace():
    for x in range(taille):
        for y in range(taille):
            p=Point(x*(xMax-xMin)/taille+xMin,y*(yMax-yMin)/taille+yMin)
            n=calcule(p)
            if n==-1: coul='black'
            else:
                coul,vert='#00',(prof-n)*255//prof
                if vert>15 :coul+=hex(vert)[2:]
                else:coul+='0'+hex(vert)[2:]
                coul+='00'
            cadre.create_rectangle(x,y,x+1,y+1,width=0,fill=coul)

fen=Tk()
fen.title('Ensemble de Julia du point c')
xI,yI=-0.8,-0.2
taille,prof=600,10
iPoint=Point(xI,yI)
xMin,xMax,yMin,yMax=-2,2,-2,2
cadre=Canvas(fen,width=600,height=600,bg='white')
cadre.grid(row=1,column=1,columnspan=9)
Label(fen,text="xc=").grid(row=2,column=1)
entree1=Entry(fen,width=5)
entree1.insert(END,"-0.8")
entree1.grid(row=2,column=2)
Label(fen,text="yc=").grid(row=2,column=3)
entree2=Entry(fen,width=5)
entree2.insert(END,"-0.2")
entree2.grid(row=2,column=4)
Button(fen,text='Changer c',command=modifie).grid(row=2,column=5)
Label(fen,text="profondeur=").grid(row=2,column=6)
coeffProf=Listbox(fen,height=1,width=7)
for i in [10,20,30,50,75,100,150,200,300,500,1000]:
    coeffProf.insert(END,str(i))
coeffProf.grid(row=2,column=8)
Button(fen,text='Changer p',command=profondeur).grid(row=2,column=9)
trace()
fen.mainloop()
```

Nous avons défini une classe *Point* qui ne contient que deux attributs (*x* et *y*) et deux méthodes (*suivant*() qui renvoie un nouvel objet de la classe *Point* répondant à la définition du point suivant donnée dans l'énoncé et *distance*() qui renvoie la distance du point à l'origine). Pour introduire un peu d'interactivité dans ce programme initial, nous avons disposé deux cadres de texte qui permettent d'entrer de nouvelles valeurs pour les coordonnées du point de référence *I* (nous l'avons renommé *C* pour une commodité d'affichage, le *I* ne se lisant pas bien), un bouton pour relancer le dessin après la modification de *C* (celui-ci déclenche la fonction *modifie*() qui change la variable globale iPoint (le point de référence)) et une *listBox*() pour choisir la profondeur d'investigation dans une liste préétablie (cette dernière option nécessitant l'usage des flèches du clavier pour atteindre les différentes valeurs ne semble pas particulièrement adaptée, nous ne la conserverons pas dans les versions ultérieures).



La disposition des « widgets » dans la fenêtre est assurée ici par la méthode *grid*() qui permet de les disposer selon une grille : ligne 1 nous plaçons le cadre graphique (de la classe *Canvas*) et dans la ligne 2, tous les widgets permettant l'interaction : les cadres de texte (*Entry*) accompagnés de leur étiquette (*Label*), les boutons (*Button*) et la liste déroulante (*Listbox*). Sur cette ligne (*row*=2), les emplacements sont fixés par le numéro de colonne (*column*=i). Lorsqu'un widget s'étale sur plusieurs colonnes, on indique leur nombre (*columnspan*=9 pour le cadre ici). La largeur des cadres de texte peut se paramétrer (*width*=5) ainsi que bien d'autres caractéristiques dont nous ne faisons pas usage, et ces zones peuvent contenir un texte

initial (ici, on y a indiqué la valeur par défaut).
Le résultat est déjà assez satisfaisant mais la palette de couleur utilisée se révèle trop pauvre, juste suffisante pour donner un aperçu de l'ensemble de Julia considéré. La nécessité de disposer d'un zoom se fait aussi sentir, car on aimerait pouvoir agrandir certaines parties de la figure pour l'observer avec plus de détails. Notre deuxième version du programme va ajouter ces éléments pour une meilleure performance et un plus grand confort d'utilisation (on pourrait aussi bien modifier les paramètres d'affichages directement dans le programme mais cela n'est pas très commode). Nous ne travaillerons pas directement l'optimisation de nos calculs (il faudrait en particulier utiliser les nombres complexes), mais pour accélérer l'affichage de la figure nous disposons déjà du paramètre profondeur qui, en limitant les investigations donne une image plus ou moins affinée et, par voie de conséquence, plus ou moins lente à s'afficher. Notre 2ème version ajoute une autre disposition pour accélérer l'affichage (tout en diminuant sa qualité) : la pixellisation peut être augmentée (au lieu de carrés de 1 pixel de côté, on peut dessiner avec des carrés de 2, 3, 4, etc. pixels de côtés. Le paramètre *pixel* a donc pour fonction de réduire le nombre de points. Il va être ajouté aux dispositifs d'interactivité dans la ligne 2 du cadre pour être réglable par l'utilisateur. La profondeur va être réglée par un cadre de texte (suppression de la *Listbox* qui n'apporte rien). Le zoomage va se réaliser en deux temps, comme il est dit dans le texte : on clique tout d'abord sur deux points de l'image (donc le *Canvas* doit être rendu sensible avec la méthode *bind*), et un cadre carré noir indique le cadre sélectionné pour être agrandi (cette opération doit pouvoir être recommencée jusqu'à satisfaction), on lance le recalcul du dessin en validant la sélection (un bouton *actualise* lance, d'un coup, toutes les opérations de modification). L'option de pixellisation est réglée par défaut sur 4 pour accélérer le choix du dessin que l'on veut obtenir.



Cette deuxième version ne règle pas le défaut de la palette de couleur. On aimerait aussi avoir des indications sur les valeurs des coordonnées réelles. Ces indications seront données par une option d'affichage des axes gradués dans une version ultérieure. Mais, pour l'essentiel, cette version n°2 nous donne satisfaction, car on peut observer à loisir (avec des temps d'attente importants lorsque la pixelisation est minimale et la profondeur maximale) ces magnifiques ensembles fractals. Le fonctionnement du zoom nécessite d'utiliser les évènements de souris *<ButtonPress-1>* et *<ButtonRelease-1>* qui lance chacun une fonction (*coin1*() quand on clique sur le bouton gauche et *coin2*() quand on relâche ce bouton). Les variables *x1, y1, x2* et *y2* enregistrent les coordonnées des clics (dans le système de repérage lié à la fenêtre) et l'on s'arrange pour que *x1* et *y1* soient les coordonnées du point supérieur gauche (on utilise pour cela les fonction *min* et *max* de Python) car l'utilisateur peut décrire la diagonale dans 4 sens différents. On recalcule aussi, dans la fonction *coin2*(), la valeur de *x2* et *y2* pour que la fenêtre soit carrée (et non pas rectangulaire). La mise à jour des nouvelles valeurs de *xMin, xMax, yMin* et *yMax* se fait dans

la fonction *modifie*() qui est lancée lorsque l'on souhaite modifier quelque chose dans l'affichage.

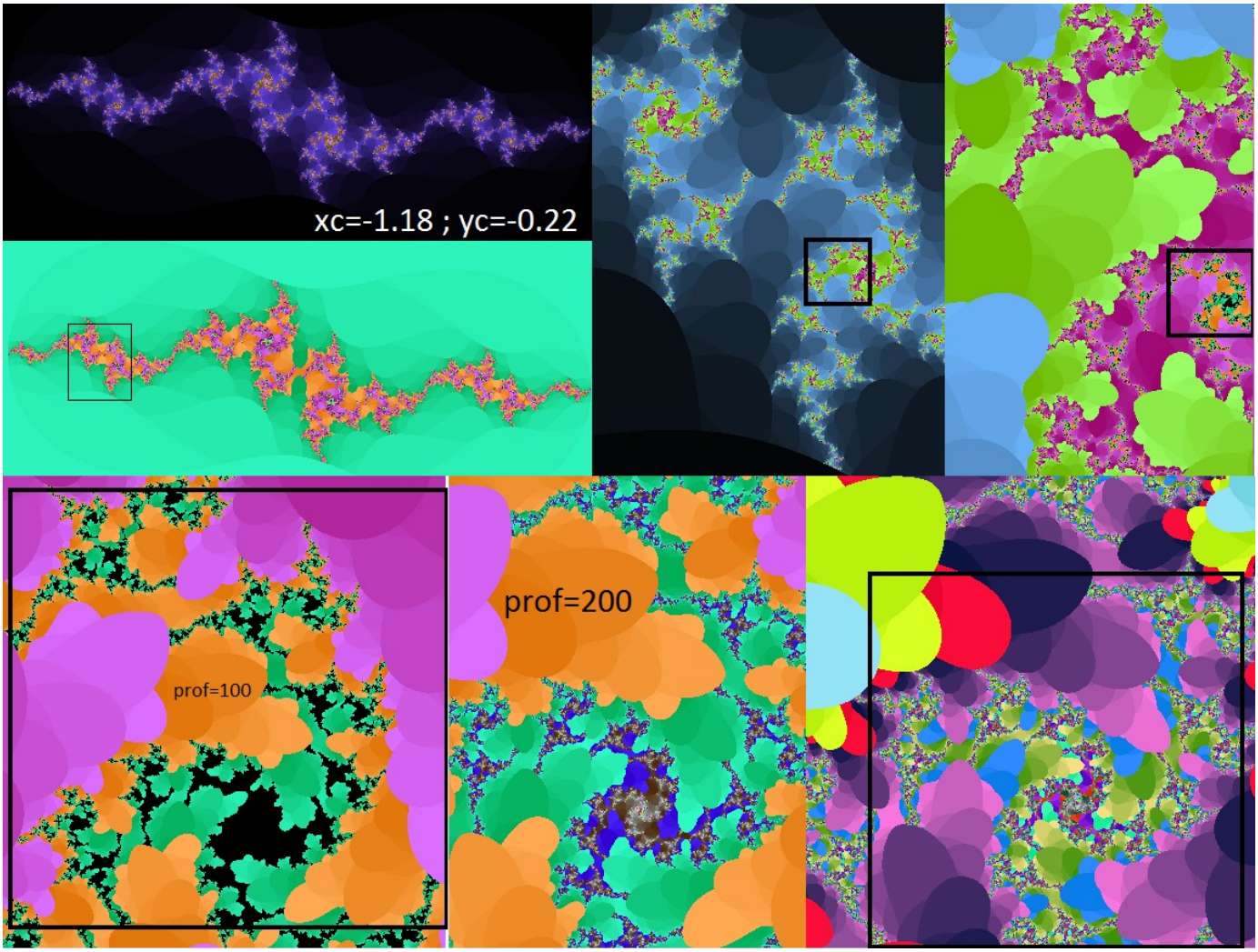
```
def actualise():
    global prof,pixel,iPoint,xMin,xMax,yMin,yMax,treelle
    xI=float(entree1.get())
    yI=float(entree2.get())
    iPoint=Point(xI,yI)
    pixel=int(entree3.get())
    prof=int(entree4.get())
    if x2>0:
        xMin,yMin=x1*treelle/taille+xMin,y1*treelle/taille+yMin
        c=min(max(x1,x2)-min(x1,x2),max(y1,y2)-min(y1,y2))
        treelle=c*treelle/taille
        xMax,yMax=xMin+treelle,yMin+treelle
    trace()
def coin1(event):
    global x1,y1
    x1,y1=event.x,event.y
def coin2(event):
    global x1,y1,x2,y2
    x2,y2=event.x,event.y
    c=min(max(x1,x2)-min(x1,x2),max(y1,y2)-min(y1,y2))
    x1,y1=min(x1,x2),min(y1,y2)
    x2,y2=x1+c,y1+c
    retrace()
    cadre.create_rectangle(x1,y1,x2,y2,width=4)
def dezoom():
    global xMin,xMax,yMin,yMax,treelle
    xMin,xMax,yMin,yMax,treelle=-2,2,-2,2,4
    cadre.delete(ALL)
    trace()
def calcule(p):
    p1=p
    for i in range(prof):
        p1=p1.suivant()
        if p1.distance()>=2: return i
    return -1
def retrace():
    for pix in image:
        cadre.create_rectangle(\
            pix[0]*pixel,pix[1]*pixel,\
            (pix[0]+1)*pixel,(pix[1]+1)*pixel,\
            width=0,fill=pix[2])
```

```
def trace():
    global image
    image=[]
    for x in range(taille//pixel):
        for y in range(taille//pixel):
            p=Point(x*pixel*treelle/taille+xMin,y*pixel*treelle/taille+yMin)
            n=iteration(p)
            if n==-1: coul='black'
            else:
                coul,bleu='#0000',(prof-n)*255//prof
                if bleu>15 :coul+=hex(bleu)[2:]
                else:coul+='0'+hex(bleu)[2:]
            image.append([x,y,coul])
            cadre.create_rectangle(x*pixel,y*pixel,(x+1)*pixel,(y+1)*pixel,width=0,fill=coul)

fen=Tk()
fen.title('Ensemble de Julia du point c')
taille,treelle,prof,xMin,xMax,yMin,yMax=600,4,100,-2,2,-2,2
xI,yI,pixel,x1,x2,y1,y2,image=0.4,-0.2,4,0,0,0,0,[]
iPoint=Point(xI,yI)
cadre=Canvas(fen,width=600,height=600,bg='white')
cadre.grid(row=1,column=1,columnspan=10)
cadre.bind('<ButtonPress-1>',coin1)
cadre.bind('<ButtonRelease-1>',coin2)
Label(fen,text="xc=").grid(row=2,column=1)
entree1=Entry(fen,width=4)
entree1.insert(END,"0.4")
entree1.grid(row=2,column=2)
Label(fen,text="yc=").grid(row=2,column=3)
entree2=Entry(fen,width=4)
entree2.insert(END,"-0.2")
entree2.grid(row=2,column=4)
Label(fen,text="pixel=").grid(row=2,column=5)
entree3=Entry(fen,width=4)
entree3.insert(END,"4")
entree3.grid(row=2,column=6)
Label(fen,text="profondeur=").grid(row=2,column=7)
entree4=Entry(fen,width=4)
entree4.insert(END,"100")
entree4.grid(row=2,column=8)
Button(fen,text='Actualise',command=actualise).grid(row=2,column=9)
Button(fen,text='Dézoomer',command=dezoom).grid(row=2,column=10)
trace()
fen.mainloop()
```

Dans cette version bleue, nous enregistrons les données de l'image (*x, y* et *n*) dans la liste *image* afin de pouvoir redessiner celle-ci sans devoir recalculer les valeurs de *n* (lorsqu'on ne fait que changer les paramètres d'affichages). Le bouton « Dézoomer » permet de revenir aux conditions initiales (on pourrait remettre également les valeurs par défaut de *pixel* et de *profondeur*).

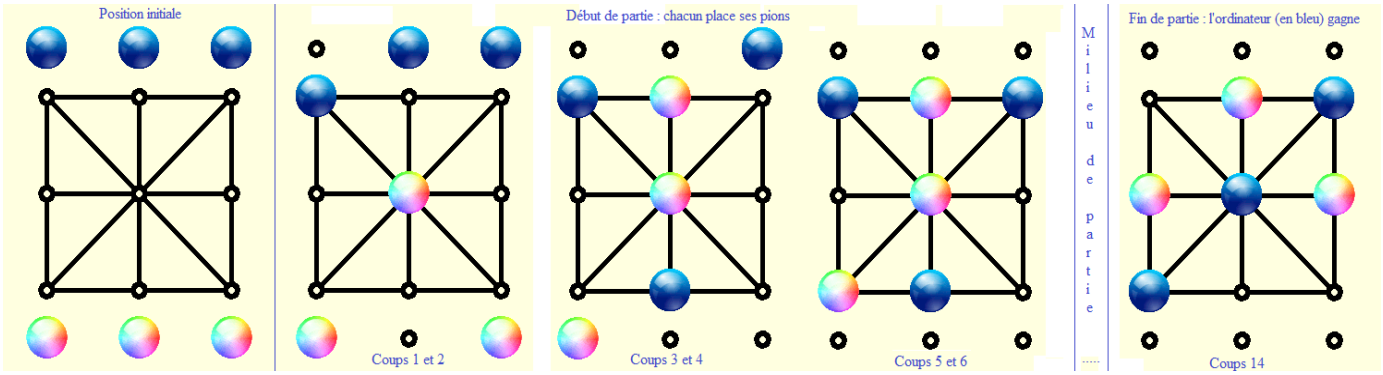
Notre objectif d'interactivité ne s'arrête pas avec cette version. Nous avons programmé une palette plus riche, en nous inspirant du modèle proposé par Laurent Signac dans son très sympathique livre « Divertissements mathématiques et informatiques » (paru en 2011 aux éditions HK, collection MiniMax) où cette situation est traitée dans le langage *Ruby*. Cet auteur propose ((4*n*)%256, 2*n*, (6*n*)%256) comme palette RVB (les composantes Rouge, Verte et Bleue sont des entiers entre 0 et 255 qui sont exprimées en hexadécimal (base 16) dans la syntaxe Python utilisée). Nous avons adopté quelque chose de similaire mais modifiable par des boutons, dans une 3ème ligne de notre fenêtre. Notre couleur associée à *n* est : ((*prof*−*n*)**col*[0])%256, (*prof*−*n*)**col*[1])%256, (*prof*−*n*)**col*[2])%256) où *col*=[3,5,7] par défaut. L'idée de multiplier le nombre *prof*−*n* (qui décroit de *prof* à 0 quand *n* croit de 0 à *prof*) par un coefficient réglable permet de modifier la palette de façon assez riche (et imprévisible, il faut bien le dire). Avec 10 valeurs au choix pour les paramètres *col*[i] (nous avons pris les 10 premiers nombres premiers mais ce choix n'apporte rien de spécial sinon la diversité), nous obtenons plus de 1000 palettes différentes pour apporter une très légère touche créative à cette application. Une case à cocher permet d'inverser les coloris en utilisant les valeurs de *n* au lieu de celles de *prof*−*n* dans le calcul des composantes de la couleur. Le fonctionnement des boutons ne prévoit pas l'envoi d'un paramètre (on exécute une fonction sans arguments) mais nous avons six boutons pour régler les valeurs des *col*[i] et nous ne voulons pas écrire six fonctions différentes. La solution à ce problème réside dans l'utilisation de fonctions *lambda*, une particularité de Python, qui s'écrit simplement : *lambda*:*palette*(0,0) pour lancer la fonction *palette*() avec les deux arguments 0 et 0. Pour construire le bouton « rouge- » qui, en lançant *palette*(0,0), diminue la valeur de *col*[0], l'instruction complète est : *Button*(*fen*, *text*='rouge-', *command*=*lambda*:*palette*(0,0), *bg*='red').*grid*(*row*=3, *column*=1).

Le résultat de ces améliorations est visible au début de cette partie et dans l'image qui suit. Nous pourrions encore enrichir l'interactivité de ce programme en proposant différentes options d'enregistrement ou d'impression de l'image, ou en combinant une passerelle entre ces ensembles de Julia et l'ensemble de Mandelbrot auxquels ils sont liés (nous avions fait cela il y a quelques années dans un programme *Java* qui se trouve sur mathadomicile). L'intérêt cependant de ces améliorations est amoindri par les temps d'attente pour recalculer ou même redessiner une de ces images. Il faudrait d'abord réduire ceux-ci pour espérer motiver de nouveaux développements.



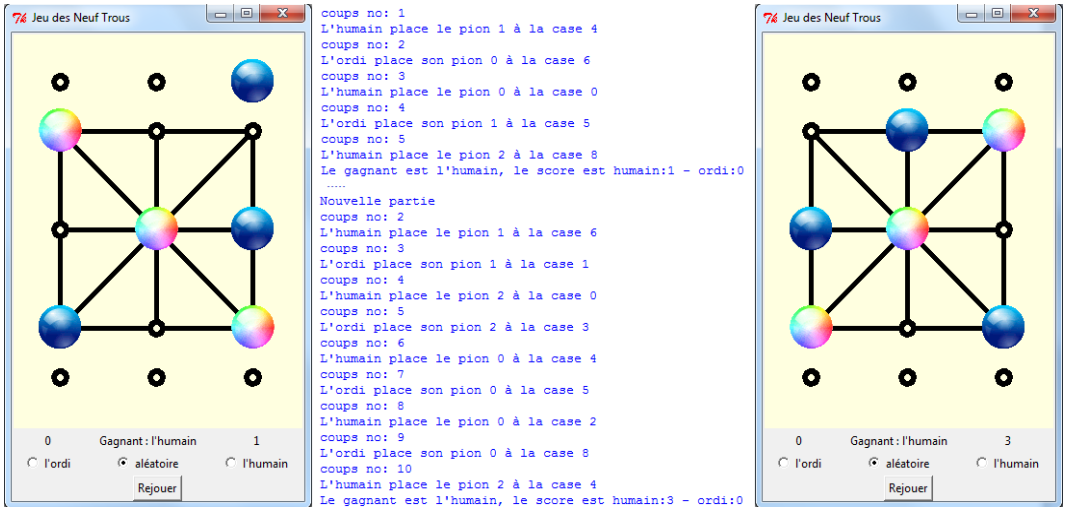
c) Donnons-nous un objectif un peu plus ludique que la simple exploration d'une image statique, fusse t-elle magnifique, et un peu plus ambitieux que la programmation du sudoku où les contraintes ne laissent aucune possibilité de variation. Un jeu simple à deux joueurs dont l'un des joueurs est l'ordinateur et qui nécessite une forme d'intelligence de sa part : le jeu des neuf trous appelé aussi *tapatan*. Le plateau de jeu est une grille carrée de 3 sur 3 et chaque joueur dispose de trois pions qu'il doit aligner sur le plateau. Au début, chacun à tour de rôle pose un pion, n'importe où sur le plateau. Une fois les six pions posés, chacun peut déplacer un de ses pions d'une position, selon un des tracés figurés sur le plateau (le long des six lignes ou des deux diagonales), si le nouvel emplacement est libre. Programmer ce jeu avec un compteur des parties gagnées/perdues et un dispositif permettant de choisir si c'est le joueur qui commence (plus facile), si c'est l'ordinateur (plus difficile) ou bien si ce choix est laissé au hasard (équilibré). Dans un 1er temps, on peut se consacrer à la réalisation du jeu, sans chercher une réponse intelligente de la part de l'ordinateur (rendre ses choix aléatoires).

On dessiner le plateau dans un *Canvas* cadre une fois pour toute. Les pions sont créés à partir de deux images *photo1* et *photo2 (*au format gif) qui sont utilisées pour instancier des variables pion0[0], pion0[1] et pion0[2] pour les trois pions de l'ordinateur (désigné par 0 ou ordi) et pion1[*i*] pour ceux de l'humain (*i* variant de 0 à 2 pour l'humain désigné par 1). L'emplacement de ces pions est réglé par les coordonnées qui sont initialisées (pion0=[] puis pion0.*append*(cadre.*create_image*(*i**100+50,50,*image*=photo1) crée le pion0[0]), utilisées ultérieurement (avec [*xDeb*,*yDeb*]=cadre.*coords*(pion1[*i*]), les variables *xDeb* et *yDeb* contiennent les coordonnées de pion1[*i*]) ou modifiées (cadre.*coords*(pion1[*i*],*x1*,*y1*) affecte les coordonnées *x1* et *y1* au pion pion1[*i*]). Le mouvement des pions est lié aux mouvements de la souris : le clic déclenche la fonction *clic*() qui détecte que le pion *choixPion* a été sélectionné (on ne s'intéresse qu'aux pions de l'humain car l'ordinateur bouge lui-même ses pions) en acceptant un certain domaine de variation des coordonnées du clic (*x.event* et *y.event*) autour de coordonnées du pion ; le déplacement de la souris, bouton de gauche enfoncé, lance la fonction *drag*() qui redessine le pion lors de son déplacement ; le relâchement de la souris lance la fonction *lache*() qui examine le nouvel emplacement, fixe le pion à cet emplacement s'il est vide et s'il correspond à un placement valide (fonction *avoisine*() de la classe *Jeu*).

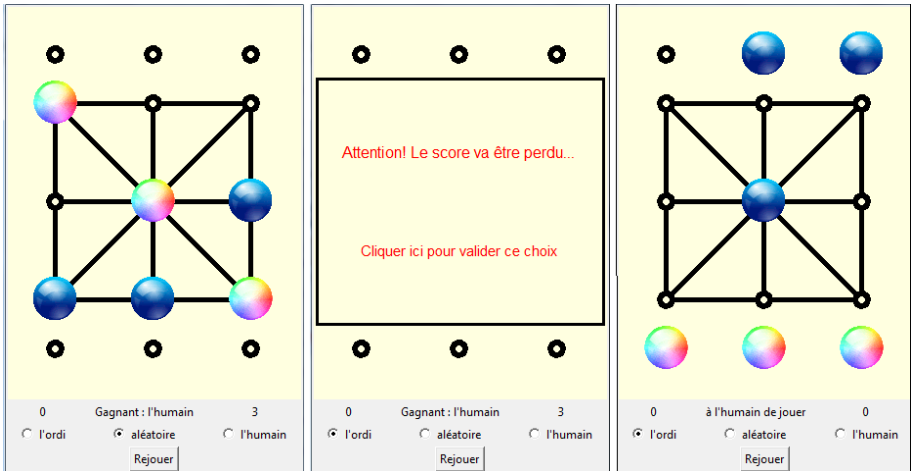


La classe *Jeu* que nous avons définie n'est pas forcément indispensable. Elle prend en arguments les emplacement des six pions et le joueur qui a la main (0 ou 1). Pour l'instant, elle dispose de la méthode *voisin*() qui se contente d'examiner les voisins valides, c'est-à-dire les placements valides de chacun des pions du joueur, et d'en renvoyer la liste. Cette classe est aussi utilisée pour sa méthode *finished*() qui détermine si une configuration de jeu est gagnante (dans ce cas, elle renvoie le joueur qui gagne, ou −1 si personne n'a encore gagné).

Lorsqu'un joueur a gagné, le score est modifié et un petit message est affiché à la place où, le reste du temps, on affiche que c'est à l'humain de joueur (lorsque l'ordinateur doit jouer, l'affichage le signale, mais ici, la réflexion de l'ordinateur est trop rapide pour que cela se voie). Le bouton « Rejouer » permet de rejouer une partie, ce qui repositionne les pions à l'extérieur du plateau et réinitialise les variables qui doivent l'être. Le mécanisme d'une telle interactivité, aussi modeste soit-elle, est assez délicat à régler. En plus des inévitables erreurs flagrantes de programmation, il y a souvent des petites erreurs bien cachées qui obligent à s'interroger sur les variables : à quels moments sont-elles initialisées, à quels moments changent-elles de valeurs ? Combien valent alors les autres ? La portée d'une variable est-elle celle que l'on attend (la portée d'une variable est souvent limitée à la fonction, ou à la boucle dans laquelle elle apparaît, à moins que la variable soit déclarée *global*, dans ce cas elle vaut la même chose pour toutes les parties qui reconnaissent cette variable globale...) ? Quelques instructions *print*() judicieusement placées aident souvent dans cette démarche de débogage.



Pour le changement du mode d'utilisation (les trois boutons « radio »), nous devons avertir l'utilisateur que cela va remettre les scores à 0-0 (car on n'utilise plus les mêmes règles). Afin de réaliser cela simplement, nous affichons juste un bandeau provisoire qui s'efface lorsqu'on clique dedans. On peut prévoir aussi qu'en cas d'erreur de manipulation, on souhaite éviter la remise à zéro des compteurs et remettre le réglage précédent. Pour l'instant, ces subtilités ne sont pas vraiment justifiées mais elles pourraient le devenir avec une application opérationnelle. Au stade de développement où l'on est, on pourrait aussi apporter quelques améliorations : l'animation étant encore assez pauvre, il serait judicieux de



simuler une certaine lenteur sur une trajectoire pour les mouvements du pion de l'ordinateur ; on peut aussi prévoir de jouer un son lors du déplacement d'un pion et un autre lors de son placement. On peut aussi envisager d'enregistrer le score sur un serveur, mais cela dépasse légèrement les objectifs de ce document. Consacrons-nous plutôt à la partie « intelligence artificielle » qui va faire de notre ordinateur un adversaire plus sérieux.

```
from tkinter import *
from random import randint
class Jeu :
    def __init__(self,mp=[-1,-1,-1],op=[-1,-1,-1],j=0):
        self.mesPlaces=[]
        self.ordiPlaces=[]
        self.joueur=j
        for i in range(3):
            self.mesPlaces.append(mp[i])
            self.ordiPlaces.append(op[i])
    def voisin(self):#retourne une liste contenant [pion à bouger,place libre à occuper]
        voisin=[]
        for i in range(3):
            for j in range(9):
                if j not in self.mesPlaces and j not in self.ordiPlaces:
                    if self.joueur==1 and self.isVoisine(self.mesPlaces[i],j):voisin.append([i,j])
                    elif self.joueur==0 and self.isVoisine(self.ordiPlaces[i],j):voisin.append([i,j])
        return voisin
    def isVoisine(self,c1,c2):
        if c1==4 or c2==4: return True
        if c1>c2:c1,c2=c2,c1
        if (c1==0 and (c2==1 or c2==3))or(c1==1 and c2==2)or(c1==2 and c2==5)or\
           (c1==3 and c2==6)or(c1==5 and c2==8)or(c1==6 and c2==7)or(c1==7 and c2==8): return True
        return False
    def finished(self):#retourne 1 si l'humain gagne, 0 si c'est l'ordi, -1 dans les autres cas
        winner=-1
        gagne=[[0,3,6],[1,4,7],[2,5,8],[0,1,2],[3,4,5],[6,7,8],[0,4,8],[2,4,6]]
        if sorted(self.mesPlaces) in gagne: winner=1
        if sorted(self.ordiPlaces) in gagne: winner=0
        return winner
```

```
def ordinateur():
    global coup,possible,tour,debut,ordiPlaces
    choixPion,choixPlace,anciennePlace=-1,-1,-1
    if debut:
        choixPion=coup//2
        choixPlace=randint(0,8)#jeu provisoirement aléatoire
        while not possible[choixPlace]:choixPlace=randint(0,8)
    else:
        listeChoixPion=Jeu(mesPlaces,ordiPlaces).voisin()
        randy=randint(0,len(listeChoixPion)-1)
        choixPion=listeChoixPion[randy][0]
        choixPlace=listeChoixPion[randy][1]#jeu provisoirement aléatoire
        anciennePlace=ordiPlaces[choixPion]
    coup+=1
    print('coups no:',str(coup))
    print("L'ordi place son pion {} à la case {}".format(choixPion,choixPlace))
    possible[choixPlace]=False #on occupe une place qui était libre
    if coup==6: debut=False
    x1,y1=choixPlace%3*100+50,choixPlace//3*100+100
    cadre.coords(pion0[choixPion],x1,y1)
    ordiPlaces[choixPion]=choixPlace
    if notGagne(0):
        tour="l'humain"
        message.configure(text="à {} de jouer".format(tour))
    if not debut:
        possible[anciennePlace]=True #on libère une place
def clic(event):
    global detectionPion,choixPion,xDeb,yDeb,gagnant,avertissement
    x1,y1,detectionPion=event.x,event.y,False
    if avertissement and 0<x1<300 and 70<y1<330: eraseMessage()
    if tour=="l'humain" and gagnant==-1:
        for i in range(3):
            [xDeb,yDeb]=cadre.coords(pion1[i])
            if (x1-xDeb)**2+(y1-yDeb)**2<400:#estimation du rayon du pion à 20
                choixPion=i
                if debut and resteAjouer.count(choixPion)==0 :
                    choixPion=-1
                    break
                detectionPion=True
                break
```

```
def changeDebut():
    global coup,score,avertissement,previousChoix
    if coup!=0 and score!=[0,0]:
        avertissement=True
        fenMessage('Attention! Le score va être perdu...')
def fenMessage(s):#bandeau d'avertissement
    global band,mess1,mess2
    band=cadre.create_rectangle(5,75,298,325,width=3,fill='light yellow')
    mess1=cadre.create_text(150,150,text=s,font="Arial 12",fill='red')
    mess2=cadre.create_text(150,250,text='Cliquer ici pour valider ce choix'\
                            ,font="Arial 11",fill='red')
def eraseMessage():
    global coup,score,tour,band,mess1,mess2,avertissement
    score=[0,0]
    monScore.configure(text="0")
    ordiScore.configure(text="0")
    cadre.delete(band)
    cadre.delete(mess1)
    cadre.delete(mess2)
    avertissement=False
    rejouer()
def rejouer():
    global coup,resteAjouer,possible,debut,detectionPion,choixPion,mesPlaces\
           ,ordiPlaces,tour,gagnant,avertissement,band,mess1,mess2,previousChoix
    for i in range(3):
        cadre.coords(pion0[i],i*100+50,50)
        cadre.coords(pion1[i],i*100+50,350)
    coup,resteAjouer,possible=0,[0,1,2],[]
    debut,detectionPion,choixPion=True,False,-1
    mesPlaces,ordiPlaces,gagnant=[-1,-1,-1],[-1,-1,-1],-1
    if avertissement:
        cadre.delete(band)
        cadre.delete(mess1)
        cadre.delete(mess2)
        choixDebut.set(previousChoix)
        avertissement=False
    previousChoix=choixDebut.get()
    for i in range(9): possible.append(True)
    if choixDebut.get()==0:tour=joueurDebut[0]
    elif choixDebut.get()==1:tour=joueurDebut[randint(0,1)*2]
    else:tour=joueurDebut[2]
    message.configure(text="à {} de jouer".format(tour))
    if tour=="l'ordi": ordinateur()
    print('\nNouvelle partie')
def notGagne(j):
    global coup,gagnant
    if coup<5:return True
    gagnant=Jeu(mesPlaces,ordiPlaces,j).finished()
    if gagnant>=0:
        message.configure(text="Gagnant : {}".format(joueurDebut[gagnant*2]))
        score[gagnant]+=1
        monScore.configure(text=str(score[1]))
        ordiScore.configure(text=str(score[0]))
        print("Le gagnant est {}, le score est humain:{} - ordi:{}".\
              format(joueurDebut[gagnant*2],score[1],score[0]))
        return False
    return True
def tracePlateau():
    cadre.create_rectangle(50,100,250,300,width=5)
    cadre.create_line(50,100,250,300,width=5,fill='black')
    cadre.create_line(50,300,250,100,width=5,fill='black')
    cadre.create_line(50,200,250,200,width=5,fill='black')
    cadre.create_line(150,100,150,300,width=5,fill='black')
    for i in range(15):
        xt,yt=i%3*100+50,i//3*100
        if i<3:yt+=50
        if i>11:yt-=50
        cadre.create_oval(xt-7,yt-7,xt+7,yt+7,width=5,fill='black')
        cadre.create_oval(xt-3,yt-3,xt+3,yt+3,width=0,fill='light yellow')
def drag(event):
    x1,y1=event.x,event.y
    if detectionPion:
        if x1<0:x1=0
        if x1>300:x1=300
        if y1<50:y1=50
        if y1>350:x1=350
        cadre.coords(pion1[choixPion],x1,y1)
```

```
def lache(event):
    global detectionPion,possible,choixPion,coup,tour,debut,mesPlaces,resteAjouer,xDeb,yDeb
    if detectionPion and gagnant==-1:
        x1,y1,x2,y2,choixPlace=event.x,event.y,0,0,-1
        for i in range(9):
            x2,y2=i%3*100+50,i//3*100+100
            if (x2-x1)**2+(y2-y1)**2<400:
                choixPlace=i
                break
        if not debut:listeChoixPion=Jeu(mesPlaces,ordiPlaces,1).voisin()
        if possible[choixPlace] and choixPlace>=0 and \
           (debut or (not debut and [choixPion,choixPlace] in listeChoixPion)):
            anciennePlace=mesPlaces[choixPion]
            cadre.coords(pion1[choixPion],x2,y2)
            mesPlaces[choixPion]=choixPlace
            coup+=1
            print('coups no:',str(coup))
            print("L'humain place le pion {} à la case {}".format(choixPion,choixPlace))
            if notGagne(1):
                tour="l'ordi"
                message.configure(text="à {} de jouer".format(tour))
                resteAjouer[choixPion]=-1
                possible[choixPlace]=False #on occupe une place qui était libre
                if not debut:
                    possible[anciennePlace]=True #on libère une place
                    print("possible:",possible)
                if coup==6: debut=False
                choixPion=-1
                ordinateur()
        else:cadre.coords(pion1[choixPion],xDeb,yDeb)
        detectionPion=False
```

```
fen=Tk()
fen.title('Jeu des Neuf Trous')
joueurDebut=["l'ordi","aléatoire","l'humain"]
tour=joueurDebut[randint(0,1)*2]
cadre=Canvas(fen,width=300,height=400,bg='light yellow')
cadre.grid(row=1,column=1,columnspan=3)
cadre.bind("<Button-1>",clic)
cadre.bind("<B1-Motion>",drag)
cadre.bind("<ButtonRelease-1>",lache)
ordiScore=Label(fen,text="0")
ordiScore.grid(row=2,column=1)
message=Label(fen,text="à {} de jouer".format(tour))
message.grid(row=2,column=2)
monScore=Label(fen,text="0")
monScore.grid(row=2,column=3)
choixDebut=IntVar()
choixDebut.set(1)
for i in range(3): # Création des trois 'boutons radio' :
    bouton=Radiobutton(fen,text=joueurDebut[i],variable=choixDebut,
                       value=i,command=changeDebut)
    bouton.grid(row=3,column=i+1)
bouton=Button(fen,text="Rejouer",command=rejouer)
bouton.grid(row=4,column=1,columnspan=3)
photo1=PhotoImage(file='buttonBleu.gif')
photo2=PhotoImage(file='buttonArcEnCiel.gif')
#initialisations des variables globales
debut,detectionPion,avertissement=True,False,False
possible,coup,score,resteAjouer=[],0,[0,0],[0,1,2]
mesPlaces,ordiPlaces,choixPion=[-1,-1,-1],[-1,-1,-1],-1
xDeb,yDeb,gagnant,previousChoix=0,0,-1,1
band,mess1,mess2=None,None,None
for i in range(9): possible.append(True)
pion0,pion1=[],[]
tracePlateau()
for i in range(3):# Création des six pions :
    pion0.append(cadre.create_image(i*100+50,50,image=photo1))#ordi
    pion1.append(cadre.create_image(i*100+50,350,image=photo2))#humain
if tour=="l'ordi": ordinateur()
fen.mainloop()
```

La question qui se pose est de nature algorithmique. Que doit faire l'ordinateur pour bien jouer : il a, à sa disposition, une liste de possibilités valides (*listeChoixPion*) parmi lesquelles il doit choisir la plus prometteuse. On pourrait se contenter d'éliminer les possibilités qui permettraient à l'humain de gagner au coup suivant. Ce serait déjà un peu mieux que de faire un choix totalement aléatoire. La solution idéale serait d'examiner les prolongements de chacun des coups possibles jusqu'à l'issue la pire (perte de l'ordinateur), la meilleure (gain de l'ordinateur), ou jusqu'à la conviction que la partie est nulle (un cycle

sans vainqueur). Cette solution risque d'être coûteuse en temps d'exécution aussi nous allons examiner l'option alternative qui consiste à offrir un choix de plusieurs niveaux pour la réflexion de l'ordinateur : au niveau 0, il répond aléatoirement. Au niveau 1, il examine les prolongements jusqu'au coup suivant. Au niveau 2, il les examine jusqu'à deux coups (après le coup de l'ordinateur, l'humain joue, puis l'ordinateur, puis encore l'humain). Un humain doit, avec un peu d'attention et d'entraînement, jouer à ce niveau 2. Pour être un peu plus difficile à battre, il faudrait pouvoir faire jouer l'ordinateur jusqu'au niveau 3.
L'algorithme du *minimax* est parfaitement adapté à cette situation. Parmi toutes les possibilités de jeu, l'ordinateur choisit celle qui maximise son gain. Lorsqu'il examine les prolongements de son coup, parmi toutes les possibilités de jeu de l'humain, il choisit celle qui minimise son gain. Arrivé à la profondeur maximale de sa réflexion ou bien à une situation où l'un des deux joueurs a gagné, il note 0 les configurations indécises, +10 celles où il gagne et −10 celles où il perd. Cet algorithme remplace le choix aléatoire fait dans la fonction *ordinateur*() et quelques fonctions nouvelles s'ajoutent à notre programme : *chercheMaxi*() et *chercheMini*() sont les deux fonctions qui s'appellent alternativement, jusqu'à atteindre la profondeur désirée ; *evaluer*() note les différentes configurations aux extrémités (feuilles) de l'arbre de la réflexion de l'ordinateur. Nous invitons le lecteur curieux d'en apprendre davantage sur l'algorithme du *minimax* d'effectuer quelques recherches à ce sujet sur internet.

```
class Jeu :
    ....
    def setPion(self,pion,case,joueur):
        if joueur==0:self.ordiPlaces[pion]=case
        else: self.mesPlaces[pion]=case
    def getPion(self,pion,joueur):
        if joueur==0: return self.ordiPlaces[pion]
        else: return self.mesPlaces[pion]
```

```
    ....
def evaluer(j):
    gagnant=j.finished()
    if gagnant==0:return [0,0,10]
    elif gagnant==1: return [0,0,-10]
    else:return [0,0,0]
def chercheMaxi(mp,op,prof,cp): # L'ordi joue
    max_pion,max_case,maximum=-1,-1,-100
    mesPlaces,ordiPlaces=[-1,-1,-1],[-1,-1,-1]
    for i in range(3):
        mesPlaces[i],ordiPlaces[i]=mp[i],op[i]
    jeu=Jeu(mesPlaces,ordiPlaces)
    if prof==0 or jeu.finished()>=0: return evaluer(jeu)
    if cp//2<3:
        pion=cp//2#l'ordi joue ses pions dans l'ordre
        for i in range(9):
            if i not in ordiPlaces and i not in mesPlaces:#case vide
                jeu.setPion(pion,i,0)
                score=chercheMini(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp+1)[2]
                if score>maximum:
                    maximum=score
                    max_case=i
                    max_pion=pion
                jeu.setPion(pion,-1,0)
    else:
        listeChoixPion=jeu.voisin()
        for i in range(len(listeChoixPion)):
            pion=listeChoixPion[i][0]
            case=listeChoixPion[i][1]
            ancienneCase=jeu.getPion(pion,0)
            jeu.setPion(pion,case,0)
            score=chercheMini(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp)[2]
            if score>maximum:
                maximum=score
                max_case=case
                max_pion=pion
            jeu.setPion(pion,ancienneCase,0)
    return [max_pion,max_case,maximum]
```

```
def ordinateur():
    global coup,possible,tour,debut,ordiPlaces
    choixPion,choixPlace,anciennePlace=-1,-1,-1
    [choixPion,choixPlace,c]=chercheMaxi(mesPlaces,ordiPlaces,6,coup)
    if not debut:anciennePlace=ordiPlaces[choixPion]
    coup+=1
    print('coups no:',str(coup))
    print("L'ordi place son pion {} à la case {}".format(choixPion,choixPlace))
    possible[choixPlace]=False #on occupe une place qui était libre
    if coup==6: debut=False
    x1,y1=choixPlace%3*100+50,choixPlace//3*100+100
    cadre.coords(pion0[choixPion],x1,y1)
    ordiPlaces[choixPion]=choixPlace
    if notGagne(0):
        tour="l'humain"
        message.configure(text="à {} de jouer".format(tour))
    if not debut:
        possible[anciennePlace]=True #on libère une place
def chercheMini(mp,op,prof,cp): # coup de l'humain
    min_pion,min_case,minimum=-1,-1,100
    mesPlaces,ordiPlaces=[-1,-1,-1],[-1,-1,-1]
    for i in range(3):
        mesPlaces[i],ordiPlaces[i]=mp[i],op[i]
    jeu=Jeu(mesPlaces,ordiPlaces,1)
    if prof==0 or jeu.finished()>=0: return evaluer(jeu)
    if cp//2<3:
        for i in range(3):
            if mesPlaces[i]==-1:
                pion=i
                break
        for i in range(9):
            if i not in ordiPlaces and i not in mesPlaces:#case vide
                jeu.setPion(pion,i,1)
                score=chercheMaxi(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp+1)[2]
                if score<minimum:
                    minimum=score
                    min_case=i
                    min_pion=pion
                jeu.setPion(pion,-1,1)
    else:
        listeChoixPion=jeu.voisin()
        for i in range(len(listeChoixPion)):
            pion=listeChoixPion[i][0]
            case=listeChoixPion[i][1]
            ancienneCase=jeu.getPion(pion,1)
            jeu.setPion(pion,case,1)
            score=chercheMaxi(jeu.mesPlaces,jeu.ordiPlaces,prof-1,cp)[2]
            if score<minimum:
                minimum=score
                min_case=case
                min_pion=pion
            jeu.setPion(pion,ancienneCase,1)
    return [min_pion,min_case,minimum]
```

Voilà notre adversaire bien doté maintenant et difficile à battre, ce qui était notre objectif. Au niveau 3, l'ordinateur examine jusqu'à 6 coups successifs (3 aller-retours ordinateur-humain), ce qui représente une durée sensible mais raisonnable. Nous laissons au lecteur le soin d'améliorer l'animation. Nous n'avons pas rendu la profondeur réglable par l'utilisateur (cela ne semble pas nécessaire ici), le lecteur pourra apporter ce petit changement et d'autres encore auxquels nous ne pensons pas. Par contre, ce jeu des neuf trous étant un programme censé intéresser des utilisateurs non-initiées à Python, il nous est apparu indispensable de transformer ce *tapatan2.py* en un *tapatan2.exe* − un fichier portable (on dit aussi « standalone ») qui peut être exécuté sans avoir Python sur son ordinateur. L'opération n'est pas simple à exécuter, c'est une des faiblesses reconnues de Python, mais avec beaucoup de patience, on finit par comprendre comment réaliser cela : un module externe (*py2exe*) est nécessaire, il faut le télécharger, l'installer, créer un fichier *setup.py* sur le modèle de notre illustration, et lancer la commande *python setup.py py2exe* depuis une « invite de commande » *Windows* (nous n'avons pas essayé sur d'autres systèmes d'exploitation). C'est tout ! On obtient alors un ensemble de fichiers que l'on peut compresser avec un utilitaire en un fichier *zip*. Notre archive *tapatan.zip* est complètement autonome (en principe) : on peut la copier sur une clef ou la mettre sur un site, l'utilisateur final la décompresse et lance le fichier *tapatan2.exe* qui y est présente, sans même

savoir que c'est un programme Python.

```
from distutils.core import setup
import py2exe
Mydata_files = [('image', ['C:/Python34/image/buttonArcEnCiel.gif','C:/Python34/image/buttonBleu.gif'])]
setup(name="tapatan",version="1.0",description="Jeu des neuf trous ou Tapatan",author="PM",zipfile=None,
windows=['tapatan2.py'],data_files=Mydata_files,options={"py2exe":{"compressed":1,"bundle_files":2}})
```

d) Pour finir en beauté, envisageons une application graphique qui utilise un menu. Fixons un but à ce programme : réaliser des rosaces, des figures ayant *n* axes de symétrie concourants (*n* est variable, supérieur ou égal à 2). Les « pinceaux » ont une couleur et une taille réglables. On trace des lignes de plusieurs types (des segments, des arcs de cercles, des lignes libres, etc.) et on doit pouvoir gommer (supprimer des éléments). Dans un 2ème temps, on peut prévoir un enregistrement de l'image ainsi crée au format *png* ou une exportation de cette image dans le « presse-papier ».

L'installation du menu Python est réalisée dans une *Frame* (un *widget* pouvant contenir d'autres *widgets*) que l'on nomme mBar (mBar=*Frame*(fen), fen étant le conteneur *Tk*). On dispose ensuite des *Menubutton* dans cette *Frame* (par exemple, fich=*Menubutton*(mBar, *text*='Fichier') est un bouton portant l'étiquette « Fichier ») et à l'intérieur de ceux-ci, on met les différentes options par un dispositif un peu complexe :

me1=Menu(fich) on appelle me1 l'ensemble des options qui apparaissent dans fich

me1.*add_command*(*label*='Nouveau',*command*=effacer) on ajoute les différentes options avec la méthode

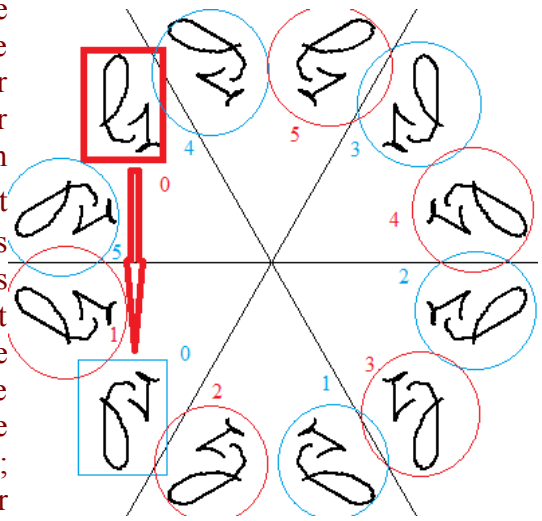
…. *add_command* qui prend en arguments un nom (label), la fonction qui sera appelée (*command*)

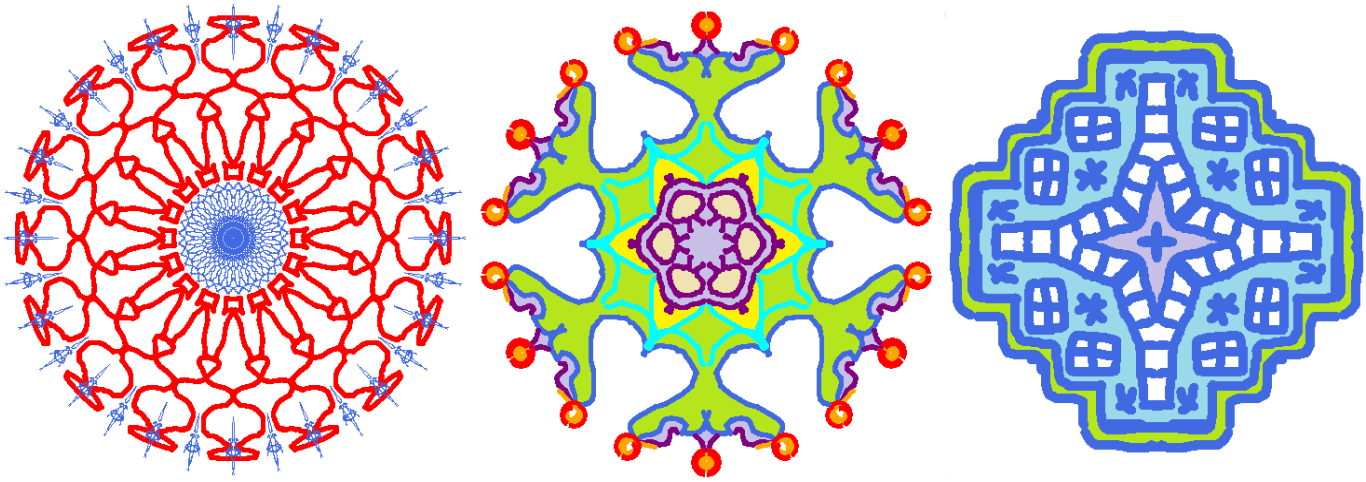
fich.*configure*(*menu*=me1) on enregistre notre menu en configurant le *Menubutton* fich avec ces options de *Menu* me1

La complexité augmente encore si l'on veut qu'une option de menu ouvre un volet contenant d'autres options de menu. Nous avons fait cela pour le *Menubutton* « Pinceau » qui contient les volets « forme », « taille » et « ordre » qui, chacun, ouvre des menus en cascade (la méthode Python est *add_cascade*). Dans le menu « forme » nous avons disposé un autre volet qui ouvre en cascade d'autres menus, l'ensemble pouvant être complexifié selon ce principe général. On peut ajouter autre chose que des menu de commande : des *checkbutton* (par la méthode *add_checkbutton*) qui sont liés à une commande mais surtout à une variable *IntVar* dont le contenu (obtenu grâce à la méthode *get*()) renseigne sur l'état de la case (1:cochée – 0:décochée) ; des *radiobutton* (par la méthode *add_radiobutton*) qui sont aussi liés à une commande et à une variable *IntVar*. Tous les *radiobutton* liés à une même variable forment un ensemble dans lequel on reconnaît l'option activée par la *value*, toujours obtenue par *get*(). On pourrait utiliser encore d'autres dispositifs (des *spinbox*, des cases d'entrée de texte, des curseurs, etc.) mais nous préférons tester le fonctionnement global du menu en n'utilisant que les dispositifs de base (nous aurions d'ailleurs pu nous limiter à la méthode *add_command*, la plus basique). Une 1ère version du programme avec un menu fonctionnant comme prévu est tout d'abord mise au point. Seule, la possibilité d'écrire dans le cadre est implémentée ici, les différentes options de pinceau seront mises au point progressivement, de même que la spécificité du dessin de la rosace (symétries+rotations).

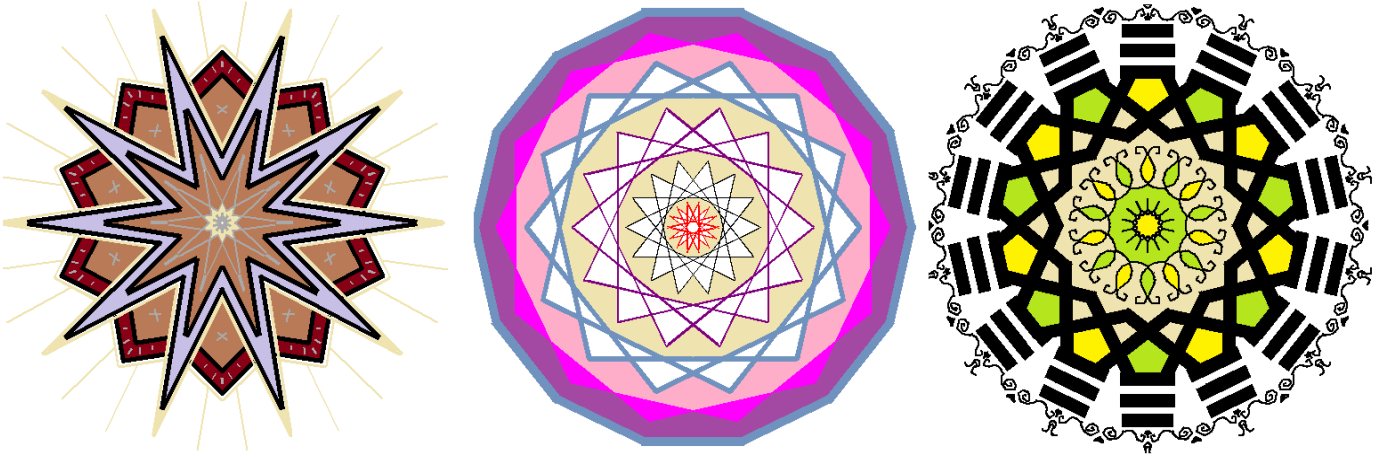
```
from tkinter import *
from math import *
def effacer():
    cadre.delete(ALL)
    if axes.get()==1:traceAxes()
def record():
    None
def clic(event):
    global coordL
    x1,y1=event.x,event.y
    coordL=[]
    if 0<x1<500 and 0<y1<500: coordL.append([x1,y1])
def drag(event):
    global coordL
    x1,y1=event.x,event.y
    coordL.append([x1,y1])
    cadre.create_line(coordL,width=taille,joinstyle=ROUND,fill=color)
def deleteGrille():
    global grilV,grilH
    for i in range(50):
        cadre.delete(grilV[i])
        cadre.delete(grilH[i])
def traceGrille():
    global grilV,grilH
    grilV,grilH=[],[]
    for i in range(50):
        grilV.append(cadre.create_line(i*10,0,i*10,500,width=1,fill='black'))
        grilH.append(cadre.create_line(0,i*10,500,i*10,width=1,fill='black'))
def traceAxes():
    global axe
    axe=[]
    for i in range(nbrAxes):
        axe.append(cadre.create_line(250+250*cos(i*pi/nbrAxes),250+250*sin(i*pi/nbrAxes),250+250*cos(pi+i*pi/nbrAxes),
                                     250+250*sin(pi+i*pi/nbrAxes),width=1,fill='black') )
def deleteAxes():
    global axe
    for i in range(nbrAxes):
        cadre.delete(axe[i])
def setCoul():
    global color
    color=col[couleur.get()]
def setPinc():
    global pinceau
    pinceau=pin[forme.get()]
def taillePinc():
    global taille
    taille=large.get()
def setOrdre():#changement du nombre d'axes
    global nbrAxes
    if axes.get()==1:deleteAxes()
    nbrAxes=nbr.get()
    if axes.get()==1:traceAxes()
def choixGrille():
    if grille.get()==1:traceGrille()
    else:deleteGrille()
def choixAxes():
    if axes.get()==1:traceAxes()
    else:deleteAxes()
fen=Tk()
fen.title('Rosaces V.1')
mBar=Frame(fen)
mBar.pack()
cadre=Canvas(fen,bg='white',height=500,width=500,borderwidth=2)
cadre.pack()
cadre.bind("<Button-1>",clic)
cadre.bind("<B1-Motion>",drag)
nbr,large,axes,grille,forme,couleur=IntVar(),IntVar(),IntVar(),IntVar(),IntVar(),IntVar()#variables tkinter
colF=['noir','bleu roi','rouge','vert clair','violet','cyan','marob','orange','jaune','vert foncé','gris']
col=['black','royal blue','red','light green','purple','cyan','maroon','orange','yellow','dark green','dark grey']
pinF=['ligne libre','segment','arc de cercle','triangle','carré','étoile']
pin=['lib','seg','arc','tri','car','etc']
nbr.set(2)
large.set(2)
couleur.set(0)
axes.set(1)
grille.set(0)
forme.set(0)
fich=Menubutton(mBar,text='Fichier',relief=RAISED)# Menu <Fichier> #
fich.pack(side=LEFT,padx='5')
me1=Menu(fich)
me1.add_command(label='Nouveau',underline=0,command=effacer)
me1.add_command(label='Enregistrer',underline=0,command=record)
me1.add_command(label='Quitter',underline=0,command=fen.quit)
fich.configure(menu=me1)
# Menu <Pinceau> #
pinc=Menubutton(mBar,text='Pinceau',relief=RAISED)# Menu <Pinceau> #
pinc.pack(side=LEFT,padx='5')
m0=Menu(pinc)
m1=Menu(m0)#couleur
for i in range(11):
    if i==0:m1.add_radiobutton(label=colF[i],foreground ='white',background=col[i],variable=couleur,value=i,command=setCoul)
    else:m1.add_radiobutton(label=colF[i],foreground ='black',background=col[i],variable=couleur,value=i,command=setCoul)
m0.add_cascade(label='couleur',underline=0,menu=m1)
pinc.configure(menu=m0)
m1=Menu(m0)#forme
m1.add_radiobutton(label='ligne libre',underline=1,variable=forme,value=0,command=setPinc)
m1.add_radiobutton(label='segment',underline=1,variable=forme,value=1,command=setPinc)
m1.add_radiobutton(label='arc de cercle',underline=1,variable=forme,value=2,command=setPinc)
m0.add_cascade(label='forme',underline=0,menu=m1)
m2=Menu(m1)
m2.add_radiobutton(label='Triangles',underline=1,variable=forme,value=3,command=setPinc)
m2.add_radiobutton(label='Carrés',underline=1,variable=forme,value=4,command=setPinc)
m2.add_radiobutton(label='Etoiles',underline=1,variable=forme,value=5,command=setPinc)
m1.add_cascade(label='polygones',underline=0,menu=m2)
pinc.configure(menu=m0)
m1=Menu(m0)#forme
m1.add_radiobutton(label='ligne libre',underline=1,variable=forme,value=0,command=setPinc)
m1.add_radiobutton(label='segment',underline=1,variable=forme,value=1,command=setPinc)
m1.add_radiobutton(label='arc de cercle',underline=1,variable=forme,value=2,command=setPinc)
m0.add_cascade(label='forme',underline=0,menu=m1)
m2=Menu(m1)
m2.add_radiobutton(label='Triangles',underline=1,variable=forme,value=3,command=setPinc)
m2.add_radiobutton(label='Carrés',underline=1,variable=forme,value=4,command=setPinc)
m2.add_radiobutton(label='Etoiles',underline=1,variable=forme,value=5,command=setPinc)
m1.add_cascade(label='polygones',underline=0,menu=m2)
pinc.configure(menu=m0)
m1=Menu(m0)#taille
for (i,lab) in [(1,'1 px'),(2,'2 px'),(5,'5 px'),(10,'10 px'),(15,'15 px'),(20,'20 px'),(30,'30 px')]:
    m1.add_radiobutton(label=lab,underline=1,variable=large,value=i,command=taillePinc)
m0.add_cascade(label='taille',underline=0,menu=m1)
pinc.configure(menu=m0)
optMenu=Menubutton(mBar,text='Options',relief=RAISED)# Menu <Options> #
optMenu.pack(side=LEFT,padx='3')
m0=Menu(optMenu)
m0.add_command(label='Activer :',foreground='blue')
m0.add_checkbutton(label='Axes de symétrie',variable=axes,command=choixAxes)
m0.add_checkbutton(label='Grille',variable=grille,command=choixGrille)
m0.add_separator()
m0.add_command(label='Symétrie :',foreground='blue')
m1=Menu(m0)#ordre
for i in range(2,25):
    m1.add_radiobutton(label='{} axes'.format(i),underline=1,variable=nbr,value=i,command=setOrdre)
m0.add_cascade(label='ordre',underline=0,menu=m1)
optMenu.configure(menu=m0)
color,pinceau,nbrAxes,taille,axe,grilV,grilH,coordL='black','lib',2,2,[],[],[],[]#variables globales
traceAxes()
fen.mainloop()
```

Pour dessiner une ligne « libre », comme avec un crayon, il faut utiliser la méthode *create_line*(). Cette méthode crée un segment si l'on donne quatre nombres comme premiers arguments (ce sera alors *create_line*(x1,y1,x2,y2)), mais si on donne une liste de coordonnées plus longue, la méthode crée une ligne apparente, constituée en réalité de micro-segments. C'est la méthode retenue, par défaut, pour dessiner dans le *Canvas* cadre. Notre objectif étant la création de rosaces, nous devons simultanément dessiner quatre fois le nombre d'axes cette ligne et ses images par symétrie et par rotation autour du centre. S'il y a trois axes, comme sur l'illustration, le motif (ℓ_1) est dessiné six fois par rotation d'un angle égal à $\frac{i\times\pi}{3}$, i variant de 0 à 5. La rotation d'un point est réalisée grâce à la fonction *rotatione*() qui travaille sur les coordonnées. Nous devons penser que les coordonnées données par le clic de souris sont dans un repère lié au cadre (l'origine est en haut à gauche) alors que nous voulons faire tourner notre point autour du centre de ce cadre. Un petit changement d'origine est donc nécessaire. Ensuite, nous utilisons une formule issue directement du cours de trigonométrie ($x'=x\cos(\alpha)+y\sin(\alpha)$; $y'=-x\sin(\alpha)+y\cos(\alpha)$). Pour les autres images, obtenues par symétrie des premières par rapport aux différents axes, nous prenons juste le symétrique (encadré en bleu) du motif original (encadré en rouge) et nous faisons subir à ce symétrique les différentes rotations. Ceci est rendu possible par le fait que l'un des axes est toujours horizontal ; les coordonnées du symétrique sont alors très simples à calculer (seul y est changé en $500-y$, si 500 est la hauteur du cadre).

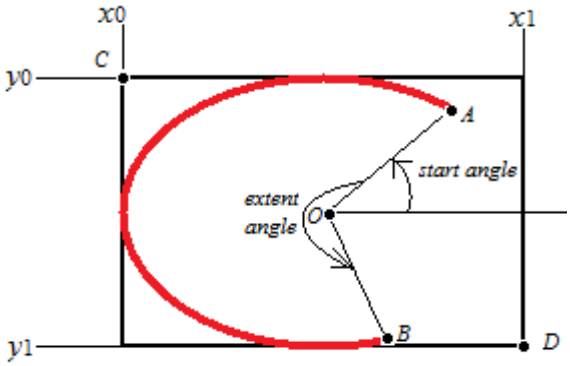




Les rosaces obtenues par ces lignes libres sont déjà assez satisfaisantes, surtout lorsqu'on les reprend dans un petit outil graphique comme le « *Paint* » de *Windows*, pour colorier l'intérieur des surfaces délimitées par nos lignes (voir notamment la rosace du milieu ci-dessus). Mais nous voulons ajouter maintenant les autres options de pinceau. La plus simple est l'option « segment » : au clic de souris, on enregistre les coordonnées du point initial pour les deux extrémités du segment dessiné, et pendant le glissé de la souris (*drag*), on modifie les coordonnées du 2ème point, ce qui a pour effet de modifier le segment dessiné au lieu d'en dessiner d'autres. Bien sûr, il faut simultanément dessiner les images de ce segment par les symétries et rotations de la rosace, mais pour cela on utilise le même principe que pour la ligne.

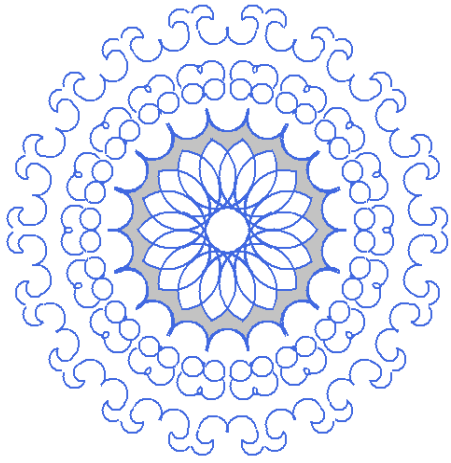
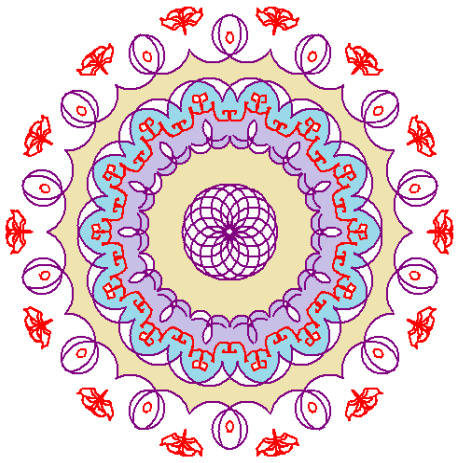
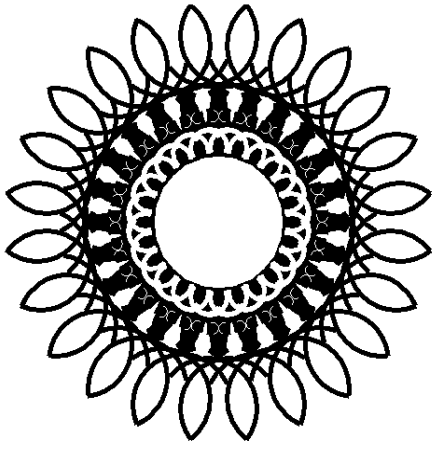


Pour dessiner les arcs de cercle, le travail mathématiques est un peu plus important, car la méthode *create_arc*() qui dessine des arcs de cercle nécessite qu'on précise un angle de départ (*start*) et un angle d'extension (*extent*). L'angle d'extension est toujours 180° si on veut dessiner des demi-cercles (c'est notre cas). Mais l'angle de départ, qui est l'angle entre le point situé le plus à droite sur le cercle et le point de départ choisi, va varier selon les valeurs des coordonnées du 2ème point et aussi, accessoirement, avec les rotations et symétries exercées sur ces points. Les coordonnées fournies à la méthode ne sont pas les coordonnées des extrémités *A* et *B* de l'arc, mais celles des deux coins de la diagonale du carré circonscrit au cercle (voir notre illustration qui montre le cas plus général d'un arc d'ellipse inscrite dans un rectangle). C'est peut-être là le point le plus difficile à régler, mathématiquement parlant. Nous laissons le lecteur cogiter notre solution qui réalise le dessin de ces arcs de cercle. Le résultat est assez saisissant et récompense largement ce petit effort cérébral.

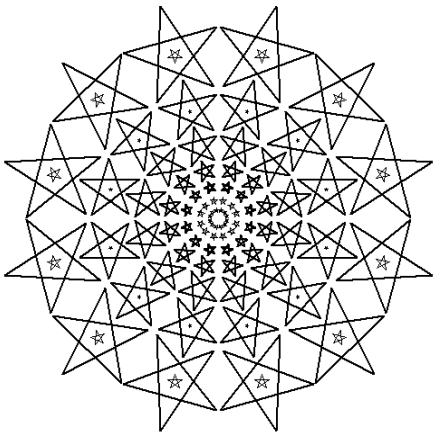
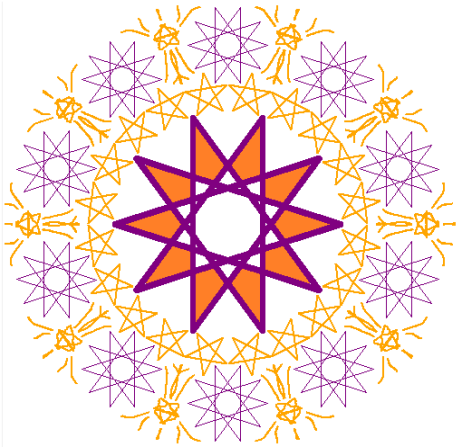
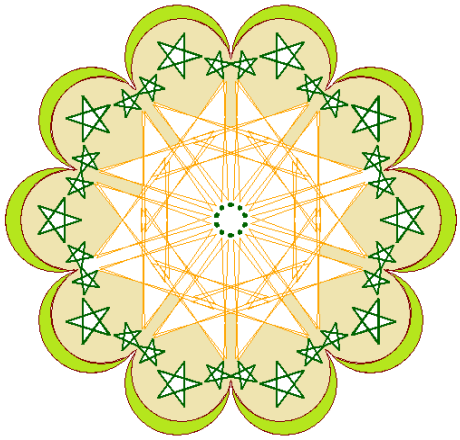



```
def clic(event):
    global coordL,ligne
    x1,y1=event.x,event.y
    if pinceau=='lib'
     .........
    elif pinceau=='arc' and 0<x1<500 and 0<y1<500:
        ligne=[]# contient les arcs (demi-cercles) définis par deux points
        coordL=[x1,y1,x1,y1]
        for i in range(4*nbrAxes):
            ligne.append(cadre.create arc(x1,y1,x1,y1,width=taille,start=0,extent=180,outline=color,style='arc',fill=''))
    .........
```

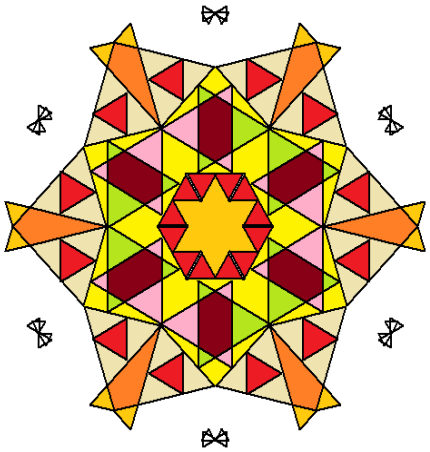
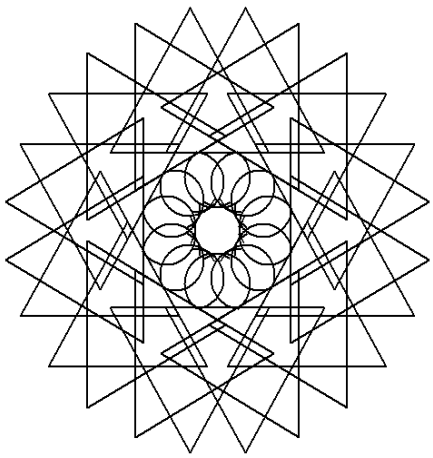
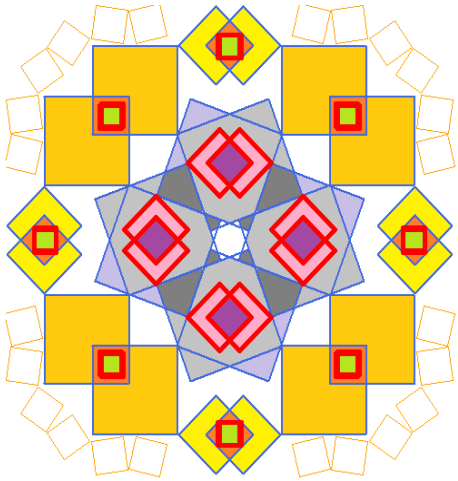
```
def drag(event):
    global coordL,ligne
    x1,y1=event.x,event.y
    if pinceau=='lib'
     .........
    elif pinceau=='arc' and 0<x1<500 and 0<y1<500:
        coordL[2],coordL[3]=x1,y1           # modification des coordonnées du 2e point
        x0,y0=(coordL[0]+coordL[2])/2,(coordL[1]+coordL[3])/2      # milieu
        R=sqrt((coordL[0]-coordL[2])**2+(coordL[1]-coordL[3])**2)/2# rayon
        x2,y2,x3,y3=x0-R,y0-R,x0+R,y0+R # coins opposés du carré contenant le cercle
        startAngle=asin((coordL[1]-coordL[3])/2/R)     # position angulaire de A si A est à droite
        if coordL[2]<coordL[0]:startAngle=pi-startAngle# position angulaire de A si A est à gauche
        for i in range(2*nbrAxes):
            cadre.itemconfigure(ligne[i],start=(startAngle+i*pi/nbrAxes)*180/pi)
            x0r,y0r=rotatione([x0,y0],i)           #seul le centre du cercle tourne
            cadre.coords(ligne[i],x0r-R,y0r-R,x0r+R,y0r+R)
        for i in range(2*nbrAxes):
            cadre.itemconfigure(ligne[2*nbrAxes+i],start=(pi-startAngle+i*pi/nbrAxes)*180/pi)
            x0r,y0r=rotatione(symetrise([x0,y0]),i)#le centre du cercle est symétrisé, puis tourne
            cadre.coords(ligne[2*nbrAxes+i],x0r-R,y0r-R,x0r+R,y0r+R)
     .........
```

Il ne nous reste plus qu'à implémenter le dessin des polygones que nous avons prévu dans notre « cahier des charges » : des triangles (équilatéraux), des carrés et des étoiles pentagonales (régulières). Cela ne devrait pas poser beaucoup de problèmes. Il s'agit juste de traduire ces figures à l'aide des coordonnées, ce qui ne nécessite pas vraiment d'explications à ce stade. Les étoiles pentagonales sont évidemment un peu moins simples à dessiner car on souhaite que les deux points du clic de souris (le 1er étant le lieu où l'on a cliqué et le 2d celui où l'on va lâcher le bouton de la souris) correspondent à deux sommets consécutifs de l'étoile (comme pour le triangle et le carré).

Les possibilités créatives de notre petit programme, prévues à notre cahier des charges, étant désormais atteintes, il ne nous reste plus qu'à régler quelques petits détails. La possibilité de gommer a été implémentée par la commande *effacer*() du menu fichier. Celle-ci se contente de supprimer tout le dessin excepté les axes que nous laissons affichés (si l'option axes est sélectionnée). Sans doute serait-il judicieux de prévoir, en plus de ce bouton « Nouveau » qui recommence tout, un autre bouton « Gommer Dernier » qui ne supprimerait que le dernier objet tracé. Comme nous avons utilisé une liste *ligne* pour tracer les différents objets (sauf les « lignes libres »), nous allons supprimer ces objets avec la nouvelle fonction *gommer*(). Nous devons donc modifier notre façon de dessiner les lignes libres pour qu'elles utilisent également la liste *ligne*. L'option de gommer le dernier objet ne peut être utilisée pour effacer tous les éléments du dessin les uns après les autres, il faudrait pour cela garder les noms de ces objets dans une autre liste. Nous avons, par contre, désactivé l'option « Gommer Dernier » quand il n'y a plus d'objet à gommer. Cette option se réactive quand on crée un objet.

La dernière fonction que nous avons à implémenter dans notre cahier des charges est le bouton « Enregistrer » qui doit générer un fichier image à partir du dessin actuel. Pour faire cela directement avec le programme, ce n'est pas très facile. Pour les besoins de ce petit fascicule, nous avons créé nos images en faisant des copies d'écran (cela met l'image dans le presse-papier, on peut alors la « coller » dans un logiciel tel que « *Paint* », la retravailler et l'enregistrer au format souhaité ou bien l'importer dans un traitement de texte). Si on veut vraiment réaliser l'enregistrement à partir du programme, la méthode la plus simple produit un fichier au format *ps* (*postscript*, format utilisé par les imprimantes) qui n'est pas un format d'image valide. La transformation du format *ps* en format *pdf* (facile à ouvrir et à imprimer) demande, au préalable, d'avoir téléchargé et installé *Ghostscript*, le programme qui va effectuer la conversion. Après installation de ce programme (on doit, sous Windows, déclarer le *path* dans les variables d'environnement pour indiquer le chemin vers les dossiers *lib* et *script* du programme), on peut implémenter notre fonction *record*() avec le code qui suit :

```
image=cadre.postscript(file="saved.ps",height=500,width=500,colormode="color")
process=Popen(["ps2pdf","saved.ps","iRosace.pdf"],shell=True)
process.wait()
remove("saved.ps")
```

Il faut tout de même importer des modules de Python mais ce sont des modules standard (déjà présents et

utilisables) : *from subprocess import Popen* et *from os import* *. La méthode *Popen* du module *subprocess* ouvre le script *ps2pdf* du programme *Ghostscript*, tandis que, du module *os*, on n'utilise que la fonction *remove* qui supprime le fichier intermédiaire *saved.ps*. À ce stade, il devrait rester le fichier *iRosace.pdf* dans le répertoire où se situe votre programme. Plus simple (à mon humble avis) que cet enregistrement en *pdf* (on voulait du *png* au départ) : enregistrer le fichier *ps* et l'ouvrir avec un logiciel comme *ImageMagick* (à télécharger et installer s'il ne se trouve pas déjà sur votre ordinateur). La conversion en un format image ne pose alors aucun problème.
Ajoutons la partie qui va permettre de créer directement le fichier à l'emplacement voulu avec un nom choisi au lieu d'un nom prédéfini ou, pire, d'un nom qui serait demandé à l'utilisateur par le biais de la console (le *shell*). Il faut ouvrir le répertoire courant de l'ordinateur et attendre que l'utilisateur saisisse un nom pour le fichier (l'extension *ps* étant ajoutée automatiquement). Cela se fait facilement grâce à la méthode *asksaveasfilename*() du sous-module *tkinter.filedialog* que l'on doit importer (*import tkinter.filedialog*). Tant que nous y sommes, pourquoi ne pas offrir la possibilité d'ouvrir un fichier image existant (en créant un sous-menu « Ouvrir » à côté du sous-menu « Enregistrer ») pour servir de fond à notre *Canvas* cadre ? Nous pourrions alors dessiner nos rosaces sur des photos, des dessins... La méthode cadre.*create_image*(xCentre,yCentre,*image*=photo) accepte des photos de différents types (*gif*, *png*) mais pas des images *ps* (ce sont pourtant les seules images crées par la méthode *postscript* de *tkinter* !) ni des images *jpeg*. La photo qui est donnée comme valeur du paramètre image est définie par l'instruction photo=*PhotoImage*(*file=tkinter.filedialog.askopenfilename*()).

Voilà notre programme terminé, ce qui clôture ce petit fascicule de questions de programmation.
PM-2015