Courbes et surfaces

Calcul formel – TP 6

1. IFS

1. (a) Cette question ne pose pas de problème particulier :

```
def transformation(x,y,a,b,c,d,e,f):
    xx = a*x + b*y + e
    yy = c*x + d*y + f
    return xx, yy
```

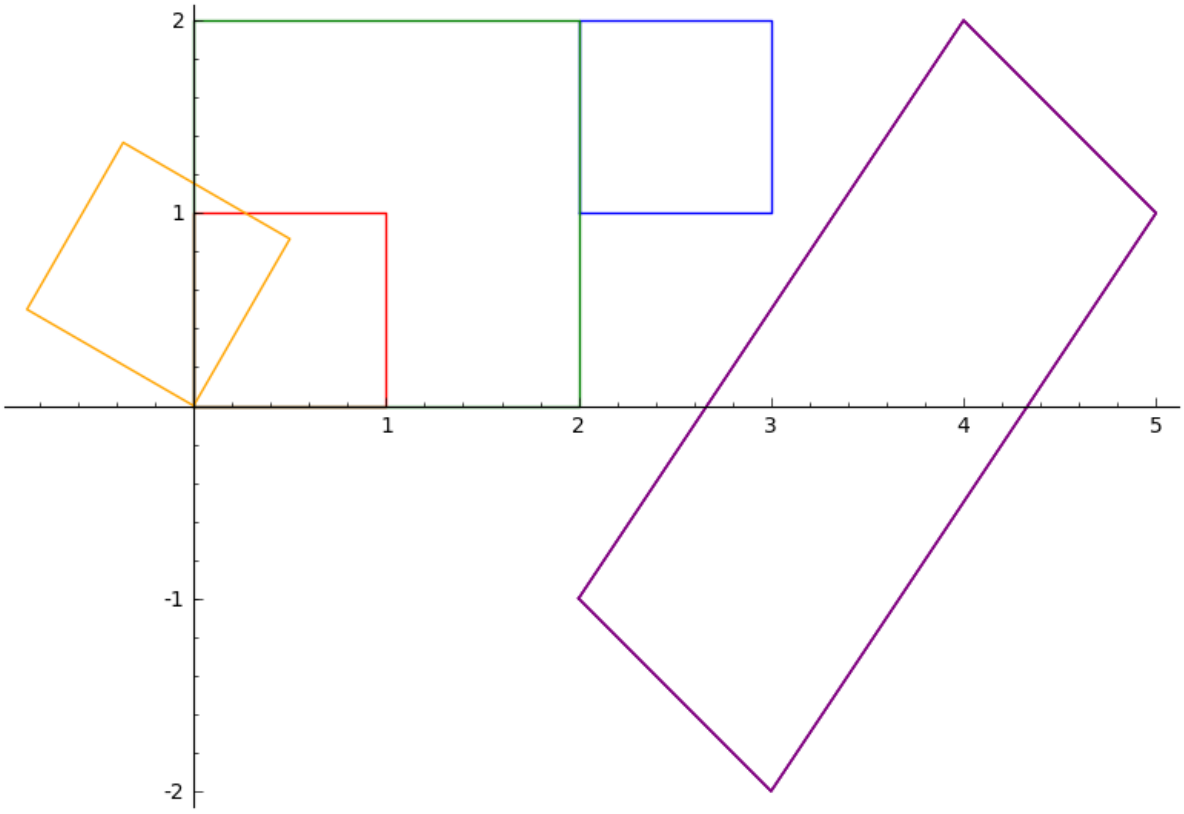
(b) On commence par définir l'image du carré par une transformation affine :

```
def image_carre(a,b,c,d,e,f):
    P1 = transformation(0,0,a,b,c,d,e,f)
    P2 = transformation(1,0,a,b,c,d,e,f)
    P3 = transformation(1,1,a,b,c,d,e,f)
    P4 = transformation(0,1,a,b,c,d,e,f)
    return P1, P2, P3, P4, P1
```

Ensuite, on calcule le tracé des côtés du carré initial et de ses transformés, puis on affiche le tout :

```
# Carre initial
G = line([(0,0),(1,0),(1,1),(0,1),(0,0)], color='red')

# Translation
a=1; b=0; c=0; d=1; e=2; f=1
G = G + line(image_carre(a,b,c,d,e,f), color='blue')

# Homothetie
a=2; b=0; c=0; d=2; e=0; f=0
G = G + line(image_carre(a,b,c,d,e,f), color='green')

# Rotation
theta = pi/3
a=cos(theta); b = -sin(theta); c=sin(theta); d=cos(theta); e=0; f=0
G = G + line(image_carre(a,b,c,d,e,f), color='orange')

# Transformation affine quelconque
```

```
a=1; b=2; c=-1; d=3; e=2; f=-1
G = G + line(image_carre(a,b,c,d,e,f), color='purple')

G.show(aspect_ratio=1)
```



2. Étant donnée une liste de nombres réels $(p_0, \cdots, p_n)$ de somme 1, la fonction suivante renvoie l'indice k tel que le nombre x tiré au hasard vérifie $p_0 + \cdots + p_{k-1} \leqslant x < p_0 + \cdots + p_k$:

```
def hasard(pliste):
    x = random()
    k = 0
    while k < len(pliste) and x > pliste[k]:
        x = x - pliste[k]
        k = k + 1
    if k >= len(pliste):
        k = k - 1
    return k
```

Avec nos notations, la longueur de la liste est $n+1$, la condition finale sert à éviter que la fonction ne nous retourne le rang $n+1$ si x est dans le dernier intervalle $[p_0 + \cdots + p_{n-1}, p_0 + \cdots + p_n[$, mais retourne bien le rang n.

3. Voici notre fonction pour tracer un IFS avec en entrée une liste de transformations, chacune munie d'une probabilité. Chaque élément de la liste est du type `[a,b,c,d,e,f,p]`.

```
def ifs(transfoliste,nbiter):
    pliste = [transfo[6] for transfo in transfoliste] # Les probas
    x=0; y=0
    listepoints = []
```

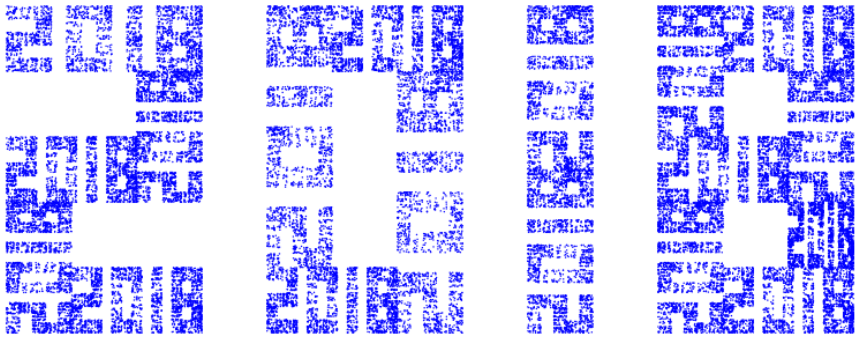
```
    for i in range(nbiter):                    # Iteration
        k = hasard(pliste)                     # On choisit une tranformation au hasard
        transfo = (transfoliste[k])[0:6]       # Voici la transformation
        x,y = transformation(x,y,*transfo)     # L'image du point
        if i > 100:                            # On n'affiche pas les premiers points
            listepoints.append((x,y))
    return listepoints
```

4. Par exemple, pour le triangle de Sierpinski, il suffit de faire

```
sierpinski = [
[0.5,0,0,0.5,0,0,0.33],
[0.5,0,0,0.5,0.5,0,0.33],
[0.5,0,0,0.5,0.25,0.5,0.34]
]

monifs = ifs(sierpinski,50000)
G = points(monifs,size=1)
G.show(aspect_ratio=1,axes=False)
```

   pour voir s'afficher l'IFS de l'énoncé.

5. **Énigme.** En reprenant la démarche de la question précédente, on trouve l'IFS suivante :



   Réponse attendue 2018.

2. Loxodromie de la sphère

1. (a) La commande `parametric_plot3d` s'utilise de manière naturelle, en définissant au préalable les paramètres comme variables :

```
var('u,v')
G = parametric_plot3d([cos(u)*cos(v),cos(u)*sin(v),sin(u)],
                      (u,-pi/2,pi/2),(v,-pi,pi))
```

   (b) Traitons par exemple le cas des méridiens. Pour tracer un méridien, il faut donc bloquer le paramètre v. Plutôt que d'en tracer un seul, on va en tracer 24 équidistants (un pour chaque fuseau horaire). On est amené à faire une boucle pour chaque valeur du paramètre bloqué v :

```
for myv in srange(-pi,pi,pi/24):
    G = G+parametric_plot3d([cos(u)*cos(myv),cos(u)*sin(myv),sin(u)],
                            (u,-pi/2,pi/2), color='black', thickness=2)
```

On peut afficher le résultat avec :

```
G.show()
```

2. La commande pour tracer une courbe paramétrique en 3D est la même qu'à la question précédente :

```
alpha = pi/4  # Cap (= angle par rapport au Nord)
V = tan(alpha)*ln(tan(u/2+pi/4))
G = G + parametric_plot3d([cos(u)*cos(V(u=u)),cos(u)*sin(V(u=u)),sin(u)],
                          (u,0,pi/2), plot_points=200, color='red', thickness=5)
```

3. On peut simplement reprendre le code précédent en remplaçant la côte par 0 :

```
V = tan(alpha)*ln(tan(u/2+pi/4))
H = parametric_plot3d([cos(u)*cos(V(u=u)),cos(u)*sin(V(u=u)),0],
                      (u,0,pi/2),plot_points=500,color='red',thickness=1)
#Affichage
H.show()
```

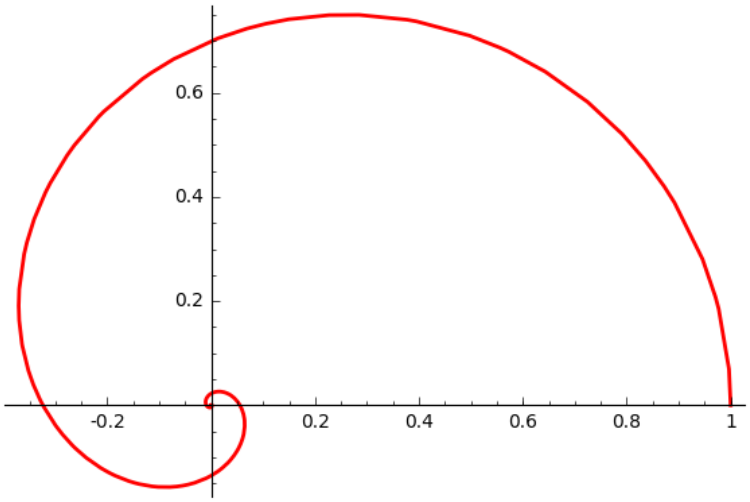
Cependant, le résultat en 3D n'est pas très lisible. Pour un meilleur rendu en 2D, on montre que la projection orthogonale de la loxodromie sur le plan Oxy est la spirale de Poinsot d'équation polaire :

$$r = \frac{1}{\mathrm{ch}(k\theta)}$$

avec $k = \pi/2 - \tan(\alpha)$, où α est le cap (angle par rapport au Nord), ici $\pi/4$. On peut donc l'obtenir avec :

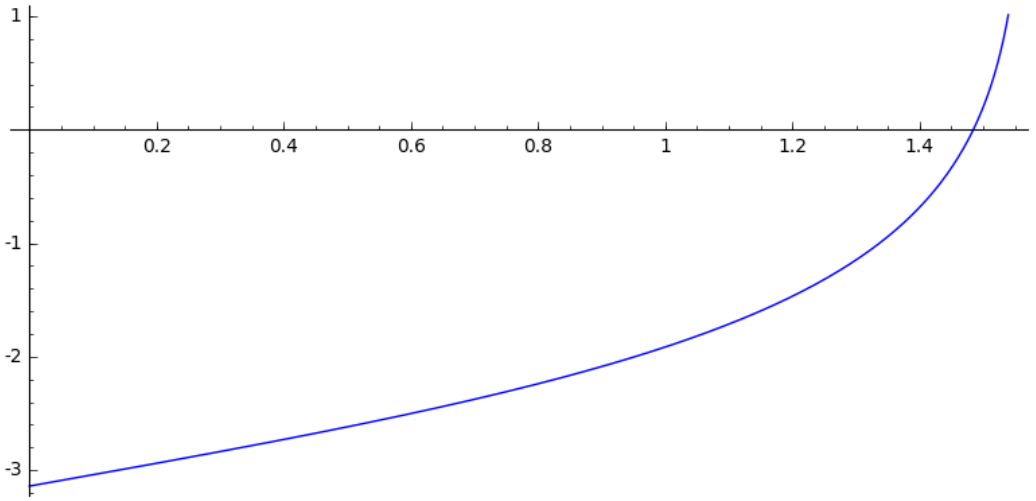
```
var('x')
k = pi/2-tan(alpha)
GG = polar_plot(1/cosh(k*x),
                (x, 0, 12*pi), color='red', plot_points=500, thickness=2)
#Affichage
GG.show(aspect_ratio=1)
```

ce qui donne le résultat suivant :



4. **Énigme.** On cherche le premier $u > 0$ tel que $V(u) = \pi$. On pose donc `f = V - pi`. À présent, comme le direct `print(solve(f==0,u))` échoue, on localise la solution à l'aide du graphe de V :

```
GGG = plot(f,(u,0,pi/2-pi/100))
GGG.show()
```



On a donc une solution dans l'intervalle $[0, \pi/2[$. On peut alors chercher une solution approchée en radians puis en degrés par :

```
u180 = f.find_root(0,pi/2)
print(u180, 'radians')
print((u180*180/pi).n(), 'degres')
```

Réponse attendue 85.

5. On rappelle qu'on a défini `V = tan(alpha)*ln(tan(u/2+pi/4))` avec `alpha = pi/4`. Notre courbe est paramétrée par :

   ```
   x = cos(u)*cos(V)
   y = cos(u)*sin(V)
   z = sin(u)
   ```

   On calcule les dérivées :

   ```
   xx = diff(x,u)
   yy = diff(y,u)
   zz = diff(z,u)
   ```

   puis l'intégrale formellement :

   ```
   ll = sqrt( xx^2+yy^2+zz^2 )
   L = integral( ll, u, 0, pi/2)
   print(L)
   ```

   On trouve : $\frac{\pi\sqrt{2}}{2}$.

3. Théorème de Pappus

1. La fonction suivante donne un triplet (a, b, c) tel que $ax + by + c = 0$ soit l'équation de la droite (MN) (un tel triplet est déterminé à multiplication par un scalaire non nul près).

```
def droite(M,N):
    xm,ym = M
    xn,yn = N
    a = (yn-ym)
    b = -(xn-xm)
    c = -(a*xm+b*ym)
    return (a,b,c)
```

2. Il suffit d'appliquer les formules de Cramer :

```
def intersection(D,DD):
    a,b,c = D
    aa,bb,cc = DD
    deter =  a*bb-aa*b
    if deter == 0:
        print 'Droites␣paralleles'
        return 0,0
    x = (cc*b-c*bb)/deter
    y = (aa*c-a*cc)/deter
    return (x,y)
```

3. On calcule les coordonnées des deux vecteurs $\overrightarrow{BA}$ et $\overrightarrow{CA}$, puis leur déterminant :

```
def sont_alignes(A,B,C):
    a,b = A[0]-B[0],A[1]-B[1]
    c,d = A[0]-C[0],A[1]-C[1]
    deter = a*d-b*c
    if deter == 0:
        return True
    else:
        return False
```

4. On commence par définir les coordonnées des 6 points donnés :

```
# Points A, B = k A, C = l A
var('xa,ya,k,l')
A = (xa,ya)
B = (k*xa,k*ya)
C = (l*xa,l*ya)

# Points D, E = kk D, F = kk L
var('xd,yd,kk,ll')
D = (xd,yd)
E = (kk*xd,kk*yd)
F = (ll*xd,ll*yd)
```

puis les coordonnées des 3 points d'intersection :

```
# P = intersection des droites (AE) et (BD)
P = intersection(droite(A,E),droite(B,D))

# Q = intersection des droites (AF) et (DC)
Q = intersection(droite(A,F),droite(D,C))

# R = intersection des droites (BF) et (EC)
R = intersection(droite(B,F),droite(E,C))
```

On peut alors tester l'alignement :

```
# Preuve du theoreme de Pappus
print 'Le␣theoreme␣de␣Pappus␣est-il␣vrai␣?'
print sont_alignes(P,Q,R)
```

Sage renvoie `True` : on a bien prouvé le résultat formellement.

5. **Énigme.**

   Il suffit de reprendre le code précédent pour ces valeurs explicites :

```
xa = 1
ya = 0
k = 5
l = 6
A = (xa,ya)
B = (k*xa,k*ya)
C = (l*xa,l*ya)
xd = 1
yd = 1
kk = 2
ll = 3
D = (xd,yd)
E = (kk*xd,kk*yd)
F = (ll*xd,ll*yd)
P = intersection(droite(A,E),droite(B,D))
Q = intersection(droite(A,F),droite(D,C))
R = intersection(droite(B,F),droite(E,C))
print P, Q, R
```

   Sage renvoie : `(13/9, 8/9) (27/17, 15/17) (9/2, 3/4)` .

   Les dénominateurs des abscisses sont 9, 17 et 2.

   Réponse attendue 306**.**

4. Le cœur des tables de multiplications

1. On commence par définir les éléments de la table de la multiplication par b modulo n :

```
def table(b,n):
    return [ (a*b) % n for a in range (n)]
```

Ensuite, on peut définir la visualisation proposée à l'aide de la fonction :

```
def dessine_table(b,n):
    G = circle((0,0),1,color='orange')
    mespoints =  [ (cos(2*pi*a/n).n(),sin(2*pi*a/n).n()) for a in range(n)]
    G = G + points(mespoints,color='red',size=15)
    matable = table(b,n)
    messegments = [ [ mespoints[a],mespoints[matable[a]] ] for a in range(n)]
    for seg in messegments:
        G = G + line(seg)
    for a in range(n):
         G = G + text(a,(1.1*cos(2*pi*a/n).n(),1.1*sin(2*pi*a/n).n()))
    return G
```

La démarche suivie est claire : on commence par inclure dans le graphe : le cercle trigonométrique, les n points équirépartis (en partant de $(1,0)$), puis pour chaque indice a dans $0,\ldots,n-1$, le segment reliant les sommets numérotés de a à ab modulo n. La dernière boucle ne fait que rajouter à chaque point une « étiquette » avec l'entier a correspondant.

On peut alors faire un test d'affichage avec :

```
b = 2
n = 100
G = dessine_table(b,n)
G.show(aspect_ratio=1,axes=False)
```

2. **Énigme.** La fonction de comptage suivante permet de dénombrer exactement le nombre de segments recherché, en évitant les segments réduits à un point et les répétitions :

```
def compte_table(b,n):
    messegments = []
    for a in range(n):
        c = (a*b) % n
        if a < c:
            if [a,c] not in messegments:
                messegments.append([a,c])
        if c < a:
            if [c,a] not in messegments:
                messegments.append([c,a])
    return  len(messegments)
```

Avec l'application numérique :

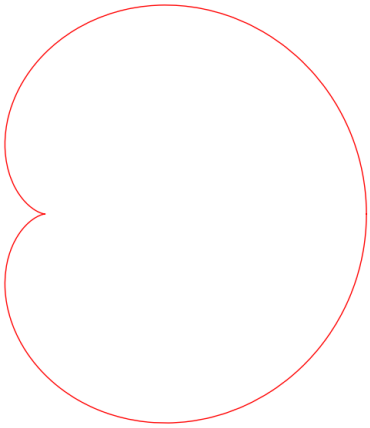
```
b = 29
n = 112
print "Nombre␣de␣segments"
print compte_table(b,n)
```

Réponse attendue 70.

3. Le plus commode est de trouver l'équation paramétrique de $\mathscr{C}$ pour la tracer :

```
var('t')
H = parametric_plot( (2/3*(1+cos(t))*cos(t) -1/3 , 2/3*(1+cos(t))*sin(t)),
                     (t,0,2*pi),color='red')
H.show(aspect_ratio=1,axes=False)
```

ce qui donne



4. (a) On commence par définir le point $N(t)$ du cercle :

```
var('t')
xN = cos(t)
yN = sin(t)
N = vector([xN,yN])
```

puis le point $N' = N(2t)$ du cercle à vitesse double :

```
xNN = cos(2*t)
yNN = sin(2*t)
NN = vector([xNN,yNN])
```

Et enfin, le point M(t) de la cycloïde :

```
xM = 2/3*(1+cos(t))*cos(t) - 1/3
yM = 2/3*(1+cos(t))*sin(t)
M =  vector([xM,yM])
```

On teste l'alignement avec un déterminant :

```
deter = (xN-xM)*(yNN-yM) - (xNN-xM)*(yN-yM)
print "Points␣alignes␣?"
print(deter.simplify_trig())
```

Cela fait 0 après simplification : les points sont donc bien alignés.

(b) On calcule un vecteur directeur de la droite $(N(t)N'(t))$:

```
vx = xNN-xN
vy = yNN-yN
```

puis $x'(t)$ et $y'(t)$:

```
xxM = diff(xM,t)
yyM = diff(yM,t)
```

On peut conclure en utilisant un déterminant

```
deter_bis = vx*yyM - vy*xxM
print "Droites␣paralleles␣?"
print(deter_bis.simplify_trig())
```

Cela fait 0 après simplification : les droites sont donc bien parallèles.

(c) Les deux questions précédentes montrent que la tangente à la cardioïde $\mathscr{C}$ au point $M(t)$ est la droite $(N(t)N'(t))$.