Algèbre linéaire

Calcul formel – TP 5

1. Matrices circulantes

1. La fonction suivante conviendra :

   ```
   def circulante(liste):
       n = len(liste)
       M = [liste[k:]+liste[:k] for k in range(n,0,-1)]
       return matrix(M)
   ```

   Expliquons la : l'entrée est une liste dont on récupère la longueur n. On définit ensuite une matrice M comme une liste de listes. L'indice k va varier de n à 1 (en décroissant car le pas est -1, on s'arrête une étape avant 0). Par exemple la dernière ligne sera donnée par `liste[1:]+liste[:1]`, c'est-à-dire tous les éléments de la liste initiale à partir de a_1, puis a_0 ; c'est bien la dernière ligne dont on a besoin.

2. L'entrée est cette fois-ci une matrice. L'idée est simple : on en extrait la première ligne, on construit la matrice circulante associée puis on teste l'égalité avec la matrice donnée :

   ```
   def is_circulante(M):
       n = M.nrows()
       liste = list(*M[0,:])
       N = circulante(liste)
       return M-N == matrix(n)
   ```

   On écrit `liste = list(*M[0,:])` pour obtenir une liste à partir de la matrice ligne formée de la première ligne de M.

3. Grâce à la fonction qu'on vient de définir, on arrive facilement à tester formellement le fait que le produit de deux matrices circulantes l'est encore :

   ```
   var('a0,a1,a2,a3,a4,b0,b1,b2,b3,b4')
   M = circulante([a0,a1,a2,a3,a4])
   N = circulante([b0,b1,b2,b3,b4])
   is_circulante(M*N)
   ```

4. **Énigme.**
   On commence par définir une matrice M de taille 5 quelconque, la première ligne de la matrice J, la matrice J elle-même, ainsi que ses puissances :

```
n = 5
var('a0,a1,a2,a3,a4')
M = circulante([a0,a1,a2,a3,a4])
liste = [0,1]+[0]*(n-2)
J = circulante(liste)
for k in range(1,n+1):
    print 'J^',k,'\n',J^k
```

On constate que la première ligne de J^k admet un 1 en position k et des zéros ailleurs, ce qui ne laisse aucun choix pour la décomposition $M = \sum_{k=0}^{n-1} a_k J^k$: le coefficient a_k doit être celui en position k de la première ligne de M. Il suffit donc maintenant de définir `N = a0*J^0+a1*J^1+a2*J^2+a3*J^3+a4*J^4` puis de tester son égalité avec M.

Réponse attendue : 5.

2. Le 20 000^{ème} nombre bis de Fibonnacci

On vérifie immédiatement que $AX_n = X_{n+1}$. On en déduit par récurrence que $X_n = A^n X_0$.

1. On réécrit le pseudo-code en un code pour Sage :

```
def puissance(A,n):
    produit = identity_matrix(A.nrows())
    puissance = A
    while n > 0:
       if is_odd(n):
           produit = produit * puissance
       puissance = puissance * puissance
       n = n//2
    return produit
```

   Si $n = \sum_{k\geqslant 0} a_k 2^k$ est le développement de n en base 2, alors la variable `produit` prend successivement les valeurs $I, A^{a_0}, A^{2a_1+a_0}, A^{4a_2+2a_1+a_0}$, ... , et la boucle s'arrête quand on a épuisé les chiffres du développement binaire. On obtiendra donc bien A^n.

2. **Énigme.**
   Il suffit d'ajouter un compteur à la fonction précédente :

```
def puissance_compteur(A,n):
    produit = identity_matrix(A.nrows())
    puissance = A
    compteur = 0
    while n > 0:
       if is_odd(n):
           produit = produit * puissance
           compteur = compteur + 1
       puissance = puissance * puissance
       compteur = compteur + 1
       n = n//2
    return produit, compteur
```

   Réponse attendue : 20.

   On aurait pu, en raffinant l'algorithme, éviter deux multiplications :

- la première multiplication `produit = produit * puissance` rencontrée ne doit pas être comptée car `produit` est alors matrice identité ;
- la dernière puissance calculée `puissance = puissance * puissance` est inutile car alors ce dernier élément n'est pas utilisé par la suite (sortie de la boucle `for`).

3. On peut utiliser les instructions suivantes :

```
X0 =  vector([2,0,3])
A = matrix([[0,1,1],[1,0,0],[0,1,0]])
n = 20000
An = puissance(A,n)
element = (An*X0)[2]
print element, floor(log(element,10))+1
```

Évidemment, le nombre obtenu est assez impressionnant, il contient 2443 chiffres décimaux (le calcul de la longueur a déjà été présenté précédemment).

3. Loi de Laplace

1. Clairement, $\ln P + \gamma \ln V = c$.
2. On commence par regrouper les données en une liste de points (P, V) :

```
points = [ [120, 2], [67, 3], [30, 5], [15, 8], [12, 10] ]
```

On la convertit en une liste $(\ln P, \ln V)$:

```
logpoints = [ [ln(p[0]).n(),ln(p[1]).n()] for p in points ]
```

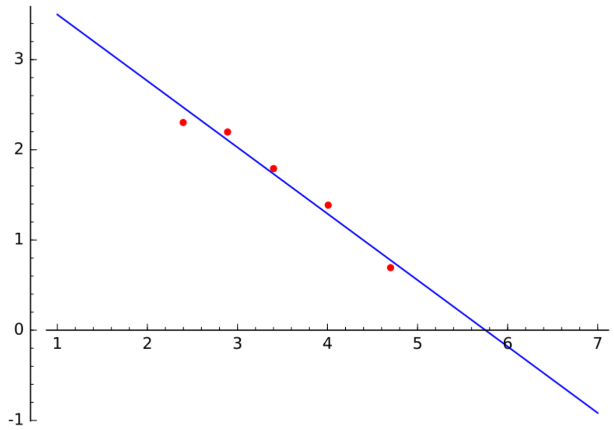
On applique la formule des moindres carrés : on cherche un vecteur $X = \binom{c}{\gamma}$ vérifiant $AX - B$ le plus proche de zéro possible, où A est la matrice de taille 5×2 dont les lignes sont les $(1, -\ln V)$ et B est la matrice de taille 5×1 formée par les $\ln P$. On sait d'après le cours que $X = (A^T A)^{-1} A^T B$.
On pose donc :

```
A = matrix([ [1,-p[1]] for p in logpoints ])
B = vector([ p[0] for p in logpoints ])
```

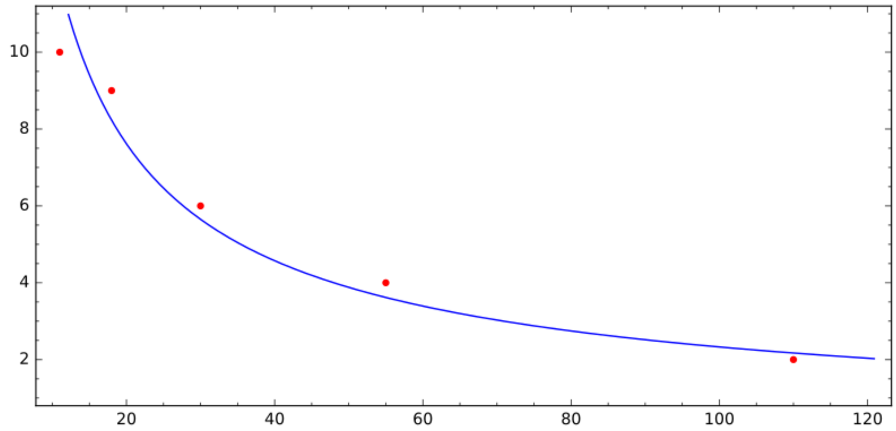
puis

```
def moindres_carres(A,B):
    return (A.transpose()*A)^-1 * A.transpose() * B

X = moindres_carres(A,B)

print('Solutions')
print 'C', exp(X[0])
print 'gamma', X[1]
```

La valeur obtenue est $\gamma = 1.4603\ldots$
Voici la droite obtenue qui interpole notre problème linéaire en coordonnées $(\ln P, \ln V)$.



Voici la courbe obtenue pour notre problème initial encoordonnées (P, V) :



3. **Énigme.**

   On peut par exemple chercher les premières fractions continues partielles de γ :

```
logC,gamma = X
cont_frac = QQ(gamma).continued_fraction()
for k in range(1,7):
    print cont_frac.convergent(k)
```

   On obtient la suite 3/2, 4/3, 15/11, 19/14, 72/53, 379/279. Le rationnel 92/63 est un candidat naturel, et on peut vérifier que l'on ne trouvera pas mieux :

```
frac_gamma = 1

for p in srange(1,100):
    for q in srange(1,100):
        diff = gamma - p/q
        if abs(( gamma-p/q ).n()) < abs((gamma-frac_gamma).n()):
            print 'mieux', p,q
            frac_gamma=p/q
```

   Réponse attendue : 7253.

4. Transformations 3D

1. On commence par définir les trois ensembles de sommets, arêtes et faces du cube initial :

```
sommets = [[0,0,0],[1,0,0],[1,1,0],[0,1,0],[0,0,1],[1,0,1],[1,1,1],[0,1,1]]
aretes = [[0,1],[1,2],[2,3],[3,0],[4,5],[5,6],[6,7],[7,4],[0,4],[1,5],[2,6],[3,7]]
faces = [[4,5,6,7],[0,1,2,3],[0,1,5,4],[1,2,6,5],[2,3,7,6],[3,0,4,7]]

cube = sommets
```

Ici $[0,1]$ dans la liste des arêtes signifie qu'il y a une arête joignant les deux premiers sommets de la liste. On utilise le même porcédé pour les faces. Les deux dernières listes ne vont pas changer après transformation, contrairement à la première. On définit les transformations du cube par deux fonctions, tout d'abord la transformation par une matrice :

```
def transformation(A,cube):
    transfo_cube = [ A*vector(point) for point in cube ]
    return transfo_cube
```

puis la translation par un vecteur :

```
def translation(v,cube):
    transfo_cube = [ vector(point)+vector(v) for point in cube ]
    return transfo_cube
```

Pour l'affichage, en s'inspirant de l'exemple, on peut par exemple utiliser la fonction suivante :

```
def affiche_cube(sommets):
    # Sommets
    G = point(sommets, size=15)
    # Aretes
    for arete in aretes:
        if sommets[arete[0]] != sommets[arete[1]]:
            monarete = [ sommets[arete[0]], sommets[arete[1]] ]
            G = G + line(monarete, thickness=5)
    # Faces
    macouleur = 'red'
    monopacite = 1

    for face in faces:
        maface = [ sommets[i] for i in face ]
        G = G + polygon3d(maface, color=macouleur,opacity=monopacite)
        macouleur = 'blue'
        monopacite = 0.1
    return G
```

La condition apparaissant dans le tracé des arêtes consiste à éviter de tracer une arête entre deux points "identiques" du cube (par exemple deux points identifiés par une projection).

2. On commence par définir les transformations :

```
# Homothetie
k = 3
A0 = k*identity_matrix(QQ,3)
```

```
# Rotation autour de l axe (Oy)
theta = -pi/6
A2 = matrix([[cos(theta),0,sin(theta)],[0,1,0],[-sin(theta),0,cos(theta)]])

# Translation
v = (4,3,2)

# Rotation autour de l axe (Ox)
theta = pi/4
A1 = matrix([[1,0,0],[0,cos(theta),-sin(theta)],[0,sin(theta),cos(theta)]])
```

Puis on définit les transformés successifs du cube et la hauteur minimale du sommet du dernier cube :

```
cube0 = transformation(A0,cube)
cube1 = transformation(A1,cube0)
cube2 = translation(v,cube1)
cube_enigme = transformation(A2,cube2)
print 'coordonnee␣z␣la␣plus␣basse␣d␣un␣sommet', min([c[2] for c in cube_enigme])
```

On pourrait afficher les cubes grâce à la fonction de la question précédente.

3. **Énigme.**
   On définit l'intensité comme un produit scalaire :

```
def lumiere(vecteur_lumiere,vecteur_face):
    u = vecteur_face/norm(vecteur_face)
    v = vecteur_lumiere/norm(vecteur_lumiere)
    return abs(u.dot_product(v))
```

On trouve un vecteur normal à la transformée de la face supérieure en normalisant un produit vectoriel :

```
def vecteur_face_superieure(cube):
    # Trois points de la face "superieure" rouge
    P, Q, R = cube[4], cube[5], cube[6]
    # Deux vecteurs de la face
    u, v = vector(Q)-vector(P), vector(R)-vector(P)
    # Vecteur normal
    w = u.cross_product(v)
    # On le rend unitaire
    w = w/norm(w)
    return w
```

À l'aide de ces fonctions, on peut calculer l'intensité voulue par :

```
vecteur_lumiere = vector((0,0,-1))
vecteur_face = vecteur_face_superieure(cube_enigme)
intensite = lumiere(vecteur_lumiere,vecteur_face)
intensite = intensite.full_simplify()
print(intensite, intensite.n())
```

L'intensité vaut $\frac{1}{4}\sqrt{6}$, soit environ 61%.

Réponse attendue : 61.

4. On procède comme à la deuxième question en utilisant la projection sur le plan.