Polynômes

Calcul formel – TP 8

1. Racines évidentes d'un polynôme

1. On commence par définir l'anneau de polynômes dans lequel on travaille : c'est-à-dire l'indéterminée X et les coefficients $\mathbb{Q}$.

```
R.<X> = QQ[]
P = 45*X^5 + 71*X^4 - 126*X^3 - 138*X^2 + 72*X  - 8
```

On écrit une fonction qui renvoie la liste des racines rationnelles potentielles :

```
def racines_rationnelles_potentielles(P):
    azero = P[0]
    an = P[P.degree()]
    ensemble = Set()
    for p in divisors(azero):
        for q in divisors(an):
            ensemble = ensemble.union(Set([p/q,-p/q]))
    return ensemble
```

On définit les coefficients extrêmes du polynôme, on initialise l'ensemble des racines potentielles à l'ensemble vide, puis on fait une double boucle pour lister tous les nombres rationnels $\frac{p}{q}$, avec $p|a_0$ et $q|a_n$. L'avantage d'utiliser les ensembles `Set` par rapport aux listes est qu'il n'y a pas d'éléments redondants.

2. Il suffit de déterminer, parmi les racines potentielles, celles qui sont effectivement racines :

```
def racines_rationnelles(P):
    racines_possibles = racines_rationnelles_potentielles(P)
    racines = Set()
    for x in racines_possibles:
        if P(X=x)==0:
            racines = racines.union(Set([x]))
    return racines
```

3. Pour le polynôme donné, les lignes suivantes

```
racines_verifiees = racines_rationnelles(P)
print racines_verifiees
print P.roots()
```

renvoient

```
{2/9, 1/5, -2}
[(2/9, 1), (1/5, 1), (-2, 1)]
```

On a bien trouvé les racines rationnelles de P (la commande `roots()` donne en plus les multiplicités).

4. **Énigme.** Il suffit de faire une boucle pour ne retenir que les valeurs de k convenables :

```
for k in srange(0,1000):
    P = 336*X^5 + 208*X^4 + k*X^3 - 147*X^2 - 19*X + 10
    zeros = racines_rationnelles(P)
    if len(zeros) >= 3:
        print k
```

On ne trouve qu'une seule valeur.

Réponse attendue : 179.

2. Polynômes de Tchebychev

1. (a) Après avoir créé l'anneau de polynômes à coefficients rationnels par `R.<X> = QQ[]`, on écrit :

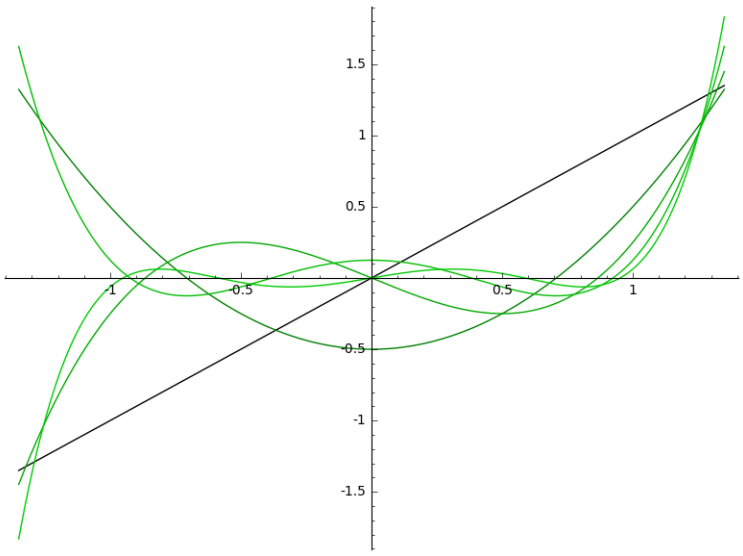
```
def tchebychev(n):
    T = []
    T = T + [1, X]
    for k in range(2,n+1):
        T = T + [2*X*T[k-1]-T[k-2]]
    return T
```

On part d'une liste vide, à laquelle on ajoute T_0 et T_1, puis pour chaque valeur de k : $2, 3, \ldots, n$, on ajoute T_k. La fonction retourne la liste $T_0, \ldots, T_n$.

(b) Par récurrence, on montre facilement que T_n est de degré n et de coefficient dominant 2^{n-1} si $n \geqslant 1$.

(c) La boucle suivante permet de construire sur un même graphique les courbes représentatives des fonctions $\frac{1}{2^{k-1}} T_k$ pour $k = 1, \ldots, n$:

```
var('x')
n = 5
T = tchebychev(n)
G = Graphics()
for k in range(1,n+1):
    Tk = 1/2^(k-1)*(T[k])(X=x)
    G = G + plot(Tk,x,-1.5,1.5,rgbcolor=(0,(1-1/k),0))
G.show(aspect_ratio=1)
```



On observe que les polynômes unitaires $\frac{1}{2^{k-1}}T_k$ réalisent une très bonne approximation uniforme de la fonction nulle. Ce sont en fait les meilleurs polynômes unitaires possibles (pour un chaque degré k fixé).

2. (a) Il suffit de demander à Sage de simplifier l'expression trigonométrique :

```
var('n,theta')
eq = cos(n*theta) - 2*cos(theta)*cos((n-1)*theta) + cos((n-2)*theta)
print 'Test egalite', eq.reduce_trig()
```

Sage retourne bien 0.

(b) On fixe θ. L'égalité est vraie pour $n = 0$ et $n = 1$. Comme les deux membres de l'égalité vérifient la même relation de récurrence, l'expression est vraie pour tout n.

(c) Les n nombres donnés sont deux à deux distincts et sont des racines de T_n d'après la question précédente. Comme T_n est de degré n, ce sont les seuls.

3. **Énigme.**

Première méthode.

La boucle suivante permet de construire la liste des coefficients de T_i dans X^n en commençant par le coefficient de T_n :

```
n = 30
T = tchebychev(n)
P = X^n
coeff = []
for i in range(n,-1,-1):
    q = P // T[i]
    coeff = [q] + coeff # coeff de T_i dans X^n
    P = P - q*T[i]
```

Noter que la boucle retourne la liste des coefficients en commençant par le coefficient de T_0 (même si elle est construite dans l'autre sens !) :

```
[9694845/67108864, 0, 145422675/536870912, 0, 59879925/268435456, 0...
```

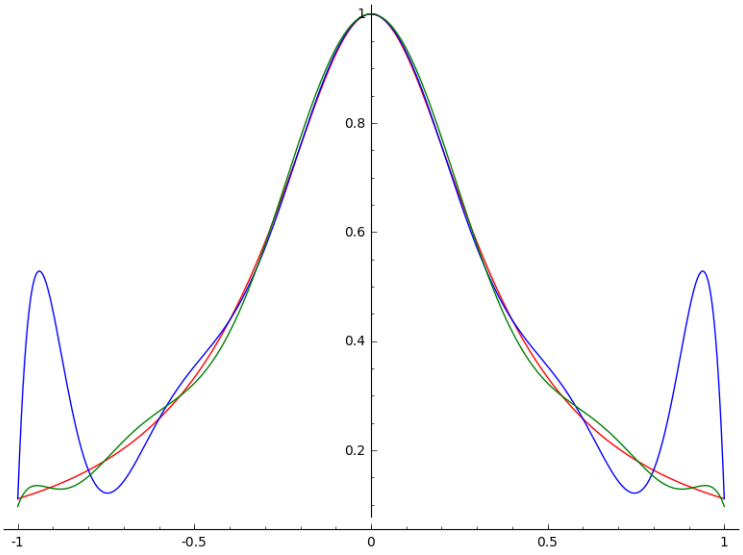
Seconde méthode.

Il suffit de définir la matrice A, puis de lire le coefficient de A^{-1} en position $(0, n)$:

```
n = 30
T = tchebychev(n)
A = matrix(n+1)
A[0,0] = 1 # T_0 = 1
for j in range(1,n+1):
    for i in range(j+1):
        A[i,j] = (T[j])[i]  # coeff de X^i dans T_j
print 'Enigme', (A.inverse())[0,n]
```

Réponse attendue : 9694845.

4. Voici les résultats pour $n = 10$ (11 points d'interpolation) : la courbe de la fonction f est en rouge ; l'interpolation de Lagrange basée sur les racines des polynômes de Tchebychev (courbe verte) donne de biens meilleurs résultats que lorsque les abscisses sont réparties uniformément (courbe bleue).



3. Algorithme d'Estrin

1. (a) Voici une fonction de paramètre n et α qui retourne la liste des puissances α^{2^i} pour $i = 1, \ldots, p$ avec $p = \lceil \log_2(n) \rceil - 1$.

```
def liste_puissances(n,alpha):
    p = ceil(log(n,2))-1
    m = 2^p
    liste = []
    puiss = alpha
    for k in range(p):
        puiss = puiss * puiss
        liste = liste + [puiss]
    return liste
```

   (b) Le nombre de multiplications préalables est p.

2. On se place dans l'anneau `R` des polynômes à coefficients rationnels ; `R.<X> = QQ[]`. On traduit le pseudo-code en la fonction suivante :

```
def eval_estrin(P,alpha):
```

```
    n = P.degree()
    if n <= 1:
        return P[0]+alpha*P[1]
    else:
        m = 2^(ceil(log(n,2))-1)
        Pbas = R( [P[k] for k in range(0,m)] )
        Phaut = R( [P[k] for k in range(m,n+1)] )
        eval = eval_estrin(Phaut,alpha)*alpha^m + eval_estrin(Pbas,alpha)
        return eval
```

La commande `R` permet de créer un polynôme dans l'anneau `R` à partir d'une liste de coefficients donnée en paramètre.

3. **Énigme.** On affine la fonction précédente en introduisant un compteur qui dénombre le nombre de multiplications autres que les multiplications préalables. La fonction renvoie le résultat de l'évaluation aussi que le nombre de multiplications effectuées jusque là.

```
def eval_estrin_compte(P,alpha):
    n = P.degree()
    if n <= 1:
        if P[1] == 0:
            return P[0],0                # 0 multiplication
        else:
            return P[0]+alpha*P[1],1     # 1 multiplication
    else:
        m = 2^(ceil(log(n,2))-1)
        Pdroite = R( [P[k] for k in range(0,m)] )
        Pgauche = R( [P[k] for k in range(m,n+1)] )
        Pg,cg = eval_estrin_compte(Pgauche,alpha)
        Pd,cd = eval_estrin_compte(Pdroite,alpha)
        c = cg+cd
        eval = Pg*alpha^m + Pd               # 1 multiplication
        if Pg != 0:
            c = c + 1
        return eval,c
```

Pour les données de l'énoncé :

```
n = 50
P = sum( (k+1)*X^(3*k) for k in range(0,n+1))
alpha = 2
```

on fait un bilan qui tient compte de toutes les multiplications : 101 multiplications dans `eval_estrin_compte` et 7 multiplications préalables (longueur de la liste produite par `liste_puissances(P.degree(),alpha)`.

Réponse attendue : 108.

4. On fait de même avec le schéma de Horner, d'abord simple :

```
def eval_horner(P,alpha):
    n = P.degree()
    val = P[n]
    for k in range(n-1,-1,-1):
        val = alpha*val + P[k]
    return val
```

puis avec compteur :

```
def eval_horner_compte(P,alpha):
    n = P.degree()
    val = P[n]
    c = 0
    for k in range(n-1,-1,-1):
        if val != 0:
            c = c + 1              # 1 multiplication
        val = alpha*val + P[k]
    return val,c
```

Le nombre de multiplications ici nécessaires pour l'algorithme de Horner est 150.

4. Courbes de Bézier

1. La définition des polynômes de Bernstein est immédiate :

```
def bernstein(n,k):
    return binomial(n,k)*X^k*(1-X)^(n-k)
```

Questions facultatives :

(a) C'est clair car si $0 < x < 1$, alors $0 < 1-x < 1$, donc on somme des termes strictement positifs.

(b) On écrit $1 = 1^n = (X+(1-X))^n = \sum_{k=0}^{n} \binom{n}{k} X^k (1-X)^{n-k} = \sum_{k=0}^{n} B_k^n(X)$, la troisième égalité venant de la formule du binôme de Newton.

(c) C'est immédiat soit à partir de la relation du triangle de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

soit directement par récurrence.

2. En partant de la liste des points, on commence par définir les fonctions abscisse et ordonnée de notre courbe paramétrée :

```
def bezier(mespoints):
    n = len(mespoints)-1
    B = [ bernstein(n,k)(X=x) for k in range(n+1) ]
    eqx = sum( (mespoints[k])[0] * B[k] for k in range(n+1) )
    eqy = sum( (mespoints[k])[1] * B[k] for k in range(n+1) )
    return eqx, eqy
```

On a utilisé la liste intermédiaire B des polynômes de Bézier calculée à partir de la fonction de la première question.

On peut alors définir une fonction qui trace les différents éléments (les points, les segments qui joignent deux points consécutifs, la courbe) :

```
def trace_bezier(mespoints):
    G = point(mespoints, color='red', size=10)
    G = G + line(mespoints, color='orange')
    courbe = bezier(mespoints)
    G = G + parametric_plot( courbe, (x,0,1) )
    return G
```

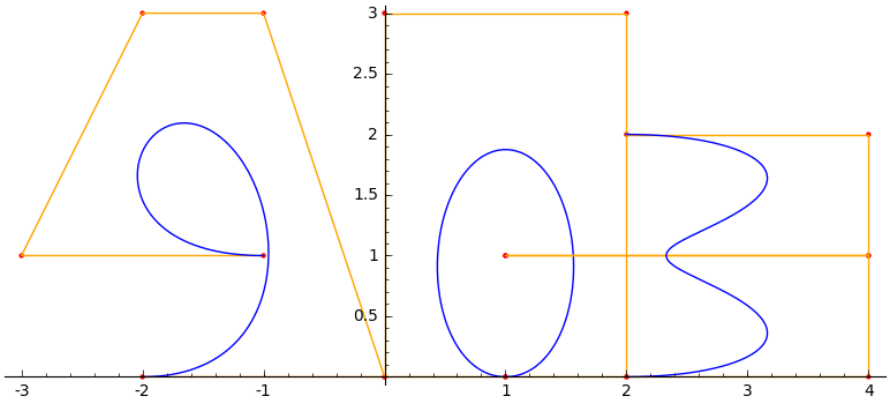
3. **Énigme.** On commence par définir nos trois listes de points :

```
mespoints1 = [ (-1,1), (-3,1), (-2,3), (-1,3), (0,0), (-2,0) ]
mespoints2 = [ (1,0), (0,0), (0,3), (2,3), (2,0), (1,0) ]
mespoints3 = [ (2,0), (4,0), (4,1), (1,1), (1,1), (4,1), (4,2), (2,2) ]
```

Puis on définit les courbes correspondantes grâce à la question précédente, et on les trace :

```
G1 = trace_bezier(mespoints1)
G2 = trace_bezier(mespoints2)
G3 = trace_bezier(mespoints3)
G = G1 + G2 + G3
G.show(aspect_ratio=1)
```

Voici ce que l'on obtient :



Réponse attendue : 903.

4. On commence par définir une fonction qui calcule le barycentre :

```
def barycentre(P,Q,t):
    return ( (1-t)*P[0]+t*Q[0], (1-t)*P[1]+t*Q[1] )
```

Ensuite, on calcule les coordonnées du point de la courbe correspondant au paramètre t par l'algorithme de Casteljau :

```
def casteljau(mespoints,t):
    n = len(mespoints)-1
    while n>0:
        souspoints = [barycentre(mespoints[i],mespoints[i+1],t) for i in range(n)]
        mespoints = souspoints
        n = n-1
    return souspoints[0]
```

On définit la liste des points de la courbe correspondant aux paramètres $0, 1/N, 2/N \ldots$ à laquelle on ajoute le dernier point de la liste de départ :

```
def bezier_bis(mespoints,N):
    ligne = []
    for t in srange(0,1,1/N):
        ligne = ligne + [casteljau(mespoints,t)]
    ligne = ligne + [mespoints[-1]]
    return ligne
```

Enfin, on définit la ligne brisée à partir de la liste précédente et il ne reste plus qu'à tracer le tout :

```
def trace_bezier_bis(mespoints,N):
    G = point(mespoints, color='red', size=10)
    G = G + line(mespoints, color='orange')
    courbe = bezier(mespoints)
    G = G + parametric_plot( courbe, (x,0,1), color = 'gray' )
    ligne = bezier_bis(mespoints,N)
    G = G + line(ligne)
    return G
```

On a ajouté successivement la liste des points de départ, la ligne brisée correspondante, la courbe de Bézier obtenue par l'équation paramétrique, la ligne brisée obtenue par l'algorithme de Casteljau. En réponse à

```
mespoints = [ (0,0), (0,1), (3,3), (4,1) ]  # Une cubique
N = 7
G = trace_bezier_bis(mespoints,N)
G.show(aspect_ratio=1)
```

Sage affiche :