Calculs d'intégrales

Calcul formel – TP 7

1. Polynômes de Legendre

1. On définit les polynômes de degré 0 et 1 par $P_0 = 1$, $P_1 = 2x - 1$:

   ```
   P0 = 1
   P1 = 2*x-1
   ```

   puis on cherche le polynôme suivant, P_2, sous forme indéterminée :

   ```
   var('a,b,c')
   P2 = a*x^2+b*x+c

   I0 = integral(P2*P0,(x,0,1))
   I1 = integral(P2*P1,(x,0,1))
   I2 = integral(P2*P2,(x,0,1))
   ```

   Ces trois intégrales sont des expressions linéaires ou quadratiques en a, b et c. On est ensuite amené à résoudre le système :

   ```
   sol = solve( [I0==0,I1==0,I2==1/5], a,b,c)
   ```

   La solution de coefficient dominant positif est $P_2 = 6x^2 - 6x + 1$.

2. Pour $P3$, on procède exactement comme à la question précédente :

   ```
   var('a3,a2,a1,a0')
   P3 = a3*x^3+a2*x^2+a1*x+a0

   I0 = integral(P3*P0,(x,0,1))
   I1 = integral(P3*P1,(x,0,1))
   I2 = integral(P3*P2,(x,0,1))
   I3 = integral(P3*P3,(x,0,1))

   sol = solve( [I0==0,I1==0,I2==0,I3==1/7], a3,a2,a1,a0)
   ```

   La solution de coefficient dominant positif est $P_3 = 20x^3 - 30x^2 + 12x - 1$.

   Pour P_4, on pourrait bien sûr faire de même, mais on va adopter une méthode un peu plus générale. On commence par définir un polynôme de degré 4 quelconque par :

```
n = 4
Legendre = [P0,P1,P2,P3]

liste_coeff =  [ var('a%d' % i) for i in (0..n) ]

print liste_coeff
Pn = sum(liste_coeff[i]*x^i for i in (0..n))
```

puis on définit le système d'équations linéaires ou quadratiques et on le résout :

```
equations = [ integral(Pn*Legendre[i],(x,0,1)) == 0 for i in (0..n-1) ]
equations = equations + [ integral(Pn*Pn,(x,0,1)) == 1/(2*n+1) ]
sol = solve( equations, *liste_coeff, solution_dict=True)[0]
```

Enfin, on affiche la solution comme un polynôme :

```
Pn = sum(sol[liste_coeff[i]]*x^i for i in (0..n))
```

On trouve alors $P_4 = 70x^4 - 140x^3 + 90x^2 - 20x + 1$.

3. On peut calculer la dérivée n-ème par `diff( (x^2-x)^n, x, n )` pour les premières valeurs de n, on conjecture que $c_n = 1/n!$.
4. **Énigme.** En utilisant ce qui a été mis en oeuvre pour la détermination de P_4, voici une fonction qui calcule un polynôme de Legendre connaissant la liste des polynômes de Legendre précédemment trouvés :

```
def legendre_suivant(Legendre):
    n = len(Legendre)
    liste_coeff =  [ var('a%d' % i) for i in (0..n) ]
    Pn = sum(liste_coeff[i]*x^i for i in (0..n))
    equations = [ integral(Pn*Legendre[i],(x,0,1)) == 0 for i in (0..n-1) ]
    equations = equations + [ integral(Pn*Pn,(x,0,1)) == 1/(2*n+1) ]
    sol = solve( equations, *liste_coeff, solution_dict=True)[0]
    Pn = sum(sol[liste_coeff[i]]*x^i for i in (0..n))
    return Pn
```

On peut alors calculer la liste complète

```
def tout_legendre(n):
    Legendre = [1,2*x-1]
    for i in (2..n):
        P = legendre_suivant(Legendre)
        Legendre += [P]
    return Legendre
```

On trouve que $P_7 = 3432x^7 - 12012x^6 + 16632x^5 - 11550x^4 + 4200x^3 - 756x^2 + 56x - 1$.

Réponse attendue : 16632.

2. Calcul approché d'intégrales

2.1. Méthode des rectangles

1. Pour la méthode des rectangles à gauche, on peut utiliser une boucle pour sommer :

```
def somme_rectangles_gauche(f, a, b, n):
    somme = 0
    epsilon = (b-a)/n
    xk = a
    for k in range(n):
        somme = somme + f(x=xk)
        xk = xk + epsilon
    return (b-a)/n*somme
```

De même pour la méthode des rectangles à droite :

```
def somme_rectangles_droite(f, a, b, n):
    somme = 0
    epsilon = (b-a)/n
    xk = a + epsilon
    for k in range(n):
        somme = somme + f(x=xk)
        xk = xk + epsilon
    return (b-a)/n*somme
```

2. (a) Dans le cas d'une fonction décroissante, l'intégrale est minorée par la somme des aires des rectangles à droite et majorée par la somme des aires des rectangles à gauche. On peut noter que la différence entre ces deux sommes est :

$$(f(a)-f(b))\frac{b-a}{n}$$

(en effet les rectangles qui interviennent dans ces sommes sont les mêmes sauf aux extrémités) ce qui donne une majoration de l'erreur commise.

(b) **Énigme.** On utilise nos fonctions :

```
f = 1/(1+x^2)
a = 0
b = 2
n = 9

Sg = somme_rectangles_gauche(f, a, b, n)
Sd = somme_rectangles_droite(f, a, b, n)
```

ce qui donne :

$$Sg = \frac{2825131943877506}{2363375715915825} \quad \text{et} \quad Sd = \frac{2404976261048026}{2363375715915825}.$$

Réponse attendue : 2363375715915825.

(c) On teste des valeurs de n de plus en plus grandes jusqu'à ce que les valeurs des sommes des rectangles à droite et à gauche aient les mêmes deux décimales après la virgule.

Par exemple $n = 300$:

$$Sg = 1.10981479186880 \quad \text{et} \quad Sd = 1.10448145853546.$$

Comme la valeur de l'intégrale est entre ces deux valeurs, on est assuré des deux premières décimales : $\arctan(2) = 1,10\ldots$.

3. On procède comme à la question précédente, en changeant les entrées numériques :

```
f = sqrt(4*cos(x)^2+sin(x)^2)
a = 0
b = pi/2
n = 10

Sg = somme_rectangles_gauche(f, a, b, n)
Sd = somme_rectangles_droite(f, a, b, n)
```

On obtient pour valeurs approchées :

$$Sg = 2.50065187147255 \quad \text{et} \quad Sd = 2.34357223879306.$$

2.2. Méthode de Simpson

1. On utilise à nouveau une boucle pour sommer :

```
def simpson(f, a, b, n):
    somme = 0
    epsilon = (b-a)/n
    xk = a
    for k in range(n):
        somme = somme +  f(x=xk)+f(x=xk+epsilon)+4*f(x=xk+epsilon/2)
        xk = xk + epsilon
    return (b-a)/(6*n)*somme
```

2. Il suffit d'utiliser notre fonction :

```
f = 1/(1+x^2)
a = 0
b = 2
n = 5
S = simpson(f, a, b, n)
```

On obtient pour $n = 5$:

$$S = \frac{232293966116776}{209813209414575}$$

ayant pour valeur approchée 1.10714652697477 et pour $n = 10$:

$$S = \frac{66540223565933213120272511116547}{60100535886246172707367723921 50}$$

ayant pour valeur approchée 1.10714859002049.

3. (a) On procède comme pour l'exemple précédent :

```
f = sqrt(4*cos(x)^2+sin(x)^2)
a = 0
b = pi/2
n = 10

S = simpson(f, a, b, n)
print 'n = ', n
print 'Simpson : ', S.n()
```

On obtient en retour :

```
n =  10
Simpson                      :  2.42211205513829
```

(b) **Énigme.** On commence par définir une fonction contenant une boucle et qui compte le nombre de décimales communes :

```
def decimales_communes(x,y):
    if x == y:  return -1
    commun = 0
    while ceil(x)-ceil(y) == 0:
        commun = commun + 1
        x = x * 10
        y = y * 10
    return commun-1
```

On peut alors répondre à l'énigme grâce à :

```
f = sqrt(4*cos(x)^2+sin(x)^2)
a = 0
b = pi/2
precision = 150
Sinfini = simpson(f, a, b, 200).n(digits=precision)

print 'Decimales␣Ellipse'
for n in srange(95,115):
    Sn = simpson(f, a, b, n)
    print 'n␣=␣', n, '->␣Simpson␣:␣', Sn.n(digits=150)
    print 'decimales␣stables', decimales_communes(Sn.n(digits=precision),
                                                  Sinfini)
```

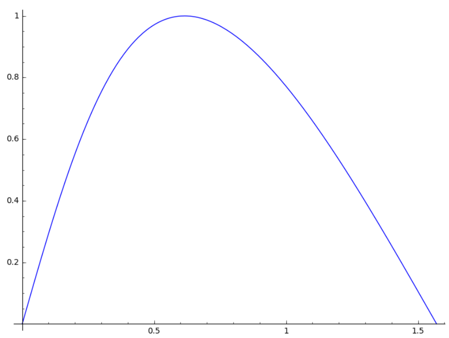
Brève explication : on a constaté expérimentalement que le nombre de sous-intervalles nécessaires pour avoir 110 décimales stables est inférieur à 200, on se sert de l'approximation de l'intégrale correspondante comme valeur de référence.

Ensuite, on compare le nombre de décimales communes de l'approximation correspondant à un découpage en n sous-intervalles avec cette valeur de référence pour des valeurs de n bien choisies (ici $95 \leqslant n < 115$) jusqu'à obtenir le nombre de décimales stables voulu.

Réponse attendue 112.

2.3. Majoration de l'erreur

1. On demande à Sage de résoudre l'inéquation en n : "majorant de l'erreur inférieur à précision voulue".



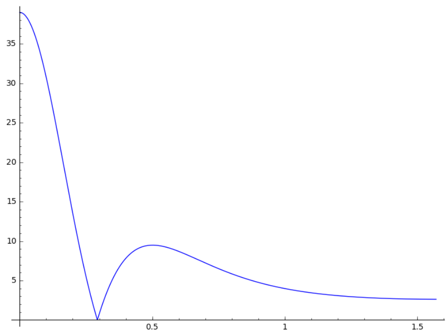
Graphiquement on voit qu'il n'y a qu'un seul maximum local pour $|f'(x)|$ et que c'est le maximum global.

```
def erreur_rectangle(f,a,b,prec):
  print('Erreur␣rectangles,␣ellipse')
  ff = abs(diff(f,x))
  ffmax = ff.find_local_maximum(a,b)[0]
  print 'maximum', ffmax
  var('n')
  eq = (b-a)^2/(2*n) * ffmax <= prec
  sol = solve(eq,n)[1]
  print 'precision␣',prec,'␣pour␣n␣>=', sol[0].rhs().n()
  return
```

La commande `erreur_rectangle(sqrt(4*sin(x)^2+cos(x)^2),0,pi/2,10**(-15))` retourne alors

```
Erreur rectangles, ellipse
maximum 1.0
precision  1/1000000000000000 pour n >= 1.23370055013617e15
```

2. **Énigme.** On fait exactement comme à la question précédente :



Graphiquement on voit que le maximum pour $|f^{(4)}(x)|$ est atteint en $x = 0$ (attention il y a un autre maximum local, mais qui n'est pas global).

```
def erreur_simpson(f,a,b,prec):
  print('Erreur␣simpson,␣ellipse')
  ff = abs(diff(f,x,4))
  ffmax = ff.find_local_maximum(0,0.1)[0]
  print 'maximum', ffmax
  eq = (b-a)^5/(2880*n^4) * ffmax <= prec
  sol = solve(eq,n)[1]
  print 'precision␣',prec,'␣pour␣n␣>=', sol[0].rhs().n()
  return
```

La commande `erreur_simpson(sqrt(4*sin(x)^2+cos(x)^2),0,pi/2,10**(-15))` renvoie :

```
Erreur simpson, ellipse
maximum 39.0
precision 1/1000000000000000 pour n >= 3373.40027582705
```

Réponse attendue 3374.