```
# -*- coding: utf-8 -*-
"""
Created on Wed May 17 14:09:48 2017

@author: stephane
"""

from numpy import *                  # Exos 1, 2, 3
import matplotlib.pyplot as pypl     # Exos 1, 3
from scipy.integrate import quad     # Exos 1, 2, 3, 4
from fractions import Fraction       # Exo 4
import numpy.polynomial as poly      # Exo 2

"""
Premier exercice : UNE INTÉGRALE IMPROPRE
"""

"""
Pour avoir T^2−T−1>=0 il suffit d'avoir T>=(1+sqrt(5))/2.
Je dis ça en pensant à T=exp(t/2)
On majore alors grossièrement (pour a>=x) F(a) par int(exp(−t/2), t=a..infini)
"""

x0 = 2*log((1+sqrt(5)/2))
alpha = -2*log(10**(-5)/2)
"""
>>> x0
1.50097658923773
>>> f(x0)
0.28615724035011497
>>> exp(−x0/2)
0.47213595499957939
>>> alpha
24.412145291060348
"""


def f(x):
    if x == 0:
        return 1
    else:
        return sin(x)/(exp(x)-1)

les_x = linspace(0, 4*pi, 200)
les_y = [f(x) for x in les_x]
pypl.plot(les_x, les_y)
pypl.grid()
pypl.savefig('dessin−int−impropre−br.pdf')
pypl.clf()


def rectangles(f, a, b, n):
    return sum(f(a+k*(b-a)/n) for k in range(n))*(b-a)/n


def trapezes(f, a, b, n):
    correctif = (f(b)-f(a))*(b-a)/n
    return rectangles(f, a, b, n) + correctif

for n in [10**2, 10**3, 10**4, 10**5]:
    print n, rectangles(f, 0., 25, n), trapezes(f, 0., 25, n)
"""
```



```
100 1.20428093085 0.954280930846
1000 1.0892000894 1.0642000894
10000 1.07792430788 1.07542430788
100000 1.07679905007 1.07654905007

>>> [quad(f, 0, A)[0] for A in [10, 25, 40]]
[1.0767054444173747, 1.0766740474626173, 1.0766740474685814]
"""


def serie(n):
    return sum(1/(k**2+1.) for k in range(1, n+1))
"""
>>> [serie(10**k) for k in range(2, 7)]
[1.0667242091524087, 1.0756745476347518, 1.0765740524687444, 1.0766640475185798, 1.0766730474691077]

>>> quad(f, 0, inf)
int−impropre−br.py:30: RuntimeWarning: overflow encountered in exp return sin(x)/(exp(x)−1)
(1.0766740474685814, 1.4744553779865095e−08)

Valeur donnée par Wolfram alpha :
>>> (−1+pi/tanh(pi))/2
1.076674047468581

Conclusion ?

L'erreur numérique des rectangles/trapèzes est prépondérante devant l'erreur mathématique.
"""

"""
Deuxième exercice : DES POLYNÔMES ORTHOGONAUX (TCHEBYCHEV)
"""

X = poly.Polynomial([0, 1])


def P(n):  # Jusqu'à n=10-15, une version récursive resterait acceptable
    pk, pkp1 = poly.Polynomial([1]), 2*X
    for k in range(n):
        pk, pkp1 = pkp1, 2*X*pkp1-pk
    return pk

for n in range(9):
    print(P(n))
"""
poly([ 1.])
poly([ 0.  2.])
poly([−1.  0.  4.])
poly([ 0. −4.  0.  8.])
poly([ 1.  0. −12.  0.  16.])
poly([ 0.  6.  0. −32.  0.  32.])
poly([ −1.  0.  24.  0. −80.  0.  64.])
poly([  0.  −8.  0.  80.  0. −192.  0.  128.])
poly([  1.  0. −40.  0.  240.  0. −448.  0.  256.])
"""


def scal(P, Q):  # P et Q sont des polynômes, donc peuvent être évalués en t
    def f(t):
        return P(t)*Q(t)*sqrt(1-t**2)
    return quad(f, -1, 1)[0]*2/pi
```

```
ps = array([[scal(P(i), P(j)) for j in range(4)] for i in range(4)])
"""
>>> ps
[[ 1.00000000e+00   0.00000000e+00  −1.01289102e−16   0.00000000e+00]
 [ 0.00000000e+00   1.00000000e+00   0.00000000e+00  −6.17513974e−16]
 [ −1.01289102e−16   0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [ 0.00000000e+00  −6.17513974e−16   0.00000000e+00   1.00000000e+00]]
"""


def phi(p):
    return -3*X*p.deriv()+(1-X**2)*p.deriv().deriv()

for i in range(5):
    print phi(P(i)), phi(P(i)) // P(i), phi(P(i)) % P(i)
"""
poly([ 0.]) poly([ 0.]) poly([ 0.])
poly([ 0. −6.]) poly([−3.]) poly([ 0.])
poly([  8.   0. −32.]) poly([−8.]) poly([ 0.])
poly([   0.   60.    0. −120.]) poly([−15.]) poly([ 0.])
poly([ −24.    0.  288.    0. −384.]) poly([−24.]) poly([ 0.])

B est donc une base diagonalisante. Comme elle est orthonormée, phi est autoadjoint; une IPP l'aurait aussi prouvé.

Et si on regardait la matrice de phi dans la base canonique pour s'entrainer ? On va transposer les coefficients des phi(X^j)...

Manipulation horrible... pas vraiment exigible !
"""

Mat = transpose(array([(list(phi(X**j).coef)+([0]*4))[:5] for j in range(5)]))
"""
>>> vap, vep = np.linalg.eig(Mat)
>>> vap
array([  0.,  −3.,  −8., −15., −24.])
>>> vep
array([[ 1.        ,  0.        , −0.24253563,  0.        ,  0.04993762],
       [ 0.        ,  1.        ,  0.        , −0.4472136 ,  0.        ],
       [ 0.        ,  0.        ,  0.9701425 ,  0.        , −0.5992514 ],
       [ 0.        ,  0.        ,  0.        ,  0.89442719,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.79900187]])

"""

"""
Troisième exercice : UN ARC PARAMÉTRÉ
"""


def x(t):
    return t/(1+t**3)


def y(t):
    return t**2/(1+t**3)

les_t = linspace(-10, 10, 1000)
les_x = [x(t) for t in les_t]
les_y = [y(t) for t in les_t]

pypl.plot(les_x, les_y)
pypl.grid()
```



```
pypl.xlim(-1.5, 1.5)        # Pour limiter le champ des abscisses
pypl.ylim(-1.5, 1.5)
pypl.savefig('dessin−arc−parametre−pezza.pdf')


def v(t):  # Bon, il a fallu faire un petit calcul...
    return sqrt((1−2*t**3)**2+(2*t−t**4)**2) / (1.+t**3)**2

"""
>>> quad(v, 0, inf)[0]
1.639162907227178
"""
# Recommençons avec un calcul numérique pour la dérivée


def der(f, t, dt):
    return (f(t+dt)−f(t−dt))/(2*dt)


def vapprox(t, dt):
    return sqrt(der(x, t, dt)**2 + der(y, t, dt)**2)


def approx(dt):
    def vv(t):
        return vapprox(t, dt)
    return quad(vv, 0, inf)[0]

"""
>>> [approx(10**(−k)) for k in range(1, 5)]
[1.6307059373845887, 1.6390776000237486, 1.6391620540806437, 1.63916289868996]
"""
pypl.clf()

"""
Tout d'abord, si (X,Y)=(x(t),y(t)) avec t<>0, alors t=Y/X puis X=t/(1+t^3) ... puis X^3+Y^3=XY
Ceci reste vrai pour $(X,Y)=(x(0),y(0)).

Réciproquement, si X^3+Y^3=XY :
 − si X=0, alors Y=0 donc (X,Y)=(x(0),y(0))
 − sinon, on pose t=Y/X, et on a alors X^3(1+t^3)=XY=tX^2, or Y<>0 donc ..., donc X=... etc.
"""


def f(x, y):
    return x**3+y**3−x*y

les_x = linspace(−1, 1, 100)
les_y = linspace(−1, 1, 100)
gx, gy = meshgrid(les_x, les_y)
les_z = f(gx, gy)
pypl.contour(gx, gy, les_z, 0)
pypl.axis('equal')
pypl.grid()
pypl.savefig('dessin−aveccontour.pdf')


"""
Quatrième exercice : INTÉGRALES ET RATIONNELS
"""
```

```
def u_numer(m, n):
    def f(x):
        return x**m*(1-x)**n
    return quad(f, 0, 1)[0]


def u(m, n):
    if m == 0:
        return Fraction(1, n+1)
    else:
        return Fraction(m, n+1)*u(m-1, n+1)


for (m, n) in [[2, 5], [12, 4]]:
    theo = u(m, n)
    print (m,n),u_numer(m, n), theo, float(theo)
"""
(2, 5) 0.00595238095238 1/168 0.00595238095238
(12, 4) 3.23206205559e-05 1/30940 3.23206205559e-05
"""


def g(x):
    return (x**6-4*x**5+5*x**4+4)/(4+x**4*(1-x)**4)
"""
Grâce au 1/4^k les convergences sont rapides (très précisément)...

>>> quad(g, 0, 1)[0]
1.1186233467369096
>>> sum((-0.25)**k*(u(4*k+6, 4*k)-4*u(4*k+5, 4*k)+5*u(4*k+4, 4*k)
...                     + 4*u(4*k, 4*k)) for k in range(10))/4
1.1186233467369093
>>> sum((-0.25)**k*(u(4*k+6, 4*k)-4*u(4*k+5, 4*k)+5*u(4*k+4, 4*k)
                        + 4*u(4*k, 4*k)) for k in range(20))/4
1.1186233467369093
>>> sum((-0.25)**k*(u(4*k+6, 4*k)-4*u(4*k+5, 4*k)+5*u(4*k+4, 4*k)
                        + 4*u(4*k, 4*k)) for k in range(5))/4
1.1186233467369096
>>> sum((-0.25)**k*(u(4*k+6, 4*k)-4*u(4*k+5, 4*k)+5*u(4*k+4, 4*k)
                        + 4*u(4*k, 4*k)) for k in range(2))/4
1.1186230436230435

C'est encore plus rapide que ce que je pensais !

Exercice : expliquer !
"""
```