In [264]:

```
import numpy as np
import matplotlib.pyplot as plt
```

Exercice 1 - Q1

In [265]:

```
A = np.array([
    [1, 1, 1],
    [1,-2, 1],
    [2,-1, 1]
], np.float)
```

In [266]:

```
b = np.array([1, 0, 2], np.float)
```

Résolution du premier système : (Méthode 1)

In [267]:

```
X = np.linalg.solve(A, b)
```

In [268]:

```
X
```

Out[268]:

```
array([ 1.66666667,  0.33333333, -1.        ])
```

Résolution du premier système : (Méthode 2)

In [269]:

```
X = np.dot(np.linalg.inv(A), b)
```

In [270]:

```
X
```

Out[270]:

```
array([ 1.66666667,  0.33333333, -1.        ])
```

Exercice 1 - Q2

In [271]:

```
C = np.array([
    [3,1,-1],
    [1,1,1],
    [1,-2,2]
], np.float)
```

In [272]:

```
d = np.array([3, 3, 1], np.float)
```

Résolution du deuxième système : (Méthode 1)

Il faur calculer les valeurs de a,b, et c, solution du deuxième système

In [273]:

```
X1 =  np.linalg.solve(C, d)
```

In [274]:

```
X1
```

Out[274]:

```
array([1., 1., 1.])
```

In [275]:

```
X = np.linalg.solve(A, X1)
```

In [276]:

```
X
```

Out[276]:

```
array([ 0., -0.,  1.])
```

Résolution du deuxième système : (Méthode 2)

In [277]:

```
X1 = np.dot(np.linalg.inv(C), d)
```

In [278]:

```
X1
```

Out[278]:

```
array([1., 1., 1.])
```

In [279]:

```
X = np.dot(np.linalg.inv(A),X1)
```

In [280]:

```
X
```

Out[280]:

```
array([1.11022302e-16, 5.55111512e-17, 1.00000000e+00])
```

Exercice 1 - Q3

On constate que solve a calculé une solution exacte, par rapport à inv qui a générer une erreur.

np.linalg.solve(A, b) ne calcule pas l'inverse de A. Au lieu de cela, il appelle une des routines qui factorise d'abord A en utilisant la décomposition LU, puis résout le système en utilisant la substitution avant et arrière.

np.linalg.inv utilise la même méthode pour calculer l'inverse de A. L'étape de factorisation est exactement la même que ci-dessus, mais il faut plus d'opérations en virgule flottante pour résoudre A^{-1}. De plus, si vous vouliez alors obtenir x via l'identité A^{-1}·b = x, alors la multiplication de matrice supplémentaire entraînerait encore plus d'opérations en virgule flottante, et donc des performances plus lentes et plus d'erreur numérique.

Exercice 2

Faut résoudre le système suivant :

In [281]:

```
A = np.array([
    [0, 0, 1],
    [1, 1, 1],
    [4, 2, 1]
], np.float)
```

In [282]:

```
b = np.array([1, 4, 3])
```

In [283]:

```
X = np.linalg.solve(A, b)
```

In [284]:

```
X
```

Out[284]:

```
array([-2.,  5.,  1.])
```

Donc : P(x) = -2x^{2} + 5x + 1

Exercice 3 - Q1

In [285]:

```
t = np.linspace(0, 10, 100)
```

In [286]:

```
X = t*np.cos(t)
```

In [287]:

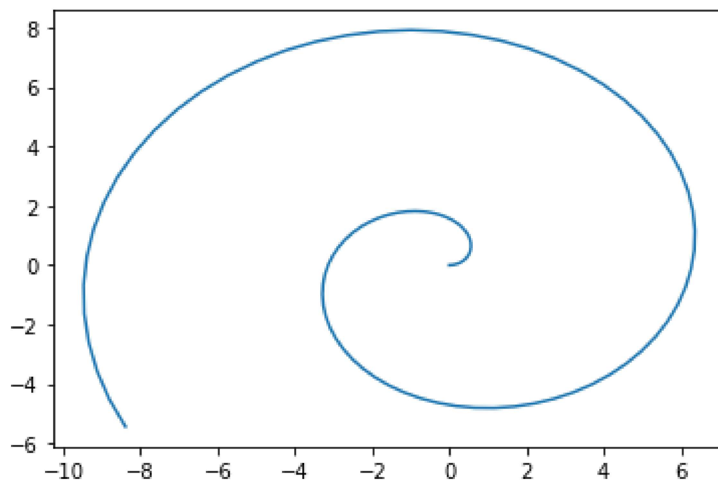
```
Y = t*np.sin(t)
```

In [288]:

```
plt.plot(X, Y)
```

Out[288]:

```
[<matplotlib.lines.Line2D at 0x2f2e81c6048>]
```



Exercice 3 - Q2

In [289]:

```
t = np.linspace(0, 10, 100)
```

In [290]:

```
u=10
v=10
a=5
b=5
```

In [291]:

```
X = u + a*np.cos(t)
```

In [292]:

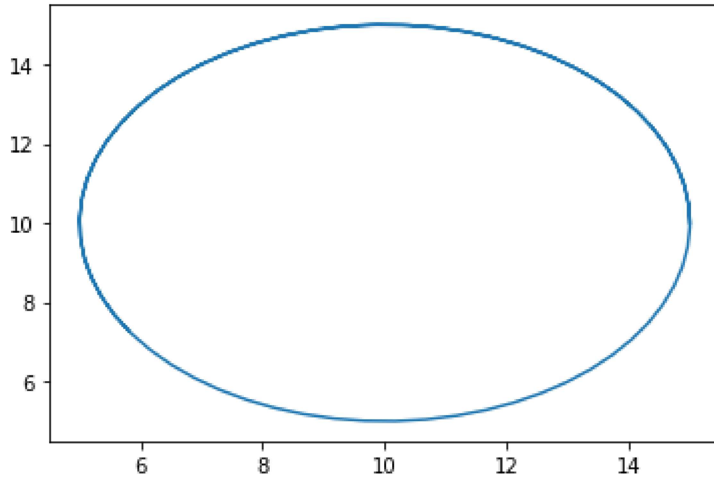
```
Y = v + b*np.sin(t)
```

In [293]:

```
plt.plot(X, Y)
```

Out[293]:

```
[<matplotlib.lines.Line2D at 0x2f2e8282608>]
```



Exercice 4

Q1 :

In [294]:

```
# Première version
def est_triangulaire_sup(M) :
    n,p = M. shape
    if n!= p:
        return False
    for i in range (n):
        for j in range (i):
            if M[i,j ]!= 0:
                return False
    return True
```

In [295]:

```
# Deuxième version
def est_triangulaire_sup(M) :
    return (M == np.triu(M)). all()
```

Q2 :

In [296]:

```
def est_diagonale(M):
    n,p = M.shape
    if n != p:
        return False
    return (M == np.diag (np.diag(M))).all()
```

Q3 :

In [297]:

```
def est_inversible(M):
    return np.linalg.det(M)!=0
```

Exercice 5

Q1 :

In [298]:

```
T = np.array([
    [1, 2, 2],
    [0, 1, 1],
    [0, 0, 1]
], np.float)
```

In [299]:

```
b = np.array([1,2,1], np.float)
```

In [300]:

```
def remontee(T, b):
    X = np.zeros(b.size)
    for i in range(len(T)-1,-1, -1):
        X[i] = (b[i] - np.sum(T[i,i+1:]*X[i+1:]))/T[i,i]
    return X
```

Q2 :

In [301]:

```
def echange_ligne(A,i,j):
    """Applique l'opération élémentaire Li <-> Lj à la matrice A"""
    A[i], A[j] = np.copy(A[j]), np.copy(A[i])
```

Q3 :

In [302]:

```
def pivot_partiel(A, ind):
    n = len(A) # nbre de lignes de A
    imax = ind # indice de ligne avec pivot max
    for i in range(ind+1, n):
        if abs(A[i][ind]) > abs(A[imax][ind]):
            imax = i
    return imax
```

Q4 :

In [303]:

```
def transvection(A,i,j,mu):
    A[i] = A[i] + mu*A[j]
```

In [304]:

```
def solution_systeme(A0,b0):
    """ Renvoie la solution X du système de Cramer A0 X= b0"""
    A,b = np.copy(A0), np.copy(b0) # des copies des matrice initiales
    n = len(A)
    # Mise sous forme triangulaire
    for j in range(n-1): # on itère n-1 fois
        i0 = pivot_partiel(A,j)
        echange_ligne(A,j,i0)
        echange_ligne(b,j,i0)
        print(A)
        for i in range(j+1,n):
            x = A[i][j]/A[j][j]
            transvection(A,i,j,-x)
            transvection(b,i,j,-x) # attention b matrice colonne
        print(A)
    # phase de remontée
    return remontee(A,b)
```

Exercice 6

Q1

Le nombre d'adittions et de multiplications est : n.q.(2p-1)

Q2

In [305]:

```
def produit_strassen(A, B):
    n = len(A)
    if n == 1:
        return np.matrix([[A[0][0] * B[0][0]]])

    A11 = A[0:n//2, 0:n//2]
    A12 = A[0:n//2, n//2:n]
    A21 = A[n//2:n, 0:n//2]
    A22 = A[n//2:n,n//2:n]
    B11 = B[0:n//2, 0:n//2]
    B12 = B[0:n//2, n//2:n]
    B21 = B[n//2:n, 0:n//2]
    B22 = B[n//2:n,n//2:n]

    M1 = produit_strassen(A11+A22,B11+B22)
    M2 = produit_strassen(A21+A22,B11)
    M3 = produit_strassen(A11,B12-B22)
    M4 = produit_strassen(A22,B21-B11)
    M5 = produit_strassen(A11+A12,B22)
    M6 = produit_strassen(A21-A11,B11+B12)
    M7 = produit_strassen(A12-A22, B21+B22)

    R = np.zeros((n,n))
    R[0:n//2, 0:n//2] = M1 + M4 - M5 + M7
    R[0:n//2, n//2:n] = M3 + M5
    R[n//2:n, 0:n//2] = M2 + M4
    R[n//2:n,n//2:n] = M1 - M2 + M3 + M6

    return R
```