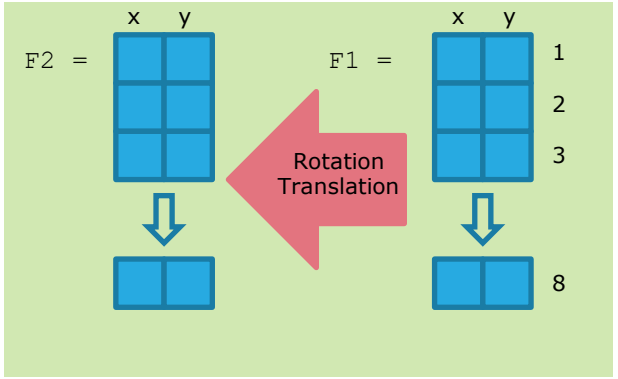
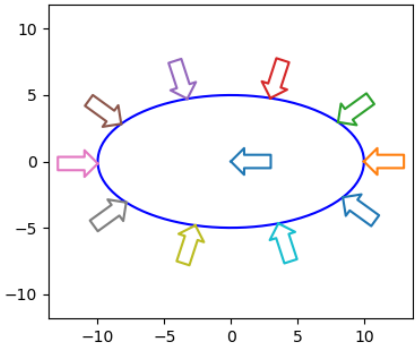
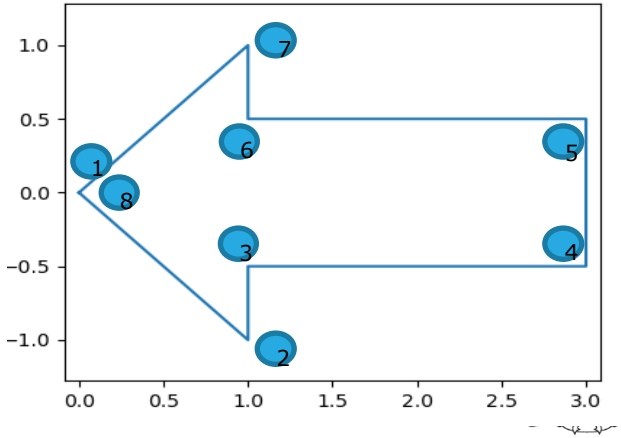
COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

Translation et rotation C2_E1b

```python
import numpy as np

import matplotlib.pyplot as plt

from Fonction_transformation import rotation2D, translation2D


#Transpose F1 pour alléger la notation de np.array

F1=np.array([[0,1,1,3,3,1,1,0],[0,-1,-0.5,-0.5,0.5,0.5,1,0]]).T


ti,tf,npts=0,2*np.pi,100      # Rotation de t = 0 à 2 pi

t=np.linspace(ti,tf,npts)

delt=2*np.pi/10               # Angle de rotation relative constant


#Equation paramétrique d'une ellipse

x,y=10*np.cos(t),5*np.sin(t)

plt.plot(F1[:,0],F1[:,1])     # Flèche initiale

plt.plot(x,y,'-b')            # Ellipse bleue

plt.axis('equal')             # Échelle égale en x et y


#Boucle de translation et rotation pour la prochaine étape

for i in range(0,npts,10):

    F2=F1+[x[i],y[i]]         # F2=translation2D(F1,[x[i],y[i]])

    plt.plot(F2[:,0],F2[:,1])

    F1=rotation2D(F1, delt)   # Rotation relative additionnelle
```







COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

Translation et rotation C2_E1b (suite)

```python
def translation2D(obj, dep):
    import numpy as np
    obj=obj+dep
    return obj
```

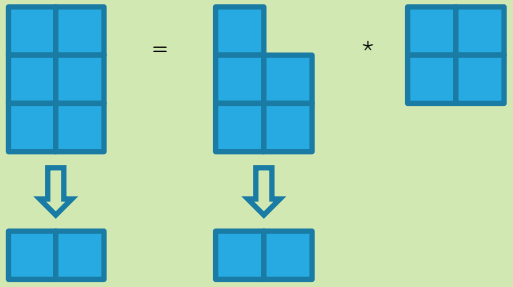
Provenant du fichier de Fonction_transformation

```
obj =   obj  +  vecteur;
```

```python
def rotation2D(obj, theta):
    import numpy as np
    C, S = np.cos(theta), np.sin(theta)
    R = np.array([[C, S], [-S, C]])
    obj = obj.dot(R)
    return obj
```

$$R(\theta) = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}$$

```
Obj     =     obj     *     R;
```

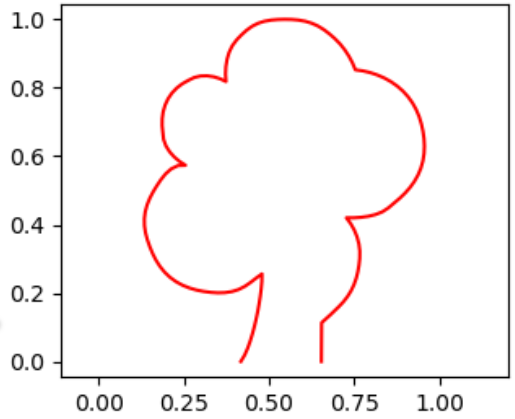


COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

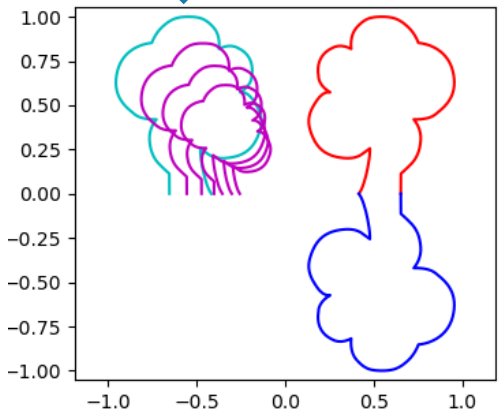
Lecture de fichier, réflexion et homothétie C2_E2

```python
import numpy as np
import matplotlib.pyplot as plt
from Fonction_transformation import reflection2D


nom_fichier='arbre.txt'
p=np.loadtxt(nom_fichier)

plt.figure()
plt.plot(p[:,0],p[:,1],'-r')

#Symétrie de p original par rapport à laxe des x en bleu
T1=reflection2D('x')
p1=np.dot(p,T1)
plt.plot(p1[:,0],p1[:,1],'-b')
plt.axis('equal')
#Symétrie de p original par rapport à l'axe des y et homothétie en magenta
T2=reflection2D('y')
p2=np.dot(p,T2)
plt.plot(p2[:,0],p2[:,1],'-c')
for i in range(3):
    p2=p2*0.85
    plt.plot(p2[:,0],p2[:,1],'-m')
```

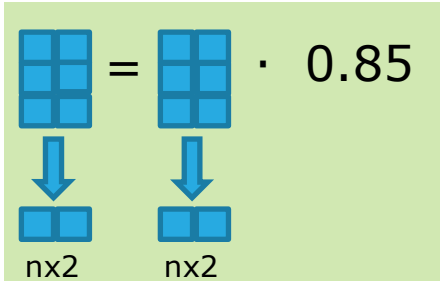
```
0.417120762 1.29141E-06
0.428494746 0.018994414
0.437371656 0.039302194
0.444823815 0.060181166
0.451298859 0.081381571
0.457024077 0.102795123
0.462123767 0.124369004
```

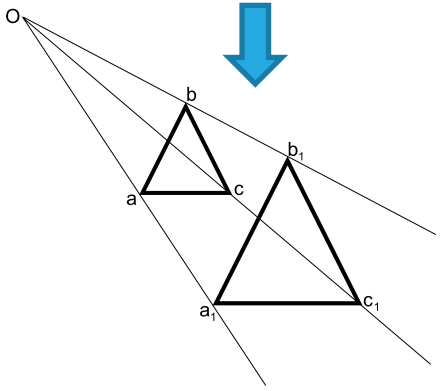


Homothétie par rapport à l'origine et non à la figure, décallage quand changement d'échelle



$$P2 = P2 \cdot 0.85$$





COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

Lecture de fichier, réflexion et homothétie C2_E2 (suite)

```python
def reflection2D(axe):
    import numpy as np
    axe=axe.lower()
    matrice={'x': np.array([[1, 0],[0, -1]]),
             'y': np.array([[-1, 0],[0, 1]]),
             'o': np.array([[-1, 0],[0, -1]])}
    return matrice[axe]
```

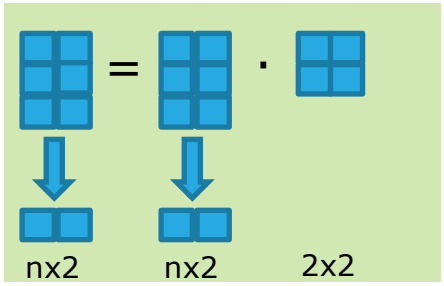
$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

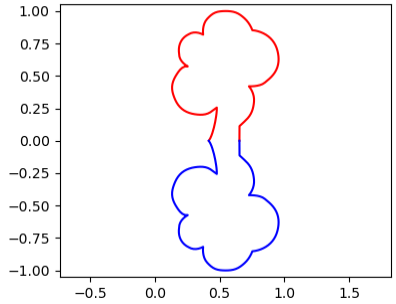
$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

Utilisation d'un dictionnaire pour éviter une notation lourd avec if, else, if. Équivalent du switch case dans d'autre langage de programmation

$$P_1 = P \cdot T_1$$




```python
import numpy as np
import matplotlib.pyplot as plt
from Fonction_transformation import reflection2D


nom_fichier='arbre.txt'
p=np.loadtxt(nom_fichier)

plt.figure()
plt.plot(p[:,0],p[:,1],'-r')

#Symétrie de p original par rapport à laxe des x en bleu
T1=reflection2D('x')
p1=np.dot(p,T1)
plt.axis('equal')
plt.plot(p1[:,0],p1[:,1],'-b')
```

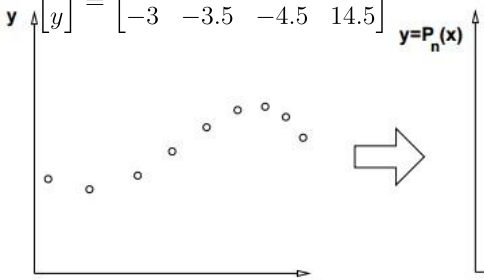


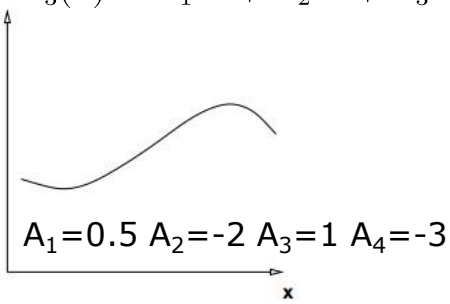
COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

Méthode de Vandermonde pour interpolation C2_E3

Extraction d'un polynôme à partir de points: n conditions, n-1 degrés

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 & 3 & 5 \\ -3 & -3.5 & -4.5 & 14.5 \end{bmatrix}$$

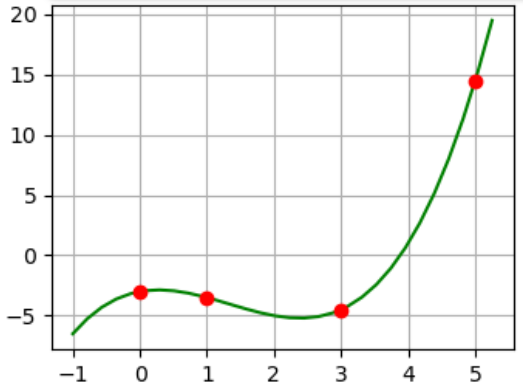




$$P_3(x) = A_1x^3 + A_2x^2 + A_3x^1 + A_4x^0$$

A_1=0.5 A_2=-2 A_3=1 A_4=-3

4 équations et 4 inconnus



$$y_1 = A_1x_1^3 + A_2x_1^2 + A_3x_1^1 + A_4x_1^0$$
$$y_2 = A_2x_1^3 + A_2x_2^2 + A_3x_2^1 + A_4x_2^0$$
$$y_3 = A_2x_3^3 + A_2x_3^2 + A_3x_3^1 + A_4x_3^0$$
$$y_4 = A_2x_4^3 + A_2x_4^2 + A_3x_4^1 + A_4x_4^0$$

$$Ax = b$$

$$\begin{bmatrix} x_1^3 & x_1^2 & x_1^1 & 1 \\ x_2^3 & x_2^2 & x_2^1 & 1 \\ x_3^3 & x_3^2 & x_3^1 & 1 \\ x_4^4 & x_4^2 & x_4^1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

$$x = A^{-1}b$$

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} x_1^3 & x_1^2 & x_1^1 & 1 \\ x_2^3 & x_2^2 & x_2^1 & 1 \\ x_3^3 & x_3^2 & x_3^1 & 1 \\ x_4^4 & x_4^2 & x_4^1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 27 & 9 & 3 & 1 \\ 125 & 25 & 5 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

Solution

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  #Interpolation polynomiale avec système d'équations sous forme de matrice
5  x_p=np.array([0,1,3,5])
6  y_p=np.array([-3,-3.5,-4.5,14.5])
7  #Résoltution MA=y_p
8  M=np.vstack((x_p**3,x_p**2,x_p**1,x_p**0)).T     #Empile 4 lignes de 4 valeurs
9  a=np.linalg.inv(M).dot(y_p)                      #Résoltuion a=inv(M)*yp
10 #A=np.linalg.solve(M,y_p)
11
12 'Calcul x_analytique et y_analytique'
13 x=np.linspace(-1,5.25,30)
14 '-------Méthode 1---------------------'
15 # y=np.empty(30)
16 # for i in range(30):
17 #     y[i]=a[0]*x[i]**3+a[1]*x[i]**2+a[2]*x[i]**1+a[3]*x[i]**0
18 '-------Méthode 2---------------------'
19 y=np.polyval(a,x)                                #Calcul y=A_9
20 '-------Méthode 3---------------------'
21 # y=a[0]*x**3+a[1]*x**2+a[2]*x**1+a[3]*x**0
22
23
24 plt.plot(x,y,'-g')
25 plt.plot(x_p,y_p,'or')
26 plt.grid(axis='both')
```

On transpose pour avoir le bon format de matrice Vandermonde

Méthodes équivalentes

COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

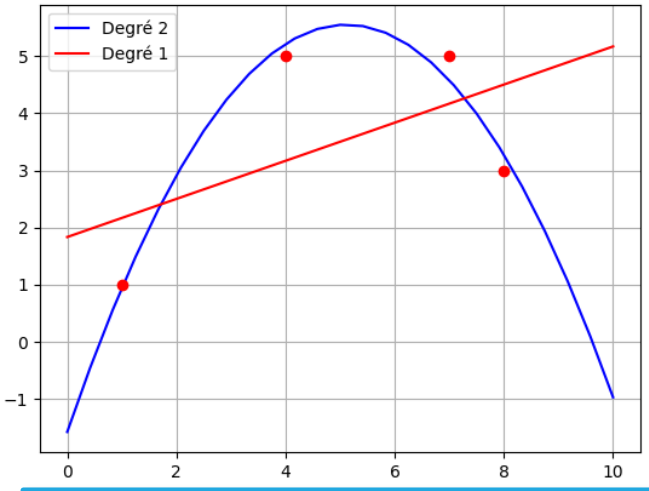
Moindre carré par pseudo-inverse C2_E4

$$Ma \approx y \qquad m \text{ equations} > n \text{ coefficients}$$

$$M \equiv \begin{bmatrix} x_1^2 & x_1^1 & x_1^0 \\ x_2^2 & x_2^1 & x_2^0 \\ x_3^2 & x_3^1 & x_3^0 \\ x_4^2 & x_4^1 & x_4^0 \end{bmatrix}, a \equiv \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, y \equiv \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

$$a = M^p y, M^p = (M^T M)^{-1} M^T \qquad M \equiv \begin{bmatrix} 1^2 & 1^1 & 1^0 \\ 3^2 & 3^1 & 3^0 \\ 7^2 & 7^1 & 7^0 \\ 8^2 & 8^1 & 8^0 \end{bmatrix}, y \equiv \begin{bmatrix} 1 \\ 5 \\ 5 \\ 3 \end{bmatrix}$$

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

#Approximation de moindre carré de degré 2 avec 4 équations
#Système sur-contraint
xP=np.array([1,4,7,8])
yP=np.array([1,5,5,3])
x=np.linspace(0,10,25)
#==================  Degré 2   ================================
M=np.vstack((xP**2,xP**1,xP**0)).T  #Empile 3 lignes de 4 valeurs, puis transpose
a=np.linalg.pinv(M).dot(yP)         #Résolution a=M^p * yp
#a=np.polyfit(xP,yP,2)              #Résolution avec polyfit de numpy
y1=np.polyval(a,x)                  #Calcul y=a_0 x^2 + a_1 x^1 + a_2
#y=a[0]*x**2+a[1]*x**1+a[2]*x**0    #Calcul y=a_0 x^2 + a_1 x^1 + a_2 (aussi)
print('Degre 2: a.T=',a.T)
#=================   Degré 1     ================================
M=np.vstack((xP**1,xP**0)).T        #Empile 2 lignes de 4 valeurs, puis transpose
a=np.linalg.pinv(M).dot(yP)         #Résolution a=M^p * yp
y2=np.polyval(a,x)                  #Calcul y= a_0 x^1 + a_1
print('Degre 1: a.T=',a.T)

#================= Traçage de graphique ======================
plt.plot(x,y1,'-b',label='Degré 2')
plt.plot(x,y2,'-r',label='Degré 1')
plt.plot(xP,yP,'or')
plt.grid(axis='both')
plt.legend()
```



4 équations et 3 inconnus (degré 2)
4 équations et 2 inconnus (degré 1)

Points :

i	1	2	3	4
xD_i	1	4	7	8
yD_i	1	5	5	3

Résultats :

A_1=-0.2727 A_2=2.7879 A_3=-1.5758

COURS C2 : TRANSFORMATION GÉOMÉTRIQUE

Résolution de système 3 équations 3 inconnus C2_E6

$$
\begin{aligned}
x_1 + x_2 + 2x_3 &= 3 \\
x_1 + 2x_2 + x_3 &= 1 \\
2x_1 + x_2 + x_3 &= 0
\end{aligned}
$$

$$\mathbf{Ax} = \mathbf{b}$$

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix}$$

```python
import numpy as np
from numpy.linalg import det, inv, solve

A=np.array([[1, 1, 2],[1, 2, 1],[2, 1, 1]]) #Matrice 3x3
# ========= Vecteur 1D ne peut pas être transposé!!  =========
b=np.array([3, 1, 0]).reshape(3,1)  #Vecteur colonne
b2=np.array([3, 1, 0])
# ========= Vecteur 2D peut être transposé, crochet supplémentaire
# b=np.array([[3,1,0]]).T            #Vecteur colonne à partir de transposé

print('A=\n',A)
print('b^T=',b.T)
print('det(A)=%.1f'%det(A))
Ai=inv(A)
print('inv(A)=\n',Ai)
x=np.dot(Ai,b)                      #Calcul x = inv(A) * b
print('x^T=',x.T)                   #Un peu moin precis
y=solve(A,b)                        #Resout le systeme Ay=b
print('y^T=',y.T)                   #Un peu plus precis
```

Vecteur 1D en agit comme une matrice ligne 1xn, est affiché (n,) dans python

Nam	Type	Size	Value
A	Array of int32	(3, 3)	[[1 1 2] [1 2 1]
Ai	Array of float64	(3, 3)	[[-0.25 -0.25 0.75] [-0.25 0.75 -0.25] [0.75 -0.25 -0.25]]
b	Array of int32	(3, 1)	[[3] [1] [0]]
b2	Array of int32	(3,)	[3 1 0]
x	Array of float64	(3, 1)	[[-1.00000000e+00] [1.66533454e-16] [2.00000000e+00]]
y	Array of float64	(3, 1)	[[-1.00000000e+00] [7.40148683e-17] [2.00000000e+00]]

Vecteur 2D en python agit comme une matrice, information sur ligne et colonne