Toutes calculatrices autorisées.
Le sujet comporte un total de 3 exercices.

CORRIGE
(Exercices 1 et 2)

EXERCICE 1. Décollage d'un engin spatial.

On s'intéresse dans cet exercice au décollage d'un engin spatial depuis le sol terrestre.

On modélise la situation comme suit :

- La trajectoire de l'engin est supposée verticale et il est repéré (variable *z*) le long d'un axe vertical passant par le centre de la terre, pris comme origine ($z = 0$), et le centre du pas de tir ($z = R_T > 0$ où R_T désigne le rayon de la Terre assimilée à une sphère).
- L'origine des temps est l'instant du décollage. On a donc : $z(0) = R_T$ et $\left(\frac{dz}{dt}\right)(0) = 0$.
- La masse de l'engin est notée *m* et est une fonction du temps.
- La vitesse d'éjection des gaz par rapport à l'engin est supposée constante et notée *u* ($u > 0$).
- On néglige les frottements.

Avec les hypothèses précédentes en considérant le système fermé {fusée, gaz} entre les instants *t* et $t + dt$, on effectue un bilan de la variation de la quantité de mouvement globale et le principe fondamental de la dynamique nous donne :

$$m\frac{d^2z}{dt^2} + \frac{dm}{dt}u = -G\frac{mM_T}{z^2} \qquad (\mathrm{E})$$

où *G* désigne la constante universelle de la gravitation et M_T la masse de la Terre.

1. On s'intéresse en fait à la variable *h* correspondant à l'altitude de l'engin (i.e. sa distance au sol). Vérifier que la variable *h* satisfait l'équation différentielle (E') suivante et préciser les conditions initiales (à $t = 0$) :

$$\frac{d^2h}{dt^2} = -\frac{GM_T}{(R_T + h)^2} - \frac{1}{m}\frac{dm}{dt}u \qquad (\mathrm{E}')$$

On a $z = R_T + h$ et donc, R_T étant une constante : $\frac{dz}{dt} = \frac{dh}{dt}$ et $\frac{d^2z}{dt^2} = \frac{d^2h}{dt^2}$.

Ainsi, l'équation différentielle (E) se récrit : $m\frac{d^2h}{dt^2} + \frac{dm}{dt}u = -G\frac{mM_T}{(R_T+h)^2}$, soit, en divisant par m : $\frac{d^2h}{dt^2} = -\frac{GM_T}{(R_T+h)^2} - \frac{1}{m}\frac{dm}{dt}u$. C'est l'équation (E').

On a : $z(0) = R_T$ et $\left(\frac{dz}{dt}\right)(0) = 0$. On en déduit immédiatement : $h(0) = z(0) - R_T = 0$ et $\left(\frac{dh}{dt}\right)(0) = \left(\frac{dz}{dt}\right)(0) = 0$.

On cherche à résoudre l'équation différentielle (E') avec les conditions initiales précisées à la question précédente à l'aide d'un schéma d'Euler explicite sur un intervalle de temps $[0\,;t_f]$. On utilise un pas de temps constant Δt. Pour la masse *m* et la dérivée $\frac{dm}{dt}$, on dispose d'une fonction Python `FM` qui reçoit comme argument un flottant correspondant au temps *t* et renvoie un tuple de deux éléments correspondant respectivement à $m(t)$ et $\left(\frac{dm}{dt}\right)(t)$. Les variables `G`, `MT`, `RT` et `u` correspondant respectivement à la constante *G*, à la masse M_T, au rayon R_T et à la vitesse d'éjection *u* seront déclarées globales.

2. Ecrire une fonction Python `D2H` qui reçoit comme arguments une variable `t` correspondant à un instant *t*, une variable `h` correspondant à l'altitude de l'engin à l'instant *t* et une variable `v` correspondant à sa vitesse (dérivée de *h* par rapport au temps) à ce même instant. La fonction `D2H` renverra la dérivée seconde de *h* par rapport au temps à l'instant *t*.

```
def D2H(t,h,v):
    …
```

Il s'agit ici de traduire l'équation (E') : $\frac{d^2h}{dt^2} = -\frac{GM_T}{(R_T+h)^2} - \frac{1}{m}\frac{dm}{dt}u$.

Puisque la fonction FM renvoie un tuple de deux éléments correspondant respectivement à $m(t)$ et $\left(\frac{dm}{dt}\right)(t)$, ces grandeurs seront respectivement données par `FM(t)[0]` et `FM(t)[1]`. On aura donc :

```
def D2H(t,h,v) :
    global G,MT,RT,u
    return(-G*RT/(RT+h)**2-FM(t)[1]*u/FM(t)[0])
```

3. Ecrire une fonction `EULER_decollage` qui permet de résoudre numériquement l'équation différentielle (E') sur un intervalle de temps $[t_i ; t_f]$ ($t_i < t_f$). Elle reçoit comme arguments :
   - Une variable `ti` correspondant au temps initial t_i.
   - Une variable `tf` correspondant au temps final t_f.
   - Une variable `pas` correspondant au pas de temps Δt.
   - Une variable `h0` correspondant à l'altitude initiale.
   - Une variable `v0` correspondant à la vitesse initiale.

   La fonction construira trois listes `Lt`, `Lh` et `Lv` contenant les valeurs des instants, de l'altitude et de la vitesse de l'engin aux instants 0, *pas*, $2\times$ *pas* , $3\times$ *pas* , …
   La fonction affichera un graphique donnant l'altitude de l'engin en fonction du temps (le script contient l'instruction `import matplotlib.pyplot as plt`).

```
def EULER_décollage(ti,tf,pas,h0,v0):
    …
```

   Par exemple, si l'on prend la seconde comme unité de temps et si l'on souhaite obtenir l'altitude atteinte par l'engin au bout de 2 minutes avec un pas de temps égal à une seconde, on appellera la fonction `EULER_decollage` dans le script comme suit :

```
EULER_décollage(0,120,1,0,0)
```

On a par exemple, en tenant compte du fait que les grandeurs évaluées à l'instant t_{i+1} le sont à partir de grandeurs disponibles à l'instant t_i :

```
def EULER_decollage(ti,tf,pas,h0,v0):
    Lt, Lh, Lv = [ti], [h0], [v0]
    while Lt[-1] < tf:
        Lt.append(Lt[-1] + pas)
        Lh.append(Lh[-1] + pas * Lv[-1])
        # Attention ! Les listes Lt et Lh viennent d'être
        # modifiées…
        Lv.append(Lv[-1] + pas * D2H(Lt[-2],Lh[-2],Lv[-1]))
    plt.clf()
    plt.plot(Lt,Lh)
    plt.show()
```

4. Le décollage se divise en fait en quatre phases principales correspondant aux intervalles de temps $I_1 = [0\,;t_1]$, $I_2 = [t_1\,;t_2]$, $I_3 = [t_2\,;t_3]$ et $I_4 = [t_3\,;t_4]$. On dispose ainsi de quatre fonctions Python `FM1`, `FM2`, `FM3` et `FM4` donnant à un instant *t* quelconque, dans l'un de ces intervalles, la masse $m(t)$ et la valeur prise par sa dérivée $\left(\frac{dm}{dt}\right)(t)$. On suppose par ailleurs que les temps t_1, t_2, t_3 et t_4 sont fournis dans une liste `LT`.

   Comment modifier le script et la fonction `EULER_decollage` pour tenir compte de ces quatre phases et résoudre l'équation différentielle (E') sur l'intervalle de temps $[0\,;t_4]$?

Nous allons en fait exécuter quatre fois la fonction `EULER_decollage`.

Dans le script, on fait apparaître les quatre fonctions `FM1`, `FM2`, `FM3` et `FM4` (voir ci-dessous). La masse de la navette variant en permanence, nous utilisons une variable `M_init_curr` qui correspond à la masse de la navette au début de chaque phase du vol (i.e. aux instants 0, t_1, t_2 et t_3. `M_init_curr` est donc initialisée avec la masse initiale de la navette) qui est déclarée globale dans chacune de ces quatre fonctions. Cette variable est également déclarée globale dans la fonction `EULER_decollage` car elle doit être mise à jour après chaque exécution de la boucle principale de cette fonction.

On modifie également la fonction `EULER_decollage` afin que :

- Elle puisse récupérer de la fonction `D2H` la masse courante de la navette (à la fin de l'exécution de la boucle, cette masse courante sert à mettre à jour la variable `M_init_curr`.
- Elle puisse renvoyer les derniers éléments des listes `Lt`, `Lh` et `Lv` afin que ces valeurs servent de valeurs initiales dans le prochain appel de la fonction.

Les noms des fonctions `FM1`, `FM2`, `FM3` et `FM4` sont stockés dans la liste `FM` et on pourra de la sorte appeler telle ou telle de ces fonctions en faisant simplement référence à l'élément correspondant de la liste `FM`. La fonction `D2H` est modifiée en conséquence.

On a par exemple, en supposant la fonction *m* affine par morceaux :

```
# ================================ #
# Décollage d'une navette spatiale #
# Janvier 2016 #
# ================================ #

import matplotlib.pyplot as plt

def FM1(t):
    global rho1, M_init_curr
    return(-rho1*t+M_init_curr,-rho1)
```

```
def FM2(t):
    global rho2, M_init_curr
    return(-rho2*t+M_init_curr,-rho2)

def FM3(t):
    global rho3, M_init_curr
    return(-rho3*t+M_init_curr,-rho3)

def FM4(t):
    global rho4, M_init_curr
    return(-rho4*t+M_init_curr,-rho4)

def D2H(t,h,v):
    global G, MT, RT, u, it
    m = FM[it](t)[0]
    return(-G*MT/(RT+h)**2-FM[it](t)[1]*u/m,m)

def EULER_decollage(t_init,t_final,pas,h_init,v_init):
    global M_init_curr
    Lt, Lh, Lv = [t_init], [h_init], [v_init]
    while Lt[-1] < t_final:
        Lt.append(Lt[-1] + pas)
        Lh.append(Lh[-1] + pas * Lv[-1])
        (a,Mnew) = D2H(Lt[-1],Lh[-1],Lv[-1])
        Lv.append(Lv[-1] + pas * a)
    M_init_curr = Mnew
    plt.plot(Lt,Lh)
    plt.show()
    return(Lt[-1],Lh[-1],Lv[-1])

# Constante universelle de la gravitation
G = 6.67384*10**-11

# Données Terre
MT = 5.972*10**24
RT = 6.371*10**6

# Données navette
# Masse initiale
M_init_curr = 2.046*10**6
# Débits des moteurs (pour les 4 phases de vol)
rho1 = 9300
rho2 = 5000
rho3 = 9000
rho4 = 4000
# Vitesse (relative) d'éjection des gaz
u = 3237

# Conditions initiales
h0 = RT
v0 = 0

# Les bornes des intervalles de temps...
t0 = 0
tmax = 120
```

```
LT = [20,80,100,tmax]

t_init_curr = t0
t_final_curr = LT[0]
h_init_curr = h0
v_init_curr = v0

# Les fonctions FMi
FM = [FM1,FM2,FM3,FM4]

plt.clf()
for it in range(4):
    (t_init_curr,h_init_curr,v_init_curr) =
    EULER_decollage(t_init_curr,LT[it],1,h_init_curr,v_init_curr)
```

EXERCICE 2. Les courbes de BEZIER

Une courbe de Bézier à $N+1$ ($N \geq 1$) points de contrôle $P_0, P_1, P_2, ..., P_N$ est une courbe paramétrée d'extrémités les points P_0 et P_N définie par :

$$\left\{ P(t) = \sum_{i=0}^{N} B_i^N(t).P_i, t \in [0;1] \right\}$$

où les B_i^N sont les polynômes de Bernstein définis par :

$$\forall t \in \mathbb{R}, B_i^N(t) = \binom{N}{i} \times t^i \times (1-t)^{N-i}$$

Remarque importante : l'écriture B_i^N ne désigne en rien une puissance !

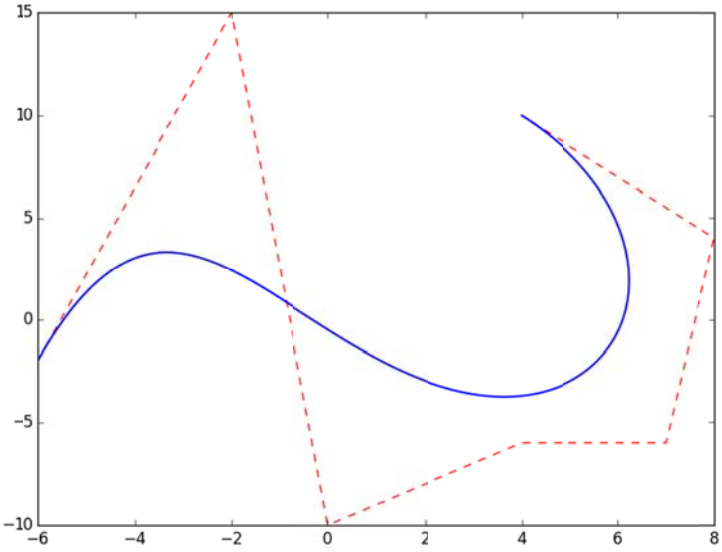
Par exemple, pour $N = 6$ et en considérant les points de contrôle suivants :

$$P_0\begin{pmatrix}-6\\-2\end{pmatrix}, P_1\begin{pmatrix}-2\\15\end{pmatrix}, P_2\begin{pmatrix}0\\-10\end{pmatrix}, P_3\begin{pmatrix}4\\-6\end{pmatrix}, P_4\begin{pmatrix}7\\-6\end{pmatrix}, P_5\begin{pmatrix}8\\4\end{pmatrix} \text{ et } P_6\begin{pmatrix}4\\10\end{pmatrix}$$

on obtient la courbe de Bézier correspondant aux points $P(t)\begin{pmatrix}x(t)\\y(t)\end{pmatrix}$ ($t \in [0;1]$) avec :

$$\begin{cases} x(t) = \sum_{i=0}^{6} B_i^N(t) \times x_i = -6B_0^N(t) - 2B_1^N(t) + 4B_3^N(t) + 7B_4^N(t) + 8B_5^N(t) + 4B_6^N(t) \\ y(t) = \sum_{i=0}^{6} B_i^N(t) \times y_i = -2B_0^N(t) + 15B_1^N(t) - 10B_2^N(t) - 6B_3^N(t) - 6B_4^N(t) + 4B_5^N(t) + 10B_6^N(t) \end{cases}$$

Graphiquement (la ligne en pointillés rouges est la ligne polygonale des points de contrôle) :



Partie A : évaluation directe des polynômes de Bernstein

Dans cette partie, on se donne un réel *t* dans l'intervalle $[0;1]$ et un entier naturel *i* dans $[\![0;N]\!]$. On cherche à évaluer le nombre $C(N,i)$ de multiplications et de divisions requises par le calcul de $B_i^N(t)=\binom{N}{i}\times t^i\times(1-t)^{N-i}$.

1. On a : $B_i^N(t)=\dfrac{N!}{i!\times(N-i)!}\times t^i\times(1-t)^{N-i}$.

   Dans cette question, on suppose que l'on calcule $B_i^N(t)$ en calculant chacune des trois factorielles apparaissant dans le coefficient binomial.

   Calculer $C(N,i)$.

Traitons le cas général (où *i* est différent de 0 et *N*) :

- Le calcul de $N!$ requiert $N-1$ multiplications.
- Le calcul de $i!\times(N-i)!$ requiert $(i-1)+1+(N-i-1)=N-1$ multiplications.
- Le calcul de $t^i\times(1-t)^{N-i}$ requiert $(i-1)+1+(N-i-1)=N-1$ multiplications.

Il convient donc d'effectuer un total de : $3\times(N-1)+1=3N-2$ multiplications.
A ce total, il faut ajouter une division.
Finalement : $C(N,i)=3N-2+1=3N-1$.

Lorsque $i=0$ (ou N) :

- Le calcul de $N!$ requiert $N-1$ multiplications.
- Le calcul de $i!\times(N-i)!=0!\times N!$ requiert $1+(N-1)=N$ multiplications.
- Le calcul de $t^i\times(1-t)^{N-i}=1\times(1-t)^N$ requiert $1+(N-1)=N$ multiplications.

Il convient donc cette fois d'effectuer un total de : $3N-1$ multiplications.
A ce total, il faut ajouter une division.
Finalement : $C(N,0)=C(N,N)=3N-1+1=3N$.

$C(N,0)=C(N,N)=3N$ et pour tout i dans $[\![1;N-1]\!]$, $C(N,i)=3N-1$

2. On a, pour $i\neq 0$:

$$B_i^N(t)=\frac{N\times(N-1)\times(N-2)\times\ldots\times(N-i+1)}{i\times(i-1)\times(i-2)\times\ldots\times2\times1}\times t^i\times(1-t)^{N-i}=\frac{\prod_{k=0}^{i-1}(N-k)}{\prod_{k=0}^{i-1}(k+1)}\times t^i\times(1-t)^{N-i}$$

On a donc calculé $\binom{N}{i}$ en ayant simplifié le rapport $\frac{N!}{(N-i)!}$.

a. Calculer $C(N,i)$.

Les produits $\prod_{k=0}^{i-1}(N-k)$ et $\prod_{k=0}^{i-1}(k+1)$ comportent chacun i facteurs et leur évaluation requiert donc un total de $2\times(i-1)$ multiplications.

Comme on l'a vu à la question précédente, le calcul de $t^i\times(1-t)^{N-i}$ requiert $N-1$ multiplications.
Finalement, ce calcul de $B_i^N(t)$ requiert au total $2(i-1)+(N-1)+1=N+2(i-1)$ multiplications et une division : $C(N,i)=N+2(i-1)+1=N+2i-1$.

Pour tout i dans $[\![1;N]\!]$, $C(N,i)=N+2i-1$.

On a aussi $\binom{N}{i}=\binom{N}{N-i}$.

b. Calculer $C(N,N-i)$.

En utilisant le résultat de la question précédente (à condition que $N-i$ soit différent de 0, c'est-à-dire que *i* soit différent de *N*), il vient :

$$C(N,N-i)=N+2(N-i)-1=3N-2i-1$$

Pour tout *i* dans $[\![0\,;N-1]\!]$, $C(N,N-i)=3N-2i-1$.

c. Ecrire une fonction Python `Bernstein_base` recevant en argument deux entiers *N* et *i* et un réel *t* dans $[0;1]$ et renvoyant $B_i^N(t)$ en effectuant un minimum de multiplications.

Dans tous les cas, on doit calculer la valeur de : $t^i\times(1-t)^{N-i}$.

Ensuite, pour *i* dans $[\![1;N-1]\!]$, on peut choisir, d'après la question précédente, de calculer $\binom{N}{i}$ ou $\binom{N}{N-i}$.

On a : $C(N,N-i)-C(N,i)=3N-2i-1-(N+2i-1)=2N-4i=2(N-2i)$.

D'où : $C(N,N-i)-C(N,i)<0\Leftrightarrow 2(N-2i)<0\Leftrightarrow i>\frac{N}{2}$. On aura donc intérêt à mener le calcul avec $N-i$ au lieu de *i* dès lors que $i>\frac{N}{2}$.

D'où la fonction :

```
def Bernstein_base(n,i,t):
    if i == 0 :
        res = (1 - t)**(n - i)
    elif i == n :
        res = t**i
    else:
        res = t**i + (1 - t)**(n - i)
        if i > n/2 :
            i = n - i
        num, denum = 1, 1
        for k in range(i):
            num *= (n - k)
            denum *= (k + 1)
        # On renvoie un float…
        res *= num/denum
    return(res)
```

3. On a, pour *i* non nul : $\binom{N}{i} = \frac{N}{i} \times \binom{N-1}{i-1}$.

   On propose le code récursif suivant pour le calcul de $\binom{N}{i}$:

```
def binomial(n,i):
    if i == 0 or i == n:
        return 1
    else:
        return((n / i) * binomial(n - 1,i - 1))
```

   Ce code peut conduire à des résultats erronés. Pouvez dire pourquoi ?
   En ne modifiant que le deuxième `return`, proposer un code correct.

Le calcul « `n / i` » génère un flottant qui va être multiplié par le résultat de l'appel récursif à « `binomial(n - 1,i - 1)` » qui fournira lui-même un flottant…
La multiplication de ces flottants peut engendrer des amplifications des erreurs d'arrondis. De telles erreurs peuvent rapidement apparaître du fait des valeurs élevées des coefficients binomiaux, même pour des valeurs relativement « faibles » de *n* (faites des essais !).

Cette mise en œuvre est évidemment à proscrire !

Partie B : évaluation récursive des polynômes de Bernstein

Les polynômes de Bernstein peuvent être calculés selon la définition récursive suivante :

$B_0^1(t) = 1-t$, $B_1^1(t) = t$ et pour $N > 1$:

$$B_i^N(t) = \begin{cases} (1-t) \times B_i^{N-1}(t) & \text{si } i = 0 \\ (1-t) \times B_i^{N-1}(t) + t \times B_{i-1}^{N-1}(t) & \text{si } i \in [\![1 ; N-1]\!] \\ t \times B_{i-1}^{N-1}(t) & \text{si } i = N \end{cases}$$

4. Ecrire une fonction récursive `Bernstein_rec` qui calculera $B_i^N(t)$ et recevra en argument deux entiers *N* et *i* (dans $[\![0 ; N]\!]$) et un réel *t* (dans $[0 ; 1]$). Les appartenances mentionnées ne seront pas testées par la fonction.

On doit prendre garde de ne pas oublier certains cas !

Voici un exemple de code possible (la dernière ligne est coupée en deux par manque de place…) :

```
def Bernstein_rec(n,i,t):
    if n == 1 and i == 1:
        return(t)
    elif n == 1 and i == 0:
        return(1 - t)
    elif i == 0:
        return((1 - t) * Bernstein_rec(n - 1, i, t))
    elif i == n:
        return(t * Bernstein_rec(n - 1, i - 1, t))
    else:
        return((1 - t) * Bernstein_rec(n - 1, i, t) + t *
               Bernstein_rec(n - 1, i - 1, t))
```

Partie C : l'algorithme de Casteljau

L'algorithme de Casteljau est un algorithme visant à déterminer, pour *t* fixé dans $[0;1]$, les coordonnées $\begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ d'un point $P(t)$ en tirant parti de la définition récursive précédente. Le principe en est le suivant :

- On note $P_0^0, P_1^0, P_2^0, ..., P_N^0$ la liste initiale des points de contrôle. Cette liste comporte $N+1$ points.
- A partir de la liste précédente, on construit la liste des *N* barycentres $P_i^1 = \left\{\left(P_i^0, t\right), \left(P_{i+1}^0, 1-t\right)\right\}$ avec $i \in [\![0\,;N-1]\!]$. On a ainsi : $P_i^1 = t.P_i^0 + (1-t).P_{i+1}^0$, c'est-à-dire, en notant $P_i^1\begin{pmatrix} x_i^1(t) \\ y_i^1(t) \end{pmatrix}$: $\begin{cases} x_i^1(t) = t \times x_i^0(t) + (1-t) \times x_{i+1}^0(t) \\ y_i^1(t) = t \times y_i^0(t) + (1-t) \times y_{i+1}^0(t) \end{cases}$.
- On recommence l'étape précédente jusqu'à obtenir le point P_0^N qui correspond au point $P(t)$ de la courbe de Bézier ayant les points $P_0^0, P_1^0, P_2^0, ..., P_N^0$ pour points de contrôle.

5. Pourquoi l'algorithme précédent se termine-t-il ?

L'algorithme précédent se termine car on part d'un nombre fini de points (les $N+1$ points de contrôle) pour aboutir à un unique point, chaque étape de l'algorithme correspondant à des calculs sur une liste de points dont la longueur est une suite strictement décroissante (elle diminue de 1 à chaque appel récursif).

6. Ecrire une fonction récursive `PointBezier_rec` qui renverra le point $P(t) = P_0^N$ et recevra en argument une liste de points et un réel *t* (dans $[0;1]$). Un point sera représenté par une liste de deux éléments correspondant à ses coordonnées. De fait, une liste de points sera une liste de listes. Avec l'exemple fourni en début d'exercice, on appellera initialement la fonction avec la liste *PC* :

`[[-6,-2],[-2,15],[0,-10],[4,-6],[7,-6],[8,4],[4,10]]`

Le cas d'arrêt correspond à une liste de longueur 1. (Attention, il s'agit toujours d'une liste de liste même si elle ne contient… qu'une seule liste !).
Dans les autres cas, on crée une nouvelle liste (`L2` dans le code ci-dessous) :

```
def PointBezier_rec(L,t):
    l = len(L)
    if l == 1:
        return L[0]
    else:
        L2 = []
        for i in range(l-1):
            L2.append([t*L[i][0]+(1-t)*L[i+1][0],t*L[i][1]
            +(1-t)*L[i+1][1]])
        return(PointBezier_rec(L2,t))
```

7. En supposant disponible la fonction `PointBezier_rec`, écrire une fonction `Bezier_Casteljau` qui recevra comme arguments une liste *PC* de points et un entier *NP* et tracera *NP* points (correspondant à $NP+1$ valeurs équiréparties de *t* dans $[0;1]$) de la courbe de Bézier ayant pour points de contrôle les points de *PC*.

```
def Bezier_Casteljau(PC,N):
    x, y = [], []
    pas = 1/N
    for i in range(N+1):
        t = i * pas
        L = PointBezier_rec(PC,t)
        x.append(L[0])
        y.append(L[1])
    # AFFICHAGE GRAPHIQUE
    # --> Listes des coordonnées
    xPC, yPC = [],[]
    for i in range(len(PC)):
        xPC.append(PC[i][0])
        yPC.append(PC[i][1])
    plt.clf()
    # --> Le polygône des points de contrôle
    plt.plot(xPC,yPC,"--r")
    # --> La courbe proprement dite
    plt.plot(x,y,"b",linewidth=1.5)
    plt.show()
```