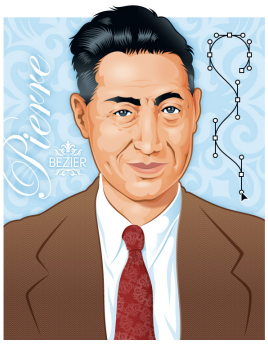
Calculer avec des points : courbes de Bézier (Lycée Maths/ISN)

Guillaume Connan*- Irem de Nantes

JA de l'Irem de Nantes - Jeudi 9 avril 2015

Résumé Au collège et au lycée, on demande aux élèves d'éviter d'effectuer des opérations arithmétiques sur des points ou même sur des coordonnées. Il serait en effet de définir à ce niveau des espaces affines...
D'un autre côté l'usage de logiciels de dessin et de calcul s'impose et exige de ne pas s'étonner d'additionner des points par exemple. Nous essaierons d'introduire ces notions sans trop de théorie en prenant comme prétexte que ça marche comme ça sur les machines...Non, ne me jettez pas de tomates...

1 La petite histoire



Dans les années 60, les ingénieurs Pierre Bézier et Paul De Casteljau travaillant respectivement chez Renault et Citroën, réfléchissent au moyen de définir de manière la plus concise possible la forme d'une carrosserie.
Le principe a été énoncé par Bézier mais l'algorithme de construction par son collègue de la marque aux chevrons qui n'a d'ailleurs été dévoilé que bien plus tard, la loi du secret industriel ayant primé sur le développement scientifique...
Pour la petite histoire Pierre Bézier (diplômé de l'Ensam et de Supélec) fut à l'origine des premières machines à commandes numériques et de la Cao ce qui n'empêcha pas sa direction de le mettre à l'écart : il se consacra alors presque exclusivement aux mathématiques et à la modélisation des surfaces et obtint même un doctorat en 1977.
Paul De Casteljau était lui un mathématicien d'origine, ancien élève de la Rue d'Ulm, qui a un temps été employé par l'industrie automobile.
Aujourd'hui, les courbes de Bézier sont très utilisées en informatique.
Une Courbe de Bézier est une courbe paramétrique aux extrémités imposées avec des points de contrôle qui définissent les tangentes à cette courbe à des instants donnés.

2 Algorithme de Casteljau

2.1 Espaces affines ? ?

Nous allons assimiler les points M_1, M_2 et M à des courbes paramétrées...
Au lycée ?...Peut-on « vendre » un point comme une fonction du temps définie par son absciesse et son ordonnée ?...
Par exemple, un point P sera assimilé à ses coordonnées : $P = (2, 3)$.
On se permet d'additionner des points, de multiplier des points par des réels.
Nous illustrerons nos propos avec Python qui est un langage assez largement utilisé. Nous aurions pu utiliser Xcas ou tout autre logiciel utilisé au lycée.
Nous représenterons nos points par des « **`array`** » de la bibliothèque **`NumPy`** qui ont l'avantage de supporter ces opérations. Nous définirons une fonction **`pt`** pour clarifier le code.

*Guillaume.Connan@univ-nantes.fr

```python
import numpy as np

def pt(x,y):
    return np.array([x,y])
```

Ainsi :

```
In [17]: p = pt(2,3)

In [18]: p
Out[18]: array([2, 3])

In [19]: 2*p
Out[19]: array([4, 6])

In [20]: p + p
Out[20]: array([4, 6])
```

Soit t un paramètre de l'intervalle $[0,1]$ et P_1, P_2 et P_3 les trois points de contrôle.
On construit le point M_1 barycentre du système

$$\{(P_1, 1-t), (P_2, t)\}$$

et M_2 celui du système

$$\{(P_2, 1-t), (P_3, t)\}$$

On construit ensuite le point M, barycentre du système

$$\{(M_1, 1-t), (M_2, t)\}$$

Exprimez M comme barycentre des trois points P_1, P_2 et P_3.
Avec ou sans vecteur ?
Est-ce qu'on peut dire qu'un vecteur est un déplacement et donc qu'il « bouge » un point en un autre ?

$$P = M + \vec{u}$$

Alors $P - M = \overrightarrow{MP}$...
Pour ? Contre ? Excuse du TP ? Débat...
Faites la construction à la main avec $t = 1/3$ par exemple.
Ainsi $M_1(t) = (1-t)P_1 + tP_2$, $M_2(t) = (1-t)P_2 + tP_3$ puis

$$M(t) = (1-t)M_1(t) + tM_2(t) = (1-t)^2 P_1 + 2t(1-t)P_2 + t^2 P_3$$

Vérifiez que $M'(t) = 2(M_2(t) - M_1(t))$. Comment l'interpréter ?

2.2 Sur machine ?

Bon, là c'est moins intéressant. On utilisera la bibliothèque **`matplotlib.pyplot`** et en particulier sa fonction **`plot`** qui demande la liste des abscisses puis la liste des ordonnées.
Mais nos points ne sont pas donnés comme ça... On va créer une fonction magique pour pouvoir tracer nos courbes en donnant une liste de points en argument.
Il se trouve que la commande **`tab[:,k]`** permet d'extraire toutes les k^e composantes d'un « **`array`** ».

```python
import matplotlib.pyplot as plt

def myplot(tab):
""" tab est une liste (python) de points """
    tab = np.array(tab)
    plt.plot(tab[:,0],tab[:,1])
```

Il reste à définir la fonction qui calcule $M(t)$ en fonction des trois points de contrôle :

```
def b2(p,t):
    return (1 - t)**2 * p[0] + 2*t*(1 - t)*p[1] + t**2*p[2]
```

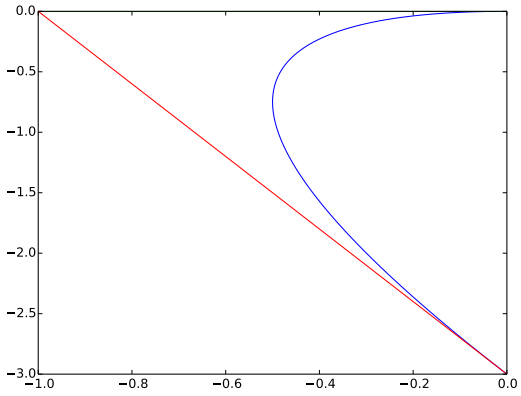
Voici une liste de trois points :

```
p_2 = [ pt(0,0), pt(-1,0), pt(0,-3) ]
```

Comment la tracer ?

```
def bezier2(p):
    ps = [ b2(p, t*0.01) for t in range(101) ]
    myplot(ps)     # Courbe de Bezier
    myplot(p[:2])  # Depart
    myplot(p[-2:]) # Arrivee
    #plt.savefig('./bezier2.pdf') # si on veut l'enregistrer
    plt.show() # pour voir
```

et cela donne :

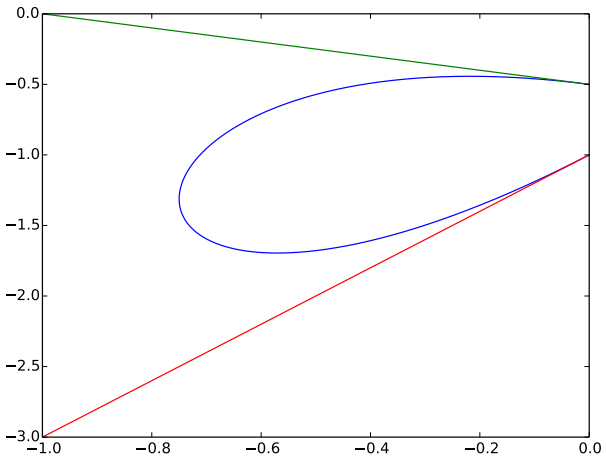


2.3 Avec 4 points de contrôle

Faites une étude similaire (« à la main ») avec 4 points de contrôle.
On pourra utiliser une représentation similaire aux arbres de probabilité.
Quel est le rôle des différents points de contrôle ?
Par exemple avec ça :

```
p_3 = [pt(0,-0.5), pt(-1,0), pt(-1,-3), pt(0,-1)]
```

on obtient :



2.4 Calsteljau modèle automatique

Pourrait-on oser ça :

```
def castel_der(p,t,n):
    if n == 0 :
        return p
    else :
        ps = zip( p[1:], p[:-1] )
        lp = [ t*p1 + (1 - t)*p2 for (p1,p2) in ps ]
        return castel_der( lp, t, n - 1 )
```

Ou bien ça :

```
def binom(n,k):
    assert (0 <= k <= n), "Coefficient binomial non calculable"
    if k == 0 :
        return 1
    else:
        return (n - (k - 1)) * binom(n, k - 1) // k

def bernstein(t,n,j):
    return binom(n,j)* t**j * (1 - t)**(n - j)

def casteljau(p,t):
    n = len(p)
    return sum([bernstein(t,n - 1,j) * p[j] for j in range(n)])
```

On peut ainsi traiter un nombre quelconque de points.

2.5 Animation

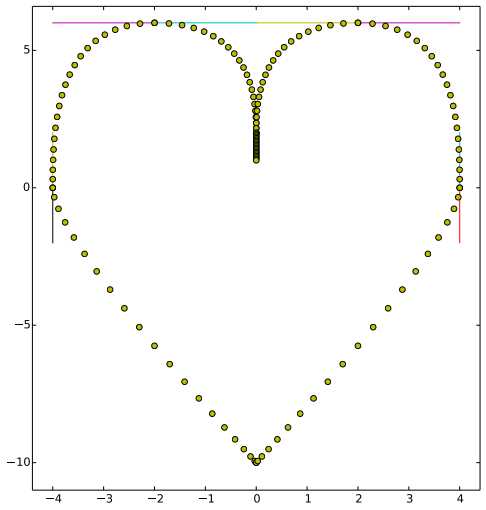
```
def bezier_anim(p):
    plt.ion() # mode interaction on
    myplot(p[:2])  # Depart
    myplot(p[-2:]) # Arrivee
    #X , Y = [pp[0] for pp in p], [pp[1] for pp in p]
    #xmin, xmax,ymin,ymax = min(X),max(X),min(Y),max(Y)
    for t in range(101):
```

```
        pb = casteljau(p, t*0.01)
        plt.plot(pb[0], pb[1], 'yo') # Courbe de Bezier
        myplot(castel_der(p, t*0.01, len(p) - 2)) # les tangentes
        #plt.axis([xmin*1.1,xmax*1.1,ymin*1.1,ymax*1.1])
        plt.draw()
    plt.ioff()
    plt.show()
```

2.6 Courbe de Bézier du 3e degré avec un nombre quelconque de points de contrôle

Il est pratique de travailler avec des polynômes de degré trois pour avoir droit à des points d'inflexion.
Augmenter le nombre de points de contrôle implique a priori une augmentation du degré de la fonction polynomiale.
Pour remédier à ce problème, on découpe une liste quelconque en liste de listes de 4 points.
Cependant, cela est insuffisant pour obtenir un raccordement de classe $\mathcal{C}^1$ (oui, bon, on présentera les choses autrement....avoir une carrosserie sans pics)
Pour assurer la continuité tout court, il faut que le premier point d'un paquet soit le dernier du paquet précédent.
Le dernier « vecteur vitesse » de la liste $[P_1, P_2, P_3, P_4]$ est $\overrightarrow{P_3P_4}$. Il faut donc que ce soit le premier vecteur vitesse du paquet suivant pour assurer la continuité de la dérivée.
Appelons provisoirement le paquet suivant $[P'_1, P'_2, P'_3, P'_4]$. On a d'une part $P'_1 = P_4$ et d'autre part $\overrightarrow{P_3P_4} = \overrightarrow{P'_1P'_2}$, i.e. $P'_2 = P_4 + \overrightarrow{P_3P_4}$.
On a donc $P'_3 = P_5$ et $P'_4 = P_6$.
Pour la Saint-Valentin offrez un code Python...

```
def bezier3_anim(p):
    plt.ion()
    pc = p[:2]
    for (p1,p2) in zip(p[2::2],p[3::2]):
        pc += [p1,p2,p2,2*p2 - p1]
    myplot(pc[:2])
    myplot(pc[-2:])
    X , Y = [pp[0] for pp in pc], [pp[1] for pp in pc]
    xmin, xmax,ymin,ymax = min(X),max(X),min(Y),max(Y)
    for k in range(0,len(pc),4):
        quad = pc[k:k+4]
        myplot(quad[:2])
        myplot(quad[-2:])
        for t in range(21):
            pb = casteljau(quad, t*0.05)
            plt.plot(pb[0], pb[1], 'yo')
            #myplot(castel_der(quad, t*0.05, len(quad) - 2))
            plt.axis([xmin*1.1,xmax*1.1,ymin*1.1,ymax*1.1])
            plt.draw()
    plt.ioff()
    plt.show()
```



B-splines uniformes

Tout ceci est très beau mais il y a un hic : en changeant un point de contrôle, on modifie grandement la figure.
On considère m nœuds $t_0, t_1 n, \cdots, t_m$ de l'intervalle $[0,1]$.
Introduisons une nouvelle fonction :

$$\boldsymbol{S}(t) = \sum_{i=0}^{m-n-1} \boldsymbol{P}_i b_{i,n}(t) \ , \ t \in [0,1]$$

les P_i étant les points de contrôle et les fonctions $b_{j,n}$ étant définies récursivement par

$$b_{j,0}(t) = \begin{cases} 1 & \text{si} \quad t_j \leqslant t < t_{j+1} \\ 0 & \text{sinon} \end{cases}$$

et pour $n \geqslant 1$

$$b_{j,n}(t) = \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n+1} - t_{j+1}} b_{j+1,n-1}(t).$$

On ne considérera par la suite que des nœuds équidistants : ainsi on aura $t_k = \frac{k}{m}$.
On parle de B-splines uniformes et on peut simplifier la formule précédente en remarquant également des invariances par translation.
À l'aide des formules précédentes, on peut prouver que dans le cas de 4 points de contrôle on obtient :

$$S(t) = \frac{1}{6}\Big((1-t)^3 P_0 + (3t^3 - 6t^2 + 4)P_1 + (-3t^3 + 3t^2 + 3t + 1)P_2 + t^3 P_3\Big)$$

Calculez $S(0)$, $S(1)$ puis $S'(0)$ et $S'(1)$: que peut-on en conclure ?
Reprenez l'étude faite avec les courbes de Bézier