Bezier

March 22, 2022

1 Courbes de Bézier

Marc Lorenzi

19 mars 2022

[1]:
```
import matplotlib.pyplot as plt
import math
```

1.1 1. Introduction

1.1.1 1.1 Les polynômes de Bernstein

Pour tout entier naturel n et tout entier $k \in [\![0, n]\!]$, notons

$$B_k^n = \binom{n}{k} X^k (1-X)^{n-k}$$

Les polynômes B_k^n sont les *polynômes de Bernstein*. Ces polynômes sont inéressants à plus d'un titre, ils interviendront dans tout le notebook. Commençons par une propriété facile.

Proposition. Pour tout $n \in \mathbb{N}$,

$$\sum_{k=0}^{n} B_k^n = 1$$

Démonstration.

$$\sum_{k=0}^{n} B_k^n = \sum_{k=0}^{n} \binom{n}{k} X^k (1-X)^{n-k} = (X + (1-X))^n = 1 \diamond$$

Voici un résultat moins évident.

Proposition. Soit $n \in \mathbb{N}$. La famille $(B_k^n)_{0 \leq k \leq n}$ est une base de l'espace vectoriel $\mathbb{R}_n[X]$ des polynômes de degré inférieur ou égal à n.

Démonstration. Montrons-le par récurrence sur n. Pour $n = 0$, c'est évident car $B_0^0 = 1$. Soit $n \geq 1$. Supposons la propriété vraie pour $n-1$. Remarquons que si $1 \leq k \leq n-1$, on a

$$\begin{aligned}(B_k^n)' &= k\binom{n}{k}X^{k-1}(1-X)^{n-k} - (n-k)\binom{n}{k}X^k(1-X)^{n-k-1}\\ &= n\binom{n-1}{k-1}X^{k-1}(1-X)^{n-k} - n\binom{n-1}{k}X^k(1-X)^{n-k-1}\\ &= nB_{k-1}^{n-1} - nB_k^{n-1}\end{aligned}$$

De plus,

$$(B_0^n)' = -nB_0^{n-1}$$

et

$$(B_n^n)' = nB_{n-1}^{n-1}$$

Soient $\lambda_0, \ldots, \lambda_n$ $n+1$ réels. Supposons que

$$(1)\ \sum_{k=0}^{n} \lambda_k B_k^n = 0$$

En dérivant et en divisant par n, on obtient

$$-\lambda_0 B_0^{n-1} + \lambda_n B_{n-1}^{n-1} + \sum_{k=1}^{n-1} \lambda_k \left(B_{k-1}^{n-1} - B_k^{n-1}\right) = 0$$

Ceci peut encore s'écrire, avec un petit changement d'indice,

$$-\lambda_0 B_0^{n-1} + \lambda_n B_{n-1}^{n-1} - \sum_{k=1}^{n-1} \lambda_k B_k^{n-1} + \sum_{k=0}^{n-2} \lambda_{k+1} B_k^{n-1} = 0$$

Par l'hypothèse de récurrence, la famille $(B_k^{n-1})_{0\leq k\leq n-1}$ est libre. On en déduit que pour $0 \leq k \leq n-1$, le coefficient de B_k^{n-1} dans l'égalité ci-dessus est nul. Il en résulte facilement que

$$\lambda_0 = \ldots = \lambda_n$$

En reportant dans (1), on en déduit que

$$\lambda_0 \sum_{k=0}^{n} B_k^n = \lambda_0 = 0$$

La famille $(B_k^n)_{0\leq k\leq n}$ est donc libre. Comme elle est de cardinal $n+1 = \dim \mathbb{R}_n[X]$, c'est une base de $\mathbb{R}_n[X]$. $\diamond$

La fonction `binom` prend en paramètres deux entiers n et k. Elle renvoie le coefficient binomial $\binom{n}{k}$. Plus exactement, elle en renvoie une approximation flottante.

[2]:
```
def binom(n, k):
    p = 1
    for i in range(k):
        p *= (n - i) / (i + 1)
    return p
```

[3]:
```
print([binom(10, k) for k in range(11)])
```

```
[1, 10.0, 45.0, 120.0, 210.0, 252.0, 210.0, 120.0, 45.0, 10.0, 1.0]
```

La fonction `bernstein` renvoie $B_k^n(t)$.

[4]:
```
def bernstein(n, k, t):
    return binom(n, k) * t ** k * (1 - t) ** (n -k)
```

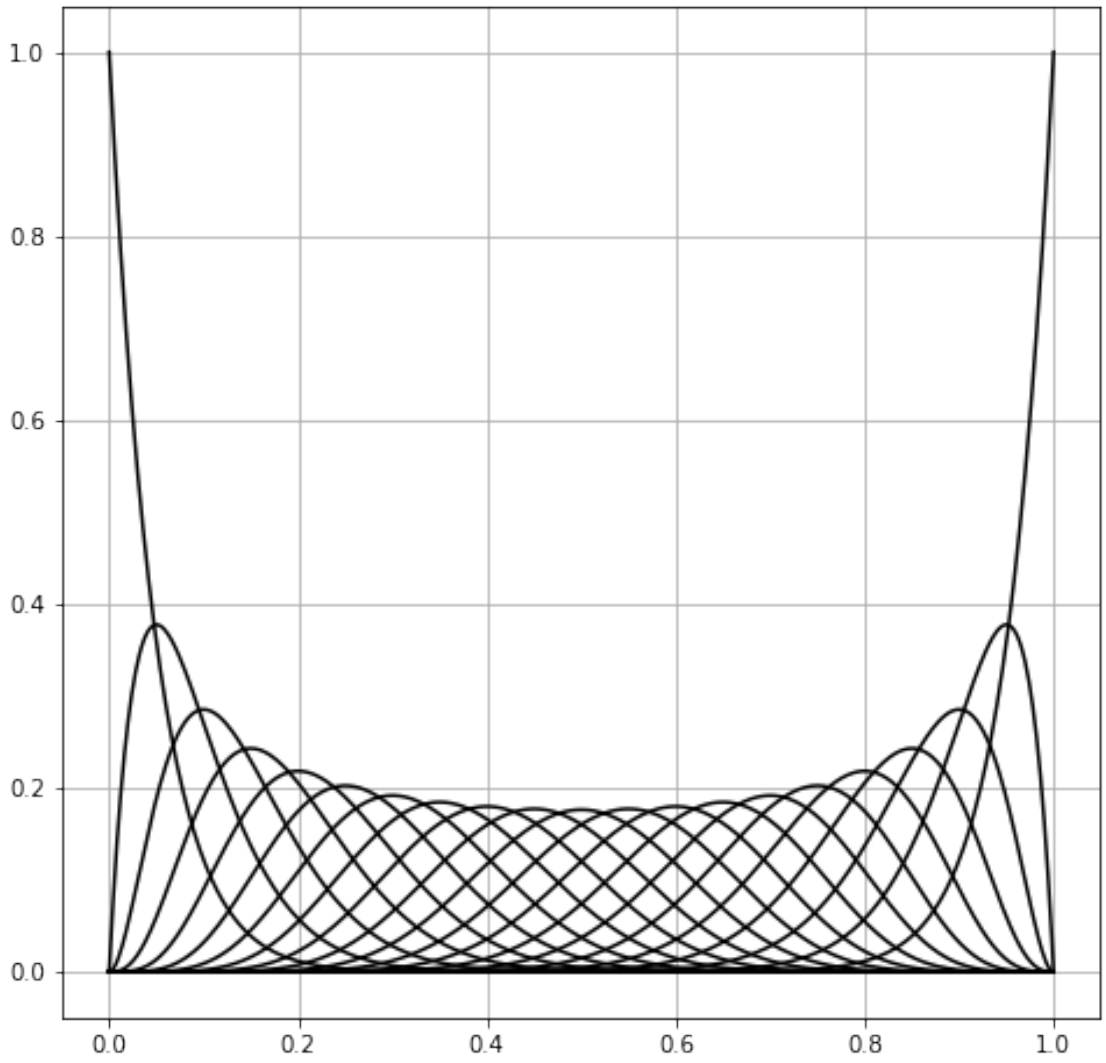
Traçons les courbes des polynômes de Bernstein sur $[0, 1]$.

[5]:
```
def subdi(a, b, n):
    d = (b - a) / n
    return [a + k * d for k in range(n + 1)]
```

[6]:
```
def plot_bernstein(n, rng=None):
    ts = subdi(0, 1, 500)
    if rng == None: rng = range(n + 1)
    for k in rng:
        bs = [bernstein(n, k, t) for t in ts]
        plt.plot(ts, bs, 'k')
    plt.grid()
```

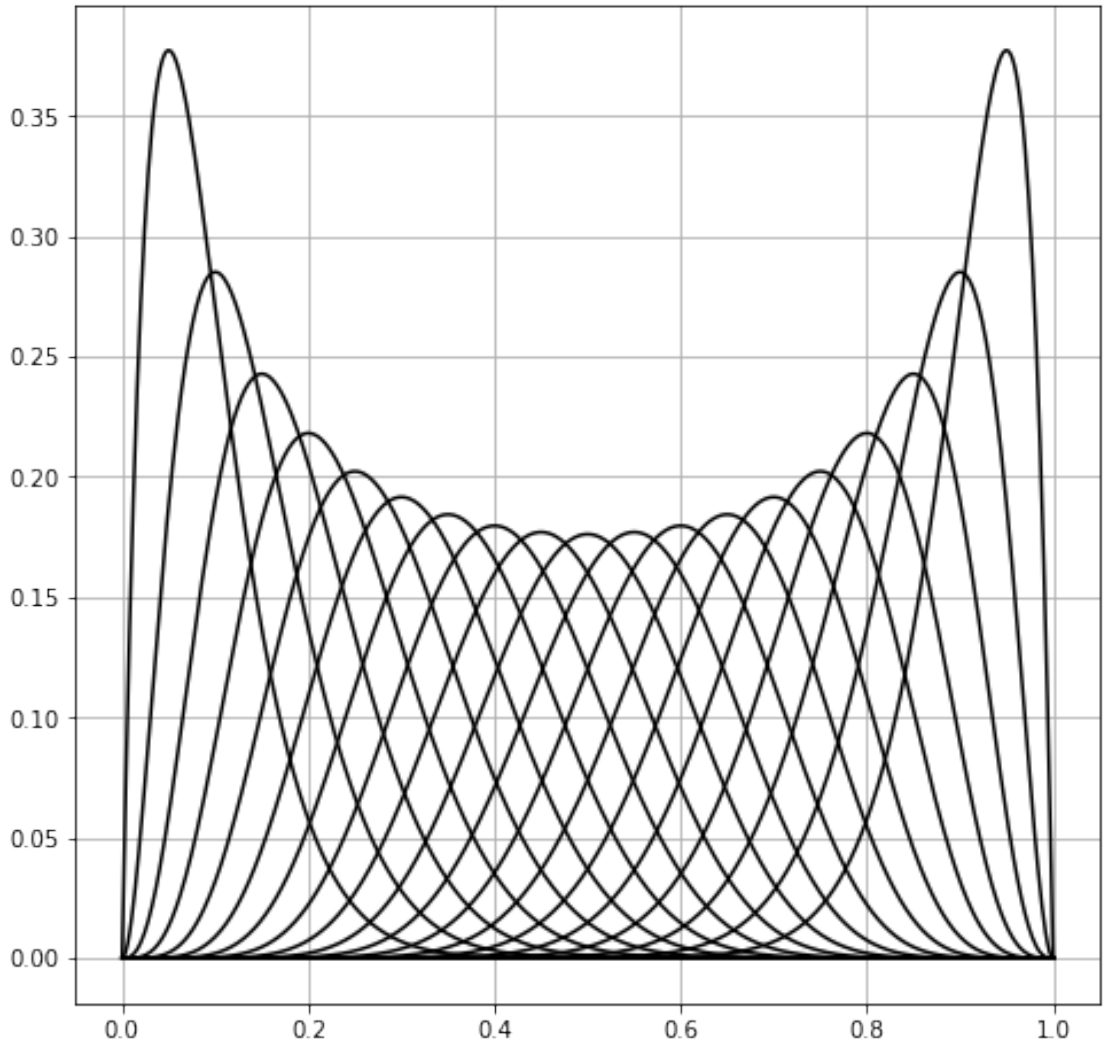
[7]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

[8]:
```
plot_bernstein(20)
```



Voici les mêmes courbes, en supprimant celles de B_0^n et B_n^n.

[9]:
```
plot_bernstein(20, range(1, 20))
```



1.1.2 1.2 Courbes de Bézier

Soit $n \in \mathbb{N}$. Soient $P_0, \ldots, P_n$ $n+1$ points du plan. Soit $t \in [0,1]$. Considérons le point $B_{P_0,\ldots,P_n}(t)$ défini par

$$\overrightarrow{OB_{P_0,\ldots,P_n}(t)} = \sum_{k=0}^{n} B_k^n(t)\overrightarrow{OP_k}$$

On définit ainsi une fonction

$$B_{P_0,\ldots,P_n} : [0,1] \longrightarrow \mathbb{R}^2$$

Lorsqu'il n'y a pas d'ambiguïté sur les points P_i, nous noterons plus simplement $B(t)$ au lieu de $B_{P_0,\dots,P_n}(t)$. Pour alléger encore les notations, nous écrirons

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k$$

Les points $P_0, \dots, P_n$ sont appelés *points de contrôle*. Lorsque t décrit le segment $[0,1]$, le point $B(t)$ décrit une courbe qui s'appelle la *courbe de Bézier* associée aux points de contrôle $P_0, \dots, P_n$.

Les courbes de Bézier possèdent des propriétés tout à fait remarquables que nous allons explorer. Avant tout, remarquons les évidences. L'abscisse et l'ordonnée de $B(t)$ sont des fonctions polynomiales en t, combinaisons linéaires des B_k^n qui forment une base de $\mathbb{R}_n[X]$. Il est donc possible d'obtenir de cette manière, par un choix judicieux des points de contrôle, *toutes* les fonctions de $[0,1]$ vers $\mathbb{R}^2$ dont l'abscisse et l'ordonnée sont des fonctions polynomiales.

Remarquons aussi que $B(0) = P_0$ et $B(1) = P_n$. En revanche, sauf coïncidences, les points $P_1, \dots, P_{n-1}$ n'appartiennent pas à la courbe de Bézier.

Regardons ce qui se passe pour $n = 0$ et $n = 1$.

- Si $n = 0$, on a pour tout $t \in [0,1]$, $B(t) = P_0$. La fonction B est constante.
- Si $n = 1$, on a pour tout $t \in [0,1]$, $B(t) = (1-t)P_0 + tP_1$. La courbe de Bézier est le segment $[P_0, P_1]$.

La fonction `bezier0` prend en paramètres une liste `Ps` de points du plan (représentés chacun par un couple de flottants) et un réel $t \in [0,1]$. Elle renvoie $B(t)$.

[10]:
```
def bezier0(Ps, t):
    x = 0
    y = 0
    n = len(Ps) - 1
    for k in range(n + 1):
        xk, yk = Ps[k]
        m =  bernstein(n, k, t)
        x += m * xk
        y += m * yk
    return (x, y)
```

[11]:
```
Ps = [(0, 0), (1, 2), (2, 2), (4, 1), (4, 0)]
print(bezier0(Ps, 0.5))
```

```
(2.25, 1.5)
```

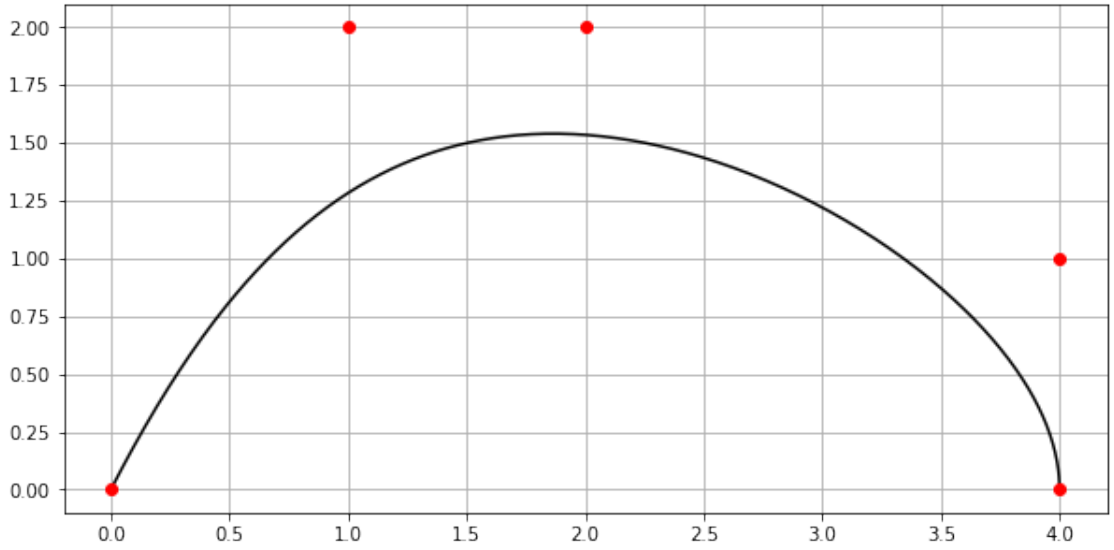
1.1.3 1.3 Tracé dezs courbes de Bézier

La fonction `plot_bezier1` prend en paramètre une liste de points du plan. Si le paramètre `pts` vaut `True` elle affiche ces points. Elle affiche également la courbe de Bézier dont ces points sont les points de contrôle.

[12]:
```
def plot_bezier1(Ps, pts=True):
    ts = subdi(0, 1, 500)
    Qs = [bezier0(Ps, t) for t in ts]
    xs = [x for (x, y) in Qs]
    ys = [y for (x, y) in Qs]
    plt.plot(xs, ys, 'k')
    if pts:
        for (x, y) in Ps:
            plt.plot([x], [y], 'or')
    plt.grid()
```

[13]:
```
plt.rcParams['figure.figsize'] = (10, 5)
```

[14]:
```
Ps = [(0, 0), (1, 2), (2, 2), (4, 1), (4, 0)]
plot_bezier1(Ps)
```

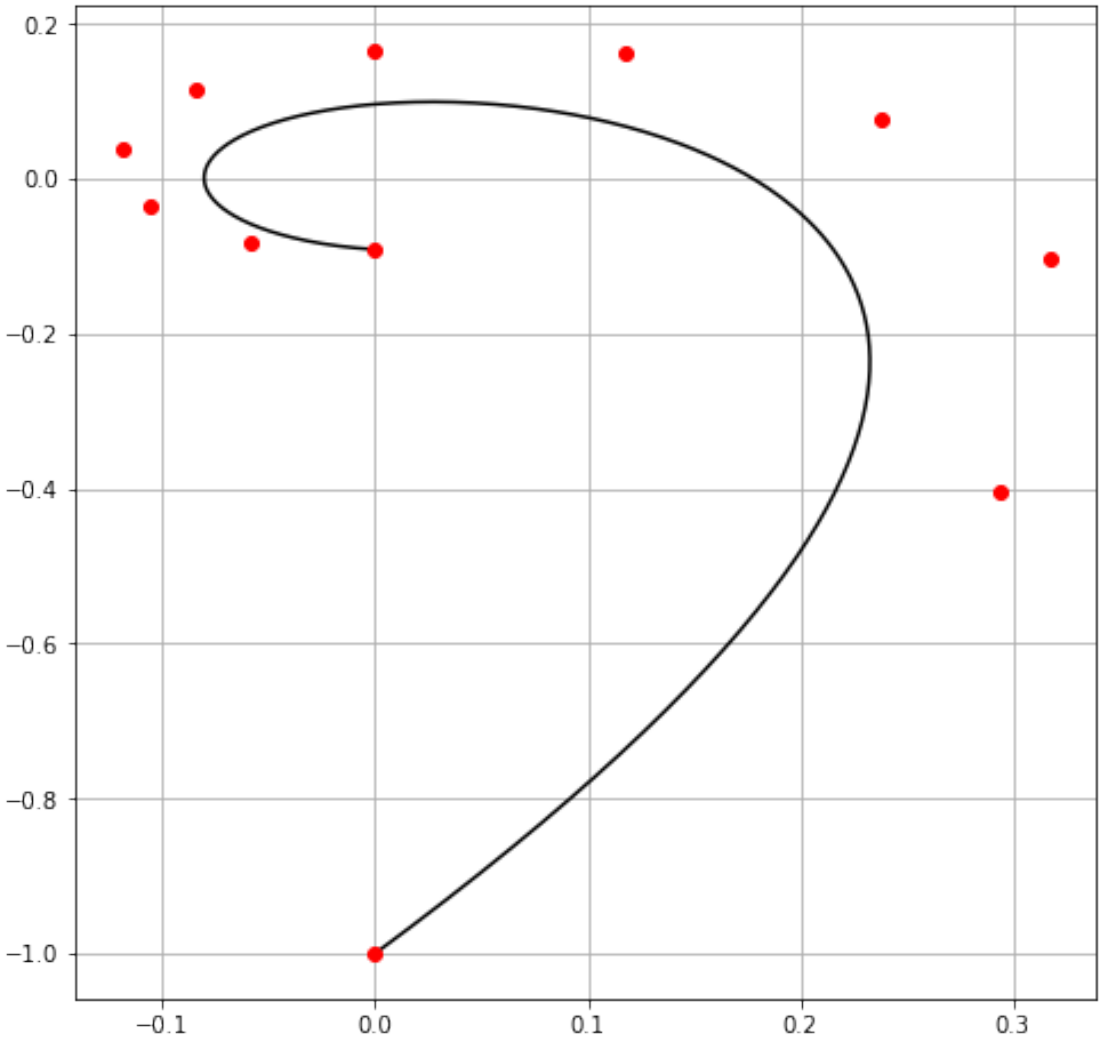


Voici une fonction `cycle` qui renvoie une liste de $n+1$ points situés sur une « spirale ».

[15]:
```
def cycle(n):
    Ps = []
    for k in range(n + 1):
        t = - math.pi / 2 + 2 * k * math.pi / n
        Ps.append((math.cos(t) / (k + 1), math.sin(t) / (k + 1)))
    return Ps
```

[16]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

[17]:
```
plot_bezier1(cycle(10))
```



1.1.4 1.4 Quelques remarques

Deux remarques concernant notre fonction `bezier0` :

- Lorsque n est grand les coefficients binomiaux deviennent **très** grands. En fait, **trop** grands. Les erreurs d'arrondis sur les sommes deviennent prépondérandes et `bezier0` renvoie des résultats faux. Essayez donc de prendre $n = 200$ ci-dessus. Vous verrez que la courbe a « disparu ».
- Le calcul de $B(t)$ nécessite n additions et, pour chaque terme, le calcul d'un coefficient binomial. La complexité de ce calcul est $O(n^2)$.

1.2 2. Premières propriétés

1.2.1 2.1 Barycentres

Rappelons la formule

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k$$

Pour tout $t \in [0,1]$, $B_k^n(t) \geq 0$ et nous avons vu que

$$\sum_{k=0}^{n} B_k^n(t) = 1$$

Ainsi, le point $B(t)$ est un barycentre à coefficients positifs des points de contrôle P_k. Il en résulte que la courbe de Bézier est incluse dans *l'enveloppe convexe* de l'ensemble $\{P_0, \ldots, P_n\}$, c'est à dire la plus petite partie convexe du plan qui contient les points de contrôle.

1.2.2 2.2 Tangentes aux extrémités

Pour tout $k \in [\![0, n]\!]$, 0 est racine d'ordre k du polynôme B_k^n. On a donc pour tout $k \geq 2$, $(B_k^n)'(0) = 0$. De plus,

$$(B_0^n)'(t) = -n(1-t)^{n-1}$$

et

$$(B_1^n)'(t) = n(1-t)^{n-1} - n(n-1)t(1-t)^{n-2} = n(1-t)^{n-2}(1-nt)$$

De là,

$$B'(0) = -nP_0 + nP_1 = n\overrightarrow{P_0P_1}$$

Ainsi, la tangente à la courbe de Bézier au point $P_0 = B(0)$ est dirigée par le vecteur $\overrightarrow{P_0P_1}$.

Des calculs analogues montrent que la tangente à la courbe de Bézier au point $P_n = B(1)$ est dirigée par le vecteur $\overrightarrow{P_{n-1}P_n}$.

Nous commençons à comprendre pourquoi les points P_k sont appelés des *points de contrôle*, en tout cas pour 4 de ces points.

- Le point P_0 est l'origine de la courbe de Bézier.
- Le point P_1 contrôle la tangente à l'origine.
- Le point P_n est l'extrémité de la courbe de Bézier.
- Le point P_{n-1} contrôle la tangente à l'extrémité.

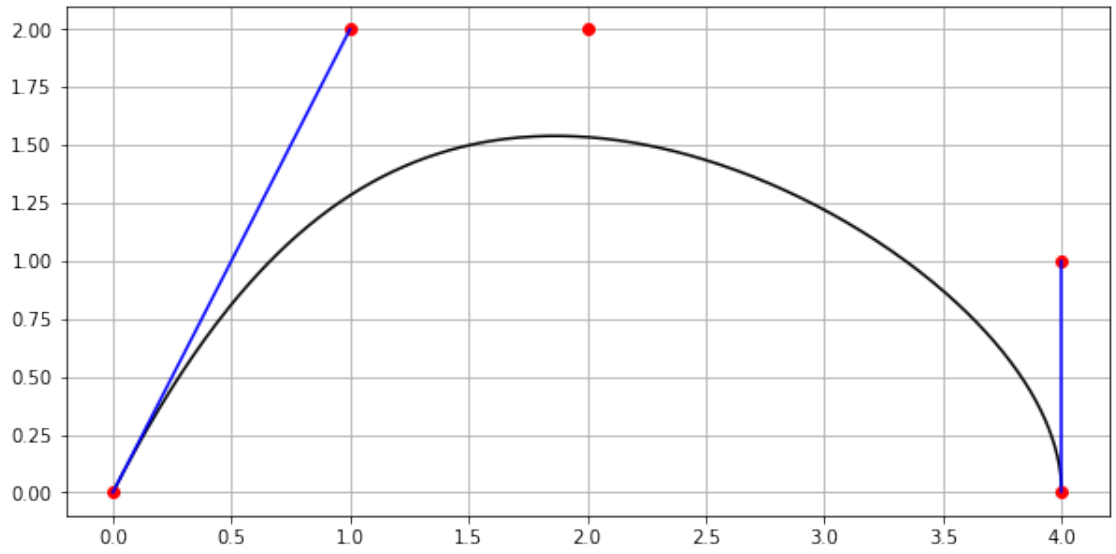
Remarquons que si $n = 3$ nous avons tout compris. Évidemment, si $n \geq 4$ il reste encore beaucoup à dire.

La fonction `plot_bezier2` trace la courbe de Bézier, les points de contrôle, et les tangentes aux extrémités de la courbe.

[18]:
```
def plot_bezier2(Ps):
    n = len(Ps) - 1
    ts = subdi(0, 1, 500)
    Qs = [bezier0(Ps, t) for t in ts]
    xs = [x for (x, y) in Qs]
    ys = [y for (x, y) in Qs]
    plt.plot(xs, ys, 'k')
    for (x, y) in Ps:
        plt.plot([x], [y], 'or')
    a, b = Ps[0]
    c, d = Ps[1]
    plt.plot([a, c], [b, d], 'b')
    a, b = Ps[n - 1]
    c, d = Ps[n]
    plt.plot([a, c], [b, d], 'b')
    plt.grid()
```

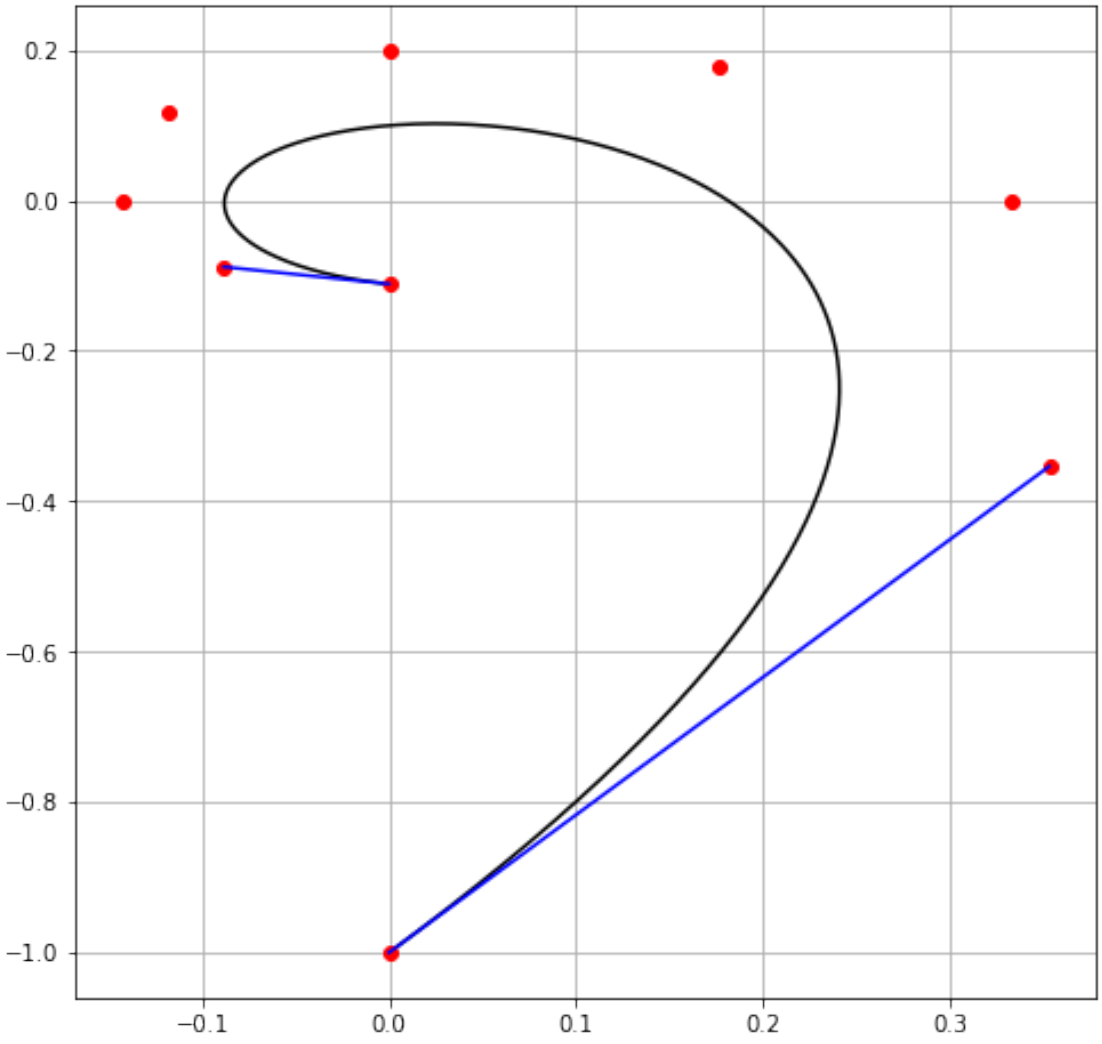
[19]:
```
plt.rcParams['figure.figsize'] = (10, 5)
```

[20]:
```
Ps = [(0, 0), (1, 2), (2, 2), (4, 1), (4, 0)]
plot_bezier2(Ps)
```



[21]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

[22]:
```
plot_bezier2(cycle(8))
```



1.3 3. L'algorithme de De Casteljau

Nous allons dans cette section discuter d'un algorithme qui permet le calcul des courbes de Bézier sans passer par des coefficients du binôme. Il s'agit de *l'algorithme de De Casteljau*. Cet algorithme a été développé à la fin des années 1950 par Paul de Faget de Casteljau, alors qu'il travaillait chez Citroën. Il a été « redécouvert » par Pierre Bézier au début des années 1960. Ce dernier travaillait chez Renault (il faut savoir que les recherches de De Casteljau étant protégées par le secret industriel, Bézier ne pouvait pas connaître les résultats de De Casteljau).

1.3.1 3.1 Introduction

Soit $t \in [0,1]$. Pour $k = 0, \ldots, n-1$, notons Q_k le barycentre de P_k et P_{k+1} affectés des coefficients $1-t$ et t. On a donc

$$Q_k = (1-t)P_k + tP_{k+1}$$

Considérons maintenant la courbe de Bézier associée aux points de contrôle $Q_0, \ldots, Q_{n-1}$. On a le résultat suivant.

Proposition $B_{Q_0,\ldots,Q_{n-1}}(t) = B_{P_0,\ldots,P_n}(t)$.

Démonstration. On a

$$B_{Q_0,\ldots,Q_{n-1}}(t) = \sum_{k=0}^{n-1} \binom{n-1}{k} t^k (1-t)^{n-1-k}((1-t)P_k + tP_{k+1}) = \sum_{k=0}^{n-1} \binom{n-1}{k} t^k (1-t)^{n-k} P_k + \sum_{k=0}^{n-1} \binom{n-1}{k} t^{k+1} (1-t)^n$$

Dans la deuxième somme (notons-la S_2), faisons le changement d'indice $k' = k+1$. Il vient

$$S_2 = \sum_{k=1}^{n} \binom{n-1}{k-1} t^k (1-t)^{n-k} P_k$$

De là,

$$B_{Q_0,\ldots,Q_{n-1}}(t) = (1-t)^n P_0 + t^n P_n + \sum_{k=1}^{n-1} \left(\binom{n-1}{k} + \binom{n-1}{k-1} \right) t^k (1-t)^{n-k} P_k$$

Remarquons que

$$\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$$

et donc

$$B_{Q_0,\ldots,Q_{n-1}}(t) = (1-t)^n P_0 + t^n P_n + \sum_{k=1}^{n-1} \binom{n}{k} t^k (1-t)^{n-k} P_k$$

On reconnaît dans les deux termes isolés du membre de droite deux quantités qui peuvent être intégrées à la somme. Finalement :

$$B_{Q_0,\ldots,Q_{n-1}}(t) = \sum_{k=0}^{n} \binom{n}{k} t^k (1-t)^{n-k} P_k = B_{P_0,\ldots,P_n}(t) \diamond$$

1.3.2 3.2 Itération

Quel est l'intérêt du calcul que nous venons de faire ? La courbe de Bézier $B_{Q_0,\dots,Q_{n-1}}$ est associée à n points de contrôle, alors que la courbe $B_{P_0,\dots,P_n}$ est associée à $n+1$ points. Nous pouvons recommencer en prenant les barycentres de deux points Q_k successifs pour fabriquer une courbe de Bézier $B_{R_0,\dots,R_{n-2}}$ associée à $n-1$ points de contrôle et telle que

$$B_{R_0,\dots,R_{n-2}}(t) = B_{Q_0,\dots,Q_{n-1}}(t) = B_{P_0,\dots,P_n}(t)$$

Et ainsi de suite. Après n itérations nous obtenons une courbe de Bézier B_S associée à un unique point de contrôle S. On a alors

$$B_{P_0,\dots,P_n}(t) = B_S(t) = S$$

Ainsi, par des calculs de barycentres successifs de deux points, on est capables de déterminer $B_{P_0,\dots,P_n}(t)$.

La fonction `casteljau` prend en paramètres une liste `Ps` de points du plan et un réel $t \in [0,1]$. Elle renvoie $B(t)$. Elle appelle une fonction `bary` dont nous parlerons juste après.

[23]:
```
def casteljau(Ps, t):
    n = len(Ps) - 1
    for i in range(n):
        Ps = bary(Ps, t)
    return Ps[0]
```

La fonction `bary` prend en paramètres une liste `Ps` de points du plan et un réel $t \in [0,1]$. Si `Ps`$= [P_0, \dots, P_n]$, la fonction renvoie $[Q_0, \dots, Q_{n-1}]$ où Q_k est le barycentre de P_k et P_{k+1} affectés des coefficients $1-t$ et t.

[24]:
```
def bary(Ps, t):
    n = len(Ps)
    Qs = []
    for k in range(n - 1):
        a, b = Ps[k]
        c, d = Ps[k + 1]
        Qs.append([(1 - t) * a + t * c, (1 - t) * b + t * d])
    return Qs
```

[25]:
```
Ps = [[0, 0], [1, 2], [2, 2], [4, 1], [4, 0]]
print(casteljau(Ps, 0.5))
```

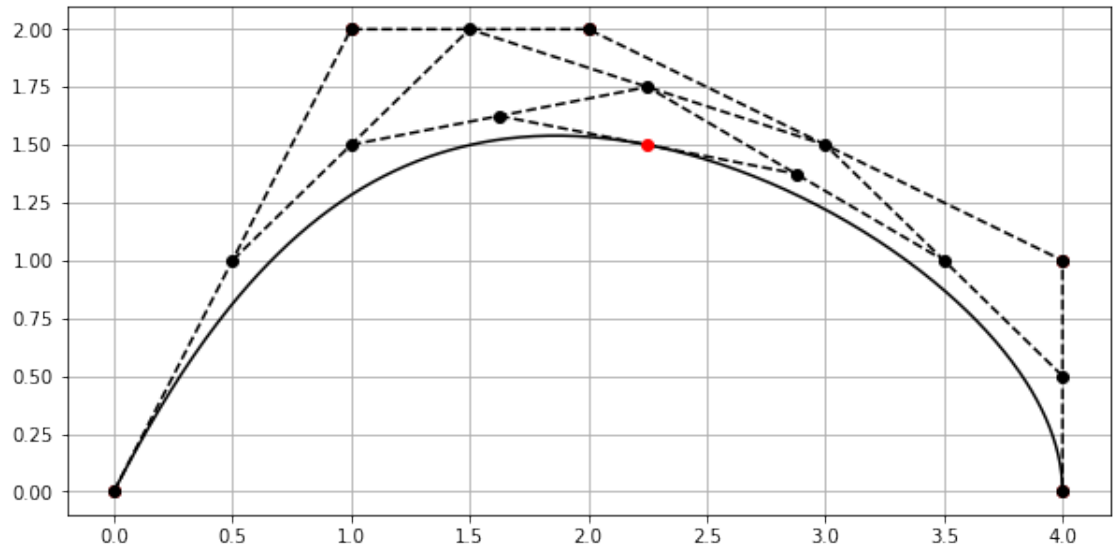
```
[2.25, 1.5]
```

Un petit dessin vaut mieux qu'un long discours … La fonction `demo_casteljau` prend en paramètres une liste `Ps` de points de contrôle et un réel $t \in [0,1]$. Elle affiche la construction de $B(t)$ par l'algorithme de Casteljau.

[26]:
```
def demo_casteljau(Ps, t):
    n = len(Ps) - 1
    ts = subdi(0, 1, 500)
    Qs = [bezier0(Ps, t) for t in ts]
    xs = [x for (x, y) in Qs]
    ys = [y for (x, y) in Qs]
    plt.plot(xs, ys, 'k')
    for (x, y) in Ps:
        plt.plot([x], [y], 'or')
    for i in range(n):
        xs = [x for (x, y) in Ps]
        ys = [y for (x, y) in Ps]
        plt.plot(xs, ys, '--k')
        plt.plot(xs, ys, 'ok')
        Ps = bary(Ps, t)
    x, y = casteljau(Ps, t)
    plt.plot([x], [y], 'or')
    plt.grid()
```

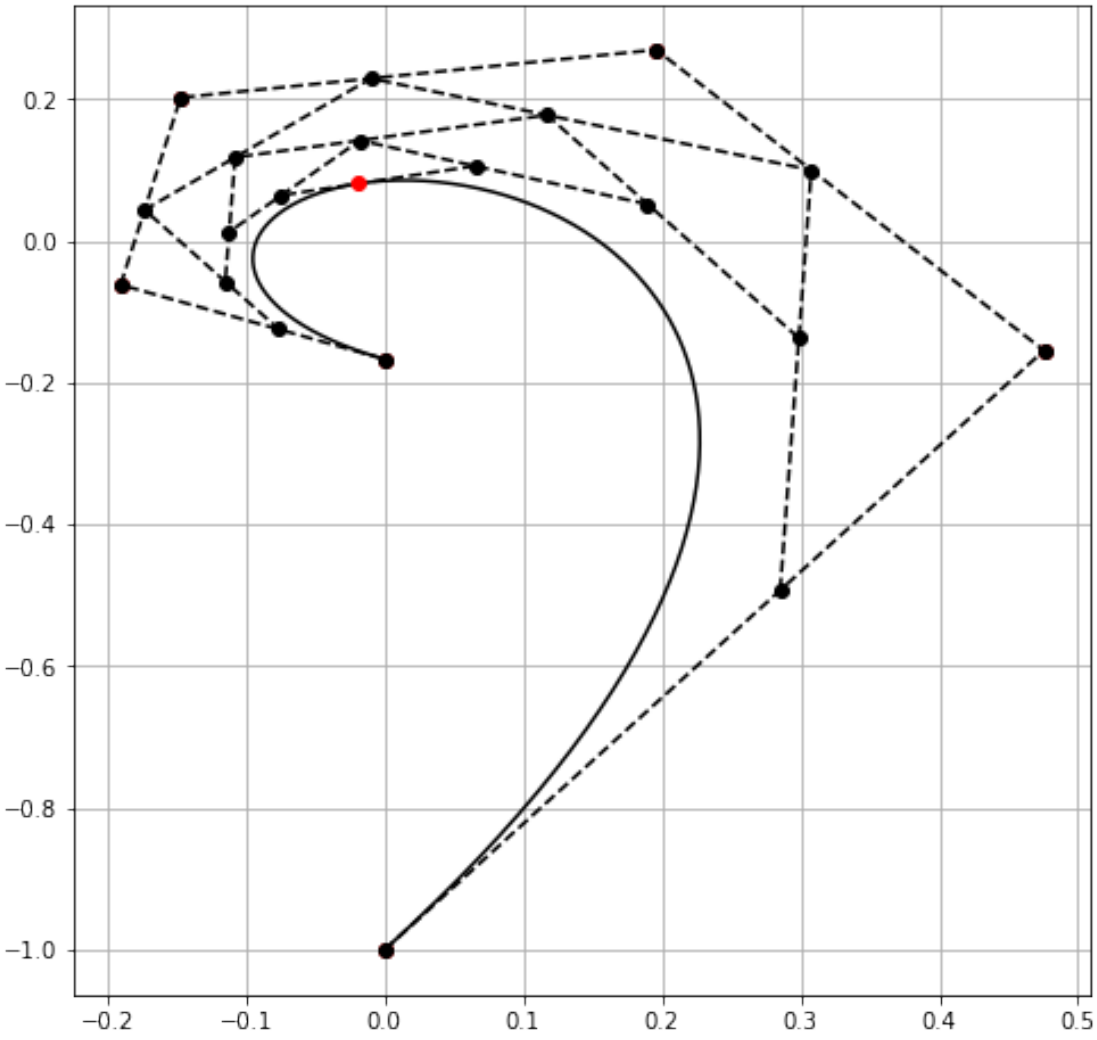
[27]:
```
plt.rcParams['figure.figsize'] = (10, 5)
```

[28]:
```
Ps = [(0, 0), (1, 2), (2, 2), (4, 1), (4, 0)]
demo_casteljau(Ps, 0.5)
```



[29]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

[30]:
```
Ps = [[0, 0], [1, 2], [2, 2], [4, 1], [4, 0]]
demo_casteljau(cycle(5), 0.6)
```



Remarque. Sur les dessins ci-dessus, il semble que la tangente à la courbe de Bézier au point rouge soit l'un des segments tracés. C'est effectivement le cas, nous y reviendrons plus loin.

1.3.3 3.3 Comment utiliser l'algorithme de De Casteljau

Utilisé naïvement, l'algorithme de De Casteljau calcule $B(t)$ en temps $O(n^2)$. Il n'y a donc aucun gain de temps par rapport à l'algorithme initial utilisant la formule. Notons tout de même que nous avons réussi à éliminer les problèmes liés aux erreurs cumulées. Il reste maintenant à voir comment on utilise précisément cet algorithme.

Mettons en place quelques notations. Notons tout d'abord $P_0^0, \dots, P_n^0$ les points de contrôle.

Par un premier calcul de barycentres, on obtient les points $P_0^1, \dots, P_{n-1}^1$ où P_i^1 est le barycentre de P_i^0 et P_{i+1}^0 affectés des coefficients $1-t$ et t.

Un second calcul de barycentres fournit les points $P_0^2, \ldots, P_{n-2}^2$ où P_i^2 est le barycentre de P_i^1 et P_{i+1}^1 affectés des coefficients $1-t$ et t.

Etc. On obtient ainsi une famille (P_i^k) de points du plan, où $0 \leq k \leq n$ et $0 \leq i \leq n-k$. Le point $B(t)$ est le point P_0^n. Nous avons le résultat suivant.

Proposition. La courbe de Bézier de points de contrôle $P_0, \ldots, P_n$ est la réunion de deux courbes de Bézier :

- La courbe de points de contrôle $P_0^0, P_0^1, \ldots, P_0^n$.
- La courbe de points de contrôle $P_0^n, P_1^{n-1}, \ldots, P_n^0$.

Pour prouver cette proposition, montrons d'abord le lemme suivant.

Lemme. Pour tout $k \in [\![0, n]\!]$, pour tout $i \in [\![0, n-k]\!]$,

$$P_i^k = \sum_{j=0}^{k} B_j^k(t) P_{i+j}^0$$

Démonstration du lemme. Montrons la propriété par récurrence sur k.

Pour $k = 0$, c'est immédiat. Soit $k \in [\![0, n-1]\!]$. Supposons que pour tout $i \in [\![0, n-k]\!]$,

$$P_i^k = \sum_{j=0}^{k} B_j^k(t) P_{i+j}^0$$

Soit $i \in [\![0, n-k-1]\!]$. On a

$$\begin{aligned} P_i^{k+1} &= (1-t)P_i^k + tP_{i+1}^k \\ &= (1-t)\textstyle\sum_{j=0}^{k} B_j^k(t) P_{i+j}^0 + t\sum_{j=0}^{k} B_j^k(t) P_{i+j+1}^0 \\ &= (1-t)\textstyle\sum_{j=0}^{k} B_j^k(t) P_{i+j}^0 + t\sum_{j=1}^{k+1} B_{j-1}^k(t) P_{i+j}^0 \\ &= (1-t)B_0^k(t)P_i^0 + tB_k^k(t)P_{i+k+1}^0 + \textstyle\sum_{j=1}^{k}((1-t)B_j^k(t) + tB_{j-1}^k(t))P_{i+j} \end{aligned}$$

Remarquons que

$$(1-t)B_j^k(t) + tB_{j-1}^k(t) = \left(\binom{k}{j} + \binom{k}{j-1}\right) t^j(1-t)^{k+1-j} = \binom{k+1}{j} t^j(1-t)^{k+1-j}$$

De plus,

$$(1-t)B_0^k(t) = (1-t)^{k+1} = B_0^{k+1}(t)$$

et

$$tB_k^k(t) = t^{k+1} = B_{k+1}^{k+1}(t)$$

De là,

$$P_i^{k+1} = \sum_{j=0}^{k+1} B_j^{k+1}(t) P_{i+j}$$

Démonstration de la proposition. On a pour tout $k \in [\![0, n]\!]$,

$$P_0^k = \sum_{j=0}^{k} B_j^k(t) P_j^0$$

Soit $u \in [0, 1]$. On a

$$\sum_{k=0}^{n} B_k^n(u) P_0^k = \sum_{k=0}^{n} B_k^n(u) \sum_{j=0}^{k} B_j^k(t) P_j^0 = \sum_{j=0}^{n} S_j P_j^0$$

où

$$S_j = \sum_{k=j}^{n} B_j^k(t) B_k^n(u)$$

Que vaut S_j ? On a

$$S_j = \sum_{k=j}^{n} B_j^k(t) B_k^n(u) = \sum_{k=j}^{n} \binom{k}{j} t^j (1-t)^{k-j} \binom{n}{k} u^k (1-u)^{n-k}$$

Remarquons que

$$\binom{j}{k}\binom{k}{n} = \binom{n}{j}\binom{n-j}{k-j}$$

De là,

$$\begin{array}{rcl} S_j & = & \binom{n}{j} \sum_{k=j}^{n} \binom{n-j}{k-j} t^j (1-t)^{k-j} u^k (1-u)^{n-k} \\ & = & \binom{n}{j} \sum_{k=0}^{n-j} \binom{n-j}{k-j} t^j (1-t)^k u^{k+1} (1-u)^{n-k-j} \\ & = & \binom{n}{j} (tu)^j \sum_{k=0}^{n-j} \binom{n-j}{k} (u(1-t))^k (1-u)^{n-j-k} \\ & = & \binom{n}{j} (tu)^j (1-u+u(1-t))^{n-j} \\ & = & \binom{n}{j} (tu)^j (1-tu)^{n-j} \\ & = & B_j^n(tu) \end{array}$$

Ainsi,

$$\sum_{k=0}^{n} B_k^n(u) P_0^k = \sum_{j=0}^{n} B_j^n(tu) P_j^0 = B(tu)$$

Pour $u = 0$, $tu = 0$. Pour $u = 1$, $tu = t$. On obtient donc la partie de la courbe de B pour les valeurs du paramètre entre 0 et t.

où B est la fonction de Bézier associée aux points de contrôle $P_0, \ldots, P_n$.

On montre de même que

$$\sum_{k=0}^{n} B_k^n(u) P_{n-k}^k = B(1 - (1-t)u)$$

Pour $u = 0$, $1 - (1-t)u = 1$. Pour $u = 1$, $1 - (1-t)u = t$. On obtient donc la partie de la courbe de B pour les valeurs du paramètre entre t et 1.

Remarque. Nous pouvons maintenant confirmer ce que nous avions remarqué un peu plus haut au sujet des tangentes à la courbe de Bézier. Le point $B(t)$ est l'extrémité de la courbe de Bézier associée aux points de contrôle P_0^0, P_0^1, … P_0^n. La tangente à gauche à la courbe en ce point est donc dirigée par le vecteur $\overrightarrow{P_0^{n-1}P_0^n}$. De même, la tangente à droite est dirigée par le vecteur $\overrightarrow{P_{n-1}^1 P_n^0}$.

Pour obtenir la courbe de Bézier complète, il suffit d'appeler récursivement notre algorithme sur deux « moitiés » de courbe. L'algorithme s'arrête lorsqu'un certain critère est vérifié. Dans les applications pratiques, on travaille avec des points à coordonnées entières représentant des pixels et on prend $t = \frac{1}{2}$. Ainsi, tous les calculs peuvent être réalisés avec des entiers. Dans ce cas, un bon critère d'arrêt est que les points de contrôle soient à des distances inférieures à 1. En effet, comme la courbe de Bézier est contenue dans l'enveloppe convexe de l'ensemble des points de contrôle, elle est alors incluse dans un pixel.

Nous travaillons quant à nous avec des points dont les coordonnées sont des flottants. Pour simplifier, dans la fonction `bezier` ci-dessous, le critère choisi est le nombre r d'appels récursifs, passé en paramètre à la fonction. Lorsque $r = 0$, la fonction renvoie les deux points de contrôle extrêmes : ces deux points, nous le savons, font partie de la courbe de Bézier. Cette fonction prend également un paramètre optionnel t, fixé par défaut à $\frac{1}{2}$. Il s'agit du paramètre utilisé pour les calculs de barycentres.

Un appel à `bezier(Ps, r)` renvoie ainsi une liste de $2^r + 1$ points. Dans la pratique, $r = 9$ suffit, puisqu'on obtient environ 500 points (c'est ce nombre de points que nous prenons depuis le début du notebook pour tracer les courbes). Si la courbe de Bézier est vraiment très compliquée, on peut être amenés à prendre une valeur de r plus grande.

[31]:
```
def bezier(Ps, r, t=0.5):
    if r == 0: return [Ps[0], Ps[-1]]
    n = len(Ps) - 1
    Qs1 = [Ps[0]]
    Qs2 = [Ps[-1]]
    for k in range(n):
        Ps = bary(Ps, t)
        Qs1.append(Ps[0])
        Qs2.append(Ps[-1])
    Qs2.reverse()
    Ps1 = bezier(Qs1, r - 1)
    Ps2 = bezier(Qs2, r - 1)
```

```
    return Ps1 + Ps2[1:]
```

[32]:
```
Ps = [(0, 0), (1, 2), (2, 3), (4, 1), (5, 3)]
len(bezier(Ps, 10))
```

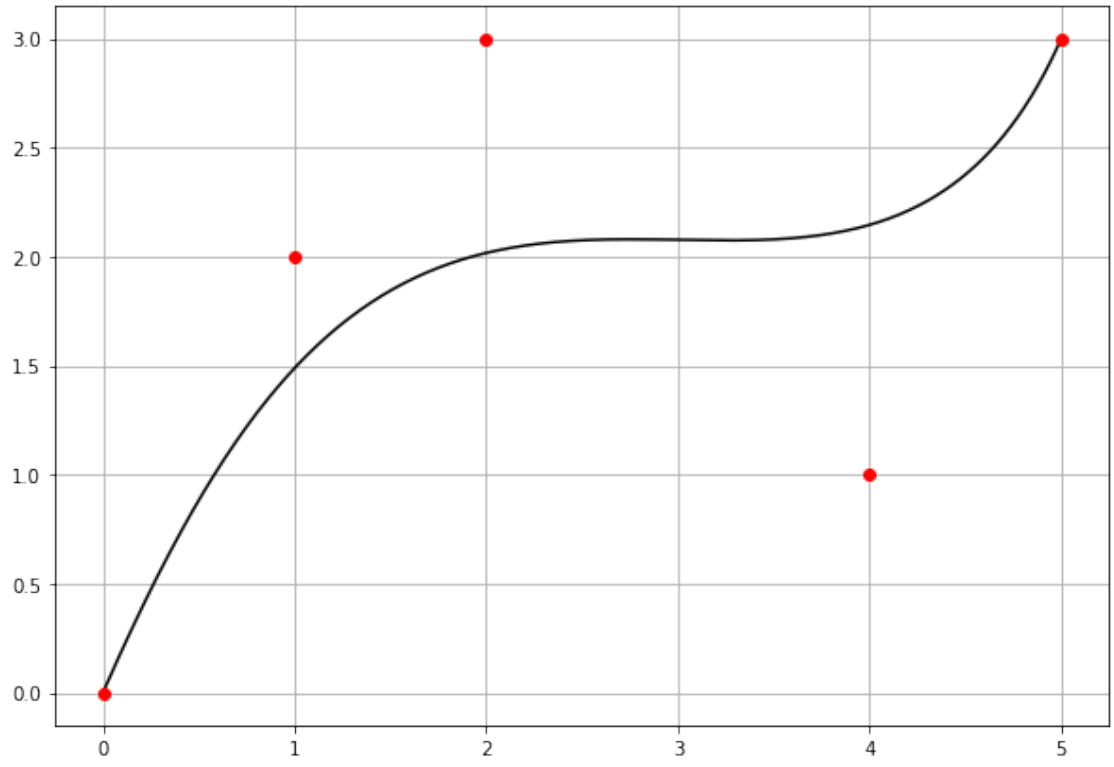
[32]: 1025

La fonction `plot_bezier` affiche la courbe de Bézier avec notre nouvel algorithme. Elle prend en paramètres la liste des points de contrôle et le nombre r d'appels récursifs à effectuer.

[33]:
```
def plot_bezier(Ps, r):
    Qs = bezier(Ps, r)
    xs = [x for (x, y) in Qs]
    ys = [y for (x, y) in Qs]
    plt.plot(xs, ys, 'k')
    for (x, y) in Ps:
        plt.plot([x], [y], 'or')
    plt.grid()
```

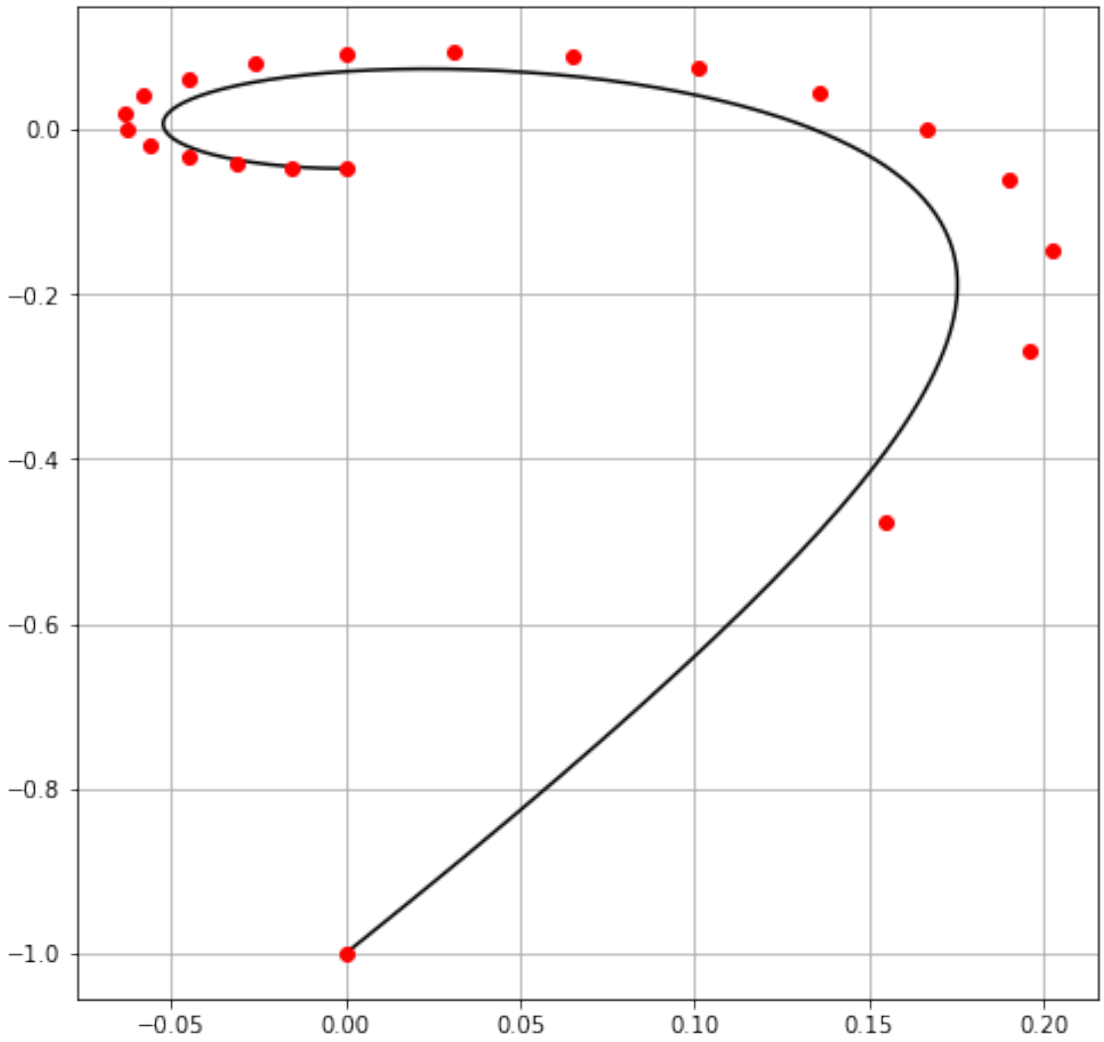
[34]:
```
plt.rcParams['figure.figsize'] = (10, 7)
```

[35]:
```
Ps = [(0, 0), (1, 2), (2, 3), (4, 1), (5, 3)]
plot_bezier(Ps, 9)
```



[36]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

[37]:
```
plot_bezier(cycle(20), 9)
```



1.4 4. Les courbes de Bézier dans le monde « réel »

Une recherche rapide sur Internet permet de trouver facilement un grand nombre d'applications des courbes de Bézier. Voici, en quelques lignes, une porte d'entrée pour les recherches.

1.4.1 4.1 Quelques applications concrètes des courbes de Bézier

Les courbes de Bézier sont utilisées par un grand nombre d'applications logicielles. Pour n'en citer que quelques unes :

- Le logiciel Blender
- Le format graphique SVG
- Le programme de manipulation d'images Gimp
- Les polices de caractères TrueType
- Les polices de caractères utilisées par le système de composition de documents LaTeX : le langage **MetaFont**, qui est un langage utilisé pour créer des polices de caractères vectorielles, utilise des courbes de Bézier.

1.4.2 4.2 B-splines, NURBS

Des généralisations des courbes de Bézier existent. Citons en particulier les B-splines, et une généralisation des B-splines, les NURBS (non uniform rational B-Splines).

1.4.3 4.3 Surfaces de Bézier

Les *surfaces de Bézier* sont une généralisation à deux dimensions des courbes de Bézier. Celles-ci sont en particulier utilisées pour le rendu d'objets 3D. Plutôt que de décomposer l'objet à représenter en triangles, on le décompose en surfaces de Bézier. Cette technique a ses avantages, et aussi quelques inconvénients que nous ne détaillerons pas ici.