La décomposition QR

Marc Lorenzi

14 février 2020

Théorème : Toute matrice inversible à coefficients réels $A \in GL_n(\mathbb{R})$ s'écrit de façon unique sous la forme

$$A = QR$$

où $Q \in \mathcal{O}_n(\mathbb{R})$ est une matrice orthogonale et $R \in \mathcal{T}_n(\mathbb{R})$ est une matrice triangulaire supérieure à coefficients strictement positifs.

Il existe différents algorithmes permettant d'obtenir la décomposition QR : algorithme de Gram-Schmidt, rotations "élémentaires" (matrices de Givens), symétries "élémentaires" (matrices de Householder). J'ai choisi ici d'utiliser des rotations.

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

Entrée [2]:
```
plt.rcParams['figure.figsize'] = (12, 6)
```

1. Quelques fonctions auxiliaires

1.1 Facile

Nous représenterons les matrices en Python par des listes de listes. Si A est une matrice, la liste $A[i]$ est la $i$ème ligne de A. Et $A[i][j]$ est le coefficient ligne i, colonne j de A. Les fonctions `nb_lig` et `nb_col` renvoient le nombre de lignes et le nombre de colonnes de la matrice A.

Avertissement : Dans toutes nos discussions mathématiques, les indices de ligne ou de colonne des matrices commenceront à 0, afin de pouvoir facilement transposer nos calculs en Python.

Entrée [3]:
```
def nb_lig(A): return len(A)
def nb_col(A): return len(A[0])
```

Entrée [4]:
```
A = [[1, 2, 3], [4, 5, 6]]
print(nb_lig(A), nb_col(A))
```

```
2 3
```

La fonction `eye` renvoie la matrice identité d'ordre n.

Entrée [5]:
```
def eye(n):
    A = [n * [0] for i in range(n)]
    for i in range(n): A[i][i] = 1
    return A
```

Entrée [6]:
```
print(eye(4))
```

```
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
```

1.2 Opérations matricielles

La fonction `prodmat` prend en paramètres deux matrices A et B. Elle renvoie leur produit AB. Pour deux matrices $A \in \mathcal{M}_{pq}(\mathbb{R})$ et $B \in \mathcal{M}_{qr}(\mathbb{R})$, cette fonction effectue $2pqr$ opérations. En particulier, si $p = q = r = n$, le nombre d'opérations est $2n^3$. Retenez ce nombre, il va bientôt réapparaître.

Entrée [7]:
```
def prodmat(A, B):
    if nb_col(A) != nb_lig(B):
        raise Exception('Matrices incompatibles')
    p = nb_lig(A)
    q = nb_col(A)
    r = nb_col(B)
    C = [r * [0] for i in range(p)]
    for i in range(p):
        for j in range(r):
            for k in range(q):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Entrée [8]:
```
A = [[1, 2, 3], [4, 5, 6]]
B = [[5, 6], [7, 8], [9,10]]
print(prodmat(A, B))
```

```
[[46, 52], [109, 124]]
```

La fonction `submat` prend en paramètres deux matrices A et B. Elle renvoie leur différence $A - B$. Elle effectue pq opérations sur des flottants si A et B sont de taille $p \times q$.

Entrée [9]:
```
def submat(A, B):
    p = nb_lig(A)
    q = nb_col(A)
    if (p != nb_lig(B)) or (q != nb_col(B)):
        raise Exception('Matrices incompatibles')
    C = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(q):
            C[i][j] = A[i][j] - B[i][j]
    return C
```

Entrée [10]:
```
A = [[1, 2, 3], [4, 5, 6]]
B = [[6, 5, 4], [3, 2, 1]]
print(submat(A, B))
```

```
[[-5, -3, -1], [1, 3, 5]]
```

La fonction `transp` renvoie la transposée A^T de la matrice A.

Entrée [11]:
```
def transp(A):
    p = nb_lig(A)
    q = nb_col(A)
    B = [p * [0] for i in range(q)]
    for i in range(q):
        for j in range(p):
            B[i][j] = A[j][i]
    return B
```

Entrée [12]:
```
A = [[1, 2], [3, 4], [5, 6]]
print(transp(A))
```

```
[[1, 3, 5], [2, 4, 6]]
```

1.3 Affichage

La fonction `prnt` affiche proprement une matrice de flottants. Si le choix de 5 chiffres après la virgule ne vous satisfait pas, changez-le.

Entrée [13]:
```
def prnt(A):
    p = nb_lig(A)
    q = nb_col(A)
    for i in range(p):
        s = ''
        for j in range(q):
            s += '%+10.5f ' % A[i][j]
        print(s)
```

Entrée [14]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
prnt(A)
```

```
+1.00000   +2.00000   +3.00000
+4.00000   +5.00000   +6.00000
+7.00000   +8.00000   +9.00000
```

1.4 Matrices aléatoires

Voici enfin une fonction `randmat` qui nous sera bien utile pour nos tests. Elle renvoie une matrice "aléatoire" ayant p lignes et q colonnes. Chacun des coefficients de la matrice est un réel choisi uniformément dans l'intervalle $[-r, r]$. Le réel r est par défaut égal à 1.

Entrée [15]:
```
def randmat(p, q, r=1):
    A = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(q):
            A[i][j] = random.uniform(-r, r)
    return A
```

Entrée [16]:
```
A = randmat(4, 7, r=3)
prnt(A)
```

```
  -0.72643   -2.50094   -2.45817   -1.33647   +2.70453   -2.78421
+2.92412
  +1.04611   -2.78120   +1.61659   +2.49790   -1.76825   +0.11942
-0.01753
  +1.82519   +0.15449   +1.61930   +1.08622   -0.67155   -0.27654
+1.00220
  -1.74155   -2.96496   +0.65671   +2.23671   -2.61753   +0.40092
+2.84477
```

2. L'algorithme QR

2.1 Quelques rappels sur les matrices orthogonales

Soit $Q \in \mathcal{M}_n(\mathbb{R})$. La matrice Q est dite **orthogonale** lorsque

$$Q^T Q = I_n$$

Cette égalité est équivalente à $QQ^T = I_n$. Cela revient encore à dire que Q est inversible et $Q^{-1} = Q^T$. L'ensemble $O_n(\mathbb{R})$ des matrices orthogonales est un groupe multiplicatif, le **groupe orthogonal d'ordre** n.

Il est facile de voir que les matrices orthogonales ont un déterminant égal à ± 1. Celles dont le déterminant est égal à 1 sont appelées **matrices de rotation**. Nous noterons $SO_n(\mathbb{R})$ l'ensemble des matrices de rotation. L'ensemble $SO_n(\mathbb{R})$ est un sous-groupe de $O_n(\mathbb{R})$, le **groupe spécial-orthogonal d'ordre** n.

2.2 Rotations élémentaires

Parmi les éléments de $SO_n(\mathbb{R})$, certains sont plus "simples" que les autres en ce sens qu'ils possèdent beaucoup de coefficients nuls.

Soit $\theta \in \mathbb{R}$. Notons $c = \cos\theta$ et $s = \sin\theta$. Donnons nous $0 \leq j < i \leq n-1$. Définissons la matrice $Q_{ij} = Q_{ij}(\theta)$ comme suit :

- Pour tout $k \neq i, j$, $(Q_{ij})_{kk} = 1$.
- $(Q_{ij})_{ii} = (Q_{ij})_{jj} = c$.
- $(Q_{ij})_{ij} = s$ et $(Q_{ij})_{ji} = -s$.
- Tous les autres coefficients de Q_{ij} sont nuls.

Les matrices $Q_{ij}(\theta)$ sont des matrices de rotations, que nous qualifierons d'**élémentaires**. On les appelle aussi **rotations de Givens**. Voici $Q_{ij}(\theta)$ ci-dessous : le s se trouve ligne i colonne j, et le $-s$ se trouve ligne j colonne i. Tous les coefficients qui ne sont pas explicités sont nuls.

$$Q_{ij}(\theta) = \begin{pmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c & & & & -s & & \\ & & & & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & & & \\ & & & s & & & & c & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{pmatrix}$$

Exercice : Vérifier que $Q_{ij}(\theta) \in SO_n(\mathbb{R})$.

2.3 Faire tourner pour annuler

Soit $A \in \mathcal{M}_n(\mathbb{R})$. Soient $0 \leq j < i \leq n-1$. Est-il possible de trouver un réel θ tel que $A' = Q_{ij}(\theta)A$ ait son coefficient ligne i, colonne j égal à 0 ? La réponse est oui.

Par souci de légèreté, notons $Q = Q_{ij}(\theta)$. Calculons A'_{kl} pour tous $k, l \in [0, n-1]$. On a

$$A'_{kl} = \sum_{m=0}^{n-1} Q_{km} A_{ml}$$

Plusieurs cas se présentent :

- Si $k \neq i, j$, alors $A'_{kl} = A_{kl}$
- Si $k = i$, $A'_{il} = Q_{ii}A_{il} + Q_{ij}A_{jl} = cA_{il} + sA_{jl}$.
- Si $k = j$, $A'_{jl} = Q_{ji}A_{il} + Q_{jj}A_{jl} = -sA_{il} + cA_{jl}$.

En particulier, $A'_{ij} = cA_{ij} + sA_{jj}$. Si nous voulons annuler ce coefficient, il suffit donc de choisir θ tel que

$$cA_{ij} + sA_{jj} = 0$$

Posons $x = A_{ij}$ et $y = A_{jj}$.

- Si $x = y = 0$, $\theta = 0$ fait l'affaire.
- Sinon, soit $r = \sqrt{x^2 + y^2}$. Il existe un réel φ tel que $\cos\varphi = \frac{y}{r}$ et $\sin\varphi = -\frac{x}{r}$. L'équation qui nous intéresse devient

$$\sin(\varphi - \theta) = 0$$

Ainsi, $\theta = \varphi$ convient. Remarquons que nous n'avons absolument pas besoin de calculer θ. Seuls son sinus et son cosinus nous intéressent.

La fonction `angle` prend en paramètres deux réels x et y. Elle renvoie un couple (c, s) où $c = \cos\theta$ et $s = \sin\theta$, de sorte que $cx - sy = 0$.

Entrée [17]:
```
def angle(x, y):
    if x == 0 and y == 0: return (1, 0)
    else:
        r = math.sqrt(x ** 2 + y ** 2)
        return (y / r, -x / r)
```

Entrée [18]:
```
x, y = 2, 7
c, s = angle(x, y)
print(c, s)
print(c ** 2 + s ** 2)
print(c * x + s * y)
```

```
0.9615239476408232 -0.27472112789737807
1.0
-2.220446049250313e-16
```

Il est maintenant aisé, étant donnée une matrice A, d'annuler un coefficient non diagonal de A en multipliant A à gauche par une matrice de rotation élémentaire. La fonction `rotation` ci-dessous fait le travail. Elle prend en paramètres la matrice A ainsi que deux indices i et j, supposés distincts. Elle prend également en paramètre une matrice Q supposée être une matrice de rotation.

La fonction calcule un réel θ judicieux, puis effectue le produit $Q_{ij}(\theta)A$. Le coefficient ligne i colonne j de A est annulé. Elle effectue également le produit $QQ_{ij}(\theta)^T = QQ_{ij}(-\theta)$. Comment la matrice Q est-elle modifiée ? Notons $Q' = QQ_{ij}(\theta)^T$. Calculons Q'_{kl} pour tous $k, l \in [0, n-1]$. On a

$$Q'_{kl} = \sum_m Q_{km}(Q_{ij}(-\theta))_{ml}$$

Plusieurs cas se présentent :

- Si $l \neq i, j$, alors $Q'_{kl} = Q_{kl}$
- Si $l = i$, $Q'_{ki} = cQ_{ki} + sQ_{kj}$.
- Si $l = j$, $Q'_{kj} = -sQ_{ki} + cQ_{kj}$.

Le calcul de ce produit modifie donc les **colonnes** i et j de la matrice Q. Nous verrons plus loin le pourquoi de cete opération.

Remarquons pour terminer que les matrices A et Q sont modifiées sur place.

Entrée [19]:
```
def rotation(A, i, j, Q):
    n = len(A)
    c, s = angle(A[i][j], A[j][j])
    for l in range(n):
        Ail, Ajl = A[i][l], A[j][l]
        A[i][l] = c * Ail + s * Ajl
        A[j][l] = -s * Ail + c * Ajl
    for k in range(n):
        Qki, Qkj = Q[k][i], Q[k][j]
        Q[k][i] = c * Qki + s * Qkj
        Q[k][j] = -s * Qki + c * Qkj
```

Entrée [20]:
```
A = randmat(4,4)
Q = eye(4)
prnt(A)
```

```
+0.26146   -0.60065   -0.07299   -0.37301
+0.76981   +0.37692   -0.93024   -0.40002
-0.22304   +0.97695   -0.37038   -0.87521
-0.94926   -0.86271   -0.02390   +0.19449
```

Entrée [21]:
```
rotation(A, 1, 0, Q)
prnt(A)
print()
prnt(Q)
```

```
+0.81300   +0.16372   -0.90429   -0.49873
-0.00000   +0.68996   -0.23006   +0.22455
-0.22304   +0.97695   -0.37038   -0.87521
-0.94926   -0.86271   -0.02390   +0.19449

+0.32160   -0.94688   +0.00000   +0.00000
+0.94688   +0.32160   +0.00000   +0.00000
+0.00000   +0.00000   +1.00000   +0.00000
+0.00000   +0.00000   +0.00000   +1.00000
```

Exercice : Réévaluez la cellule ci-dessus en prenant (i, j) successivement égaux à $(2, 0), (3, 0), (2, 1), (3, 1)$ et $(3, 2)$. Regardez comment les coefficients de A au-dessous de la diagonale sont l'un après l'autre mis à 0.

2.4 Itérations

Soit A une matrice $n \times n$. En itérant la fonction `rotation` sur tous les couples (i, j) tels que $0 \leq j < i \leq n-1$, les coefficients de A au-dessous de la diagonale sont successivement annulés.

Nous n'allons pas faire cela au hasard. En effet, il ne s'agit pas qu'une rotation "désannule" un coefficient précédemment annulé ! Nous allons opérer colonne par colonne. On calcule donc $R = Q_{n(n-1)} \dots Q_{31} Q_{21} A$, où les Q_{ij} sont des matrices de rotation élémentaires avec des angles judicieux.

Proposition : La matrice R est triangulaire.

Démonstration : On munit l'ensemble $E = \{(i, j), 0 \leq j < i \leq n-1\}$ de l'ordre lexicographique :

$$(i', j') \preceq (i, j) \Leftrightarrow j' < j \text{ ou } (j' = j \text{ et } i' \leq i)$$

On montre ensuite par induction sur le couple (i, j) que pour tout $(i, j) \in E$, après l'itération numéro (i, j), on a

$$\forall (i', j') \preceq (i, j), A_{i'j'} = 0$$

Récurrence bien posée étant à moitié démontrée, je ne donnerai pas ici les détails de la preuve.

Qu'en est-il de la matrice Q ?

Si, lors de l'appel à `QR`, la matrice Q est égale à I_n, la matrice identité d'ordre n, à la fin de la dernière itération elle est égale à $Q = Q_{21}^T Q_{31}^T \dots Q_{n(n-1)}^T$. On a ainsi $R = Q^T A$. Ainsi, $A = QR$ puisque $Q^{-1} = Q^T$.

Ceci explique pourquoi, dans la fonction `QR`, nous avons modifié à chaque itération la matrice Q en la multipliant **à droite** par Q_{ij}^T. Cela évite, en fin de boucle, de transposer la matrice obtenue : nous avons déjà la bonne matrice.

Un dernier petit détail à régler : si A est inversible, les coefficients diagonaux de R sont non nuls puisque R, tout comme A, est inversible. Cependant, certains coefficients diagonaux peuvent être négatifs. Si jamais $R_{ii} < 0$, il suffit de changer le signe des coefficients de la ligne i de R et de la colonne i de Q. Le produit QR reste égal à A (exercice). De plus, la matrice Q demeure orthogonale (ce n'est en revanche plus forcément une matrice de rotation) et la matrice R reste triangulaire supérieure. La fonction `ajuster_signes` fait cette modification :

Entrée [22]:
```
def ajuster_signes(Q, R):
    n = nb_lig(R)
    for i in range(n):
        if R[i][i] < 0:
            for j in range(n):
                R[i][j] = - R[i][j]
                Q[j][i] = - Q[j][i]
```

Exercice : Soient Q, R deux matrices carrées $n \times n$. Soit $i \in [0, n-1]$. On multiplie la colonne i de Q et la ligne i de R par -1. Montrer que le produit QR n'est pas modifié.

Avant d'écrire notre fonction `QR` améliorons un peu `rotation`. Comme nous procédons colonne par colonne, les coefficients dans chaque colonne de A s'annulent au fur et à mesure. Lors d'un appel à `rotation` ligne i, colonne j, inutile de calculer ce qui se passe à gauche de la colonne j ... car il ne s'y passe rien. Voici donc notre nouvelle fonction `rotation2`.

Cela n'a l'air de rien mais lors de l'exécution du futur algorithme QR, le nombre d'opérations sur la matrice A est divisé ... par 3.

Entrée [23]:
```
def rotation2(A, i, j, Q):
    n = len(A)
    c, s = angle(A[i][j], A[j][j])
    for l in range(j, n):
        Ail, Ajl = A[i][l], A[j][l]
        A[i][l] =  c * Ail + s * Ajl
        A[j][l] = -s * Ail + c * Ajl
    for k in range(n):
        Qki, Qkj = Q[k][i], Q[k][j]
        Q[k][i] =  c * Qki + s * Qkj
        Q[k][j] = -s * Qki + c * Qkj
```

Combien `rotation2` effectue-t-elle d'opérations sur des flottants ? Précisément $6(n-j)+6n$. Retenons ce résultat.

Nous y voilà. L'algorithme QR est maintenant clair.

Entrée [24]:
```
def QR(A):
    n = nb_lig(A)
    R = [A[i].copy() for i in range(n)]
    Q = eye(n)
    for j in range(n):
        for i in range(j + 1, n):
            rotation2(R, i, j, Q)
    ajuster_signes(Q, R)
    return (Q, R)
```

2.5 Complexité

Quelle est la complexité C_n de `QR` en termes d'opérations sur des flottants ? Ne comptons pas `ajuster_signes` qui effectue $O(n^2)$ changements de signes. On a

$$C_n = \sum_{j=0}^{n-1} (n-j-1)(6(n-j)+6n)$$

Posons $k = n - j - 1$.

$$\begin{aligned} C_n &= \sum_{k=0}^{n-1} k(6k+6n+6) \\ &= 6\sum_{k=0}^{n-1} k^2 + 6n\sum_{k=0}^{n-1} k + 6n \\ &= 6\frac{(n-1)n(2n-1)}{6} + 6n\frac{(n-1)n}{2} + 6n \\ &\sim 5n^3 \end{aligned}$$

À titre de comparaison, la complexité d'un produit matriciel est $2n^3$. Notre fonction `QR` effectue environ 2.5 fois plus d'opérations pour décomposer une matrice A que ce qu'il n'en faut pour calculer A^2.

Lançons `QR` sur une matrice de grande taille. Regardons le temps de calcul et comparons avec le temps de calcul de A^2.

Entrée [25]:
```
A = randmat(200, 200)
t1 = time.time()
Q, R = QR(A)
t2 = time.time()
print('Temps: %.1fs' % (t2 - t1))
t1 = time.time()
B = prodmat(A, A)
t2 = time.time()
print('Temps: %.1fs' % (t2 - t1))
```

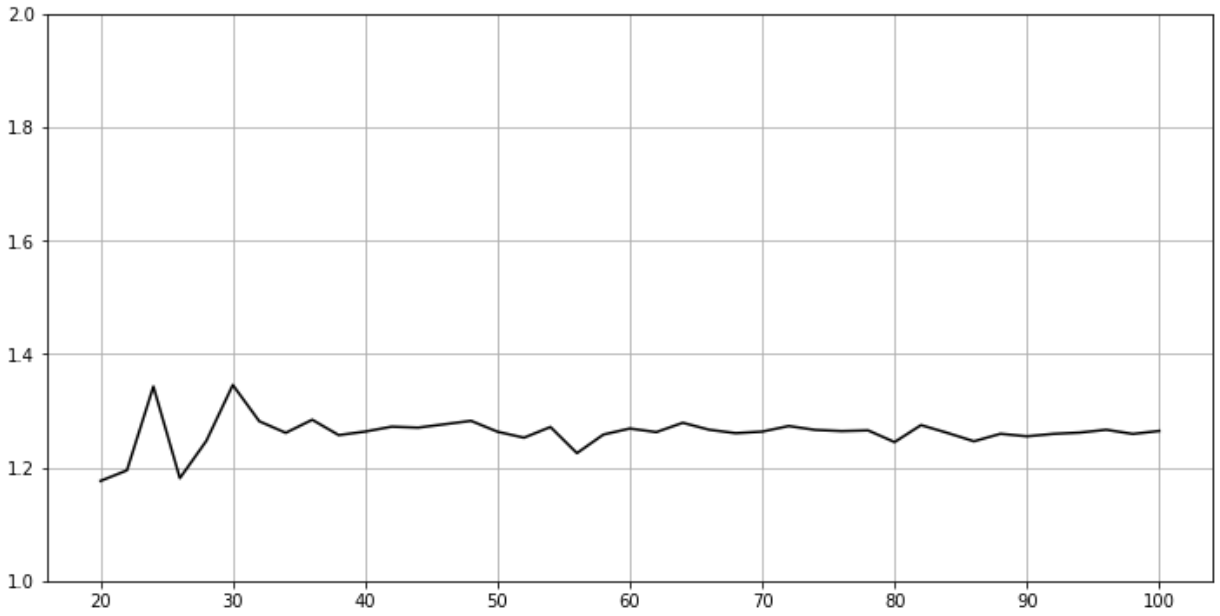
```
Temps: 3.7s
Temps: 2.9s
```

La différence entre les deux temps est plus faible que prévu, ce qui est une bonne nouvelle pour QR. **Trop** bonne nouvelle ? Clairement, compter les opérations sur les flottants n'est pas suffisant. En Python, d'autres opérations sont importantes, je pense en particulier à l'obtention de la valeur d'un coefficient de matrice, ou le changement de la valeur d'un tel coefficient.

Tentons une expérience graphique : affichons $\frac{t'}{t}$ en fonction de n, où t est le temps mis pour calculer le carré d'une matrice $n \times n$ et t' est le temps mis pour calculer la décomposition QR de cette même matrice.

Entrée [26]:
```
def test_temps():
    s = []
    rg = range(20, 101, 2)
    for n in rg:
        A = randmat(n, n)
        t1 = time.time()
        Q, R = QR(A)
        t2 = time.time()
        B = prodmat(A, A)
        t3 = time.time()
        s.append((t2 - t1) / (t3 - t2))
    plt.ylim(1, 2)
    plt.grid()
    plt.plot(rg, s, 'k')
```

Entrée [27]:
```
test_temps()
```



Après quelques égarements, le quotient des deux temps se stabilise autour d'une valeur environ égale à 1.3. Notre fonction `QR` est donc (avec Python) à peine plus coûteuse en temps que la fonction `prodmat`.

2.6 Le théorème

Venons-en au résultat annoncé au début du notebook.

Proposition : Soit $A \in GL_n(\mathbb{R})$ une matrice inversible. Il existe une matrice $Q \in \mathcal{O}_n(\mathbb{R})$ orthogonale et une matrice $R \in \mathcal{T}_n(\mathbb{R})$ triangulaire supérieure dont les coefficients diagonaux sont strictement positifs telles que $A = QR$.

Démonstration : Nous venons de la faire. Nous avons même un algorithme pour calculer Q et R.

Proposition : Les matrices Q et R sont définies de façon unique.

Démonstration : Supposons $A = QR = Q'R'$. On a alors $Q'^{-1}Q = R'R^{-1}$. La matrice $R'' = R'R^{-1}$ est une matrice triangulaire supérieure à coefficients strictement positifs. C'est de plus une matrice orthogonale puisqu'elle est égale à $Q'^{-1}Q$. Il est alors facile de voir que $R'' = I_n$. Ainsi, $R' = R$ et $Q' = Q$.

Exercice : Montrer qu'une matrice à la fois triangulaire et orthogonale est symétrique et n'a que des ± 1 sur sa diagonale.

2.6 Vérifications

Rappelons-le, la fonction `QR` prend une matrice A en paramètre et renvoie un couple (Q, R) de matrices tel que

- $A = QR$
- Q est orthogonale.
- R est triangulaire supérieure
- Les coefficients diagonaux de R sont strictement positifs

Ne rêvons pas ! Bien entendu, les calculs en virgule flottante ne donnent que des résultats approchés. La matrice Q est presque orthogonale, R est quasi-triangulaire, et QR est vaguement égal à A. **Qu'entendons-nous par "presque", "quasiment" et "vaguement" ?**

La fonction `norme2` renvoie la **norme de Frobenius** d'une matrice A :

$$\|A\| = \left(\sum_{i,j} A_{ij}^2 \right)^{1/2}$$

La fonction $A \mapsto \|A\|$ est une norme sur l'espace vectoriel $\mathcal{M}_n(\mathbb{R})$. La quantité $\|A\|$ est donc un bon indicateur de la "petitesse" d'une matrice A et la quantité $\|A - B\|$ est un bon indicateur de la "proximité" de deux matrices A et B.

Entrée [28]:
```
def norme2(A):
    s = 0
    p = nb_lig(A)
    q = nb_col(A)
    for i in range(p):
        for j in range(q):
            s += A[i][j] ** 2
    return math.sqrt(s)
```

La fonction `verif_orthog` prend une matrice carrée Q en paramètre et renvoie $\|Q^TQ - I\|$. Elle devrait renvoyer presque 0 lorsque Q est quasi-orthogonale.

Entrée [29]:
```
def verif_orthog(Q):
    n = nb_lig(Q)
    return norme2(submat(prodmat(Q, transp(Q)), eye(n)))
```

La fonction `verif_triang` prend une matrice R en paramètre et renvoie

$$\Gamma(R) = \left(\sum_{i>j} R_{ij}^2 \right)^{1/2}$$

Cette quantité devrait être quasi-nulle pour une matrice R presque triangulaire supérieure.

Entrée [30]:
```
def verif_triang(R):
    n = nb_col(R)
    s = 0
    for j in range(n):
        for i in range(j + 1, n):
            s += R[i][j] ** 2
    return math.sqrt(s)
```

La fonction `verif_diag_pos` renvoie `True` si tous les coefficients diagonaux de R sont strictement positifs, et `False` sinon. Ici, c'est tout ou rien. Un coefficient presque positif ne nous satisferait point.

Entrée [31]:
```
def verif_diag_pos(R):
    n = nb_col(R)
    for i in range(n):
        if R[i][i] <= 0: return False
    return True
```

La fonction `verif_QR`

- Appelle `verif_orthog`
- Appelle `verif_triang`
- Appelle `verif_diag_pos`
- Calcule $\|QR - A\|$

et affiche les résultats.

Entrée [32]:
```
def verif_QR(A, Q, R):
    print('||Q^TQ-I||: %e' % (verif_orthog(Q)))
    print('G(R)      : %e' % (verif_triang(R)))
    if verif_diag_pos(R): print('Diag pos  : OK')
    else: print('Diag pos : PAS OK')
    print('||QR-A||  : %e' % (norme2(submat(prodmat(Q, R), A))))
```

Allons-y.

Entrée [33]:
```
A = randmat(100, 100)
Q, R = QR(A)
verif_QR(A, Q, R)
```

```
||Q^TQ-I||: 1.640086e-14
G(R)      : 3.288495e-15
Diag pos  : OK
||QR-A||   : 8.984951e-14
```

Pas trop mal. Sachant qu'une matrice 100×100 a 10^4 coefficients, la moyenne quadratique des erreurs est de l'ordre de 10^{-18}, c'est à dire moins que la précision de la machine.

2.7 Un test avec des matrices de mauvaise réputation

La matrice de Hilbert $(H_{ij})_{0 \leq i,j \leq n-1}$ est définie par $H_{ij} = \frac{1}{i+j+1}$. Les matrices de Hilbert sont connues pour jouer des tours à beaucoup d'algorithmes numériques. Elles sont inversibles mais très mal "conditionnées" (je n'entrerai pas dans les détails). Comment se comporte `QR` sur ces matrices ?

Entrée [34]:
```
def hilbert(n):
    H = [[1 / (i + j + 1) for j in range(n)] for i in range(n)]
    return H
```

Entrée [35]:
```
prnt(hilbert(10))
```

```
  +1.00000   +0.50000   +0.33333   +0.25000   +0.20000   +0.16667
+0.14286   +0.12500   +0.11111   +0.10000
  +0.50000   +0.33333   +0.25000   +0.20000   +0.16667   +0.14286
+0.12500   +0.11111   +0.10000   +0.09091
  +0.33333   +0.25000   +0.20000   +0.16667   +0.14286   +0.12500
+0.11111   +0.10000   +0.09091   +0.08333
  +0.25000   +0.20000   +0.16667   +0.14286   +0.12500   +0.11111
+0.10000   +0.09091   +0.08333   +0.07692
  +0.20000   +0.16667   +0.14286   +0.12500   +0.11111   +0.10000
+0.09091   +0.08333   +0.07692   +0.07143
  +0.16667   +0.14286   +0.12500   +0.11111   +0.10000   +0.09091
+0.08333   +0.07692   +0.07143   +0.06667
  +0.14286   +0.12500   +0.11111   +0.10000   +0.09091   +0.08333
+0.07692   +0.07143   +0.06667   +0.06250
  +0.12500   +0.11111   +0.10000   +0.09091   +0.08333   +0.07692
+0.07143   +0.06667   +0.06250   +0.05882
  +0.11111   +0.10000   +0.09091   +0.08333   +0.07692   +0.07143
+0.06667   +0.06250   +0.05882   +0.05556
  +0.10000   +0.09091   +0.08333   +0.07692   +0.07143   +0.06667
+0.06250   +0.05882   +0.05556   +0.05263
```

Entrée [36]:
```
H = hilbert(100)
Q, R = QR(H)
verif_QR(H, Q, R)
```

```
||Q^TQ-I||: 1.701308e-14
G(R)      : 6.973587e-17
Diag pos  : OK
||QR-A||   : 4.451049e-15
```

Nous n'avons plus qu'à féliciter notre fonction `QR` . Rien ne lui résiste :-).

3 Approximation au sens des moindres carrés

3.1 Généralisation de la décomposition QR à des matrices non carrées

Considérons un système linéaire $Ax = b$ de p équations à q inconnues, où $p \geq q$. Supposons que la matrice $A \in \mathcal{M}_{pq}(\mathbb{R})$ est de rang maximal q.

- Lorsque $p = q$ on a un système de Cramer. Ce système possède une unique solution.
- En revanche, lorsque $p > q$, ce système peut fort bien ne pas avoir de solution. Peut-on cependant trouver $x \in \mathbb{R}^q$ tel que Ax soit *le plus près possible* de b ? Plus précisément, est-il possible de minimiser $||Ax - b||$, où $||u||$ dénote la norme euclidienne du vecteur u ?

Complétons la matrice A en une matrice A' de taille $p \times p$ inversible, en rajoutant $p - q$ colonnes à A. Cela est possible car A est de rang q. Ses colonnes sont libres dans $\mathbb{R}^p$ et peuvent donc être complétées en une base de $\mathbb{R}^p$. Écrivons ensuite $A' = QR'$ où $Q \in \mathcal{O}_p(\mathbb{R})$ est orthogonale et $R' \in \mathcal{T}_p(\mathbb{R})$ est triangulaire à coefficients diagonaux strictement positifs.

Tronquons les colonnes de A' et R' à partir de la colonne $q + 1$. A' redevient A et R' se transforme en une matrice R. On a alors $A = QR$ où Q est orthogonale et $R = \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix}$ où la matrice $\hat{R} \in \mathcal{T}_q(\mathbb{R})$ est triangulaire à coefficients diagonaux strictement positifs. On a donc le théorème suivant.

Proposition : Soient $1 \leq q \leq p$ deux entiers. Soit $A \in \mathcal{M}_{pq}(\mathbb{R})$ une matrice que rang q. Il existe une matrice orthogonale $Q \in \mathcal{O}_p(\mathbb{R})$ et une matrice triangulaire supérieure à coefficients strictement positifs $\hat{R} \in \mathcal{T}_q(\mathbb{R})$ telles que

$$A = Q \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix}$$

où 0 désigne la matrice nulle de taille $(p - q) \times q$.

Remarque : Si $p > q$, il n'y a plus unicité : la matrice A peut-être complétée d'une infinité de façons en une matrice inversible.

3.2 L'algorithme

La fonction `QR1` utilise la fonction `QR`. Il suffit de

- Rajouter $p - q$ colonnes à A (on prend ici des coefficients aléatoires) pour obtenir une matrice carrée A_1.
- Appliquer l'algorithme QR à la matrice A_1.
- Tronquer la matrice R.

Entrée [37]:
```
def QR1(A):
    p = nb_lig(A)
    q = nb_col(A)
    A1 = [A[i].copy() for i in range(p)]
    for i in range(p):
        A1[i] = A1[i] + [random.uniform(-1, 1) for k in range(p - q)
    Q, R = QR(A1)
    for i in range(p): R[i] = R[i][:q]
    return (Q, R)
```

Prenons un exemple.

Entrée [38]:
```
A = randmat(8, 6)
prnt(A)
Q, R = QR1(A)
print()
prnt(Q)
print()
prnt(R)
```

```
  -0.90039   -0.62576   -0.26835   -0.60663   -0.25862   -0.31119
  -0.77785   -0.23911   -0.13041   -0.03759   -0.87340   -0.04774
  +0.76498   +0.47252   -0.88142   -0.06976   +0.55907   -0.38362
  +0.73982   +0.86633   -0.61430   -0.68319   +0.15987   +0.65497
  +0.19011   -0.51399   +0.55065   +0.09571   -0.62071   +0.88448
  -0.05605   -0.33956   -0.42263   +0.90059   +0.72124   -0.42298
  -0.46424   -0.73375   +0.44394   -0.36646   -0.69415   -0.11713
  +0.52364   +0.07742   +0.55287   +0.53627   +0.22613   +0.51470

  -0.51327   -0.02500   -0.37471   -0.36313   +0.34205   +0.36649
-0.10170   +0.44945
  -0.44342   +0.28373   -0.04007   +0.18427   -0.77690   +0.02362
+0.16755   +0.23477
  +0.43608   -0.03864   -0.63210   -0.12498   -0.19253   -0.45884
-0.12657   +0.35993
  +0.42174   +0.37719   -0.14370   -0.43234   -0.14317   +0.49724
+0.41993   -0.16799
  +0.10838   -0.64906   +0.05547   -0.23907   -0.46004   +0.36932
-0.38622   -0.09767
  -0.03195   -0.30599   -0.55648   +0.58973   +0.07273   +0.28348
+0.30984   -0.25738
  -0.26464   -0.42923   +0.01185   -0.42388   -0.03664   -0.40013
+0.61404   -0.16558
  +0.29850   -0.27561   +0.35342   +0.21687   +0.05996   +0.17458
+0.38342   +0.69420

  +1.75422   +1.17106   -0.32715   +0.24814   +0.99214   +0.58384
  +0.00000   +0.98743   -0.79895   -0.57870   +0.21517   -0.28013
  +0.00000   +0.00000   +1.21759   +0.06043   -0.60854   +0.73184
  +0.00000   +0.00000   +0.00000   +1.29731   +0.71098   -0.43064
  -0.00000   +0.00000   +0.00000   +0.00000   +0.83655   -0.49178
  +0.00000   +0.00000   +0.00000   -0.00000   +0.00000   +0.73000
  +0.00000   -0.00000   +0.00000   +0.00000   +0.00000   +0.00000
  +0.00000   +0.00000   +0.00000   +0.00000   +0.00000   +0.00000
```

Le concepteur de `verif_QR` a pensé à tout, cette fonction marche encore pour des matrices non carrées.

Entrée [39]:
```
verif_QR(A, Q, R)
```

```
||Q^TQ-I||: 1.309108e-15
G(R)      : 1.271920e-16
Diag pos  : OK
||QR-A||   : 1.750404e-15
```

3.3 Application à l'approximation au sens des moindres carrés

Dans ce paragraphe, nous confondons allègrement matrices à q lignes et une colonne avec des vecteurs de $\mathbb{R}^q$.

Revenons à notre problème. Pour tout $x \in \mathbb{R}^q$, on a

$$\|Ax - b\| = \|QRx - b\| = \|Q(Rx - Q^T b)\|$$

Comme Q est orthogonale, les produits par Q conservent la norme euclidienne. Ainsi,

$$\|Ax - b\| = \|Rx - Q^T b\|$$

Posons $Q^T b = \begin{pmatrix} c \\ d \end{pmatrix}$ où c possède q lignes et d possède $p - q$ lignes. On a alors

$$\|Ax - b\|^2 = \left\| \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} x - \begin{pmatrix} c \\ d \end{pmatrix} \right\|^2 = \|\hat{R}x - c\|^2 + \|d\|^2 \geq \|d\|^2$$

Et on a égalité si et seulement si $\hat{R}x = c$. La matrice $\hat{R}$ étant inversible, il existe un unique $x \in \mathbb{R}^q$ vérifiant l'égalité.

La fonction `moindres_carres` prend en paramètres une matrice $A \in \mathcal{M}_{pq}(\mathbb{R})$ de rang q telle que $1 \leq q \leq p$ et une matrice colonne $b \in \mathcal{M}_{p1}(\mathbb{R})$. Elle renvoie l'unique $x \in \mathcal{M}_{q1}(\mathbb{R})$ minimisant $\|Ax - b\|^2$.

Entrée [40]:
```
def moindres_carres(A, b):
    p = nb_lig(A)
    q = nb_col(A)
    Q, R = QR1(A)
    b1 = prodmat(transp(Q), b)
    R = R[:q]
    b1 = b1[:q]
    x = solve_triang_sup(R, b1)
    return x
```

Il reste à expliquer la ligne `solve_triang_sup(R, b1)` ci-dessus. Elle résout le système $Rx = b_1$. Cela dit, résoudre un système dont la matrice est triangulaire n'est pas bien difficile ...

3.4 Résolution d'un système triangulaire

Soit à résoudre le système $Rx = b$ où $R \in \mathcal{T}_n(\mathbb{R})$ est triangulaire supérieure à coefficients diagonaux non nuls. Ce système est de Cramer et possède donc une unique solution. On a pour tout $i \in [0, n-1]$,

$$\sum_{j=0}^{n-1} R_{ij} x_j = b_i$$

On en déduit donc

$$x_{n-1} = \frac{1}{R_{(n-1)(n-1)}}$$

et pour $i = n-2, n-3, \dots, 0$,

$$R_{ii} x_i + \sum_{j=i+1}^{n-1} R_{ij} x_j = b_i$$

et donc

$$x_i = \frac{1}{R_{ii}} \left(b_i - \sum_{j=i+1}^{n-1} R_{ij} x_j \right)$$

Entrée [41]:
```
def solve_triang_sup(R, b):
    n = nb_lig(R)
    x = [b[i] for i in range(n)]
    for i in range(n - 1, -1, -1):
        for j in range(i + 1, n):
            x[i][0] = x[i][0] - R[i][j] * x[j][0]
        x[i][0] = x[i][0] / R[i][i]
    return x
```

3.5 Quelques tests

Testons tout d'abord la qualité de l'approximation sur la résolution d'un système "aléatoire" de $n+1$ équations à n inconnues.

Entrée [42]:
```
def norme_euclidienne(u):
    s = 0
    for i in range(len(u)):
        s += u[i][0] ** 2
    return math.sqrt(s)
```

Entrée [43]:
```
norme_euclidienne([[1], [2], [3]])
```

Out[43]: 3.7416573867739413

Dans la cellule ci-dessous on minimise $||Ax - b||$ où A possède $n + 1$ lignes et n colonnes. On renvoie $||Ax - b||$.

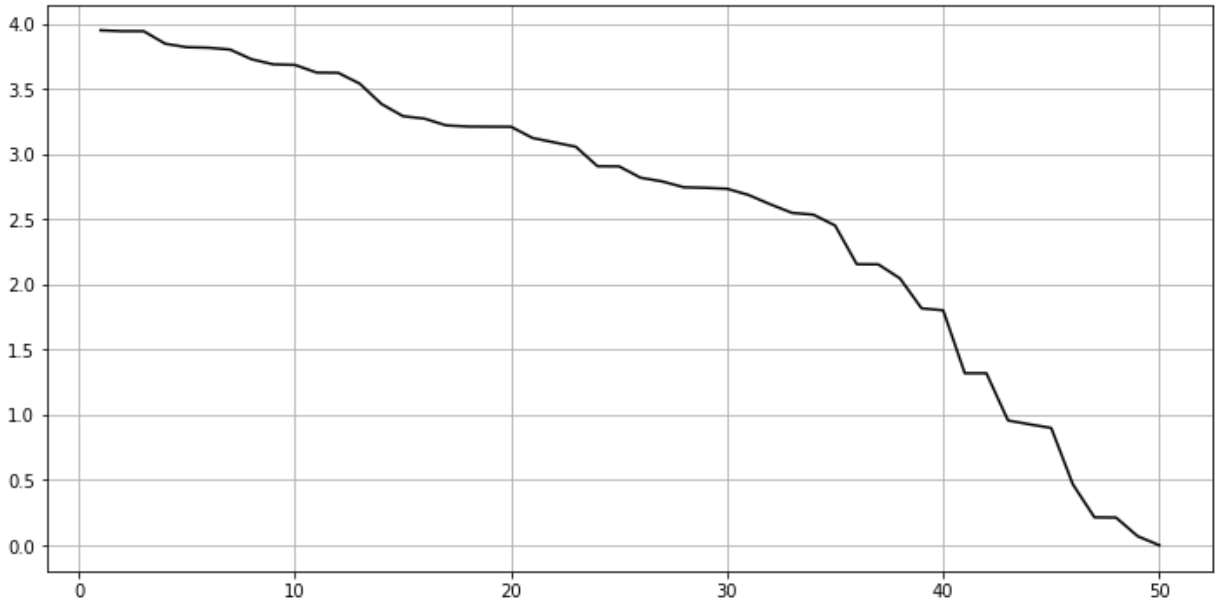
Entrée [44]:
```
n = 50
A = randmat(n + 1, n)
b = randmat(n + 1, 1)
x = moindres_carres(A, b)
E = submat(prodmat(A, x), b)
print(norme_euclidienne(E))
```

```
0.8443447335892869
```

Second test, on se donne une matrice A carrée de taille n et une matrice $b \in \mathcal{M}_{n1}(\mathbb{R})$. Pour $k = 1, \dots, n$, on minimise successivement les quantités $||A_k x - b||$ où $A - k$ est la matrice formée des k premières colonnes de A. Puis on trace le graphe de l'erreur commise en fonction de k. Cette erreur devrait décroître, et devenir nulle lorsque $k = n$.

Entrée [45]:
```
s = []
n = 50
A = randmat(n, n)
b = randmat(n, 1)
for k in range(1, n + 1):
    A1 = [[A[i][j] for j in range(k)] for i in range(n)]
    x = moindres_carres(A1, b)
    t = norme_euclidienne(submat(prodmat(A1, x), b))
    s.append(t)
plt.grid()
plt.plot(range(1, n + 1), s, 'k')
```

Out[45]:
```
[<matplotlib.lines.Line2D at 0x117cec730>]
```



4 Approximation polynomiale au sens des moindres carrés

Passons pour finir à une application intéressante de ce que nous venons d'étudier : l'approximation polynomiale.

4.1 Approximation versus interpolation

Soient $x_0 < x_1 < \ldots < x_{p-1}$ p réels distincts. Soient $y_0, \ldots, y_{p-1}$ p réels. Soit $q \leq p$. Existe-t-il un polynôme $P \in \mathbb{R}_{q-1}[X]$ tel que $P(x_0) = y_0, \ldots, P(x_{p-1}) = y_{p-1}$?

- Si $p = q$, un tel polynôme existe et est unique. Il s'agit du *polynôme d'interpolation de Lagrange*.
- Si, en revanche, $p > q$ l'existence d'un tel polynôme n'est plus assurée. On peut néanmoins chercher un polynôme P tel que la quantité

$$\sum_{k=0}^{p-1} (P(x_k) - y_k)^2$$

soit minimale. Au lieu de faire de **l'interpolation**, nous allons faire de **l'approximation**. En fait, nous savons déjà résoudre ce problème. Posons $P = \sum_{k=0}^{q-1} a_k X^k$. On a pour tout $i \in [0, p-1]$,

$$P(x_i) = \sum_{k=0}^{q-1} a_k x_i^k$$

ce qui peut encore s'écrire matriciellement

$$\begin{pmatrix} P(x_0) \\ \vdots \\ P(x_{p-1}) \end{pmatrix} = A \begin{pmatrix} y_0 \\ \vdots \\ y_{p-1} \end{pmatrix}$$

où

$$A = \begin{pmatrix} x_0^0 & x_0^1 & \ldots & x_0^{q-1} \\ x_1^0 & x_1^1 & \ldots & x_1^{q-1} \\ \vdots & \vdots & & \vdots \\ x_{p-1}^0 & x_{p-1}^1 & \ldots & x_{p-1}^{q-1} \end{pmatrix}$$

La matrice A est une **matrice de Vandermonde** de taille $p \times q$ et de rang q car les x_i sont distincts deux à deux. On peut donc en faire la décomposition $A = QR$ pour obtenir $x = \begin{pmatrix} a_0 \\ \vdots \\ a_{q-1} \end{pmatrix}$ tel que $||Ax - b||$ soit minimal, où $b = \begin{pmatrix} y_0 \\ \vdots \\ y_{p-1} \end{pmatrix}$. Mais précisément,

$$||Ax - b||^2 = \sum_{k=0}^{p-1} (P(x_k) - y_k)^2$$

et les coordonnées de x sont les coefficients d'un polynôme P réalisant le minimum cherché.

Créons tout d'abord une fonction `vandermonde`. Celle-ci prend en paramètres une liste $xs = [x_0, \dots, x_{p-1}]$ de p réels distincts et un entier q. Elle renvoie la matrice de Vandermonde de taille $p \times q$ définie ci-dessus.

Entrée [46]:
```
def vandermonde(s, q):
    p = len(s)
    A = [[s[i] ** j for j in range(q)] for i in range(p)]
    return A
```

Entrée [47]:
```
prnt(vandermonde([1, 2, 3, 4, 5], 3))
```

```
+1.00000   +1.00000   +1.00000
+1.00000   +2.00000   +4.00000
+1.00000   +3.00000   +9.00000
+1.00000   +4.00000  +16.00000
+1.00000   +5.00000  +25.00000
```

La fonction `approximer` ci-dessous se passe de commentaires. Elle prend en paramètres une liste $xs = [x_0, \dots, x_{p-1}]$ de p réels distincts, une liste $b = [y_0, \dots, y_{p-1}]$ de p réels et un entier q. Elle renvoie la liste des coefficients d'un polynôme P de degré inférieur ou égal à $q - 1$ tel que $\sum_{k=0}^{p-1}(P(x_k) - y_k)^2$ soit minimal.

Entrée [48]:
```
def approximer(xs, b, q):
    A = vandermonde(xs, q)
    b = [[x] for x in b]
    P = moindres_carres(A, b)
    P = [x for [x] in P]
    return P
```

4.2 Tests

La fonction `eval_poly` prend en paramètre la liste des coefficients d'un polynôme P et un réel x. Elle renvoie $P(x)$.

Entrée [49]:
```
def eval_poly(P, x):
    y, u = 0, 1
    for a in P:
        y += a * u
        u *= x
    return y
```

Prenons $1 = 1 + 2X + 3X^2$. Que vaut $P(2)$?

Entrée [50]:
```
P = [-1, 7, 2]
eval_poly(P, 2)
```

Out[50]: 21

Prenons une liste de points situés sur une sinusoïde déformée aléatoirement. Cherchons ensuite un polynôme d'approximation et traçons le tout. En noir, les points sur la sinusoïde déformée. En rouge, le polynôme d'approximation.

On a pris sur l'exemple 30 points approximés par un polynôme de degré < 10. Changez à votre guise ces valeurs. Voyez en particulier ce qui se passe si le degré du polynôme est trop petit, s'il est trop grand. Changez aussi la fonction, les perturbations apportées à celle-ci, etc. Bref, testez.

Avertissement : Pour un nombre de points trop élevé, le polynôme d'approximation possède des coefficients extrêmement grands et aussi des coefficients extrêmement petits. En conséquence, les erreurs d'arrondi deviennent prépondérantes dans la fonction `eval_poly` et le résultat renvoyé n'a plus de sens. Par exemple, en prenant $p = q = 100$ ci-dessous on devrait obtenir le polynôme d'interpolation : la courbe rouge *devrait* passer par les points noirs. Je vous laisse tester, je détecte des problèmes dès que p dépasse environ 35 ...

Entrée [51]:
```
def test_approx_poly(p, q):
    a = -3
    b = 3
    d = (b - a) / p
    xs = [a + k * d for k in range(p)]
    f = lambda x: math.sin(x)
    B = [f(x) for x in xs]
    C = [t + 0.2 * random.uniform(-1, 1) for t in B]
    P = approximer(xs, C, q)

    n1 = 10 * p
    d1 = (b - a) / float(n1)
    xs1 = [a + k * d1 for k in range(n1)]
    ys = [eval_poly(P, t) for t in xs1]
    plt.grid()
    plt.ylim(-2, 2)
    plt.plot(xs, C, '.k', markersize=8)
    plt.plot(xs1, ys, 'r')
```

Entrée [52]:
```
test_approx_poly(30, 10)
```