Carres_Magiques2

February 23, 2023

1 Carrés magiques normaux (II)

Marc Lorenzi

22 février 2023

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

[2]:
```
plt.rcParams['figure.figsize'] = (8, 4)
```

Nous supposons que le lecteur a déjà regardé le premier notebook sur les carrés magiques. Le module `utils` que nous venons d'importer contient les fonctions utiles que nous avions définies dans la partie I de ce notebook (taille d'une matrice, somme des coefficients sur les lignes, colonnes et diagonales, etc.).

1.1 1. Une approche énergétique

1.1.1 1.1 Introduction

L'algorithme que nous avons décrit dans le premier notebook fournit, pour tout entier $n \neq 2$, un carré magique normal de taille n. Le « problème » est que, pour une valeur donnée de n, il renvoie toujours le même carré magique : notre algorithme est *déterministe*. Serait-il possible d'écrire un algorithme fournissant des carrés magiques « aléatoires » ? C'est ce que nous allons maintenant voir.

1.1.2 1.2 L'énergie d'un carré normal

Soit A une matrice (pas forcément magique) de taille $n \geq 1$. Posons

$$M_n = \frac{1}{2}n(n^2+1)$$

M_n est la *constante magique* des carrés magiques normaux de taille n. Appelons maintenant **énergie** de A la quantité

$$E(A) = \sum_{i=0}^{n-1} \left| M_n - \sum_{j=0}^{n-1} A_{ij} \right| + \sum_{j=0}^{n-1} \left| M_n - \sum_{i=0}^{n-1} A_{ij} \right| + \left| M_n - \sum_{i=0}^{n-1} A_{ii} \right| + \left| M_n - \sum_{i=0}^{n-1} A_{i(n-1-i)} \right|$$

Proposition. Le carré normal A est magique si et seulement si $E(A) = 0$.

Démonstration. La somme qui définit l'énergie de A est une somme de réels positifs. Elle est nulle si et seulement si chacun de ses termes est nul, c'est à dire si et seulement si

- Pour tout $i \in [|0, n-1|]$, $\sum_{j=0}^{n-1} A_{ij} = M_n$
- Pour tout $j \in [|0, n-1|]$, $\sum_{i=0}^{n-1} A_{ij} = M_n$
- $\sum_{i=0}^{n-1} A_{ii} = M_n$
- $\sum_{i=0}^{n-1} A_{i(n-1-i)} = M_n$

c'est à dire si et seulement si A est magique.

La recherche des carrés magiques normaux devient donc un problème de **minimisation**.

La fonction `energie` renvoie l'énergie de la matrice A.

[3]:
```
def energie(A):
    n = utils.taille(A)
    S = utils.somme_magique(n)
    s = 0
    for i in range(n):
        s += abs(S - utils.somme_ligne(A, i))
        s += abs(S - utils.somme_colonne(A, i))
    s += abs(S - utils.somme_diag1(A))
    s += abs(S - utils.somme_diag2(A))
    return s
```

1.1.3 1.3 Carrés normaux aléatoires

La fonction `random_normal` renvoie un carré normal aléatoire de taille n. Pour cela, elle crée une permutation aléatoire des entiers entre 1 et n^2, puis les place dans une matrice initialement vide.

[4]:
```
def random_normal(n):
    s = list(range(1, n ** 2 + 1))
    random.shuffle(s)
    A = utils.matrice_vide(n)
    k = 0
    for i in range(n):
        for j in range(n):
            A[i][j] = s[k]
            k += 1
    return A
```

[5]:
```
A = random_normal(3)
print(energie(A))
utils.print_carre(A)
```

```
26
+----+----+----+
|   2|   9|   5|
+----+----+----+
|   6|   8|   3|
+----+----+----+
|   1|   7|   4|
+----+----+----+
```

1.1.4 1.4 Quelques statistiques

Pour nous faire une idée des valeurs possibles de l'énergie d'un carré normal, nous allons faire quelques statistiques. Ce paragraphe est sans prétention, aucune théorie, seulement des dessins.

La fonction **répartition** prend en paramètres un entier $n \geq 1$ et un entier N. Elle tire au hasard N carrés normaux de taille n. Elle renvoie une liste de couples (E, k) où E est l'énergie d'un des carrés normaux tirés et k est le nombre de carrés tirés d'énergie E.

[6]:
```
def repartition(n, N):
    d = dict()
    for k in range(N):
        C = random_normal(n)
        E = energie(C)
        if E in d: d[E] += 1
        else: d[E] = 1
    s = [(E,d[E]) for E in d]
    s.sort(key=lambda x:x[0])
    return s
```

Tirons au hasard 10000 carrés normaux de taille 3.

[7]:
```
s = repartition(3, 10000)
print(s)
```

```
[(2, 1), (3, 2), (5, 2), (6, 7), (7, 3), (8, 11), (9, 21), (10, 15), (11, 20),
(12, 30), (13, 42), (14, 81), (15, 109), (16, 149), (17, 203), (18, 289), (19,
325), (20, 368), (21, 383), (22, 599), (23, 589), (24, 749), (25, 824), (26,
826), (27, 902), (28, 942), (29, 746), (30, 823), (31, 518), (32, 301), (33,
103), (34, 17)]
```

Soyons bien conscients que la liste s ci-dessus ne contient peut-être pas toutes les valeurs possibles de l'énergie, elle n'est que le résultat de tirages sur un échantillon de carrés normaux. Par exemple, regardez combien il y a de carrés d'énergie 0. Très souvent, il n'y en a pas (mais une fois de temps en temps, il y en a).

La fonction `stats` prend en paramètre une liste s de couples (E, k). Elle renvoie un couple (m, σ). Si s est une liste qui a été obtenue par la fonction précédente, m est une estimation de l'« énergie moyenne » d'un carré normal de taille n et σ est une évaluation de l'écart-type de l'énergie.

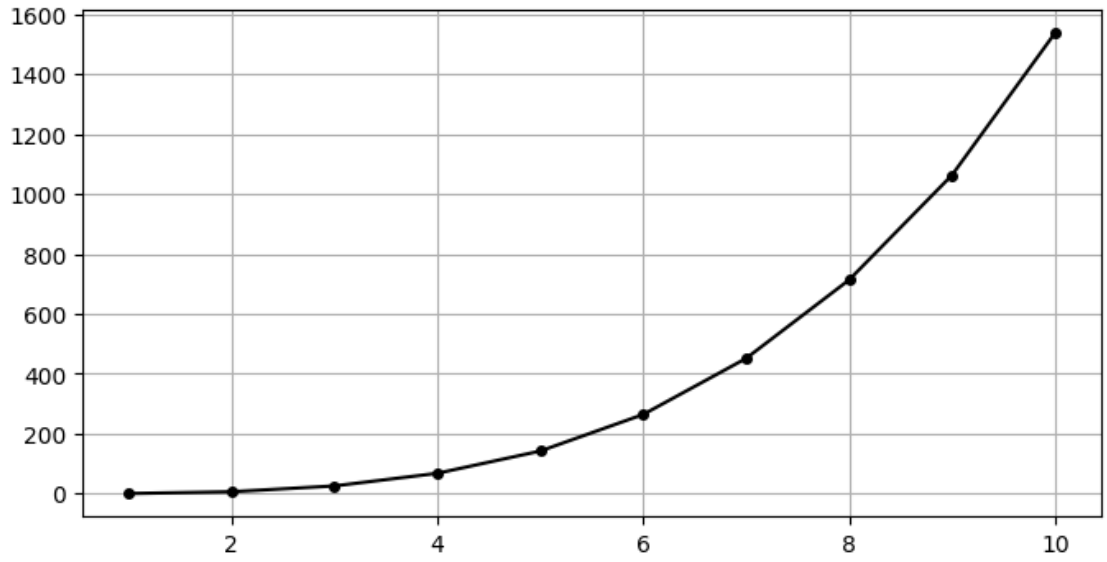
[8]:
```
def stats(s):
    m, m2, N = 0, 0, 0
    for (E, k) in s:
        N += k
        m += k * E
        m2 += k * E ** 2
    m = m / N
    sigma = math.sqrt(m2 / N - m ** 2)
    return (m, sigma)
```

[9]:
```
s = repartition(3, 10000)
print(stats(s))
```

```
(25.1078, 4.626119233223449)
```

Voici, en fonction de n, l'énergie moyenne d'un carré normal de taille n pour $n \in [|1, 10|]$.

[10]:
```
xs = range(1, 11)
ys = [stats(repartition(n, 1000))[0] for n in xs]
plt.plot(xs, ys, 'k')
plt.plot(xs, ys, 'ok', ms=4)
plt.grid()
```



Une moyenne c'est bien, mais avoir une vue complète de la répartition des énergies, c'est encore mieux. La fonction `histo` dessine cette répartition. En abscisses, les énergies possibles. En

ordonnées, les probabilités de chacune des énergies. On trace aussi en rouge le graphe de la fonction de densité d'une loi normale de même moyenne et de même écart-type que pour les carrés normaux.

[11]:
```
def gauss(x, m, sigma):
    return math.exp(-(x - m) ** 2 / (2 * sigma ** 2)) / (sigma * math.sqrt(2 *␣
 ↪math.pi))
```

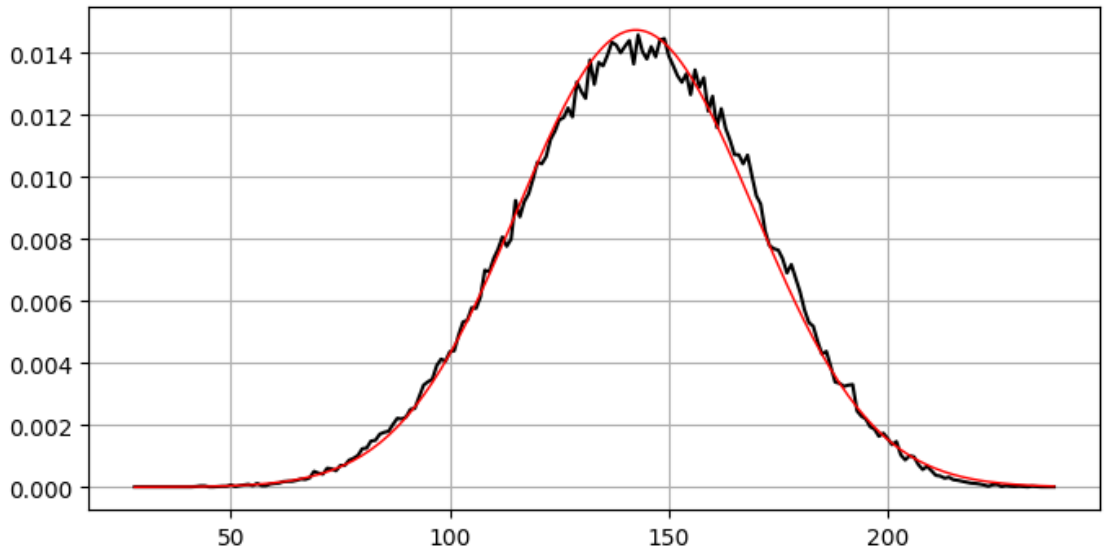
[12]:
```
def histo(n, N):
    s = repartition(n, N)
    xs = [z[0] for z in s]
    ys = [z[1] / N for z in s]
    plt.plot(xs, ys, 'k')
    m, sigma = stats(s)
    zs = [gauss(x, m, sigma) for x in xs]
    plt.plot(xs, zs, 'r', lw=1)
    plt.grid()
```

Voici la répartition des énergies des carrés normaux de taille 5. Rappelons l'énergie moyenne d'un tel carré.

[13]:
```
stats(repartition(5, 10000))
```

[13]:
```
(142.6669, 27.060165269081438)
```

[14]:
```
histo(5, 100000)
```



Clairement, une étude plus approfondie serait intéressante. Nous ne la ferons pas. Avant d'en venir à ce qui nous intéresse, regardons d'un peu plus près les carrés normaux de taille 3.

1.2 2. Les carrés normaux de taille 3

Il existe $9! = 362880$ carrés normaux de taille 3. Ce nombre n'est pas très grand : au lieu de faire des statistiques, nous pouvons donc calculer l'énergie de **tous** les carrés.

Écrivons tout d'abord un générateur qui énumère toutes les permutations d'une liste s.

[15]:
```
def permutations(s):
    if s == []: yield []
    else:
        for a in s:
            for p in permutations([x for x in s if x != a]):
                yield([a] + p)
```

[16]:
```
for s in permutations([1, 2, 3]): print(s)
```

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

La fonction `faire_carre3` prend en paramètre une permutations s de la liste $[1, \ldots, 9]$ et renvoie un carré de taille 3.

[17]:
```
def faire_carre3(s):
    return[[s[0], s[1], s[2]], [s[3], s[4], s[5]], [s[6], s[7], s[8]]]
```

La fonction `tous_carres3` renvoie la liste des carrés normaux de taille 3.

[18]:
```
def tous_carres3():
    cs = []
    for s in permutations(list(range(1, 10))):
        cs.append(faire_carre3(s))
    return cs
```

[19]:
```
Cs = tous_carres3()
```

[20]:
```
len(Cs)
```

[20]:
```
362880
```

La fonction `energies3` renvoie la liste triée des couples (E, n) où E est l'énergie d'un carré normal de taille 3 et n est le nombre de carrés ayant cette énergie.

[21]:
```
def energies3(Cs):
    d = dict()
    for C in Cs:
        E = energie(C)
```

```
        if E in d: d[E] += 1
        else: d[E] = 1
    Es = [(E, d[E]) for E in d]
    Es.sort(key= lambda z:z[0])
    return Es
```

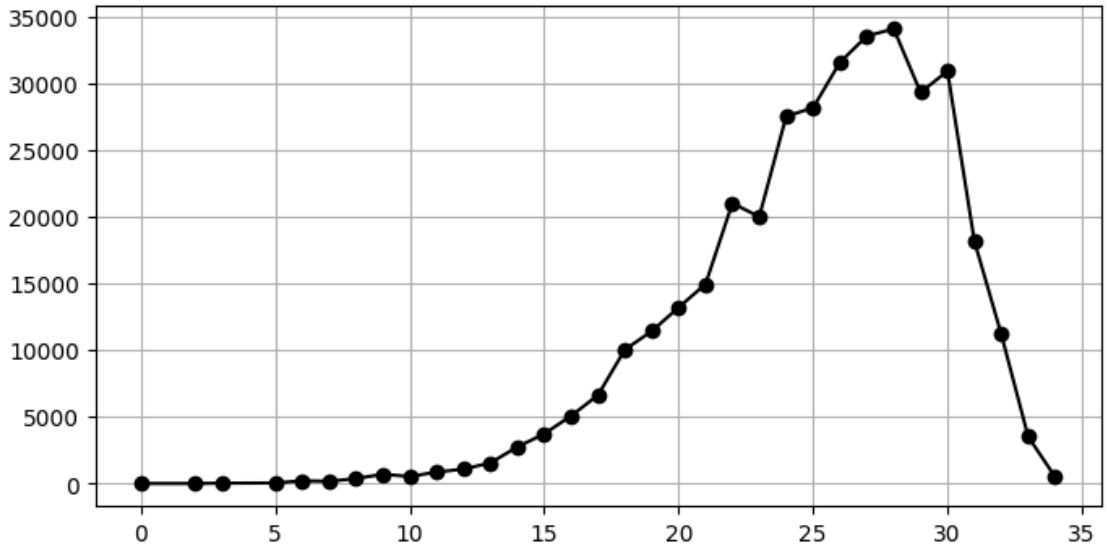
[22]:
```
Es = energies3(Cs)
print(Es)
```

```
[(0, 8), (2, 8), (3, 32), (5, 48), (6, 216), (7, 176), (8, 376), (9, 704), (10,
536), (11, 864), (12, 1088), (13, 1552), (14, 2744), (15, 3728), (16, 5072),
(17, 6624), (18, 10032), (19, 11424), (20, 13200), (21, 14944), (22, 21008),
(23, 20000), (24, 27520), (25, 28160), (26, 31568), (27, 33536), (28, 34080),
(29, 29312), (30, 30912), (31, 18176), (32, 11200), (33, 3520), (34, 512)]
```

Remarquons les 8 carrés normaux d'énergie 0, c'est à dire les carrés magiques normaux.

Traçons enfin le nombre de carrés normaux en fonction de l'énergie.

[23]:
```
xs = [z[0] for z in Es]
ys = [z[1] for z in Es]
plt.plot(xs, ys, 'k')
plt.plot(xs, ys, 'ok')
plt.grid()
```



Venons-en maintenant à ce qui nous intéresse : du code Python capable de trouver des carrés magiques « aléatoires ».

1.3 3. Le recuit simulé

1.3.1 3.1 Voisins d'une matrice

Définition. Soit A une matrice. Un *voisin* de A est une matrice obtenue à partir de A en échangeant deux de ses coefficients.

Si A est une matrice de taille $n \geq 1$, le nombre de voisins de A est le nombre de paires de coefficients de A. Comme A possède n^2 coefficients, ce nombre est donc

$$\binom{n^2}{2} = \frac{1}{2}n^2(n^2-1) \sim \frac{1}{2}n^4$$

Ce nombre augmente donc très vite lorsque n augmente.

[24]:
```
def nombre_voisins(n):
    n2 = n ** 2
    return n2 * (n2 - 1) // 2
```

[25]:
```
for n in range(1, 21):
    print(n, nombre_voisins(n))
```

```
1 0
2 6
3 36
4 120
5 300
6 630
7 1176
8 2016
9 3240
10 4950
11 7260
12 10296
13 14196
14 19110
15 25200
16 32640
17 41616
18 52326
19 64980
20 79800
```

Une matrice de taille 20 possède ainsi 79800 voisins. Un algorithme qui aurait besoin d'examiner tous les voisins d'une matrice A aurait une complexité effroyable. Écrivons donc une fonction qui renvoie un voisin **aléatoire** de A. En réalité nous n'allons pas prendre un voisin tout à fait aléatoire …

La fonction `voisin_aleatoire` prend en paramètre une matrice A de taille n. Elle tire au hasard un entier k entre 1 et $n^2 - 1$. Elle renvoie le quadruplet $q = (i_1, j_1, i_2, j_2)$ d'entiers entre 0 et

$n-1$ tel que $A_{i_1 j_1} = k$ et $A_{i_2 j_2} = k+1$. L'idée est que ce quadruplet fournit, en cas de besoin, la matrice obtenue à partir de A en échangeant $A_{i_1 j_1}$ et $A_{i_2 j_2}$. On échange donc virtuellement deux coefficients de A qui sont des entiers successifs, et donc on obtient une matrice dont l'énergie n'est pas très éloignée de celle de A.

Pour obtenir une fonction de complexité $O(1)$, on fait également passer en paramètre à la fonction `voisin_aleatoire` un **dictionnaire** d : celui-ci est une liste de n^2 couples (i, j). Pour $k \in [|0, n^2 - 1|]$, $d[k]$ est le couple (i, j) tel que $A_{ij} = k$.

La fonction `creer_dictionnaire`, de complexité $O(n^2)$, crée le dictionnaire en question. Elle ne sera appelée qu'une seule fois, à l'initialisation de l'algorithme.

[26]:
```
def creer_dictionnaire(A):
    n = utils.taille(A)
    d = (n ** 2) * [None]
    for i in range(n):
        for j in range(n):
            d[A[i][j] - 1] = (i, j)
    return d
```

[27]:
```
A = random_normal(5)
utils.print_carre(A)
```

```
+----+----+----+----+----+
|  13|  18|   3|  12|  25|
+----+----+----+----+----+
|  23|   4|   1|  17|  22|
+----+----+----+----+----+
|  24|  15|   7|  16|   8|
+----+----+----+----+----+
|   6|  11|  19|  10|  20|
+----+----+----+----+----+
|   2|   9|  14|   5|  21|
+----+----+----+----+----+
```

[28]:
```
d = creer_dictionnaire(A)
print(d)
```

```
[(1, 2), (4, 0), (0, 2), (1, 1), (4, 3), (3, 0), (2, 2), (2, 4), (4, 1), (3, 3),
(3, 1), (0, 3), (0, 0), (4, 2), (2, 1), (2, 3), (1, 3), (0, 1), (3, 2), (3, 4),
(4, 4), (1, 4), (1, 0), (2, 0), (0, 4)]
```

Voici la fonction `voisin_aleatoire`.

[29]:
```
def voisin_aleatoire(A, d):
    n = utils.taille(A)
    k = random.randint(1, n ** 2 - 1)
    i1, j1 = d[k - 1]
    i2, j2 = d[k]
```

```
    return (i1, j1, i2, j2)
```

[30]:
```
A = random_normal(5)
utils.print_carre(A)
d = creer_dictionnaire(A)
```

```
+----+----+----+----+----+
|  13|  18|  21|   1|   5|
+----+----+----+----+----+
|  16|   3|  12|   8|  10|
+----+----+----+----+----+
|   4|  22|  15|   2|  20|
+----+----+----+----+----+
|  19|  24|   7|   6|  14|
+----+----+----+----+----+
|  25|  11|   9|  23|  17|
+----+----+----+----+----+
```

[31]:
```
print(voisin_aleatoire(A, d))
```

```
(1, 3, 4, 2)
```

1.3.2 3.2 Le principe du recuit simulé

L'algorithme que nous allons mettre en place s'appelle *l'algorithme du recuit simulé*. L'idée est que l'énergie $E(A)$ d'un carré normal A de taille n est une fonction de n^2 variables. Il s'agit d'en trouver le minimum global, problème ô combien délicat ! Imaginons une surface, qui représente l'ensemble des énergies possibles de tous les carrés normaux. Il s'agit, en partant d'un point A de la surface, de se déplacer de proche en proche vers le point de la surface de plus faible altitude. L'approche naïve est de se déplacer vers un voisin de A d'altitude plus faible, et de recommencer tant que c'est possible. En fait, par ce procédé on arrive la plupart du temps à un minimum **local** (une cuvette, en quelque sorte) de l'énergie. Il faut donc laisser une chance à l'algorithme de sortir des cuvettes. C'est ce qui explique la possibilité, dans la fonction `recuit_simule` que nous allons écrire, de choisir un carré voisin de A quoique d'énergie plus élevée. Pour cela,

- On choisit un voisin aléatoire B du carré A.
- Si l'énergie de B est strictement inférieure à celle de A, on renvoie B.
- Sinon, soit p un réel aléatoire entre 0 et 1. Si p est inférieur à une probabilité qui est d'autant plus faible que l'énergie de B est grande, on renvoie B. Sinon, on recommence au début.

Remarquons que le calcul de l'énergie d'un carré de taille n nécessite n^2 opérations. Comme le recuit simulé va procéder à un grand nombre d'itérations, il n'est pas efficace de calculer à chaque fois l'énergie des carrés mis en jeu. Il est plus efficace de calculer la **différence** entre l'énergie d'un carré A et l'énergie du voisin B qui nous interesse. Pour déterminer cette différence, il suffit de posséder toutes les sommes de lignes, colonnes et diagonales relatives à A et de savoir quels sont les coefficients qui sont échangés pour passer de A à B.

1.3.3 3.3 La fonction de recuit

Commençons par écrire quelques fonctions auxiliaires.

La fonction `calcul_sommes` renvoie la liste de toutes les sommes de lignes, colonnes, diagonales relatives au carré A de taille n. Cette fonction a une complexité en $O(n^2)$, mais nous ne l'appellerons qu'une seule fois, pour amorcer l'algorithme du recuit simulé.

[32]:
```
def calcul_sommes(A):
    n = utils.taille(A)
    sommes = [utils.somme_ligne(A, i) for i in range(n)]
    sommes += [utils.somme_colonne(A, j) for j in range(n)]
    sommes += [utils.somme_diag1(A), utils.somme_diag2(A)]
    return sommes
```

[33]:
```
A = random_normal(4)
utils.print_carre(A)
print(calcul_sommes(A))
```

```
+----+----+----+----+
|   5|   1|   2|   7|
+----+----+----+----+
|   9|  10|   8|  14|
+----+----+----+----+
|  13|  15|   3|  12|
+----+----+----+----+
|  16|   4|  11|   6|
+----+----+----+----+
[15, 41, 43, 37, 43, 30, 24, 39, 24, 46]
```

La fonction `modif_sommes` prend en paramètres

- Un carré normal A de taille n.
- Un quadruplet $q = (i_1, j_1, i_2, j_2)$ d'entiers entre 0 et $n-1$.
- Une liste `sommes` qui contient toutes les sommes relatives au carré A.

Elle renvoie la liste des sommes relatives au carré obtenu à partir de A en échangeant les coefficients $A_{i_1 j_1}$ et $A_{i_2 j_2}$.

Cette fonction a une complexité en $O(1)$, car il y a en fait au plus 6 sommes à modifier : 2 sommes de lignes, 2 sommes de colonnes, et 0, 1 ou 2 sommes de diagonales.

[34]:
```
def modif_sommes(A, q, sommes):
    i1, j1, i2, j2 = q
    n = utils.taille(A)
    sommes1 = sommes.copy()
    sommes1[i1] = sommes1[i1] - A[i1][j1] + A[i2][j2]
    sommes1[i2] = sommes1[i2] - A[i2][j2] + A[i1][j1]
    sommes1[n + j1] = sommes1[n + j1] - A[i1][j1] + A[i2][j2]
    sommes1[n + j2] = sommes1[n + j2] - A[i2][j2] + A[i1][j1]
```

```
    if j1 == i1: sommes1[2 * n] = sommes1[2 * n] - A[i1][i1] + A[i2][j2]
    if j2 == i2: sommes1[2 * n] = sommes1[2 * n] - A[i2][i2] + A[i1][j1]
    if j1 == n - 1 - i1: sommes1[2 * n + 1] = sommes1[2 * n + 1] - A[i1][n - 1 -␣
 ↪i1] + A[i2][j2]
    if j2 == n - 1 - i2: sommes1[2 * n + 1] = sommes1[2 * n + 1] - A[i2][n - 1 -␣
 ↪i2] + A[i1][j1]
    return sommes1
```

[35]:
```
A = random_normal(4)
d = creer_dictionnaire(A)
q = voisin_aleatoire(A, d)
sommes = calcul_sommes(A)
sommes1 = modif_sommes(A, q, sommes)
utils.print_carre(A)
print(q)
print(sommes)
print(sommes1)
```

```
+----+----+----+----+
|   6|   9|   2|   3|
+----+----+----+----+
|  14|   4|  15|  10|
+----+----+----+----+
|   7|   5|  12|  13|
+----+----+----+----+
|   1|   8|  16|  11|
+----+----+----+----+
(0, 0, 2, 0)
[20, 43, 37, 36, 28, 26, 45, 37, 33, 24]
[21, 43, 36, 36, 28, 26, 45, 37, 34, 24]
```

La fonction `diff_energies` prend en paramètres

- Un entier $n \geq 1$.
- Un quadruplet $q = (i_1, j_1, i_2, j_2)$ d'entiers entre 0 et $n-1$.
- Deux listes de sommes `sommes1` et `sommes2`.

Elle renvoie $E_2 - E_1$ où E_1 est l'énergie du carré dont les sommes sont celles de la liste `sommes1` et E_2 est l'énergie du carré dont les sommes sont celles de la liste `sommes2`.

[36]:
```
def diff_energie(n, q, sommes1, sommes2):
    i1, j1, i2, j2 = q
    M = utils.somme_magique(n)
    DE = 0
    DE = DE - abs(M - sommes1[i1]) + abs(M - sommes2[i1])
    DE = DE - abs(M - sommes1[i2]) + abs(M - sommes2[i2])
    DE = DE - abs(M - sommes1[n + j1]) + abs(M - sommes2[n + j1])
    DE = DE - abs(M - sommes1[n + j2]) + abs(M - sommes2[n + j2])
    if j1 == i1 or j2 == i2:
```

```
        DE = DE - abs(M - sommes1[2 * n]) + abs(M - sommes2[2 * n])
    if j1 == n - 1 - i1 or j2 == n - 1 - i2:
        DE = DE - abs(M - sommes1[2 * n + 1]) + abs(M - sommes2[2 * n + 1])
    return DE
```

La fonction `echanger_coefs` échange « physiquement » deux coefficients du carré A. Elle prend également en paramètre le dictionnaire d associé à A dont nous avons parlé plus haut.

[37]:
```
def echanger_coefs(A, q, d):
    i1, j1, i2, j2 = q
    k1 = A[i1][j1]
    k2 = A[i2][j2]
    A[i1][j1], A[i2][j2] = A[i2][j2], A[i1][j1]
    d[k1 - 1] = (i2, j2)
    d[k2 - 1] = (i1, j1)
```

Voici enfin la fonction de recuit simulé. Elle prend en paramètres

- Un carré normal A.
- Le dictionnaire d associé à A.
- Une liste `sommes1` qui est la liste des sommes relatives à A.
- Une entier E qui est l'énergie de A.

Elle renvoie

- La liste des sommes relatives au voisin de A obtenu à l'issue de la boucle.
- L'énergie de ce voisin.
- Un entier k qui est le nombre d'itérations effectuées.

La fonction de recuit prend également un paramètre optionnel β, qui est égal par défaut à 2. Il est possible que d'autres valeurs fournissent de meilleurs résultats …

[38]:
```
def recuit_simule(A, d, sommes1, E, beta=2):
    n = utils.taille(A)
    k = 0
    while True:
        k += 1
        q  = voisin_aleatoire(A, d)
        sommes2 = modif_sommes(A, q, sommes1)
        DE = diff_energie(n, q, sommes1, sommes2)
        p = random.uniform(0, 1)
        if p <= math.exp(-beta * DE): break
    echanger_coefs(A, q, d)
    return (sommes2, E + DE, k)
```

1.3.4 3.4 La fonction de minimisation

Voici la fonction de minimisation. Elle prend en paramètre un carré normal A. Elle appelle `recuit_simule` tant que l'énergie obtenue est non nulle. Enfin, elle renvoie le nombre total

d"itérations effectuées. À chaque itération, la matrice A est physiquement modifiée. À la sortie de la fonction `minimiser`, A est donc un carré magique normal.

La fonction `minimiser` prend également en paramètre optionnel un booléen `dbg`. S'il vaut `True`, la fonction affiche, une fois de temps en temps, le nombre d'itérations et l'énergie de la matrice en cours.

Remarquons que rien ne garantit la terminaison de cette fonction !!!

[39]:
```
def minimiser(A, beta=2, dbg=False):
    k = 0
    sommes = calcul_sommes(A)
    E = energie(A)
    d = creer_dictionnaire(A)
    while E > 0:
        (sommes, E, c) = recuit_simule(A, d, sommes, E, beta)
        k += c
        if dbg and random.uniform(0, 1) < 1e-5: print('(%d,%d)'% (k,␣
 ↪energie(A)), end=',')
    return k
```

1.4 4. Tests

1.4.1 4.1 Une fonction de test

La fonction `test_recuit` illustre tout ce qui précède. Elle appelle `minimiser`. Une fois de temps en temps, elle affiche le nombre d'itérations et l'énergie du carré en cours. Elle affiche en fin de calcul (si le calcul termine !) le nombre d'itérations effectuées et un carré magique normal de taille n.

[40]:
```
def test_recuit(n, beta=2):
    print('Somme magique : ', utils.somme_magique(n), '\n')
    A = random_normal(n)
    print('Carré initial. Énergie = ', energie(A))
    #print_carre(A)
    print('Énergies successives : ', end='')
    k = minimiser(A, beta, dbg=True)
    print('\nNombre d\'itérations : ', k)
    print('Carré final. Énergie = ', energie(A))
    print('Sommes des lignes : ', [utils.somme_ligne(A, i) for i in range(n)])
    print('Sommes des colonnes : ', [utils.somme_colonne(A, i) for i in␣
 ↪range(n)])
    print('Sommes des diagonales : ', utils.somme_diag1(A), ',', utils.
 ↪somme_diag2(A))
    utils.print_carre(A)
```

1.4.2 4.2 Testons ...

Commençons par tout petit.

[41]:
```
test_recuit(3)
```

```
Somme magique :  15

Carré initial. Énergie =  19
Énergies successives :
Nombre d'itérations :  14
Carré final. Énergie =  0
Sommes des lignes :  [15, 15, 15]
Sommes des colonnes :  [15, 15, 15]
Sommes des diagonales :  15 , 15
+----+----+----+
|   2|   9|   4|
+----+----+----+
|   7|   5|   3|
+----+----+----+
|   6|   1|   8|
+----+----+----+
```

Tentons $n = 4$. Il existe 20922789888000 carrés normaux de taille 4, et seulement 7040 sont magiques.

[42]:
```
test_recuit(4)
```

```
Somme magique :  34

Carré initial. Énergie =  69
Énergies successives :
Nombre d'itérations :  3665
Carré final. Énergie =  0
Sommes des lignes :  [34, 34, 34, 34]
Sommes des colonnes :  [34, 34, 34, 34]
Sommes des diagonales :  34 , 34
+----+----+----+----+
|   1|  12|   7|  14|
+----+----+----+----+
|  13|   8|  11|   2|
+----+----+----+----+
|  16|   5|  10|   3|
+----+----+----+----+
|   4|   9|   6|  15|
+----+----+----+----+
```

Essayons avec $n = 8$. La probabilité qu'un carré normal de taille 8 soit magique est environ 3.3×10^{-34}.

[43]:
```
test_recuit(8)
```

```
Somme magique :  260

Carré initial. Énergie =  774
Énergies successives :
Nombre d'itérations :  4832
Carré final. Énergie =  0
Sommes des lignes :  [260, 260, 260, 260, 260, 260, 260, 260]
Sommes des colonnes :  [260, 260, 260, 260, 260, 260, 260, 260]
Sommes des diagonales :  260 , 260
+----+----+----+----+----+----+----+----+
|  50|  56|  23|  18|  32|   5|  43|  33|
+----+----+----+----+----+----+----+----+
|  52|  13|  64|  62|   6|   7|  22|  34|
+----+----+----+----+----+----+----+----+
|  26|  25|  24|  10|  53|  57|  37|  28|
+----+----+----+----+----+----+----+----+
|  29|  58|   9|  38|  54|  17|   4|  51|
+----+----+----+----+----+----+----+----+
|  12|  63|  36|  30|  41|   8|  39|  31|
+----+----+----+----+----+----+----+----+
|  55|  11|  48|  14|   3|  47|  21|  61|
+----+----+----+----+----+----+----+----+
|  35|  15|  16|  42|  27|  60|  45|  20|
+----+----+----+----+----+----+----+----+
|   1|  19|  40|  46|  44|  59|  49|   2|
+----+----+----+----+----+----+----+----+
```

Avec de grandes valeurs de n, la minimisation peut prendre *beaucoup de temps* ... ou pas si on a de la chance ! On rend en général les calcul un peu moins longs en augmentant, pour les grandes valeurs de n, la valeur du paramètre β du recuit simulé. Par exemple, $\beta = 3$ ou 4 permet d'obtenir très rapidement (sauf pas de chance) un carré magique de taille $n = 20$.

[44]:
```
test_recuit(20, 3)
```

```
Somme magique :  4010

Carré initial. Énergie =  16921
Énergies successives :
Nombre d'itérations :  325983
Carré final. Énergie =  0
Sommes des lignes :  [4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010,
4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010]
Sommes des colonnes :  [4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010,
4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010, 4010]
Sommes des diagonales :  4010 , 4010
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

| 383| 240| 100| 303| 52| 59| 82| 235| 396| 373| 219| 74| 73| 187| 371|
39| 127| 175| 387| 135|

| 206| 113| 80| 242| 285| 385| 291| 257| 132| 53| 54| 376| 189| 150| 194|
283| 343| 92| 171| 114|

| 79| 399| 117| 138| 182| 195| 78| 106| 400| 170| 225| 172| 340| 116| 265|
156| 76| 274| 157| 365|

| 270| 308| 301| 68| 325| 266| 134| 122| 330| 32| 63| 215| 146| 183| 232|
186| 43| 388| 221| 177|

| 125| 368| 335| 89| 344| 133| 384| 259| 355| 97| 273| 228| 306| 18| 309|
205| 44| 75| 60| 3|

| 297| 341| 296| 103| 141| 16| 98| 280| 251| 245| 201| 95| 70| 121| 264|
258| 374| 57| 226| 276|

| 26| 246| 392| 337| 119| 197| 362| 317| 38| 20| 180| 87| 64| 269| 142|
110| 153| 313| 334| 304|

| 299| 1| 72| 320| 120| 23| 212| 88| 50| 354| 347| 282| 397| 217| 67|
360| 236| 333| 203| 29|

| 128| 346| 287| 36| 167| 137| 345| 314| 379| 364| 96| 56| 224| 223| 159|
148| 164| 11| 35| 391|

| 19| 48| 108| 202| 229| 390| 181| 284| 61| 267| 253| 324| 143| 288| 123|
250| 192| 190| 289| 169|

| 211| 275| 179| 310| 24| 339| 163| 272| 6| 166| 210| 199| 7| 278| 28|
363| 292| 4| 357| 327|

| 348| 14| 49| 152| 252| 107| 145| 101| 375| 200| 338| 398| 30| 356| 349|
81| 249| 260| 15| 191|

```
+----+----+----+----+
| 218| 204| 277| 372| 158| 231|  13| 255|  12|  55|  37| 214| 161| 222| 352|
377| 302| 369| 173|   8|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 316|  51|  69| 184| 174| 227| 213| 367| 140| 220|  71|  58| 315|  41| 370|
115| 326| 262| 105| 386|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 198|  31| 238| 216| 336| 230| 382| 124|  93| 130| 381| 319| 112| 295| 139|
162|  66| 380| 144|  34|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
|   9| 243| 271| 154| 109| 350| 307| 129| 328| 268| 168|  94| 358|  91|  22|
84|  21| 353| 329| 322|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 300| 359|  17|  27| 193|  83| 312| 102|  99| 261| 151| 331|  42| 342| 254|
351| 366|  77| 281|  62|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 247|   2| 248| 318| 321| 241| 147|   5| 293| 311| 279| 290| 244| 294|  47|
65| 104| 118| 131| 305|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 176| 165| 196| 393| 389|  40| 136| 233|  33| 126| 286| 188| 395| 208| 237|
85| 209|  45| 207| 263|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
| 155| 256| 378|  46|  90| 361|  25| 160| 239| 298| 178|  10| 394| 111|  86|
332| 323| 234| 185| 149|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+----+----+
```

1.5 5. La fonction finale

Maintenant que tout est testé, écrivons la fonction finale. `carre_magique_aleatoire` prend en paramètre un entier n et renvoie un carré magique normal « aléatoire » de taille n. Le paramètre β permet de régler la finesse du recuit simulé.

[45]:
```python
def carre_magique_aleatoire(n, beta=2):
    A = random_normal(n)
    minimiser(A, beta)
    return A
```

Soyons conscients que pour de grandes valeurs de n, cette fonction peut mettre beaucoup de temps à renvoyer un carré magique. Disons que le temps d'attente reste raisonnable pour $n \leq 20$ …

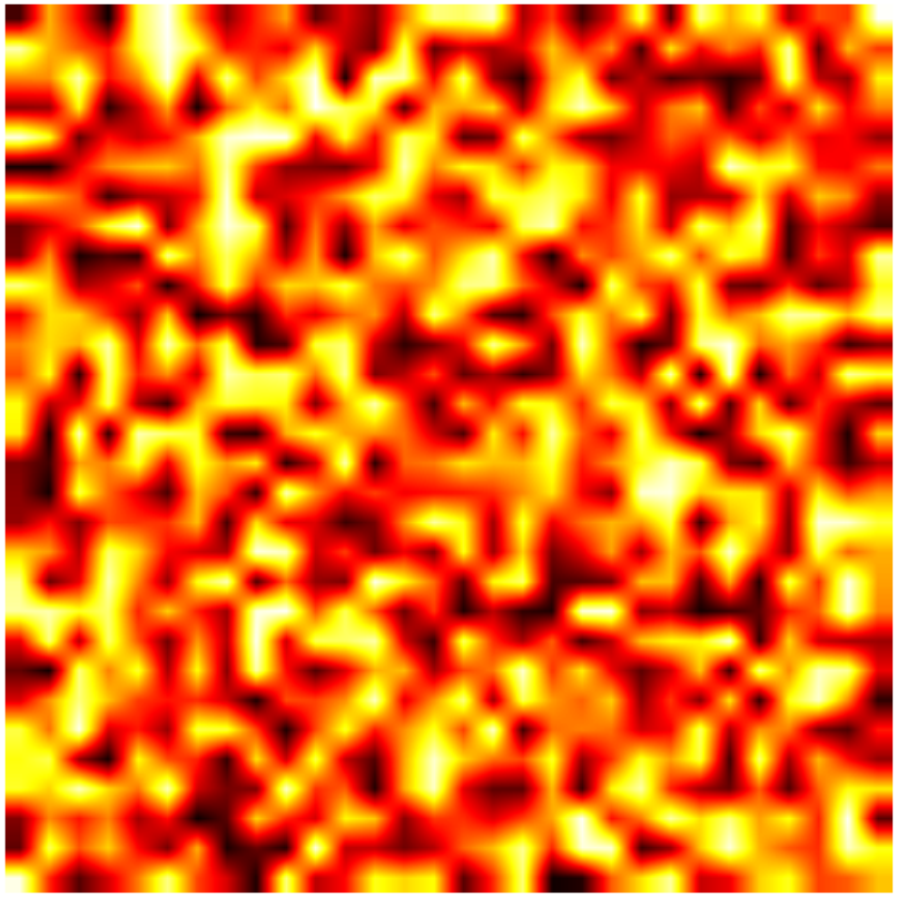
Terminons par une jolie image ...

[49]:
```
A = carre_magique_aleatoire(30, 3)
print(utils.est_carre_magique_normal(A))
```

```
True
```

[50]:
```
plt.rcParams['figure.figsize'] = (6, 6)
plt.axis(False)
plt.imshow(A, cmap='hot', interpolation='bilinear')
```

[50]:
```
<matplotlib.image.AxesImage at 0x1209ad720>
```



[ ]: