Poker

June 5, 2021

1 Poker

Marc Lorenzi

4 juin 201

[1]:
```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np
import random
import math
```

1.1 1. Introduction

1.1.1 1.1 Qu'est-ce que le poker ?

Disons-le tout de suite, nous n'allons pas jouer au poker au sens où l'entendent les joueurs usuels, mais au sens où l'entendent les probabilistes.

On dispose d'un jeu de 52 cartes.

- Les cartes ont une valeur qui, dans l'ordre de leur « force » est un entier entre 1 et 10, puis le valet, la dame et le roi. Nous attribuons à ces dernières cartes les valeurs 11, 12 et 13. Il y a donc 13 valeurs possibles. Petite subtilité, l'as peut être vu soit comme la carte de plus petite valeur, soit comme la carte de plus grande valeur. Il faudra nous en souvenir.
- Chaque éventail de valeurs a également une **couleur**, qui est trèfle (club), carreau (diamond), coeur (heart) ou pique (spade). Nous modéliserons les couleurs par les entiers de 1 à 4.

Définition. Une **partie de poker** est un ensemble de 5 cartes. Un tel ensemble est appelé une **main**.

Nous sommes donc loin d'une partie acharnée. On tire 5 cartes et la partie s'arrête. Bien entendu, les cartes sont tirées au hasard. Nous représenterons les cartes par des couples (valeur couleur).

1.1.2 1.2 Simuler une partie

Simuler une partie, c'est tirer 5 cartes **distinctes** au hasard.

[2]:
```
def partie():
    s = []
    while len(s) != 5:
        x = random.randint(1, 13)
        y = random.randint(1, 4)
        if (x, y) not in s: s.append((x, y))
    return s
```

[3]:
```
partie()
```

[3]:
```
[(3, 1), (5, 3), (2, 4), (7, 1), (10, 3)]
```

Ce n'est pas très parlant, alors rajoutons des images.

1.1.3 1.3 Images

Le répertoire `Cartes` contient des images de toutes les cartes d'un jeu, dans des fichiers `.png`. Écrivons un peu de code qui nous permettra de retrouver le nom du fichier qui contient l'image de la carte qui, pour nous, est le couple (x, y).

Tout d'abord, deux dictionnaires qui seront des variables globales du notebook.

[4]:
```
dico_cartes = dict()
for k in range(1, 11):
    dico_cartes[k] = str(k)
dico_cartes[11] = '11-JACK'
dico_cartes[12] = '12-QUEEN'
dico_cartes[13] = '13-KING'
```

[5]:
```
dico_couleurs = dict()
dico_couleurs[1] = 'CLUB'
dico_couleurs[2] = 'DIAMOND'
dico_couleurs[3] = 'HEART'
dico_couleurs[4] = 'SPADE'
```

La fonction `nom_carte` prend en paramètre un couple (x, y) et renvoie le nom cherché.

[6]:
```
def nom_carte(z):
    x, y = z
    return 'Cartes/' + dico_couleurs[y] + '-' + dico_cartes[x] + '.png'
```

[7]:
```
nom_carte((11, 3))
```

[7]:
```
'Cartes/HEART-11-JACK.png'
```

La fonction `vide` renvoie un tableau `numpy` qui sera utilisé pour espacer les images des cartes affichées.

[8]:
```
def vide(n):
    return np.zeros((167, n, 4))
```

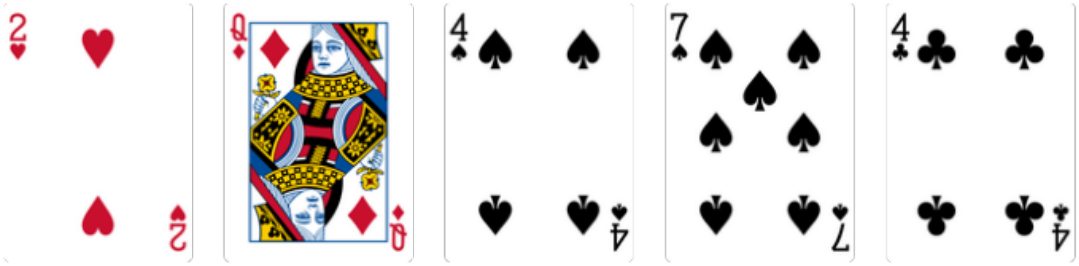
Voici enfin la fonction `afficher_cartes` qui affiche une partie.

[9]:
```
def afficher_cartes(cartes):
    plt.rcParams['figure.figsize'] = (15, 3)
    imgs = []
    for c in cartes:
        imgs.append(mpimg.imread(nom_carte(c)))
        imgs.append(vide(20))
    img = np.concatenate(tuple(imgs), axis=1)
    plt.axis(False)
    plt.imshow(img)
```

ALlons-y. On joue, on affiche.

[10]:
```
main = partie()
print(main)
afficher_cartes(main)
```

```
[(2, 3), (12, 2), (4, 4), (7, 4), (4, 1)]
```



Les matheux intégristes sont sans doute horrifiés par une telle perte de temps … quel est donc l'intérêt de **voir** ? Eh bien parce que c'est amusant.

1.2 2. Les différentes façons de gagner

Nous allons passer en revue les différents événements intéressants d'une partie de poker. Ils sont listés dans l'ordre de facilité à les reconnaître « informatiquement », du plus facile au plus difficile.

Nous nous interrogerons plus tard sur les probabilités de ces événements. Pour l'instant, nous nous contenterons de simulations : on joue des parties de poker jusqu'à la réalisation de l'événement. La fonction `jouer_jusqua_succes` prend un événement en paramètre et effectue cette simulation.

[11]:
```
def jouer_jusqua_succes(evt, img=True):
    k = 0
```

```
    while True:
        k = k + 1
        main = partie()
        if evt(main):
            if img: afficher_cartes(main)
            return k
```

1.2.1 2.1 Quinte flush

Définition. Une main est une **quinte flush** lorsqu'elle est composée de 5 cartes de même couleur et de valeurs successives.

La fonction `meme couleur` teste si toutes les cartes d'une main sont de même couleur.

[12]:
```
def meme_couleur(main):
    c = main[0][1]
    for (x, z) in main:
        if z != c: return False
    return True
```

La fonction `successives` teste si les cartes d'une main ont des valeurs successives. Pour cela, elle trie la main selon les valeurs des cartes. Rappelons nous que l'as joue un rôle particulier.

[13]:
```
def successives(main):
    main.sort(key=lambda z:z[0])
    s = [main[k][0] for k in range(5)]
    x = s[0]
    if x != 1:
        return s == [x, x + 1, x + 2, x + 3, x + 4]
    else:
        return s == [1, 2, 3, 4, 5] or s == [1, 10, 11, 12, 13]
```

Voici la fonction `est_quinte_flush` qui renvoie `True` lorsque la main est une quinte flush.

[14]:
```
def est_quinte_flush(main):
    return meme_couleur(main) and successives(main)
```

Obtenir une quinte flush est difficile. Il se peut que vous soyez obligés de patienter un peu en évaluant la cellule ci-dessous.

[15]:
```
jouer_jusqua_succes(est_quinte_flush)
```

[15]: 28627

Nous allons maintenant passer en revue les 8 autres événements d'une partie de poker.

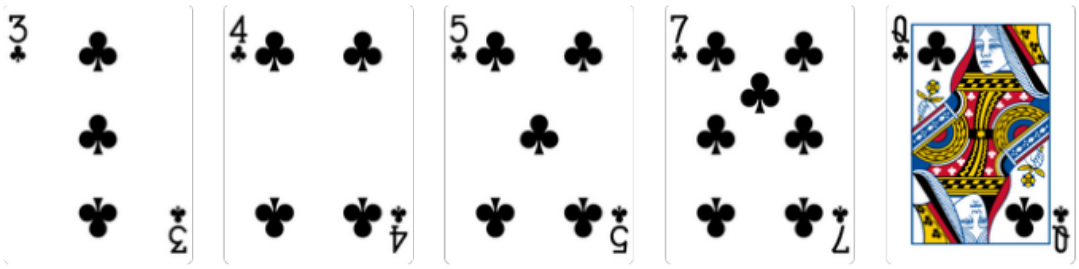
1.2.2 2.2 Couleur

Définition. Une main est une **couleur** lorsqu'elle est composée de 5 cartes de même couleur mais que ces cartes ne sont pas successives.

[16]:
```
def est_couleur(main):
    return meme_couleur(main) and not successives(main)
```

[17]:
```
jouer_jusqua_succes(est_couleur)
```

[17]: 948



1.2.3 2.3 Quinte

Définition. Une main est une **quinte** lorsqu'elle est composée de 5 cartes de valeurs successives, mais pas toutes de même couleur.

[18]:
```
def est_quinte(main):
    return not meme_couleur(main) and successives(main)
```

[19]:
```
jouer_jusqua_succes(est_quinte)
```

[19]: 223

1.2.4 2.4 Carré

Définition. Une main est un **carré** lorsqu'elle contient 4 cartes de même valeur.

[20]:
```
def est_carre(main):
    main.sort(key=lambda z:z[0])
    if main[1][0] == main[0][0] and main[2][0] == main[0][0] and main[3][0] ==␣
↪main[0][0]: return True
    elif main[2][0] == main[1][0] and main[3][0] == main[1][0] and main[4][0] ==␣
↪main[1][0]: return True
    else: return False
```

[21]:
```
jouer_jusqua_succes(est_carre)
```

[21]: 2987

1.2.5 2.5 Full

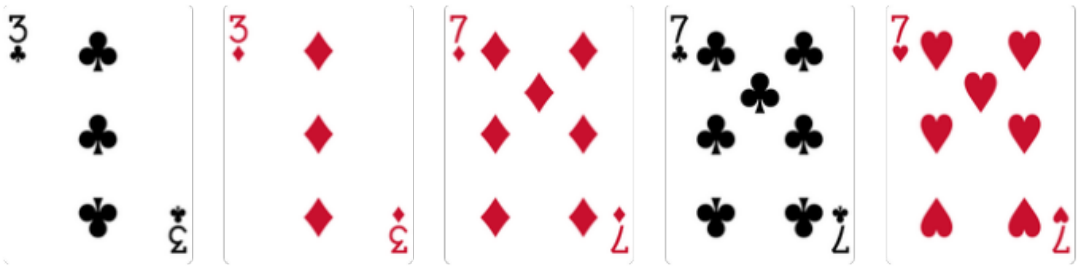
Définition. Une main est un **full** lorsqu'elle contient 3 cartes de même valeur et que les deux cartes restantes sont de même valeur.

On trie les cartes par valeur. Si les deux premières cartes ou les deux dernières cartes n'ont pas la même valeur, la main n'est pas un full. Sinon, c'est un full si et seulement si la troisième carte a la même valeur que les deux premières ou que les deux dernières.

[22]:
```
def est_full(main):
    main.sort(key=lambda z:z[0])
    if main[0][0] != main[1][0] or main[3][0] != main[4][0]: return False
    elif main[2][0] != main[1][0] and main[2][0] != main[3][0]: return False
    else: return True
```

[23]:
```
jouer_jusqua_succes(est_full)
```

[23]: 1002



1.2.6 2.6 Brelan

Définition. Une main est un **brelan** si elle contient trois cartes de même valeur et si ce n'est ni un carré ni un full.

La fonction `est_brelan` n'est certainement pas la plus efficace possible. Elle écarte les carrés et les fulls. Sinon, elle trie la main par ordre croissant des valeurs. Elle regarde si trois cartes successives ont la même valeur.

[24]:
```
def est_brelan(main):
    if est_carre(main) or est_full(main): return False
    else:
        main.sort(key=lambda z:z[0])
        if main[1][0] == main[0][0] and main[2][0] == main[0][0]: return True
        elif main[2][0] == main[1][0] and main[3][0] == main[1][0]: return True
        elif main[3][0] == main[2][0] and main[4][0] == main[2][0]: return True
        else: return False
```

[25]:
```
jouer_jusqua_succes(est_brelan)
```

[25]: 23

1.2.7 2.7 Deux paires

Définition. Une main est une double paire lorsque ce n'est ni un brelan, ni un carré ni un full, et qu'elle contient deux paires de cartes de même valeur.

[26]:
```
def est_double_paire(main):
    if est_carre(main) or est_full(main) or est_brelan(main): return False
    else:
        main.sort(key=lambda z:z[0])
        if main[1][0] == main[0][0]:
            if main[3][0] == main[2][0]: return True
            elif main[4][0] == main[3][0]: return True
            else: return False
        elif main[2][0] == main[1][0]:
            if main[4][0] == main[3][0]: return True
            else: return False
        else: return False
```

[27]:
```
jouer_jusqua_succes(est_double_paire)
```

[27]: 2

1.2.8 2.8 Une paire

Définition. Une main est une **paire** lorsqu'elle contient une paire de cartes de même valeur, et que les trois autres cartes sont de valeurs différentes, et aussi de valeurs différentes de la valeur de celles de la paire.

Pour simplifier le code, au risque de le rendre moins efficace, une main est une paire lorsque ce n'est ni une double paire, ni un brelan, ni un carré, ni un full, et qu'elle contient deux cartes de même valeur.

[28]:
```
def est_paire(main):
    if est_full(main) or est_double_paire(main) or est_carre(main) or␣
 ↪est_brelan(main): return False
    else:
        main.sort(key=lambda z:z[0])
        if main[1][0] == main[0][0] or main[2][0] == main[1][0] or \
            main[3][0] == main[2][0] or main[4][0] == main[3][0]: return True
        else: return False
```

[29]:
```
jouer_jusqua_succes(est_paire)
```

[29]: 1

1.2.9 2.9 Rien

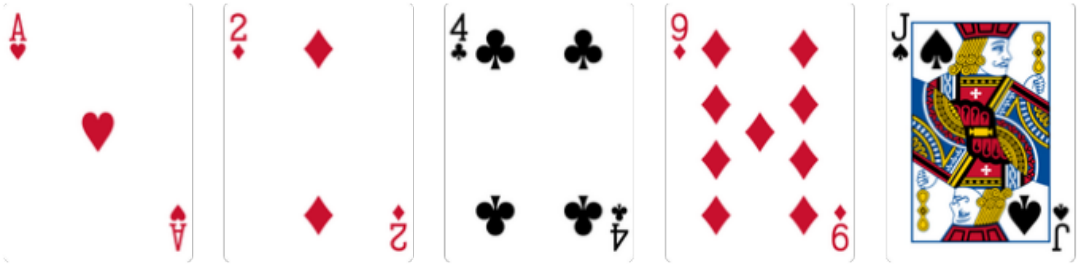
Définition. Une main est **rien** lorsqu'elle n'est aucune des mains définies précédemment.

La fonction `est_rien`est assez inefficace, mais cela suffira car tous les joueurs de poker savent qu'il est facile d'avoir **rien**.

[30]:
```
def est_rien(main):
    for f in [est_couleur, est_quinte, est_quinte_flush, est_carre, est_full,
              est_brelan, est_double_paire, est_paire]:
        if f(main): return False
    else: return True
```

[31]:
```
jouer_jusqua_succes(est_rien)
```

[31]: 1



1.3 3. Probabilités

1.3.1 3.1 Choix de l'univers

Il est assez amusant de voir qu'en gros, les événements les plus faciles à décrire en Python ont les probabilités les plus faibles. Y aurait-il là dessous une loi encore inconnue ? Cela dit, ces événements de faible probabilité sont aussi ceux pour lesquels il est le plus facile de calculer la probabilité, en tout cas ceux pour lesquels l'expression de la probabilité est la plus simple.

Nommons $N = 52$ le nombre de cartes d'un jeu de poker. Notons également $V = 13$ le nombre de cartes dans une couleur données. Prenons pour univers $\Omega = \mathcal{P}_5([|1, N|])$ des parties à 5 éléments de l'ensemble $[|1, N|]$, muni de la probabilité $\mathbb{P}$ uniforme. Remarquons que

$$|\Omega| = \binom{N}{5}$$

1.3.2 3.2 Quinte flush

Soit $\hat{Q}$ l'événement « la main est une quinte flush ». On crée une quinte flush comme suit.

- On choisit la couleur : 4 possibilités.
- On choisit la **hauteur** de la quinte, c'est à dire la carte la plus forte : $V - 3$ possibilités, qui sont 5, 6, ..., valet, dame roi, as.

Ainsi,

$$|\hat{Q}| = 4(V - 3)$$

et donc

$$\mathbb{P}(\hat{Q}) = \frac{4(V-3)}{\binom{N}{5}}$$

[32]:
```
def binom(n, k):
    p = 1
    for i in range(k):
        p *= (n - i) / (i + 1)
    return p
```

[33]:
```
def card_quinte_flush(N, V):
    return 4 * (V - 3)
```

[34]:
```
print(card_quinte_flush(52, 13))
```

```
40
```

[35]:
```
def proba_quinte_flush(N, V):
    return card_quinte_flush(N, V) / binom(N, 5)
```

[36]:
```
print(proba_quinte_flush(52, 13))
```

```
1.5390771693292702e-05
```

1.3.3 3.3 Quinte

Soit Q l'événement « la main est une quinte ». Déterminons la probabilité de Q. On crée une quinte comme suit.

- On choisit la hauteur de la quinte : $V - 3$ possibilités, comme pour la quinte flush.
- On choisit les cartes : 4 possibilités par carte puisqu'il y a 4 couleurs, donc 4^5 possibilités.

Il faut supprimer de ce décompte les quintes flush, qui sont au nombre de $4(V - 3)$. Ainsi,

$$|Q| = (4^5 - 4)(V - 3)$$

et donc

$$\mathbb{P}(Q) = \frac{(4^5 - 4)(V - 3)}{\binom{N}{5}}$$

[37]:
```
def card_quinte(N, V):
        return (4 ** 5 - 4) * (V - 3)
```

[38]:
```
print(card_quinte(52, 13))
```

```
10200
```

[39]:
```
def proba_quinte(N, V):
    return card_quinte(N, V) / binom(N, 5)
```

[40]:
```
print(proba_quinte(52, 13))
```

```
0.003924646781789639
```

1.3.4 3.4 Loi du premier succès

Dans la section 2, nous avons joué à un jeu qui consiste à faire des parties de poker **jusqu'à ce** qu'un événement se réalise. Combien de parties doit-on jouer ? Eh bien un nombre potentiellement non borné de parties. Pour modéliser ce jeu, il est nécessaire de se placer, par exemple, dans l'univers $\mathcal{U} = \Omega^{\mathbb{N}}$ des suites de jeux de poker. Cet univers est infini, et même non dénombrable. Pour le munir d'une probabilité $\hat{\mathbb{P}}$, des difficultés surviennent. Je ne les détaillerai pas ici. Je vais dans la suite affirmer un certain nombre de choses sans preuve.

Soit A un événement de l'univers Ω, de probabilité $p \in]0,1[$. Pour tout $i \in \mathbb{N}$, soit $X_i : \mathcal{U} \to \{0,1\}$ définie pour tout $u = (u_0, u_1, \ldots) \in \mathcal{U}$ par $X_i(u) = 1$ si $u_i \in A$, et 0 sinon. La variable aléatoire X_i suit une loi de Bernoulli de paramètre p, et les X_i, $i \in \mathbb{N}$, sont indépendantes.

Soit $Y : \mathcal{U} \to \mathbb{N}^*$ la variable aléatoire définie par $Y(u) = n$ si $X_0(u) = \ldots = X_{n-2}(u) = 0$, et $X_{n-1}(u) = 1$. On a pour tout $n \in \mathbb{N}$,

$$\mathbb{P}(Y = n) = \mathbb{P}(X_0 = 0, \ldots, X_{n-2} = 0, X_{n-1} = 1)$$

Par l'indépendance des X_i,

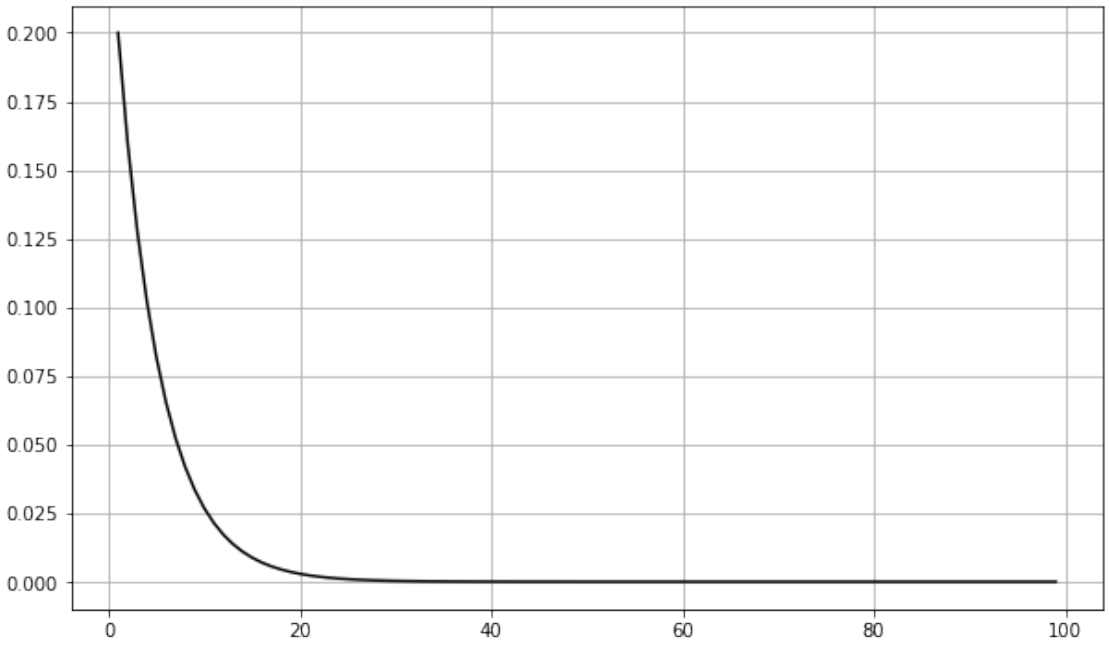
$$\mathbb{P}(Y = n) = \mathbb{P}(X_0 = 0) \ldots \mathbb{P}(X_{n-2} = 0)\mathbb{P}(X_{n-1} = 1) = p(1-p)^{n-1}$$

Y suit ce que l'on appelle une **loi géométrique** de paramètre p.

[41]:
```
def proba_geom(p, n): return p * (1 - p) ** (n - 1)
```

[42]:
```
def plot_geom(p, nmax, step):
    plt.rcParams['figure.figsize'] = (10, 6)
    ns = range(1, nmax, step)
    ps = [proba_geom(p, n) for n in ns]
    plt.plot(ns, ps, 'k')
    plt.grid()
```

[43]:
```
plot_geom(0.2, 100, 1)
```



Remarquons que, comme on on peut s'y attendre,

$$\sum_{n=1}^{\infty} \mathbb{P}(Y=n) = p\sum_{n=0}^{\infty}(1-p)^{n-1} = \frac{p}{1-(1-p)} = 1$$

L'espérance de Y est

$$E(Y) = \sum_{n=1}^{\infty} n\mathbb{P}(Y=n) = p\sum_{n=1}^{\infty} n(1-p)^{n-1}$$

La somme de série ci-dessus se calcule facilement en considérant la fonction $\varphi :]-1,1[\to \mathbb{R}$ définie par

$$\varphi(x) = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

En admettant que l'on peut dériver la série « terme à terme », on a

$$\varphi'(x) = \sum_{n=1}^{\infty} nx^{n-1} = \frac{1}{(1-x)^2}$$

d'où

$$\varphi'(1-p) = \sum_{n=1}^{\infty} n(1-p)^{n-1} = \frac{1}{p^2}$$

Ainsi,

$$E(Y) = \frac{1}{p}$$

Par des calculs analogues, on peut montrer que la variance de Y est

$$V(Y) = \frac{1-p}{p^2}$$

Remarquons que si p est petit, alors $V(Y) \sim \frac{1}{p^2}$. Ainsi, l'espérance de Y et son écart-type sont équivalents.

1.3.5 3.5 Retour aux quintes

Dans nos expériences de la partie 2, nous jouions au poker jusqu'à ce que l'on obtienne une quinte. Si l'on appelle Y le nombre de parties nécessaires à l'obtention d'une quinte, Y suit donc une loi géométrique de paramètre p, où p est la probabilité d'une quinte. Ainsi, $E(Y) = \frac{1}{p}$.

[44]:
```
print(1 / proba_quinte(52, 13))
```

```
254.8
```

[45]:
```
def ecart_type_quinte(N, V):
    p = proba_quinte(N, V)
    return math.sqrt((1 - p) / (p ** 2))
```

[46]:
```
ecart_type_quinte(52, 13)
```

[46]:
```
254.29950845410616
```

Le nombre moyen de parties nécessaires à l'obtention d'une quinte est environ 255. Faisons une simulation.

[47]:
```
def simul_succes(evt, nbsim):
    s = 0
    for k in range(nbsim):
        n = jouer_jusqua_succes(evt, img=False)
        s += n
    return s / nbsim
```

Bien entendu, on ne trouvera sans doute pas pile 254. Mais le résultat devrait ne pas en être très éloigné. Pour plus de précision (mais aussi plus de temps de calcul), on peut augmenter le nombre de simulations.

[48]:
```
simul_succes(est_quinte, 1000)
```

[48]:
```
254.536
```

Avons nous envie de lancer la simulation pour une quinte flush ?

[49]:
```
print(1 / proba_quinte_flush(52, 13))
```

```
64974.0
```

Décidément, non.

1.4 4. Les autres événements

1.4.1 4.2 Couleur

Soit C l'événement « la main est une couleur ». On crée une couleur comme suit.

- On choisit la couleur : 4 possibilités.
- On choisit 5 cartes dans la couleur considérée : $\binom{V}{5}$ possibilités.

N'oublions pas de supprimer les quintes flush du décompte. Ainsi,

$$|C| = 4\binom{V}{5} - 4(V-3)$$

et donc

$$\mathbb{P}(C) = \frac{4\binom{V}{5} - 4(V-3)}{\binom{N}{5}}$$

[50]:
```
def card_couleur(N, V):
    return 4 * binom(V, 5) - 4 * (V - 3)
```

[51]:
```
print(card_couleur(52, 13))
```

```
5108.0
```

[52]:
```
def proba_couleur(N, V):
    return card_couleur(N, V) / binom(N, 5)
```

[53]:
```
print(1 / proba_couleur(52, 13))
```

```
508.80187940485513
```

[54]:
```
simul_succes(est_couleur, 1000)
```

[54]:
```
514.438
```

1.4.2 4.3 Carré

Soit $\hat{C}$ l'événement « carré ». On crée un carré comme suit.

- On choisit la valeur du carré : V possibilités
- On choit la carte restante : $N - 4$ possibilités.

Ainsi,

$$|\hat{C}| = V(N-4)$$

et donc

$$\mathbb{P}(\hat{C}) = \frac{V(N-4)}{\binom{N}{5}}$$

[55]:
```
def card_carre(N, V):
    return V * (N - 4)
```

[56]:
```
print(card_carre(52, 13))
```

```
624
```

[57]:
```
def proba_carre(N, V):
    return card_carre(N, V) / binom(N, 5)
```

[58]:
```
print(1 / proba_carre(52, 13))
```

```
4165.0
```

L'événement $\hat{C}$ ayant une toute petite probabilité, prenons un petit nombre de simulations. Évidemment, il ne faut pas s'attendre à une bonne précision du résultat.

[59]:
```
simul_succes(est_carre, 100)
```

[59]:
```
4117.49
```

1.4.3 4.4 Full

Soit F l'événement « full ». On crée un full comme suit.

- On choisit la valeur des trois cartes qui ont la même valeur : V possibilités.
- On choisit trois des 4 cartes ayant cette valeur : $\binom{4}{3} = 4$ possibilités.
- On choisit la valeur des deux cartes qui ont la même valeur : $V - 1$ possibilités.
- On choisit deux des 4 cartes ayant cette valeur : $\binom{4}{2} = 6$ possibilités.

Ainsi,

$$|F| = 24V(V-1)$$

et donc

$$\mathbb{P}(F) = \frac{24V(V-1)}{\binom{N}{5}}$$

[60]:
```
def card_full(N, V):
    return 24 * V * (V - 1)
```

[61]:
```
print(card_full(52, 13))
```

```
3744
```

[62]:
```
def proba_full(N, V):
    return card_full(N, V) / binom(N, 5)
```

[63]:
```
print(1 / proba_full(52, 13))
```

```
694.1666666666667
```

[64]:
```
simul_succes(est_full, 500)
```

[64]:
```
709.822
```

1.4.4 4.5 Brelan

Soit B l'événement « brelan ». On crée un brelan comme suit :

- On choisit la valeur des trois cartes qui ont la même valeur : V possibilités.
- On choisit trois des 4 cartes ayant cette valeur : $\binom{4}{3} = 4$ possibilités.
- On choisit les deux dernières cartes. Ces deux cartes ne doivent pas avoir la valeur des trois cartes déjà choisies. On les choisit donc dans un ensemble de cardinal $N-4$. Ensuite, ces deux cartes ne doivent pas avoir la même valeur. Une fois choisie la première carte, il ne reste donc que $N-8$ possibilités. Il faut également faire attention au fait que l'ordre dans lequel on choisit ces deux cartes ne compte pas. Une division par deux s'impose donc : $\frac{1}{2}(N-4)(N-8)$ possibilités.

Ainsi,

$$|B| = 2V(N-4)(N-8)$$

et donc

$$\mathbb{P}(B) = \frac{2V(N-4)(N-8)}{\binom{N}{5}}$$

[65]:
```
def card_brelan(N, V):
    return 2 * V * (N - 4) * (N - 8)
```

[66]:
```
print(card_brelan(52, 13))
```

```
54912
```

[67]:
```
def proba_brelan(N, V):
    return card_brelan(N, V) / binom(N, 5)
```

[68]:
```
print(1 / proba_brelan(52, 13))
```

```
47.32954545454545
```

[69]:
```
simul_succes(est_brelan, 1000)
```

[69]:
```
49.108
```

Il nous reste à nous occuper des paires.

1.4.5 4.6 Deux paires

Soit P_2 l'événement « deux paires ». On crée deux paires comme suit.

- On choisit la première paire : $V\binom{4}{2} = 6V$ possibilités.
- On choisit la deuxième paire : $(V-1)\binom{4}{2} = 6(V-1)$ possibilités.

Ici aussi, l'ordre du choix de la première et de la deuxième paire n'est pas important, une division par 2 s'imposera dans le décompte final.

- On choisit la valeur de la carte restante : $N-8$ possibilités.

Ainsi,

$$|P_2| = 18V(V-1)(N-8)$$

et donc

$$\mathbb{P}(P_2) = \frac{18V(V-1)(N-8)}{\binom{N}{5}}$$

[70]:
```
def card_2paires(N, V):
    return 18 * V * (V - 1) * (N - 8)
```

[71]:
```
print(card_2paires(52, 13))
```

```
123552
```

[72]:
```
def proba_2paires(N, V):
    return card_2paires(N, V) / binom(N, 5)
```

[73]:
```
print(1 / proba_2paires(52, 13))
```

```
21.035353535353536
```

[74]:
```
simul_succes(est_double_paire, 1000)
```

[74]:
```
21.037
```

1.4.6 4.7 Une paire

Soit P_1 l'événement « une paire ». On crée une paire comme suit.

- On choisit la paire : $V\binom{4}{2} = 6V$ possibilités.
- On choisit la première des trois cartes restantes : $N - 4$ possibilités.
- On choisit la deuxième des trois cartes restantes : $N - 8$ possibilités.
- On choisit la troisième des trois cartes restantes : $N - 12$ possibilités.

Ici aussi, l'ordre du choix des trois dernières cartes n'est pas important, une division par $3! = 6$ s'impose dans le décompte final.

Ainsi,

$$|P_1| = V(N-4)(N-8)(N-12)$$

et donc

$$\mathbb{P}(P_1) = \frac{V(N-4)(N-8)(N-12)}{\binom{N}{5}}$$

[75]:
```
def card_1paire(N, V):
    return V * (N - 4) * (N - 8) * (N - 12)
```

[76]:
```
print(card_1paire(52, 13))
```

```
1098240
```

[77]:
```
def proba_1paire(N, V):
    return card_1paire(N, V) / binom(N, 5)
```

[78]:
```
print(1 / proba_1paire(52, 13))
```

```
2.3664772727272725
```

[79]:
```
simul_succes(est_paire, 5000)
```

[79]:
```
2.3312
```

1.4.7 4.8 Rien

Appelons Z l'événement « rien ». On a

$$Q \cup \hat{Q} \cup \hat{C} \cup C \cup F \cup B \cup P_2 \cup P_1 \cup Z = \Omega$$

Les événements de cette réunion sont disjoints deux à deux. Ainsi,

$$\mathbb{P}(Z) = 1 - \mathbb{P}(Q) - \mathbb{P}(\hat{Q}) - \mathbb{P}(C) - \mathbb{P}(\hat{C}) - \mathbb{P}(F) - \mathbb{P}(B) - \mathbb{P}(P_2) - \mathbb{P}(P_1)$$

[80]:
```
def card_rien(N, V):
    return binom(N, 5) - card_quinte(N, V) - card_quinte_flush(N, V) -␣
↪card_carre(N, V) - card_couleur(N, V) \
        - card_full(N, V) - card_brelan(N, V) - card_2paires(N, V) -␣
↪card_1paire(N, V)
```

[81]:
```
print(card_rien(52, 13))
```

```
1302540.0
```

[82]:
```
def proba_rien(N, V):
    return card_rien(N, V) / binom(N, 5)
```

[83]:
```
print(1 / proba_rien(52, 13))
```

```
1.995301487862177
```

[84]:
```
simul_succes(est_rien, 5000)
```

[84]:
```
1.9918
```

1.5 5. Bilan

Hsitoire de résumer tout cela, il nous reste à donner la liste des événements du poker triés par probabilités croissantes. Bien entendu, Python fera cela pour nous.

Voici une fonction qui renvoie la liste des couples (nom de l'événement, probabilité).

[85]:
```
def evenements(N, V):
    return [
    ('couleur', card_couleur(N, V)),
    ('quinte', card_quinte(N, V)),
    ('quinte flush', card_quinte_flush(N, V)),
    ('carre', card_carre(N, V)),
    ('full', card_full(N, V)),
    ('brelan', card_brelan(N, V)),
    ('2 paires', card_2paires(N, V)),
    ('1 paire', card_1paire(N, V)),
    ('rien', card_rien(N, V))
]
```

Et voici la conclusion. On affiche pour chaque événement, son nom, le cardinal de l'événement, et l'inverse (arrondi) de sa probabilité, c'est à dire l'espérance du nombre de parties nécessaires à la réalisation de cet événement.

[86]:
```
evts = evenements(52, 13)
evts.sort(key=lambda z:z[1])
for evt, card in evts:
    p = card / binom(52, 5)
    print('%-14s: %8.0f %10.2f' % (evt, card, 1 / p))
```

```
quinte flush  :       40   64974.00
carre         :      624    4165.00
full          :     3744     694.17
couleur       :     5108     508.80
quinte        :    10200     254.80
brelan        :    54912      47.33
2 paires      :   123552      21.04
1 paire       :  1098240       2.37
rien          :  1302540       2.00
```