1110-PROBA-Ejercicios-3

April 22, 2018

En estos ejercicios hay que determinar las probabilidades pedidas mediante simulación, pero también es interesante resolver el ejercicio teóricamente y comparar el resultado.

Ejercicio 1 Supongamos una familia con dos hijos y uno de ellos es un chico. Calcula la probabilidad de que los dos hijos sean chicos.

Suponemos en este caso que es aleatorio el que sea chico o chica, y vamos a repetir el experimento N veces para calcular la probabilidad.

De manera teórica, como solo puede ser chico o chica, queremos hallar P(El segundo hijo sea chico| El primer hijo es chico) = P(El segundo hijo sea chico), pues son eventos independientes, con lo que la probabilidad es un medio.

```
In [9]: def probChico(N):
            total = 0
            for _ in xrange(N):
                #Si sale menor que 0.5 es chico, si no es chica
                if(random() <= 0.5):
                    total += 1

            return (total/N).n()

        probChico(10^5)
```

```
Out[9]: 0.498880000000000
```

Así, se aprecia claramente como la probabilidad es aproximadamente de 1/2

```
In [28]: def chico():
             if(random() <= 0.5):
                 return 1
             return 0

         def probChico(N):
             unHijoChico = 0
             dosHijoChico = 0
             for _ in xrange(N):
                 hijo1 = chico()
                 hijo2 = chico()
```

```
            if(hijo1 == 1 or hijo2 == 1):
                unHijoChico += 1
            if(hijo1 == 1 and hijo2 == 1):
                dosHijoChico += 1

        return (dosHijoChico/unHijoChico).n()

probChico(100000)
```

```
Out[28]: 0.332658551527582
```

Ejercicio 2 Supongamos una familia con dos hijos y el menor es una chica. Calcula la probabilidad de que los dos hijos sean chicas.

Tenemos el mismo experimento que en el apartado anterior, pues al ser eventos independientes, simplemente tenemos que calcular la probabilidad de que el segundo hijo sea una chica, con lo que valdría con usar el mismo código del ejercicio anterior, y el resultado sería también de 1/2.

Ejercicio 3 Un juego de azar, con dos jugadores A y B, es *justo* si el premio de A por la probabilidad de que A gane es igual al premio de B por la probabilidad de que B gane. Si el juego es justo y se juega un número muy grande de veces es muy probable que los dos jugadores terminen con ganancia prácticamente cero.

Supongamos la siguiente apuesta de casino: se lanza una moneda 1000 veces y si aparecen 10 caras seguidas el jugador gana mientras que si no aparecen el casino gana. Si el casino pide que el jugador apueste 1 euro para poder jugar, £cuál debe ser el premio para el jugador si el juego es justo?

Los juegos de azar en los casinos nunca son justos, ya que si el casino no gana grandes cantidades de dinero, que los jugadores pierden, cerraría.

Sabemos que el premio del casino es 1 euro, y la probabilidad de que el casino gane es la probabilidad de que en 1000 lanzamientos no salgan 10 caras seguidas.

Vamos a calcular mediante una simulación la probabilidad de que en 1000 lanzamientos aparezcan 10 caras seguidas

```
In [14]: def ganaJugador(N):
             gana = 0
             for _ in xrange(N):
                 n = 0
                 for i in xrange(1000):
                     if random() <= 0.5:
                         n += 1
                         if n >= 10:
                             gana += 1
                             break
                     else:
                         n = 0
             return (gana/N).n()

         ganaJugador(10^3)
```

```
Out[14]: 0.386000000000000
```

Así, como el casino gana un euro aproximadamente el 62% de las veces, $0.38 * Gananciajugador = 0.62 * 1$ y por tanto, el jugador debería ganar 1,63€ para que el juego fuese justo.

Ejercicio 4 El jugador A lanza un dado y el jugador B dos dados. Si A consigue un número mayor o igual al máximo obtenido por B entonces gana A, y en caso contrario gana B. £Cuál es la probabilidad de que gane B?

```
In [27]: def probGanaB(N):
             gana = 0
             for _ in xrange(N):
                 A = randint(1, 6)
                 B = randint(1, 6) + randint(1, 6)
                 if A < B:
                     gana += 1

             return (gana/N).n()

         probGanaB(10^5)
```

```
Out[27]: 0.839240000000000
```

Así, podemos ver claramente que la probabilidad de que gane B es aproximadamente del 84%.

Para resolver teóricamente, podemos hacerlo por casos, calculando la probabilidad de que gane A como

$$P(GanaA) = \sum_{i=1}^{6} \frac{1}{6} * P(B <= A) = \frac{1}{6} * (0 + \frac{1}{36} + \frac{3}{36} + \frac{6}{36} + \frac{10}{36} + \frac{13}{36})$$

```
In [25]: 1 - ((1+3+6+10+13)/(36*6)).n()
```

```
Out[25]: 0.847222222222222
```

Y por tanto, podemos ver como ambas probabilidades se acercan bastante

Ejercicio 5 Consideramos el siguiente 'juego': inicialmente hay n jugadores y en cada fase del juego cada jugador vivo elige **al azar** otro jugador vivo, distinto de sí mismo, y lo mata. El juego se repite hasta que queda un único jugador, *el elegido del destino*, o bien ninguno. En principio, es perfectamente posible que el juego se juegue una única vez.

1) Estima, con dos cifras decimales 'correctas', la probabilidad de que, partiendo de $n = 100$ jugadores, haya un superviviente. Como se indicó en clase, entendemos que que son cifras decimales 'correctas' las que no cambian cuando se incrementa suficientemente el número N de 'casos posibles'.

2) £Qué probabilidad tengo, si decido jugar con otros 99 jugadores, de ser yo el *elegido del destino*? Modifica el programa del apartado anterior para estimar la probabilidad y explica el resultado obtenido.

3) Ahora queremos estudiar la duración del juego. Para eso debemos calcular *promedios* de la duración. Define una función *promedio(n,N)* que calcule el promedio y la desviación estándar de la duración con n jugadores iniciales y N repeticiones del juego. Evalúa *promedio(100,10^5)* y comenta los resultados obtenidos.

4) Finalmente, estudia la variación de los promedios al variar n entre 10 y 200 saltando de 10 en 10. Realiza un gráfico de los resultados y analiza la dependencia funcional (el promedio como función de n). En este cuarto apartado probablemente tendrás que **elegir con más cuidado un** N (el número de repeticiones del juego utilizadas para calcular cada promedio) **adecuado** dadas las capacidades bastante limitadas de las máquinas que usamos.

Apartado 1 Vamos a representar a los jugadores como una lista de n elementos, donde cada elemento es un 1 si el jugador esta vivo, o un 0 en caso contrario.

```
In [2]: def jugarRonda(lista):
            for j in xrange(len(lista)):
                if lista[j] != 0:
                    #Matamos a otro jugador vivo
                    i = randint(0, len(lista) - 1)
                    while i == j and lista[i] == 0:
                        i = randint(0, len(lista) - 1)

                    lista[i] = 0
            return [1 for i in xrange(lista.count(1))]

        def jugar(N):
            L = [1 for _ in xrange(N)]
            while len(L) > 1:
                L = jugarRonda(L)
            return len(L)

        jugar(1000)
```

```
Out[2]: 1
```

```
In [60]: def comprobarSuperviviente(N, n):
             suma = 0
             for _ in xrange(N):
                 suma += jugar(n)

             return (suma/N).n()

         for i in range(1, 6):
             print comprobarSuperviviente(10^i, 100)
```

```
0.900000000000000
0.820000000000000
0.859000000000000
0.838400000000000
```

```
0.838270000000000
```

Así, la probabilidad de que haya un elegido, con dos crifras decimales correctas, es de un 83,8%

Apartado 2 Vamos a ver la probabilidad que tenemos de ser nosotros el elegido del destino. Para ello fijamos una posición, y vemos que le pasa a dicha posición, cuando consigue ser el elegido.

```
In [65]: def jugarRonda2(lista):
             for j in xrange(len(lista)):
                 if lista[j] != 0:
                     #Matamos a otro jugador vivo
                     i = randint(0, len(lista) - 1)
                     while i == j and lista[i] == 0:
                         i = randint(0, len(lista) - 1)
                     # Si nos matan, devolvemos -1
                     if i == 0:
                         return -1
                     lista[i] = 0
             return [1 for i in xrange(lista.count(1))]

         def jugar2(N):
             L = [1 for _ in xrange(N)]
             while len(L) > 1:
                 L = jugarRonda2(L)
                 # Solo devolvemos 1 cuando he ganado yo,
                 # en el momento en que nos matan, devolvemos 0
                 if L == -1:
                     return 0
             return len(L)

         def comprobarSupervivienteYo(N, n):
             suma = 0
             for _ in xrange(N):
                 suma += jugar2(n)

             return (suma/N).n()

In [67]: %time comprobarSupervivienteYo(10^5, 100)

CPU times: user 32.7 s, sys: 12 ms, total: 32.7 s
Wall time: 32.7 s


Out[67]: 0.0100100000000000
```

Por tanto, podemos ver que la probabilidad de ganar es de, aproximadamente, el 1%. Tiene cierto setido, pues la gran mayoría de las partidas hay uno que sale elegido, y hay 100 jugadores.

Apartado 3 Ahora queremos estudiar la duración del juego. Para eso debemos calcular promedios de la duración. Define una función promedio(n,N) que calcule el promedio y la desviación estándar de la duración con nn jugadores iniciales y NN repeticiones del juego. Evalúa promedio(100,10^5) y comenta los resultados obtenidos.

```
In [11]: def jugar3(N):
             #Devuelve el numero de rondas que se ha jugado al juego.
             n = 0
             L = [1 for _ in xrange(N)]
             while len(L) > 1:
                 n += 1
                 L = jugarRonda(L)
             return n

         def promedio(N, n):
             L = []
             for _ in xrange(N):
                 L.append(jugar3(n))

             avg = (sum(L)/N).n()
             der = sqrt(sum([(i-avg)^2 for i in L])/N).n()
             return avg, der

         %time print promedio(10^5, 100)
```

```
(6.12836000000000, 0.534924023016021)
CPU times: user 1min 1s, sys: 96 ms, total: 1min 1s
Wall time: 1min 1s
```

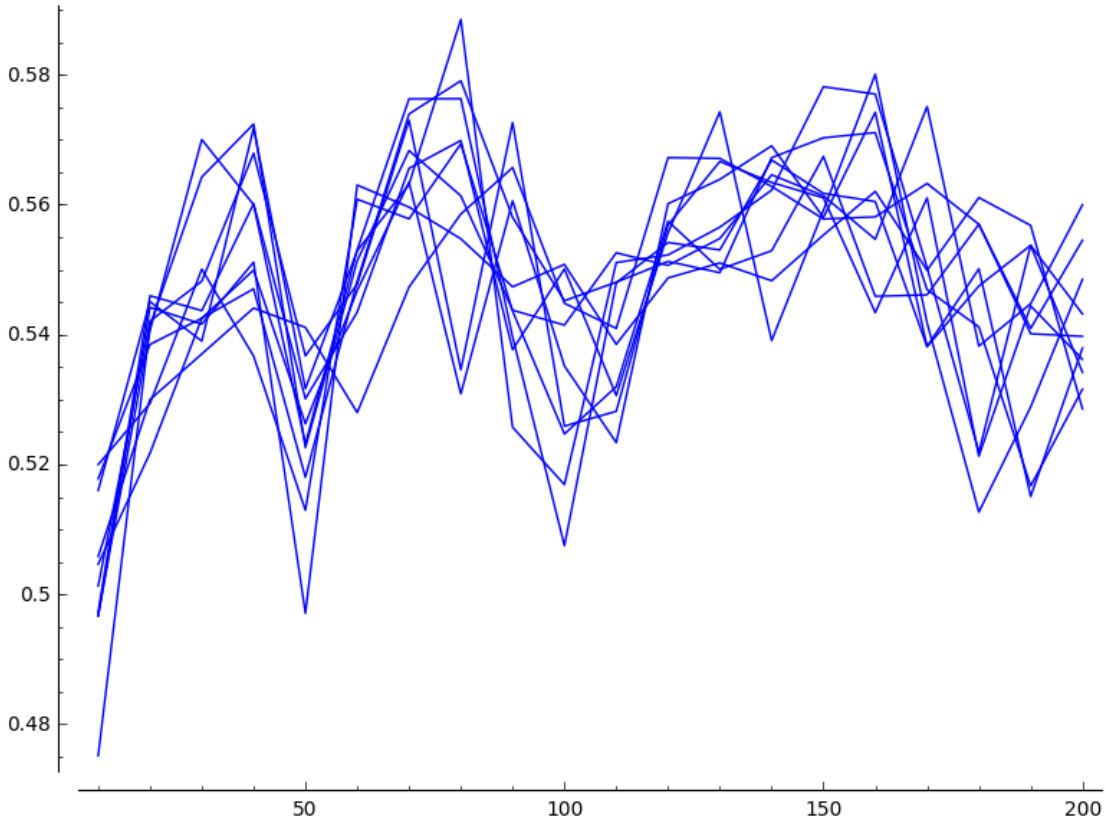
Así podemos ver que en unas 6-7 rondas el juego ha acabado, cuando juegan 100 personas.

Apartado 4 Vamos a probar primero con N=10^3 y luego con N = 10^4, pues con 10^5 tendríamos que esperar mas de quince minutos.

Suponemos que al hablar de la variación de los promedios se refiere a la derivación estándar, con lo que la representamos gráficamente en función de n.

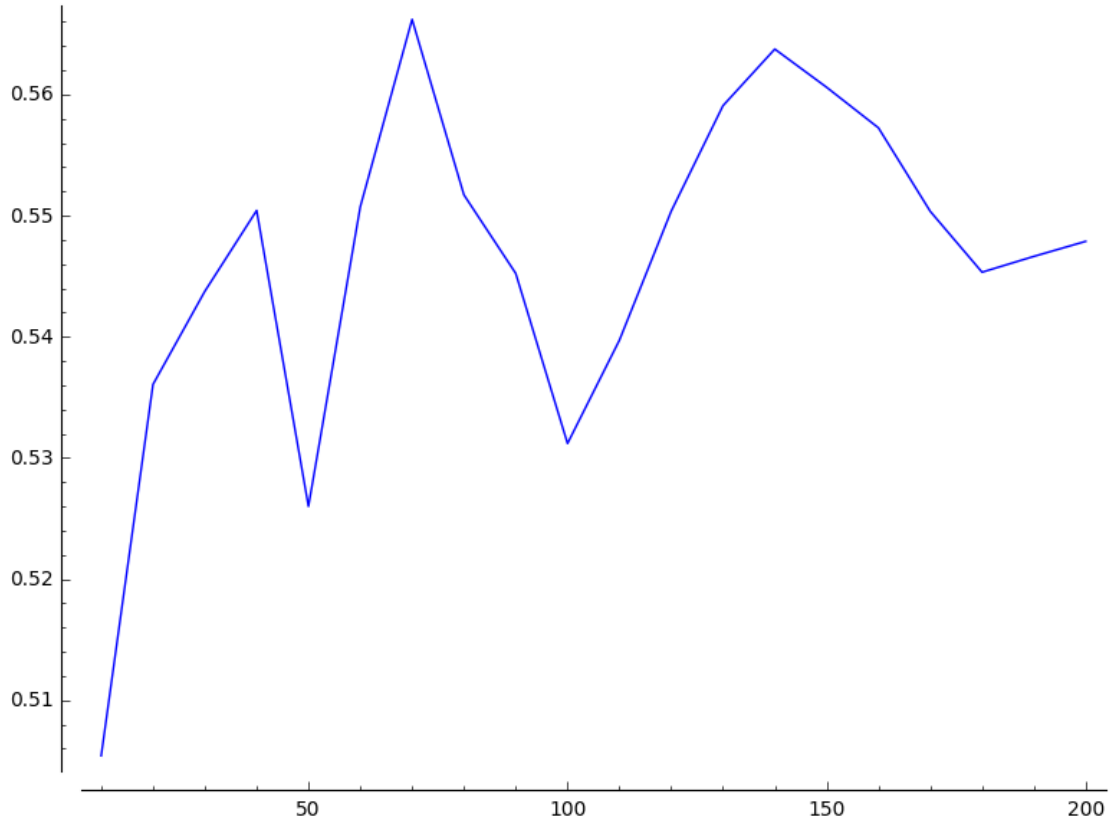
```
In [14]: G = []
         for _ in xrange(10):
             L = []
             for n in xrange(10, 201, 10):
                 L.append((n, promedio(10^3, n)[1]))

             G.append(line2d(L))

         sum(G)
```

```
Out[14]:
```



In [13]:
```
L = []
for n in xrange(10, 201, 10):
    L.append((n, promedio(10^4, n)[1]))

line2d(L)
```

Out[13]:



Así podemos ver fácilmente como la derivación se mantiene estable entre 0.5 y 0.6. Sin embargo, como se ve en el primer gráfico, no parece haber ninguna relación entre n y la derivación.

Ejercicio 6 Estudiamos un modelo llamado "perros y pulgas", o también de las "urnas de Ehrenfest". Este modelo es de gran interés en la Física de sistemas con un gran número de partículas (mecánica estadística).

Suponemos dos perros A y B tales que en el instante $t = 0$ A tiene n pulgas (numeradas de 0 a $n-1$) y B ninguna. Los perros están durmiendo uno al lado del otro y las pulgas pueden saltar de uno a otro sin problema. La regla de evolución del sistema es la siguiente: Si en el instante t, $L_A(t)$ y $L_B(t)$ son las listas de pulgas en A y B, elegimos un entero aleatorio en el intervalo cerrado $[0, n-1]$ y la pulga que lleva ese número salta cambiando de perro. Así obtenemos las nuevas listas $L_A(t+1)$ y $L_B(t+1)$ que determinan el estado del sistema en el instante $t+1$ (el tiempo es, como en otros ejemplos que hemos considerado, discreto $t = 0, 1, 2, 3, \ldots$).

A) En primer lugar programa una función $siguiente(n, L_A, L_B)$ que reciba el estado del sistema en un instante y devuelva el estado en el siguiente.

B) Realiza un gráfico que represente, comenzando con 1000 pulgas en A, la evolución temporal del número de pulgas en B. Debe observarse claramente la estabilización del número de pulgas, y, a partir de ese momento, oscilaciones pequeñas.

C) Cambia la función del apartado A por otra, $siguiente1(n, n_B)$, que únicamente tenga en cuenta el número de pulgas en el perro B, ya que todas las demás están en el perro A. A

fin de cuentas, £qué nos importa cómo se llama la pulga que ha saltado al pasar del instante t al $t+1$? Nos debe bastar con saber cuántas pulgas hay en cada perro, y el comportamiento del sistema debe ser el mismo que con la versión en el apartado A). Repite el gráfico del apartado B) y comprueba que se obtiene, esencialmente, el mismo.

D) Estudia ahora la siguiente variante del modelo: para pasar del estado t al $t+1$:

1) Elegimos aleatoriamente uno de los dos perros.

2) Cada perro tiene una probabilidad, p_A o p_B que no cambia, de que una pulga salte al otro.

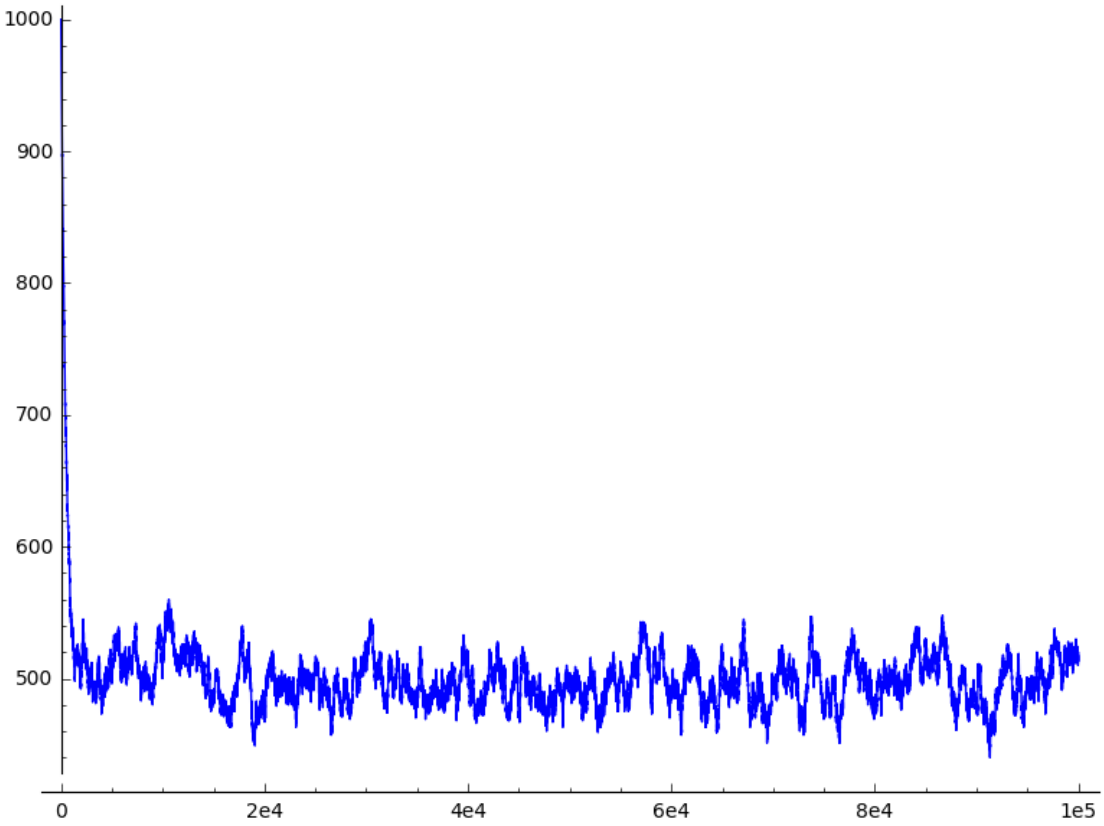
3) Cuando un perro ha sido elegido en el sorteo realizado en 1) y tiene pulgas, una pulga salta al otro perro con probabilidad p_A, si se trata del perro A, o p_B si el elegido es B.

£Se estabiliza el número de pulgas en cada perro? A largo plazo, £qué se obtiene para el número de pulgas en B? La respuesta dependerá de los valores de p_A y p_B utilizados, y para responder se pueden realizar gráficos con diferentes valores de las probablidades.

Apartado A y B

In [23]:
```
def siguiente(n, La, Lb):
    pulga = randint(0, n-1)
    if pulga in A:
        A.remove(pulga)
        B.append(pulga)
    else:
        A.append(pulga)
        B.remove(pulga)

A = [i for i in xrange(0, 1000)]
B = []

L = []
for i in range(0, 10^5):
    L.append((i, len(A)))
    siguiente(1000, A, B)

line2d(L)
```

Out[23]:



Una vez desarrollada la función con listas, es muy sencillo representar gráficamente las pulgas del perro A, y se ve claramente como tiende a un equilibrio alrededor de 500, la mitad.

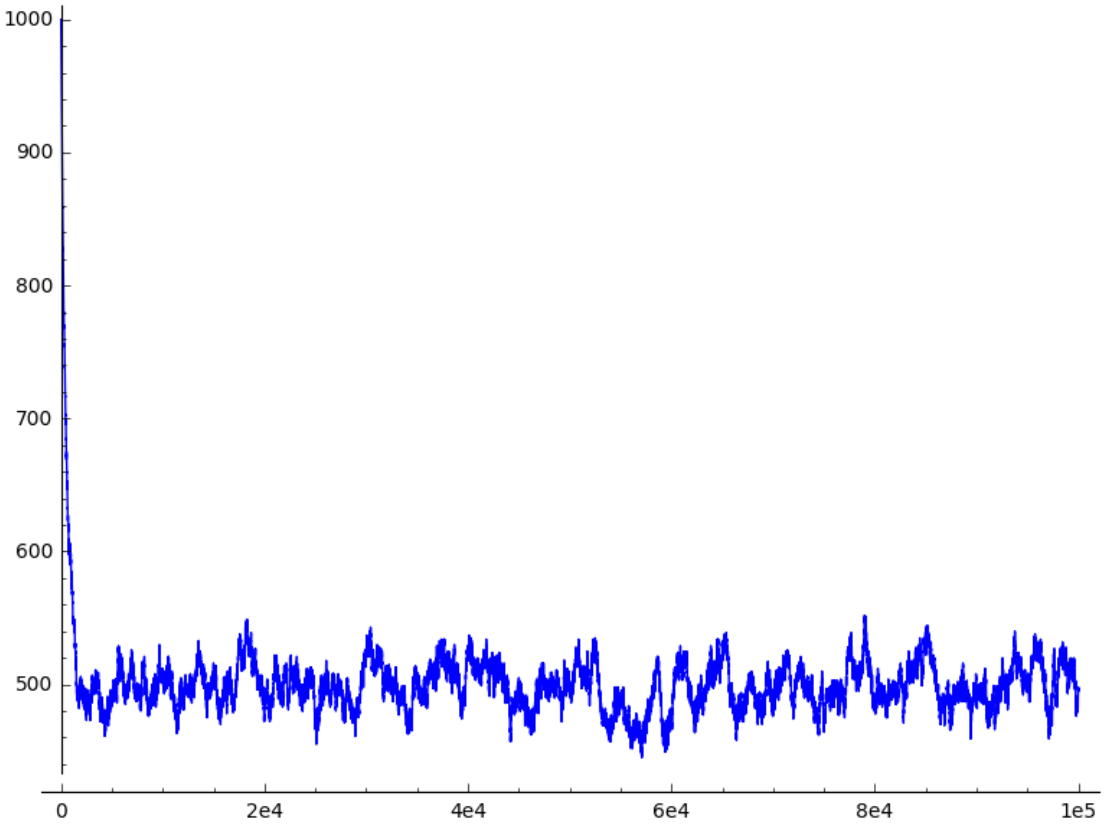
Apartado C Decidimos, en vez de usar listas, usar un numero entero A y B que describen el número de pulgas que tiene cada uno de los perros.

```
In [19]: def siguiente(n, A, B):
             pulga = randint(0, n-1)
             if pulga <= A:
                 A -= 1
                 B += 1
             else:
                 A += 1
                 B -= 1

             return A, B

         A = 10
         B = 0
         A, B = siguiente(10, A, B)
```

```
Out[19]: (9, 1)
```

```
In [22]: L = []

         A = 1000
         B = 0
         for i in range(0, 10^5):
             L.append((i, A))
             A, B = siguiente(A+B, A, B)

         line2d(L)
```

Out[22]:



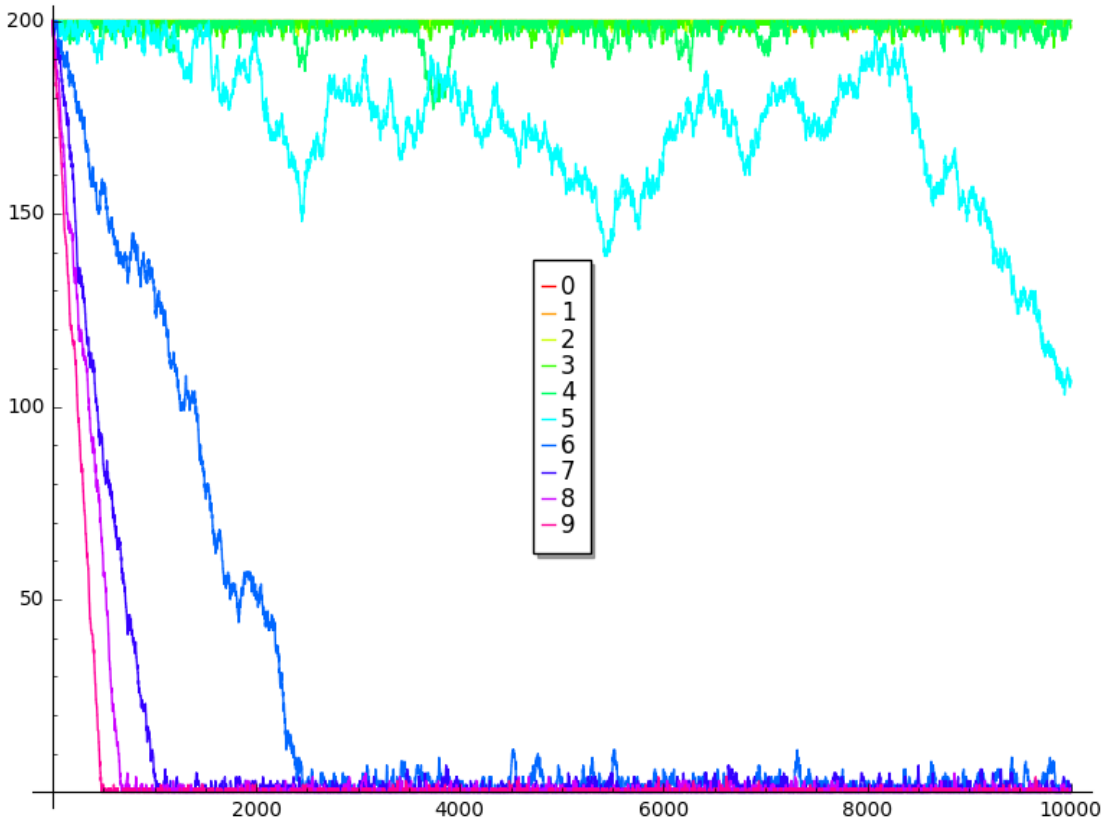
Así, podemos ver fácilmente como el número de pulgas en cada perro se estabiliza en torno a 500 pulgas en cada uno, la mitad.

Apartado D Nos basamos en la segunda versión, donde solo tenemos en cuenta el numero de pulgas de cada perro. Tomamos además Pa y Pb como valores enre cero y 1. Además, usamos un n menor, 200, para ver más rápidamente si tienden a estabilizarse o no.

```
In [2]: def siguiente2(A, B, Pa, Pb):
            if random() <= 0.5:
                #Perro A
```

```
            if random() <= Pa and A > 0:
                #Salta al otro perro
                A -= 1
                B += 1
        else:
            #Perro B
            if random() <= Pb  and B > 0:
                #Salta al otro perro
                A += 1
                B -= 1

        return A, B

    G = []
    for i in xrange(0, 10):
        L = []
        A = 200
        B = 0
        Pa = i/10
        Pb = (10-i)/10
        for j in xrange(0, 10^4):
            L.append((j, A))
            A, B = siguiente2(A, B, Pa, Pb)
        G.append(line2d(L, hue = i/10, legend_label=str(i)))

    sum(G)
```

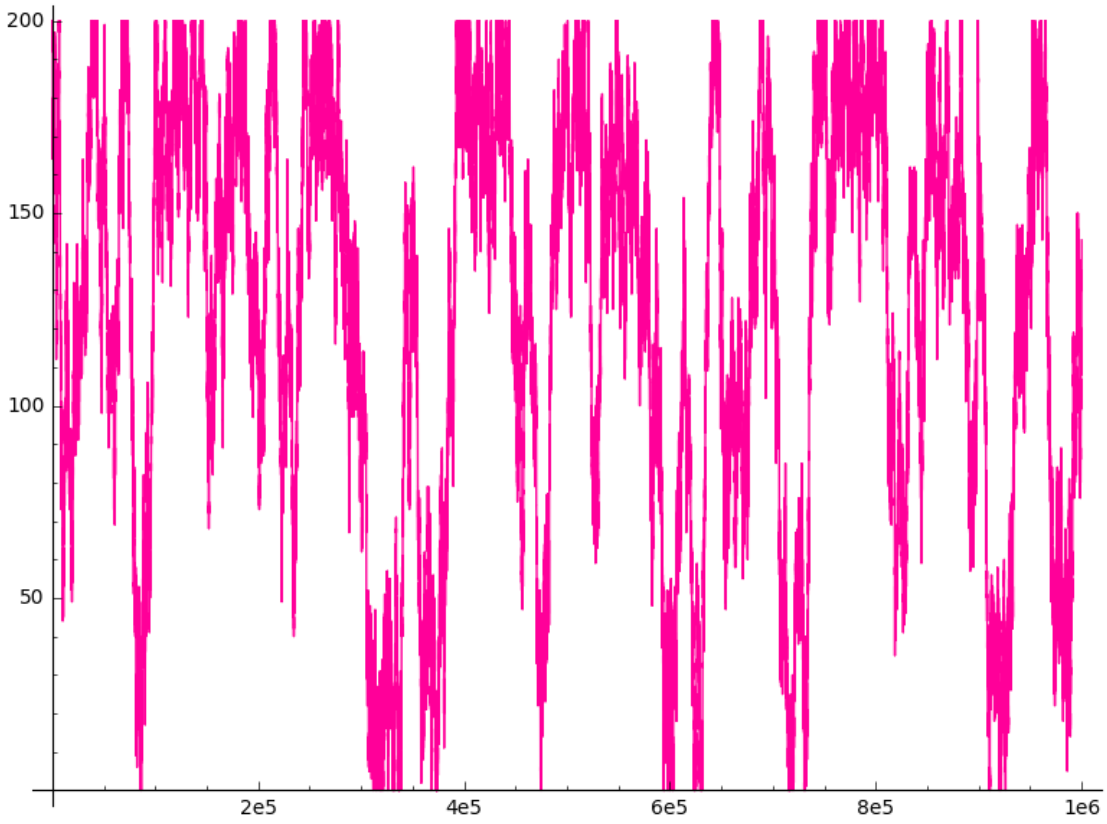
Out[2]:



Así, para $i = 5$ es donde parece que la función no se estabiliza. Vamos a representarla ahora, un número de repeticiones mayor, para ver si efectivamente no se estabiliza.

```
In [6]: L = []
        A = 200
        B = 0
        for j in xrange(0, 10^6):
            L.append((j, A))
            A, B = siguiente2(A, B, 0.5, 0.5)
        line2d(L, hue = i/10)
```

Out[6]:



Efectivamente, es muy sencillo ver cómo, cuando ambos tienen la misma probabilidad de que las pulgas pasen de uno a otro, no se llegan a estabilizar.

Quedaría ver el caso en el que $Pa + Pb \neq 1$, sin embargo ya hemos visto que para algunos valores de Pa y Pb no se estabilizan, mientras que para otros sí.

Ejercicio 7 Supongamos una baraja de cartas inicialmente ordenada. Conocemos, con absoluta certeza, dónde está cada carta de la baraja. Cuando empezamos a barajar la incertidumbre va aumentando, y queremos estudiar este proceso de "aumento de incertidumbre". Representamos el estado inicial de la baraja de n cartas mediante la lista $srange(n)$, y cualquier estado mediante una reordenación de esta lista.

A) Define una función $barajar(L)$ que reciba una lista, que representa un estado de la baraja, y devuelva la lista L barajada una única vez de acuerdo al siguiente algoritmo:

   1) Producimos, aleatoriamente con igual probabilidad de cero o uno, una lista LA de ceros y unos de longitud n. Sea k el número de ceros que hemos obtenido.
   2) Llamemos $L1$ a la sublista de L formada por los primeros k elementos de L y $L2$ al resto de L.
   3) Sea $L3$, inicialmente una lista vacía, la lista en la que vamos a guardar la reordenación de la lista L. Recorremos la lista LA y cada vez que encontramos un cero pasamos el primer elemento de la lista $L1$ a $L3$, y cada vez que encontramos un 1 pasamos el primer elemento de $L2$ a $L3$.

4) Al terminar devolvemos $L3$. Se llama a esta forma de barajar *riffle shuffle*.

B) £Qué es un estado del sistema en este caso? Podríamos decir que es una permutación de la lista $srange(n)$, pero en nuestra situación de incertidumbre acerca de la ordenación obtenida después de barajar es mejor decir que *un estado del sistema es una distribución de probabilidad sobre el conjunto de las* $n!$ *reordenaciones de la lista*, y, más concretamente, *una lista de* $n!$ *reales positivos* p_i *tales que su suma total es* 1.

En el estado inicial de la baraja podemos suponer que el estado es $[1, 0, 0, \ldots, 0, 0]$, y cualquier estado sobre el que no hay incertidumbre tendría un 1 en algún lugar y el resto serían ceros. En cambio, si barajamos, una o varias veces, y no miramos la ordenación obtenida el estado es una distribución de probabilidad que nos dice cómo de probable es cada reordenación.

C) £Cómo se mide la cantidad de información? Claude Shannon, creador en 1948 de la teoría de la información, llegó a una solución que ya era conocida en física con el nombre de *entropía*. Concretamente, si L es una lista que representa una distribución de probablidad sobre N objetos, definió su cantidad promedio de información en bits $I(L)$ en la forma

$$I(L) = -\sum_{p_i \in L} p_i log_2(p_i),$$

de forma que un estado seguro (sin incertidumbre alguna) tiene una cantidad de información nula.

D) Como $n!$ crece bestialmente con n, no es posible realizar el experimento que propongo con una baraja real de 52 cartas. Tomamos entonces $n = 7$, y se trata de averiguar, mediante un experimento adecuadamente planeado, cuántas veces debemos barajar hasta que el contenido de información del estado resultante (la incertidumbre) es máximo. En ese momento podremos decir que *"la baraja ha sido bien barajada"*.

```
In [47]: def barajar(L):
             LA = []
             L3 = []
             k = 0
             for i in range(len(L)):
                 if random() <= 0.5:
                     LA.append(0)
                     k += 1
                 else:
                     LA.append(1)
             L1 = L[:k]
             L2 = L[k:]
             for n in LA:
                 if n == 0:
                     L3.append(L1[0])
                     L1 = L1[1:]
                 else:
                     L3.append(L2[0])
                     L2 = L2[1:]
             return L3
```

```
In [51]: def ent_termino(n):
             return n*log(n, base = 2)

         def entropia(L):
             return -sum(map(ent_termino, L)).n()

         for _ in xrange(10):
             print entropia(barajar([i+1 for i in range(8)]))
```

```
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
-85.5257874353304
```

Ejercicio 8 Hemos visto que es posible estimar el área de un disco unidad *lanzando dardos al cuadrado unidad y contando los que caen dentro del disco*. El mismo procedimiento permite, en principio, estimar el volumen de la hiperesfera $\mathbb{B}_n$ de radio 1 en $\mathbb{R}^n$ (i.e. el conjunto de puntos de coordenadas $(x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ tales que $x_1^2 + x_2^2 + \cdots + x_n^2 \leq 1$), pero los volúmenes que vamos obteniendo al incrementar n son cada vez más pequeños y bastante pronto obtenemos cero como respuesta.

Ésto se debe a que, cuando n es grande, muy pocos dardos caen dentro de la hiperesfera debido a que ocupa muy poco volumen con respecto al volumen, 2^n, del hipercubo unidad $[-1, 1]^n$. En este ejercicio vemos una manera distinta de *lanzar dardos*, que nos va a asegurar que suficientes caen dentro de la hiperesfera. Denotemos por $V(n)$ el volumen de $\mathbb{B}_n$, que es lo que queremos calcular.

Descripción del método

1) El *truco básico* consiste en considerar la hiperesfera de dimensión n dentro del hipercilindro $\mathbb{C}_n := \mathbb{B}_{n-1} \times [-1, 1]$, en lugar de dentro del hipercubo $[-1, 1]^n$. El volumen del hipercilindro es, gracias al teorema de Fubini del cálculo integral, igual al volumen de $\mathbb{B}_{n-1}$ multiplicado por dos.

2) En segundo lugar observamos que para obtener puntos aleatorios en el hipercilindro $\mathbb{C}_n$ basta generar puntos aleatorios $(x_1, x_2, \ldots, x_{n-1}) \in \mathbb{B}_{n-1}$ en la hiperesfera de dimensión $n-1$ y para cada uno producir un real aleatorio x_n en el intervalo $[-1, 1]$.

3) Para que esto funcione debemos encontrar una **manera eficiente** de generar puntos aleatorios en la hiperesfera $\mathbb{B}_{n-1}$, y el método que vamos a usar se llama de *cadena de Markov* (una especie de *paseo aleatorio generalizado*):

A) El primer punto de la cadena $\mathbf{x}_0$ es, por ejemplo, el origen de coordenadas.

B) Para cada punto $\mathbf{x}_t = (x_1, x_2, \ldots, x_{n-1}) \in \mathbb{B}_{n-1}$ obtenemos un nuevo punto $\mathbf{x}_{t+1}$ eligiendo una coordenada al azar, supongamos que hemos obtenido x_i, y un real aleatorio Δ en el intervalo $[-\delta, \delta]$. Entonces, cambiamos x_i por $x_i + \Delta$ en $\mathbf{x}$ y si todavía estamos dentro de la hiperesfera ese es el nuevo punto $\mathbf{x}_{t+1}$ en la cadena. Si al hacer el cambio nos vamos fuera de la hiperesfera dejamos $\mathbf{x}_{t+1} := \mathbf{x}_t$.

C) La cadena $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_N\}$ es entonces un conjunto de $N+1$ puntos, todos en la hiperesfera $\mathbb{B}_{n-1}$, y se puede demostrar que, para N suficientemente grande, están uniformemente distribuidos en la hiperesfera.

4) Después de estos preparativos podemos ya plantear el cálculo del volumen $V(n)$ de la hiperesfera $\mathbb{B}_n$:

Generamos un número muy grande N de puntos en la hiperesfera de dimensión $n-1$, usando el apartado 3), y para cada uno de ellos vamos calculando un punto en el hipercilindro, como en el apartado 2). Si el punto obtenido en el hipercilindro cae dentro de la hiperesfera de dimensión n, $\mathbb{B}_n$, incrementamos un contador, al que por ejemplo hemos llamado *dentro*.

La fracción $dentro/N$, calculada después de haber generado N puntos del hipercilindro, es aproximadamente igual al cociente de volúmenes $V(n)/(2V(n-1))$ (volumen de la hiperesfera dividido por volumen del hipercilindro).

Entonces, podemos reducir el cálculo de $V(n)$ al de $V(n-1)$, el de $V(n-1)$ al de $V(n-2)$, etc., hasta llegar al volumen de la hiperesfera en dimensión 1 que es igual a dos.

Ejercicios 2.1) Define las funciones necesarias para implementar esta forma, *Monte Carlo con cadenas de Markov*, de calcular $V(n)$.

2.2) El volumen exacto de una hiperesfera se puede calcular mediante integrales, y se obtiene una fórmula que se indica en la celda siguiente:

```
In [23]: def Vol_exacto(dim):
             return (pi**(dim/2.0)/gamma(dim/2.0+1.0)).n()

         Vol_exacto(3)

Out[23]: 4.18879020478639

In [25]: def cuadrado(n):
             return n^2

         def pertenece_esfera(C):
             return sqrt(sum(map(cuadrado, C))).n() <= 1

         def puntos_hiperesfera(n, N):
             P = []
             x0 = [0 for _ in xrange(n-1)]
             for i in xrange(N):
                 xi = randint(0, n - 2)
                 delta = random() - 0.5
                 x1 = list(x0)
                 x1[xi] += delta
```

```
                if pertenece_esfera(x1):
                    P.append(x1)
                    x0 = x1
                else:
                    P.append(x0)
            return P

        def V(n, N):
            if n == 1:
                return 2

            dentro = 0
            P = puntos_hiperesfera(n, N)
            for p in P:
                #Añadimos una coordenada mas que va desde -1 hasta 1
                p.append((random() - 0.5)* 2)
                if pertenece_esfera(p):
                    dentro += 1
            return 2*V(n-1, N)*(dentro/N).n()

        %time V(2, 10^6)
```

```
CPU times: user 1min 22s, sys: 1.47 s, total: 1min 23s
Wall time: 1min 22s
```

```
Out[25]: 2.98005200000000
```

De esta forma, tenemos programa la función, sin embargo, dada la lentitud de python es completamente ineficiente.

Ejercicio 9 Supongamos que para cubrir un puesto de trabajo hay n candidatos que deben ser entrevistados para decidir cuál de ellos obtendrá el puesto. Las condiciones del problema son las siguientes:

El número n de candidatos es conocido al comenzar el proceso de selección.

Si entrevistáramos a todos los candidatos los podríamos ordenar de mejor a peor sin empates.

Los candidatos son entrevistados de uno en uno y en orden aleatorio, con cada una de las $n!$ posibles ordenaciones elegida con probabilidad $1/n!$.

Después de cada entrevista el candidato es aceptado o rechazado, y esta decisión es irrevocable.

La estrategia que utilizamos consiste en entrevistar a un cierto número $r < n$ de candidatos y elegir al siguiente entrevistado que es mejor que los r primeros. Si no existe elegimos al último aunque no será el mejor. Se puede demostrar que esta estrategia es óptima.

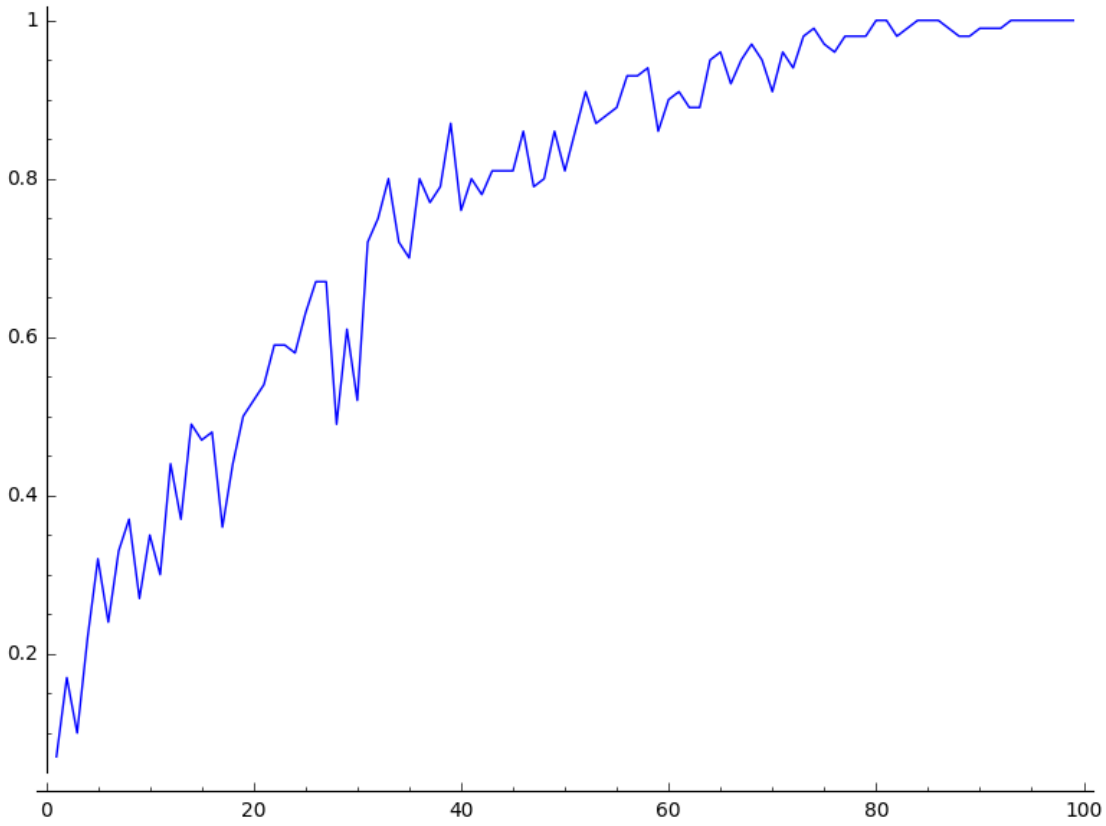
Define una función $probabilidad(n, N)$ que nos devuelva el valor de r que hace máxima la probabilidad de elegir al mejor candidato cuando hay n candidatos y la probabilidad máxima obtenida. El segundo parámetro N es el el número de vueltas del bucle que usamos para calcular (experimentalmente) las probabilidades, y debe ser suficientemente grande para que las probabilidades tengan, al menos, un par de decimales "correctos" y no tan grande que el programa tarde demasiado. Cuando n tiende a infinito, £qué límite parece tener la probabilidad máxima?

Representamos cada repetición del experimento "se presentan n candidatos en orden aleatorio" como una permutación aleatoria de los enteros $1, 2, 3, \ldots, n$ y suponemos que n es el mejor y 1 el peor.

```
In [38]: def probabilidad(n, N):
             L = [i+1 for i in range(n)]
             Resultados = []
             for r in range(1, n):
                 nMax = 0
                 for _ in range(N):
                     shuffle(L)
                     L2 = L[:r+1]
                     #Entrevistamos a los r primeros y cogemos al mejor
                     c = max(L2)
                     #Elegimos el siguiente mayor
                     for c2 in L[r+1:]:
                         if c2 > c:
                             c = c2
                             break;
                     if c == n:
                         nMax += 1
                 Resultados.append((r, (nMax/N).n()))
             return Resultados

         %time line2d(probabilidad(100, 100))
```

```
CPU times: user 520 ms, sys: 0 ns, total: 520 ms
Wall time: 518 ms
```

```
Out[38]:
```



Creo que no he entendido correctamente el ejercicio, pues la respuesta me parece demasiado obvia, sin embargo, he programado el algoritmo siguiente:

- Tomamos una permutación aleatoria de la lista, y entrevistamos a r personas.
- De estas r personas cogemos al mejor.
- Una vez tenemos al mejor de los r entrevistados, seguimos entrevistando hasta que encontramos a uno que es mejor que él.
  - Entonces escojemos a este como el mejor, aunque puede no serlo.
  - En caso de que no haya ninguno más mejor que él, nos quedamos con él.

Así, es bastante obvio que la probabilidad aumenta a medida que aumenta la r, pues entrevistamos a más personas.

Ejercicio 10 Consideramos un tablero infinito (en la práctica será muy grande porque infinito no lo podemos hacer en el ordenador) bidimensional que representamos mediante los puntos de coordenadas enteras del cuadrado $T_N := [0, 2N] \times [0, 2N] \subset \mathbb{R}^2$. El estado del tablero es entonces una matriz cuadrada de ceros y unos cuyos índices van de 0 a $2N$. En el instante $t = 0$ la única casilla negra es la central correspondiente al punto (N, N). En cada instante de tiempo t llamamos X_t al conjunto de todas las casillas negras en ese instante y V_t al conjunto de todas las casillas blancas que comparten un lado con una casilla negra. Obtenemos X_{t+1} eligiendo al azar una casilla de V_t y cambiando su color a negro.

Nos interesa estudiar el aspecto de X_t para t muy grande (por ejemplo, $t = 10^5, 10^6$). Define una función *tablero*(t, N) (teniendo cuidado de no saturar la RAM) que devuelva un estado X_t del tablero obtenido mediante las reglas indicadas. Representa gráficamente el tablero devuelto por la función. El tamaño del tablero debe ser suficientemente grande para que quede una zona totalmente blanca cerca del borde del tablero. Compara las gráficas para $t = 100, 200, 300, 1000$ con las que se obtienen para t muy grande, £qué observas?

```
In [2]: def matriz(N):
            M = []
            for _ in xrange(2*N+1):
                M.append([0 for _ in xrange(2*N+1)])
            M[N][N] = 1
            return M

        matriz(1)
```

```
Out[2]: [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
```

```
In [26]: def tablero(t, N):
             M = matriz(N)
             for hu in xrange(t + 1):
                 Vt = []
                 # Rellenamos Vt
                 for fila in xrange(len(M)):
                     j = 0
                     nj = M[fila][j::].count(1)
                     while nj > 0:
                         #Incluimos todos en el conjunto Vt
                         pos = M[fila][j::].index(1)
                         if M[(fila - 1)%(2*N+1)][pos] == 0:
                             Vt.append(((fila - 1)%(2*N+1), pos))

                         if M[(fila + 1)%(2*N+1)][pos] == 0:
                             Vt.append(((fila + 1)%(2*N+1), pos))

                         if M[fila][(pos - 1)%(2*N+1)] == 0:
                             Vt.append((fila, (pos - 1)%(2*N+1)))

                         if M[fila][(pos - 1)%(2*N+1)] == 0:
                             Vt.append((fila, (pos + 1)%(2*N+1)))

                         nj -= 1
                         j = M[fila][j:].index(1) + 1
                 #Elegimos uno
                 if len(Vt) != 0:
                     n = randint(0, len(Vt) - 1)
                     M[Vt[n][0]][Vt[n][1]] = 1
             return M
```

```
%time tablero(4, 10)
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 1.23 ms
```

```
Out[26]: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Tras hacer esta primera función y darme cuenta de que, especialmente para matrices y t grandes era tremendamente lenta, hice la siguiente, mucho mas rápida y eficiente.

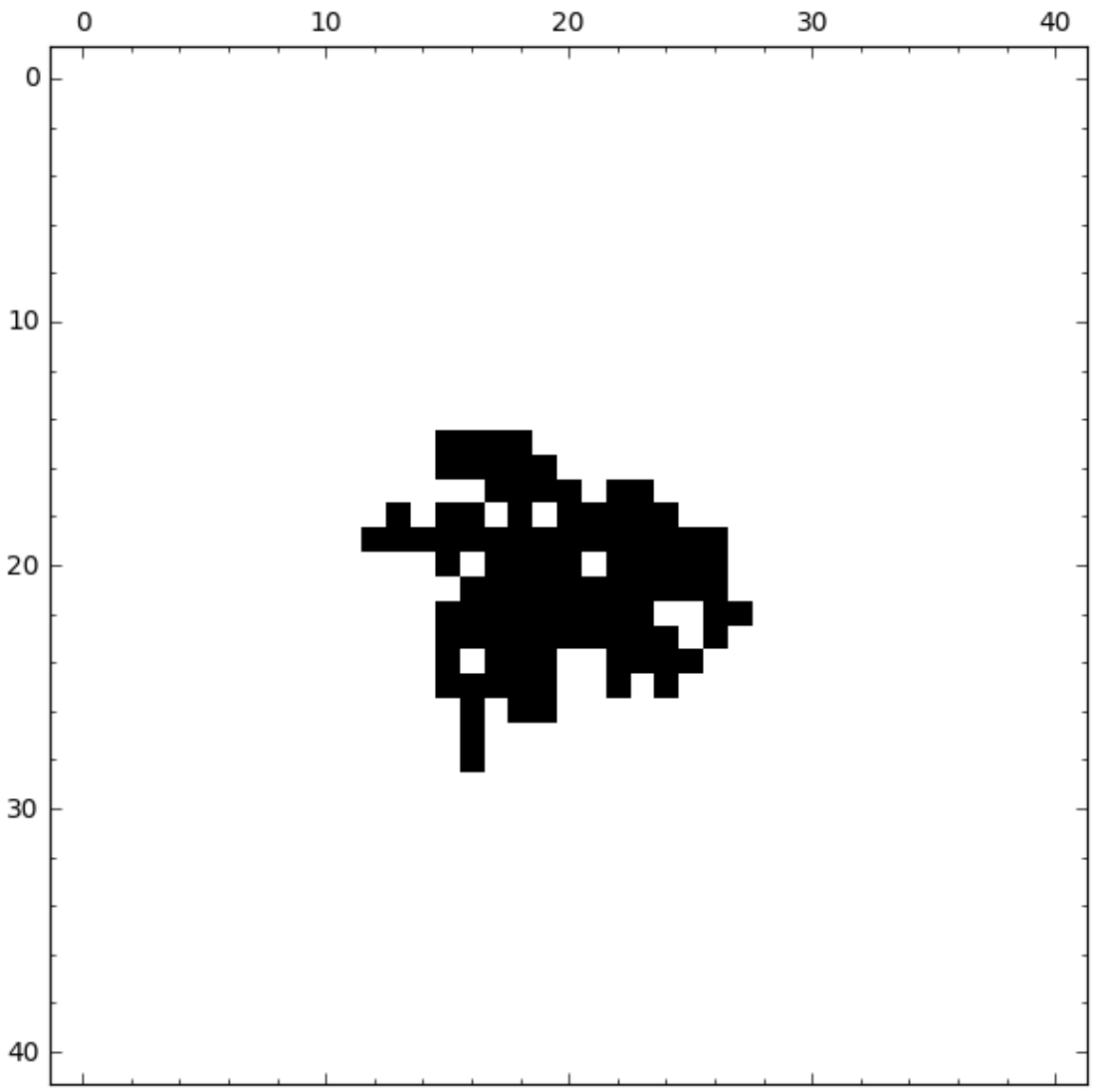
```
In [17]: # Dada la casilla C, devuelve las coordenadas
         # de las casillas blancas de su alrededor
         def VtCasilla(C, N, M):
             v = []
             a = C[0]
             b = C[1]
             if M[(a-1)%(2*N+1)][b] == 0:
                 v.append(((a-1)%(2*N+1), b))

             if M[(a+1)%(2*N+1)][b] == 0:
                 v.append(((a+1)%(2*N+1), b))

             if M[a][(b+1)%(2*N+1)] == 0:
                 v.append((a, (b+1)%(2*N+1)))

             if M[a][(b-1)%(2*N+1)] == 0:
                 v.append((a, (b-1)%(2*N+1)))
```

```
            return v


        def tablero2(t, N):
            M = matriz(N)
            Vt = VtCasilla((N,N), N, M)
            for hu in xrange(t + 1):
                #Elegimos una casilla y la cambiamos a negro
                if len(Vt) != 0:
                    n = randint(0, len(Vt) - 1)
                    M[Vt[n][0]][Vt[n][1]] = 1

                    #Actualizamos ahora Vt
                    L = VtCasilla(Vt[n], N, M)
                    for C in L:
                        if C not in Vt:
                            Vt.append(C)
                    Vt.remove(Vt[n])
                else:
                    #Todas las casillas estan blancas ya
                    return M

            return M
```

```
In [8]: %time t = tablero2(100, 20)
        matrix_plot(t)
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 3.46 ms
```

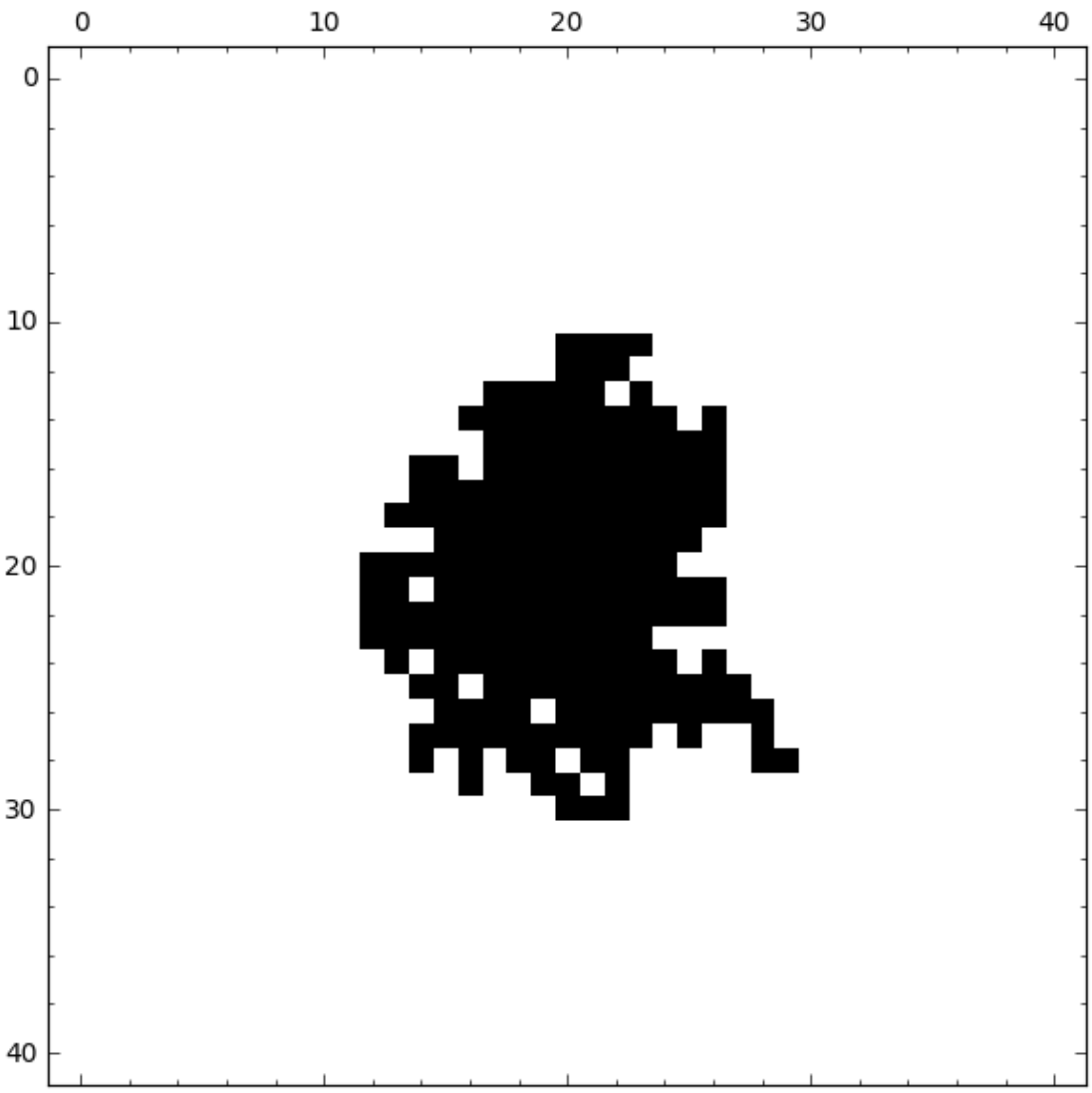
Out[8]:




```
In [9]: %time t= tablero2(200, 20)
        matrix_plot(t)
```

```
CPU times: user 12 ms, sys: 0 ns, total: 12 ms
Wall time: 8.18 ms
```

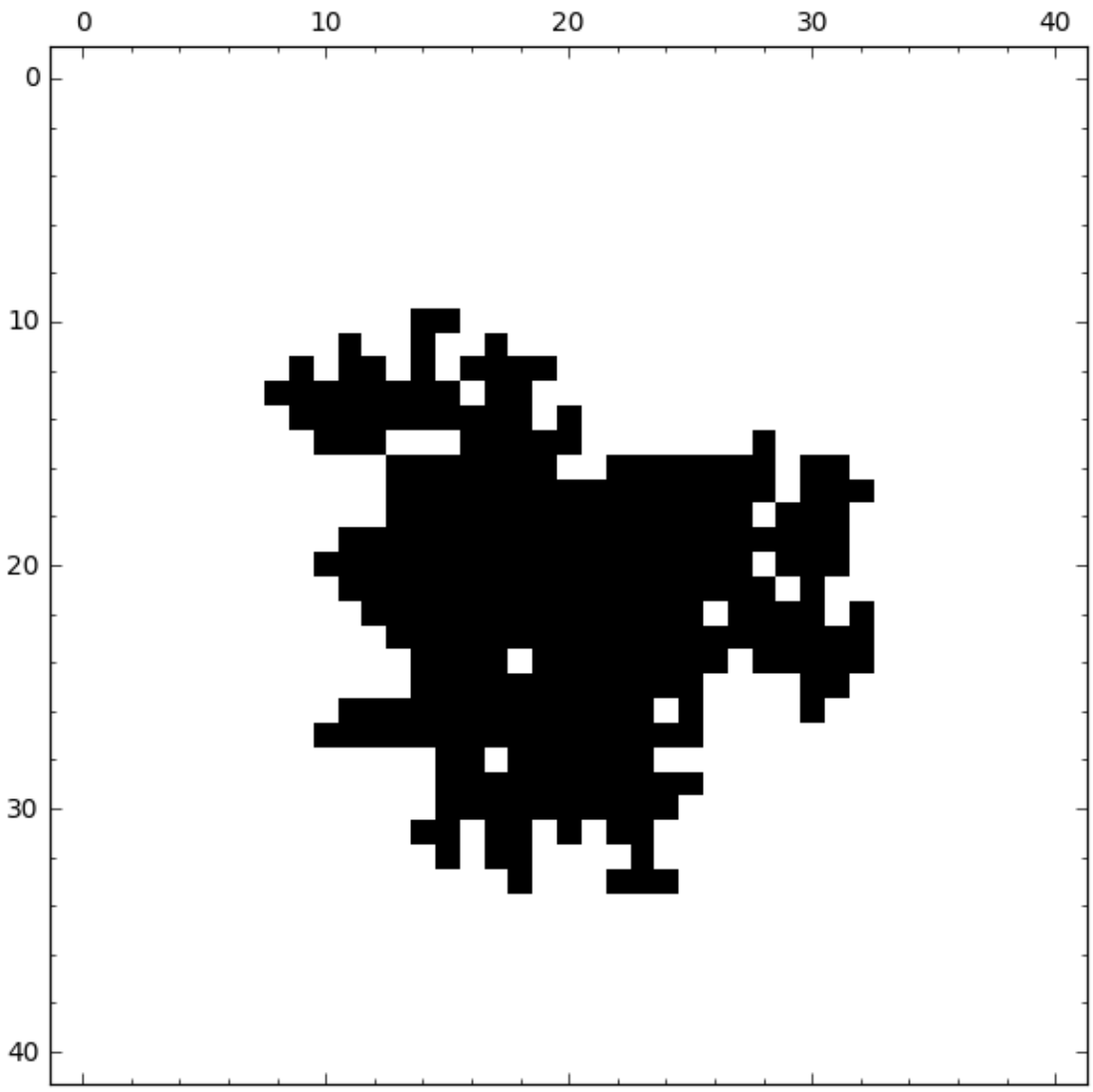
Out[9]:




```
In [5]: %time t = tablero2(300, 20)
        matrix_plot(t)
```

```
CPU times: user 4 ms, sys: 4 ms, total: 8 ms
Wall time: 6 ms
```

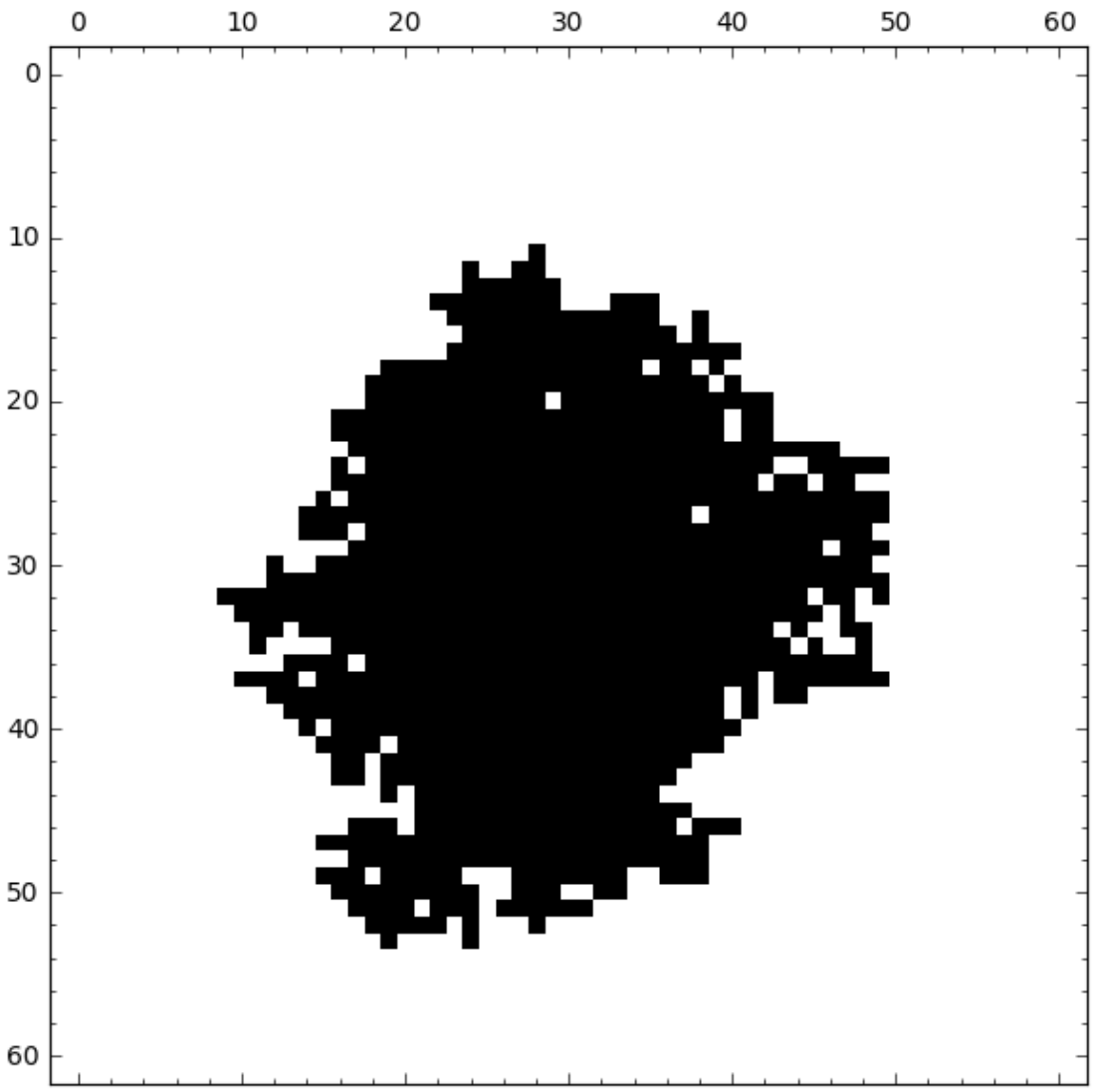
Out[5]:




```
In [7]: %time t = tablero2(1000, 30)
        matrix_plot(t)
```

```
CPU times: user 52 ms, sys: 0 ns, total: 52 ms
Wall time: 42.3 ms
```

Out[7]:

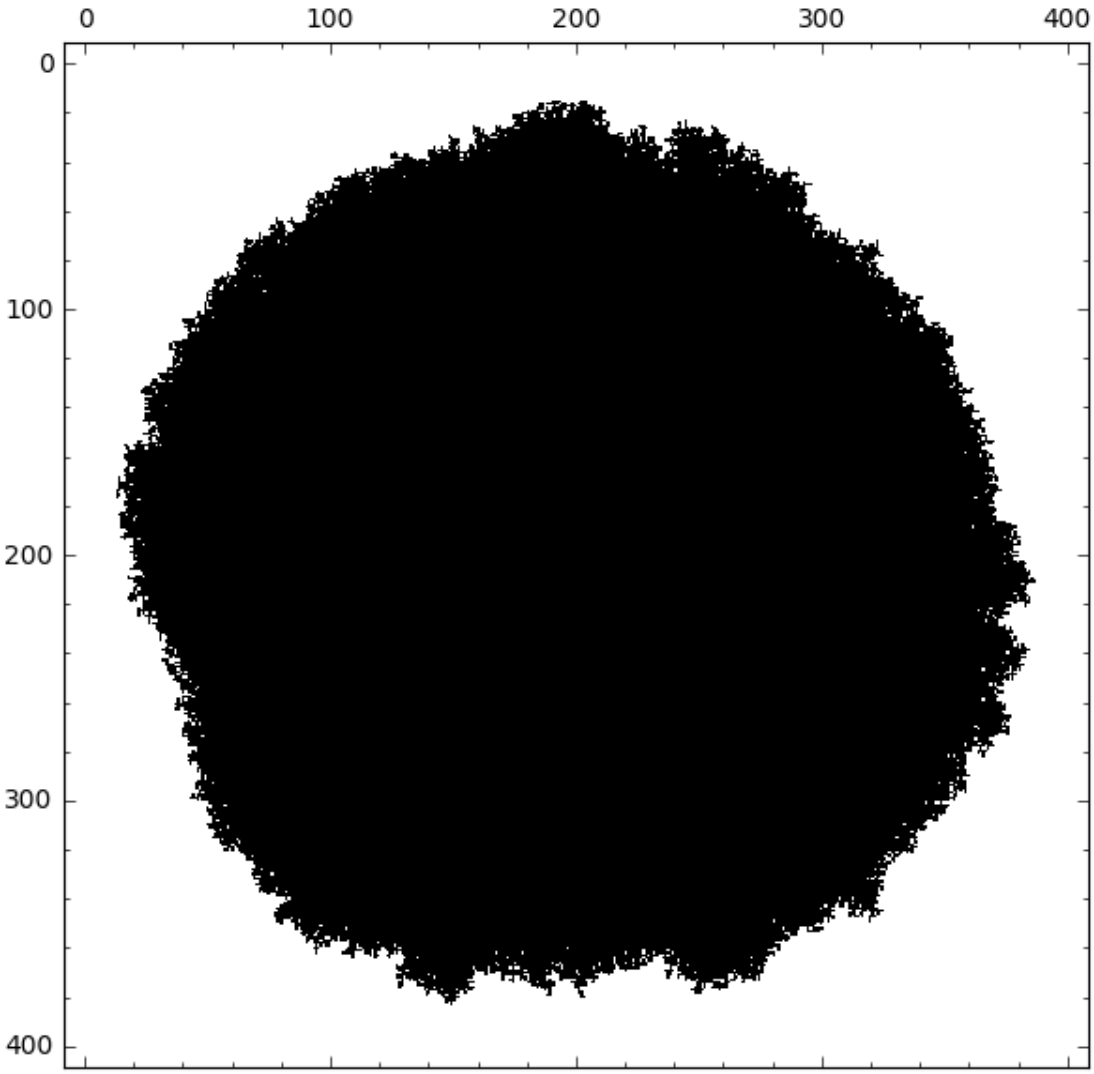


Se puede ver así como se va rellenando poco a poco el centro completamente, mientras que la zona exterior permanece blanca en su mayor parte. Podemos ver así como la parte central se va rellenando poco a poco y uniformemente, mientras ue los bordes continuan siendo blancos en su mayor parte, hasta que al llegar a un número de iteraciones enorme como es 10^5, la matriz es completamente negra.

Teniendo en cuenta que en cada iteración vamos a añadir un uno, para que tras 10^5 iteraciones quede algun cuadrado en blanco, el numero de elementos de la matriz tiene que ser mayor que 10^5, y por tanto, $4 * N^2 > 10^5$, con lo que N tiene que ser mayor que 159.

```
In [16]: %time T = tablero2(10^5, 200)
         matrix_plot(T)

CPU times: user 15 s, sys: 80 ms, total: 15.1 s
Wall time: 15 s
```

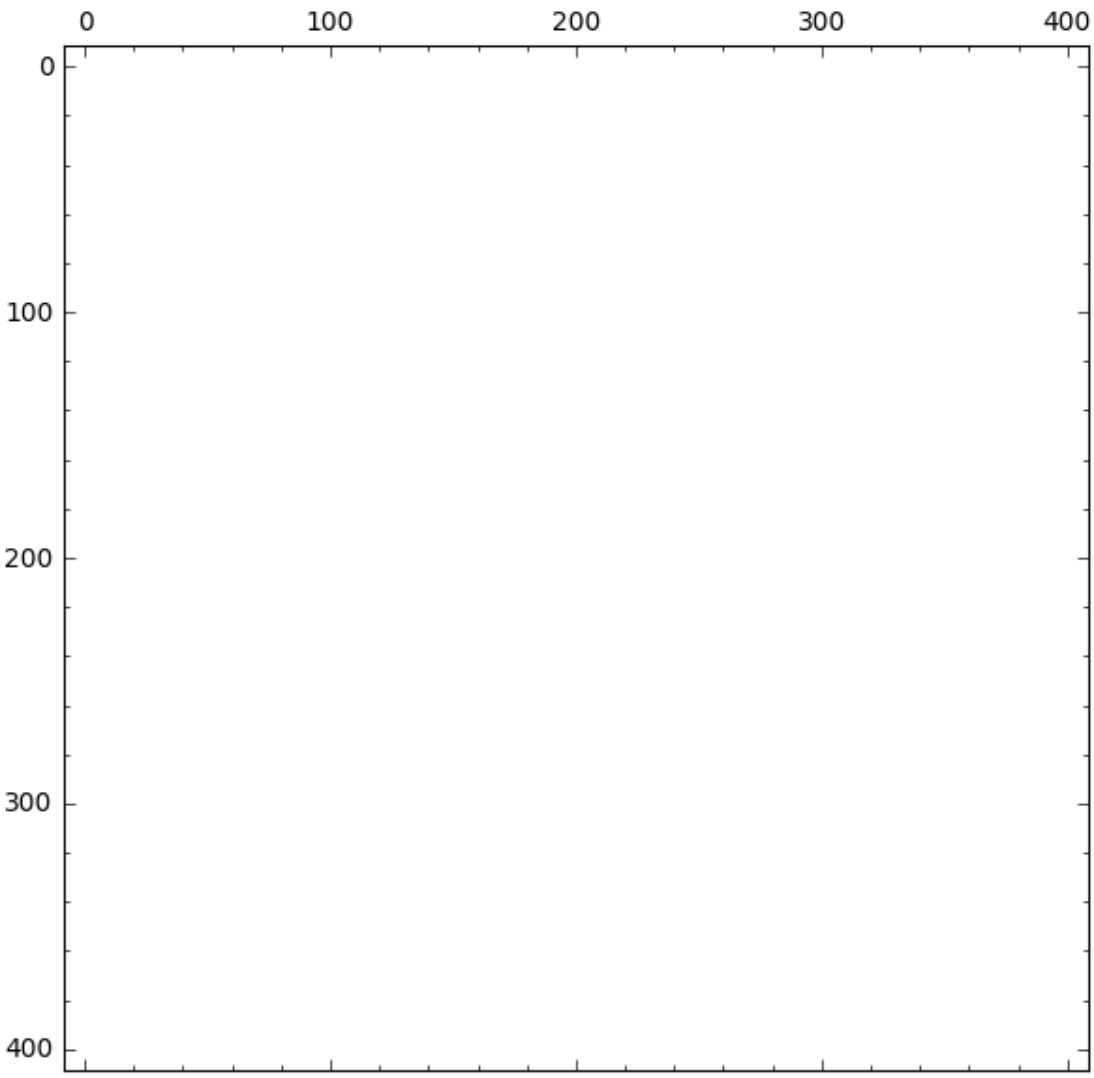
Out[16]:



In [18]:
```
%time T = tablero2(10^6, 200)
matrix_plot(T)
```

```
CPU times: user 23.5 s, sys: 132 ms, total: 23.6 s
Wall time: 23.3 s
```

Out[18]:



In [ ]: