Generación de variables aleatorias discretas

Existen diversos métodos para generar variables aleatorias discretas a partir de un generador de v.a. uniformes. En particular veremos los siguientes:

1. Método de la **Transformada Inversa**.
2. Método de **aceptación y rechazo**, o simplemente **método de rechazo**.

Cada uno de estos métodos puede ser mejorado u optimizado según cuál es la variable aleatoria en particular que se desea simular. En todos los casos se asume la existencia de un generador de números aleatorios, es decir, de valores de una variable aleatoria uniforme en $(0, 1)$.

1. Método de la transformada inversa

Consideremos una variable aleatoria discreta X, con función de probabilidad de masa dada por:

$$P(X = x_j) = p_j, \qquad j = 0, 1, \ldots, \qquad 0 < p_j < 1,$$

donde los valores $\{x_n\}$ de la variable están ordenados en forma creciente. Esto es, si $i < j$ entonces $x_i < x_j$.

La función de distribución acumulada de X, F_X, satisface:

$$F_X(x) = \begin{cases} 0 & x < x_0 \\ p_0 & x_0 \leq x < x_1 \\ p_0 + p_1 & x_1 \leq x < x_2 \\ \vdots & \vdots \\ p_0 + p_1 + \cdots + p_{n-1} & x_{n-1} \leq x < x_n \\ \vdots & \vdots \end{cases},$$

y notemos que

$$p_0 < p_0 + p_1 < p_0 + p_1 + p_2 < \ldots < p_0 + p_1 + p_2 + \cdots + p_n < \ldots.$$

El método de la transformada inversa lleva este nombre porque se basa en la inversa de F_X. Es claro que si X es una v.a. discreta, entonces F_X no es inyectiva y por lo tanto tampoco es inversible, pero podemos establecer una correspondencia biunívoca entre ciertos subintervalos de $(0, 1)$ y los valores de la variable aleatoria X a través de la función de distribución F_X. La correspondencia es la siguiente:

$$\begin{aligned} x_0 &\longrightarrow (0, p_0) \\ x_1 &\longrightarrow [p_0, p_0 + p_1) \\ x_2 &\longrightarrow [p_0 + p_1, p_0 + p_1 + p_2) \\ &\vdots \end{aligned}$$

Por otro lado, si consideremos una variable aleatoria uniforme $U \sim U(0, 1)$, entonces la probabilidad que U tome valores en el intervalo

$$[p_0 + \cdots + p_{n-1}, p_0 + \cdots + p_{n-1} + p_n)$$

es justamente p_n, que es la probabilidad que la variable aleatoria X tome el valor x_n. Esto es:

$$\begin{aligned} P(0 < U < p_0) = p_0 &= P(X = x_0) \\ P(p_0 \leq U < p_0 + p_1) = p_1 &= P(X = x_1) \\ P(p_0 + p_1 \leq U < p_0 + p_1 + p_2) = p_2 &= P(X = x_2) \\ \vdots &= \vdots \\ P(p_0 + \cdots + p_{n-1} \leq U < p_0 + p_1 + p_2 + \cdots + p_n) = p_n &= P(X = x_n) \end{aligned}$$

Así, el método de la transformada inversa propone generar valores de X generando una v.a. uniforme, y según a qué intervalo pertenece U es el valor de X que se genera. Es decir: si $U \in (0, p_0)$, se genera x_0. Si $U \in (p_0, p_0 + p_1)$, se genera x_1, y así siguiendo.

El algoritmo general sería como el siguiente:

```
def discretaX(p, x):
    U = random()
    i=0
    F=p[0]
    while U >= F:
        i = i+1
        F = F+p[i]
    return x[i]
```

Ejemplo 1.1. Si consideramos la variable aleatoria X que toma valores en el conjunto $\{1, 2, 3, 4\}$, con las probabilidades:

$$p_1 = 0.20, \quad p_2 = 0.15, \quad p_3 = 0.25, \quad p_4 = 0.40,$$

el algoritmo, en forma detallada, es el siguiente:

```
def TInversa(u):
    if u<0.20:
        return 1
    elif u<0.35:
        return 2
    elif u<0.60:
        return 3
    else:
        return 4
```

Una primera mejora que puede hacerse a este algoritmo, y en general vale para cualquier generador de v.a. discretas, es ordenar las probabilidades de mayor a menor, manteniendo la correspondencia entre la longitud del intervalo y la probabilidad que ocurra cada valor. Esto reduce el número de comparaciones, puesto que cada condicional ($U < F_X$) es más probable que sea aceptado cuanto mayor sea F_X. Así, en el ejemplo anterior, las probabilidades se ordenarían de la siguiente forma:

$$p_4 = 0.40, \qquad p_3 = 0.25, \quad p_1 = 0.20, \quad p_2 = 0.15,$$

y el algoritmo modificado sería como sigue:

```
def TInversa(u):
    if u<0.40:
        return 4
    elif u<0.65:
        return 3
    elif u<0.85:
        return 1
    else:
        return 2
```

Así, el número de comparaciones esperado en el primer caso es

$$1 \cdot 0.20 + 2 \cdot 0.15 + 3 \cdot 0.25 + 4 \cdot 0.40 = 2.85$$

mientras que en el segundo es:

$$1 \cdot 0.40 + 2 \cdot 0.25 + 3 \cdot 0.20 + 4 \cdot 0.15 = 2.10$$

1.1. Generación de una variable aleatoria uniforme discreta

Si $X \sim \mathcal{U}[1, n]$, entonces $p_1 = p_2 = \cdots = p_n = \dfrac{1}{n}$. Si se aplica el método de la transformada inversa, el algoritmo sería como el siguiente:

```
def udiscreta(n):
    u=random()
    F=1/n
    x=1
    while u>=F:
        F+=1/n
        x+=1
    return x
```

Básicamente, el algoritmo recorre los subintervalos de longitud $1/n$ hasta encontrar aquel donde se encuentra el valor de u. Ahora bien, notemos entonces que el algoritmo arrojará el valor $X = j$ si y sólo si u cae en el j-ésimo subintervalo:

$$X = j \qquad \text{si y sólo si} \qquad \frac{j-1}{n} \leq U < \frac{j}{n}$$

Ahora bien,

$$U \in \left[\frac{j-1}{n}, \frac{j}{n}\right) \quad \text{si y sólo si} \quad nU \in [j-1, j) \quad \text{si y sólo si} \quad \lfloor nU \rfloor + 1 = j,$$

donde $\lfloor x \rfloor$ denota la parte entera inferior de x. Así, el algoritmo puede reescribirse como:

```
def udiscreta(n):
    u=random()
    return int(n*u)+1
```

Este algoritmo puede extenderse al caso de generación de una variable aleatoria discreta uniforme con valores enteros en el intervalo $[m, k]$, esto es, $U \sim U[m, k]$. Notemos que X toma $k-m+1$ valores:

$$m, \quad m+1, \quad m+2, \quad \ldots \quad k = m + (k - m).$$

Por lo tanto es suficiente generar una variable uniforme con valores en $[1, k - m + 1]$ y sumarle $m - 1$ a cada valor.

```
def udiscreta(m,k):
    u=random()
    return int(u*(k-m+1))+m
```

1.2. Generación de una permutación aleatoria de un conjunto de cardinal N

Una aplicación de la generación de variables aleatorias con distribución uniforme discreta es el de generar **permutaciones aleatorias** en un conjunto de cardinal N. El número de permutaciones de un conjunto de N elementos es $N!$, y el objetivo es poder generar permutaciones **equiprobables**, es decir, cada una con probabilidad $\frac{1}{N!}$ de ocurrencia.

Consideramos un ordenamiento de los elementos de un conjunto A, de cardinal N:

$$(a_1, a_2, \ldots, a_N.)$$

Una primera idea, pero errónea, es intercambiar a_1 con cualquier elemento de A elegido con distribución uniforme, luego intercambiar a_2 con otro elegido uniformemente en A, y así siguiendo. Si bien se obtiene una permutación de A, ocurre que este procedimiento no genera todas las permutaciones con la misma probabilidad. Un algoritmo que ejecute estos pasos tiene N^{N-1} secuencias diferentes de ejecución, pero

$$\frac{N^{N-1}}{N!}$$

no es un número entero, salvo para $N = 2$. Por lo tanto, para $N > 2$ no es posible que todas las permutaciones se obtengan con la misma probabilidad.

En cambio, el siguiente método sí produce todas las permutaciones con igual probabilidad:

PERMUTACIONES(N)

// Paso 1:
Elegir I uniformemente en $\{1, N\}$
Intercambiar a_1 con a_I
// Paso 2:
Elegir I uniformemente en $\{2, N\}$
Intercambiar a_2 con a_I
// ...
// Paso $N-1$
Elegir I uniformemente en $\{N-1, N\}$
Intercambiar a_{N-1} con a_I

Notemos que en los sucesivos pasos se generan uniformes en $[1, N]$, $[2, N]$, $\ldots$, $[N-1, N]$. Si se recorre el arreglo en sentido inverso, comenzando por intercambiar a_N en lugar de a_1, entonces el algoritmo requiere generar uniformes en $[1, N]$, $[1, N-1]$, $\ldots$, $[1, 2]$, que se reducen a calcular una parte entera inferior:

```
def permutacion(N,x): #x=[x[0],x[1],...,x[N-1]]
    for j in range(N,0,-1):
        indice=int(j*random())
        temp=x[j]
        x[j]=x[indice]
        x[indice]=temp
    return x
```

En ciertos casos de muestreo, puede requerirse obtener un subconjunto aleatorio de cierto conjunto de individuos. Es decir, dado un conjunto de N elementos, obtener un subconjunto de r elementos, con $r < N$, pero elegidos aleatoriamente.

Si el algoritmo anterior se ejecuta para $j = N, N-1, \ldots, N-(r-1)$, los últimos r elementos del vector permutado son un subconjunto aleatorio de cardinal r, y en consecuencia los restantes son un subconjunto aleatorio de tamaño $N-r$. Así, para mayor eficiencia, si $r < N/2$ conviene ejecutar el algoritmo r veces y tomar los últimos r elementos, y de lo contrario conviene ejecutarlo $N-r$ veces, y tomar los r primeros.

```
def subcAleatorio(r,A): #subconjunto aleatorio de r elementos
                        # de un conjunto A  de cardinal N
    N=len(A)
    for j in range(N,N-r,-1):
        indice=int(j*random())
        temp=A[j]
        A[j]=A[indice]
        A[indice]=temp
    return A[1:r]
```

Un **método alternativo** para obtener permutaciones aleatorias de un conjunto de cardinal N consiste en generar N números aleatorios: $u_1, \quad u_2, \quad u_3, \quad \ldots, u_N$, y luego ordenarlos, por ejemplo, de menor a mayor:

$$u_{i_1} < u_{i_2} < u_{i_3} < \ldots < u_{i_N}.$$

Así, los índices de los números ya ordenados forman una permutación del conjunto $\{1, 2, \ldots, N\}$:

$$(i_1, i_2, i_3, \ldots, i_N).$$

Sin embargo este método tiene la desventaja de requerir $O(N \log(N))$ comparaciones.

1.3. Cálculo de promedios

Recordemos que el Método de Monte Carlo se basa en dos resultados teóricos fundamentales: la Ley Fuerte de los Grandes Números, y la posibilidad de calcular el valor esperado $E[g(X)]$ a partir de la función de densidad o de probabilidad de masa, según corresponda, de la variable aleatoria X.

Supongamos que se quiere calcular un promedio de una gran cantidad de valores, esto es, se desea calcular:

$$\overline{a} = \frac{1}{N} \sum_{i=1}^{N} a_i = \frac{a_1 + a_2 + \cdots + a_N}{N},$$

donde $N >> 1$.

Notemos que si $X \sim U[1, N]$, y g es una función tal que $g(i) = a_i$, entonces el valor que se desea calcular es justamente $E[g(X)]$:

$$\overline{a} = E[g(X)].$$

Luego, por la Ley Fuerte de los grandes números, se tiene que si X_1, X_2, $\ldots$, X_n, $\ldots$ son v.a. independientes, uniformes en $[1, N]$, entonces

$$\lim_{n\to\infty} \frac{1}{n}(g(X_1) + g(X_2) + \cdots + g(X_n)) = \lim_{n\to\infty} \frac{1}{n}(a_{X_1} + a_{X_2} + \cdots + a_{X_n}) = \overline{a}.$$

Por ejemplo, si se quiere calcular

$$S = \sum_{i=1}^{10000} e^{\frac{1}{i}},$$

tomamos $g(i) = \exp(1/i)$, y estimamos $E[g(X)]$ para $X \sim U[1, 10000]$. Notemos que

$$S = 10000 \cdot \frac{1}{10000} \sum_{i=1}^{10000} e^{\frac{1}{i}} = 10000 \cdot E[g(X)].$$

Usando Monte Carlo, se puede estimar el valor de $\frac{S}{10000}$ con 100 simulaciones de la siguiente forma:

- Generar 100 valores de U, $U = u_i$, $1 \leq i \leq 100$, con distribución uniforme en $[1, 10000]$.
- Calcular para cada uno $\exp(1/u_i)$.
- Sumarlos y dividir por 100.

Como estos pasos llevan a una aproximación de $S/10000$, la estimación de S se obtiene multiplicando por 10000.

$$S = \sum_{i=1}^{10000} e^{\frac{1}{i}} \sim 10000 \cdot \frac{1}{100} \sum_{i=1}^{100} e^{\frac{1}{i}}.$$

1.4. Generación de una variable aleatoria geométrica

Una variable aleatoria geométrica con probabilidad de éxito p tiene una probabilidad de masa dada por:

$$p_i = P(X = i) = pq^{i-1}, \qquad i \geq 1, \qquad q = (1 - p),$$

y la función de distribución acumulada cumple:

$$F_X(j-1) = P(X \leq j-1) = 1 - P(X > j-1) = 1 - q^{j-1}.$$

El método de la transformada inversa asigna entonces el valor $X = j$ si $U \in [1 - q^{j-1}, 1 - q^j)$. Analicemos esta propiedad:

$$1 - q^{j-1} \leq U < 1 - q^j \qquad \text{si y sólo si} \qquad q^j < 1 - U \leq q^{j-1}.$$

Dado que $0 < q < 1$, las potencias de q son decrecientes. Por lo tanto la propiedad anterior equivale a encontrar el menor j tal que $q^j < 1 - U$.

$$X = \text{mín}\{j \ : \ q^j < 1 - U\}$$

Para determinar j, aplicamos el logaritmo, y por lo tanto

$$X = j \qquad \text{si y sólo si} \qquad j = \text{mín}\{k \mid k\log(q) < \log(1-U)\}.$$

Ahora, si $U \sim U(0,1)$, también lo es $V = 1 - U$. Además $\log(q) < 0$, por lo tanto la propiedad se puede escribir como:

$$X = j \qquad \text{si y sólo si} \qquad j = \text{mín}\{k \mid k > \frac{log(V)}{\log(q)}\} = \left\lfloor \frac{\log(V)}{\log(q)} \right\rfloor + 1$$

Así, el algoritmo para generar valores de una variable aleatoria con distribución geométrica $X \sim Geom(p)$ es el siguiente:

```
def geometrica(p):
    return int(log(random())/log(1-p))+1
```

1.5. Generación de variables Bernoulli

Recordemos que una variable aleatoria geométrica es el número de v.a. de Bernoulli hasta obtener un éxito. Así, si bien un método bastante sencillo de generar valores de una Bernoulli $B(p)$ es:

```
def Bernoulli(p):
    U=random()
    if U<p:
        return 1
    else:
        return 0
```

podemos optimizar este algoritmo utilizando la generación de una v.a. geométrica, $X \sim Geom(p)$. Esto es, si por ejemplo se genera el valor $X = 5$, esto es equivalente a generar 5 valores de v.a. independientes $B(p)$:

$$0 \quad 0 \quad 0 \quad 0 \quad 1,$$

Así, si se quieren generar N valores de variables aleatorias $B(p)$, independientes, es suficiente generar valores de $X \sim Geom(p)$, $x_1, x_2, \ldots, x_k$, hasta que $x_1 + x_2 + \cdots + x_k \geq N$, y así se obtiene la secuencia de Bernoulli:

$$\underbrace{0 \quad 0 \quad \ldots \quad 1}_{(x_1-1) \text{ ceros y un } 1} \quad \underbrace{0 \quad 0 \quad \ldots \quad 1}_{(x_2-1) \text{ ceros y un } 1} \ldots \quad \underbrace{0 \quad 0 \quad \ldots \quad 1}_{(x_k-1) \text{ ceros y un } 1}.$$

1.6. Generación de una variable aleatoria Poisson

La función de probabilidad de masa de una variable aleatoria Poisson de razón λ está dada por:

$$p_i = P(X = i) = e^{-\lambda}\frac{\lambda^i}{i!}, \qquad i = 0, 1, \ldots$$

Notemos que las probabilidades cumplen una relación de recurrencia:

$$p_0 = e^{-\lambda}, \qquad p_{i+1} = e^{-\lambda}\frac{\lambda^{i+1}}{(i+1)!} = p_i\,\frac{\lambda}{i+1}.$$

Así, el algoritmo obtenido por el método de la transformada inversa se puede escribir:

```
def Poisson(lambda):
    U=random()
    i = 0
    p= exp(-lambda)
    F=p
    while U>= F:
        i=i+1
        p=p*lambda/i
        F=F+p
    return i
```

Una observación con respecto a este algoritmo, es que chequea desde el valor 0 en adelante, pero p_0 no es la probabilidad mayor. Se podría optimizar si el algoritmo recorre primero las probabilidades más grandes. Además, el algoritmo realiza $n + 1$ comparaciones para generar el valor n. Dado que el valor esperado de la variable es λ, entonces el número de comparaciones es, en promedio, $\lambda + 1$. Así, si $\lambda >> 1$ el algoritmo realiza muchas comparaciones.

Una forma de mejorar este algoritmo es comenzar por analizar el valor más probable. El valor máximo de las probabilidades es $p_{\lfloor\lambda\rfloor}$, es decir, que los valores cercanos a λ son los más probables. Así, se comienza el algoritmo buscando a partir del valor

$$I = \lfloor\lambda\rfloor,$$

y luego se busca de manera ascendente o descendente según el valor de la variable uniforme U generada:

POISSON(λ)

$I = \lfloor \lambda \rfloor$

// Calcular $F(I)$ usando la definición recursiva de p_i.

$U = random()$

if $U \geq F(I)$

// *generar X haciendo búsqueda ascendente*

while $U \geq F(I)$

$I = I + 1$

$p = p * \lambda / I$

$F = F + p$

return $I - 1$ // *Devuelve el valor anterior.*

else

// *generar X haciendo búsqueda descendente.*

while $U < F(I)$

$F = F - p$

$p = p * I / \lambda$

$I = I - 1$

return I // *El algoritmo ejecuta hasta el valor a generar.*

De esta manera, el promedio de búsquedas se reduce a

$$1 + E\left[|X - \lambda|\right].$$

En particular, si $\lambda \gg 1$, la distribución se aproxima a una normal de media y varianza λ, $N(\lambda, \sqrt{\lambda})$, por lo cual $\frac{X-\lambda}{\sqrt{\lambda}}$ se aproxima a una normal estándar $Z \sim N(0, 1)$.

Por lo tanto, el promedio de búsquedas, $1 + E[|X - \lambda|]$, puede estimarse como:

$$\begin{aligned} 1 + \sqrt{\lambda} E\left[\frac{|X - \lambda|}{\sqrt{\lambda}}\right] &= 1 + \sqrt{\lambda} E[|Z|] \\ &= 1 + 0.798\sqrt{\lambda}. \end{aligned}$$

1.7. Generación de una variable aleatoria binomial

La generación de una variable aleatoria binomial responde a un caso similar al de una variable Poisson. Si $X \sim B(n, p)$, entonces la función de masa de probabilidad es:

$$p_i = P(X = i) = \frac{n!}{i!(n-i)!} p^i (1-p)^{n-i}, \qquad i = 0, 1, \ldots, n.$$

En este caso, la fórmula recursiva para las probabilidades está dada por:

$$p_0 = (1-p)^n, \qquad p_{i+1} = \frac{n-i}{i+1}\frac{p}{1-p}p_i, \quad 0 \leq i < n.$$

Recordamos además que el valor esperado y la varianza están dados por

$$\mathrm{E}[X] = n\,p \qquad\qquad \mathrm{Var}[X] = n\,p(1-p).$$

Si se aplica directamente el método de transformada inversa, el algoritmo resulta:

```
def Binomial(n,p):
    U=random()
    i=0
    c= p/(1-p)
    prob=(1-p)**n
    F=prob
    while U >= F
        prob=c*(n-i)/(i+1)]*prob
        F= F+prob
        i= i+1
    return i
```

Al igual que con la generación de v.a. Poisson, el inconveniente de este algoritmo es que al chequear desde 0 hasta generar el valor correspondiente, el número de comparaciones es 1 más que el valor generado. Por lo tanto, el promedio de comparaciones es $E[X] + 1 = np + 1$.

Además de la optimización análoga al método de Poisson, una alternativa que podría mejorar el número de comparaciones es si $p > 1/2$. En este caso, $1 - p < \frac{1}{2}$, y por lo tanto es conveniente generar $Y \sim B(n, 1-p)$, que lleva menor número de comparaciones, y tomar $X = n - Y$.

En cuanto a mejorar el algoritmo de manera análoga que Poisson, en este caso el valor más probable corresponde a

$$i = \lfloor n\,p \rfloor,$$

y el promedio de comparaciones, para n grande resulta del orden de

$$1 + \sqrt{np(1-p)}\,0.798.$$

2. Método de aceptación y rechazo

El **método de aceptación y rechazo**, o de rechazo, para generar una variable aleatoria X, asume la utilización de un método para la generación de otra variable aleatoria Y que cumpla con las siguientes propiedades:

- Si $P(X = x_j) > 0$, entonces $P(Y = x_j) > 0$, para todo x_j en el rango de X.
- Existe una constante positiva c tal que:

$$\frac{P(X = x_j)}{P(Y = y_j)} \leq c,$$

para todos los x_j tal que $P(X = x_j) > 0$.

Si denotamos $p_j = P(X = x_j)$ y $q_j = P(Y = x_j)$, entonces de la segunda propiedad vemos que:

$$\sum_{j \geq 1} p_j \leq c \cdot \sum_{j \geq 1} q_j \leq c$$

Como el miembro izquierdo de la desigualdad es 1, se tiene que la constante c es siempre mayor o igual a 1. Además la igualdad se daría sólo en el caso en que X e Y tienen la misma distribución de probabilidad, en cuyo caso ya se tendría un método para generar X.

Asumimos entonces que $c > 1$, y por lo tanto $\frac{1}{c} < 1$.

El algoritmo propuesto consiste en repetir los siguientes pasos:

MÉTODO DE ACEPTACIÓN Y RECHAZO

1 Generar un valor de Y
2 $U = random()$
3 **if** $U < \frac{p(Y)}{c \cdot q(Y)}$
4 **return** Y y terminar
5 **else**
6 Volver a 1

o equivalentemente:

MÉTODO DE ACEPTACIÓN Y RECHAZO

1 **repeat**
2 Generar un valor de Y
3 $U = random()$
4 **until** $U < \frac{p(Y)}{c \cdot q(Y)}$
5 **return** Y

Como puede verse de la descripción del algoritmo, se trata de un ciclo o lo que es lo mismo, una secuencia de condicionales:

C 1. : Generar Y, Generar U, si se acepta Y devolver X, si no seguir.

C 2. : Generar Y, Generar U, si se acepta Y devolver X, si no seguir.

C 3. : Generar Y, Generar U, si se acepta Y devolver X, si no seguir.

: ....

Esto es, el valor de X puede ser generado en el primer condicional, o en el segundo, o en el tercero, o en alguno de los siguientes.

Existe en esto una cierta analogía con una variable geométrica, en el sentido que se continúa el ciclo o ejecución de los condicionales (ensayos) hasta que se obtiene un éxito: aceptar el valor de Y.

En efecto, en cada paso, la probabilidad de generar algún valor de X es la probabilidad de aceptar el valor de Y, y esto está dado por:

$$\begin{aligned} P(\text{aceptar Y}) &= P\left(\{Y = y_1,\ U < \frac{p_1}{c\,q_1}\} \cup \{Y = y_2,\ U < \frac{p_2}{c\,q_2}\} \cup \cdots\right) \\ &= P\left(\bigcup_{j\geq 1}\{Y = y_j,\ U < \frac{p_j}{c\,q_j}\}\right). \end{aligned}$$

Notemos que esta unión es disjunta, y además U e Y son independientes. Por lo tanto resulta:

$$\begin{aligned} P(\text{aceptar Y}) &= \sum_{j\geq 1} P(Y = y_j) \cdot P(U \leq \frac{p_j}{c\,q_j}) \\ &= \sum_{j\geq 1} q_j \cdot \frac{p_j}{c\,q_j} = \frac{1}{c}. \end{aligned}$$

De esta manera, el número de iteraciones del algoritmo hasta aceptar el valor de Y es una variable aleatoria geométrica con probabilidad de éxito $\frac{1}{c}$ y de fracaso $1 - \frac{1}{c}$.

Ahora bien, un valor x_j de la variable X será generado sí o sí es generado en alguna iteración:

$$\begin{aligned} P(\text{generar } x_j) &= \sum_{k\geq 1} P(\text{generar } x_j \text{ en la iteración } k) \\ &= \sum_{k\geq 1} P(\text{rechazar } Y\ (k-1) \text{ veces y aceptar } Y = x_j \text{ en la iteración } k) \\ &= \sum_{k\geq 1} (1 - \frac{1}{c})^{k-1} P(Y = x_j,\ U \leq \frac{p_j}{c\,q_j}) \\ &= \sum_{k\geq 1} (1 - \frac{1}{c})^{k-1} q_j\ \frac{p_j}{c\,q_j} \\ &= p_j. \end{aligned}$$

Por lo tanto, el algoritmo simula una variable con la distribución deseada. Notemos en particular que el número esperado de iteraciones es el valor esperado de una variable geométrica $Geom(\frac{1}{c})$, es decir, c. Por lo tanto el algoritmo será más eficiente cuanto más chico sea el valor de c.

Ejemplo 2.1. Sea X una variable aleatoria con valores en $\{1, 2, \ldots, 10\}$ y probabilidades $0.11,\ 0.12,\ 0.09,\ 0.08,\ 0.12,\ 0.10,\ 0.09,\ 0.09,\ 0.10,\ 0.10$. Si se generan valores con el método de la transformada inversa, optimizando de modo que los primeros condicionales tengan mayor probabilidad de ser aceptados, el número esperado de iteraciones será:

$$(1+2)\cdot 0.12 + 3\cdot 0.11 + (4+5+6)\cdot 0.10 + (7+8+9)\cdot 0.09 + 10\cdot 0.08 = 5.15.$$

Por otro lado, si se genera esta variable rechazando con una variable aleatoria $Y \sim U[1, 10]$, tenemos que $P(Y = j) = 0.1$, y por lo tanto

$$P(X = j) \leq 1.2 \cdot P(Y = j), \qquad 1 \leq j \leq 10.$$

Luego el valor esperado del número de iteraciones es igual al valor esperado de una v.a. geométrica con $p = \frac{1}{1.2}$, que es $c = 1.2$.

2.1. Método de composición

El método de composición permite generar una v.a. X con función de probabilidad de masa dada por:

$$P(X = j) = \alpha p_j + (1 - \alpha) q_j \qquad j = 0, 1, \ldots,$$

donde α es algún número entre 0 y 1.

Entonces, si se tienen métodos eficientes para generar valores de dos variables aleatorias X_1 y X_2, con funciones de probabilidad de masa

$$P(X_1 = x_j) = p_j, \qquad P(X_2 = x_j) = q_j, \qquad j = 1, \ldots,$$

entonces estos métodos pueden ser utilizados para generar valores de X:

$$X = \begin{cases} X_1 & \text{con probabilidad } \alpha \\ X_2 & \text{con probabilidad } 1 - \alpha \end{cases}$$

El algoritmo sería como el que sigue:

1 $U = random()$
2 **if** $U < \alpha$
3 Simular X_1
4 **return** X_1
5 **else** :
6 Simular X_2
7 **return** X_2

Así,

$$\begin{aligned} P(\text{ generar } x_j) &= P(U < \alpha,\ X_1 = x_j) + P(\alpha \leq U < 1,\ X_2 = x_j) \\ &= P(U < \alpha)\, P(X_1 = x_j) + P(\alpha \leq U < 1)\, P(X_2 = x_j) \\ &= \alpha\, p_j + (1 - \alpha)\, q_j. \end{aligned}$$

Ejemplo 2.2. Si X es la variable aleatoria que toma los valores del 1 al 5 con probabilidad 0.05, y los valores del 6 al 10 con probabilidad 0.15, entonces X toma algún valor entre 1 y 5 con probabilidad 0.25, y un valor entre 6 y 10 con probabilidad 0.75. Por otro lado, los valores del 1 al 5 son equiprobables y los del 6 al 10 también. Luego se pueden elegir X_1 y X_2 uniformes discretas en $[1, 5]$ y $[6, 10]$ respectivamente, y generar X con el siguiente algoritmo:

```
U=random()
if U<0.75:
    V=random()
    return int(5*V)+1
else:
    V=random()
    return int(5*V)+6
```

También se pueden considerar las variables aleatorias $Y_1 \sim U[1, 10]$ e $Y_2 \sim U[6, 10]$, de modo que los valores de 1 a 5 sean generados con Y_1 y los restantes puedan ser generados con Y_1 o con Y_2. Esto es:

$$P(Y_1 = j) = 0.1, \quad 1 \leq j \leq 10, \qquad P(Y_2 = j) = 0.2, \quad 6 \leq j \leq 10.$$

Dado que X toma algún valor entre 1 y 5 con probabilidad 0.25, y estos son la mitad de los valores que toma Y_1, se da una ponderación $\alpha = 0.5$ a Y_1, y la misma ponderación a Y_2. Así, en este caso el algoritmo es el siguiente:

```
U=random()
V=random()
If U<0.5:
    return int(10*V)+1
else:
    return int(5*V)+6
```

En general, si se tienen n variables aleatorias con funciones de distribución $F_1, F_2, \ldots, F_n$, y $\alpha_1, \alpha_2, \ldots, \alpha_n$ son números positivos tales que

$$\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1,$$

entonces la función de distribución dada por

$$F(x) = \alpha_1 F_1(x) + \alpha_2 F_2(x) + \cdots + \alpha_n F_n(x),$$

se llama una **mezcla** o **composición** de las funciones de distribución F_1, F_2, ..., F_n.

Una variable aleatoria X con función de distribución X puede generarse a través de la simulación de una variable discreta I que toma valores en $\{1, 2, \ldots, n\}$:

$$P(I = j) = \alpha_j,$$

y una segunda simulación de una variable con distribución F_j, si $I = j$. Esta forma de generar X se llama **método de composición**.

2.2. El método del alias

Una aplicación del método de composición es el llamado **método del alias**. Este método sirve para generar variables aleatorias que toman un número finito de valores, digamos, $1, 2, \ldots, n$.

El método requiere construir en primer lugar los llamados **alias**, pero una vez completada esta etapa la simulación en sí es muy simple. La distribución de la variable X, F_X, se describe como una composición equiprobable de $n - 1$ variables aleatorias:

$$F_X(x) = \frac{1}{n-1}F_1(x) + \frac{1}{n-1}F_2(x) + \cdots + \frac{1}{n-1}F_{n-1}(x),$$

y $F_j(x)$ es la distribución de una variable X_j que toma a lo sumo dos valores posibles.

Para definir esta composición o mezcla, se utiliza el siguiente resultado:

Proposición 1. Sea $P(X = j) = p_j$, $1 \leq j \leq n$ una función de probabilidad de masa, con $n > 1$. Entonces:

a) Existe j, $1 \leq j \leq n$, tal que $p_j < \frac{1}{n-1}$.

b) Dado j como en a), existe $i \neq j$ tal que $p_i + p_j \geq \frac{1}{n-1}$.

La demostración es sencilla. Para ver a), notemos que si no fuera cierto entonces la suma de las probabilidades cumpliría

$$\sum_{j=1}^{n} p_j \geq n \cdot \frac{1}{n-1} > 1,$$

lo cual es absurdo.

Por otro lado, dado j como en a), debe existir i que cumpla b). De lo contrario, se tendría que:

$$1 - p_j = \sum_{i \neq j} p_i < (n-1)\left(\frac{1}{n-1} - p_j\right) = 1 - (n-1)p_j.$$

Esto es, $p_j > (n-1)p_j$, lo cual sólo es válido si $n = 1$.

El procedimiento es como sigue. En primer lugar, notemos que si se multiplican todas las probabilidades por $n-1$, entonces su suma es igual a $n-1$:

$$(n-1)p_1, \qquad (n-1)p_2, \qquad , \ldots , \qquad (n-1)p_n. \tag{1}$$

La idea es distribuir estos $n-1$ valores dados en (1) como probabilidades de masa de $n-1$ variables aleatorias, de modo que estas variables tomen a lo sumo dos valores distintos. Por ejemplo, si se tuviera una variable aleatoria X que toma valores en $\{1, 2, 3\}$ con probabilidades de masa:

$$p_1 = 0.3, \qquad p_2 = 0.6, \qquad p_3 = 0.1,$$

al mutiplicar por 2 tenemos los valores:

$$2\,p_1 = 0.6, \qquad 2\,p_2 = 1.2, \qquad 2\,p_3 = 0.2.$$

Ahora distribuimos estas probabilidades en dos variables aleatorias X_1 y X_2, usando que $1.2 = 0.4+0.8$, de modo que:

$$X_1 = \begin{cases} 1 & p = 0.6 \\ 2 & q = 0.4 \end{cases} \qquad\qquad X_2 = \begin{cases} 2 & p = 0.8 \\ 3 & q = 0.2 \end{cases}$$

y ahora, para cada $x = 1, 2, 3$ se cumple que:

$$P(X = x) = \frac{1}{2}\,P(X_1 = x) + \frac{1}{2}\,P(X_2 = x).$$

En un caso general, se eligen j e i como en la Proposición 1, y se define

$$X_1 = \begin{cases} j & \text{con probabilidad } (n-1)p_j \\ i & \text{con probabilidad } 1-(n-1)p_j \end{cases}.$$

Notemos que X_1 toma todo el peso del valor j, y parte del peso del valor i. Digamos que de la lista de valores en (1) se han utilizado p_j y una "parte" de p_i, de modo que suman 1. Si se restan estos dos valores en (1), la suma total es igual a $n-2$.

Es decir, si dividimos por $n-2$ se tendría nuevamente una variable aleatoria $\tilde{X}$ con valores en $\{1, 2, \ldots, n\} - \{j\}$, que tiene probabilidades

$$P(\tilde{X} = k) = \frac{1}{n-2}\,(n-1)\,p_k, \qquad k \neq j, i,$$

y $P(\tilde{X} = i)$ es tal que la suma de las probabilidades es 1.

Ahora se eligen nuevos índices j e i con las propiedades dadas en a) y b) de la Proposición 1, respectivamente, y se construye una variable X_2, cambiando n por $n-1$. El procedimiento sigue hasta que sólo quedan a lo sumo dos valores por generar.

Notemos que en cada paso, la variable X_k definida toma todo el peso de su correspondiente valor j, por lo cual este valor ya no será generado por las siguientes variables X_k. En consecuencia, el algoritmo de construcción del alias tiene $n-1$ pasos.

Ilustramos estos pasos con un ejemplo:

Ejemplo 2.3. Por ejemplo, si X es la v. a. que toma valores en $\{1,2,3,4\}$ con

$$p_1 = \frac{7}{16}, \quad p_2 = \frac{1}{4}, \quad p_3 = \frac{1}{8}, \quad p_4 = \frac{3}{16}.$$

Multiplicando estas probabilidades por $n-1=3$, obtenemos los siguientes valores tabulados en la Tabla 1:

Pasos	1	2	3	4	
$p_j \times (3)$	$\frac{21}{16}$	$\frac{3}{4}$	$\frac{3}{8}$	$\frac{9}{16}$	Suma=3

Tabla 1: Método del alias - Paso 1

La Proposición 1 nos dice que alguno de los valores de la tabla es menor que 1, y que para este valor hay otro cuya suma excede a 1. En general, se sugiere tomar el menor y mayor valor de la tabla respectivamente. En este caso: $j=3$ e $i=1$.

Consideramos entonces la variable aleatoria:

$$X_1 = \begin{cases} 3 & \text{con probabilidad } 3 \cdot p_3 = \frac{3}{8} \\ 1 & \text{con probabilidad } 1 - \frac{3}{8} = \frac{5}{8}. \end{cases}.$$

Así, restando las probabilidades que toma X_1 para los valores 3 y 1, la tabla resulta como en la Tabla 2:

j	1	2	3	4	
$p_j \times 3$	$\frac{21}{16}$	$\frac{3}{4}$	$\frac{3}{8}$	$\frac{9}{16}$	
	$\frac{11}{16}$	$\frac{3}{4}$		$\frac{9}{16}$	Suma=2

Tabla 2: Método del alias - Paso 2

Notemos que esta segunda fila suma 2, y al menos un elemento es menor que 1 y la suma de este con algún otro elemento supera a 1. Podemos elegir en este caso $j=4$ e $i=2$, y definir la variable:

$$X_2 = \begin{cases} 4 & \text{con probabilidad } \frac{9}{16} \\ 2 & \text{con probabilidad } 1 - \frac{9}{16} = \frac{7}{16}. \end{cases}.$$

j	1	2	3	4	
$p_j \times 3$	$\frac{21}{16}$	$\frac{3}{4}$	$\frac{3}{8}$	$\frac{9}{16}$	
	$\frac{11}{16}$	$\frac{3}{4}$		$\frac{9}{16}$	
	$\frac{11}{16}$	$\frac{5}{16}$			Suma=1

Tabla 3: Método del alias - Paso 3

Repitiendo el paso anterior, la tabla resulta como en el Cuadro 3:
y en este último paso directamente definimos

$$X_3 = \begin{cases} 1 & \text{con probabilidad } \frac{11}{16} \\ 2 & \text{con probabilidad } \frac{5}{16}. \end{cases}.$$

De esta forma, la distribución F de la variable aleatoria original X se puede escribir como:

$$F_X(x) = \frac{1}{3} F_{X_1}(x) + \frac{1}{3} F_{X_2}(x) + \frac{1}{3} F_{X_3}(x),$$

y el algoritmo por el método del alias para simular valores de X será como el siguiente:

```
def aliasX():
    I=int (random()*3)
    V=random()
    if I==1:
        if V<5/8: return 1
        else:     return 3
    elif I==2:
        if V<9/16: return 4
        else:      return 2
    else: #I==3
        if V<11/16: return 1
        else:       return 2
```

Ejercicio 2.1. Comprobar que el algoritmo anterior genera los valores del 1 al 4 con las probabilidades deseadas.

El nombre de **alias** proviene del hecho que es posible definir las variables X_k de modo que $P(X_k = k)$ sea positiva, para $1 \leq k < n$. En ese caso, X_k toma el valor k y posiblemente otro valor que se denomina su "alias".

También puede optimizarse este algoritmo generando una única variable uniforme $U \sim U(0,1)$, y tomar $I = \lfloor (n-1)U \rfloor + 1$, y $V = (n-1)U - I$.

2.3. Método de la urna

Otro método simple, pero quizás menos eficiente en cuanto al uso de memoria, es el **método de la urna**. Dada una variable aleatoria X, que toma un número finito de valores, digamos $\{1, 2, \ldots, n\}$, llamamos $p_j = P(X = j)$. El método consiste en considerar un valor $k \in \mathbb{N}$ tal que $k\,p_j$ sea entero, para todo j, $1 \leq j \leq n$.

Ahora se considera un arreglo A de k posiciones, y se almacena cada valor i en $k\,p_i$ posiciones del arreglo.

El algoritmo simplemente selecciona una posición al azar del arreglo y devuelve el valor en dicha posición.

```
def urnaX():
    I=int(random()*k)+1
    return A[i]
```

Ejemplo 2.4. Si X toma los valores $\{1, 2, 3\}$ con probabilidades

$$p_1 = 0.24, \qquad p_2 = 0.46, \qquad p_3 = 0.30,$$

se puede tomar $k = 100$. Así, se define un arreglo A con 100 posiciones, en las cuales se almacena el 1 en 24 posiciones, el 2 en 46 posiciones y el 3 en 30 posiciones.

Una mejora de este método, en cuanto al lugar de almacenamiento, es considerar las posiciones de los décimos, centésimos, milésimos, etc., por separado.

Así, en el Ejemplo 2.4, los décimos en las probabilidades están dados por 2, 4 y 3 , y los centésimos están dados por 4, 6 y 0.

Luego se consideran dos arreglos:

1. A_1: de $2 + 4 + 3 = 9$ posiciones, con 2 posiciones con el valor 1, 4 posiciones con valor 4 y 3 posiciones con el valor 3.

2. A_2: de $4 + 6 + 0 = 10$ posiciones, con 4 posiciones con el valor 1, 6 con el valor 2 y ninguna con el valor 3.

El arreglo A_1 se le da peso $\frac{90}{100} = 0.9$, y al arreglo A_2 se le da un peso de $\frac{10}{100} = 0.10$, y el algoritmo es como sigue:

```
def urnaXmejorada():
    U=random()
    if U<0.9:
        I=int(random()*9)+1
        return A1[I]
    else:
        I=int(random()*10)+1
        return A2[I]
```