Modelos y Simulación: Parcial 2

Ferré Valderrama, Eduardo

12 de mayo de 2025

Modelos y Simulación

Índice

1. Generación de Variables Aleatorias Discretas

1.1. Método de la Transformada Inversa

Consideremos una variable aleatoria discreta X, con funcion 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 la variable aleatoria discreta es:

$$F(x) = P(X \leq x) = \sum_{x_j \leq x} p_j$$

El algoritmo general para una variable aleatoria discreta que toma un numero finito de valores es como el siguiente

```
# x: vector de valores posibles de X
# p: vector de probabilidades
def discretaX(p, x):
    U = random()
    i, F = 0, p[0]
    while U >= F:
        i += 1
        F += p[i]
    return x[i]
```

1.1.1. Generación de una variable aleatoria uniforme discreta

Si X es una variable aleatoria con distribución uniforme discreta en $\{1, \ldots n\}$ entonces $p_1 = p_2 = \ldots = p_n = \frac{1}{n}$.
La aplicación del método de la transformada inversa conduce al siguiente algoritmo:

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

Una mejora del algoritmo:

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

Para generar X uniforme en $[m, k]$, discreta:

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

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:

$$(a_0, a_1, \ldots, a_{N-1}).$$

Un algoritmo es el siguiente:

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

Otro algoritmo pero recorriendo el vector de atrás hacia adelante:

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

1.1.2. Cálculo de promedios

$$\overline{a} = \frac{1}{N} \sum_{N}^{i} a_i$$

Queremos estimar $\overline{a}$
Definimos

- $g : \{1, \ldots, N\} \to \mathbb{R}$;
- $g(j) = a_j$
- $X \sim \mathcal{U}(1, N)$;
- $P(X = j) = \frac{1}{N}$

$$\implies \overline{a} = \frac{1}{N} \sum_{j=1}^{N} g(i) \cdot P(X = i) = E[g(X)]$$

Por la ley de los grandes números tomamos $X_1, X_2, \ldots, X_k$, $X_i \sim \mathcal{U}(1, N)$ y estimamos $\overline{a} = E[g(x)]$ como:

$$\frac{g(X_1) + g(X_2) + \ldots g(X_k)}{k} \simeq \overline{a}$$

Si queremos estimar

$$S = \sum_{i=1}^{N} b_i$$

Entonces escribimos:

$$B = \frac{S}{N} = \frac{1}{N} \sum_{i=1}^{N} b_i$$

$$h(i) = b_i$$

Estimamos

$$B \simeq \frac{1}{k}(h(X_1) + h(X_2) + \ldots h(X_k))$$

$$\implies S = \frac{N}{k}(h(X_1) + h(X_2) + \ldots h(X_k))$$

1.1.3. Generación de una variable aleatoria geométrica

$$X \in \{1, 2, 3 \ldots\} \qquad \text{con probabilidad de éxito } p$$

$$q = 1 - p \qquad \text{probabilidad de fracaso}$$

$$P(X = i) = p \cdot q^{i-1} \qquad F(i) = P(X \leq i) = 1 - P(X > i) = 1 - q^i$$

El algoritmo para la generación de una variable aleatoria geométrica es el siguiente:

```
def geom(p):
  U = random()
  return int(log(1 - U) / log(1 - p)) + 1
```

1.1.4. Generación de varaibles Bernoulli

$$X \sim B(p)$$

$$P(X = 1) = p \qquad P(X = 0) = 1 - p$$

El algoritmo para la generación de una variable aleatoria Bernoulli es el siguiente:

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

Si queremos obtener $X_1.X_2, \ldots, X_N \sim B(p)$; independientes

- $Y \sim geom(p)$
- $Y = j$ equivale a $X_1 = 1, X_2 = 1, \ldots, X_{j-1} = 1; X_j = 0$

El algoritmo para generar una lista de variables aleatorias Bernoulli es el siguiente:

```
##devuelve una lista de N Bernoullis B(p)
def NBernoullis(N, p):
  Bernoullis = [0] * N
  j = geom(p) - 1
  while j < N:
    Bernoullis[j] = 1
    j += geom(p)
  return Bernoullis
```

1.1.5. Generación de una variable aleatoria Poisson

$$X \sim \mathcal{P}(\lambda) \qquad X \in \{0, 1, 2, \ldots\}; \qquad \lambda > 0$$

$$P(X = j) = e^{-\lambda} \cdot \frac{\lambda^j}{j!} = p_j; \qquad j \geq 1$$

$$\Longrightarrow \; p_j = p_{j-1} \cdot \frac{\lambda}{j}$$

El algoritmo para la generación de una variable aleatoria Poisson es el siguiente:

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

Mejora del algoritmo:

```
def Poisson(lamda):
  p = exp(-lamda); F = p
  for j in range(1, int(lamda) + 1):
    p *= lamda / j
    F += p
  U = random()
  if U >= F:
    j = int(lamda) + 1
    while U >= F:
      p *= lamda / j
      F += p
      j += 1
    return j
  else:
    j = int(lamda)
    while U < F:
      F -= p; p *= j / lamda
      j -= 1
    return j + 1
```

1.1.6. Generación de una variable aleatoria binomial

$$X \sim B(n,p) \qquad X \in \{0,1,2,\ldots,n\}$$

$$p_j = P(X=j) = \binom{n}{j} p^j (1-p)^{n-j} \qquad j = 0,1,\ldots,n$$

$$p_0 = (1-p)^n \qquad p_{j+1} = p_j \cdot \frac{p}{1-p} \cdot \frac{(n-j)}{j+1}$$

El algoritmo para la generación de una variable aleatoria binomial es el siguiente:

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

1.2. Método de Aceptación-Rechazo

$$X \quad \text{variable aleatoria} \qquad X \in \{x_1, x_2, \ldots, x_n\}$$

$$P(X = x_j) = p_j, \qquad j \geq 1$$

Se conoce un método para generar una v.a Y:

- Si $P(X = x_j) = p_j > 0 \implies P(Y = x_j) = q_j > 0$
- $\exists$ constante $c > 0$ tal que:

$$p_j \leq c \cdot q_j, \quad \forall j \geq 1$$

$$\left[1 = \sum_{j \geq 1} p_j \leq c \cdot \sum_{j \geq 1} q_j \leq c \right]$$

En general, si X e Y tienen distinta distribución $\Rightarrow c > 1$.

El algoritmo es:

```
def AceptacionRechazo():
  while True:
    Simular Y
    U = random()
    if U <= p(Y) / (c * q(Y)):
      return Y
    else:
      continue
```

En general la constante c se elige como $n \cdot max_j\{p_j\}$

1.3. Método de composición

X v.a discreta

$$P(X = x_j) = \alpha_1 \cdot P(X_1 = x_j) + \alpha_2 \cdot P(X_2 = x_j) + \ldots + \alpha_n \cdot P(X_n = x_j)$$

$$\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1; \qquad \alpha_i \geq 0$$

U uniforme en $[0, 1]$.

$P(X = x_j) = P(U < \alpha_1)$ y $P(X_1 = x_j) + P(\alpha_1 < U < \alpha_1 + \alpha_2)$ y $P(X_2 = x_j) + \ldots$

El algoritmo es:

```
def Composicion(alpha, X):
  # alpha = [alpha_1, alpha_2, \ldots, alpha_n]
  # X = [X_1, X_2, \ldots, X_n]
  U = random()
  i = 1; F = alpha[1]
  while U >= F:
    i += 1
    F += alpha[i]
  return X[i]
```

1.4. Métodos alternativos

1.4.1. Método del alias

Para entenderlo es mejor ver como se explica con un ejemplo en el apunte.

1.4.2. Método de la urna

$$X \in \{x_1, x_2, \ldots, x_n\}$$

$$p_i = P(X = x_i)$$

Si existe algún $k \in \mathbb{N}$ tal que $p_i \cdot k \in \mathbb{N}$, $1 \leq i \leq n$.

$$A = \left[\underbrace{x_1, \ldots, x_1}_{p_1 \cdot k}, \underbrace{x_2, \ldots, x_2}_{p_2 \cdot k}, \ldots, \underbrace{x_n, \ldots, x_n}_{p_n \cdot k} \right]$$

El algoritmo es:

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

2. Generación de Variables Aleatorias Continuas

2.1. Método de la Transformada Inversa

X v.a continua con función de densidad $f(x)$ y función de distribución $F(x)$.

$$F(x) = \int_{-\infty}^{x} f(s)ds \qquad F(x) = P(X \leq x)$$

El algoritmo es:

```
def Tinversa():
  U = random()
  return G(U) # G = F^{-1}
```

2.1.1. Simulación de una variable aleatoria exponencial

Si X es una variable aleatoria con distribución exponencial con parámetro $\lambda = 1$, $X \sim \mathcal{E}(1)$, entonces su función de distribución acumulada está dada por:

$$F(x) = \begin{cases} 1 - e^{-x} & \text{si } x > 0 \\ 0 & \text{si } x \leq 0. \end{cases}$$

Luego, la inversa de F sobre $(0, 1)$ está dada por:

$$F^{-1}(u) = -\ln(1 - u), \quad u \in (0, 1).$$

Así, el algoritmo de simulación para $X \sim \mathcal{E}(1)$ es:

```
def exponencial():
  U = 1 - random()
  return -log(1 - U)
```

Notemos que si X es exponencial con parametro 1, esto es $X \sim \mathcal{E}(1)$, entonces $Y = \frac{1}{\lambda}X$ con media $\frac{1}{\lambda}$: $Y \sim \mathcal{E}(\lambda)$

Para simular $Y \sim \mathcal{E}(\lambda)$ es:

```
def exponencial(lamda):
  U = 1 - random()
  return -log(U) / lamda
```

2.1.2. Simulación de una variable aleatoria Poisson $X \sim \mathcal{P}(\lambda)$

$$\text{Proceso de Poisson con tasa} \quad \lambda \quad N(1) \sim \mathcal{P}(\lambda)$$
$$N(1) = n$$

Si se simulan variables aleatorias exponenciales $X_1, X_2, \ldots$, con $X_i \sim \mathcal{E}(\lambda)$ para $i \geq 1$, hasta que

$$X_1 + X_2 + \cdots + X_n \leq 1 \quad \text{y} \quad X_1 + X_2 + \cdots + X_n + X_{n+1} > 1,$$

entonces n representa el número de arribos hasta $t = 1$. Esto es:

$$N(1) = \text{máx}\{n \mid X_1 + X_2 + \cdots + X_n \leq 1\}$$

Si se simula cada exponencial X_i con

$$X_i = -\frac{1}{\lambda}\ln(1 - U_i), \quad \text{con } U_i \sim \mathcal{U}(0,1),$$

tenemos que:

$$\begin{aligned}
N(1) &= \text{máx}\{n \mid X_1 + X_2 + \cdots + X_n \leq 1\} \\
&= \text{máx}\left\{n \mid -\frac{1}{\lambda}\left(\ln(1-U_1) + \ln(1-U_2) + \cdots + \ln(1-U_n)\right) \leq 1\right\} \\
&= \text{máx}\left\{n \mid -\frac{1}{\lambda}\ln\left((1-U_1)(1-U_2)\cdots(1-U_n)\right) \leq 1\right\} \\
&= \text{máx}\{n \mid \ln\left((1-U_1)(1-U_2)\cdots(1-U_n)\right) \geq -\lambda\} \\
&= \text{máx}\left\{n \mid (1-U_1)(1-U_2)\cdots(1-U_n) \geq e^{-\lambda}\right\}
\end{aligned}$$

Luego:

$$N(1) = \text{mín}\left\{n \mid (1-U_1)(1-U_2)\cdots(1-U_n) < e^{-\lambda}\right\} - 1$$

El algoritmo es:

```
def Poisson_con_exp(lamda):
  X = 0
  Producto = 1 - random()
  cota = exp(-lamda)
  while Producto >= cota:
    Producto *= (1 - random())
    X += 1
  return X
```

2.1.3. Simulación de una variable con distribución Gamma$(n, \frac{1}{\lambda})$

- Sean $n \in \mathbb{N}$, y $X_1, X_2, \ldots, X_n \sim \mathcal{E}(\lambda)$ independientes.
- Entonces:
$$X_1 + X_2 + \cdots + X_n \sim \text{Gamma}\left(n, \tfrac{1}{\lambda}\right)$$
- Para generar $Y \sim \text{Gamma}\left(n, \frac{1}{\lambda}\right)$, basta con:
$$\begin{aligned} Y &= X_1 + X_2 + \cdots + X_n \\ &= \frac{-\ln(1-U_1) - \ln(1-U_2) - \cdots - \ln(1-U_n)}{\lambda} \\ &= \frac{-\ln\left((1-U_1)(1-U_2)\cdots(1-U_n)\right)}{\lambda} \end{aligned}$$
- Donde $U_i \sim \mathcal{U}(0,1)$ y $U_1, U_2, \ldots, U_n$ son independientes.

El algoritmo es:

```
def Gamma(n, lamda):
  'Simula una gamma con parametros n y 1/lamda'
  U = 1
  for _ in range(n):
    U *= (1 - random())
  return -log(U) / lamda
```

Para generar X, Y exponenciales independientes con parámetro λ, podemos aplicar el siguiente algoritmo:

```
def DosExp(lamda):
  V1, V2 = 1 - random(), 1 - random()
  t = -log(V1 * V2) / lamda
  U = random()
  X = t * U
  Y = t - X
  return X, Y
```

Para simular n exponenciales:

```
def NExponenciales(n, lamda):
  t = 1
  for _ in range(n): t *= random()
  t = -log(t) / lamda
  unif = random.uniform(0, 1, n-1)
  unif.sort()
  exponenciales = [unif[0] * t]
  for i in range(n-2):
    exponenciales.append((unif[i+1] - unif[i]) * t)
  exponenciales.append((1 - unif[n-2]) * t)
  return exponenciales
```

2.2. Método de Aceptación-Rechazo

Supongamos que se quiere generar una variable aleatoria X con función de densidad f:

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(t)\,dt,$$

y que se tiene un método para generar otra variable Y, con densidad g, tal que

$$\frac{f(y)}{g(y)} \leq c, \quad \text{para todo } y \in \mathbb{R} \text{ tal que } f(y) \neq 0.$$

El **método de rechazo** para generar X a partir de Y tiene el siguiente algoritmo:

```
def Aceptacion_Rechazo_X():
  while True:
    Simular_Y()
    U = random()
    if U < f(Y) / (c * g(Y)):
      return Y
```

2.3. Simulación de variables aleatorias normales

Si $X \sim \mathcal{N}(\mu, \sigma)$, entonces:

$$\frac{X - \mu}{\sigma} \sim \mathcal{N}(0, 1)$$

Si $Z \sim \mathcal{N}(0, 1)$, entonces:

$$Z \cdot \sigma + \mu \sim \mathcal{N}(\mu, \sigma)$$

La función de densidad estándar es:

$$f_Z(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Por transformada inversa y rechazo **no se puede**. (Justificación en las notas).

2.3.1. Por composición usando $|Z|$

Generar Z:

```
def Normal_composicion():
    Generar |Z|
    if random() < 0.5:
        return |Z|
    else:
        return -|Z|
```

Una variable Y fue generada previamente, se necesita saber cómo generarla.

Generar $Y = |Z|$

```
def abs_Z():
    while True:
        X = -log(1 - random())
        Y1 = -log(1 - random())
        if Y1 > (X - 1)**2 / 2:
            return X
            # Conservar Y1 - ((X - 1)**2 / 2)
            # como el valor de X en la siguiente llamada al
               algoritmo
```

2.3.2. Método polar

```
def MetodoPolar():
    Rcuadrado = -2 * log(1 - random())
    Theta = 2 * Pi * random()
    X = sqrt(Rcuadrado) * cos(Theta)
    Y = sqrt(Rcuadrado) * sin(Theta)
    return (X * sigma + mu, Y * sigma + mu)
```

Si queremos dos variables estándar normales, retornamos X, Y.

Una mejora para no generar funciones trigonométricas

```
def Polar_Box_Muller(mu, sigma):
    # Generar un punto aleatorio en el circulo unitario
    while True:
        V1, V2 = 2 * random() - 1, 2 * random() - 1
        if V1 ** 2 + V2 ** 2 <= 1:
            S = V1 ** 2 + V2 ** 2
            X = V1 * sqrt(-2 * log(S) / S)
            Y = V2 * sqrt(-2 * log(S) / S)
            return (X * sigma + mu, Y * sigma + mu)
```

Idem para el caso estándar.

2.3.3. Método de razón entre uniformes

```
from math import exp
NV_MAGICCONST = 4 * exp(-0.5) / sqrt(2.0)

def normalvariate(mu, sigma):
    while 1:
        u1 = random()
        u2 = 1.0 - random()
        z = NV_MAGICCONST * (u1 - 0.5) / u2
        zz = z * z / 4.0
        if zz <= -log(u2):
            break
    return mu + z * sigma
```

2.4. Generación de un Proceso de Poisson

2.4.1. Procesos de Poisson homogéneos

```
def eventosPoisson(lamda, T):
    t = 0
    NT = 0
    Eventos = []
    while t < T:
        U = 1 - random()
        t += - log(U) / lamda
        if t <= T:
            NT += 1
            Eventos.append(t)
    return NT, Eventos
```

2.4.2. Procesos de Poisson no homogéneos

```
def Poisson_no_homogeneo_adelgazamiento(T):
    'Devuelve el numero de eventos NT y los tiempos en Eventos'
    # lamda_t(t): intensidad, lamda_t(t) <= lamda
    NT = 0
    Eventos = []
    U = 1 - random()
    t = -log(U) / lamda
    while t <= T:
        V = random()
        if V < lamda_t(t) / lamda:
            NT += 1
            Eventos.append(t)
        t += -log(1 - random()) / lamda
    return NT, Eventos
```