Cómputo Científico **Omar García Ramos**
Tarea IV

Ejercicio 1

Sea X es una variable aleatoria continua distribuida como según la ley de F, es decir $X \sim F_X$, tenemos que la función de distribución acumulada inversa generalizada de X se define como:

$$F_X^-(u) = \text{ínf}\{x \in \mathbb{R} : F_X(x) \geq u\}, \quad u \in (0,1)$$

donde $F_X(x) = P(X \leq x)$, demostraremos que $F_X^-(u) = F_X^{-1}(u)$

Demostración:

Como F_X es continua y estrictamente creciente, el conjunto $\{x : F_X(x) \geq u\}$ es el intervalo $[x_u, \infty)$, donde x_u es el único valor tal que $F_X(x_u) = u$.
El ínfimo de ese conjunto es x_u, que es precisamente $F_X^{-1}(u)$.

Por otro lado para poder simular variables aleatorias, sea $U \sim$ Uniforme$(0,1)$, y sea F_X la ley de X. Entonces la variable aleatoria $Y = F_X^-(U)$ tiene la misma distribución que X, es decir, $F_Y(x) = F_X(x)$ para todo x.

Demostración:

Recordemos la propiedad clave de la CDF inversa generalizada:

$$F_X^-(u) \leq x \quad \iff \quad u \leq F_X(x)$$

Esto se debe a la definición de $F_X^-(u)$ como el ínfimo del conjunto $\{x : F_X(x) \geq u\}$.
Entonces:

$$P(Y \leq x) = P(F_X^-(U) \leq x) = P(U \leq F_X(x))$$

Dado que $U \sim U(0,1)$, $P(U \leq t) = t$ para $t \in [0,1]$, es decir es la identidad, así:

$$P(Y \leq x) = F_X(x)$$

Por lo tanto, Y tiene la misma ley que F_X, es decir, $Y \sim X$.

Ejercicio 2

Se implementó un generador de números pseudoaleatorios mediante el método congruencial lineal múltiple:

$$x_i = (107374182 \cdot x_{i-1} + 104420 \cdot x_{i-5}) \quad \text{mód } (2^{31} - 1)$$

El procedimiento de generación consta de tres etapas:

1. **Inicialización**: Se define un estado inicial de 5 valores $[x_0, x_1, x_2, x_3, x_4]$ a partir de una semilla dada.
2. **Iteración**: Para cada nuevo número:
   - Se calcula x_i mediante la fórmula recursiva
   - Se actualiza el estado mediante el desplazamiento: $x_{j-1} = x_j$ para $j = 1, 2, 3, 4, 5$
   - Se asigna $x_4 = x_i$ (incorporando el nuevo valor al estado)
3. **Normalización**: Se obtiene $u_i = \dfrac{x_i}{2^{31} - 1} \in [0, 1)$

Para validar que los números generados u_i siguen una distribución $U(0, 1)$, se compararon sus propiedades estadísticas con los valores teóricos:

- **Media teórica**: $\mathbb{E}[X] = \dfrac{1}{2} = 0{,}5$
- **Varianza teórica**: $\mathbb{V}[X] = \dfrac{1}{12} \approx 0{,}08333$

Cuadro 1: Comparación entre valores teóricos y observados ($n = 10,000$)

Propiedad	**Teórico**	**Observado**
Media	0.5	0.4988
Varianza	0.08333	0.085

Los resultados muestran una concordancia cercana entre los valores teóricos y observados, confirmando que las variables generadas provienen de la distribución $U(0, 1)$.

Es importante destacar que la calidad del generador exhibe un comportamiento asintótico. Como muestran las Figuras 1, 2 y 3, la distribución empírica converge a la uniforme teórica conforme aumenta el número de simulaciones:

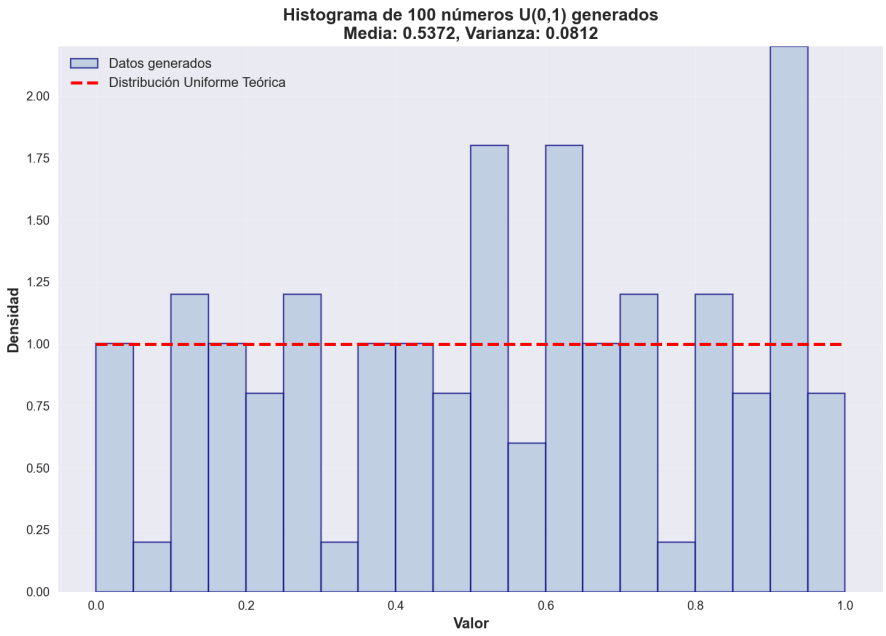

Figura 1: 100 simulaciones - Mayor variabilidad muestral

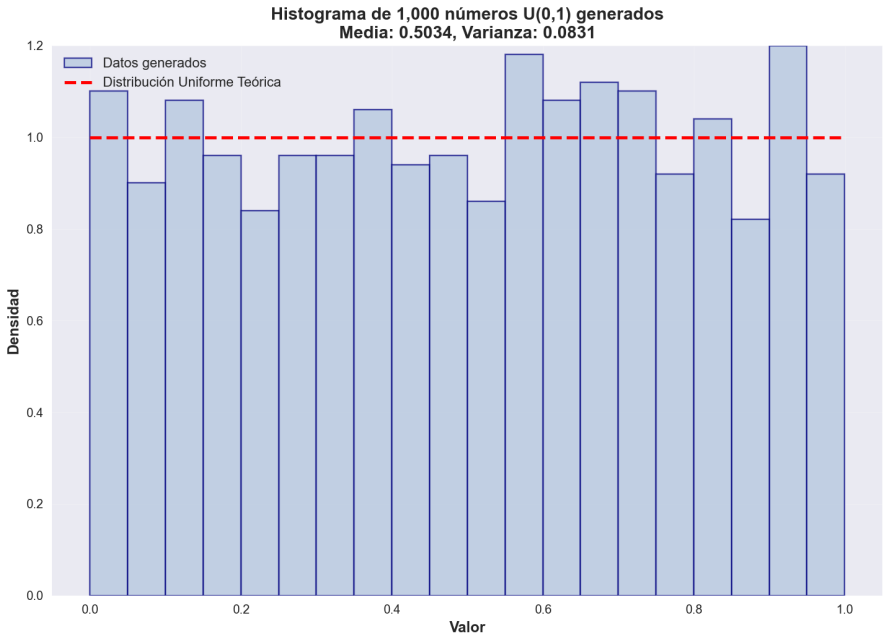

Figura 2: 1,000 simulaciones - Mejor aproximación a la uniforme

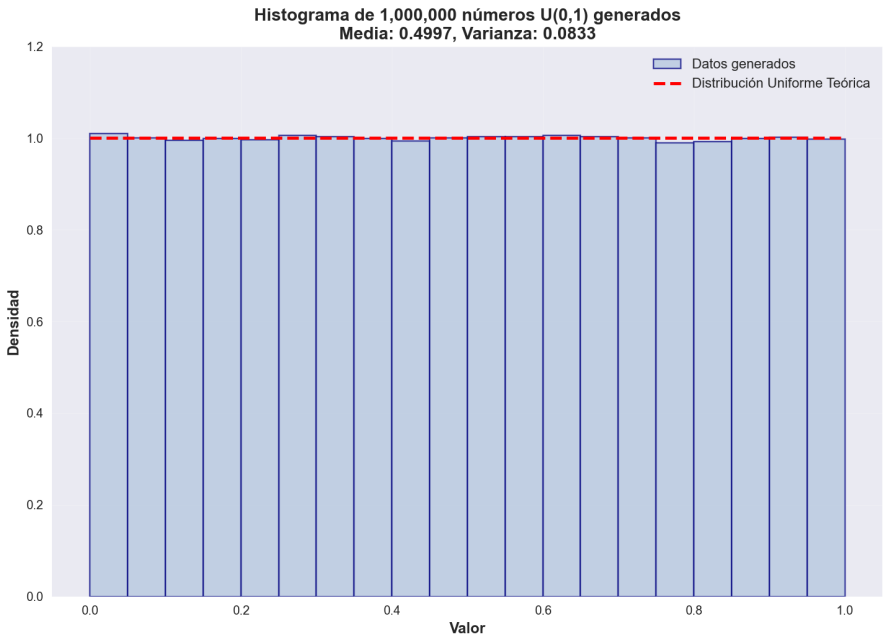

Figura 3: 1,000,000 simulaciones - Excelente ajuste a la distribución teórica

Desde el punto de vista teórico, la independencia entre variables sucesivas no puede garantizarse completamente en generadores determinísticos, no obstante el generador implementado produce secuencias que convergen asintóticamente a la distribución $U(0,1)$ teórica y proporcionan una aproximación práctica a variables i.i.d. para simulación.

Ejercicio 3

Según la documentación oficial de Python, el algoritmo que usa scipy.stats.uniform para generar números aleatorios [1]:

> "Casi todas las funciones del módulo dependen de la función básica `random()`, la cuál genera uniformemente un número flotante aleatorio en el rango semiabierto [0.0, 1.0). Python utiliza **Mersenne Twister** como generador principal. Éste produce números de coma flotante de 53 bits de precisión y tiene un periodo de $2^{19937} - 1$. La implementación subyacente en C es rápida y segura para subprocesos."

Una implementación, puede verse de la siguiente manera:

```
import random

# Método básico
random.seed(123)
random.seed()     # Usa hora del sistema por defecto

# Para reproducibilidad
random.seed(42)
muestras = [random.random() for _ in range(10)]
```

De forma analoga en R, el generador por defecto también es **Mersenne-Twister**, implementado según el algoritmo de Matsumoto y Nishimura (1998).

```
# Establecer semilla
set.seed(123)

# Generar uniformes
muestras <- runif(1000, min = 0, max = 1)
```

Cuadro 2: Comparación de generadores de números aleatorios: Python vs R

Característica	**Python**	**R**
Algoritmo por defecto	Mersenne Twister[1]	Mersenne Twister[2]
Periodo	$2^{19937} - 1$[1]	$2^{19937} - 1$[2]
Precisión flotante	53 bits[1]	32 bits
Función semilla	`random.seed()`[1]	`set.seed()`[2]
Función uniforme	`random.random()`[1]	`runif(n, min=0, max=1)`[2]
Semilla por defecto	Hora del sistema[1]	Hora del sistema[2]

[1]Python Software Foundation. (2021). *Módulo random*. Documentación oficial de Python 3.9. https://docs.python.org/es/3.9/library/random.html
[2]Casal, R. F. (2023). *Números aleatorios en R*. En Simulación Estadística y Computación. https://rubenfcasal.github.io/simbook/rrng.html

Ejercicio 4

Simulación de variables aleatorias discretas en Python

Según la documentación de Python [1], las principales funciones para variables discretas son:

1. `random.choice(seq)`

> "Retorna un elemento aleatorio de una secuencia seq no vacía. Si seq está vacía, lanza IndexError."

```
import random
# Distribución discreta simple
valores = ['cara', 'cruz']
resultado = random.choice(valores)
```

2. `random.choices(population, weights=None, k=1)`

> "Retorna una lista de elementos de tamaño k elegidos de la population con reemplazo. Si se especifica una secuencia weights, las selecciones se realizan de acuerdo con las ponderaciones relativas."

```
# Distribución discreta con pesos
valores = [0, 1, 2]
probabilidades = [0.2, 0.5, 0.3]
muestras = random.choices(valores, weights=probabilidades, k=100)
```

3. `random.sample(population, k)`

> "Retorna una lista de longitud k de elementos únicos elegidos de la secuencia de población. Se utiliza para el muestreo aleatorio sin reemplazo."

Preprocesamiento y algoritmos:

- `choice`: Utiliza el método básico de selección uniforme. No tiene preprocesamiento significativo.
- `choices`: Según la documentación:

  > "El algoritmo usado por `choices()` emplea aritmética de coma flotante para la consistencia interna y velocidad. El algoritmo usado por `choice()` emplea por defecto aritmética de enteros con selecciones repetidas para evitar pequeños sesgos de errores de redondeo."

- Para `choices` con pesos, internamente construye una distribución acumulativa (CDF) con preprocesamiento $O(n)$.

Eficiencia y características:

Cuadro 3: Funciones para variables discretas en Python

Función	Uso	Preproceso	Algoritmo
`choice()`	Una muestra uniforme	$O(1)$	Selección directa
`choices()`	Múltiples muestras con pesos	$O(n)$	Transformada inversa
`sample()`	Muestreo sin reemplazo	$O(n)$	Algoritmo de Fisher-Yates

Para distribuciones discretas complejas, se recomienda usar `numpy.random.choice()` que implementa el método de alias para mayor eficiencia con grandes soportes.

Referencias

[1] Python Software Foundation. (2021). *Módulo random*. Documentación oficial de Python 3.9. Disponible en: `https://docs.python.org/es/3.9/library/random.html`

Ejercicio 5

La simulación de densidades complejas puede realizarse mediante el método de rechazo. El *Adaptive Rejection Sampling* (ARS) es una variante eficiente para densidades log-cóncavas que aprovecha tangentes a la función logarítmica $h(x) = \log f(x)$ para construir una envolvente superior que se actualiza al recibir evaluaciones de f

1. Supuestos

La función de densidad de la distribución Gamma(α, β) está dada por:

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, \quad x > 0 \tag{1}$$

Su log-densidad es:

$$h(x) = \log f(x) = (\alpha - 1)\log x - \beta x - \log \Gamma(\alpha) + \alpha \log \beta \tag{2}$$

Para $\alpha \geq 1$, la segunda derivada es negativa:

$$h''(x) = -\frac{\alpha - 1}{x^2} < 0 \tag{3}$$

Por lo tanto, $h(x)$ es cóncava, condición que permite la construcción de envolventes lineales por partes.

1.1. Algoritmo ARS

Sea $S_n = \{x_0, x_1, \ldots, x_k\}$ un conjunto ordenado de puntos en el soporte de f. Dada la concavidad de h, se construyen:

Las rectas secantes se definen como:

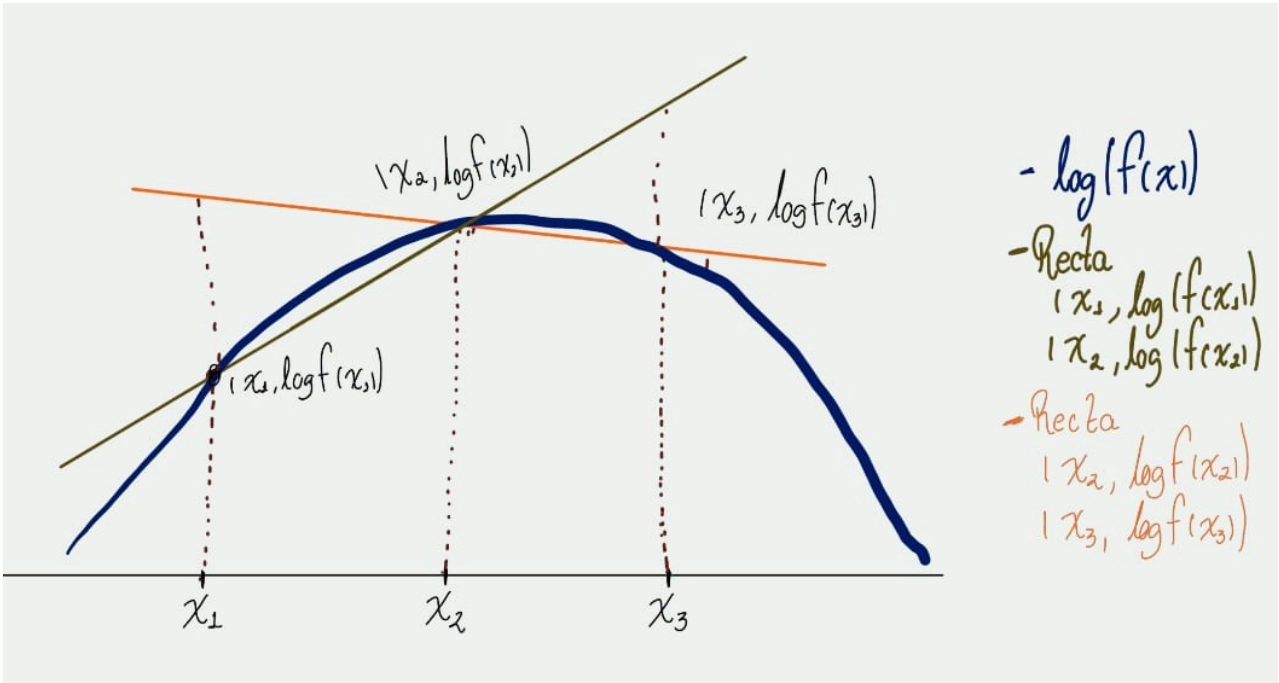

Figura 4: Envolventes superiores e inferiores para la función $h(x) = \log f(x)$ concava.

- **Envolvente superior**: $\overline{h}_n(x)$ formada por extensiones de las rectas secantes entre puntos consecutivos
- **Envolvente inferior**: $\underline{h}_n(x)$ formada por las propias rectas secantes entre puntos consecutivos

$$L_{i,i+1}(x) = h(x_i) + \frac{h(x_{i+1}) - h(x_i)}{x_{i+1} - x_i}(x - x_i) \tag{4}$$

Mientras que para k puntos, construimos $k+1$ segmentos exponenciales:

$$c \cdot g(x) = \sum_{i=1}^{k+1} \exp(a_i + b_i x) \cdot \mathbf{1}_{[z_{i-1}, z_i]}(x) \tag{5}$$

donde:

$$\begin{aligned} a_i &= h(x_i) - b_i \cdot x_i \\ b_i &= \frac{h(x_{i+1}) - h(x_i)}{x_{i+1} - x_i} \quad \text{(pendiente de la secante)} \end{aligned}$$

y z_i son los puntos de intersección entre secantes consecutivas.

1.2. Cálculo de Intersecciones

Los puntos de intersección z_i entre las secantes $L_{i-1,i}$ y $L_{i,i+1}$ se calculan resolviendo:

$$h(x_{i-1}) + b_{i-1}(z_i - x_{i-1}) = h(x_i) + b_i(z_i - x_i) \tag{6}$$

lo que da:

$$z_i = \frac{h(x_i) - h(x_{i-1}) - b_i x_i + b_{i-1} x_{i-1}}{b_{i-1} - b_i} \tag{7}$$

En la implementación del cálculo de intersecciones entre secantes, se incorporó un manejo robusto para casos donde las pendientes son numéricamente indistinguibles. Cuando $|b_i - b_{i-1}| < \epsilon$, se emplea el punto medio entre los puntos de soporte como aproximación, pues tenemos que $h(x)$ es lineal para $\alpha = 1$, y esto la solución evita inestabilidades numéricas sin afectar la calidad del muestreo.

La constante de normalización c se calcula analíticamente como:

$$\begin{aligned} c &= \sum_{i=1}^{k+1} \int_{z_{i-1}}^{z_i} \exp(a_i + b_i x) dx \\ &= \sum_{i=1}^{k+1} \frac{1}{b_i} \left(\exp(a_i + b_i z_i) - \exp(a_i + b_i z_{i-1}) \right) \\ &= \sum_{i=1}^{k+1} c_i \end{aligned}$$

De esta manera $g(x)$ es una mezcla de exponenciales truncadas.

$$\begin{aligned} g(x) &= \sum_{i=1}^{k+1} \left(\frac{1}{\sum_j c_j} \right) \exp(a_i + b_i x) \mathbf{1}_{[x_{i-1}, x_i]}(x) \\ &= \sum_{i=1}^{k+1} \left(\frac{c_i}{\sum_j c_j} \right) \frac{\exp(a_i + b_i x)}{c_i} \mathbf{1}_{[x_{i-1}, x_i]}(x) \\ &= \sum_{i=1}^{k+1} w_i f_i(x | a_i, b_i, x_{i-1}, x_i) \end{aligned}$$

Para muestrear de $g(x)$, se utiliza el método de mezclas:

1. Calcular pesos $w_i = c_i / c$ para cada segmento
2. Seleccionar segmento i con probabilidad w_i
3. Muestrear X del segmento exponencial truncado $[z_{i-1}, z_i]$

Para una distribución exponencial con parámetro b_i en el intervalo $[A, B]$, la densidad es:

$$g(x) = \frac{b_i e^{b_i x}}{e^{b_i B} - e^{b_i A}} \quad \text{para } A \leq x \leq B \tag{8}$$

La función de distribución acumulada es:

$$F(x) = \frac{\exp(b_i x) - \exp(b_i A)}{\exp(b_i B) - \exp(b_i A)} \tag{9}$$

El muestreo por transformada inversa se realiza resolviendo $F(X) = U$ donde $U \sim \text{Uniforme}(0, 1)$:

$$\begin{aligned} \frac{e^{b_i X} - e^{b_i A}}{e^{b_i B} - e^{b_i A}} &= U \\ e^{b_i X} &= e^{b_i A} + U(e^{b_i B} - e^{b_i A}) \\ X &= \frac{1}{b_i} \log \left[e^{b_i A} + U(e^{b_i B} - e^{b_i A}) \right] \end{aligned}$$

Una vez generado $X \sim g(x)$, se procede con el siguiente algoritmo de aceptación:

1. Generar $U \sim \text{Uniforme}(0, 1)$
2. Evaluar las siguientes condiciones en orden:

$$\begin{aligned} &\text{Si } U \leq \frac{\exp(\underline{h}_n(X))}{\exp(\overline{h}_n(X))} \quad \Rightarrow \quad \text{Aceptar } X \quad \text{(Aceptación por envolvente inferior)} \\ &\text{Si no, si } U \leq \frac{f(X)}{\exp(\overline{h}n(X))} \quad \Rightarrow \quad \text{Aceptar } X \text{ y actualizar } Sn + 1 = S_n \cup X \\ &\text{Si no} \quad \Rightarrow \quad \text{Rechazar } X \end{aligned}$$

Donde:
$\overline{h}_n(x)$: Envolvente superior (log-densidad de $g(x)$)
$\underline{h}_n(x)$: Envolvente inferior (construida por secantes)
$f(X)$: Densidad objetivo evaluada en X
S_n: Conjunto de puntos de soporte en la iteración n

1.3. Adaptaciones del Algoritmo ARS General

En la implementación práctica del algoritmo ARS, es crucial determinar cuándo detener el proceso de adaptación de la envolvente. El criterio se basa en el principio de que cuando la envolvente superior $g_n(x)$ se aproxima suficientemente a la densidad objetivo $f(x)$, la eficiencia del muestreo se estabiliza.
Se implementaron dos criterios complementarios:

1. **Criterio por límite máximo**: Se establece un número máximo de muestras de adaptación ($N_{\max} = 20$). Esto evita una adaptación computacionalmente costosa cuando la mejora marginal es insignificante.
2. **Criterio por eficiencia**: Se monitorea la tasa de aceptación r_n en una ventana deslizante de las últimas m muestras. Cuando $r_n \geq r_{\text{umbral}}$ (alrededor del 90 %) de manera consistente, se detiene la adaptación.

El algoritmo ARS está teóricamente definido para el soporte completo $(-\infty, \infty)$. Sin embargo, en la implementación práctica para la distribución Gamma($\alpha = 2$, $\beta = 1$), se introdujeron límites prácticos en el intervalo $(0, 12]$ debido a consideraciones numéricas y probabilísticas.

- **Estabilidad numérica:** Valores excesivamente cercanos a 0 pueden generar inestabilidades en el cálculo de $\log(x)$ durante la evaluación de la log-densidad.
- **Masa probabilística despreciable:** El análisis de la distribución revela que la probabilidad acumulada más allá de $x = 12$ es insignificante:

1.3.1. Validación Experimental del Muestreo

El algoritmo demostró una adaptación eficiente durante la fase de muestreo:

```
Probabilidad para x > 12: 0.000080
Porcentaje de masa perdida: 0.0080%
Adaptación detenida: alcanzado máximo de 20 muestras
Tasa de aceptación global: 0.996
Puntos de soporte finales: 9
```

La masa probabilística truncada es insignificante (0,008 %), confirmando que la aproximación numérica es robusta sin pérdida sustancial de precisión. El criterio de parada por límite máximo se activó después de solo 20 muestras de adaptación, indicando una rápida convergencia de la envolvente.

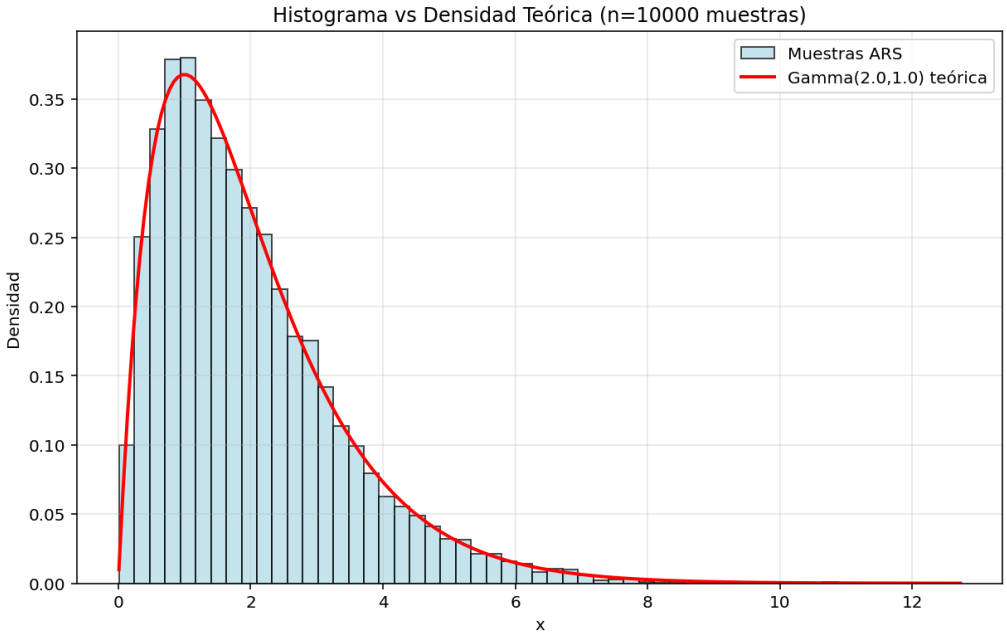

Figura 5: Comparación entre el histograma de muestras generadas por ARS y la densidad teórica Gamma($\alpha = 2$, $\beta = 1$). Se observa concordancia visual entre la distribución empírica y la teórica.

Métrica	**Valor**
Test KS (p-value)	0.5224
Media muestral	1.9901
Varianza muestral	1.9756
Puntos de soporte finales	9
Muestras totales	10,000

Cuadro 4: Resultados de la simulación ARS para Gamma($\alpha = 2$, $\beta = 1$).

Los resultados muestran que la implementación del algoritmo ARS produce muestras simulaciones válidas:

- **Test de bondad de ajuste**: El p-value de 0.5224 (> 0.05) indica que no hay evidencia estadística para rechazar la hipótesis de que las muestras siguen una distribución Gamma(2,1). El estadístico de Kolmogorov-Smirnov de 0.0081 refleja una excelente concordancia.
- **Precisión de momentos**: La media muestral (1.9901) presenta un error relativo de solo 0.50 % respecto al valor teórico (2.00), mientras que la varianza muestral (1.9756) muestra un error del 1.22 %.
- **Eficiencia del algoritmo**: La tasa de aceptación global del 99.6 % indica que la envolvente superior está muy ajustada a la densidad objetivo, minimizando el número de evaluaciones rechazadas de la función $f(x)$.

La rápida estabilización sugiere que para distribuciones Gamma con $\alpha \geq 1$, un número modesto de puntos de soporte (10-20) es suficiente para construir una envolvente efectiva..