Cómputo Científico para Probabilidad, Estadística y Ciencia de Datos

Ezau Faridh Torres Torres

TAREA 5: Simulación Estocástica, introducción.

Fecha de entrega: 09/Oct/2024.

NOTA: Los ejercicios se encuentran repartidos en los archivos:

- ejercicio2_tarea5.py
- ARS.py
- ejercicio5_tarea5.py

---

1. a) Definir la cdf inversa generalizada F_X^- y demostrar que en el caso de variables aleatorias continuas esta coincide con la inversa usual.

   b) Demostrar además que en general para simular de X podemos simular $u \sim U(0,1)$ y $F_X^-(u)$ se distribuye como X. [1 punto]

Respuesta:

a)

Sea X una variable aleatoria con función de distribución acumulada (CDF) $F_X(x) = \mathbb{P}(X \leq x)$, la **CDF inversa generalizada** $F_X^-(q)$ se define como:

$$F_X^-(q) = \inf\{x \in \mathbb{R} : F_X(x) \geq q\} \tag{1}$$

para $q \in (0,1)$.

Entonces, si X es una variable aleatoria continua, su CDF $F_X(x)$ es una función no decreciente y continua en todo su dominio aunque $F_X(x)$ puede ser constante en ciertos intervalos (donde la densidad de probabilidad $f_X(x) = 0$). Entonces, se tienen dos casos:

- Cuando $F_X(x)$ es estrictamente creciente, para cada $q \in (0,1)$, existe un único x tal que $F_X(x) = q$ gracias a que la función es biyectiva (al ser continua y estrictamente creciente). Este x es el único valor que satisface $F_X(x) = q$ y por lo tanto, la inversa usual existe, es única y cumple que $F_X^{-1}(q) = x$.

  Además, por la definición de la inversa generalizada, se busca el ínfimo de los valores de x tales que $F_X(x) \geq q$. Como $F_X(x)$ es estrictamente creciente, este conjunto solo contiene un valor de x, que es el que satisface $F_X(x) = u$. Esto implica que:

  $$F_X^-(q) = \inf\{x \in \mathbb{R} : F_X(x) \geq q\} = F_X^{-1}(q). \tag{2}$$

  Por lo tanto, cuando $F_X(x)$ es estrictamente creciente, la inversa generalizada $F_X^-(q)$ coincide exactamente con la inversa usual $F_X^{-1}(q)$ ya que en ambos casos se obtiene el único valor x que satisface $F_X(x) = q$.

- En donde $F_X(x)$ es constante, la inversa generalizada sigue siendo válida. La definición de $F_X^-(q)$ como el ínfimo de los x tales que $F_X(x) \geq q$ asegura que siempre obtenemos el valor correcto de x, incluso si la CDF no es estrictamente creciente.

b)

Sea $u \sim U(0,1)$, queremos probar que $F_X^-(u)$ se distribuye como X, es decir, que

$$\mathbb{P}\left(F_X^-(u) \leq x\right) = \mathbb{P}\left(X \leq x\right) = F_X(x). \tag{3}$$

Por definición de la inversa generalizada, se tiene que $F_X^-(u) \leq x$ implica que el valor de u debe estar en el rango de valores menores o iguales a $F_X(x)$, es decir, $u \leq F_X(x)$. Con esto,

$$\mathbb{P}\left(F_X^-(u) \leq x\right) = \mathbb{P}\left(u \leq F_X(x)\right) \tag{4}$$

Finalmente, dado que $u \sim U(0,1)$, se tiene que la probabilidad de que $u \leq a \in [0,1]$ es exactamente a. Entonces, $\mathbb{P}\left(u \leq F_X(x)\right) = F_X(x)$. Sustituyendo esto en (4), se tiene que $F_X^-(u) \sim X$.

---

2. Implementar el siguiente algoritmo para simular variables aleatorias uniformes:

$$x_i = 107374182 x_{i-1} + 104420 x_{i-5} \mod 2^{31} - 1. \tag{5}$$

regresa x_i y recorrer el estado, esto es $x_{j-1} = x_j$; $j = 1,2,3,4,5$; ¿parecen $U(0,1)$? [1 punto]

Respuesta:

En el archivo ejercicio2_tarea5.py se genera la función *GENERATE_UNIFORM()*, que genera una muestra de variables aleatorias "uniformes" $U(0,1)$ usando el método de congruencia lineal. La función recibe como argumentos el estado inicial de la secuencia (que se toma aleatorio), los coeficientes de la relación recursiva (5), el módulo de la relación recursiva y el número de muestras a generar. La función devuelve una lista con las muestras generadas.

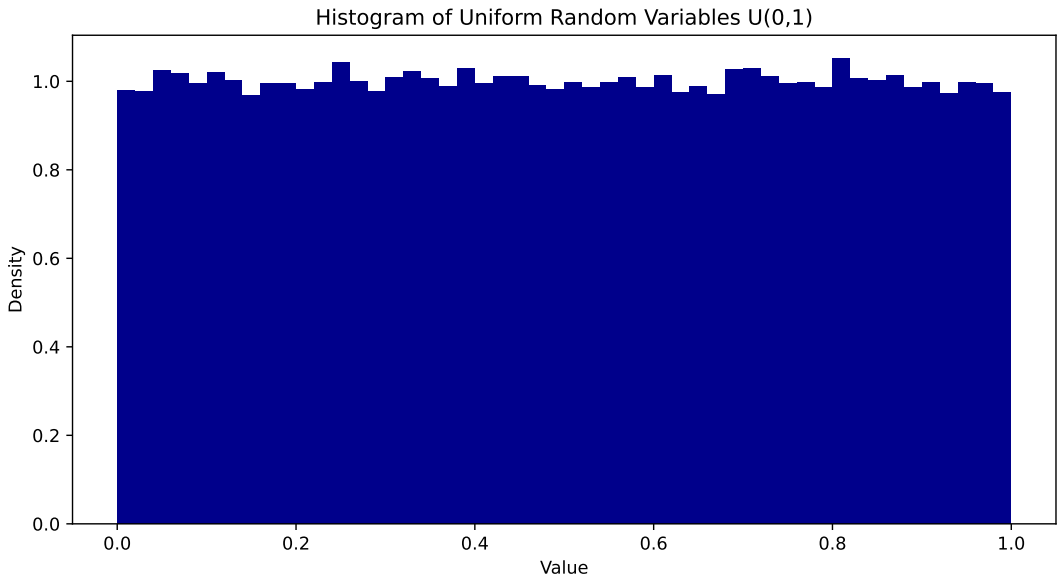


Figure 1: Ejemplo de simulación para $n = 100000$ muestras.

Después de varias simulaciones con diferentes estados iniciales, se puede ver que todas tienen un comportamiento similar al mostrado en la figura anterior, por lo que sí parecen tener una distribución uniforme en el intervalo $[0, 1]$.

---

3. ¿Cuál es el algoritmo que usa scipy.stats.uniform para generar números aleatorios? ¿Cómo se pone la semilla? ¿y en R? [1 punto]

Respuesta:

- En Python, la función scipy.stats.uniform utiliza el generador de números pseudoaleatorios de NumPy (numpy.random.Generator) para generar números aleatorios a partir de la distribución uniforme. El algoritmo detrás de este generador se basa en la implementación del método PCG64 (Permuted Congruential Generator) por defecto en versiones recientes de NumPy, que es rápido, eficiente y cumple con buenos estándares de aleatoriedad. La semilla se fija con numpy.random.seed()

  PCG64 (Permuted Congruential Generator 64) es un algoritmo de generación de números pseudoaleatorios de alta calidad que fue diseñado para ofrecer mejor rendimiento y mayor seguridad frente a defectos comunes en generadores más simples. PCG64 utiliza un generador congruencial lineal (LCG), que es uno de los métodos más antiguos y populares para generar números pseudoaleatorios (como el implementado en el ejercicio 2). El LCG genera secuencias de números pseudoaleatorios utilizando la fórmula:

  $$X_{n+1} = (a \cdot X_n + c) \mod m$$

  Donde X_n es el estado interno del generador, a, c y m son constantes. En el caso de PCG, m es una potencia de 2, lo que permite una implementación muy eficiente. PCG64 trabaja con enteros de 64 bits, lo que le permite tener un período extremadamente largo, lo que significa que puede generar una cantidad gigante de números aleatorios antes de repetir una secuencia. En particular, el período es 2^{128}, lo que garantiza que las secuencias sean muy largas antes de repetir.

- En R, la función equivalente es runif(), que genera números aleatorios de una distribución uniforme. R utiliza el generador de números pseudoaleatorios Mersenne Twister por defecto. La semilla se fija con set.seed().

  El Mersenne Twister tiene un período de $2^{19937} - 1$ (más largo que el de Numpy y Scipy, aunque ambos pasan las famosas pruebas de aleatoriedad), lo que significa que puede generar una cantidad inmensa de números aleatorios antes de repetir la secuencia. El Mersenne Twister es un generador de números pseudoaleatorios basado en operaciones de desplazamiento de bits y mezclas no lineales para generar números de alta calidad.

  El algoritmo Mersenne Twister en R genera números pseudoaleatorios utilizando un estado interno de 624 enteros de 32 bits. Mezcla y transforma estos valores con operaciones de desplazamiento y XOR para producir números de alta calidad, con un largo período de $2^{19937} - 1$.

---

4. ¿En scipy que funciones hay para simular una variable aleatoria genérica discreta? ¿tienen preproceso? [1 punto]

Respuesta:

En SciPy, para simular una variable aleatoria discreta genérica, se puede utilizar principalmente dos funciones dentro del módulo scipy.stats: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_discrete.html

- *rv_discrete:*

Es una clase para definir una distribución discreta genérica. Se pueden especificar los posibles valores de la variable aleatoria y sus probabilidades asociadas.

- *rv_sample (en scipy.stats.qmc)::*

Esta es otra función útil para variables aleatorias discretas genéricas, aunque se encuentra en el submódulo qmc para *muestreo cuasi-Monte Carlo*. Permite definir una distribución discreta en función de los valores y probabilidades que asignas.

Para la parte de preprocesamiento, en la documentación oficial de Scipy se menciona que antes de generar las muestras, ambas funciones verifican que:

- La suma de las probabilidades sea 1.
- Los valores estén correctamente definidos.

Este es el preprocesamiento que se realiza para garantizar que los datos sean válidos antes de la simulación. Además, en *rv_discrete*, las probabilidades se almacenan internamente de manera eficiente, permitiendo un acceso rápido para el muestreo durante la simulación.

---

5. Implementar el algoritmo Adaptive Rejection Sampling y simular de una $Gamma(2, 1)$ $10,000$ muestras. ¿cuando es conveniente dejar de adaptar la envolvente? [6 puntos]

Respuesta:

Sabemos que ARS es un método de muestreo que utiliza una envolvente superior de la función objetivo para proponer muestras y luego decidir si aceptarlas o no mediante el criterio de aceptación/rechazo. Esto se implementa en el archivo ARS.py.

Debido a la cantidad de código, se propone la creación de un diccionario que almacene los parámetros del método ARS y las funciones auxiliares. De esta forma, se puede acceder a los valores de los parámetros y las funciones de manera más sencilla. Además, se propone la creación de una función que inicialice los parámetros del método ARS y los guarde en el diccionario. Por último, se propone la creación de una función que genere muestras de la distribución log-concava usando el método ARS y actualice los parámetros del diccionario a medida que se generan muestras.

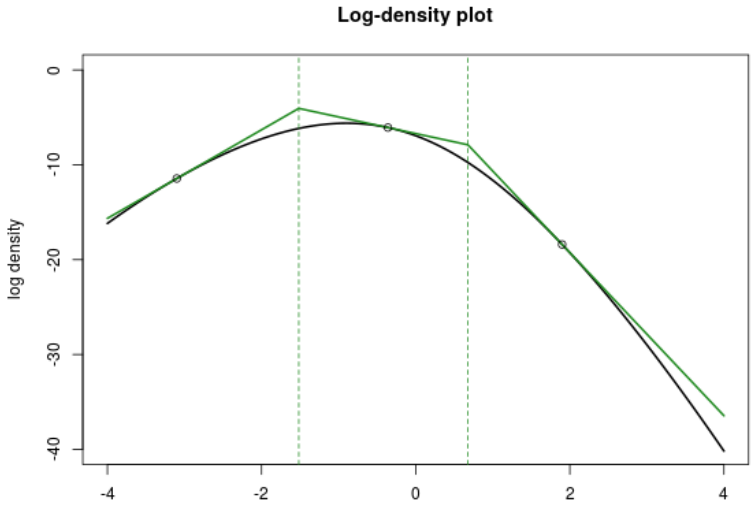
Nota: Primero se revisará, a grandes rasgos, un poco de lo visto en clase para entender mejor cada parte del código:

Sean

- $f(x)$ la función proporcional a la densidad objetivo de la que se quiere muestrear.
- $h(x) = \log(f(x))$ el logaritmo de la función objetivo (es una función cóncava) con derivada $h'(x)$.
- $\mathcal{S}_n$ un conjunto de puntos x_i con $i = 0, \dots n+1$ en el soporte de f tales que $h(x_i) = \log(f(x_i))$ se conoce hasta una constante.

- $\mathcal{L}_i(x)$ la recta con pendiente $h^{'}(x_i)$ que pasa por el punto $(x_i, h(x_i))$, es decir, la recta tangente a h.

Notemos que $\mathcal{L}_i$ siempre está por encima de h, esto es posible gracias a la concavidad de h.



Intersecciones entre las $\mathcal{L}$'s

Notemos que en cada intervalo $[x_i, x_{i+1}]$, se tiene un par de rectas tangentes correspondientes a x_i y x_{i+1}. Además cuentan con una intersección para algún punto $z_i \in [x_i, x_{i+1}]$ (ver imágen anterior). Por la ecuación de la recta que úne dos puntos, tenemos que estas rectas están dadas por

$$\mathcal{L}_i(x) = h(x_i) + h^{'}(x_i)(x - x_i) \text{ y } \mathcal{L}_{i+1}(x) = h(x_{i+1}) + h^{'}(x_{i+1})(x - x_{i+1}) \tag{6}$$

Entonces, la intersección es aquel z_i tal que cumpla ambas ecuaciones anteriores. Después de igualar estas expresiones y hacer los cálculos necesarios, se tiene que:

$$z_i = -\frac{h_{i+1} - h_i - \left(h^{'}_{i+1}x_{i+1} - h^{'}_i x_i\right)}{h^{'}_{i+1} - h^{'}_i} \tag{7}$$

donde $h_j = h(x_j)$ y $h^{'}_j = h^{'}(x_j)$. De esta forma, la envolvente superior está dada por:

$$u(x) := \begin{cases} h(x_0) + h^{'}(x_0)(x - x_0) = \mathcal{L}_0(x) & \text{si } x \leq z_0, \\ \vdots & \\ h(x_i) + h^{'}(x_i)(x - x_i) = \mathcal{L}_i(x) & \text{si } z_i \leq x \leq z_{i+1}, \\ \vdots & \\ h(x_n) + h^{'}(x_n)(x - x_n) = \mathcal{L}_n(x) & \text{si } z_n \leq x, \end{cases}$$

y la envolvente en cada punto de intersección es

$$u_i := \mathcal{L}_i(z_i). \tag{8}$$

Área bajo las rectas tangentes

A continuación, se necesitan calcular los segmentos de área bajo las rectas tangentes de la envolvente superior, o bien, el área acumulada bajo las tangentes de la envolvente superior. Esto es necesario para normalizar las probabilidades durante el proceso de muestreo:

Se tiene que $u(x) = \mathcal{L}_i(x)$ en $[x_i, x_{i+1}]$, así

$$e^{u(x)} = e^{\mathcal{L}_i(x)} = e^{h(x_i)+h'(x_i)(x-x_i)} = e^{h(x_i)}e^{h'(x_i)(x-x_i)}$$

El área bajo la curva $e^{u(x)}$ en $[x_i, x_{i+1}]$ es:

$$e^{h(x_i)}\int_{x_i}^{x_{i+1}} e^{h'(x_i)(x-x_i)}dx = e^{h(x_i)}\frac{e^{h'(x_i)(x_{i+1}-x_i)}-1}{h'(x_i)} = \frac{e^{u(x_{i+1})}-e^{u(x_i)}}{h'(x_i)}, \tag{9}$$

ya que $u(x_{i+1}) = h(x_i) + h'(x_i)(x_{i+1} - x_i)$. Entonces, la suma acumulada de estos términos es:

$$s_j := \sum_{i=1}^{j} \frac{e^{u(x_{i+1})}-e^{u(x_i)}}{h'(x_i)}. \tag{10}$$

Y la última suma acumulativa es:

$$cu_i := s_n. \tag{11}$$

ARS.py

A continuación, se da una descipción detallada de lo que hacen las funciones implementadas en el archivo:

- Las expresiones (7), (8), (10), (11) se calculan con la función auxiliar *intersecciones_y_areas()*, la cual nos ayuda a hacer estos cálculos para una muestra dada de x_i, sus evaluaciones $h(x_i)$ y sus pendientes $h'(x_i)$.
- Posteriormente, se da la función *DIC_INICIAL()* la cual crea un diccionario con los parámetros iniciales para el método de muestreo de Adaptive Rejection Sampling (ARS). Inicializa los puntos x_i, los valores $h(x_i)$, sus pendientes $h'(x_i)$ y usa la función *intersecciones_y_areas()* para calcular los puntos de intersección z_i de las tangentes. Además, calcula la envolvente superior $u(x) = h(x_i) + h'(x_i)(x - x_i)$ y el área acumulada s_i bajo las tangentes. El área acumulada hasta el último punto se guarda en cu_i. Esta función regresa el diccionario de parámetros inicializados.
- Después, se tiene la función *muestreo()* la cual devuelve un solo valor muestreado aleatoriamente desde la envolvente superior de la función log-concava que está siendo muestreada y el índice del segmento en el que cayó el valor muestreado. Genera un número aleatorio $u \sim \mathcal{U}(0, 1)$ y este se utiliza para seleccionar un punto en la envolvente superior, basándose en las áreas acumuladas bajo las rectas tangentes que representan la envolvente superior ya que se quiere encontrar en qué segmento de la envolvente superior se encuentra el valor de u:

  $$i = \max\left\{j \mid \frac{s_j}{cu_i} < u\right\} \tag{12}$$

  La función usa el concepto de muestreo inverso por transformada. Al generar un número aleatorio u, se busca un punto cuya área acumulada (normalizada) hasta dicho punto sea

igual a u (como el ejemplo contructivo visto en clase). Una vez identificado el segmento adecuado, se muestrea un punto dentro de ese segmento usando las propiedades locales de la función log-convexa (como las pendientes y las alturas).

Una vez encontrado el segmento i, se calcula el punto x_t dentro de ese segmento de la envolvente superior. Para esto, se usa la fórmula:

$$x_t = x_i + \frac{-h(x_i) + \log\left(h'(x_i) \cdot (cu_i \cdot u - s_i) + e^{u_i}\right)}{h'(x_i)} \tag{13}$$

Esta se obtuvo gracias a que, ya habíamos visto que para $x_t \in [x_i, x_{i+1}]$, entonces

$$\int_{x_i}^{x_t} e^{u(x)} dx = \frac{e^{u(x_t)} - e^{u(x_i)}}{h'(x_i)}$$

y se quiere que esta área acumulada se corresponda con el número aleatorio u generado en $[0, 1]$, es decir, se necesita resolver la ecuación:

$$\frac{e^{u(x_t)} - e^{u(x_i)}}{h'(x_i)} = cu_i \cdot u - s_i.$$

Resolviendo esta ecuación para x_t, de obtiene le expresión (13).

- En la función *insertar_puntos()* se actualizan las envolventes con nuevos puntos. Si no se proporcionan, solo recalcula la envolvente a partir de los puntos existentes. Si se proporcionan nuevos puntos, se concatenan con los existentes y se recalcula la envolvente. Además, se recalculan los puntos de intersección de las tangentes y el área acumulada bajo la envolvente con ayuda de la función auxiliar *intersecciones_y_areas()*.
- Finalmente, se tiene la función *aceptacion_rechazo()*. La cual genera N muestras a partir de la envolvente superior y actualizarla si es necesario. Esta función implementa el muestreo adaptativo, aprovechando el método ARS. La lógica se basa en aceptar o rechazar muestras propuestas de acuerdo con la envolvente superior y la función subyacente. Usa a la función *muestreo()* para obtener una propuesta de muestra x_t y el índice del segmento de la envolvente superior donde está situada. Calcular la altura de la envolvente superior en x_t, luego, genera un número aleatorio $u \sim \mathcal{U}$ para decidir si se acepta x_t o no. Esto se logra comparando a u con la probabilidad de aceptación:

$$\exp\left(h(x_t) - u(x_t)\right) \tag{14}$$

(Esto se deriva de la relación entre la envolvente superior y la función objetivo). Si

$$u < \exp\left(h(x_t) - u(x_t)\right) \tag{15}$$

se acepta la muestra y se almacena y se actualiza $\mathcal{S}_n$.

Ejemplos de uso:

Al final del archivo ARS.py se tiene un par de ejemplos:

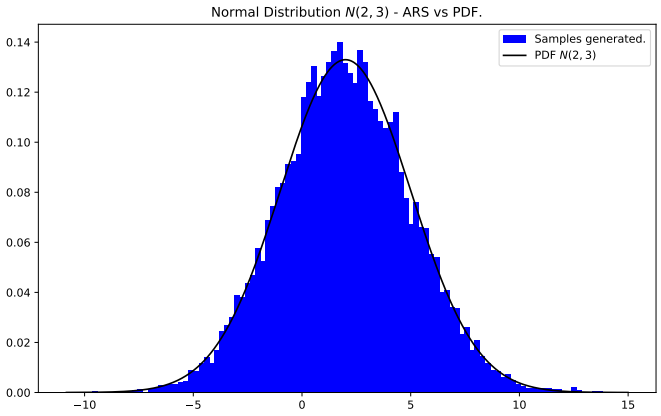
- Generación de una distribución normal $X \sim \mathcal{N}(2, 3)$. Para esto, necesitamos h cóncava tal que e^h sea proporcional a la distribución de X. Para esto, se propone una alteración de la distribución Log Normal:

$$h(x, \mu, \sigma) = -\frac{(x - \mu)^2}{2\sigma^2} \tag{16}$$

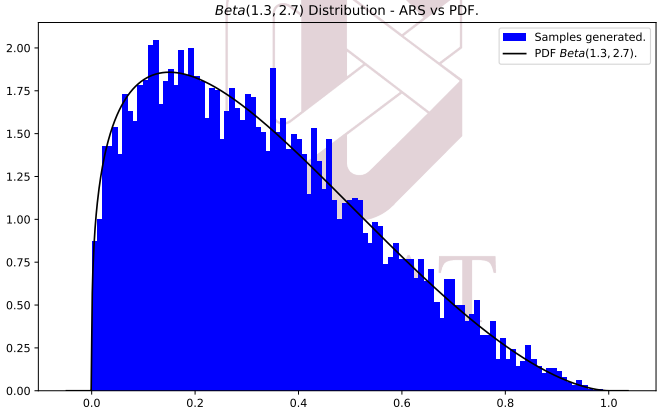
con derivada:

$$h'(x,\mu,\sigma) = -\frac{(x-\mu)}{\sigma^2} \tag{17}$$

Notemos que $e^{h(x)} \propto \mathcal{N}(\mu,\sigma)$. Se obtuvo el siguiente resultado y se comparó con la distribución verdadera $X \sim \mathcal{N}(2,3)$:



- De la misma forma, se generó una distribución $Beta(1.3, 2.7)$:



Distribución $Gamma(2,1)$

Para generar esta distribución, en el archivo ejercicio5_tarea5.py, se propone el uso de

$$h(x,k,\theta) = (k-1)\log x - \frac{x}{\theta} \tag{18}$$

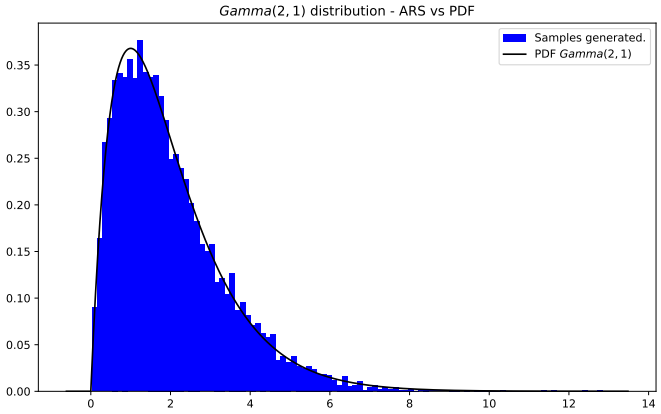
con derivada:

$$h'(x,k,\theta) = \frac{k-1}{x} - \frac{1}{\theta} \tag{19}$$

En la cual podemos notar que

$$e^{h(x)} = e^{(k-1)\log x - \frac{x}{\theta}} = x^{k-1}e^{-\frac{x}{\theta}} \propto \frac{1}{\Gamma(k)\theta^k}x^{k-1}e^{-\frac{x}{\theta}} \tag{20}$$

que es la distribución de $X \sim Gamma(k,\theta)$. Esto se usa en ejercicio5_tarea5.py para generar $N = 10000$ muestras de $Gamma(2,1)$, para los puntos iniciales: $\mathcal{S} = \{0.1, 0.5, 2, 5, 10\}$ en el soporte de X. Esto nos genera:



Para responder la pregunta ¿cuándo es conveniente dejar de adaptar la envolvente?:

En ARS, inicialmente se utilizan algunos puntos para definir la envolvente superior (compuesta por segmentos lineales), pero después, cada vez que se rechaza un valor propuesto o cuando se detecta que la envolvente puede mejorar, se añaden nuevos puntos de tangencia.

Si el criterio de rechazo se cumple en una proporción suficientemente baja (es decir, la mayoría de las muestras propuestas son aceptadas), significa que la envolvente ya está ajustada de manera adecuada. En este caso, puedes dejar de adaptar la envolvente porque el proceso de muestreo es eficiente. O bien, fijar un límite máximo para el número de puntos para mejorar el costo computacional. En mi implementación se usó un número máximo de 100 puntos.