Problema 1

Definir la cdf inversa generalizada F_X^- y demostrar que en el caso de variables aleatorias continuas esta coincide con la inversa usual. Demostrar además que en general para simular X podemos simular $u \sim \mathcal{U}(0,1)$ y $F_X^-(u)$ se distribuye como X.

Definimos la **inversa generalizada**, o función quantil, de una función de distribución acumulativa de la siguiente manera :

$$F_X^-(p) \;=\; \inf\{x \in \mathbb{R} : p \leq F_X(x)\} \tag{1}$$

Sea Y una variable aleatoria con función de distribución acumulativa F_Y continua y estrictamente creciente. Aseguramos que para todo y en el soporte de Y existe un único número real $p \in [0,1]$ tal que $F_Y(y) = p$, puesto que F_Y es estrictamente creciente en el soporte de Y. Por lo que si reducimos el dominio de F_Y a su soporte y su rango a $[0,1]$, tenemos una función biyectiva y existe una inversa F_Y^{-1}. Cumpliendose que,

$$F_Y^{-1}(F_Y(y)) \;=\; F_Y(F_Y^{-1}(y)) \;=\; y \tag{2}$$

Ahora queremos verificar que la **inversa generalizada** también funciona como inversa de la función de distribución. Tenemos que para todo $y \in \mathbb{R}$,

$$\begin{aligned} F_X^-(F_X(y)) \;&=\; \inf\{x \in \mathbb{R} : F_X(y) \leq F_X(x)\} \;=\; y \\ F_X(F_X^-(y)) \;&=\; F_X(\inf\{x \in \mathbb{R} : y \leq F_X(x)\}) \;\geq\; y \end{aligned} \tag{3}$$

Y en el caso continuo, la desigualdad es siempre la igualdad. De modo que la inversa generalizada coincide con la inversa usual.

Ahora consideremos una variable aleatoria $\mathbf{U} \sim \mathcal{U}(0,1)$ con distribución uniforme continua. Apliquemos la transformación $\mathbf{Z} = F_X^-(\mathbf{U})$ a nuestra variable aleatoria. Encontremos la funcíon de distribución de $\mathbf{Z}$.

$$\begin{aligned} F_Z(z) &= \mathbb{P}[Z \leq z], \\ &= \mathbb{P}[F_X^-(\mathbf{U}) \leq z], \\ &= \mathbb{P}[F_X(F_X^-(\mathbf{U})) \leq F_X(z)], \\ &= \mathbb{P}[\mathbf{U} \leq F_X(z)], \qquad \text{y como } f_{\mathbf{U}}(t) = 1, \\ &= F_X(z) \end{aligned} \tag{4}$$

Demostrando que $\mathbf{Z} = F_X^-(\mathbf{U})$ se distribuye como X.

Problema 2

Implementar el siguiente algoritmo para simular variables aleatorias uniformes:

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

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

A continuación presentamos la implementación en python que hicimos del **Generator Lineal Congruencial** de la descripión del problema.

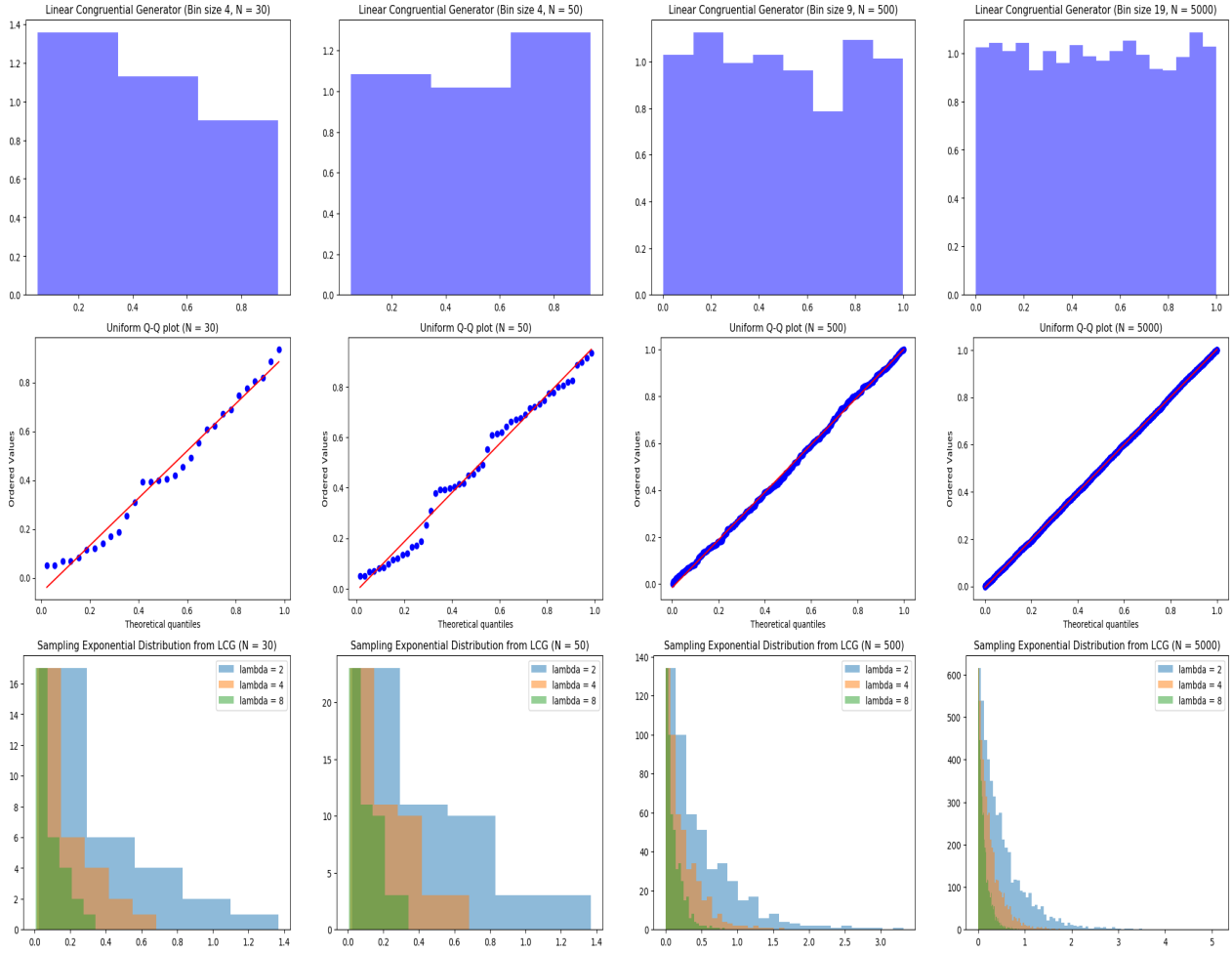
Algorithm 1: Implementación. Extracto de **linear_congruential_generator.py**

```
# Divisor for modulo operation
div = maxInt  # 2^31 - 1

# Simulate for #sample_size of iterations
for idx in range(5, sample_size + 5):
    # Linear recurrence
    x[idx] = 107374182 * x[idx - 1]
    x[idx] += 104420 * x[idx - 5]

    # Modulo operation
    x[idx] = np.mod(x[idx], div)
```

Ahora estudiemos las muestras pseudo-aleatorias que obtuvimos del algoritmo implementado. Presentamos primero un **Histograma** de la muestra al dividirla por el número del modulo $2^{31} - 1$ para transformarla al rango $(0, 1)$ donde esperamos que sea similar a la distribución $\mathcal{U}(0, 1)$. La segunda gráfica es un **QQ-Plot** que compara nuestra muestra con los cuantiles teoricos de una distribución uniforme. La tercera gráfica consiste de **Histogramas** de muestras exponeneciales generadas utilizando su función de distribución acumulada inversa y la muestra pseudo-aleatoria de nuestro generador de números.

Podemos observar de los **QQ-Plots** que la muestra ya se asemejan inequivocamente a una uniforme desde que la muestra es $\mathbf{N = 500}$, aunque las comparaciones de cuantiles para los tamaños de muestra $\mathbf{N = 30, 50}$ ya muestran un gran acercamiento. En los primeros histogramas si podemos ver una mejora sucesiva con el tamaño de la muestra, aunque como sabemos los histogramas pueden ser engañozos. Este mismo comportamiento lo vemos con los histogramas de las muestras exponenciales.

Finalmente aplicamos una prueba de Kolmogrov-Smirnov para probar la hipotesis de si en realidad generamos nuestra muestra de una distribución uniforme en el rango $(0, 1)$. Esta prueba utiliza una estadística que mide la distancióna entre la función de distribución empírica (de los datos) y la teoríca. Presentamos en una tabla los resultados :



Tamaño de Muestra, N =	**Estadística** $D_n = \sup_x \lvert F_n(x) - F(x) \rvert$	**P-Valor**
20	0.281074	0.068618
50	0.094676	0.767865
500	0.030946	0.724532
5000	0.008781	0.835369
50000	0.002950	0.776688

Notemos que utilizando la regla estandar (solo para tener una referencia) de $p-valor \leq 0.05$ podemos rechazar la hipótesis que los datos son una muestra de una distribución de una uniforme. Puesto que esta regla no se aplica en un ninguno de los casos de la tabla, entonces no tenemos suficiente evidencia para rechazar la hipótesis de uniformidad. Esto es algo bueno pues al menos nuestro algoritmo pasa todas estas pruebas sencillas. Si podemos decir que se parece a una $\mathcal{U}(0,1)$.

Problema 3

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

El algoritmo utilizado por la librería **scipy.stats.uniform** es un generador de números pseudo-aleatorios llamado **Mersenne Twister** [4]. Este algoritmo es el estandar para muchos lenguajes de programación y paqueterías de software. A continuación damos una lista de propiedades de este generador :

- Se utiliza comunmente el primo de Mersenne 19937.
- Tiene una periodo de $2^{19937} - 1$.
- Pasa satisfactoriamente muchos tests estádisticos, incluyendo la pruebas de Diehard y algunas de las pruebas TestU01.
- No es criptograficamente seguro. Es posible determinar el patron al observar el mismo número que muestras que el tamaño de su estado interno.

Hay varias formas de fijar la **semilla** para simular variables aleatorias con los métodos de scipy, pero la más facil es utilizar el método **np.random.seed()** de la librería **numpy**. Esto anterior por que **scipy** utiliza internamete el generador de números aleatorios base de **numpy** [3], aunque tiene clases propias que agregan mayor funcionalidad a las distribuciones.

Esto anterior se puede observar al encontrar el método **check_random_state** en el archivo **scipy/_lib/_util.py** [1] del repositorio de **github** para **scipy**, que es utilizado dentro de todas los objetos que simulan variables aleatorias e.g. **scipy.stats.uniform.rvs**.

Algorithm 2: Extracto del método **check_random_state** en scipy.

```
def check_random_state(seed):
    if seed is None or seed is np.random:
        return np.random.mtrand._rand
```

Algorithm 3: Fijar la semilla con Scipy

```
import numpy as np
import scipy

# Fijar la semilla
np.random.seed(seed=1)

# Simular una muestra uniforme
print(scipy.stats.uniform.rvs(size=5))
```

Fijar la semilla en **R** es un mucho más sencillo puesto que solo utilizamas el paquete **base**.

Algorithm 4: Fijar la semilla en R

```
# Fijar la semilla
set.seed(1)

# Simular una muestra uniforme
runif(5)
```

Problema 4

¿En *scipy* que funciones hay para simular una variable aleatoria genérica discreta? ¿tienen preproceso?

Las funciones disponibles en **scipy** para simular variables aleatorias discretas con distintas distribuciones son :

- **Distribución Uniforme Discreta** scipy.stats.randint
- **Distribución Bernoulli** scipy.stats.bernoulli
- **Distribución Binomial** scipy.stats.binom
- **Distribución Boltzmann** scipy.stats.boltzmann
- **Distribución Poisson** scipy.stats.poisson
- **Distribución Geométrica** scipy.stats.geom
- **Distribución Binomial Negativa** scipy.stats.nbinom
- **Distribución Hipergeométrica** scipy.stats.hypergeom
- **Distribución de Laplace Discreta** scipy.stats.dlaplace
- **Distribución Logaritmica** scipy.stats.logser
- **Distribución Zeta** scipy.stats.zipf

No es claro de solamente la documentación de cuantos pasos específicos hay de preprocessamiento al querer obtener una muestra de estas distribuciones, pero al construir el objeto **scipy.stats.rv_discrete** que es la clase base de todas las variables aleatorias discretas que presentamos anteriormente, se definen varias propiedades [5] antes de completar la instancia. Algunos ejemplos de esto son,

- La función de probabilidad **pmf**.
- Si se determino la variable aleatoria discreta con una lista de probabilidades, se tiene que precalcular las sumas de la función **cdf**.
- El logaritmo de estas funciones de masa y distribución.

Con lo que nos queda claro que internamente se definen muchas más funciones y precalculan propiedades más alla de solo determinar como muestrear de esa distribución.

Problema 5

Implementar el algoritmo *Adaptive Rejection Sampling* y simular de una $Gama(2, 1)$ **10,000 muestras**

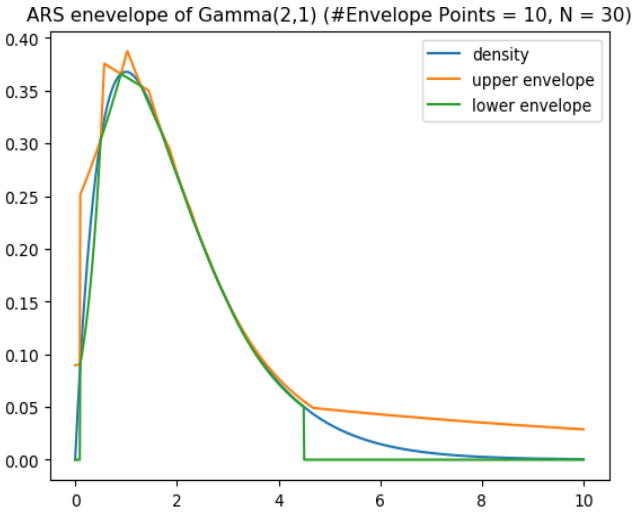
A continuación presentamos la implementación en python del algoritmo **Adaptive Rejection Sampling** que se encuentra en la pag. 57, del libro **Monte Carlo Statistical Methods** [2]. También nos basamos en el ejercicio 2.39 de ese mismo capítulo del libro para saber muestrar de la densidad estimada que envuelve superiormente a nuestra distribución. Esto se puede encontrar en la función **sample_from_upper_adaptive_envelope** en el archivo **adaptive_rejection_sampling.py**.

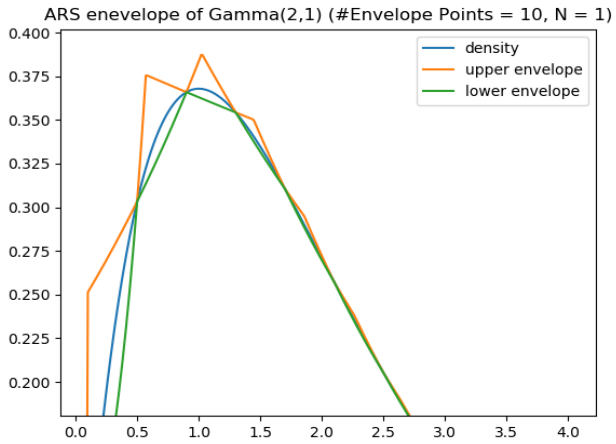
Algorithm 5: Extracto del método **check_random_state** [1]

```
# Muestreamos con una densidad exponencial a pedazos
# que es la envolvente superior
g_sample = sample_from_upper_adaptive_envelope(up_env)

# Uniform (0, 1)
rnd = np.random.rand()

if rnd <= lower(g_sample) / upper(g_sample):  # Acepta
    # Aceptamos la muestra
    ...
elif rnd <= density(g_sample) / upper(g_sample):  # Acepta y refina
    # Aceptamos la muestra y Refinamos nuestra aproximacion
    ...
else:
    # Rechaza la muestra
    ...
```

Para la discretización inicial requeridad para construir las **envolturas** superior e inferior de nuestra densidad utilizamos los puntos $x_1 = 0.1, x_2 = 0.5, x_3 = 0.9, x_4 = 1.3, x_5 = 1.7, x_6 = 2.1, x_7 = 2.5, x_8 = 2.9, x_9 = 3.0, x_{10} = 4.5$ (funcionan bien con la densidad gamma), primero

trabajando con los pares de puntos $(x_i, log(f_X(x_i)))$. Trazamos rectas con estos puntos, y luego transformamos de regreso al espacio original con la función exponencial. Se utiliza la propiedad de log-concavidad de ciertas distribuciones (como la gamma), para poder definir una cota inferior con rectas, puesto que el logaritmo de su densidad es uan función concava. A continuación visualizamos las envolventes resultantes,



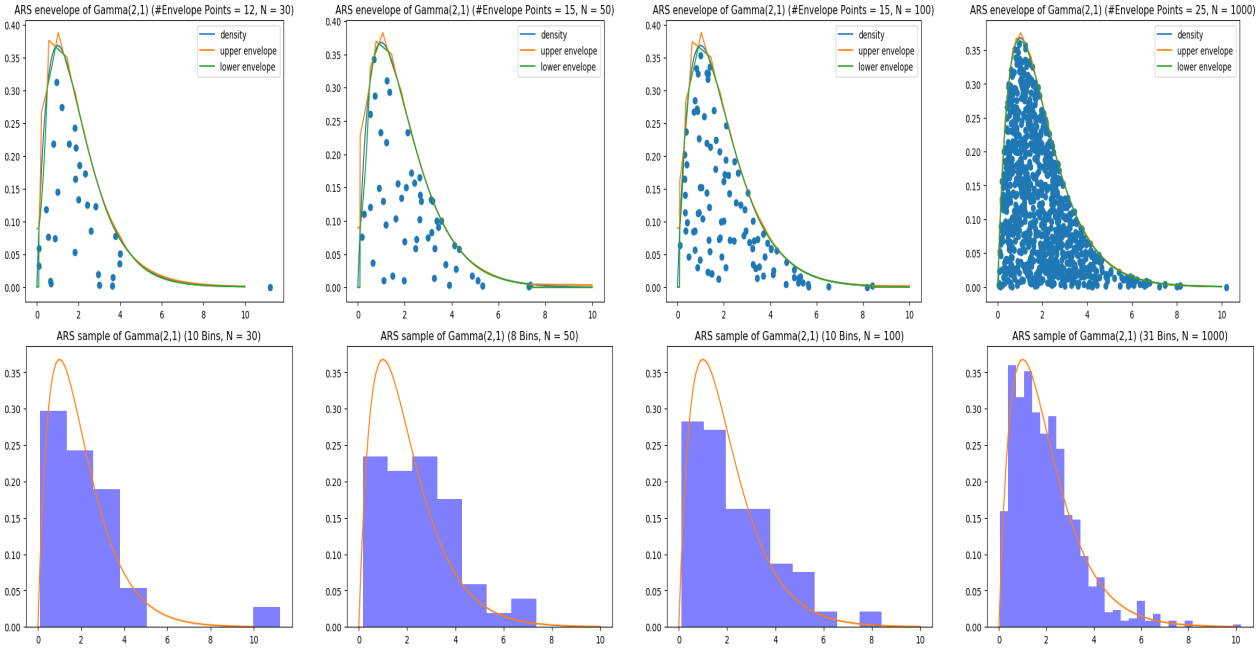


Ahora podemos observar las muestras de puntos que **aceptamos** por caer en el area bajo la densidad y como se fueron haciendo más refinadas nuestras aproximaciones superior e inferior. También podemos observar el histograma de la muestra obtenida, para verificar visualmente que en realidad tiene una distribución **Gamma(0,1)** como esperabamos.

Finalmente, podemos ver el resultado de tomar una muestra de tamaño $\mathbf{N = 50,000}$.

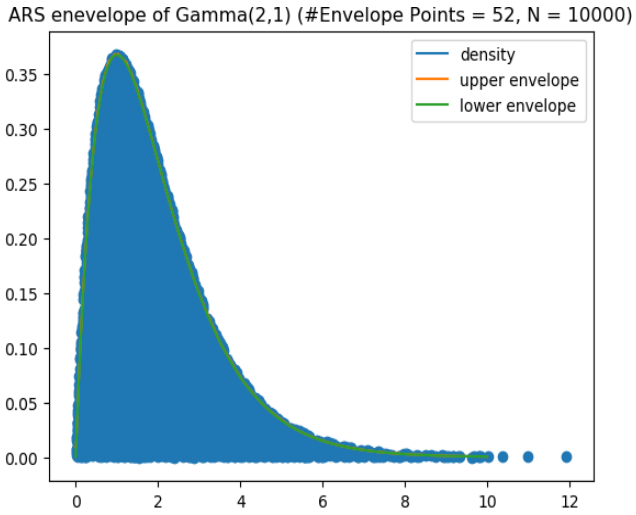
¿Cuándo es conveniente dejar de adaptar la envolvente? La idea del libro [2] es solo **refinar** la aproximación cuando una muestra fue aceptada, pero se tuvo que evaluar la densidad real para determinar esto. Osea que la muestra cayo dentro de la **densidad**, pero no dentro de nuestra envolvente inferior. De modo que solo estamos refinando cuando es necesario hacerlo en regiones de la densidad don la aproximación puede mejorar. Basandonos en esta idea es conveniente dejar de adaptar la envolvente cuando es suficientemente buena para **aceptar** a todas las muestras correctas.

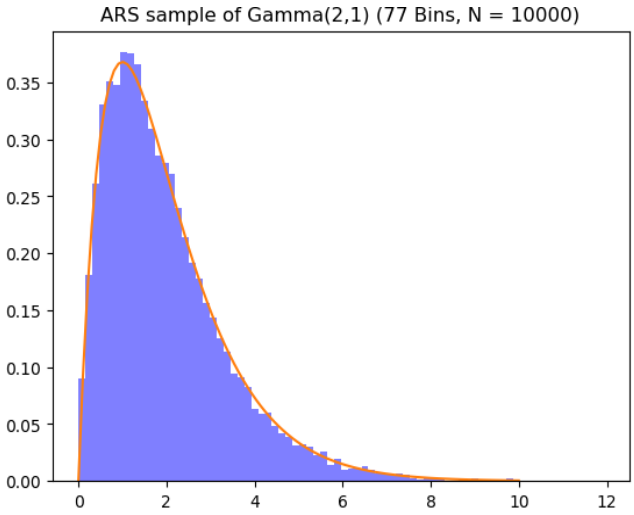
A continuación presentamos una tabla de diferentes ejecuciones del algoritmo **ARS** para simular la distribución **Gamma(0,1)** con diferentes tamaños de muestra. Llamamos **S** al conjunto de puntos que determinan nuestras dos envolventes.



N =	**#S Inicial**	**#S Final**	**Aceptados**	**Rechazados**	**Adaptado**
30	10	12	30	32	2
50	10	15	50	62	5
100	10	15	100	82	5
1000	10	25	1000	662	15
10,000	10	39	10,000	1848	39

Con estos datos también podemos determinar empíricamente que para esta distribución son necesarios alrededor de ≈ 50 puntos en la envolvente inferior para poder simular una muestra de tamaño $\mathbf{N = 10{,}000}$ de manera optima. Osea que con muy poca probabilidad de que se vaya a utilizar la densidad real para **aceptar o rechazar muestras**.





Referencias

[1] Scipy Github Repository, **scipy/_lib/_util.py**
`https://github.com/scipy/scipy/blob/master/scipy/_lib/_util.py`

[2] (Springer Texts in Statistics) - Monte Carlo Statistical Methods (2004), Christian Robert, George Casella

[3] Difference between random draws from scipy.stats...rvs and numpy.random
`https://stackoverflow.com/questions/4001577/difference-between-random-draws-from-scip`

[4] Scipy Documentation, numpy.random.RandomState
`https://docs.scipy.org/doc/numpy-1.16.1/reference/generated/numpy.random.RandomState.`

[5] Scipy Documentation, scipy.stats.rv_discrete
`https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_discrete.html#sci`