

Tarea 1. Descomposición LU y Cholesky

Ricardo Chávez Cáliz

6 de septiembre de 2017

Problema 1. Implementar los algoritmos de *Backward y Forward substitution*.

Se implementó el algoritmo de Backward substitution para la obtención del vector solución X del sistema $tS \cdot X = b$, donde tS denota una matriz no singular de $n \times n$ obtenida al tomar entradas aleatorias arriba de la diagonal con distribución $U(0,1)$ y b es un vector aleatorio de tamaño n con entradas de distribución $U(0,1)$. De igual manera se implementó el algoritmo Forward substitution para resolver $tI \cdot X = b$, donde tI denota una matriz no singular obtenida al transponer una matriz tS como se describió antes. Para verificar que los algoritmos funcionan correctamente se calcula $tI \cdot X - b$ y $tS \cdot X - b$, los cuales deben aproximarse a $0 \in \mathbb{R}^n$.

Problema 2. Implementar el algoritmo de eliminación gaussiana con pivoteo parcial *LUP*, 21.1 del Trefethen (p. 160).

Dicho algoritmo sin pivoteo parcial aparece definido en la función EGPP en el código que se adjunta. La matriz L se modificaba y no siempre quedaba triangular inferior. El algoritmo en EDPP se ejecuta en el mismo orden que cuando se implementa el pivoteo parcial y servirá para el análisis.

Problema 3. Dar la descomposición *LUP* para una matriz aleatoria de entradas $U(0,1)$ de tamaño 5×5 , y para la matriz

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

Se Verifica el algoritmo LUP calculando $PLU - A$. En ambos casos obtenemos dos matrices que se aproximan a 0 (entradas de orden pequeño).

Problema 4. Usando la descomposición *PLU* anterior, resolver el sistema de la forma $Dx = b$ donde D son las matrices del problema 3, para 5 diferentes b aleatorios con entradas $U(0,1)$. Verificando si es o no posible resolver el sistema.

Si $D = PLU$ entonces podemos transformar el sistema de la siguiente manera

$$\begin{aligned} Dx &= b \\ PLUx &= b \\ LUx &= P^{-1}b \end{aligned}$$

Si $y = Ux$ entonces el $Ly = P^{-1}b$ donde $P^{-1}b$ es fácil de calcular dado que P es de permutación. Usando el algoritmo *Forward substitution* podemos calcular el valor de y dado que L es triangular inferior. Ahora podemos solucionar el sistema $Ux = y$ dado que U es triangular superior, y es conocido y se tiene el algoritmo *Backward substitution*. Se calcula para las matrices solicitadas y se comprueba calculando $D \cdot x - b$. En todos los casos obtenemos un vector en \mathbb{R}^n de norma muy pequeña. Se verifica que el sistema tenga solución calculando $\det(D)$ y permitiendo una tolerancia en valor absoluto de 0.001

Problema 5. Implementar el algoritmo de descomposición de Cholesky 23.1 del Trefethen (p. 175).

Es importante señalar que para obtener la forma de Cholesky correcta hay que hacer 0 todos los elementos por debajo de la diagonal. Para verificar que el algoritmo funciona correctamente se obtuvo una matriz A de tamaño $n \times n$ con entradas de distribución $U(0, 1)$. Para asegurarse de que la matriz fuera simétrica y positiva definida se tomó $B = A^t \cdot A + I$. Así denotando por L a la matriz obtenida de B en el algoritmo de Choleski, $L^t \cdot L - B$ debe aproximarse a 0 $\in \mathbb{R}^{n \times n}$.

Problema 6. Comparar la complejidad de su implementación de los algoritmos de factorización de Cholesky y LUP mediante la medición de los tiempos que tardan con respecto a la descomposición de una matriz aleatoria hermitiana definida positiva. Gráficar la comparación.

Para Cholesky se calculó una complejidad computacional de $\frac{m^3}{3}$ mientras que descomposición PLU fue $\frac{2m^3}{3}$. Se calculó el tiempo de ejecución para 200 matrices hermitianas como se describe en el punto 5, para Cholesky y para PLU . Se observa que los tiempo de ejecución se comportan como lo pronosticado teóricamente.

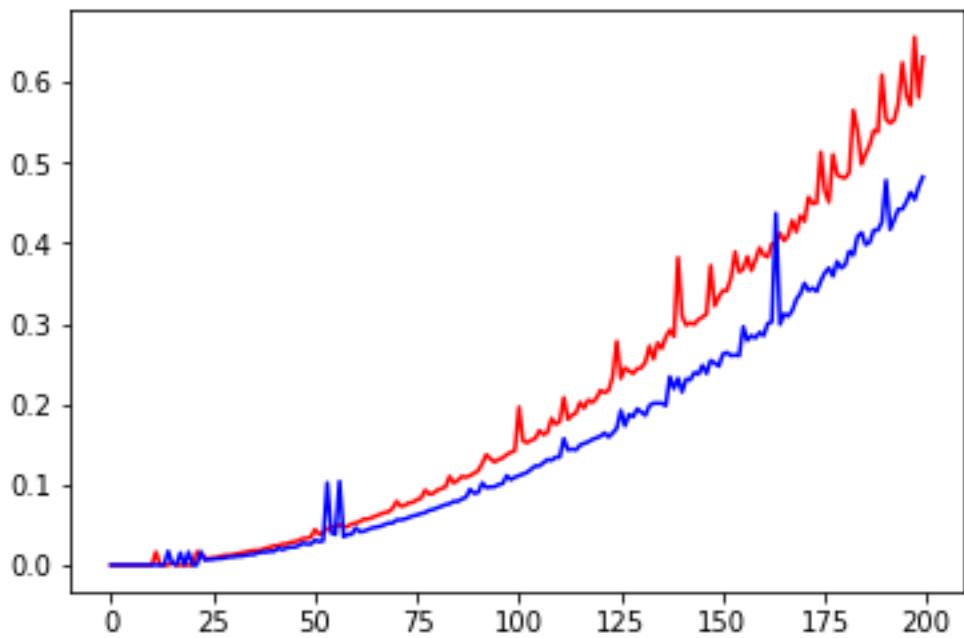


Figura 1: Tiempo de ejecución de Cholesky y descomposición PLU

Tarea 2. Descomposición QR y mínimos cuadrados

Ricardo Chávez Cáliz

September 13, 2017

Problema 1. Implementar el algoritmo de Gram-Schmidt modificado 8.1 del Trefethen (p. 58) para generar la descomposición QR .

Se implementó dicho algoritmo a una matriz A de entradas aleatorias $\sim U(0, 1)$ para la obtención de matrices Q y R correspondientes a la descomposición QR de A . Es decir:

1. $A = QR$
2. R es triangular superior
3. $Q^t \cdot Q = I$

Para verificar que 1) se satisface se calcula QR y se compara con A usando el método *allclose* de numpy. Para 2) se usa el método *VerifierTS*. El punto 3) se verifica calculando $Q^t \cdot Q$ y se compara con la matriz identidad con el método *allclose*. Se realiza esta

Problema 2. Implementar el algoritmo que calcula el estimador de mínimos cuadrados en una regresión usando la descomposición QR.

Se implementó el algoritmo en la función *estimadorMC*, el cual halla b tal que $\|Y - Xb\|_2$ sea mínimo, donde X es matriz de diseño obtenida de un vector de observaciones y y Y es el vector aleatorio a estimar. El residuo $r = Y - X\beta$ es mínimo cuando $r \in \text{Null}(P)$ donde P es un proyector ortogonal con $\text{rango}(P) = \text{rango}(X)$ esto implica que $P \cdot r = 0$, por lo tanto $X\beta = Py$. La proyección ortogonal es obtenida con $P = Q \cdot Q^t$, donde Q viene de la descomposición QR de X .

$$\begin{aligned} X \cdot \beta &= P \cdot Y \\ Q \dot{R} \cdot \beta &= Q \cdot Q^t \cdot Y \\ R \cdot \beta &= Q^t \cdot Y \end{aligned}$$

Para hacer esto se llama al método que construye la matriz de diseño con un parámetro p , y a algún método que calcula descomposición QR de esta (el descrito en el punto 1 ó el propio de Numpy). En la última ecuación R es triangular superior entonces es posible resolver β usando el método *BackwardSubst*, de esta manera encontramos β y $X \cdot \beta$ será la mejor aproximación de Y .

Problema 3. Generar \mathbf{Y} compuesto de $y_i = \sin(x_i) + \epsilon_i$ donde $\epsilon_i \sim N(0, \sigma)$ con $\sigma = 0.1$, para $x_i = \frac{4\pi i}{n}$ para $i = 1, \dots, n$.

Hacer un ajuste de mínimos cuadrados a \mathbf{Y} , con descomposición QR , ajustando un polinomio de grado $p - 1$.

- Considerar los 12 casos: $p = 3, 4, 5, 100$ y $n = 100, 1000, 10000$.
- Graficar el ajuste en cada caso.
- Medir tiempo de ejecución de su algoritmo, comparar con descomposición QR de scipy y graficar los resultados.

Se consideran los casos pedidos y se obtienen los estimadores de Y como se muestra en Figuras 1,2 y 3. Se observa que la elección del parámetro p es delicada, dado que para p muy pequeños no se obtiene una buena aproximación (estamos aproximando por un polinomio de grado pequeño) pero si p es demasiado grande, entonces la aproximación tampoco es buena.

En Figura 4, se observan las distintas aproximaciones de Y cuando $p = 10$ y n varía. En este caso se puede ver que 10 es un parámetro apropiado para aproximar Y .

Se midieron los tiempos para la estimación de mínimos cuadrados y para el algoritmo desarrollado en el punto 1 y el dado por scipy. Se presentan aquí en la Figura 5, graficando el tiempo de ejecución a medida que el n crece. Dejando p fijo en 2 para reducir el tiempo de ejecución de cada estimación y tomando n hasta 100.

Como el tiempo de ejecución depende del Y que tiene un ruido aleatorio, la gráfica presenta un sesgo. Para evitar esto se ejecuta 10 veces para cada tamaño y se toma la mediana (representa de mejor manera a los datos en este caso) y se grafica tomando este en cuenta. Vease Fig. 6 para esto.

Problema 4. Hacer $p = 0.1n$, o sea, diez veces más observaciones que coeficientes en la regresión, ¿Cuál es la n máxima que puede manejar su computadora?

Fue capaz de ejecutar para $n = 10128$ con un tiempo de 610.720312569. Generando los siguientes Runtimewarnings

- `RuntimeWarning: overflow encountered in double-scalars`
- `RuntimeWarning: overflow encountered in subtract`
- `RuntimeWarning: invalid value encountered in divide`

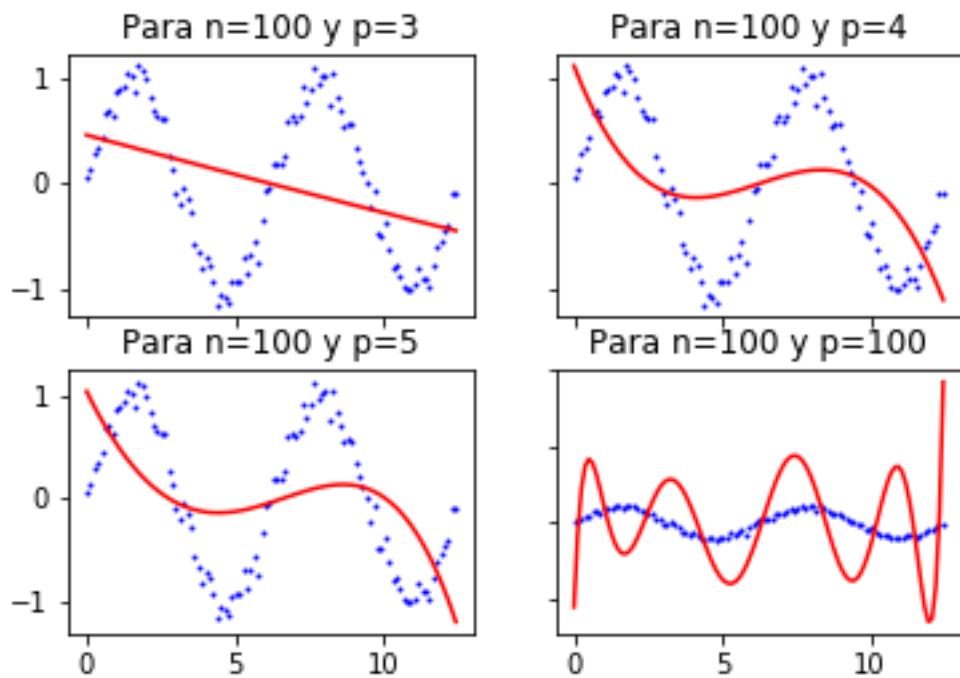


Figure 1: Estimación de mínimos cuadrados con 100 puntos y distinta p

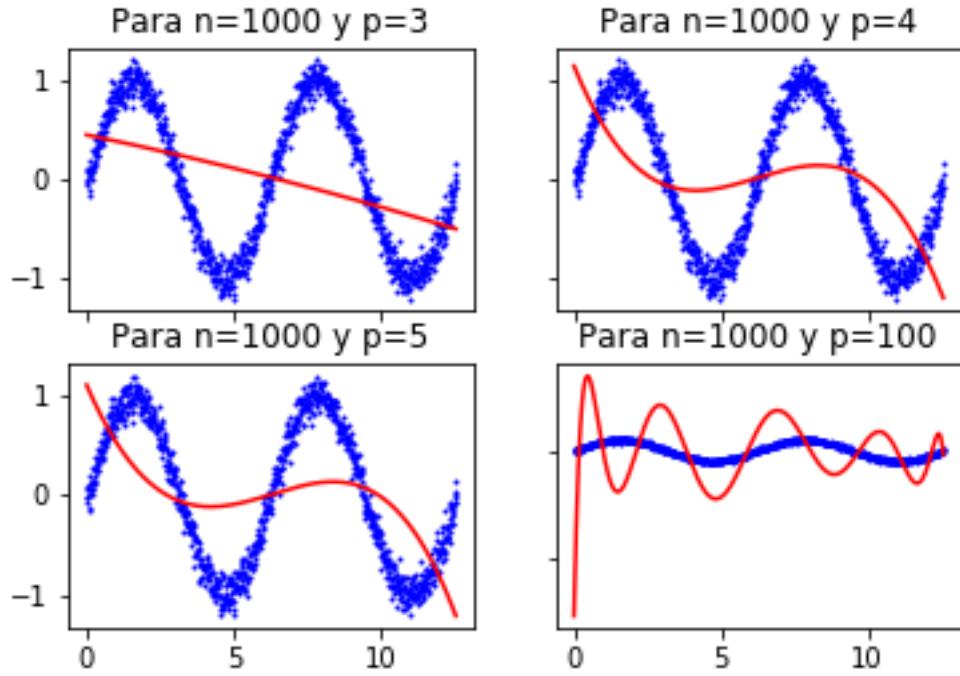


Figure 2: Estimación de mínimos cuadrados con 1000 puntos y distinta p

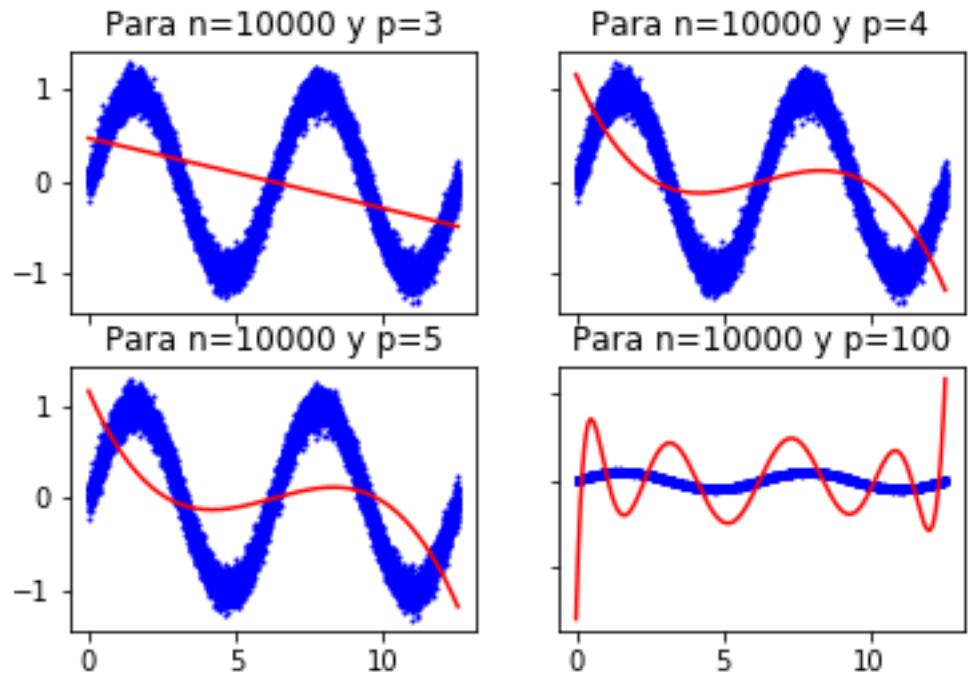


Figure 3: Estimación de mínimos cuadrados con 10000 puntos y distinta p

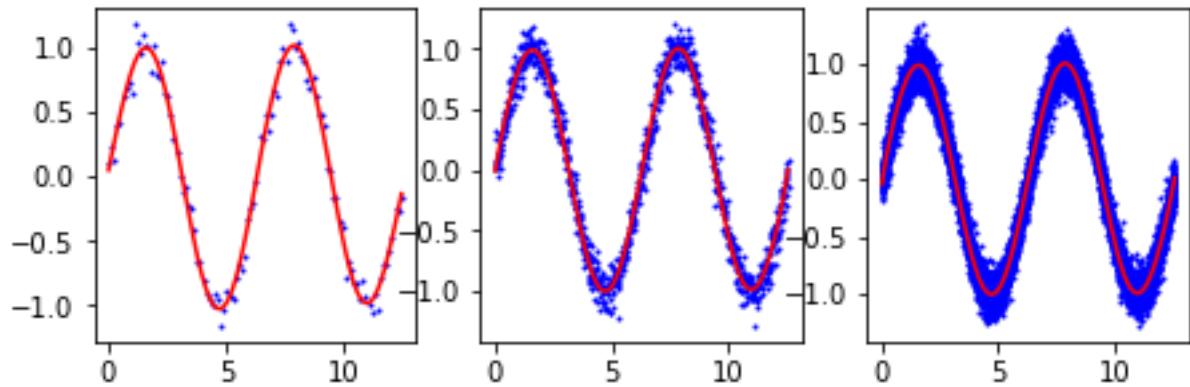


Figure 4: Estimación de mínimos cuadrados con $p=10$ y distinta n

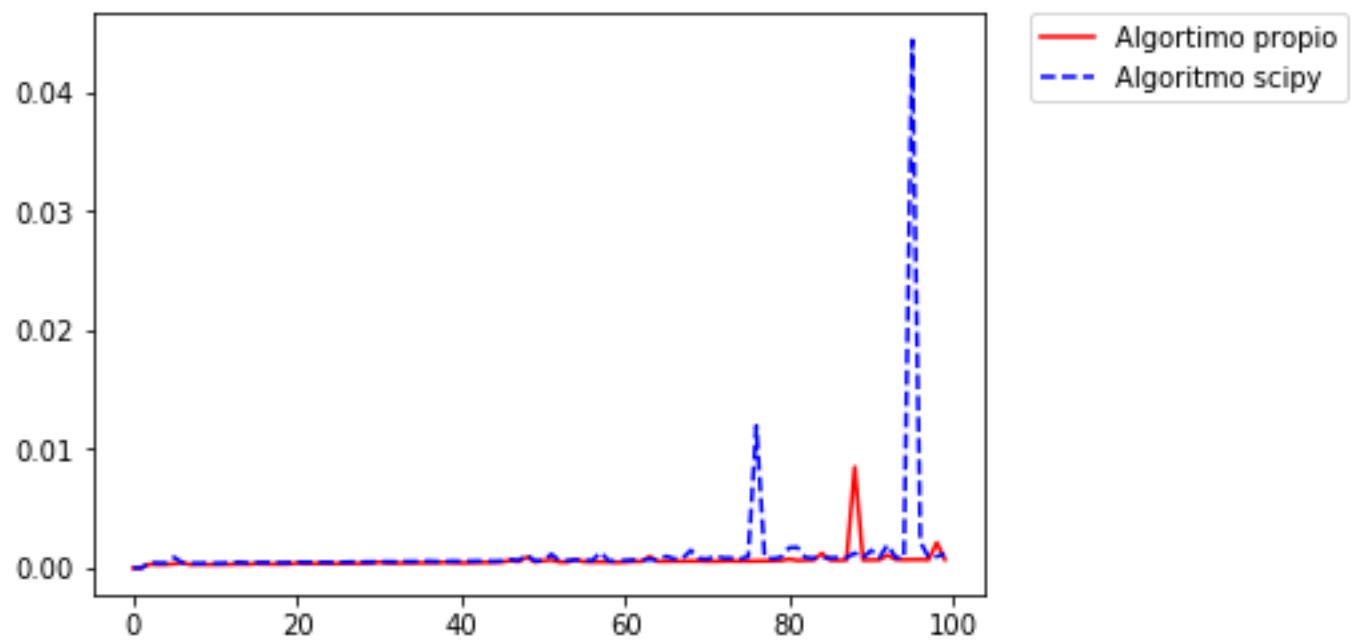


Figure 5: Tiempos de ejecucción para el algoritmo propio y el de scipy

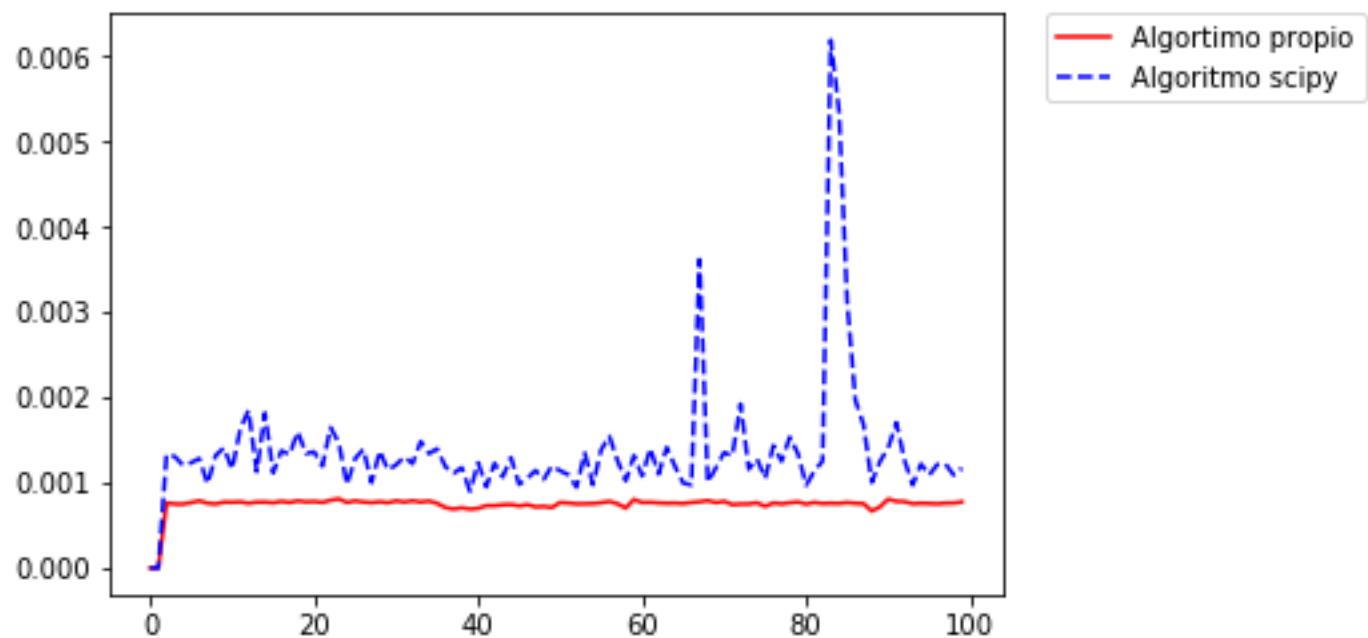


Figure 6: Tiempos medios de ejecucción para el algoritmo propio y el de scipy

Tarea 3. Condicionamiento y estabilidad

Ricardo Chávez Cáliz

September 20, 2017

Problema 1. Sea A una matriz de tamaño 50×20 , aleatoria y fija, calcular su descomposición QR . Sean $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_{20} \geq 0$ y

$$B = Q^* \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_{20}) Q \quad \text{y} \quad B_\varepsilon = Q^* \text{diag}(\lambda_1 + \varepsilon_1, \lambda_2 + \varepsilon_2, \dots, \lambda_{20} + \varepsilon_{20}) Q,$$

con $\varepsilon_i \sim N(0, \sigma)$, con $\sigma = 0.01\lambda_1$.

1. Comparar la descomposición de Cholesky de B y de B_ε usando el algoritmo de la tarea
 1. Considerar los casos cuando B tiene un *buen* número de condición y un *mal* número de condición.
 2. Con el caso mal condicionado, comparar el resultado de su algoritmo con el del algoritmo de Cholesky de scipy.
 3. Medir el tiempo de ejecución de su algoritmo de Cholesky con el de scipy.

1.1 El número de condición de una matriz A es $\kappa(A) = \|A\| \cdot \|A^{-1}\|$. Cuando $\|\cdot\| = \|\cdot\|_2$ entonces

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

En este caso los eigenvalores de B están dados por $\lambda_1 \dots \lambda_{20}$. Para controlar el condicionamiento de B se tomaron $\lambda_i = 1, \forall i \in \{2, \dots, 20\}$ y $\lambda_1 = f(n)$ donde $f(n)$ es una función creciente, con $f(n) > 1$. De esta manera para cada n tendremos B_n con

$$\kappa(B_n) = \lambda_{\max} = \lambda_1 = f(n)$$

Cuando $\lambda_1 = 1$ entonces B estará bien condicionada.

Para comparar la descomposición de Choleski de B_n y $B_{n,\epsilon}$ se calculó $\|U_n - U_{n,\epsilon}\|$ para cada n , donde U_n proviene de la descomposición de Choleski de B_n y $f(n) = n$ (es decir, el valor de λ_1 crece de manera lineal). Se graficó λ_1 contra $\|U_n - U_{n,\epsilon}\|$ (triángulos) hasta $n=5000$, dejando para cada n $\|U_1 - U_{1,\epsilon}\|$, para así tener una comparativa del caso bien condicionado contra el mal condicionado. Además se graficó $\log(\|U_n - U_{n,\epsilon}\|)$ respecto a λ_1 . Ver Figuras 1 y 2, en las cuales se puede apreciar la diferencia en el caso mal condicionado y el bien condicionado.

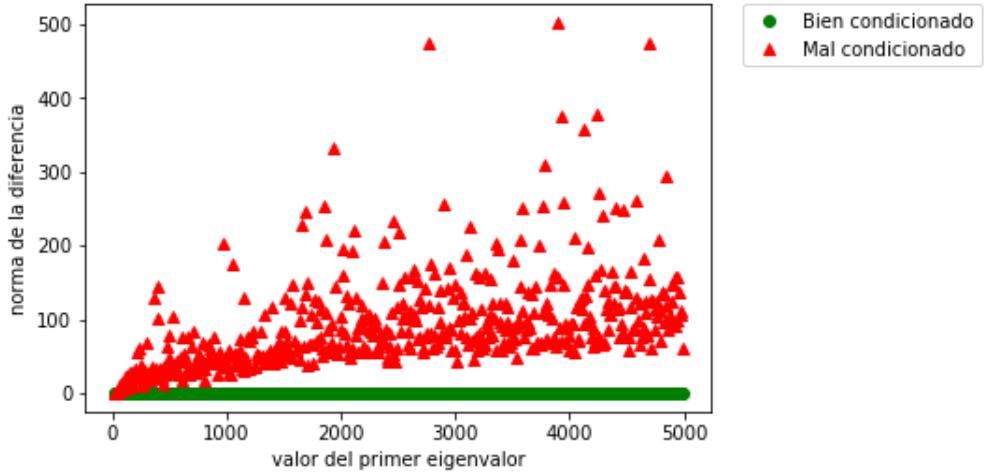


Figure 1: Gráfica de $\|U_n - U_{n,\epsilon}\|$ comparado con $\|U_1 - U_{1,\epsilon}\|$

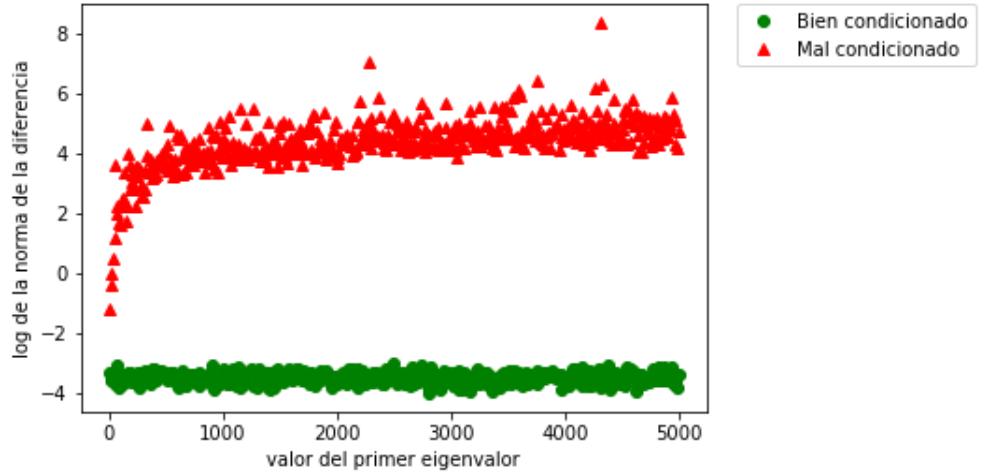


Figure 2: Gráfica de $\log(\|U_n - U_{n,\epsilon}\|)$ comparado con $\log(\|U_1 - U_{1,\epsilon}\|)$

Como $U_{n,\epsilon}$ está construido de manera aleatoria, las gráficas recién presentadas son diferentes en cada ejecución. Para conseguir gráficas *típicas* se ejecutó lo anterior 30 veces y se consideró el valor medio en cada $\|U_n - U_{n,\epsilon}\|$. Se tomó el valor medio para encontrar la tendencia central, por el sesgo presente en $\|U_n - U_{n,\epsilon}\|$. Se tomó hasta $n = 1000$ para reducir el tiempo de ejecución. Se obtuvieron las siguientes gráficas, que muestran la evidencia de antes, pero con menos ruido.

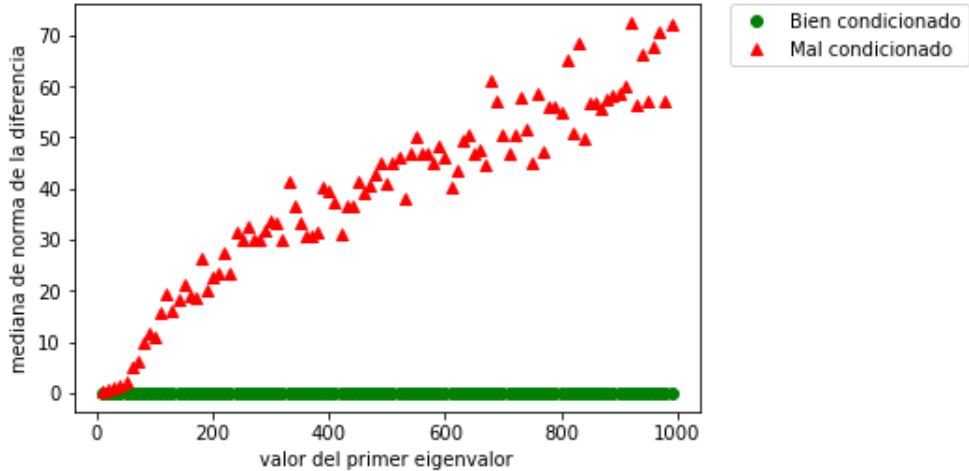


Figure 3: Gráfica del valor medio de $\|U_n - U_{n,\epsilon}\|$ comparado con $\|U_1 - U_{1,\epsilon}\|$

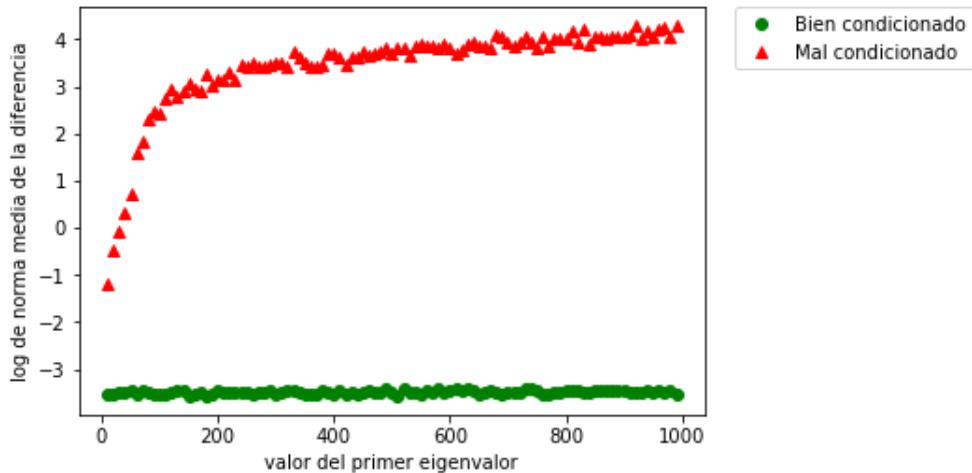


Figure 4: Gráfica del logaritmo del valor medio de $(\|U_n - U_{n,\epsilon}\|)$ comparado con $\log(\|U_1 - U_{1,\epsilon}\|)$

Dado que la perturbación está dada en términos de λ_1 , se pudiera pensar que el fenómeno reflejado en las gráficas anteriores es propio de dicho aumento en la perturbación de B . Para tomar en cuenta esto, consideremos los errores relativos. Además recuerde que el interés en comparar la descomposición de Choleski entre B y B_ϵ radica en estudiar el número de condición asociado al problema de obtener la descomposición de Cholesky de una matriz dada. Es decir, buscamos c en

$$c \frac{\|B - B_\epsilon\|}{\|B\|} = \frac{\|U - U_\epsilon\|}{\|U\|}$$

Por lo tanto para cada n se calculó el valor medio de

$$\frac{\|U - U_\epsilon\|}{\|U\|} \cdot \frac{\|B\|}{\|B - B_\epsilon\|}$$

en un experimento con 30 repeticiones y se obtuvo el siguiente resultado graficando como antes.

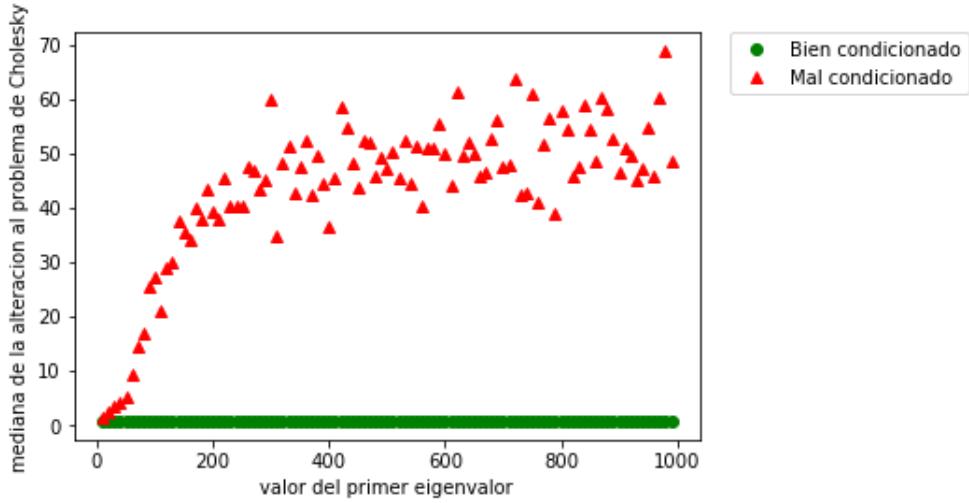


Figure 5: Gráfica del valor medio de $\frac{\|U_n - U_{n,\epsilon}\|}{\|U_n\|} \cdot \frac{\|B\|}{\|B - B_\epsilon\|}$ comparado con $\frac{\|U_1 - U_{1,\epsilon}\|}{\|U_1\|} \cdot \frac{\|B\|}{\|B - B_\epsilon\|}$

La gráfica anterior da evidencia para decir que el condicionamiento al problema de Cholesky está relacionado con el número de condición de la matriz a descomponer.

1.2 Al intentar usar el algoritmo de `scipy`, se obtiene un error: (*LinAlgError: n-th leading minor not positive definite*). Lo cual quiere decir que para matrices mal condicionadas el algoritmo de `scipy` falla. Para verificar en qué punto falla se intentó decomponer B_n para $n \in \{0, 1, \dots, 100\}$ y se verifica el primer n que genera un error. Para cada ejecución este número es diferente. Al realizar 100 pruebas el promedio fue: 41.54.

1.3 Para cada n entre 1 y 200 se generó una gráfica aleatoria de tamaño n , la cual se modificó para ser simétrica y definida positiva. Se midió el tiempo de ejecución de ambos algoritmos y graficando en escala logarítmica se obtuvo el siguiente resultado.

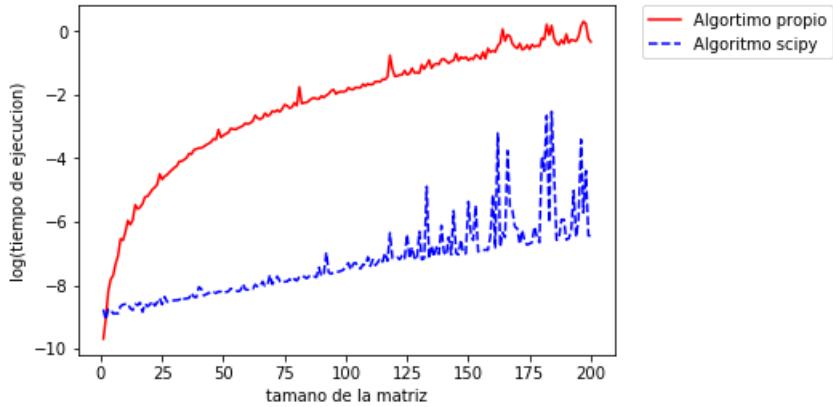


Figure 6: Diferencia en tiempos del algoritmo implementado en tarea 1 y el importado de `scipy`

En la figura se puede apreciar la eficiencia del algoritmo de scipy comparado con el implementado en la tarea 1. La "desventaja" de este es que no es capaz de realizar la descomposición para matrices mal condicionadas.

Problema 2. Resolver el problema de mínimos cuadrados,

$$y = X\beta + \varepsilon, \quad \varepsilon_i \sim N(0, \sigma)$$

usando su implementación de la descomposición QR ; β es de tamaño $d \times 1$ y X de tamaño $n \times d$.

Sean $d = 5, n = 20, \beta = (5, 4, 3, 2, 1)^t$ y $\sigma = 0.1$.

1. Hacer X con entradas aleatorias $U(0, 1)$ y simular y . Encontrar $\hat{\beta}$ y compararlo con el obtenido $\hat{\beta}_p$ haciendo $X + \Delta X$, donde las entradas de ΔX son $N(0, \sigma = 0.01)$. Comparar a su vez con $\hat{\beta}_c = ((X + \Delta X)^t(X + \Delta X))^{-1}(X + \Delta X)y$ usando el algoritmo genérico para invertir matrices `scipy.linalg.inv`.
2. Lo mismo que el anterior pero con X mal condicionada (ie. con casi colinealidad).

2.1 Una manera de comparar $\hat{\beta}, \hat{\beta}_p$ y $\hat{\beta}_c$ es calculando $\|\hat{\beta} - \hat{\beta}_p\|$ y $\|\hat{\beta}_p - \hat{\beta}_c\|$. Se realizó este cálculo con la norma usual. Para esto se tomaron 1000 repeticiones y se obtuvieron los siguientes promedios: de $\|\hat{\beta} - \hat{\beta}_p\|_2$ fue 0.234556096125, el cual representa un error de 0.0475338501589 respecto a la entrada más grande de β ; de $\|\hat{\beta}_p - \hat{\beta}_c\|_\infty$ fue $2.15743201733e-14$ el cual representa un error de $4.31486403467e-15$ respecto a la entrada más grande de β . La comparación de $\hat{\beta}$ y $\hat{\beta}_c$ se deduce de una cota dada por desigualdad triangular. Dado que $\|\hat{\beta}_p - \hat{\beta}_c\|_\infty$ es en promedio $2.15743201733e-14$ (muy pequeña). Podemos reducir a analizar solamente $\hat{\beta}$ y $\hat{\beta}_p$

Pensando el problema de los mínimos cuadrados como una función $f : V \rightarrow S$ donde V es el espacio normado de los datos y S es el espacio normado de las soluciones, con $f(X) = \hat{\beta}$ y $f(X + \delta X) = \hat{\beta}_p$, la manera apropiada de comparar $\hat{\beta}$ y $\hat{\beta}_p$ será usando el número de condición asociado a f , tomando $\|\cdot\| = \|\cdot\|_2$ tenemos:

$$c_f = \frac{\|f(X) - f(X + \Delta X)\|}{\|f(X)\|} \cdot \frac{\|X\|}{\|X - (X + \Delta X)\|} = \frac{\|\hat{\beta} - \hat{\beta}_p\|}{\|\hat{\beta}\|} \cdot \frac{\|X\|}{\|\Delta X\|}$$

Para esto, de igual manera se tomaron 1000 repeticiones y se obtuvo un c_f promedio de 1.83679946719. Es decir, que en promedio para una muestra de 1000 matrices con entradas como se establece, se tiene que el error relativo en el *output* no es más grande que el doble del error relativo en el *input*

2.2 Para el caso mal condicionado se tomaron 1000 repeticiones y se obtuvieron los siguientes promedios: de $\|\hat{\beta} - \hat{\beta}_p\|_2$ fue 9.91928629595, el cual representa un error de 1.98385725919 respecto a la entrada más grande de β ; de $\|\hat{\beta}_p - \hat{\beta}_c\|_\infty$ fue $7.87736910659e-12$ el cual representa un error de $1.57547382132e-12$ respecto a la entrada más grande de β . Se obtuvo que el promedio de c_f fue 38.8253247212 (representa un cambio grande en el error relativo de entrada y el de salida).

En base a lo observado se puede decir que el condicionamiento de X influye en gran manera en el error relativo de output. y que en estos casos es mejor aproximar $\hat{\beta}_p$ con $\hat{\beta}_c$.

Tarea 4. Calculo de eigenvalores

Ricardo Chávez Cáliz

October 4, 2017

Problema 1. Dado el siguiente

Teorema (Gershgorin):

Dada una matriz $A = a_{ij}$ de $m \times m$, cada eigenvalor de A está en al menos uno de los discos en el plano complejo con centro en a_{ii} y radio $\sum_{j \neq i} |a_{ij}|$. Además, si n de estos discos forman un dominio conexo, disjunto de los otros $m - n$ discos, entonces hay exactamente n eigenvalores en ese dominio.

Deduce estimaciones de los eigenvalores de

$$A = \begin{pmatrix} 8 & 1 & 0 \\ 1 & 4 & \epsilon \\ 0 & \epsilon & 1 \end{pmatrix}$$

Para la matriz A las estimaciones dependen completamente del valor de ϵ . Mientras más pequeño sea ϵ las estimaciones serán más precisas. Conversamente para grandes valores de ϵ las estimaciones que provee el teorema son imprácticas. De manera concreta, tenemos $\sum_{j \neq i} |a_{ij}| = 2 + 2|\epsilon|$, en este caso los valores propios se encuentran en el interior de los discos con centros en 1, 4, 8. Como $\sum_{j \neq i} |a_{ij}| \geq 2$ y $\max\{d(a, b) | a, b \in \{1, 4, 8\}\} = 4$ los discos cerrados siempre forman un dominio conexo, por lo tanto la segunda parte del teorema no nos da información adicional. La siguiente figura ejemplifica el dominio que hace referencia el teorema.

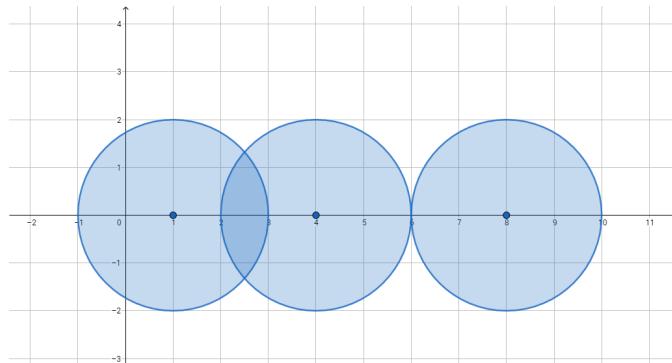


Figure 1: Se muestran discos en el plano complejo con centros en 1, 4 y 8 de radio 2 (mayor precisión a obtener)

En el caso en que A es hermitiana y por tanto sus eigenvalores son reales, entonces se tiene que $-1 - 2\epsilon \leq \lambda_i \leq 10 + 2\epsilon$

Problema 2. Implementa la iteración QR con shift. Aplícalo a la matriz A del Ejercicio 1 con $\epsilon = 10^N$ para $N = 1, \dots, 5$.

Se implementaron los algoritmos de iteración QR sin desplazamiento, con desplazamiento simple ($\sigma = a_{n,n}$) y con el desplazamiento de Wilkinson. Para este último si B es el menor de 2×2 del extremo inferior derecho de la matriz:

$$B = \begin{pmatrix} a_{n-1,n-1} & b_{n-1} \\ b_{n-1} & a_{n,n} \end{pmatrix}$$

El desplazamiento de Wilkinson es el eigenvalor de B más cercano a $a_{n,n}$. Una formula numéricamente estable para calcular el desplazamiento de Wilkinson es:

$$\mu = a_{n,n} - \frac{\text{signo}(\delta)b_{n-1}^2}{|\delta| + \sqrt{\delta^2 + b_{n-1}^2}}$$

donde $\delta = (a_{n-1,n-1} - a_{n,n})/2$

En caso de que para la iteración 100,000 no se tenga A_k cercana a una matriz diagonal ($\|A_k - \text{diag}(A)\|_\infty < 1e-8$) entonces se detiene el proceso y se imprime $\|A_k - \text{diag}(A)\|_\infty$ para saber que tan cerca se estaba de la convergencia.

Se obtuvieron los siguientes resultados

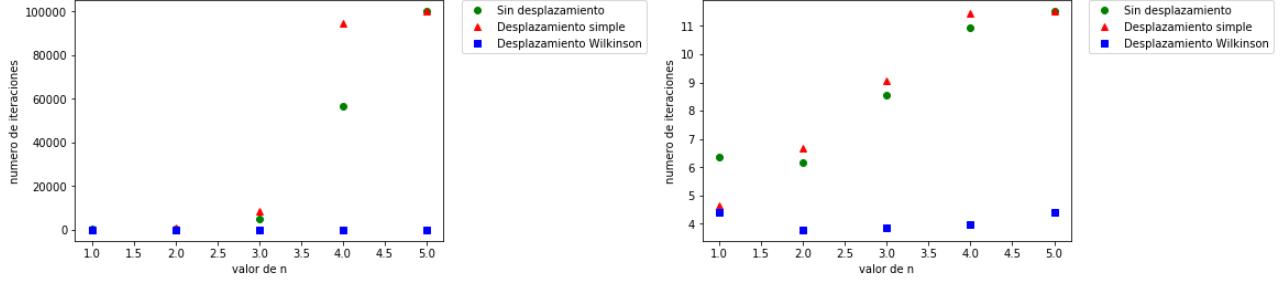


Figure 2: Se muestran el número de iteraciones requeridas en cada caso, hasta que A_k fue suficientemente cercana a una matriz diagonal ($\|A_k - \text{diag}(A)\|_\infty < 1e-8$), en escala normal y logarítmica

Para $n = 5$ la iteración QR sin desplazamiento y con desplazamiento simple rebasaron el límite establecido. Con $k = 100,000$ se obtuvo para iteración QR $\|A_k - \text{diag}(A)\|_\infty = 1347.3598064$, mientras que para iteración QR con desplazamiento simple $\|A_k - \text{diag}(A)\|_\infty = 9931.55732038$. Es decir que en ambos casos hacían falta aún muchas iteraciones para asegurar un buen cálculo de eigenvalores. Esto aunado con lo que se puede observar en la gráfica, se puede decir que la iteración QR sin desplazamiento fue más eficiente que la iteración QR con desplazamiento simple.

En la Figura 2, debido a la gran diferencia de iteraciones de cada método, no es posible apreciar la ejecución con el desplazamiento de Wilkinson, el cual es de nuestro interés por ser el más eficiente. Por eso se muestra la siguiente gráfica.

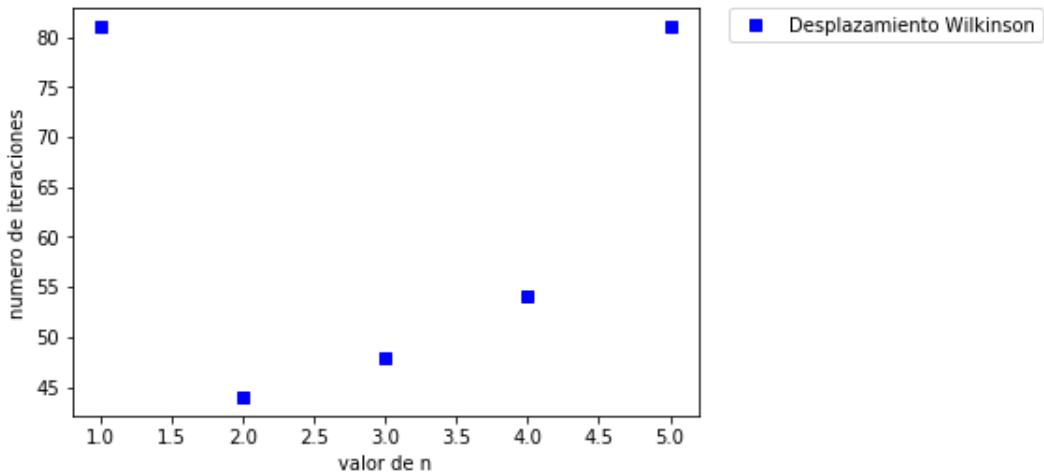


Figure 3: Se muestran el número de iteraciones requeridas en cada caso, hasta que A_k fue suficientemente cercana a una matriz diagonal ($\|A_k - \text{diag}(A)\|_\infty < 1e-8$)

Se observa una anomalía para $n = 1$, dado que requiere una cantidad elevada de iteraciones para lograr la convergencia. Por esto se comparó su tiempo de ejecución con el tiempo de ejecución del método para calcular eigenvalores de la librería de *numpy*. En el que se presenta la misma anomalía, a continuación se muestran los resultados.

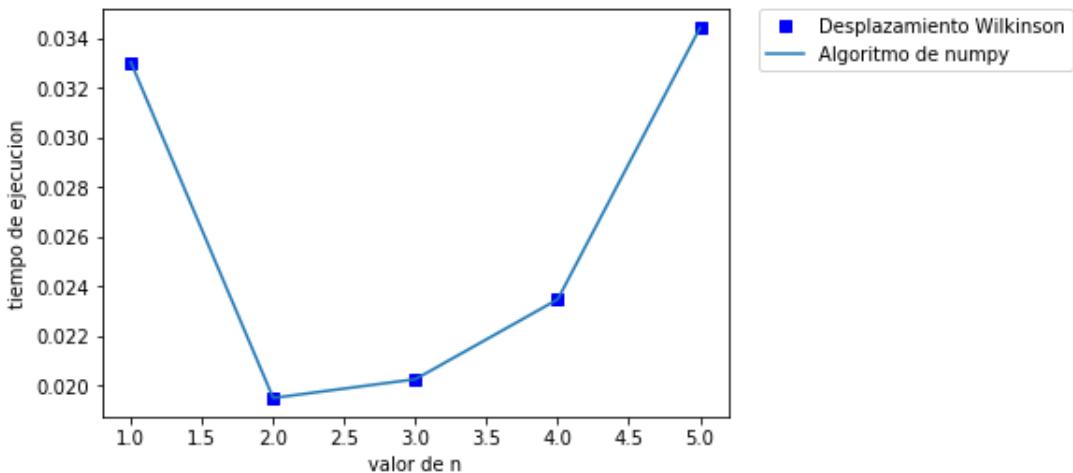


Figure 4: Se muestran tiempos de ejecución de el algoritmo de iteración Qr con desplazamiento de Wilkinson contra el algoritmo de numpy

También se compararon los eigenvalores obtenidos en ambos métodos usando el método *allclose* de numpy, obteniendo *true* en cada caso.

Problema 3. Determina todos los eigenvalores y eigenvectores de una matriz de Householder.

Las matrices de Householder son ortogonales y simétricas.

Si A es ortogonal entonces $A \cdot A^t = Id$ y si λ es un eigenvalor de A con eigenvector v entonces $Av = \lambda v$, y tenemos que:

$$\|v\| = vv^t = v^t A^t Av = (Av)^t Av = (\lambda v)^t \lambda v = |\lambda|vv^t = |\lambda| \cdot \|v\|$$

Por lo tanto los eigenvalores de matrices ortogonales tienen valor absoluto 1 (norma compleja), es decir que la multiplicación por matrices ortogonales es una isometría.

Dado que la matriz de Householder es real y simétrica, sus eigenvalores son reales y por lo tanto deben ser 1 o -1.

Dado que $Hu = u - 2u(u^t u) = -u$ (reflexión), hay al menos algún eigenvalor -1, y en efecto los vectores v perpendiculares a u satisfacen $Hv = v - 0u = v$ (proyección). De manera que hay una base completa de eigenvectores, uno para el eigenvalor -1 y el resto para los eigenvalores 1.

Es decir que una matriz de Householder tiene por valores propios $(-1, 1, \dots, 1)$

Para verificar lo anterior se generó una muestra de 10 vectores aleatorios de tamaño 5 y se obtuvieron los eigenvalores para las matrices de Householder construidas a partir de dicha muestra, usando el algoritmo de iteración QR con desplazamiento implementado anteriormente. Obteniendo en cada caso $[-1., 1., 1., 1., 1.]$

Problema 4. Demuestra que no es posible construir la transformación de similaridad del teorema de Schur con un número finito de transformaciones de similaridad de Householder.

Para dar contexto a la pregunta, recordemos que el determinante y los eigenvalores son invariantes bajo transformaciones de similitud. Por tanto una buena idea para aumentar la eficiencia de los algoritmos que calculan eigenvalores es buscar encontrar un representante sencillo en la misma clase de equivalencia por similitud de la matriz dada. Recordemos el teorema de Schur.

Teorema. Para toda matriz compleja A de $n \times n$ existen matrices S y T de $n \times n$ con S invertible y T triangular superior, tales que:

$$A = S^{-1} \cdot T \cdot S$$

La idea es aplicar transformaciones de similaridad a A de manera que podamos introducir ceros bajo la diagonal. Conocemos la matriz de Householder asociada a un vector v , la cual es una proyección ortogonal sobre v y una reflexión, por lo que una idea muy natural es usar dichas matrices una tras otra para lograr dicha descomposición.

La primer reflexión de Householder Q_1^* , multiplicada por la izquierda a A , introduce ceros por debajo de la diagonal en la primera columna de A . Ejemplificamos esto en los siguientes diagramas.

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix} \xrightarrow{Q_1^*} \begin{pmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{pmatrix}$$

Para completar la transformación de similaridad, también debemos multiplicar por Q_1 en la derecha de A :

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{pmatrix} \xrightarrow{Q_1^*} \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}$$

Lo cual tiene el efecto de reemplazar cada columna de la matriz por una combinación lineal de todas las columnas. El resultado es que los ceros inducidos anteriormente no vuelven a aparecer.

Problema 5. ¿Qué pasa si aplicas la iteración QR sin shift a una matriz ortogonal?

Recordemos que la iteración QR calcula una sucesión de matrices A_k de la siguiente manera. Se empieza con $A_0 = A$ y en la iteración k se calcula la descomposición QR de A

$$Q_k R_k = A_{k-1}$$

y luego se toma el producto invertido

$$A_k = R_k Q_k$$

Pero si A es una matriz ortogonal entonces $Q_1 = A$ y $R_1 = Id$ por lo tanto $A_1 = R_1 \cdot Q_1 = Id \cdot A = A$, y por lo tanto $A_k = A$ para toda $k \in \mathbb{N}$. Por lo que los elementos fuera de la diagonal no decrecen, y a menos de que hayamos empezado con el caso trivial ($A = Id$), A_k no converge a una matriz diagonal.

Para comprobar lo recién demostrado se construyó una matriz ortogonal y al intentar aplicar la iteración sin desplazamiento se obtuvo después de 100,000 iteraciones que los elementos fuera de a diagonal permanecieron igual.

```
In [266]: runfile('C:/Users/Pc/Documents/CC/tarea4.py', wdir='C:/Users/Pc/Documents/CC')
Matriz ortogonal de ejemplo
[[ 5.  5.]
 [-5.  5.]]
El número de iteraciones ha exedido el límite establecido
norma de A-diag(A) es 5.0
```

Figure 5: Mensaje obtenido en la terminal al tratar de aplicar iteración QR sin desplazamiento a una matriz ortogonal

Tarea 5. Simulación Estocástica, introducción

Ricardo Chávez Cáliz

29 de noviembre de 2017

1. Inversa generalizada

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

Definición 1.1. Sea F una función no decreciente en \mathbb{R} , la **inversa generalizada** de F , F^- se define como la función dada por

$$F^-(u) = \inf\{x : F(x) \geq u\}$$

En el caso en que F es una función de distribución para una variable aleatoria continua, entonces se tiene que F_X^- continua y no decreciente. Por el teorema del calor medio $F_X^-(u)$ es inyectiva y por lo tanto el ínfimo es único.

Teorema 1. Si $U \sim \mathcal{U}_{[0,1]}$, entonces la variable aleatoria $F^-(U)$ tiene distribución F . Ver 1

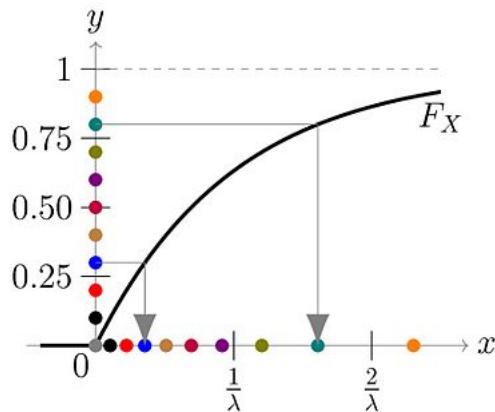


Figura 1: Se generan números aleatorios y_i generados con distribución uniforme entre 0 y 1. Los cuales se muestran como puntos de colores en el eje y. Cada punto es mapeado de acuerdo con $x = F^-(y)$, lo cual es mostrado con las dos flechas puestas en la ilustración

Demostración. Para toda $u \in [0, 1]$ y para toda $x \in F^-[0, 1]$, la inversa generalizada satisface

$$F(F^-(u)) \geq u \text{ y } F^-(F(x)) \leq x$$

Por lo tanto

$$\{(u, x) : F^-(u) \leq x\} = \{(u, x) : F(x) \geq u\} \text{ y } \mathbb{P}(F^-(U) \leq x) = \mathbb{P}(U \leq F(x)) = F(x)$$

□

2. Variables aleatorias uniformes

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

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

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

Se implementó el algoritmo recién mencionado el cual puede ser encontrado en el código adjunto de Python en la función `uniformeMCL`, MCL hace referencia a "Método de congruencias lineales". Se tomó como semilla los dígitos de en medio del valor del tiempo al cuadrado en el que se ejecuta el algoritmo.

En 2 se muestran los histogramas normalizados de muestras de tamaño 10,000 de números pseudo-aleatorios generados con el algoritmo propuesto y se hizo uso de la librería de `numpy` para comparar. En cada uno se graficó la función de densidad de la distribución uniforme.

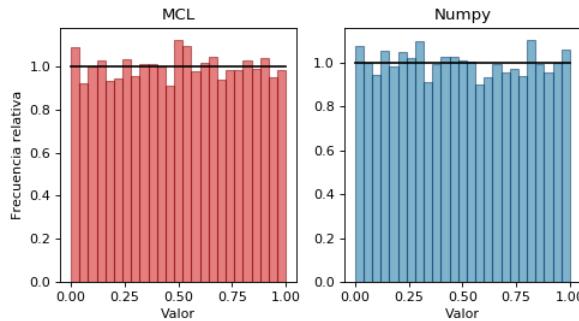


Figura 2: Histogramas normalizados y funciones de densidad

En la figura 2 se puede notar que en ambos casos el histograma normalizado se asemeja a la función de densidad apropiada.

Se aplicó el test de Kolmogorov-Smirnov de la librería `scipy.stats` el cual es llamado en el código con `kstest`. Los resultados se encuentran en 1, junto los tiempos de ejecución de cada algoritmo.

Podemos ver que algoritmo implementado no ofrece tan buenos resultados como los obtenidos por Numpy pero son *suficientemente buenos*, en base a la regla "de un p-valor *suficientemente bueno*, se puede decir que no existe evidencia muestral suficiente para rechazar la hipótesis de que los datos generados provienen de una distribución uniforme.

Sampler	estadístico (TKS)	p-valor (TKS)	tiempo de ejecución
Conguencias lineales	0.00653	0.48634	0.04548
Numpy	0.00968	0.30472	0.00065

Tabla 1: Tabla comparativa de resultados para muestras uniformes de tamaño 10,000 con distintos generadores

Para verificar la no dependencia temporal se graficó las parejas de punto $(X_t, Xt + 1)$, para muestras de tamaño 1000, los resultados fueron favorables y pueden encontrarse en 3

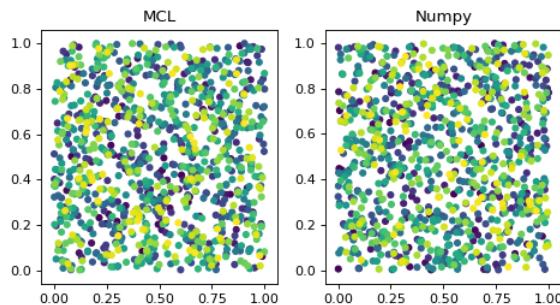


Figura 3: Gráficas de parejas (X_t, X_{t+1}) para muestras de tamaño 1000 generadas con MCL y con Numpy

3. Números aleatorios en `scipy.stats`

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

La librería de `scipy.stats` no tiene un generador de pseudonúmeros aleatorios, estos son obtenidos en una de las siguientes 3 maneras:

1. Directamente de `numpy.random`. El módulo `uniform` de la librería `scipy.stats` utiliza `Numpy`, `mtrand` es la función correspondiente en `stats` (`mtrand` es un alias para `numpy.random`), la diferencia reside en que `scipy.stats` es un poco más costoso al hacer verificaciones de errores y al hacer la interfaz más flexible. La diferencia en velocidad debería de ser mínima mientras `uniform.rvs` no se ejecute en un ciclo para cada muestra. Es mejor obtener todas las muestras de una sola vez.

En `scipy/numpy` se usa el algoritmo **Mersenne-Twister** PRNG (pseudorandom number generator) para generar los números aleatorios básicos. Los números aleatorios para distribuciones en `numpy.random` están en `cython/pyrex` y son bastante rápidos.

2. Trasformación de otros números aleatorios en `numpy.random`, también bastante rápido debido a que trabaja con arreglos de números enteros.

3. Genéricos: el único generador de números aleatorios es usando la inversa de cdf para transformar variables de números uniformes. Esto es relativamente rápido, si es que hay una expresión para la ppf, pero puede ser muy lento si la ppf se debe calcular indirectamente. por ejemplo si la única pdf está definida, entonces la cdf es obtenida a través de integración numérica y la ppf es obtenida a través de un solucionador de ecuaciones. Por lo que algunas distribuciones son muy lentas.

La **semilla** para inicializar el PRNG puede ser un entero entre 0 y $2^{32} - 1$ ó incluso un arreglo de dichos enteros o ninguno(default). En el caso por default `RandomState` tratará de leer datos de `/dev/urandom` (o el análogo en Windows) si está disponible, de no ser posible proveerá una semilla proveniente del reloj.

4. V.a. discretas en `scipy`

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

Sí, esto se hace por medio de la función "`scipy.stats.rv_discrete`". La cual es una clase base para construir clases específicas de distribuciones e instancias para variables aleatorias. También puede ser usada para construir distribuciones arbitrarias definidas por una lista de los puntos en el soporte y sus correspondientes probabilidades. Cabe mencionar que también se tiene la clase `rv_continuous`, cuyas principales diferencias son:

1. El soporte de la distribución en `rv_discrete` son los enteros
2. En lugar de una función de densidad pdf (y la correspondiente privada pdf), esta clase define la función de masa de probabilidad pmf (y la correspondiente privada pmf)
3. No está definido el parámetro de escala

Para ver un ejemplo de cómo crear una distribución discreta ver 4

```
>>> from scipy.stats import rv_discrete
>>> class poisson_gen(rv_discrete):
...     "Poisson distribution"
...     def _pmf(self, k, mu):
...         return exp(-mu) * mu**k / factorial(k)
```

para luego crear la instancia:

```
>>> poisson = poisson_gen(name="poisson")
```

Figura 4: Ejemplo de generación de v.a. poisson con `rv_discrete`

5. ARS

Implementar el algoritmo **Adaptive Rejection Sampling** y simular de una Gama(2, 1) 10,000 muestras. ¿cuándo es conveniente dejar de adaptar la envolvente?

Se implementó el siguiente algoritmo de ARS ver ¹

Algoritmo ARS

1. Inicializar n y S_n
2. Generar $X \sim g_n(X)$, $U \sim \mathcal{U}_{[0,1]}$
3. Si $U \leq \underline{f}_n(X)/\bar{\omega}_n \cdot g_n(X)$, aceptamos X ; de lo contrario,
si $U \leq f(X)/\bar{\omega}_n \cdot g_n(X)$, aceptamos X y actualizamos S_n a $S_{n+1} = S_n \cup \{X\}$

Algoritmo suplementario para ARS

1. Seleccionar intervalo $[x_i, x_{i+1}]$ con probabilidad

$$e^{\beta_i} \cdot \frac{e^{\alpha_i x_{i+1}} - e^{\alpha_i x_i}}{\bar{\omega}_n \alpha_i}$$

2. Generar $U \sim \mathcal{U}_{[0,1]}$ y tomar

$$X = \alpha_i^{-1} \cdot \log[e^{\alpha_i x_i} + U(e^{\alpha_i x_{i+1}} - e^{\alpha_i x_i})]$$

La implementación de la primer parte del algoritmo puede ser encontrada en el código de Python adjunto en la función **ARS**, la segunda parte se puede encontrar en la función **simulaG**.

Para comparar se usó que en este caso podemos obtener la v.a. simulando 2 variables aleatorias exponenciales, las cuales son muy fáciles de simular haciendo uso del Teorema 1 aquí enunciado y ya que en este caso la función inversa de $\exp(\lambda)$ es $(-1/\lambda) \cdot \log(1 - y)$.

En 5 se muestran los histogramas normalizados de muestras de tamaño 10,000 de números pseudo-aleatorios generados con la suma de la simulación de dos v.a. exponenciales, de la librería de **numpy** y del con el algoritmo ARS, para comparar los resultados. En cada uno se graficó la función de densidad de la distribución gama.

En la figura 2 se puede notar que todos los casos el histograma normalizado se asemeja a la función de densidad apropiada.

Se aplicó es test de Kolmogrov-Smirnov de la librería **scipy.stats**. Los resultados se encuentran en 2, junto los tiempos de ejecución de cada algoritmo.

¹Robert, Casella - Monte Carlo Statistical Methods Pag(56 y 71)

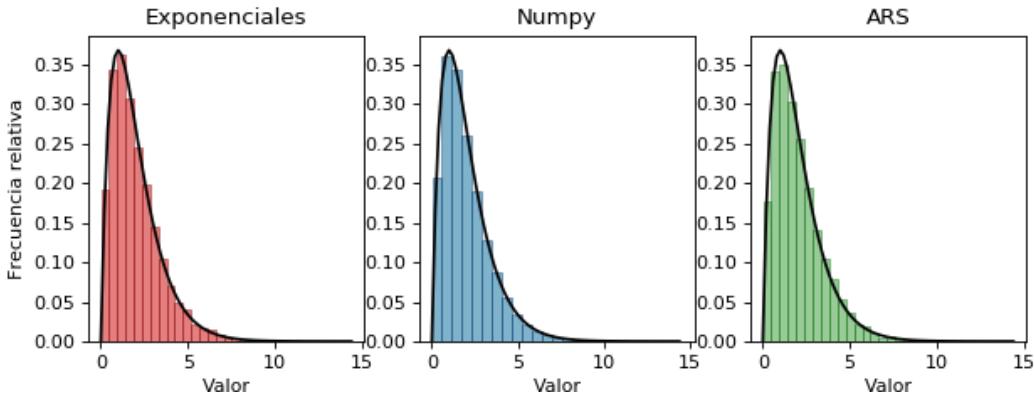


Figura 5: Histogramas normalizados y funciones de densidad

Sampler	estadístico (TKS)	p-valor (TKS)	tiempo de ejecución
Exponenciales	0.00636	0.81287	0.00529
Numpy	0.00793	0.55465	0.00634
ARS	0.007832	0.57155	3.73731

Tabla 2: Tabla comparativa de resultados para muestras $Gama(2, 1)$ de tamaño 10,000 con distintos generadores

Podemos ver que algoritmo implementado ofrece muy buenos resultados como los obtenidos por Numpy o los de la suma de exponentiales ya que el p-valor es *suficientemente pequeño*; en cada caso se puede decir que no existe evidencia muestral suficiente para rechazar la hipótesis de que los datos generados provienen de una distribución Gama. Como es de suponerse el tiempo de ejecución del algoritmo ARS es mucho mayor pero considerando que puede ser aplicado cuando los parámetros no son tan fáciles como en este caso podemos decir que es un algoritmo de gran utilidad. Para comparar los tiempos de ejecución en un caso no entero se midió el tiempo que tardaba ARS y Numpy para generar muestras de tamaños entre 0 y 10,000 (sólo múltiplos de 500) y se graficó su resultado en escala logartímica ver 6. La diferencia parece indicar una diferencia por una constante entre los tiempos de ejecución, lo cual es bueno.

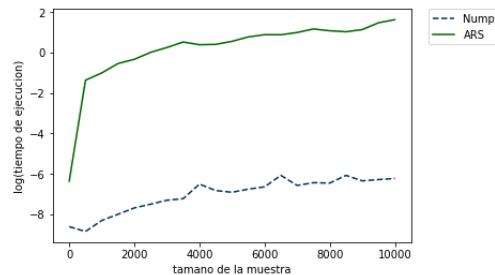


Figura 6: Tiempos de ejecución de Numpy y ARS en escala logarítmica

Tarea 6. MCMC: Metropolis-Hastings

Ricardo Chávez Cáliz

October 18, 2017

Problema 1. Simular $n = 5$ y $n = 30$ v.a Bernoulli $Be(1/3)$; sea r el número de éxitos en cada caso.

Para simular X variable aleatoria Bernoulli se tomó un elemento en una muestra simulada de manera uniforme en el intervalo $[0, 1]$, si dicho elemento era menor o igual que $1/3$ entonces se consideraba éxito i.e. $X = 1$ y de lo contrario se tomaba $X = 0$.

Para verificar este algoritmo de simulación se generaron muestras desde 1 hasta 5000 elementos y se calculó el promedio para cada uno. Obteniendo los siguientes resultados.

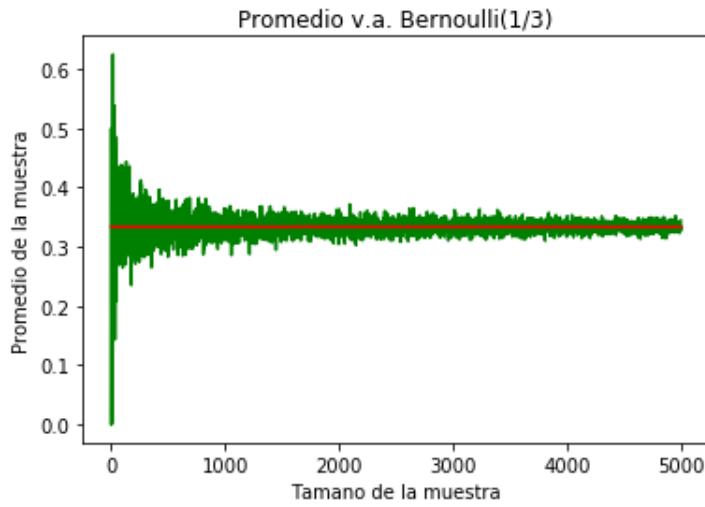


Figure 1: Se muestra el valor estimado de p para distintos tamaños de muestra

El promedio de las muestras es cada vez menos sesgado a $1/3$ a medida que el tamaño de la muestra aumenta.

Para los resultados consecutivos aquí mostrados se obtuvieron $r = 4$ cuando $n = 5$ y $r = 11$ cuando $n = 30$.

Problema 2. Implementar el algoritmo Metropolis-Hastings para simular de la posterior

$$f(p|\bar{x}) \propto p^r (1-p)^{n-r} \cos(\pi p) I_{[0, \frac{1}{2}]}(p),$$

con los dos casos de n y r de arriba. Para ello poner la propuesta $q(p'|p) = q(p') \sim Beta(r + 1, n - r + 1)$ y la distribución inicial de la cadena $\mu \sim U(0, \frac{1}{2})$.

Para el algoritmo MCMH visto en clase es necesario calcular $\rho(p, p')$, para la densidad instrumental propuesta se obtuvo.

$$\rho(p, p') = \min \left\{ 1, \frac{\mathbb{1}_{[0,1/2]}(p') \cdot \cos(\pi p')}{\mathbb{1}_{[0,1/2]}(p) \cdot \cos(\pi p)} \right\}$$

Note que como $\text{supp}(\mu) = \text{supp}(f) = [0, 1/2]$ entonces $p \in [0, 1/2]$ en el tiempo inicial y por la indicadora en el numerador $p \in [0, 1/2]$ para todo tiempo, por lo tanto $\rho(p, p')$ está bien definida para todo tiempo.

A continuación se presentan resultados de muestras de tamaño 10000 de las simulaciones para $n = 5, r = 4$ y $n = 30, r = 11$ de f , es decir para $f(p) \propto p^4(1-p)^1 \cos(\pi p)$ y $f(p) \propto p^{11}(1-p)^{19} \cos(\pi p), p \in [0, 1/2]$

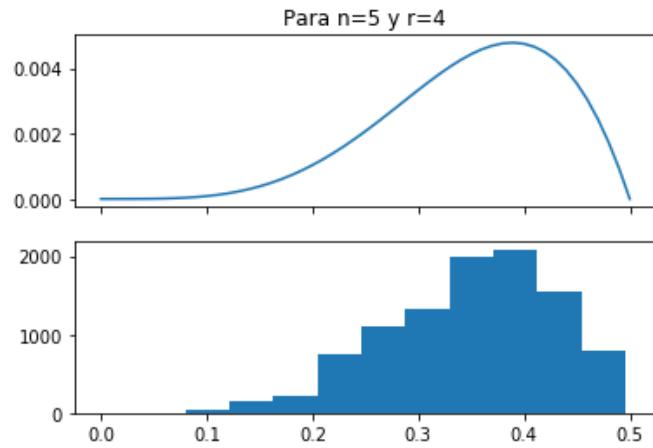


Figure 2: Se muestra la gráfica de f a simular (sin normalizar) con $n = 5$ y $r = 4$ y el histograma de la simulación

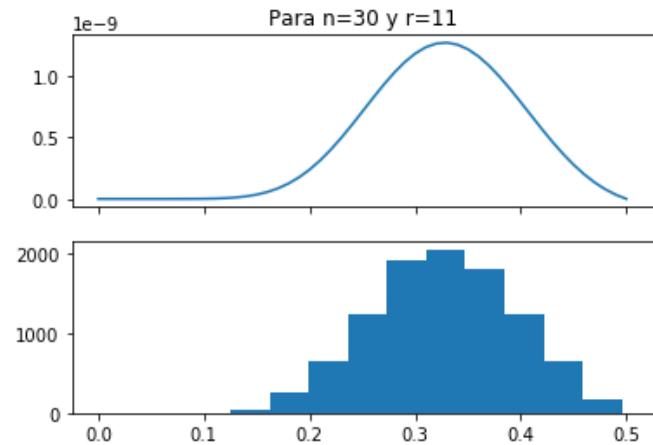


Figure 3: Se muestra la gráfica de f a simular (sin normalizar) con $n = 30$ y $r = 11$ y el histograma de la simulación

En ambas figuras se puede apreciar que el histograma sigue la forma de la función deseada. Para obtener la constante de normalización basta calcular la integral de f

Problema 3. Argumentar porque la cadena es f -irreducible y porque es ergódica. Implementar el algoritmo con los datos descritos y discutir los resultados.

Sea A un conjunto de medida positiva, de las ecuaciones de Chapman-Kolmogorov tenemos que

$$K^n(x, A) = \int K^{n-1}(x, dy) K(dy, A)$$

para ver existe un n en el que el Kernel a n pasos es positivo recordemos que el Kernel de MH está dado por

$$K(p, p') = q(p, p') \cdot \rho(p, p') + (1 - r(p)) \cdot \delta_p(p')$$

$$\text{donde } r(p) = \int q(p'|p) \cdot \rho(p, p') dp'$$

Como la densidad instrumental se tomó independiente a los datos, tenemos que $q(p'|p) = q(p') \sim Beta(r + 1, n - r + 1)$, la cual sólo se anula fuera del soporte de la función y además $supp(f) \subset supp(q)$, por lo tanto $q(p'|p) > 0$ y por lo tanto $K(p, p') > 0$ y así $K^n(x, A) > 0$ por lo tanto la cadena es f -irreducible.

Teorema 1. Si la cadena de Markov asociada a un algoritmo de Metropolis Hasting es f -irreducible entonces es Harris recurrente

Por el teorema anterior tenemos que la cadena es Harris recurrente, además sabemos que para probar que la cadena es ergódica basta probar que la cadena es fuertemente aperiódica. La aperiodicidad fuerte se deduce de que $K(p, \{p\}) > 0 \forall p \in X$ pues si $\rho(p, p') < 1$ entonces la probabilidad de rechazar el valor propuesto p' es positivo, y en este caso el algoritmo MH establece que $X_{t+1} = X_t$ (la probabilidad de que te quedarase en un estado de la cadena es positiva por cómo está dada ρ y por la construcción de MCMH).

La irreducibilidad es importante porque nos permite simular de cualquier parte del dominio de la función deseada, en las simulaciones esto se ve reflejado en la ausencia de huecos. La ergodicidad garantiza la convergencia de la cadena, lo cual también puede apreciarse al ver como los histogramas siguen la misma forma de la función buscada.

Problema 4. Implementar el algoritmo Metropolis-Hastings con la posterior de arriba tomando una propuesta diferente.

Se tomó $q(p'|p) \sim U(0, 1/2)$, como la densidad instrumental se tomó independiente entonces los resultados del ejercicio anterior se satisfacen con esta propuesta de igual manera. En este caso

$$\rho = \min \left\{ 1, \frac{\mathbb{1}_{[0,1/2]}(p') \cdot (p')^r (1-p')^{n-r} \cos(\pi p')}{\mathbb{1}_{[0,1/2]}(p) \cdot p^r (1-p)^{n-r} \cos(\pi p)} \right\}$$

De igual manera a ejercicio como $\text{supp}(\mu) = \text{supp}(f) = [0, 1/2]$ entonces $p \in [0, 1/2]$ para todo tiempo, y por lo tanto $\rho(p, p')$ está bien definida para todo tiempo.

A continuación se presentan resultados de muestras de tamaño 10000 de las simulaciones para $n = 5, r = 4$ y $n = 30, r = 11$ de f , es decir para $f(p) \propto p^4(1-p)^1 \cos(\pi p)$ y $f(p) \propto p^{11}(1-p)^{19} \cos(\pi p), p \in [0, 1/2]$

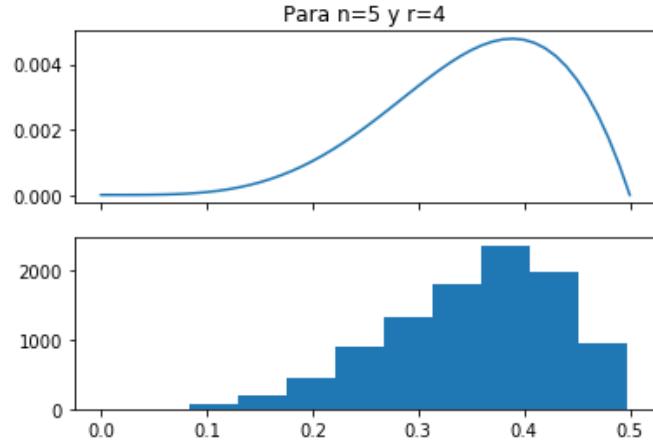


Figure 4: Se muestra la gráfica de f a simular (sin normalizar) con $n = 5$ y $r = 4$ y el histograma de la simulación

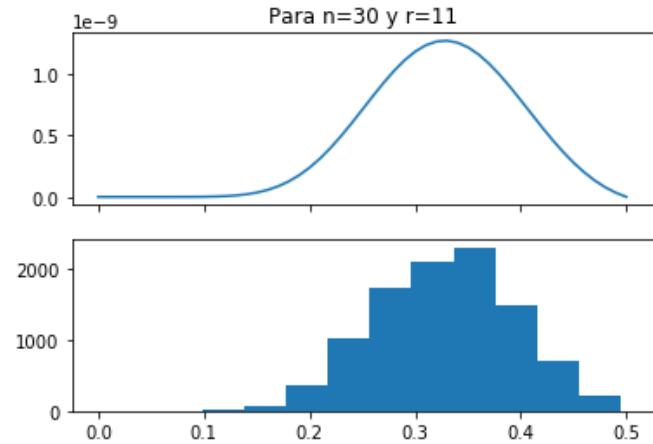


Figure 5: Se muestra la gráfica de f a simular (sin normalizar) con $n = 30$ y $r = 11$ y el histograma de la simulación

Nuevamente en ambas figuras se puede apreciar que el histograma sigue la forma de la función deseada. De nuevo para obtener la constante de normalización basta calcular la integral de f

Tarea 7. MCMC: Metropolis-Hastings II

Ricardo Chávez Cáliz

October 25, 2017

Con el algoritmo Metropolis-Hastings (MH), simular lo siguiente:

Problema 1. Sean $x_i \sim Ga(\alpha, \beta); i = 1, 2, \dots, n$. Simular datos x_i con $\alpha = 3$ y $\beta = 100$ considerando los casos $n = 5$ y 30 .

Con $\alpha \sim U(1,4)$, $\beta \sim \exp(1)$ distribuciones a priori, se tiene la posterior

$$f(\alpha, \beta | \bar{x}) \propto \frac{\beta^{n\alpha}}{\Gamma(\alpha)^n} r_1^{\alpha-1} e^{-\beta(r_2+1)} 1(1 \leq \alpha \leq 4) 1(\beta > 0),$$

con $r_2 = \sum_{i=1}^n x_i$ y $r_1 = \prod_{i=1}^n x_i$.

En ambos casos, grafica los contornos para visualizar dónde está concentrada la posterior.

Utilizar la propuesta

$$q\left(\begin{pmatrix} \alpha_p \\ \beta_p \end{pmatrix} \mid \begin{pmatrix} \alpha \\ \beta \end{pmatrix}\right) = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \end{pmatrix},$$

donde

$$\begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \sim \mathcal{N}_2\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}\right).$$

Se模拟aron muestras de tamaño 5 y 30, con distribución $Gamma(\alpha, \beta)$ con $\alpha = 3$ y $\beta = 100$, como los algoritmos de *Python* consideran el parámetro de escala θ para usar el parámetro solicitado se consideró $\theta = 1/\beta$. Se obtuvieron $r1$ y $r2$ del orden de e^{-10} y e^{-2} respectivamente, de la expresión de $f(\alpha, \beta | \bar{x})$ podemos adelantar que los cálculos realizados directamente con estos valores serán inadecuados. Para solucionar este problema se hicieron simplificaciones logarítmicas en la expresión de $f(\alpha, \beta | \bar{x})$ donde

$$\log(f(\alpha, \beta | \bar{x})) = n\alpha \cdot \log(\beta) + (\alpha - 1) \cdot \log(r1) - \beta \cdot (r2 + 1) - n \cdot \log(\Gamma(\alpha))$$

Tomaremos el dominio de graficación con $\alpha \in [1, 4]$ y $\beta \in [80, 120]$. La elección para α es evidente de la definición de f , y para β tomamos este rango dado que las simulaciones para la coordenada en β se puede obtener un estimador de β usando las medias muestrales. En la figura 1 se muestran los contornos para f y $\log(f)$, se puede notar que f está concentrada en un rango muy pequeño alrededor del 80, esto es algo que se toma en cuenta para lo consecutivo.

Se implementó el algoritmo de MCMH haciendo las correspondientes simplificaciones logarítmicas para f en el calculo de ρ

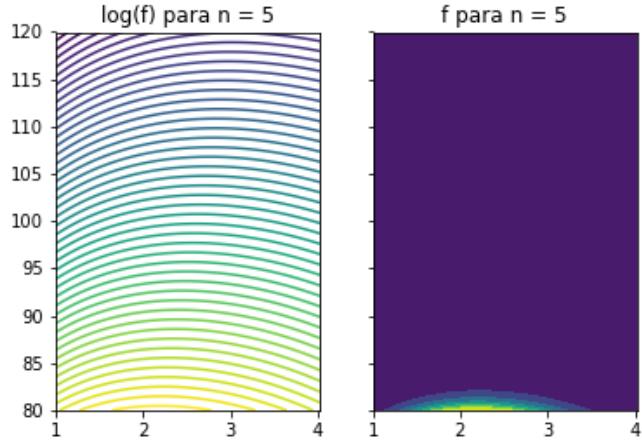


Figure 1: Se muestran las superficies de nivel de $\log(f)$ y f para $n = 5$

Tenemos:

$$\begin{aligned}\rho &= \min \left\{ 1, \frac{\beta_p^{n \cdot \alpha_p} \cdot r_1^{\alpha_p - 1} \cdot e^{-\beta_p(r_2+1)} \cdot \Gamma(\alpha)^n \cdot \mathbb{1}_{[1,4]}(\alpha_p) \cdot \mathbb{1}_{[0,\infty)}(\beta_p)}{\beta^{n \cdot \alpha} \cdot r_1^{\alpha-1} \cdot e^{-\beta(r_2+1)} \cdot \Gamma(\alpha_p)^n \cdot \mathbb{1}_{[1,4]}(\alpha) \cdot \mathbb{1}_{[0,\infty)}(\beta)} \right\} \\ \rho &= \min \left\{ 1, \frac{\beta_p^{n \cdot \alpha_p} \cdot r_1^{\alpha_p - \alpha} \cdot e^{-\beta - \beta_p} \cdot \Gamma(\alpha)^n \cdot \mathbb{1}_{[1,4]}(\alpha_p) \cdot \mathbb{1}_{[0,\infty)}(\beta_p)}{\beta^{n \cdot \alpha} \cdot \Gamma(\alpha_p)^n \cdot \mathbb{1}_{[1,4]}(\alpha) \cdot \mathbb{1}_{[0,\infty)}(\beta)} \right\}\end{aligned}$$

Tomando logaritmo y dejando un momento de lado las funciones indicadoras obtenemos

$$\rho = e^{\min\{0,w\}}$$

donde $w = n\alpha_p \cdot \log(\beta_p) + (\alpha_p - \alpha) \cdot \log(r_1) + (\beta - \beta_p) + n \cdot \log(\Gamma(\alpha)) - n\alpha \cdot \log(\beta) - n \cdot \log(\Gamma(\alpha_p))$

Note que q está definida de tal manera de que el movimiento de la cadena de Markov sea independiente en cada coordenadas. De esta manera, estando en el tiempo i pasamos en el tiempo $i+1$ moviéndonos horizontalmente con una normal de varianza σ_1 centrada en el punto actual y de igual manera el movimiento vertical con una normal de varianza σ_2 . Por lo anterior se tomaron $\sigma_1 = 0.2$ y $\sigma_2 = 1$, ocupamos proponer saltos pequeños por lo estrecho de la región, permitiéndonos mayor libertad en el movimiento vertical.

Sin otra particularidad, se implementó el algoritmo MCMH para la función objetivo dada, con las funciones ρ y q descritas. Para verificar dicha simulación se generó una muestra de tamaño 10000 y se graficaron los valores en el plano en un gradiente de color para poder analizar el comportamiento temporal de la cadena, los primeros elementos son mostrados de color azul y los últimos de color amarillo.

Se llevó registro del número de puntos rechazados, en este caso se rechazaron el 0.6531% de las propuestas.

En la figura 2 se observa un sesgo hacia la izquierda y que la cadena no parece coincidir con la función de densidad objetivo (compare con figura 1). Lo anterior no es sorprendente dado que la información a priori era muy pequeña para tener un buen estimador ($n=5$).

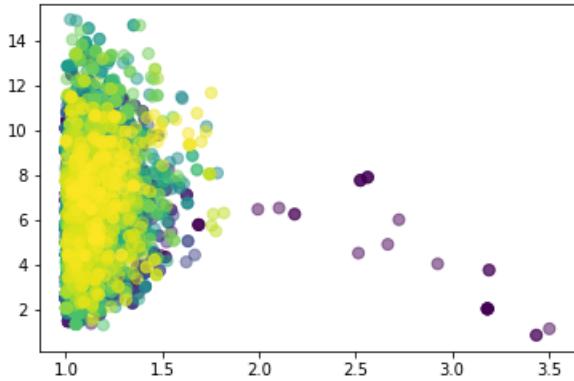


Figure 2: Se muestran la simulación de tamaño 10000, para $n = 5$, $\sigma_1 = 0.2$ y $\sigma_2 = 1$

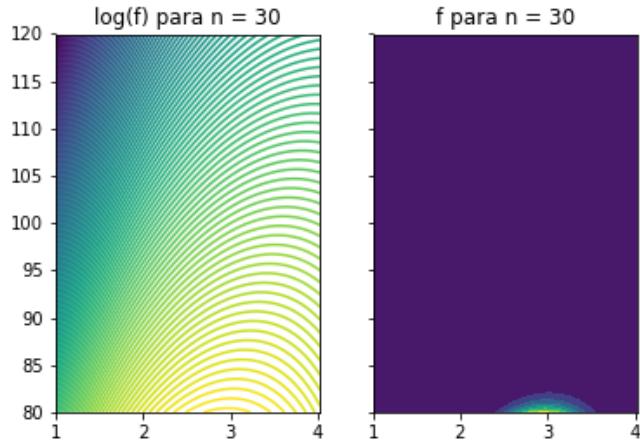


Figure 3: Se muestran las superficies de nivel de $\log(f)$ y f con $n=30$

Análogamente se llevó a cabo la simulación en el caso $n=30$, obteniendo los siguientes resultados.

Igualmente se registró del número de puntos rechazados, en este caso se rechazaron el 28.57% de las propuestas.

En la figura 4, se muestra la simulación de la cadena para $n = 30$ y a diferencia del caso anterior la cadena en este caso sí parece coincidir con la función de densidad objetivo (compare con figura 3). Note que la cadena empieza empieza con puntos de ordenada pequeña (así lo estipulamos en la distribución a priori con $\beta \sim \text{Exp}(1)$) y termina acumulándose en una franja a la altura de 80 como esperábamos y de ancho como el que se muestra en la figura 3. Como es sabido, una muestra de tamaño 30 puede ser suficiente para tener "buenos" estimadores.

Los promedios de las muestras para α fue 2.61930478486 y para β 78.8281005967.

Para encontrar el *burn-in* de la cadena se comparó $\log(X_t)$ vs t en el caso $n = 30$ separando por coordenadas. Se obtuvieron los siguientes resultados.

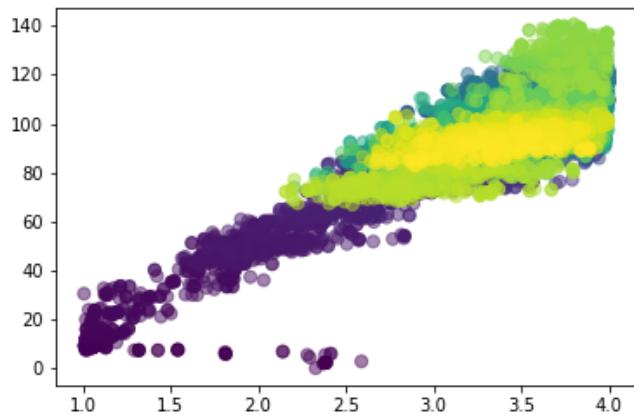


Figure 4: Se muestran la simulación de tamaño 10000, para $n = 30$, $\sigma_1 = 0.2$ y $\sigma_2 = 1$

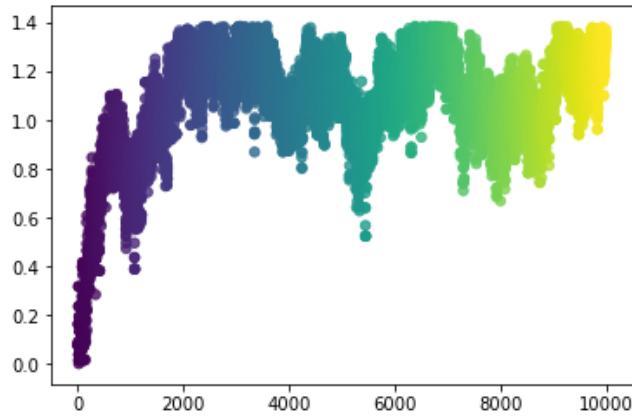


Figure 5: Tiempo de calentamiento para α con $n = 30$ en una muestra de tamaño 10000

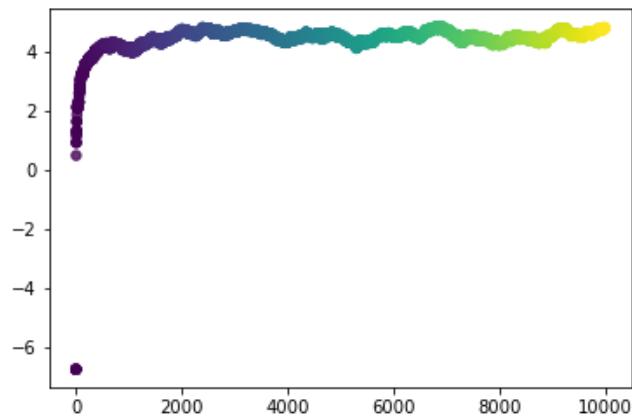


Figure 6: Tiempo de calentamiento para β con $n = 30$ en una muestra de tamaño 10000

Problema 2. Simular de la distribución $\text{Gamma}(\alpha, 1)$ con la propuesta $\text{Gamma}([\alpha], 1)$, donde $[\alpha]$ denota la parte entera de α .

Además, realizar el siguiente experimento: poner como punto inicial $x_0 = 1, 000$ y graficar la evolución de la cadena, es decir,
 $f(X_t)$ vs t .

Se implementó el algoritmo MCMH, sin alguna particularidad con

$$\rho(x, y) = \min \left\{ 1, \frac{y^{\alpha - [\alpha]}}{x^{\alpha - [\alpha]}} \right\}$$

Se obtuvieron los siguientes resultados al variar el punto inicial, para $x_0 = 1, 10, 100, 1000$. En cada caso se rechazaron alrededor el 15% de las propuestas

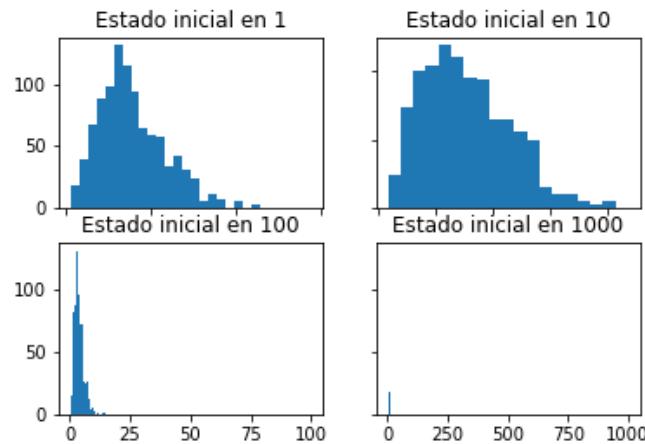


Figure 7: Histogramas de una muestra de tamaño 1000, variando el punto de inicio de la cadena

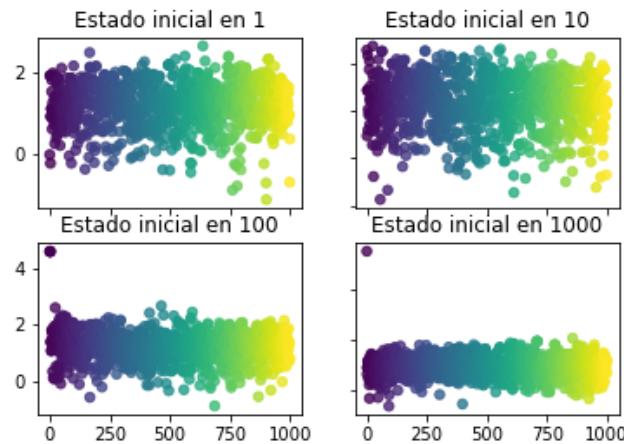


Figure 8: Tiempo de calentamiento en una muestra de tamaño 1000 variando el punto inicial

En la figura 7 se puede apreciar para los casos 1, 10 y 100 la forma de la gamma buscada, con una observación extraña en el valor de x_0 correspondiente, la figura 8 permite discriminar el tiempo en que dicha observación extraña no aparece

Problema 3. Implementar Random Walk Metropolis Hasting (RWMH) donde la distribución objetivo es $\mathcal{N}_2(\mu, \Sigma)$, con

$$\mu = \begin{pmatrix} 3 \\ 5 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix}.$$

Utilizar como propuesta $\varepsilon_t \sim \mathcal{N}_2(\mathbf{0}, \sigma I)$. ¿Cómo elegir σ para que la cadena sea eficiente? >Qué consecuencias tiene la elección de σ ?

Como experimento, elige como punto inicial $x_o = \begin{pmatrix} 1000 \\ 1 \end{pmatrix}$ y comenta los resultados.

Para todos los incisos del ejercicio anterior:

- Establece cual es tu distribución inicial.
- Grafica la evolución de la cadena.
- Indica cuál es el Burn-in.
- Comenta qué tan eficiente es la cadena.
- Implementa el algoritmo MH considerando una propuesta diferente.

Para implementar el algoritmo se tomó $y_t = x_t + \epsilon_t$ con una densidad instrumental $q(y|x) = g(\|y - x\|)$ donde g es la densidad de $\|\epsilon_t\|$, como q es simétrica entonces

$$\rho(x, y) = \min \left\{ 1, \frac{f(y)}{f(x)} \right\}$$

nuevamente para obtener cálculos adecuados de ρ se hicieron simplificaciones logarítmicas (usando monotonía de la función logaritmo) .

$$\rho(x, y) = e^c$$

donde $c = \min \{0, \log(f(y)) - \log(f(x))\}$ con $\log(f) = (-1/(2 * 0.36)) * ((y1 - 3.0) ** 2 + (y2 - 5.) ** 2 - 2. * 0.8 * (y1 - 3.) * (y2 - 5.))$

Simulamos el caso óptimo con punto inicial en la media $(3, 5)$, tomando una muestra de tamaño 1000 con $\sigma = 1$ se obtuvieron los siguientes resultados los cuales se grafican junto con los contornos de nivel de la función de densidad bivariada correspondiente. Como antes los colores hacen referencia al orden que lleva la cadena siendo azul los primeros elementos y amarillos los últimos.

Para tener una mejor noción de la caminata aleatoria realizada se graficó una línea gradiente entre cada estado de la cadena.

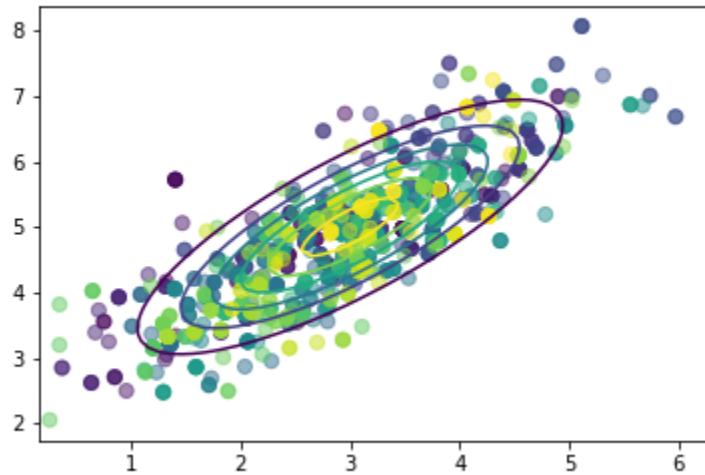


Figure 9: Muestra generada con el algoritmo RWMH iniciando en la media de la distribución objetivo y comparada con los conjuntos de nivel de la función de densidad correspondiente

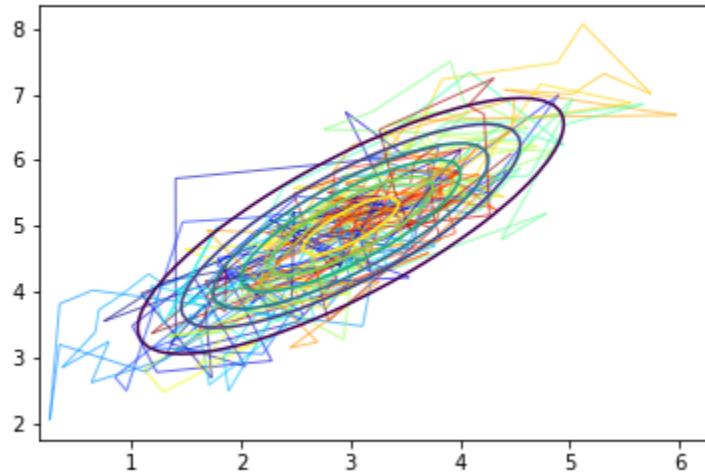


Figure 10: Caminata aleatoria generada con el algoritmo RWMH iniciando en media y comparada con los conjuntos de nivel de la función de densidad correspondiente

Se realizó el experimento realizando la caminata aleatoria desde un punto lejano a la media como $(1000, 1)$, para logar la convergencia con un tamaño de muestra como el que se tomó, se tomo $\sigma = 20$. La importancia en la elección del sigma reside en que cuando empezamos en un punto lejano a la función objetivo, podemos llegar a esta más pronto, sin embargo estando cerca de la densidad la simulación será imprecisa.

Se obtuvieron los siguientes resultados

Por las dimensiones del ejemplo, no se aprecia en esta gráfica que la cadena se acumula

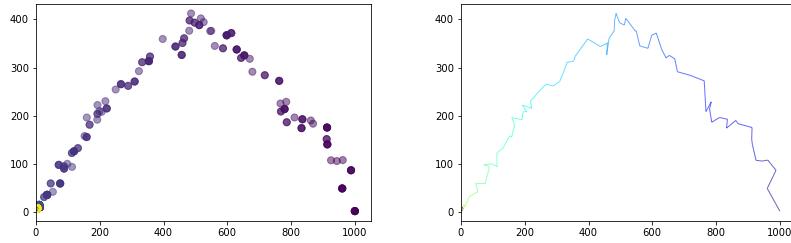


Figure 11: Muestra generada con el algoritmo RWMH iniciando en $(1000,1)$, $\sigma = 20$ y de tamaño 1000

en donde debe, por ello se grafica cuando el punto de inicio es $(100,1)$, que tendrá el mismo efecto demostrativo.

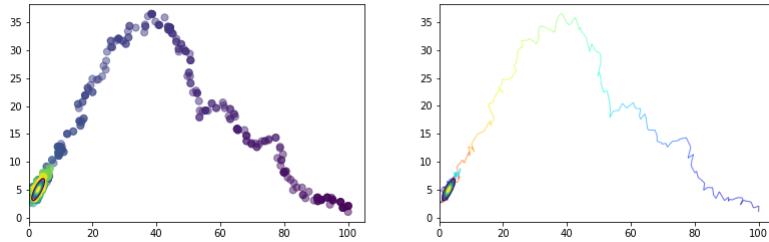


Figure 12: Muestra generada con el algoritmo RWMH iniciando en $(100,1)$, $\sigma = 2$ y de tamaño 1000

La peculiaridad de la curva seguida por la caminata aleatoria no es una coincidencia, ya que el eigenvector asociado al eigenvalor mayor de la matriz de covarianza apunta en la dirección de la caminata seguida y corresponde a una línea de flujo que proviene de la geometría de la superficie dada por la función de densidad.

Para encontrar el *burn-in* de la cadena se comparó $\log(X_t)$ vs t en el caso $n = 30$ separando por coordenadas. Se obtuvieron los siguientes resultados.

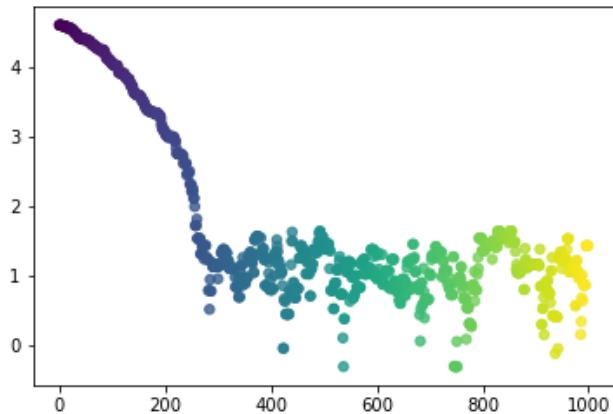


Figure 13: Tiempo de calentamiento para α con $n = 30$ en una muestra de tamaño 1000

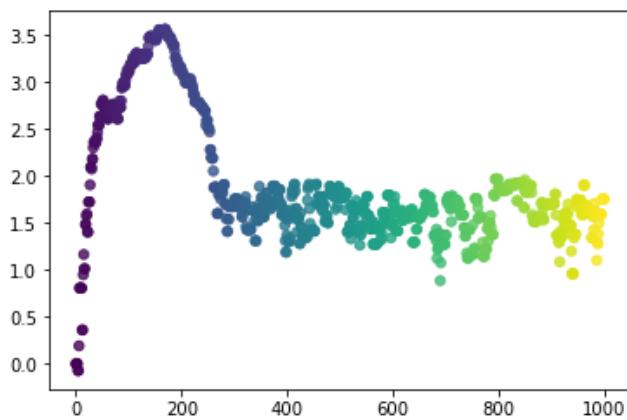


Figure 14: Tiempo de calentamiento para β con $n = 30$ en una muestra de tamaño 1000

Alrededor del tiempo 200, la cadena se estabiliza en ambas coordenadas, por lo que quitando los primeros 200 términos de la cadena obtendremos una simulación apropiada.

Tarea 8. MCMC: MH con Kerneles Híbridos y Gibbs Sampler

Ricardo Chávez Cáliz

November 1, 2017

Problema 1. Simule valores usando el algoritmo de Metropolis-Hastings para $f(x, y)$ la distribución normal bivariada con media μ y matriz de varianza Σ usando Kerneles híbridos.

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

Use las siguientes propuestas:

$$q_1 \left(\begin{pmatrix} x_p \\ y_p \end{pmatrix} \mid \begin{pmatrix} x \\ y \end{pmatrix} \right) = f_{X|Y}(x_p|y) \cdot \mathbb{1}_{(y_p=y)}$$

$$q_2 \left(\begin{pmatrix} x_p \\ y_p \end{pmatrix} \mid \begin{pmatrix} x \\ y \end{pmatrix} \right) = f_{Y|X}(y_p|x) \cdot \mathbb{1}_{(x_p=x)}$$

Considere los casos $\rho = 0.8$ y $\rho = 0.99$.

Se aplicó el algoritmo solicitado para la función especificada el cual puede ser encontrado en la función `NormalMHMC` del código adjunto. Se fijaron como puntos inciales los correspondientes a la media, ie $x_0 = \mu_1$ y $y_0 = \mu_2$.

Para verificar la simulación se graficaron las curvas de nivel de $f(x, y)$ y se sobrepusieron los valores de la muestra obtenida mediante el algoritmo MH con los kernels híbridos propuestos.

Sin perdida de generalidad se considera el vector 0 como μ . Eligiendo cada propuesta de manera uniforme, es decir $\omega_1 = \frac{1}{2}$, se obtuvieron los siguientes resultados con distintos tamaños de muestra $n = 1000$ y $n = 10000$.

Para $\rho = 0.8$, se puede observar en la figura 1 que a medida que se aumenta el número de la muestra la aproximación a $f(x, y)$ es mejor.

Para $\rho = 0.99$ (ver figura 2), a diferencia del caso anterior, la muestra no se aproxima a $f(x, y)$ a pesar de tener una gran cantidad de elementos. Esto tiene sentido dado que las propuestas que se están tomando son las marginales en dirección horizontal y vertical, por lo que la manera en que se mueve la cadena a cada paso depende de ρ y a medida que ρ tiende a 1, los pasos son más pequeños, sobre todo acercándose a las colas de $f(x, y)$ (ver las curvas de nivel de $f(x, y)$). Por lo tanto empezando en la media, será necesario dar una gran cantidad de pasos (una muestra muy grande) para poder obtener información de las colas de la densidad.

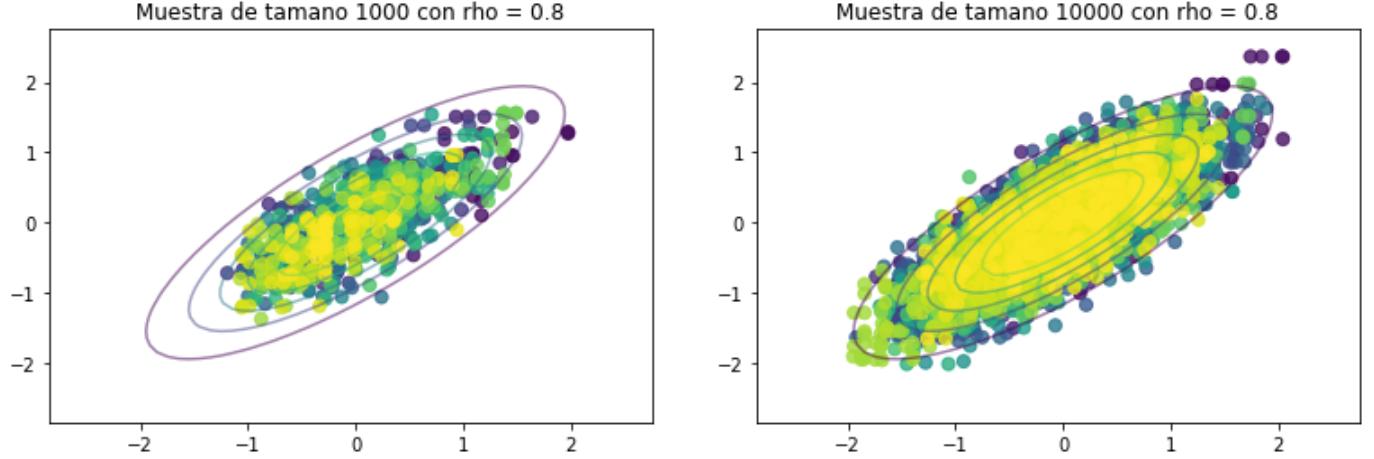


Figure 1: Se muestra las curvas de nivel de $f(x, y)$ y una muestra de tamaño 1000 con $\rho = 0.8$

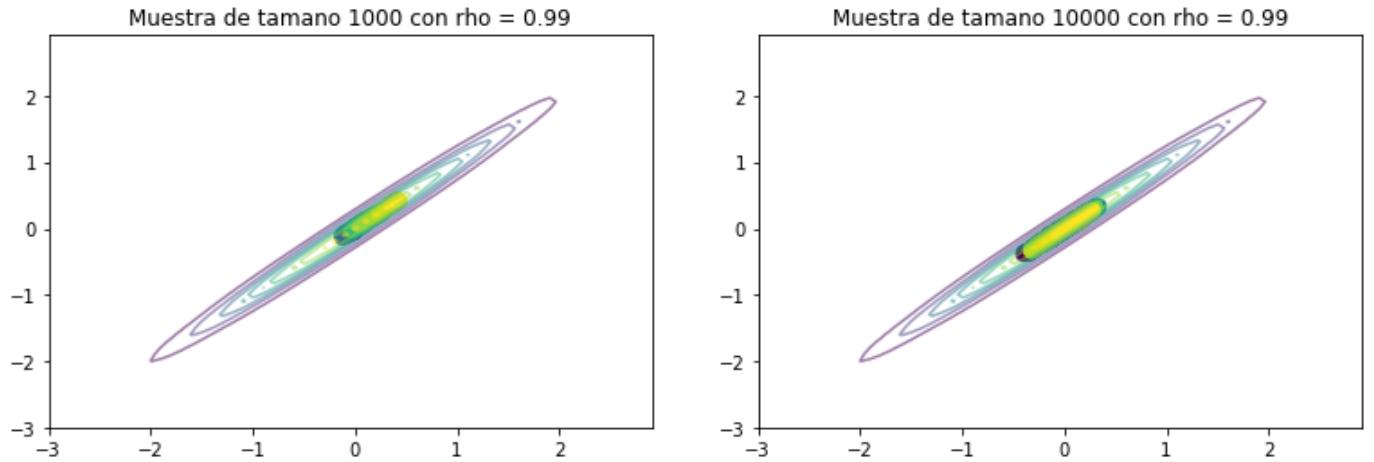


Figure 2: Se muestra las curvas de nivel de $f(x, y)$ y una muestra de tamaño 1000 con $\rho = 0.99$

Problema 2. Considere t_1, \dots, t_n tiempos de falla, con distribución $Weibull(\alpha, \lambda)$, es decir $f(t_i|\alpha, \lambda) = \alpha \lambda t_i^{\alpha-1} e^{-t_i^\alpha \lambda}$ donde α y λ se asumen con densidades apriori: $\alpha \sim \exp(c)$ y $\lambda|\alpha \sim Gama(\alpha, b)$. Así se obtiene la distribución posterior

$$f(\alpha, \lambda|\bar{t}) \propto f(\bar{t}|\alpha, \lambda)f(\alpha, \lambda)$$

A partir del algoritmo MH usando Kernes híbridos simule valores de la distribución posterior $f(\alpha, \lambda|\bar{t})$, considerando las siguientes propuestas:

Propuesta 1: $\lambda_p|\alpha, \bar{t} \sim Gama\left(\alpha + n, b + \sum_{i=1}^n t_i^\alpha\right)$ y dejando α fijo.

Propuesta 2: $\alpha_p|\lambda, \bar{t} \sim Gama(n + 1, -\log(b) - \log(r_1) + c)$, con $r_1 = \prod_{i=1}^n t_i$ y dejando λ fijo.

Propuesta 3: $\alpha_p \sim \exp(c)$ y $\lambda_p | \alpha_p \sim Gama(\alpha_p, b)$.

Propuesta 4 (RWMH): $\alpha_p = \alpha + \epsilon$, con $\epsilon \sim N(0, \sigma)$ y dejando λ fijo.

Simular datos usando $\alpha = 1$ y $\lambda = 1$ con $n = 20$. Para la a priori usar $c = 1$ y $b = 1$.

Se aplicó el algoritmo solicitado para la función especificada el cual puede ser encontrado en la función `TiemposMHMC` del código adjunto.

Para verificar la simulación se graficaron las curvas de nivel de $\ln(f(\alpha, \lambda | \bar{t}))$ y se sobrepusieron los valores de la muestra obtenida mediante el algoritmo MH con los kernels híbridos propuestos. Ver figura 3.

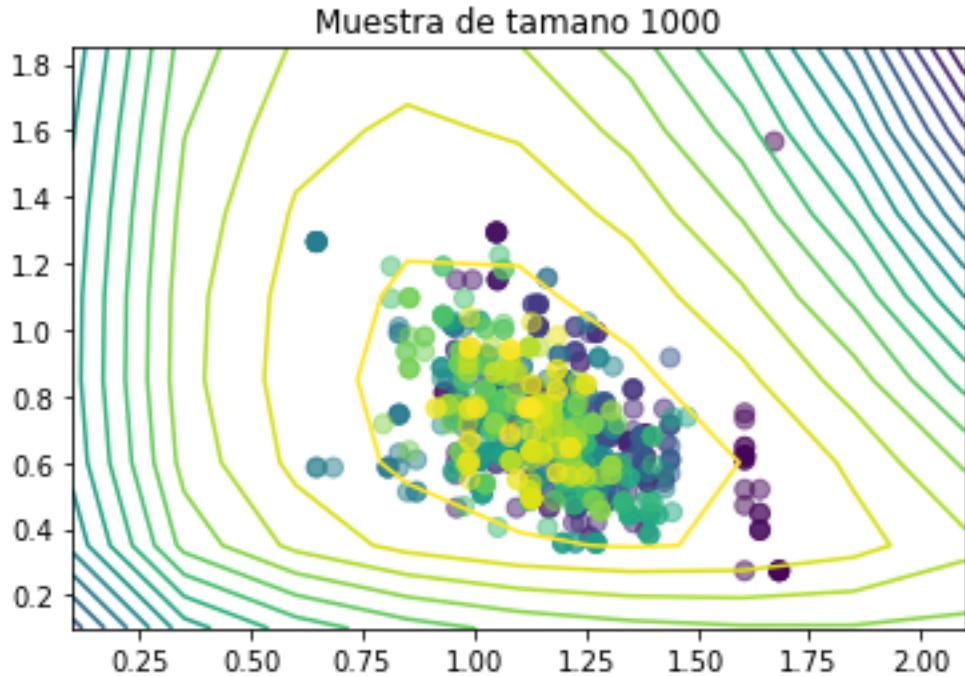


Figure 3: Se muestra las curvas de nivel de $\ln(f(\alpha, \lambda | \bar{t}))$ y una muestra de tamaño 1000

Para obtener los valores numéricos apropiados para ρ_2, ρ_3, ρ_4 se hicieron consideraciones de cálculos logarítmicos, para el primer kernel tenemos $\rho_1 = 1$, por lo tanto para esta propuesta no fue necesario. Se obtuvieron:

$$\rho_2 = e^{\min\{0, \ln(\Gamma(\alpha)) + \lambda(\alpha_p - \alpha) - \lambda \cdot (\sum_{i=1}^n t_i^{\alpha_p}) - \ln(\Gamma(\alpha_p)) + \lambda \cdot (\sum_{i=1}^n t_i^\alpha)\}}$$

$$\rho_3 = e^{\min\{0, n \cdot \ln(\frac{\alpha_p \cdot \lambda_p}{\alpha * l}) + (\alpha_p - \alpha) \cdot \ln(r_1) - \lambda_p \cdot (\sum_{i=1}^n t_i^{\alpha_p}) + \lambda \cdot (\sum_{i=1}^n t_i^\alpha)\}}$$

$$\rho_4 = e^{\min\{0, w\}}$$

donde $w = \alpha_p \cdot \ln(\lambda) - \lambda \cdot (\beta + \sum_{i=1}^n t_i^{\alpha_p}) + \alpha_p \cdot \ln(\beta) + n \cdot \ln(\alpha_p) + (\alpha_p - 1) \cdot (\sum_{i=1}^n \ln(t_i)) - \gamma \cdot \alpha_p - \ln(\Gamma(\alpha_p)) - \alpha \cdot \ln(\lambda) + \lambda \cdot (\beta + \sum_{i=1}^n t_i^\alpha) - \alpha \cdot \ln(\beta) - n \cdot \ln(\alpha) - (\alpha - 1) \cdot (\sum_{i=1}^n \ln(t_i)) + \gamma \cdot \alpha + \ln(\Gamma(\alpha))$

El algoritmo lleva registro de la cantidad de propuestas rechazadas, en la simulación aquí mostrada se rechazaron el 51.56% de las propuestas

Dependiendo de la muestra a veces es necesario calcular el logaritmo de números cercanos a cero para las curvas de nivel, por lo que se obtiene en algunas ocasiones `RuntimeWarning: divide by zero encountered in log` y las curvas de nivel pueden llegar a mostrarse extrañas, pero siempre simulando de manera correcta.

Problema 3. Considere el ejemplo referente al número de fallas de bombas de agua en una central nuclear donde p_i representa el número de fallas en el tiempo de operación t_i , con $i = 1, \dots, n$.

Se considera el modelo $p_i \sim Poisson(\lambda_i t_i)$, (las λ_i son independientes entre si), con distribuciones a priori $\lambda_i | \beta \sim Gama(\alpha, \beta)$ y $\beta \sim Gama(\gamma, \delta)$, por lo tanto:

$$f(\lambda_1, \dots, \lambda_n, \beta) = f(\lambda_1 | \beta) f(\lambda_2 | \beta) \dots f(\lambda_n | \beta) f(\beta)$$

Para la distribución posterior se tiene:

$$f(\lambda_1, \dots, \lambda_n, \beta | \bar{p}) \propto L(\bar{p}, \bar{\lambda}, \beta) f(\lambda_1, \dots, \lambda_n, \beta)$$

Simule valores de la distribución posterior $f(\lambda_1, \dots, \lambda_n, \beta | \bar{p})$, usando un kernel híbrido, considerando las propuestas:

$$\lambda_i | \bar{\lambda}_{-i}, \beta, \bar{t} \sim Gama(t_i p_i + \alpha, \beta + 1)$$

$$\beta | \bar{\lambda}, \bar{t} \sim Gama\left(n\alpha + \gamma, \delta + \sum_{i=1}^n \lambda_i\right).$$

Verifique que estas son propuestas Gibbs.

Use los datos del Cuadro 1 con los parámetros a priori $\alpha = 1.8$, $\gamma = 0.01$ y $\delta = 1$.

Bomba (i)	1	2	3	4	5	6	7	8	9	10
T. de uso (t_i)	94.32	15.72	62.88	125.76	5.24	31.44	1.05	1.05	2.1	10.48
# de fallas (p_i)	5	1	5	14	3	19	1	1	4	22

Table 1: Datos de bombas de agua en centrales nucleares (Robert y Casella, p. 385) para el ejemplo 8.3.

Se aplicó el algoritmo solicitado para la función especificada, pero se cambiaron las propuestas planteadas, debido a que no aproximaban lo debido. Más adelante se hace una justificación de dicho cambio. El algoritmo puede ser encontrado en la función `bombasAguaMHMC` del código adjunto.

El modelado está basado en la suposición de que los fallos de cada bomba siguen una distribución de Poisson cuyo parámetro es proporcional al tiempo de uso de cada bomba.¹.

¹Robert y Casella ejemplo 8.3)

Dichas constantes de proporcionalidad λ_i son las que pretendemos estimar con este método. Con las distribuciones a priori dadas, tenemos la distribución conjunta:

$$\begin{aligned} & \pi(\lambda_1, \dots, \lambda_{10}, \beta | t_1, \dots, t_{10}, p_1, \dots, p_{10}) \\ & \propto \prod_{i=1}^{10} \{(\lambda_i t_i)^{p_i} \cdot e^{-\lambda_i t_i} \cdot \lambda_i^{\alpha-1} \cdot e^{-\beta \lambda_i}\} \beta^{10\alpha} \beta^{\gamma-1} e^{-\delta \beta} \\ & \propto \prod_{i=1}^{10} \{\lambda_i^{p_i+\alpha-1} \cdot e^{-(t_i+\beta)\lambda_i}\} \beta^{10\alpha+\gamma-1} e^{-\delta \beta} \end{aligned}$$

de esta manera una descomposición natural para π en distribuciones condicionales es

$$\lambda_i | \bar{\lambda}_{-i}, \beta, \bar{t} \sim Gama(p_i + \alpha, \beta + t_i)$$

$$\beta | \bar{\lambda}, \bar{t} \sim Gama\left(n\alpha + \gamma, \delta + \sum_{i=1}^n \lambda_i\right).$$

Por lo tanto se usaron estas propuestas para la simulación y los resultados se muestran en las figuras 4 y 5. En las que se grafica la simulación de cada parametro a estimar y se toma el promedio, el cual tambien se grafica y se puede apreciar como la muestra para cada λ y para β se acumula alrededor de promedio.

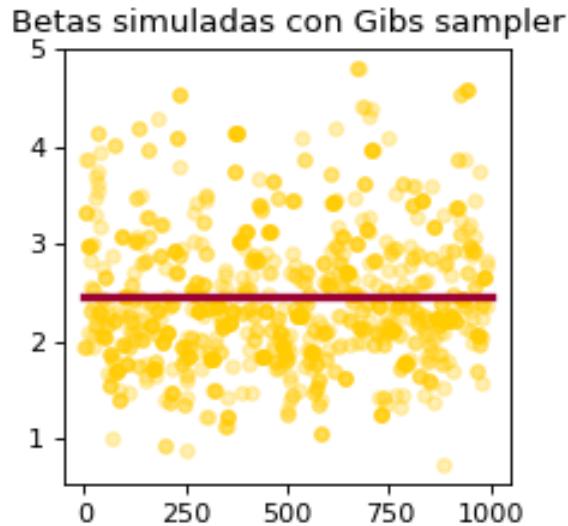


Figure 4: Se grafican las β obtenidas en una muestra de tamaño 1000 y se grafica la línea correspondiente al promedio muestral

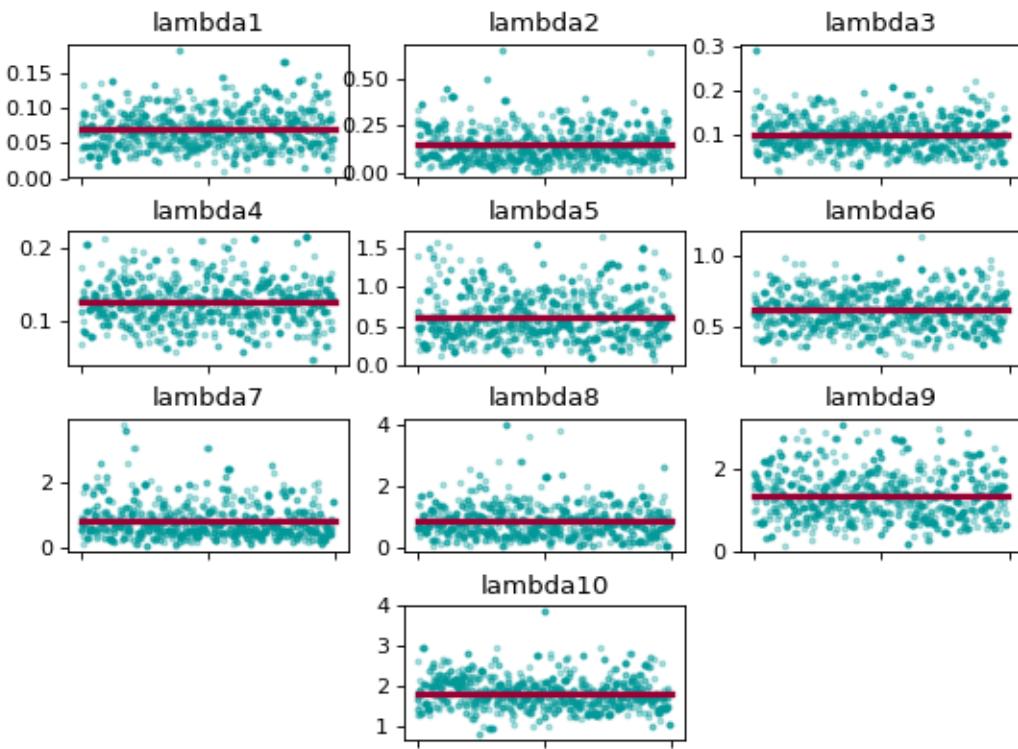


Figure 5: Se grafica cada λ_i para una muestra de tamaño 1000, y se grafica la línea correspondiente al promedio muestral para cada λ_i

Parámetro	media muestral
λ_1	0.0705370857746
λ_2	0.149854183199
λ_3	0.102397970937
λ_4	0.125647388644
λ_5	0.622845921325
λ_6	0.613016512921
λ_7	0.803668677919
λ_8	0.857987771718
λ_9	1.37131964859
λ_{10}	1.83555326618
β	2.45226496655

Table 2: Tabla de promedios

Tarea 9. MCMC: Tarea final

Ricardo Chávez Cáliz

15 de noviembre de 2017

1. Problema en ecología

Sean X_1, \dots, X_m variables aleatorias donde X_i denota el número de individuos de una especie en cierta región. Suponga que $X_i | N, p \sim \text{Binomial}(N, p)$, entonces

$$f(\bar{x} | N, p) = \prod_{i=1}^m \frac{N!}{x_i!(N-x_i)!} p^{x_i} (1-p)^{N-x_i}.$$

Como se espera que sea difícil observar a los individuos se considera a priori $p \sim \text{Beta}(\alpha, \beta)$ con $\alpha = 1, \beta = 20$, para N consideraremos una distribución uniforme discreta, i.e. $N \sim U\{0, 1, 2, \dots, N_{max}\}$, como la especie no es muy abundante consideramos $N_{max} = 1000$.

A partir del algoritmo MH, se simularon valores de la distribución posterior $f(N, p | \bar{x})$ con los siguientes datos $4, 9, 6, 7, 8, 2, 8, 7, 5, 5, 3, 9, 4, 5, 9, 8, 7, 5, 3, 2; m = 20$. Para esto se consideró como distribución inicial $p \sim U(0, 1)$ y $N \sim U_d\{M, M+1, \dots, N_{max}\}$ donde $M = \max_{i \in \{1, \dots, m\}}(X_i)$ y un kernel híbrido con las siguientes propuestas:

- Propuesta 1: Condicional total de p (kernel Gibbs).

$$p' | N, \bar{x} \sim \text{Beta}(\alpha + \sum_{i=1}^m X_i, \beta + mN \sum_{i=1}^m X_i)$$

- Propuesta 2: La apriori.

$$p' \sim \text{Beta}(1, 20) \text{ y } N_p \sim U\{0, 1, 2, \dots, N_{max}\}$$

- Propuesta 3: Caminata aleatoria

$$N_p = N + \epsilon, \quad \mathbb{P}(\epsilon = 1) = \frac{1}{2} = \mathbb{P}(\epsilon = -1).$$

Tenemos que los datos se distribuyen binomialmente por lo que la verosimilitud está dada por

$$f(\bar{x} | N, p) = \prod_{i=1}^m \frac{N!}{x_i!(N-x_i)!} p^{x_i} (1-p)^{N-x_i}.$$

Con las distribuciones a priori consideradas tenemos que la posterior $f(N, p|\bar{x})$ es proporcional a

$$\frac{(N!)^m}{\prod_{i=1}^m (N - x_i)!} \cdot p^{\sum_{i=1}^m x_i} \cdot (1-p)^{N \cdot m} \cdot (1-p)^{-\sum_{i=1}^m x_i} \cdot p^{\alpha-1} \cdot (1-p)^{\beta-1}$$

así $\log(f(N, p|\bar{x})) = m \cdot \log(N!) + \sum_{i=1}^m x_i \cdot \log(p) + N \cdot m \cdot \log(1-p) - \sum_{i=1}^m x_i \cdot \log(1-p) + (\alpha-1) \cdot \log(p) + (\beta-1) \cdot \log(1-p) - \sum_{i=1}^m \log((N - X_i)!)$. Como $\log(k!) = \log(\Gamma(k+1))$ se usó `gammaln` de `scipy.special` para calculos adecuados de $\log(f(N, p|\bar{x}))$

Con la información anterior se pudo calcular de manera adecuada la probabilidad de aceptación para cada propuesta

1. Como la propuesta es de Gibbs, entonces directamente se tiene que $\rho_1 = 1$
2. Considerando las funciones de densidad de las distibuciones a priori sin considerar las constantes si $g(N, p) = \log(f(N, p|\bar{x})) + (\alpha-1) \cdot \log(p) + (\beta-1) \cdot \log(1-p)$ se tiene

$$\rho_2 = e^{\min\{0, g(N_p, p'|\bar{x}) - g(N, p|\bar{x})\}}$$

3. Como la caminata aleatoria es una propuesta simétrica se tiene que $\rho_3 = \min\{1, \frac{f(N_p, p')}{f(N, p)}\}$, para cálculos computacionales se usó

$$\rho_3 = e^{\min\{0, \log(f(N_p, p'|\bar{x})) - \log(f(N, p|\bar{x}))\}}$$

Para la ejecución del algoritmo aquí mostrada se rechazaron el 18.58 % de las propuestas, el valor promedio de p fue de 0.0135318691986 y para N fue 455.966193239.

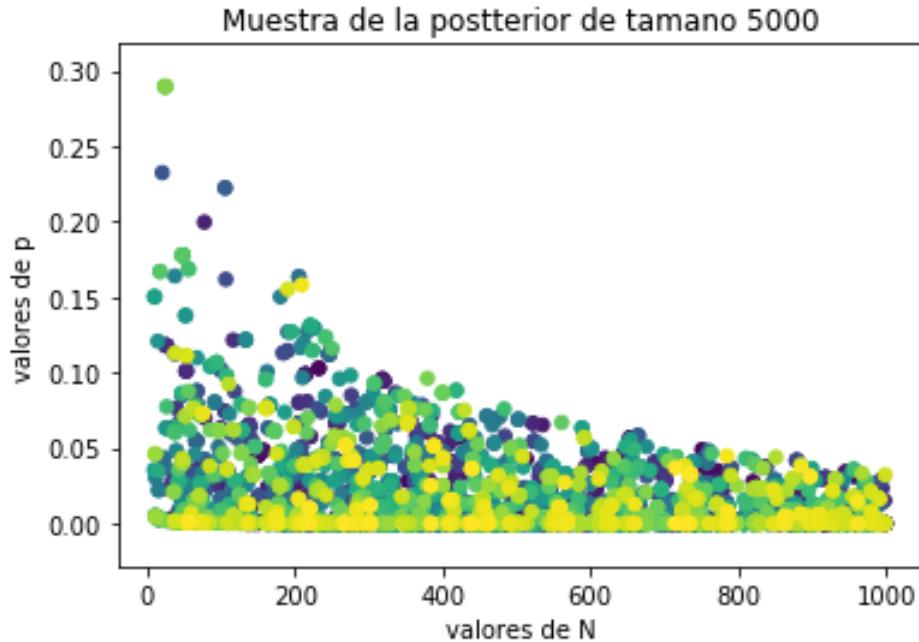


Figura 1: Muestra de tamaño 5000 para $f(N, p|\bar{x})$ obtenida del algoritmo MCMC

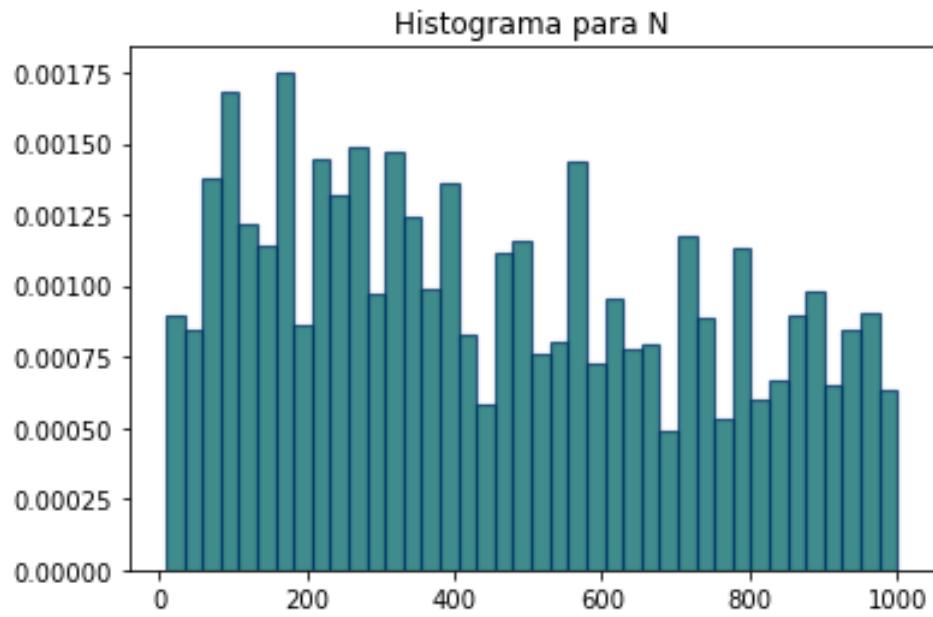


Figura 2: Histograma para N de la muestra de tamaño 5000 para $f(N, p|\bar{x})$ obtenida del algoritmo MCMC

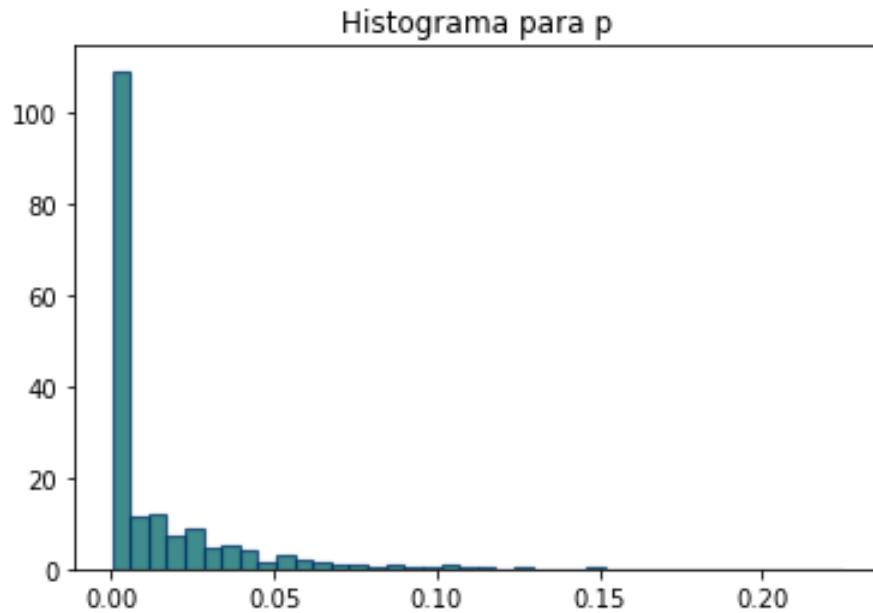


Figura 3: Histograma para p de la muestra de tamaño 5000 para $f(N, p|\bar{x})$ obtenida del algoritmo MCMC

2. Estudio de mercado

Se tiene un producto y se realiza una encuesta con el fin de estudiar cuánto se consume dependiendo de la edad. Sea Y_i el monto de compra y X_i la covariable la cual representa la edad.

Suponga que $Y_i \sim Po(\lambda_i)$ (distribución Poisson con intensidad λ_i)

$$\lambda_i = cg_b(x_i - a)$$

para g_b la siguiente función de liga

$$g_b(x) = \exp\left(-\frac{x^2}{2b^2}\right).$$

Si $\lambda_i = 0$ entonces $P(Y_i = 0) = 1$. a representa los años medio del segmento (años), c = gasto promedio (pesos), b = “amplitud” del segmento (años).

Considerando las siguientes distribuciones a priori:

$$a \sim N(35, 5), \quad c \sim Gama(3, 3/950), \quad b \sim Gama(2, 2/5)$$

El segundo parámetro de la normal es desviación estándar y el segundo parámetro de las gammas es taza (*rate*).

Usando MH se simularón de la distribución posterior de a, c y b .

Los datos son estos, $n = 100$:

```
X = [17, 14, 28, 51, 16, 59, 16, 54, 52, 16, 31, 31, 54, 26, 19, 13, 59, 48, 54, 23, 50, 59, 55, 37, 61, 53, 56, 31, 34, 15, 41, 14, 13, 13, 32, 46, 17, 52, 54, 25, 61, 15, 53, 39, 33, 52, 65, 35, 65, 26, 54, 16, 47, 14, 42, 47, 48, 25, 15, 46, 31, 50, 42, 23, 17, 47, 32, 65, 45, 28, 12, 22, 30, 36, 33, 16, 39, 50, 13, 23, 50, 34, 19, 46, 43, 56, 52, 42, 48, 55, 37, 21, 45, 64, 53, 16, 62, 16, 25, 62]
```

```
Y = [165, 9, 493, 0, 72, 0, 89, 0, 0, 70, 79, 96, 0, 1127, 548, 4, 0, 0, 0, 1522, 0, 0, 0, 0, 0, 0, 0, 80, 5, 38, 0, 11, 8, 4, 31, 0, 174, 0, 0, 1305, 0, 39, 0, 0, 18, 0, 0, 4, 0, 1102, 0, 94, 0, 13, 0, 0, 1308, 33, 0, 90, 0, 0, 1466, 156, 0, 39, 0, 0, 496, 2, 1368, 190, 0, 12, 76, 0, 0, 5, 1497, 0, 6, 533, 0, 0, 0, 0, 0, 0, 1090, 0, 0, 0, 93, 0, 88, 1275, 0]
```

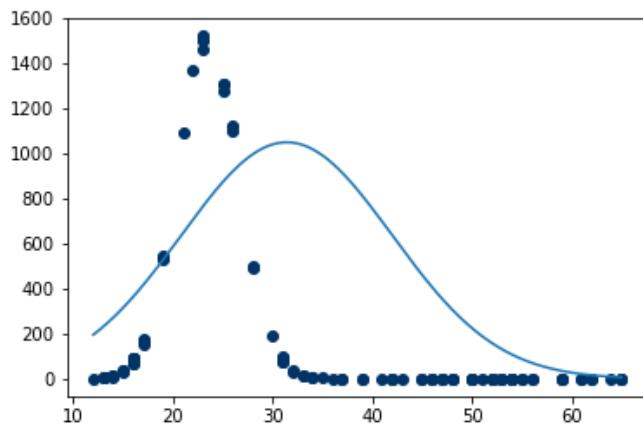


Figura 4: Muestra dada y gráfica de liga con valores aleatorios distribuidos como las funciones a priori

Para poder dar propuestas eficientes se graficó la muestra y a función de liga con parámetros aleatorios obtenidos según la distribución a priori dada, en este caso fueron $a = 31.4166339314$, $b = 10.5963029238$ y $c = 1049.90123303$, en base a lo observado parece razonable dar un kernel híbrido sencillo dado por caminatas aleatorias en cada componente, es decir

- Propuesta 1: Caminata aleatoria

$$a_p = a + N(0, 1)$$

- Propuesta 2: Caminata aleatoria

$$b_p = b + N(0, 0.1)$$

- Propuesta 3: Caminata aleatoria

$$c_p = c + N(0, 2)$$

Tenemos que los datos tienen distribución Poisson con parámetro $g_b(x - a)$, por lo tanto la verosimilitud está dada por

$$f(\bar{x}|a, b, c) = \prod_{i=1}^m \frac{c \cdot \exp\left(-\frac{(x_i-a)^2}{2b^2}\right)^{y_i} \cdot \exp\left(-c \cdot \exp\left(-\frac{(x_i-a)^2}{2b^2}\right)\right)}{y_i!}$$

Con las distribuciones a priori consideradas tenemos que el logaritmo de la posterior $f(a, b, c|\bar{x})$ es proporcional a

$$\log(c) \cdot \sum_{i=1}^m y_i + \left(\frac{1}{2b^2}\right) \cdot \sum_{i=1}^m (-y_i \cdot (x_i - a)^2) - \frac{c}{2b^2} \cdot \sum_{i=1}^m \exp(-(x_i - a)^2) - \frac{(a - 35)^2}{2 \cdot 5^2} - \frac{5}{2} \cdot b + \log(b) - \frac{950}{3} \dot{c} + 2 \cdot \log(c)$$

Con la información anterior se pudo calcular de manera adecuada la probabilidad de aceptación para cada propuesta, pues como la caminata aleatoria es una propuesta simétrica se tiene que $\rho = \min\{1, \frac{f(a', b', c')}{f(a, b, c)}\}$, para cálculos computacionales se usó

$$\rho = e^{\min\{0, \log(f(a', b', c'|\bar{x})) - \log(f(a, b, c|\bar{x}))\}}$$

Para la ejecución del algoritmo aquí mostrada se rechazaron el 62.7% de las propuestas en una muestra de 1000 elementos, los valores promedio para a, b y c fueron 23.6166478266, 3.13611422563, 1391.7006915. Usando estos valores para gráfica la función de liga se obtuvo el siguiente resultado.

Note que el parece que el modelo escogido con los parámetros estimados ajusta de buen modo a los datos y podría ser usado para la toma de decisiones en la estrategia mercadotécnica del producto.

Los resultados anteriores se pueden interpretar pensando que la edad promedio del consumidor del producto hipotético es de aproximadamente 23.61 años, gastando un promedio de 1391.70 pesos en el producto aproximadamente, con un rango de edad de + o - 3.1361.

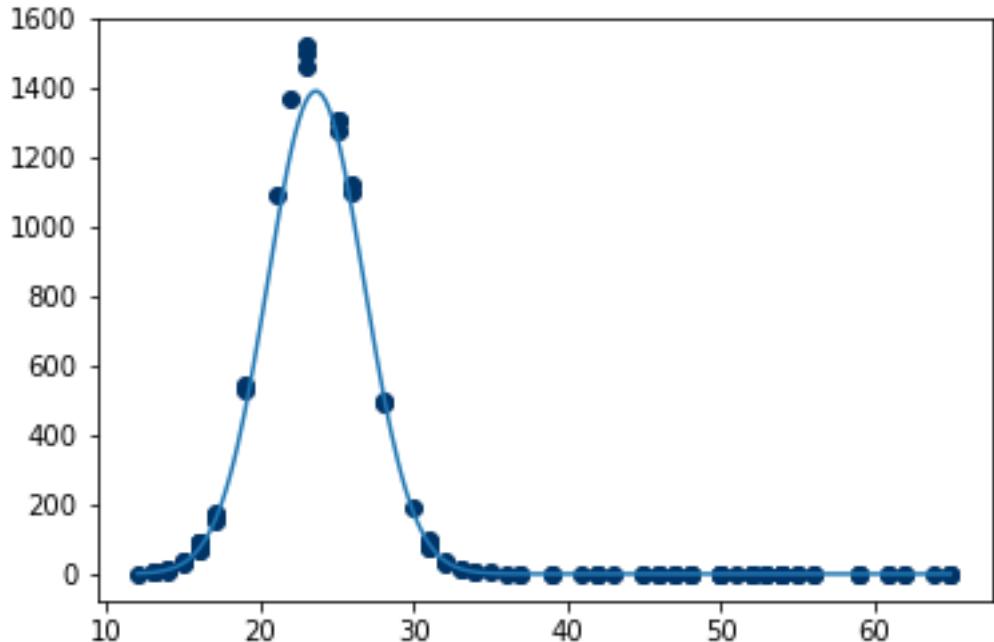


Figura 5: Muestra dada y función de liga

3. Software para estadística Bayesiana y MCMC

3.1. BUGS

En inteligencia artificial, un sistema experto es un sistema computacional que emula la habilidad de tomar decisiones como lo hace un humano experto en cierto tema. Los sistemas expertos están diseñados para resolver problemas complejos al razonar sobre conocimiento, principalmente representado por reglas del tipo "si-entonces" más que por código rutinario convencional. BUGS es un software que utiliza esto para realizar inferencia Bayesiana usando un Gibbs sampler. El usuario especifica el modelo estadístico, que puede ser de casi cualquier complejidad, simplemente al especificar las relaciones entre las variables. El software incluye un sistema experto que determina el MCMC apropiado basado en Gibbs sampler para analizar el modelo. El software permite controlar la ejecución del esquema y da la libertad de escoger entre una amplia variedad de tipos de outputs.

¿Cómo funciona? El modelo especificado pertenece a una clase conocida de Gráficas acíclicas dirigidas (DAGS), para las cuales existe una teoría matemática subyacente. Esto permite separar el análisis de estructuras complejas arbitrariamente grandes en una secuencia de cálculos computacionales relativamente sencillos. BUGS incluye una amplia variedad de algoritmos que sus sistemas expertos pueden asignar a cada tarea computacional.

Existe el OpenBUGS y el WinBUGS, una de las principales diferencias entre ellos es la manera en que el experto del sistema toma sus decisiones. WinBUGS define un algoritmo para cada tipo de cálculo computacional posible mientras que el número de los algoritmos que puedes hacer uso en OpenBUGS es mayor, haciéndolo mucho más extenso y flexible.

3.2. NIMBLE

Este software adopta y extiende BUGS como lenguaje de modelado y permite programar con los modelos que se van creando. Otras paqueterías que usan el lenguaje de BUGS sólo para MCMC, con NIMBLE puedes convertir código BUGS en objetos modelo y usarlos para cualquier algoritmo que se quiera, lo que incluye algoritmos provistos con NIMBLE y algoritmos escritos por el usuario. Usando funciones de NIMBLE, se puede extender BUGS al permitir múltiples parametrizaciones para distribuciones, funciones y distribuciones escritas por el usuario.

NIMBLE también provee MCMC, Monte Carlo secuenciales entre otras cosas.

Los algoritmos en NIMBLE están escritos para poderse adaptar a distintos modelos estadísticos, para MCMC se pueden asignar parámetros default para la elecciones del sampler pero es posible personalizar los samplers desde R por ejemplo, se pueden escoger los parámetros a muestrear en bloque, y es posible incluir tus propios samplers

3.3. JAGS

JAGS (Just Another Gibbs Sampler) es un programa para análisis de modelos Bayesiano jerárquicos que usan simulación MCMC no tan diferente a BUGS. Fue desarrollado teniendo en cuenta

- Tener un motor de cross-platform para el lenguaje de BUGs
- Se extendible, permitiendo a los usuarios escribir sus propias distribuciones y muestradores
- Ser una plataforma para experimentar con ideas en modelado Bayesiano

JAGS tiene una licencia del tipo GNU General Public License version.

3.4. DRAM

Es una propuesta hecha por Heikki Haario, Marko Laine, Antonietta Mira y Eero Saksman que combina dos poderosas ideas que aparecen en la literatura de MCMC: *Adaptive Metropolis samplers* y *delayed rejection*. Está demostrado que este sampler no-Markoviano es ergódico y se ha visto e varios ejemplos que la combinación de las ideas es eficiente. La adaptació claramente mejora la eficiencia del algoritmo y la parte del rechazo retrasado sirve en los casos donde buenas propuestas de distribución no están disponibles. Similarmente, el rechazo retrasado provee una manera sistemática de remediar un comienzo lento en el procesos de adaptación.

3.5. Rtwalk

Es una implementación del algoritmo de MCMC para "t-walk.en R, desarrollado por J. Andres Christen , Depends R (>= 2.8.0). Es un sampler de propósitos generales para distribuciones continuas arbitrarias que no requiere ajuste. Este algoritmo está implementado en Python, R, C++, C y MatLab.

emcee: The MCMC Hammer. Es un software libre para implementar de manera estable y certificada del affine-invariant ensemble sampler de MCMC propuesta por Goodman y Weare (2010). El algoritmo detrás de emcee tiene varias ventajas sobre los métodos tradicionales de muestreo de MCMC y tiene un excelente desempeño al medir la correlación en tiempo de la muestra.

Una de las mayores ventajas del algoritmo es que requiere ajuste manual en sólo uno o dos parámetros comparados con los aproximadamente N^2 que se ocupan ajustar para el algoritmo tradicional en un espacio de parámetros N-dimensional.

Explotando el paralelismo del método de ensable, emcee permite a cualquier usuario tomar ventaja de los múltiples núcleos del PC sin esfuerzo adicional. El código está disponible en línea en <http://dan.iel.fm/emcee/current/> bajo licencia del MIT

3.6. PyMCMC

Es un software creado por Chris Fonnesbeck, Anand Patil, David Huard, John Salvatier de licencia libre. PyMC es un modulo de Python que implementa modelos de estadística Bayesiana y algoritmos de ajuste, incluyendo MCMC. Debido a su flexibilidad y extensión es usado para una gran clase de problemas. Junto con funcionalidad de muestreo de núcleo, PyMC incluye métodos para resumir resultados, graficar, realizar bondad de ajuste y diagnostico de convergencia.

Características:

- Ajusta modelos estadísticos Bayesianos con cadenas de Markov y otros algoritmos
- Usa Numpy para las cuestiones numéricas siempre que es posible e incluye un sección para modelar procesos Gaussianos.
- Los muestreos en ciclos pueden ser pausados y ajustados manualmente, o bien ser salvados y reiniciados luego.
- Capacidad de resumir información, incluyendo tablas y gráficas.
- Los resultados pueden ser salvados en el disco como texto plano, Pyhton pickles, SQLite o bases de datos en MySQL, archivos hdf5.
- Disponibilidad de varios diagnósticos de convergencia.
- Extendibilidad: incorpora personalización sencilla de pasos en los métodos y distribuciones de probabilidad inusuales.
- Ciclos de MCMC pueden ser usados en programas mas extensos y los resultados puedes ser analizados con todo el poder de Python.

Tarea 10: Optimización clásica

Ricardo Chávez Cáliz

22 de noviembre de 2017

Se tienen datos provenientes de una distribución Normal truncada, es decir $x = (y_1, y_2, \dots, y_m, z)$ donde $z = (a, \dots, a)$ que consiste de $n - m$ elementos y $x_i \sim N(\theta, 1)$ si $x_i < a$, y $x_i = a$ en caso contrario. Por lo que

$$f(x_i|a, \theta) = \frac{1}{\sqrt{2\pi}} \cdot \exp\left\{-\frac{(x_i - \theta)^2}{2}\right\} \cdot [1 - \Phi(a - \theta)] \cdot I_{(a, \infty)}(x_i)$$

con función de verosimilitud

$$L(\theta|x) = \frac{1}{(2\pi)^{m/2}} \cdot \exp\left\{-\frac{1}{2} \sum_{i=1}^m (x_i - \theta)^2\right\} \cdot [1 - \Phi(a - \theta)]^{n-m}$$

donde $\Phi(\cdot)$ es la función de distribución de una normal estándar.

Se pide implementar el algoritmo EM, un Gibbs Sampler y el método de Newton-Raphson para obtener el estimador máximo verosímil de θ .

En nuestros conjunto de datos $n = 20, m = 15, a = 21$. datos = [18.221753, 18.418174, 18.720224, 19.067637, 19.128777, 19.402623, 19.507782, 19.580571, 19.632340, 19.930952, 20.116566, 20.142095, 20.445327, 20.461254, 20.646856, 21.000000, 21.000000, 21.000000, 21.000000,]

1. Algoritmo EM

Para cada paso de maximización se busca $\theta^{(j+1)} = \text{argmax} Q(\theta|\theta^{(j)}, x)$ donde $Q(\theta|\theta^{(j)}, x) = E_z [\log f(x, z|\theta)|\theta^{(j)}, x]$ calculada en el paso de expectativa. En este caso el logaritmo de la distribución conjunta está dada por

$$\log(f(x, z|\theta)) \propto -\frac{1}{2} \sum_{i=1}^m (x_i - \theta)^2 - \frac{1}{2} \sum_{i=m+1}^n (z_i - \theta)^2$$

De esta manera tenemos que:

$$Q(\theta|\theta^{(j)}, x) = \mathbb{E}_z \left[-\frac{1}{2} \sum_{i=1}^m (x_i - \theta)^2 - \frac{1}{2} \sum_{i=m+1}^n (z_i - \theta)^2 \mid \theta^{(j)}, x \right]$$

la cual probamos que es maximizada con

$$\widehat{\theta}^{(j+1)} = \frac{m}{n} \bar{X} + \frac{n-m}{n} \cdot \left[\widehat{\theta}^{(j)} + \frac{\varphi(a - \widehat{\theta}^{(j)})}{1 - \Phi(a - \widehat{\theta}^{(j)})} \right]$$

donde \bar{X} es el promedio muestral y $\varphi(\cdot)$, $\Phi(\cdot)$ son las funciones de densidad y de distribución para una normal estándar respectivamente.

Con la expresión anterior para $\widehat{\theta}^{(j)}$ la implementación del algoritmo EM es directa ya que estableciendo $\theta^0 = \bar{X}$ basta calcular $\theta^{(j)}$ hasta un cierto límite de iteraciones (se estableció 100 como límite) verificando que el $\theta^{(j+1)} - \theta^{(j)} > 1e-6$, de lo contrario se detienen las iteraciones y se devuelve el último θ^j calculado. Dicha implementación puede encontrarse en el código de Python adjunto.

Para verificar el buen comportamiento de la implementación se simuló una muestra normal estándar censurada para valores mayores o iguales a 1 de tamaño 1000. Se calculó el promedio muestral sin tomar en cuenta los datos censurados -0.306571650501 y luego el estimador obtenido con el algoritmo EM y en 4 iteraciones se obtuvo $\widehat{\theta} = -0.0496753870332$ que es más cercano a cero que el primer estimador como se espera.

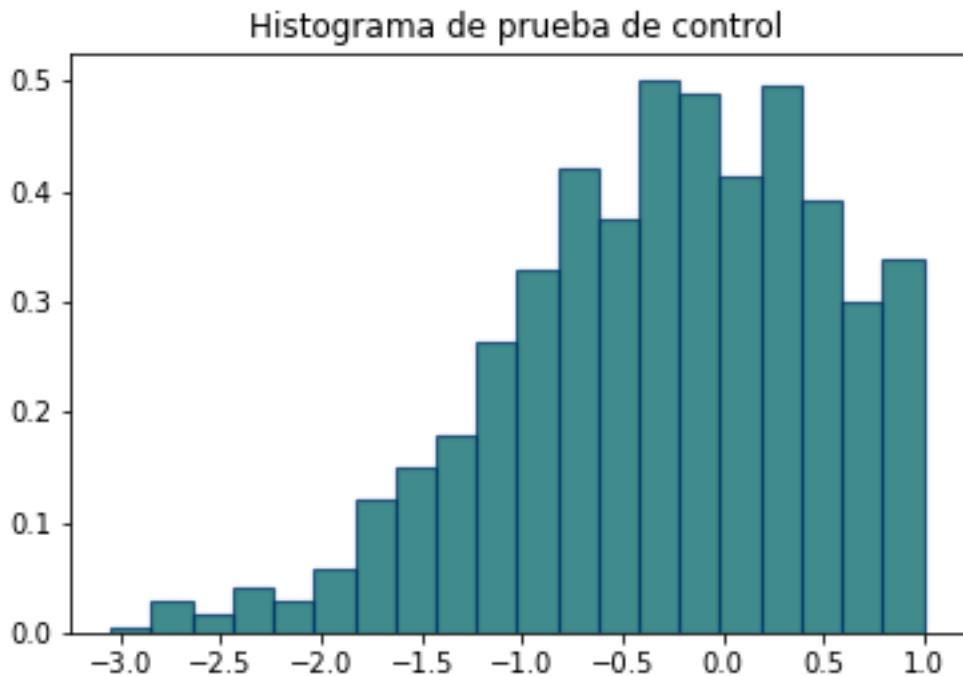


Figura 1: Histograma de la muestra dada, removiendo los datos mayores o iguales a 1

Para los datos aquí presentados se obtuvo que el estimador de la media ignorando censura es 19.92114655, el estimador de la media con algoritmoEM en 5 iteraciones es 20.0552242528 en un tiempo de ejecución 0.00895652321867.

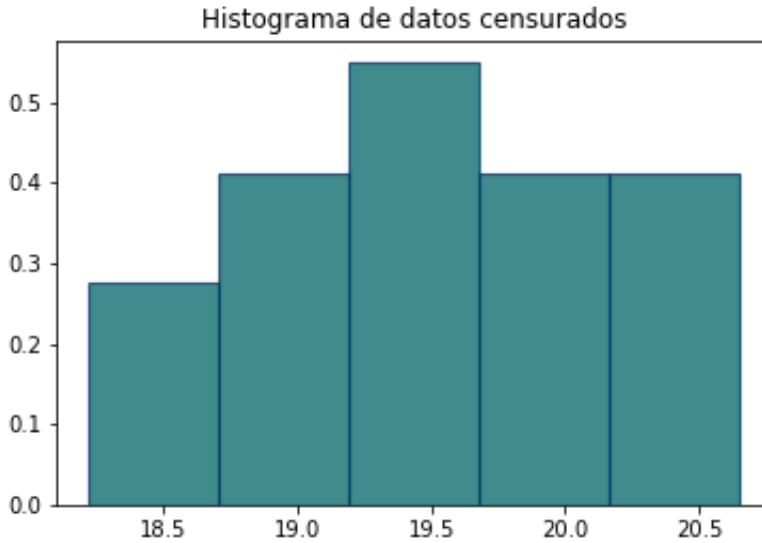


Figura 2: Histograma de la muestra normal estándar con censura, removiendo los datos mayores o iguales a 21

2. Gibbs sampler

se consideró un Gibbs Sampler con parámetro artificiales z_1, \dots, Z_{n-m} los cuales fueron añadidos en la posterior $f(\theta, z_1, \dots, z_{n-m} | x)$ y se implementó MCMC. Para esto consideramos las condicionales totales $f(z_i | \theta, x)$ las cuales se distribuyen como una normal truncada las cuales podemos simular con la inversa numérica Φ^{-1} , incluida en `scipy.stats.norm` con la función `ppf`. Considerando la distribución a priori para $\theta \sim N(18, 1)$ se obtuvo que el promedio de las θ simuladas para una muestra de tamaño 5000 fue de 20.054202679 en un tiempo de ejecución de 5.38168182815.

Para analizar el comportamiento de la cadena se analiza la muestra simulada para los θ obtenidos, para esto se grafica el logaritmo de el valor obtenido para θ en cada simulación respecto a la iteración correspondiente

Como se muestra en la figura 3 el burn-in de la cadena es muy corto y por ello en la figura 4 podemos exponer los últimos 4985 elementos de la muestra para θ y z_1 , si riesgo a notar elementos que no corresponden a las simulaciones deseadas. Note que los valores de θ se comportan como una normal con media aproximada a lo calculado antes y z_i correspondiente a los valores de la cola de una normal. En la figura 5 se muestra la gráfica de la trayectoria de la cadena con un menor número de simulaciones.

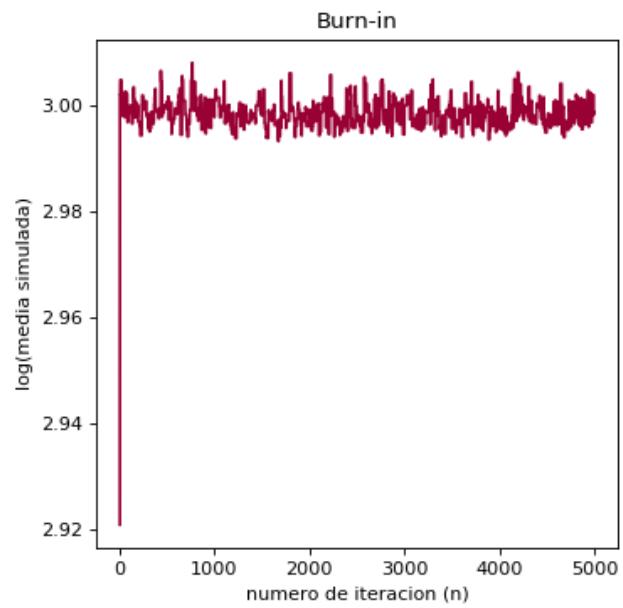


Figura 3: Se muestra el burn-in de la cadena

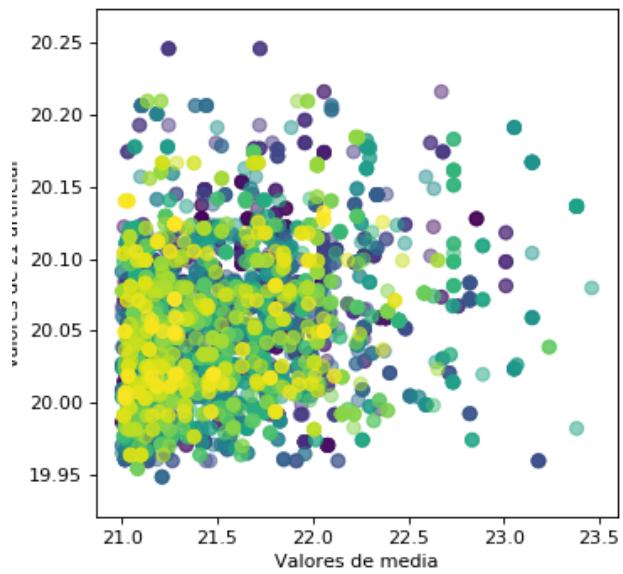


Figura 4: Muestra para θ y z_1

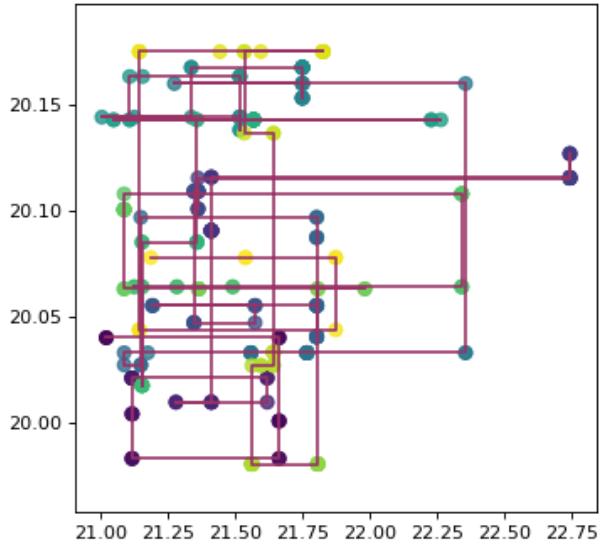


Figura 5: Trayectoria de la cadena

3. Método de Newton-Raphson

Se implementó el método de Newton-Raphson para encontrar a $\hat{\theta}$ que maximice la log-verosimilitud $l(\theta|x) = \log(L(\theta|x))$. Para esto se calculó la derivada de log-verosimilitud:

$$\frac{d l(\theta|x)}{d\theta} = \sum_{i=1}^m (X_i - \theta) + \frac{(n-m) \cdot \varphi(a-\theta)}{1 - \Phi(a-\theta)}$$

Para encontrar sus raíces por el método de Newton-Raphson fue necesario también calcular la segunda derivada.

$$\frac{d^2 l(\theta|x)}{d\theta^2} = -m - \frac{(n-m) \cdot \varphi(a-\theta)}{[1 - \phi(a-\theta)]^2} \cdot \{(a-\theta) \cdot [\phi(a-\theta) - 1] + (\varphi(a-\theta))^2\}$$

Dicha implementación puede encontrarse en el código de Python adjunto. Para los datos aquí presentados se obtuvo que la raíz es 20.0552242606 y se llegó a la solución en 3 iteraciones con tiempo de ejecución 0.0212303297718

Método	Estimación	tiempo de ejecución
AlgoritmoME	20.0552242528	0.00895652321867
Gibbs Sampler	20.054202679	5.38168182815
Newton-Raphson	20.0552242606	0.0212303297718

Tabla 1: Tabla comparativa de resultados

Topología estocástica: Gráficas aleatorias y simulación

Ricardo Esteban Chávez Cáliz

5 de diciembre de 2017

Resumen

La topología estocástica se encarga de estudiar propiedades topológicas de espacios definidos aleatoriamente en los que se tiene control sobre los parámetros que determinan el comportamiento aleatorio del espacio. Normalmente la validez de alguna propiedad depende de que los parámetros satisfagan ciertas condiciones. Para verificar la rigurosidad de estas condiciones se puede recurrir a simular dichos espacios. Dependiendo del modelo elegido dicha simulación puede ser sencilla, sin embargo puede que no siempre sea así. En este trabajo se describe el modelo Erdős-Renyi para gráficas aleatorias y se presentan simulaciones para el mismo. Después se presentan implementaciones de algoritmos de tipo MCMC discreto para simulaciones más complicadas.

1. Introducción

En muchos contextos científicos aparecen espacios topológicos complejos de manera natural. La teoría probabilista provee varias aproximaciones para modelar dichos espacios. Incluso en configuraciones complejas, se pueden estudiar invariantes topológicos tales como homología y cohomología haciendo aproximaciones. Típicamente estamos interesados en el comportamiento asintótico cuando un parámetro tiende infinito. En este sentido la topología estocástica surge como herramienta para la topología como un análogo a la mecánica estadística que estudia el comportamiento de sistemas físicos macroscópicos cuando la mecánica determinista haya estos sistemas muy complicados de resolver.

Si bien la motivación para estudiar topología estocástica encuentra su lugar en problemas de aplicación como la mecánica estadística, espacios de configuración de enlaces mecánicos, robótica, simulaciones animadas, manufactura etc., recientes trabajos han utilizado las técnicas de topología estocástica para proveer análogos probabilistas a conjeturas de topología muy importantes como la conjetura de Whitehead [AC15]. Estos análogos pueden ser interpretados como *evidencia estadística* para la conjetura en cuestión, la idea detrás de estos resultados es demostrar que el enunciado es verdadero (casi seguramente) en el modelo probabilista bajo ciertas condiciones. Una amplia variedad de técnicas han sido usadas en esta área, entre ellas el usar algoritmos que muestren el espacio de búsqueda, tanto en aplicaciones prácticas como teóricas la posibilidad de hacer simulaciones ha sido de gran ayuda. En lo aplicado han mostrado hasta el momento ser las técnicas más exitosas en planeación motriz, por ejemplo [VA15]. En lo teórico ha permitido determinar si las condiciones establecidas en los parámetros son lo suficientemente rigurosas o si las conjeturas que se tienen sobre ciertas propiedades son válidas [LA13].

Entre los espacios más manejables a considerar en topología estocástica encontramos las gráficas. El modelo Erdős–Rényi, nombrado así por ser un estudio que realizaron los matemáticos Paul Erdős y Alfréd Rényi, es uno de los métodos empleados en la generación de gráficas aleatorias. En este modelo se tiene que cada vértice se conecta independientemente y con igual probabilidad con el resto de los vértices. En la sección 2 de este trabajo se da la definición formal del modelo, algunos de los resultados teóricos que hay en este y las simulaciones realizadas en Python que verifican dichos resultados.

A veces se quiere hacer muestreros en espacios con cierta propiedad, por ejemplo asegurar que la gráfica aleatoria no tenga ciclos (ver [VA15]). De esta manera se pierde la propiedad i.i.d. en las aristas que es usada en la sección 2. Es por eso que se recurre a simular gráficas usando algoritmos del tipo MCMC discreto. En la sección 3 de este trabajo presentamos un algoritmo de esta naturaleza con la justificación de que realmente simula árboles aleatorios de tamaño n de manera uniforme.

Si bien este trabajo se enfoca en gráficas aleatorias, las ideas pueden ser extendidas a modelos de mayor dimensión usando software más sofisticado (SAGE, Javaplex, Dionysus).

2. Gráficas aleatorias Erdős Renyi

La idea del modelo consiste en considerar n vértices y ponemos cada una de las posibles aristas con probabilidad p , es decir que tendremos $\binom{n}{2}$ variables aleatorias idéntica e independientemente distribuidas Bernoulli con parámetro p que representan las aristas.

Denotaremos por $\mathbb{G}(n, p)$ a el espacio de probabilidad formado por las gráficas de n vértices, la sigma álgebra será el conjunto potencia y la medida de probabilidad está dada por

$$\mathcal{P}(G \in \mathbb{G}(n, p)) = p^m(1-p)^{\binom{n}{2}-m}$$

Note que si p es cercano a 1, es muy posible que casi todos los vértices estén enlazados entre sí. Una pregunta natural es ¿cómo debe ser p respecto a n para poder asegurar que la gráfica sea conexa cuando n tiende a infinito? El siguiente teorema, cuya demostración se encuentra en [VA59] contesta esta pregunta.

Teorema 1. *Sea $\omega(n)$ una función que tiende a infinito arbitrariamente lento a medida que n tiende a infinito.*

- Si $p \geq \frac{\log(n) + \omega(n)}{n}$ entonces

$$\lim_{n \rightarrow \infty} \mathcal{P}(G \in \mathbb{G}(n, p) \text{ sea conexa}) = 1$$

- Si $p \leq \frac{\log(n) - \omega(n)}{n}$ entonces

$$\lim_{n \rightarrow \infty} \mathcal{P}(G \in \mathbb{G}(n, p) \text{ sea desconexa}) = 1$$

Para entender mejor el teorema anterior se graficó la cota enunciada en el teorema para cada n , ver figura 1. Note que a medida que n crece el valor que debemos tomar para p puede ser cada vez más pequeño, es decir que a pesar de que el número de vértices que debemos conectar es cada vez mayor, no es necesario pedir que la probabilidad con que aparecen las aristas crezca. Este resultado que a principio no parece intuitivo puede entenderse notando que cuando el número de vértices crece en orden lineal entonces el número de aristas crece en orden cuadrático, y por ello la cota para p disminuye en el orden que indica el teorema.

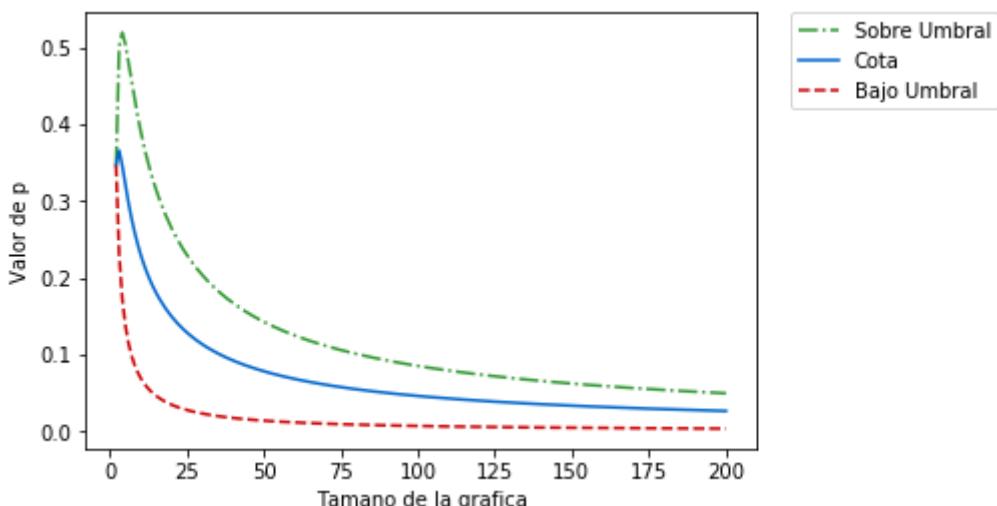


Figura 1: Gráfica de la cota en términos de n , también se muestran las gráficas cuando $\omega(n) = \log(n/2)$ que satisface la condición de teorema y es usada en las simulaciones siguientes.

2.1. Simulación en $G(n,p)$

Simular gráficas en este modelo es bastante sencillo, notando que la información de un gráfica está resumida en la matriz de adyacencia asociada a esta. Si la gráfica G tiene n vértices entonces la entrada $a_{i,j}$ en la matriz de adyacencia de tamaño $n \times n$ es igual a 1 si el vértice i y el vértice j están conectados, y es 0 en el otro caso. Al considerar matrices simples no dirigidas esta matriz es simétrica y con ceros en la diagonal. Así basta simular para cada entrada por arriba de la diagonal una variable aleatoria $Bernoulli(p)$.

La implementación en Python de estas simulaciones se puede encontrar en la función `erdosRenyi` que aparece en el código adjunto. Se usó la librería `NetworkX` que permite crear y modificar gráficas directamente al considerarlas como objetos en Python, gracias a esto no fue necesario guardar la información en una matriz para luego ser interpretado por `networkX` como una gráfica, así las aristas de la gráfica se añadieron directamente siguiendo el algoritmo antes descrito.

En la figura 2 se muestra un conjunto de gráficas obtenidas con dicho algoritmo, fijando $n = 10$ y variando el parámetro p como se indica en cada caso.

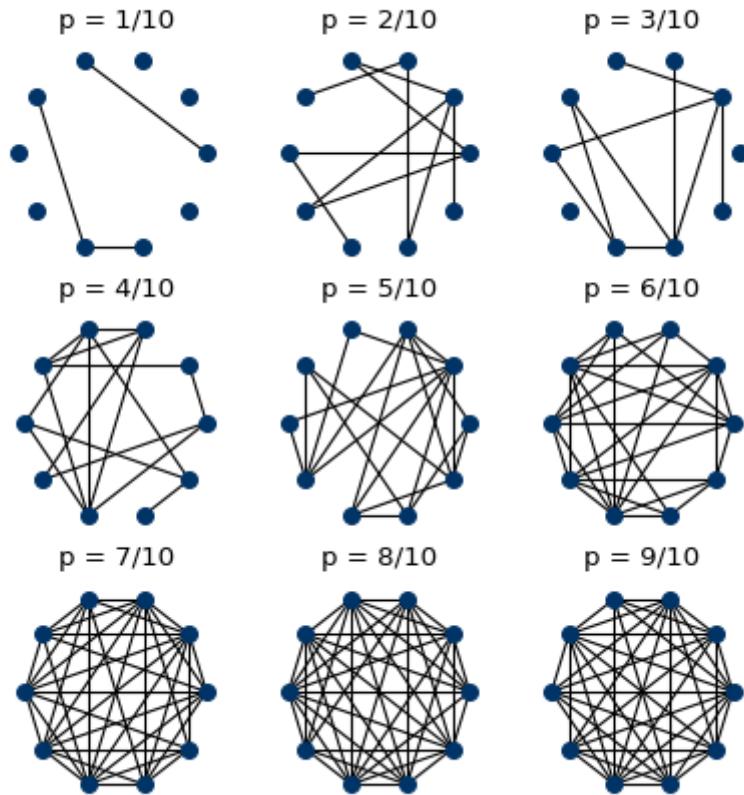


Figura 2: Elementos aleatorios de gráficas aleatorias Erdős-Renyi con $n=10$ y variando p

Como se menciona en la introducción la utilidad de las simulaciones para el desarrollo de la topología estocástica reside en poder verificar la rigurosidad de las condiciones impuestas para que cierta propiedad se satisfaga, en este caso la propiedad es la conexidad y las condiciones están dadas por la cota $\log(n)/n$ para p . Para exemplificar esto se hicieron simulaciones en Python de gráficas variando n y tomando p como indica el teorema con $\omega(n) = \log(n/2)$. Para cada tamaño de n se repite la simulación 30 veces y se verifica si la gráfica simulada es conexa o no con la función `is_connected(G)` de la librería de `networkX`. De esta manera obtenemos una aproximación de la probabilidad de que una gráfica sea conexa si se toma con las condiciones indicadas en el teorema. En la figura 3, se graficó para cada n dicha probabilidad estimada para cada caso de p .

Analizando la figura 3 es fácil convencerse de la validez del teorema viendo que para n suficientemente grande $n = 50$, la gráfica muestra lo establecido en el teorema.

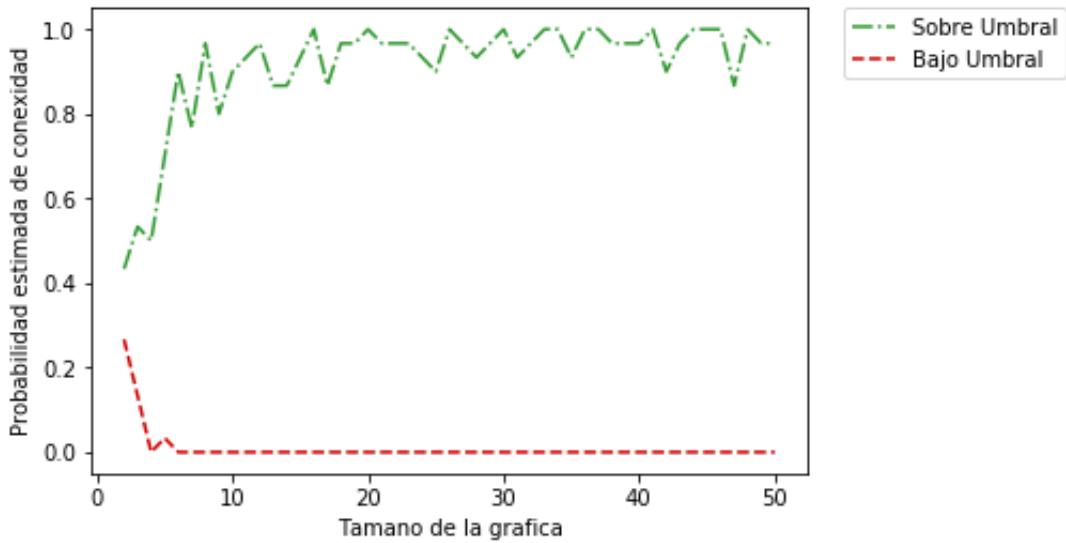


Figura 3: Probabilidad estimada de conectividad para distintas gráficas tomando el parámetro p como se indica en el teorema

La ejecución del algoritmo erdosRenyi implementado requiere de la generación de $\binom{n}{2}$ variables aleatorias Bernoulli las cuales son simuladas en el código de Python adjunto en la función **Bernoulli**, las cuales hacen uso de la librería de **numpy** para generar los números aleatorios básicos, librería que usa el algoritmo **Mersenne-Twister**. Por lo que el tiempo de ejecución del algoritmo es de $O(n^2)$. En la figura 4 se muestran los resultados de los tiempos de ejecución haciendo variar n .

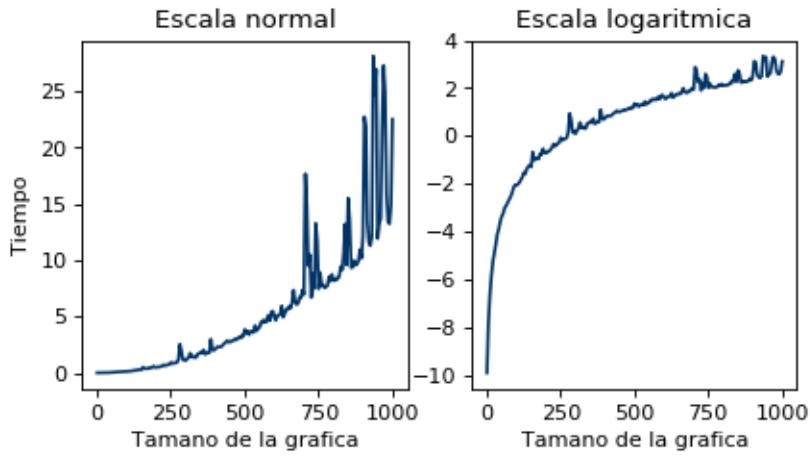


Figura 4: Tiempos de ejecución en segundos del algoritmo erdosRenyi variando el tamaño de la gráfica. Se presenta la escala norma y la logarítmica

3. Árboles aleatorios uniformes: MCMC discreto

En esta sección se describe un algoritmo que dada una gráfica G no dirigida con n vértices, produce un árbol maximal de G tomado uniformemente entre todos los posibles. También se dan referencias para ver que el tiempo esperado de ejecución del algoritmo para cada árbol es de $O(n \cdot \log(n))$ para casi todas las gráficas y $O(n^3)$ para los peores casos. Se mencionan los algoritmos deterministas conocidos los cuales son mucho más complicados y requieren un tiempo de al menos $O(n^3)$ para generar un árbol con estas condiciones.

Uno de los primeros algoritmos publicados para este problema [Guénocche83] tiene un tiempo de

ejecución de n^5 , está basado en el hecho de que el número total de árboles dirigidos en un gráficua pueden ser calculados explícitamente al calcular un determinante de tamaño $n \times n$. El algoritmo considera las aristas de la gráficua como etiquetados desde 1 a m . Cada árbol maximal es etiquetado por el conjunto de sus aristas. Esto induce un orden lexicográfico en el conjunto de los árboles y el mismo árbol puede ser encontrado calculando a lo más m determinantes. Subsecuentes mejoras [CCN88], [CCM88] reducen el número de cálculos de determinantes a hacer reduciendo así el tiempo de ejecución a $O(n^3)$ ó $O(L(n))$ donde $L(n)$ es el tiempo para multiplicar matrices de $n \times n$. Pero los nuevos algoritmos son más complicados.

3.1. Algoritmo Generate

Consideré una partícula moviéndose en una gráficua simple no dirigida $G = (V, A)$ con n vértices. En cada paso la partícula se mueve aleatoriamente del vértice en el que está a alguno de sus vecinos escogido de manera uniforme. Este proceso estocástico es una cadena de Markov; llamada caminata aleatoria simple en la gráficua G . Simulaciones de una caminata de este tipo son usadas para generar un árbol maximal de G uniformemente sobre todos los posibles árboles de G con un algoritmo muy simple:

Algoritmo Generate

1. Seleccione un vértice arbitrario s de G (uniformemente)
2. Simule una caminata aleatoria simple en una gráficua G hasta que cada vértice es visitado.
3. Para cada i en $V - s$ coleccione la arista (j, i) donde la primera entrada corresponde al vértice en dónde se encontraba la partícula antes de pasar por primera vez por i . Sea T la colección de dichas aristas
4. Devuelva el conjunto T .

Note que el conjunto T es un árbol maximal ya que contiene $|V| - 1$ aristas, pues tiene una arista para cada vértice en G excepto s y no tiene ciclos por construcción.

El algoritmo **Generate** está basado en simulación de cadenas de Markov en el espacio de objetos de interés. En este caso la cadena de Markov tiene distribución estacionaria $\pi_i = d_i / \sum_{j \in V} d_j$ donde d_i es el grado del vértice i . La gráficua ponderada asociada a esta cadena $G_M = (V, E')$, es obtenida reemplazando cada arista $\{i, j\} \in A$ por dos aristas dirigidas; (i, j) con peso $1/d_i$ y (j, i) con peso $1/d_j$. La justificación de que el algoritmo en efecto provee una manera de obtener árboles maximales con distribución uniforme, es resumida en los siguientes tres resultados, cuyas demostraciones completas pueden ser encontradas en [Bro89].

Denotaremos por $\mathcal{T}_i(G_M)$ a la familia de árboles maximales dirigidos de G_M con raíz i cuando no hacemos reparación sobre la raíz los denotaremos simplemente por $\mathcal{T}(G_M)$

Teorema 2. *Sea M una cadena de Markov irreducible en n estados con distribución estacionaria π_1, \dots, π_n . Sea G_M la gráficua ponderada asociada a M . Entonces*

$$\pi_i = \frac{\sum_{T \in \mathcal{T}_i(G_M)} \omega(T)}{\sum_{T \in \mathcal{T}(G_M)} \omega(T)}$$

donde $\omega(T) = \prod_{a \in A(T)} \omega(a)$, es decir que el peso de un árbol dirigido ponderado se define como el producto de los pesos de las aristas del árbol.

Definimos el árbol de proa (*forward tree*) a tiempo t , F_t como sigue: Sea I_t el conjunto de estados visitados hasta antes del tiempo $t + 1$. Para cada $i \in I_t$, sea $p(i, t)$ la primera vez que el estado i fue visitado. La raíz del árbol F_t es X_0 y los aristas de F_t son $\{(X_{p(i,t)}, X_{p(i,t)-1}) | i \in I_t - X_0\}$, donde $(X_t)_{t \in \mathbb{N}}$ corresponde a la cadena de Markov dada por la caminata aleatoria. En otras palabras F_t está formado superponiendo las aristas que corresponden a la primera entrada a cada estado con orientación invertida. Claramente F_t es un árbol dirigido con raíz donde cada arista apunta desde las hojas a la raíz.

Sea C el *tiempo de cubrimiento*, es decir la primera vez que todos los estados fueron visitados. Claramente para $t \geq C$ el árbol F_t es un árbol dirigido maximal y $F_t = F_C$. Note que con la definición anterior la caminata aleatoria $\{X_t\}$ en los vértices de G_M induce una cadena de Markov $\{F_t\}$ en el espacio de todos los árboles dirigidos de G_M , llamada la cadena de árboles de proa. Para esta cadena todos los árboles no maximales son estados transitivos y cada árbol maximal es un estado absorbente. Más aún, el siguiente teorema establece la distribución de F_C .

Teorema 3. *Con la notación y condiciones del teorema anterior, denotando por C el tiempo de cubrimiento de M empezando de la distribución estacionaria. Sea F_C el árbol de proa en tiempo C . Entonces para cualquier árbol maximal dirigido con raíz, T de G_M*

$$\mathcal{P}(F_C = T) = \frac{\prod_{(i,j) \in T} P_{i,j}}{\sum_{T' \in T(G_M)} \prod_{(i,j) \in T'} P_{i,j}}$$

Corolario 1. (prueba del algoritmo `Generate`) *Sea M una caminata aleatoria simple en una gráfica conexa no dirigida $G = (V, E)$ empezando en algún vértice s , G_M la gráfica dirigida asociada a M y C tiempo de cubierta para G empezando de la distribución estacionaria, tenemos que F_C (el árbol de proa en G_M en tiempo C) sin considerar dirección es un árbol maximal de G con distribución aleatoria uniforme entre todos los posibles árboles maximales de G .*

La implementación en Python del algoritmo `Generate` puede encontrarse en la función `generate` que aparece en el código adjunto. Fijando G como la gráfica completa K_n podemos mediante este algoritmo muestrear uniformemente del conjunto de todos los árboles con n vértices. En la figura 5 se muestra un conjunto de árboles obtenidos con el algoritmo, los cuales son graficados con la función `draw_random` de la librería de `networkX`. La disposición de algunos de ellos puede hacer parecer que las gráficas aquí obtenidas no son árboles, sin embargo funciones en `networkX` permiten verificar lo contrario. Cabe mencionar que determinar el mínimo número de cruces es un problema NP-hard por lo que se entiende que la visualización adecuada de estas gráficas requiera software especializado ([Gephi](#) por ejemplo),

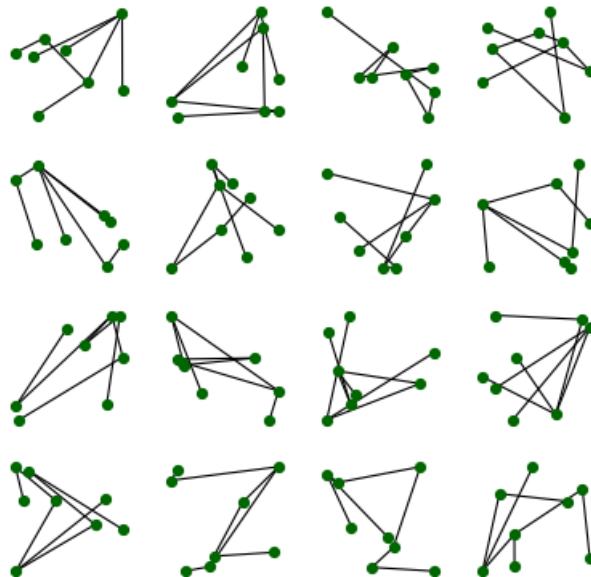


Figura 5: Árboles maximales escogidos aleatoriamente con distribución uniforme entre todos los posibles en una gráfica completa de 8 vértices

Llamaremos *tiempo de cubrimiento* C_v a el primer momento en que la partícula visitó todos los vértices de la gráfica empezando en v . Claramente el tiempo esperado de ejecución del algoritmo por árbol es igual a $\mathbb{E}(C_s)$. Se sabe que para cada gráfica conexa $\mathbb{E}(C_v) = O(n^3)$, sin embargo

en [BK89] se prueba que si la matriz de transición de una caminata aleatoria tiene el segundo eigenvalor más grande acotado lejos de 1, entonces el tiempo de cubrimiento esperado es sólo $O(n \cdot \log(n))$. Esta condición la satisfacen la mayoría de las gráficas en el modelo $\mathbb{G}(n, p)$ para cada $p > \frac{c \log(n)}{n}$ en particular $p = \frac{1}{2}$ y para casi todas las gráficas d-regulares [BS87], [JFS89]. En la figura 6 se muestran los resultados del tiempo de ejecución para el algoritmo implementado en Python.

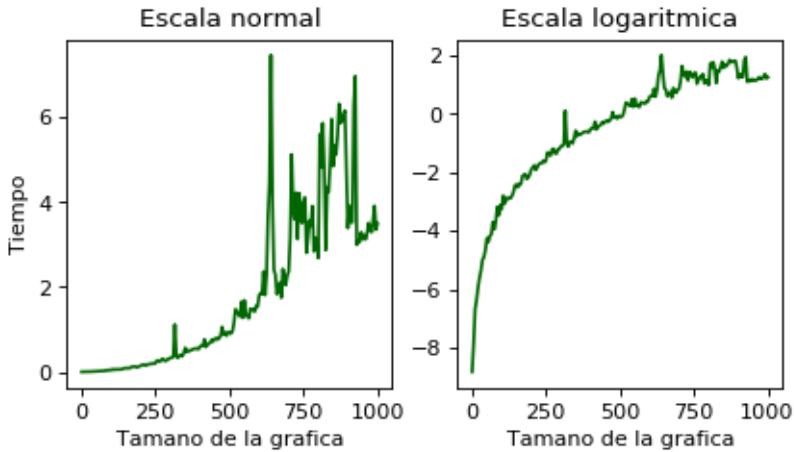


Figura 6: Tiempos de ejecución en segundos del algoritmo generate variando el tamaño del árbol. Se presenta la escala norma y la logarítmica

Referencias

- [AC15] M. Farber A. Costa. Large random simplicial complexes, ii; the fundamental groups. 2015.
- [BK89] A.Z. Broder and A.R. Karlin. Bounds on cover times. *Journal of Theoretical Probability*, pages 101—120, 1989.
- [Bro89] A. Z. Broder. Generating random spanning trees. *Journal of Theoretical Probability*, pages 101—120, 1989.
- [BS87] A.Z. Broder and E. Shamir. On the second eigenvalue of random regular graphs. *FOCS*, pages 286–294, 1987.
- [CCM88] R.P.J. Day C.J. Colbourn and W. Myrvold. Generating random spanning trees efficiently. 1988.
- [CCN88] R.P.J. Day C.J. Colbourn and L.D. Nel. Unranking and ranking spanning trees of a graph. *J. Algorithms*, 1988.
- [JFS89] J. Kahn J. Friedman and E. Szemerédi. On the second eigenvalue of random regular graphs. pages 587–598, 1989.
- [LA13] T. Luczak R. Meshulam L. Aronshtam, N. Linial. Collapsibility and vanishing of top homology in random simplicial complexes. *Discrete Computational Geometry*, pages 317—334, 2013.
- [VA59] Daniel Borrajo Manuela Veloso Vidal Alcázar, Susana Fernandez. On random graphs i. *Publicationes Mathematicae (Debrecen)*, pages 290–297, 1959.
- [VA15] Daniel Borrajo Manuela Veloso Vidal Alcázar, Susana Fernandez. Using random sampling trees for automated planning. *AI Communications*, pages 665–681, 2015.