



UNIVERSIDAD DE GRANADA

SIMULACIÓN DE SISTEMAS
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

MODELOS DE MONTE CARLO. GENERADORES DE DATOS

Autor

José María Sánchez Guerrero

Rama

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2019-2020

Índice general

| | |
|---|----------|
| 1. Mi segundo modelo de simulación de MonteCarlo | 2 |
| 1.1. Experimentación inicial | 2 |
| 1.2. Modificaciones del modelo | 6 |
| 1.2.1. Cantidad fija de devolución | 6 |
| 1.2.2. Cantidad relativa de devolución | 7 |
| 2. Generadores de datos | 9 |
| 2.1. Mejorando los generadores | 9 |
| 2.1.1. Generadores ordenados | 9 |
| 2.1.2. Implementación con búsqueda binaria | 10 |
| 2.1.3. Mejora de eficiencia constante en el generador a | 11 |
| 2.2. Generadores congruenciales | 12 |

Capítulo 1

Mi segundo modelo de simulación de MonteCarlo

Un establecimiento se abastece diariamente de un cierto producto, y necesita decidir cuántas unidades s de ese producto pedir cada día. Se desea encontrar el valor de s donde se maximice la ganancia esperada. Obtenemos una ganancia de x euros por unidad vendida, y una pérdida de y euros si no se ha vendido al final del día. También contaremos con un valor de demanda, el cual serán los productos solicitados cada día y seguirán una distribución de probabilidad \mathbf{P} con el cual jugaremos para ver los distintos resultados que nos ofrecerán.

1.1. Experimentación inicial

Empezaremos por un modelo de MonteCarlo inicial, en el cual representaremos todas las variables anteriores en el código para calcular la ganancia, y también dispondremos de varias distribuciones para calcular la demanda de productos solicitados. La ganancia viene determinada por lo siguiente:

$$g(s, x, y, d) = \begin{cases} x * s & \text{si } d \geq s \\ x * d - (s - d) * y & \text{si } d < s \end{cases} \quad (1.1)$$

Y tendremos estas tres tipos de distribuciones que podremos seleccionar en el código la que queramos. Hay que tener en cuenta que la simulación será para 100 valores de s , por lo que esto influirá también en las distribuciones:

- $P(D = d)$ se distribuye uniformemente entre 0 y 99

- $P(D = d)$ es proporcional a $100 - d$, $\forall d = 0, 1, 2, \dots, 99$
- $P(D = d)$ tiene una distribución "triangular" que crece con $d/2500$ entre 0 y 50; y decrece con $(100 - d)/2500$ entre 50 y 99.

Por último, como vamos a ejecutar un modelo varias **veces**, tendremos que obtener la media para cada s y también la desviación típica con la fórmula

$$desviaciont = \sqrt{\frac{sum2 - veces * gananciaesperada * gananciaesperada}{veces - 1}} \quad (1.2)$$

El código de este ejercicio está en el archivo *generadores.c*, proporcionado junto a la memoria. Vamos a ejecutarlo con distintos parámetros para ver cómo afectan al modelo. La ejecución la vamos a dividir en tres: una para $x = 10, y = 1$; otra para $x = 10, y = 5$; y la última para $x = 10, y = 10$. Los valores de las tablas serán tanto la ganancia como la desviación típica para los distintos valores de *veces* y contrastaremos los resultados.

Estos han sido los resultados para la distribución **uniforme**:

| Veces | s | Ganancia | Desviación | Veces | s | Ganancia | Desviación |
|---------------|----|----------|------------|---------------|----|----------|------------|
| 100 | 86 | 495.68 | 316.09 | 100 | 77 | 390.80 | 388.73 |
| 1000 | 91 | 472.38 | 308.29 | 1000 | 70 | 347.14 | 348.71 |
| 5000 | 90 | 457.66 | 310.85 | 5000 | 67 | 335.88 | 332.22 |
| 10000 | 91 | 454.09 | 311.94 | 10000 | 65 | 333.65 | 323.97 |
| 100000 | 87 | 450.85 | 304.69 | 100000 | 66 | 330.05 | 331.99 |

Cuadro 1.1: $x = 10, y = 1$

Cuadro 1.2: $x = 10, y = 5$

| Veces | s | Ganancia | Desviación |
|---------------|----|----------|------------|
| 100 | 50 | 290.60 | 314.91 |
| 1000 | 55 | 268.84 | 356.97 |
| 5000 | 48 | 254.50 | 306.41 |
| 10000 | 53 | 249.94 | 345.77 |
| 100000 | 51 | 247.03 | 333.85 |

Cuadro 1.3: $x = 10, y = 10$

A simple vista se puede observar como los resultados se van ajustando cada vez mejor a medida que aumentamos el número de repeticiones. Es lo lógico, ya que cuantas más veces se pruebe el modelo, más real será la media entre todos los valores (teniendo en cuenta que se utilizan valores aleatorios y siempre vamos a tener un pequeño error).

Todo lo contrario con los resultados que ofrecen las cantidades de repeticiones bajas. Es difícil fiarnos de un resultado así, y menos sabiendo que el modelo depende bastante de la aleatoriedad, y que cualquier valor fuera de lo normal hace que cambie todo de una forma más acentuada que en uno con más repeticiones.

No obstante, hay que tener en cuenta que cuantas más repeticiones le pongamos, más tardará en ejecutarse. En estos ejemplos no tenemos mucho problema en eso, ya que son bastante rápidos pese a ejecutarlo con $veces = 100000$.

Si nos fijamos ahora en como cambian los valores entre tablas, es decir, entre unos valores de y y otros, vemos como para un valor más bajo las ganancias son más altas que con un valor alto. Esto tiene sentido, porque la y representa cuanto aumentan las pérdidas si tenemos muchos productos sin vender.

En el caso de la $y = 1$ las ganancias y los productos adquiridos (s) son muy altos, ya que apenas perdemos si no vendemos una unidad; en cambio, con la $y = 10$ la ganancia baja y los productos adquiridos también, ya que no sale rentable comprarlos y después tener que desecharlos.

Vamos a probar ahora con otra distribución. Estos han sido los resultados para la distribución **proporcional**:

| Veces | s | Ganancia | Desviación | Veces | s | Ganancia | Desviación |
|--------|----|----------|------------|--------|----|----------|------------|
| 100 | 78 | 322.40 | 285.75 | 100 | 57 | 233.40 | 293.49 |
| 1000 | 69 | 297.04 | 243.19 | 1000 | 44 | 202.47 | 229.51 |
| 5000 | 71 | 289.95 | 244.48 | 5000 | 39 | 193.12 | 203.07 |
| 10000 | 72 | 287.13 | 227.74 | 10000 | 42 | 193.25 | 221.17 |
| 100000 | 67 | 283.75 | 235.86 | 100000 | 44 | 188.60 | 231.17 |

Cuadro 1.4: $x = 10$, $y = 1$ Cuadro 1.5: $x = 10$, $y = 5$

| Veces | s | Ganancia | Desviación |
|--------|----|----------|------------|
| 100 | 30 | 167.40 | 210.69 |
| 1000 | 30 | 146.60 | 201.91 |
| 5000 | 27 | 136.17 | 179.10 |
| 10000 | 27 | 135.01 | 178.65 |
| 100000 | 28 | 133.66 | 189.31 |

Cuadro 1.6: $x = 10$, $y = 10$

Vemos que el resto de valores son más bajos que en las tablas anteriores. En este caso no tenemos analíticamente el valor de s para contrastar el resultado, pero por la forma decreciente de la distribución utilizada podemos decir que estos

valores tienen sentido. Una distribución decreciente quiere decir que las demandas pequeñas tienen más posibilidades de salir que las grandes, es decir, que la gente comprará menos ese producto y no tendremos que adquirir tantos.

Por otra parte, cuantas más repeticiones ponemos, al igual que antes, más exactos salen nuestros resultados. Lo mismo sucede con los valores de y respecto a x , y es que para un valor más bajo las ganancias son más altas que con un valor alto.

Por último vamos a analizar los resultados para la distribución **triangular**:

| Veces | s | Ganancia | Desviación | Veces | s | Ganancia | Desviación |
|--------|----|----------|------------|--------|----|----------|------------|
| 100 | 83 | 493.40 | 217.66 | 100 | 74 | 432.50 | 282.03 |
| 1000 | 77 | 476.39 | 212.23 | 1000 | 62 | 408.09 | 225.36 |
| 5000 | 76 | 466.11 | 205.99 | 5000 | 59 | 389.14 | 218.78 |
| 10000 | 76 | 466.78 | 206.22 | 10000 | 58 | 388.80 | 215.32 |
| 100000 | 79 | 464.94 | 212.76 | 100000 | 59 | 386.79 | 221.56 |

Cuadro 1.7: $x = 10$, $y = 1$ Cuadro 1.8: $x = 10$, $y = 5$

| Veces | s | Ganancia | Desviación |
|--------|----|----------|------------|
| 100 | 51 | 367.00 | 225.96 |
| 1000 | 44 | 335.26 | 187.24 |
| 5000 | 53 | 336.00 | 251.46 |
| 10000 | 48 | 337.11 | 217.58 |
| 100000 | 49 | 333.64 | 228.09 |

Cuadro 1.9: $x = 10$, $y = 10$

Al igual que ha pasado con las otras dos distribuciones, nos vamos a quedar con los resultados que nos han ofrecido las repeticiones más altas, ya que son los más fiables. Más de lo mismo con la relación $x - y$, aunque en este caso vemos como esta vez los valores no están tan lejos los unos de otros.

Analizando los resultados que nos ofrece la s , vemos como los valores rondan el $n/2$ (en nuestro caso 50), porque la distribución triangular hace que los valores más probables estén por ahí. Esto también ha afectado a los valores de la ganancia, que en esta tabla hemos visto que son más altos que en los anteriores. Esto nos es muy útil, ya que si tuviésemos datos reales o una distribución real de la gente que compra los productos, podríamos ajustar un modelo MonteCarlo a los números aleatorios de esta distribución. El objetivo de esto sería obviamente ajustar las variables para obtener la ganancia más alta posible.

1.2. Modificaciones del modelo

1.2.1. Cantidad fija de devolución

En este apartado vamos a modificar el modelo construido anteriormente, de tal forma que el establecimiento pueda devolver las unidades no vendidas. De esta forma hay que pagar una cantidad fija de z euros de gastos de devolución de las unidades no vendidas, en vez de tener una pérdida de y euros por unidad. Esta cantidad no varía, a menos que la $z = 0$.

La función de ganancia ahora sería:

$$g(s, x, y, d) = \begin{cases} x * s & \text{si } d \geq s \\ x * d - z & \text{si } d < s \end{cases} \quad (1.3)$$

Estará implementada en el código *generadoresModificados.c* y las pruebas que vamos a realizar van a ser con el número de repeticiones más alto, que ya hemos visto que es el más fiable (100000 veces) y con los tres tipos de distribuciones. El resultado ha sido el siguiente:

| z | Distribución | s | Ganancia | Desviación |
|------------|---------------------|----------|-----------------|-------------------|
| 5 | uniforme | 95 | 490.45 | 297.18 |
| 5 | proporcional | 91 | 326.41 | 236.09 |
| 5 | triangular | 95 | 495.37 | 203.63 |
| 400 | uniforme | 59 | 178.91 | 367.37 |
| 400 | proporcional | 20 | 18.58 | 246.17 |
| 400 | triangular | 44 | 233.82 | 273.39 |
| 200 | uniforme | 79 | 318.07 | 317.13 |
| 200 | proporcional | 61 | 142.47 | 255.30 |
| 200 | triangular | 55 | 323.75 | 214.21 |

Cuadro 1.10: Resultados para todas las distribuciones con $x = 10$ y $veces = 100000$

Lo hemos ejecutado con un valor de $y = 5$ ya que no era ni tan bajo que apenas supusiese pérdidas, ni tan alto que no diese demasiadas. Puede ser que no sea el valor más adecuado, pero para lo que queremos mostrar, nos servirá.

En la ejecución vemos cómo las distribuciones se adecuan bastante bien a lo que comentamos en el punto anterior sobre ellas. Por otro lado, vamos a analizar los resultados que nos ofrece cada z , que es realmente lo interesante en este apartado. Vemos que los valores bajos, como puede ser el 5, hace que la cantidad de productos pedidos sea muy alta. Esto se debe a que la pérdidas por cada producto no vendido van a ser muy pocas.

Por otro lado, con los valores altos, como puede ser el 400, los valores ya empiezan a ser mucho más bajos, tanto en la cantidad de productos como en las ganancias que obtenemos. Esto es porque, pese a que ajustemos las ventas al máximo, con que nos sobren muy pocos productos ya estaremos pagando un precio muy alto (razón por la cual las ganancias también disminuyen).

Estos valores son poco realista, porque ni una empresa te va a cobrar una excesiva cantidad de dinero por devolver el producto, ni te va a cobrar tan poco que no les salga rentable a ellos. Por eso, ahora vamos a probar con un valor que se puede ajustar mejor, como puede ser el 200. Podemos ver que los valores de s no están mal ajustados, dependiendo del tipo de distribución, y que las ganancias están bastante bien (claramente son más altas las que nos ofrecen los z pequeños, pero no sería un valor realista).

1.2.2. Cantidad relativa de devolución

Por último, vamos a 'fusionar' los dos últimos casos en uno. Si el valor z es relativamente grande, no interesará pagar esa cantidad de dinero cuando queden pocas unidades sin vender. Por otro lado, cuando el número de unidades no vendidas sea pequeño, es preferible asumir la pérdidas de y que tener que pagar los gastos de devolución.

La función de la ganancia se nos quedaría de la siguiente forma:

$$g(s, x, y, d) = \begin{cases} x * s & \text{si } d \geq s \\ x * d - \min\{z, (s - d) * y\} & \text{si } d < s \end{cases} \quad (1.4)$$

También estará implementada en *generadoresModificados.c*, y para ejecutarla, tendremos que cambiar el parámetro *modificacion* = 2. El resto de valores, se mantendrán exactamente iguales que en las ejecuciones anterior. Los resultados han sido los siguientes:

| z | Distribución | s | Ganancia | Desviación |
|------------|---------------------|----------|-----------------|-------------------|
| 100 | uniforme | 92 | 411.21 | 309.86 |
| 100 | proporcional | 72 | 238.44 | 247.52 |
| 100 | triangular | 74 | 415.98 | 217.21 |
| 150 | uniforme | 86 | 380.06 | 327.23 |
| 150 | proporcional | 56 | 206.16 | 249.46 |
| 150 | triangular | 61 | 396.93 | 211.83 |
| 200 | uniforme | 81 | 356.57 | 317.13 |
| 200 | proporcional | 43 | 189.64 | 225.30 |
| 200 | triangular | 60 | 390.21 | 219.63 |

Cuadro 1.11: Resultados para todas las distribuciones con $x = 10$, $y = 5$ y $veces = 100000$

En esta ejecución podemos ver como para un valor intermedio de y cómo cambian los resultados. Estos se modifican siguiendo las mismas reglas comentadas anteriormente, pero en este caso vemos aumentadas las ganancias debido a que elegimos cual nos conviene más, si pagar una cantidad fija o variable.

| y | Distribución | s | Ganancia | Desviación |
|-----------|---------------------|----------|-----------------|-------------------|
| 3 | uniforme | 79 | 381.69 | 334.20 |
| 3 | proporcional | 54 | 225.98 | 240.16 |
| 3 | triangular | 64 | 419.09 | 210.75 |
| 7 | uniforme | 83 | 344.55 | 337.87 |
| 7 | proporcional | 51 | 172.44 | 258.79 |
| 7 | triangular | 59 | 373.18 | 225.49 |
| 10 | uniforme | 81 | 336.49 | 335.62 |
| 10 | proporcional | 53 | 162.06 | 260.66 |
| 10 | triangular | 57 | 358.21 | 222.97 |

Cuadro 1.12: Resultados para todas las distribuciones con $x = 10$, $z = 200$ y $veces = 100000$

En este caso hemos elegido un $z = 200$ para ver como cambian las tablas en función de la y . Como podemos ver, los modelos de MonteCarlo son bastante flexibles y gracias a ellos podemos ajustar las variables a las que más nos convengan o las que más se adecuen a nuestra situación. En estos casos, quizás el gerente de la tienda no pueda controlar ni las pérdidas que le da cada producto o los precios de devolución (y y z respectivamente), pero sí puede controlar la cantidad de productos que adquiera y adecuarla a las distintas situaciones que se pueden dar, y que en definitiva, más ganancia le den.

Capítulo 2

Generadores de datos

En esta otra parte de la práctica vamos a intentar mejorar los generadores de datos, ya que es una parte fundamental a la hora de ejecutar nuestros modelos. Un generador lo más aleatorio posible, que se adecue a las distintas situaciones que nos propongan los modelos y que además sea eficiente, sería el ideal, y es lo que vamos a buscar en este capítulo. El código se encontrará en el archivo 'generadoresMejorados.c', donde habrá distintas funciones alternativas que muestren los cambios realizados.

2.1. Mejorando los generadores

2.1.1. Generadores ordenados

Vamos a intentar mejorar la eficiencia de los generadores de datos empleados en el capítulo anterior. La primera mejora que vamos a aplicar va a ser la ordenación de las tablas de forma decreciente según la probabilidad. Como los generadores a y b (uniforme y proporcional respectivamente), vamos a centrarnos en el generador c o triangular. La ejecución ha sido probada con 1000000 de repeticiones, para que se vea mejor cómo afecta y los resultados han sido los siguientes:

```
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
g++ generadoresMejorados.c -o generadores  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
./generadores  
  
Tiempo en generar demanda sin mejora: 0.191075 segundos  
Mejora 1 aplicada para la tabla c  
Tiempo en generar demanda con mejora: 0.103206 segundos
```

Figura 2.1: Ejecución con el tiempo sin y con mejora del generador c

Vemos que se reduce el tiempo a casi la mitad. Esto se debe a que, al estar ordenado de más probabilidad a menos, los valores que más van a salir serán encontrados antes en la tabla. Pese a esto, sigue siendo una tabla y la búsqueda de ellos va a ser lineal, es decir, va a recorrer los valores uno a uno hasta llegar al correcto.

2.1.2. Implementación con búsqueda binaria

En este apartado modificaremos todos los generadores. Como hemos dicho anteriormente, la búsqueda lineal era un problema, así que en este caso vamos a implementar una búsqueda binaria que solucione el problema. En el código, se cambiará la función *genera_demanda* por *genera_demanda_binaria*. Este es el resultado que nos ofrece:

```
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
g++ generadoresMejorados.c -o generadores  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
./generadores  
  
Tiempo en generar demanda sin mejora: 0.184364 segundos  
Mejora 2 aplicada para la tabla a  
Tiempo en generar demanda con mejora: 0.090404 segundos  
  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
g++ generadoresMejorados.c -o generadores  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
./generadores  
  
Tiempo en generar demanda sin mejora: 0.149532 segundos  
Mejora 2 aplicada para la tabla b  
Tiempo en generar demanda con mejora: 0.104886 segundos  
  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
g++ generadoresMejorados.c -o generadores  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
./generadores  
  
Tiempo en generar demanda sin mejora: 0.187247 segundos  
Mejora 2 aplicada para la tabla c  
Tiempo en generar demanda con mejora: 0.090907 segundos  
  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$
```

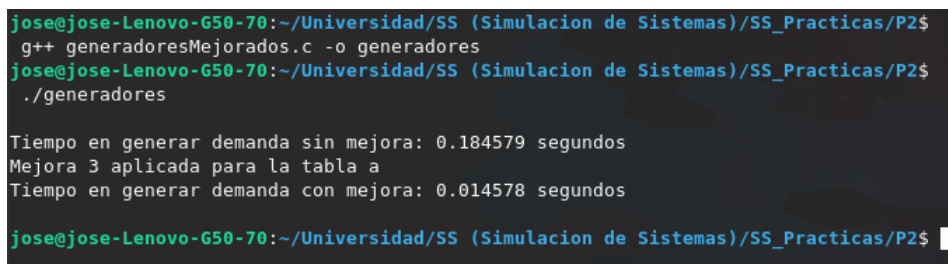
Figura 2.2: Ejecución con los tiempos sin y con mejora de búsqueda

Podemos ver que en el generador *c* los resultados son todavía mejores que antes, mientras que en los otros dos generadores también ha dado unos buenos resultados. Esto se debe a que hemos pasado de hacer una búsqueda de orden lineal a una búsqueda de orden logarítmico. La mejora se ha notado sobre todo en el primer generador, y en el último, donde ha bajado el tiempo casi un 50 %; mientras que en el generador *b* ha sido un poco menos.

2.1.3. Mejora de eficiencia constante en el generador *a*

Por último, vamos a probar a mejorar la eficiencia del generador uniforme haciendo que su tiempo de ejecución sea constante. Este generador lo que hace es generar un número aleatorio y recorrer la tabla hasta encontrar un valor mayor que este, y devolver el índice donde estaba situado. Nos podemos ahorrar este proceso si, tras generar el número aleatorio entre 0 y 1, lo multiplicamos por 100 y nos quedamos con la parte entera.

Los resultados obtenidos tras implementarlo han sido:



```
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
g++ generadoresMejorados.c -o generadores  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$  
./generadores  
  
Tiempo en generar demanda sin mejora: 0.184579 segundos  
Mejora 3 aplicada para la tabla a  
Tiempo en generar demanda con mejora: 0.014578 segundos  
  
jose@jose-Lenovo-G50-70:~/Universidad/SS (Simulacion de Sistemas)/SS_Practicas/P2$
```

Figura 2.3: Ejecución con los tiempos sin y con mejora del generador *a*

Vemos que la mejora ha sido muy grande, mucho mejor que las anteriores. Lo bueno de este método es que, por muchos elementos que tengamos, la eficiencia va a ser la misma, porque estamos ante una 'búsqueda' de orden constante. En contra de este método, decir que funciona porque el generador original en uniforme, pero a la hora de hacerlo para los generadores nos resultará imposible encontrar algo similar.

2.2. Generadores congruenciales

En esta sección vamos a analizar los generadores básicos y ver el peligro que conlleva no implementarlos de forma correcta. El ejercicio nos pide implementar dos generadores congruenciales de la siguiente forma:

$$x_{n+1} = (2061x_n + 4321) \text{ mód } m \quad (2.1)$$

Y el otro muy similar:

$$x_{n+1} = (2060x_n + 4321) \text{ mód } m \quad (2.2)$$

El código donde está implementados se llama *generadoresCongruencias.c* y para cambiar entre uno y otro sólo tendremos que cambiar la variable $a = 2060$ o $a = 2061$. El ejercicio también nos pide que se calcule el número de periodos obtenidos con distintos tipos de aritméticas: entera, real 'artesanal', real 'artesanal' corregida y real usando *fmod*. Para cambiar de aritmética sólo tendremos que cambiar la variable *aritmética* en el código.

También tendremos que seleccionar un valor de $m = 10^4$ y un valor inicial $x_0 = 14$ en mi caso. El resultado de las ejecuciones ha sido el siguiente:

| Aritmética | Periodo gen 1 | Periodo gen 2 |
|----------------------|---------------|---------------|
| entera | 10000 | 4 |
| real artesanal | 152 | 8 |
| artesanal corregida | 10000 | 4 |
| real con <i>fmod</i> | 10000 | 4 |

Cuadro 2.1: $x = 10, y = 10$

Podemos ver cómo claramente el generador 2 (cuya $a = 2060$) no es un buen generador de números aleatorios, ya que empezará a repetir valores muy temprano, por otro lado, el generador 1 es un generador correcto, ya que llega a los 10^4 periodos sin repetir ni un número. Esto se debe a que a y m no tienen divisores en común, mientras que con el generador 2 no sucede lo mismo.

Si nos fijamos en la aritmética real artesanal, vemos como termina muy pronto, en comparación al resto de aritméticas del generador 1. La razón por la que sucede esto es que no se redondea correctamente el valor obtenido, es decir, si yo obtengo el 0.99, esta aritmética lo tomará como 0, estando claramente este valor más cerca del 1.

Podemos decir que el resto de aritméticas están bien hechas, ya que ninguna de ellas comenten estos errores y consiguen alcanzar el periodo máximo.