N-ésimo Primo

Primer millón de números primos calculados con una fórmula para el n-ésimo primo

Horacio Useche Losada
Google Software Developer
horaciouseche@gmail.com

Diciembre de 2018

Diagramación en LaTeX realizada por el autor bajo Linux Fedora

Contents

1 Introducción

Conseguir una fórmula, un procedimiento o algoritmo para computar el n-ésimo primo, ha sido siempre un viejo anhelo de los matemáticos. Sin embargo, en la literatura científica solo se reportan fórmulas basadas en el teorema de Wilson, las cuales, carecen de un valor práctico y solo pueden tener un interés estrictamente teórico, ya que no se puede llegar muy lejos al intentar su uso en cálculos concretos.

Esta investigación retoma un trabajo del profesor Ramón Fandiño,[1] el cual, presenta en 1980 una relación funcional a partir de la cual se puede computar el n-ésimo primo en función de los $n-1$ primos anteriores. Para conseguir el objetivo, el profesor Fandiño realiza cinco ajustes, tres por mínimos cuadrados y dos por técnicas implementadas por él mismo, con lo cual consigue calcular los primeros 5000 primos.

Siguiendo la línea de investigación del citado profesor, pero haciendo algunos cambios importantes en el modelo matemático usado y con un menor número de ajustes, he conseguido computar **un millón de números primos**, advirtiendo que es posible computar muchos más,[2] si se cuenta con las herramientas de hardware adecuadas. En esta ocasión, he usado un PC casero[3], una máquina corriente que logró computar dicha cantidad en tan solo una hora y 21 minutos! Para hacernos una idea del esfuerzo computacional, en su momento el profesor Fandiño utilizó, no un PC, sino un computador de verdad, un IBM 360/44 que era la máquina más poderosa del centro de cómputo de la UN (y posiblemente de Colombia).[4]

Con un "juguete"de cómputo, me complace presentar esta cifra que se enmarca en una política denominada "resultados sorprendentes con recursos mediocres"tal y como acontece con otros trabajos de este autor (ver [5], [6], y [7]). Espero muy pronto superar esta cifra usando un hardware más poderoso, naturalmente.

[1]Profesor titular de la Universidad Nacional de Colombia.

[2]Con un hardware poderoso podrían ser billones!

[3]PC equipado con un procesador Intel Core 2 Quad CPU Q6600 @ 2.4 Ghz, con 4 GB de RAM y corriendo bajo linux Fedora 28 de 64 bits.

[4]El cálculo fue realizado en el Centro de Cómputo de la Universidad Nacional con software escrito por los Drs. Gloria Inés Giraldo y Jaime Herrera Bernal (ver [1]).

2 Antecedentes

Desde los tiempos de Euclides, los matemáticos han buscado una fórmula, procedimiento, o algoritmo que, dado un número entero n, entregue el n-ésimo primo, es decir, el número primo número n. No obstante, a pesar de los históricos esfuerzos, la literatura científica internacional solo denuncia algunas fórmulas basadas en el teorema de Wilson, como por ejemplo la fórmula dada en [9]:

$$f(n) = \left\lfloor \frac{n! \bmod (n+1)}{n} \right\rfloor (n-1) + 2, \quad \text{para n, un entero positivo.} \qquad (1)$$

Por el teorema de Wilson, $n+1$ es primo, si y solamente si, $n!$ mod $(n+1) = n$. Así, cuando $n+1$ es primo, el primer factor en el producto de la ecuación 1 se hace igual a la unidad y la fórmula produce el primo $n+1$. Sin embargo, cuando $n+1$ es un compuesto, el primer factor se anula y la formula anterior produce el primo 2.

La fórmula dada en la ecuación 1 no es práctica porque, aunque teóricamente cumple, evaluar el teorema de Wilson para valores grandes de n no es posible. Aún para valores modestos de n resulta muy duro evaluar el teorema de Wilson en un computador personal como el usado aquí para la presente investigación. En resumen, no se puede llegar muy lejos usando este tipo de expresiones para el n-ésimo primo.

Otra expresión similar, también basada en el teorema de Wilson, se puede consultar en [10]:

$$p_n = n(1 - \frac{\sin^2(\pi \frac{(n-1)!+1}{n})}{\sin^2 \frac{\pi}{n}})$$

Esta expresión hace que el segundo factor en la ecuación, resulte unitario cuando n es primo y se anule en cualquier otro caso. Por tanto produce el n-ésimo primo. Sin embargo, tiene el mismo problema que la expresión 1, es decir, es impráctica, dado que también se basa en el teorema de Wilson. Usando el teorema de Wilson es fácil construir funciones de este tipo, pero todas son, igualmente, de poco valor práctico.

Existen también algunos polinomios que entregan una secuencia de números primos, como por ejemplo, $n^2 + n + 41$, el cual produce un primo para $n = 1, 2, 3, \ldots 39$, es decir, entrega números primos para todo entero menor a 40. Asimismo, otros polinomios más complejos entregan números primos para ciertos valores de n, pero su grado de complejidad es tal que resultan de poco valor práctico como en los casos anteriormente citados.

3 La estrategia de Fandiño

En 1980, el profesor de matemáticas de la Universidad Nacional, Ramón Fandiño,[5] publicó un artículo (ver [1]) en el cual desarrollaba un procedimiento para calcular la secuencia completa de los números primos. En efecto, el trabajo del profesor Fandiño, consiste en una relación funcional implícita entre números primos. Dicha relación es de la forma:

$$F(p_1, p_2, p_3, \ldots, p_{k-1}, p_k) = 0$$

donde $p_k = f(p_1, p_2, p_3, \ldots, p_{k-1})$, siendo p_k el k-ésimo número primo.

La figura 1 representa el n-ésimo número primo, donde a cada entero en el eje X le asociamos un único primo en el eje Y, en principio y detallando la grafica, parece viable realizar un ajuste por mínimos cuadrados para "ajustar una curva exponencial", con la dispersión de puntos correspondientes a la secuencia de primos, es decir, $(1, p_1), (2, p_2), (3, p_3), \ldots (k, p_k)$.

El trabajo de Fandiño presenta una serie de cinco ajustes sucesivos para conseguir finalmente, una relación funcional que nos entrega, efectivamente, el n-ésimo primo.

El trabajo presentado en [1] entregó los primeros 5000 números primos, lo cual muestra que tiene un valor práctico auténtico, ya que lo podemos usar para calcular una cantidad superior de números primos, sin la dependencia con funciones factoriales que dificulten al extremo dicho cálculo.

El profesor Fandiño sigue el modelo exponencial $y = ab^x$, donde $x = 1, 2, \ldots, N$ y $y = p_1, p_2, p_3, \ldots, p_N$. En dicho modelo se trata encontrar los parámetros a y b en términos de $p_1, p_2, p_3, \ldots, p_N$ y N, que mejor se ajustan a la curva $y = ab^x$.

[5] Profesor titular del Departamento de Matemáticas de la Universidad Nacional de Colombia y quién, dicho sea de paso, resultó ser mi jurado de tesis en la Facultad de Ciencias de la misma universidad.

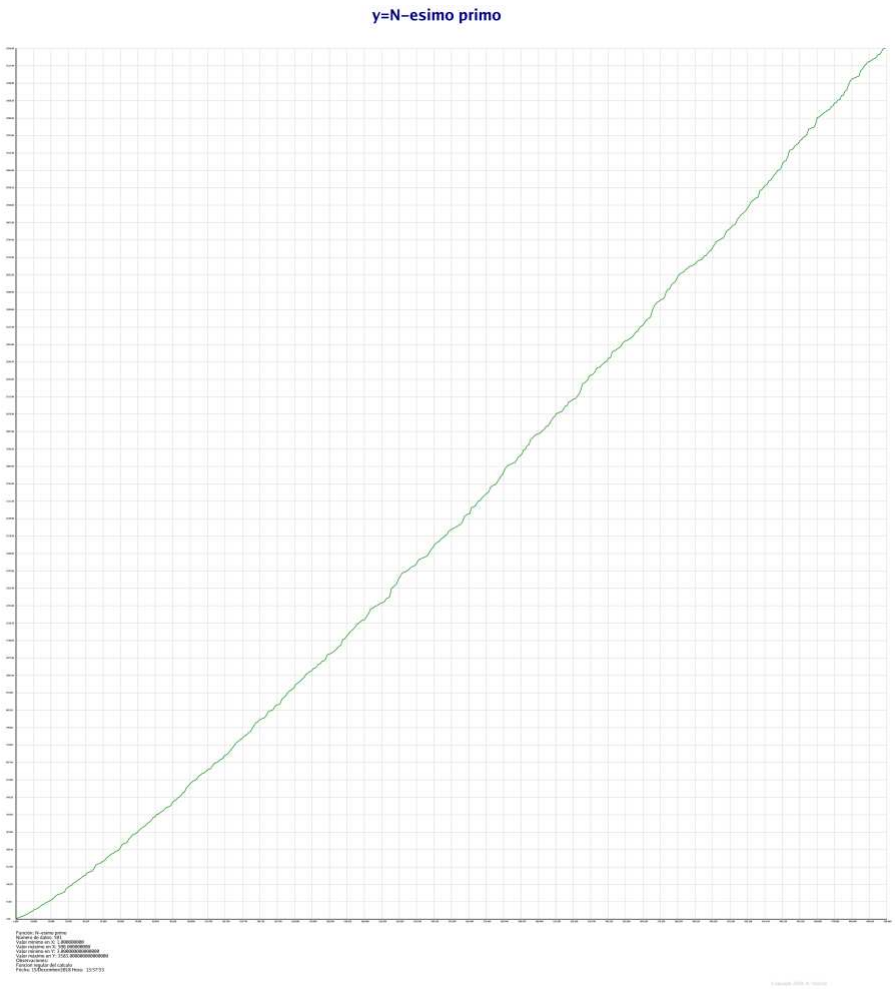

Figure 1: El n-ésimo número primo. Imagen creada por H. Useche

4 La nueva estrategia

En esta investigación, retomamos la estrategia del profesor Fandiño pero con algunos cambios importantes, a saber:

- El modelo de ajuste se basa en la curva $y = mx + b$. Es decir, es un modelo lineal, más que el exponencial usado por Fandiño.
- Sólo se necesitaron cuatro (4) ajustes para p_k, a diferencia de los cinco

(5) necesarios en el modelo de Fandiño, es decir, el camino es más corto por esta vía.

5 El modelo lineal P_k

Este modelo propone simular la curva dada en la figura 1 a través de la expresión lineal

$$y = mx + b \tag{2}$$

donde y representa la secuencia de números primos, x la secuencia de enteros naturales positivos y a y b son parámetros por calcular mediante la técnica de ajuste por mínimos cuadrados.

El modelo anterior se ajusta bien mediante los parámetros calculados por:

$$m = \frac{n\sum(x \cdot y) - \sum x \sum y}{n\sum x^2 - |\sum x|^2} \tag{3}$$

$$b = \frac{\sum y \sum x^2 - \sum x \sum(x \cdot y)}{n\sum x^2 - |\sum x|^2} \tag{4}$$

Ahora, puesto que estamos hablando de calcular el n-ésimo primo, p_n, a partir de los $n-1$ primos anteriores, la ecuación (2) se transforman en:

$$p_n = m_n \cdot n + b_n \tag{5}$$

y los parámetros en (3) y (4), se convierten en:

$$m_n = \frac{n\sum(k \cdot p_k) - \sum k \sum p_k}{\frac{n^2(n+1)(2n+1)}{6} - \frac{n^2(n+1)^2}{4}} \tag{6}$$

$$b_n = \frac{\sum p_k \sum k^2 - \sum k \sum(k \cdot p_k)}{\frac{n^2(n+1)(2n+1)}{6} - \frac{n^2(n+1)^2}{4}} \tag{7}$$

y se debe tener en cuenta que el índice k en todas las sumatorias recorre desde $k = 1$ hasta $k = n-1$. Así, $\sum p_k$ se toma como:

$$\sum_{k=1}^{n-1} p_k = p_1 + p_2 + p_3 + \cdots + p_{n-1}$$

es decir, recorre hasta el primo $n-1$. También, nótese que en las ecuaciones (6) y (7), hemos hecho:

$$\sum x = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$
$$\sum x^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

6 Primer ajuste $P_{1,k}$

Con la teoría descrita en la sección 5, procedemos a realizar el primer ajuste. Para cumplir este objetivo hemos escrito una primera rutina en C++/GMP, la cual tiene el siguiente aspecto:

```
mpf_class AjusteMC(unsigned long int uIters)
{
 // calcula el ajuste por minimos cuadrados
 int i,iPrec=100;
 mpz_class q;
 mpf_class a(0.0,iPrec);
 mpf_class b(0.0,iPrec);
 mpf_class c(0.0,iPrec);
 mpf_class d(0.0,iPrec);
 mpf_class e(0.0,iPrec);
 mpf_class f(0.0,iPrec);
 mpf_class p(0.0,iPrec);
 mpf_class n(0.0,iPrec);
 mpf_class m(0.0,iPrec);
 mpf_class s1(0.0,iPrec);
 mpf_class s2(0.0,iPrec);
 mpf_class s3(0.0,iPrec);
 mpf_class s4(0.0,iPrec);
 q=2;n=uIters;
 for(i=1;i<uIters;i++)
 {
  p=NextPrime(q);
  s1=s1+i*p;
  s2=s2+i;
```

```
  s3=s3+p;
  s4=s4+i*i;
  q=p;
 }
 c=(n*n*(n+1)*(2*n+1))/6.0;
 d=(n*(n+1))/2.0;
 e=d*d;
 f=c-e; // denominador
 c=n*s1;
 d=s2*s3;
 e=c-d; // numerador
 m=e/f;
 a=s3*s4;
 c=s2*s1;
 b=(a-c)/f;
 p=m*n+b;
 return p;
}
```

Para correr esta rutina empleamos una segunda función, la cual cita a la función `AjusteMC()` y que hemos denominado `RunAjusteMC()` cuyo aspecto es similar a:

```
void RunAjusteMC(unsigned long int uIters)
{
 // recorre el ajuste MC
 int i,iPrec=100;
 string str,s1;
 mpf_class p(0.0,iPrec);
 mpf_class d(0.0,iPrec);
 mpz_class q;
 mpz_class pv;
 // reserva memoria
 D1=new mpf_class[uIters+1];
 P1=new mpf_class[uIters+1];
 q=5;
 for(i=2;i<uIters;i++)
 {
  pv=NextPrime(q);
```

```
  p=AjusteMC(i);
  d=q-p;
  D1[i-2]=d;
  P1[i-2]=p;
  cout<<i<<" : "<<q<<" : ";
  gmp_printf("%.5Ff : %.3Ff\n",p,d);
  q=pv;
}
```

Y para concluir hemos empleado una función `main()` que cita a la función `RunAjusteMC` con la siguiente apariencia:

```
#include <stdio.h>
#include <iostream>
#include <gmpxx.h>
#include <time.h>

/*
 FUNCIONES AQUI
*/
int main()
{
 time_t tm1,tm2;
 (void)time(&tm1);
 int iIters=52;
 SetPrecision(10*iIters);
 printf("\n N-esimo primo ...\n");
 RunAjusteMC(iIters);
 (void)time(&tm2);
 cout<<"Begin time: "<<ctime(&tm1)<<"\n";
 cout<<"End   time: "<<ctime(&tm2)<<"\n";
 return 0;
}
```

Ejecutando este software para computar los primeros 50 primos en el primer ajuste, se obtiene la siguiente salida:

```
N-esimo primo ...
2 : 5 : 6.00000 : -1.000
```

```
3 : 7 : 7.66667 : -0.667
4 : 11 : 9.50000 : 1.500
5 : 13 : 13.00000 : 0.000
6 : 17 : 15.60000 : 1.400
7 : 19 : 19.00000 : 0.000
8 : 23 : 21.75000 : 1.250
9 : 29 : 25.05556 : 3.944
10 : 31 : 29.46667 : 1.533
11 : 37 : 33.07273 : 3.927
12 : 41 : 37.50000 : 3.500
13 : 43 : 41.84615 : 1.154
14 : 47 : 45.56044 : 1.440
15 : 53 : 49.36190 : 3.638
16 : 59 : 53.72500 : 5.275
17 : 61 : 58.50000 : 2.500
18 : 67 : 62.69281 : 4.307
19 : 71 : 67.28070 : 3.719
20 : 73 : 71.77368 : 1.226
21 : 79 : 75.80952 : 3.190
22 : 83 : 80.20779 : 2.792
23 : 89 : 84.54941 : 4.451
24 : 97 : 89.17754 : 7.822
25 : 101 : 94.36000 : 6.640
26 : 103 : 99.38462 : 3.615
27 : 107 : 103.98006 : 3.020
28 : 109 : 108.50000 : 0.500
29 : 113 : 112.67980 : 0.320
30 : 127 : 116.83678 : 10.163
31 : 131 : 122.26452 : 8.735
32 : 137 : 127.53427 : 9.466
33 : 139 : 132.90909 : 6.091
34 : 149 : 137.90374 : 11.096
35 : 151 : 143.48067 : 7.519
36 : 157 : 148.67778 : 8.322
37 : 163 : 153.97297 : 9.027
38 : 167 : 159.35420 : 7.646
39 : 173 : 164.60594 : 8.394
40 : 179 : 169.94231 : 9.058
```

```
41 : 181 : 175.35366 : 5.646
42 : 191 : 180.45064 : 10.549
43 : 193 : 186.00997 : 6.990
44 : 197 : 191.25687 : 5.743
45 : 199 : 196.40000 : 2.600
46 : 211 : 201.27536 : 9.725
47 : 223 : 206.75948 : 16.241
48 : 227 : 212.79521 : 14.205
49 : 229 : 218.67857 : 10.321
50 : 233 : 224.26286 : 8.737
51 : 239 : 229.73098 : 9.269
```

Por comodidad, hemos recorrido sobre los primeros 52 primos impares, aunque la salida se muestra a partir del primo $p_2 = 5$, omitiendo $p_1 = 3$. Nótese que contamos solo los primos impares, esto, desde luego, no afecta los cálculos, el primo par $p_0 = 2$ si se tiene en cuenta en el cómputo como se puede apreciar en el código fuente de la rutina `AjusteMC()`. La salida del programa muestra, de izquierda a derecha:

1. Columna 1: Corresponde a la numeración de los primos.
2. Columna 2: Muestra el valor real del primo que se está aproximando mediante el ajuste por mínimos cuadrados.
3. Columna 3: Muestra el n-ésimo primo aproximado por el ajuste.
4. Columna 4: Muestra las diferencias entre el primo aproximado y el valor real del mismo, esto es, $\delta_{1,k} = p_{1,k} - p_k$.

Vale la pena comparar los resultados obtenidos en este primer ajuste con la técnica expuesta, contra los resultados obtenidos en [1] bajo el modelo exponencial. Se observa fácilmente que estos nuevos resultados son, de lejos, una mejor aproximación al valor de p_k para el primer ajuste.

A pesar de que el resultado de arriba es mejor que el obtenido en [1], todavía está lejos de aproximar a p_k de forma aceptable. Por tanto, procedemos a realizar un segundo ajuste por mínimos cuadrados.

7 Segundo ajuste $P_{2,k}$

Para mejorar la aproximación de p_k, procedemos a realizar un segundo ajuste por mínimos cuadrados, pero esta vez, se toman las $k-1$ diferencias $\delta_{1,k-1}$

y sobre ellas se realiza un ajuste pero, como se comentó, en lugar de utilizar los valores para los $k-1$ primos anteriores, usamos la misma cantidad de diferencias, con lo cual obtenemos el valor de $\delta_{1,k}$ a partir de las $\delta_{1,k-1}$ diferencias.

El código C++/GMP para esta función es como sigue:

```
mpf_class AjusteDif(unsigned long int uIters,int iOption)
{
 // calcula el ajuste por minimos cuadrados
 int i,iPrec=100;
 mpz_class q;
 mpf_class a(0.0,iPrec);
 mpf_class b(0.0,iPrec);
 mpf_class c(0.0,iPrec);
 mpf_class d(0.0,iPrec);
 mpf_class e(0.0,iPrec);
 mpf_class f(0.0,iPrec);
 mpf_class p(0.0,iPrec);
 mpf_class n(0.0,iPrec);
 mpf_class m(0.0,iPrec);
 mpf_class s1(0.0,iPrec);
 mpf_class s2(0.0,iPrec);
 mpf_class s3(0.0,iPrec);
 mpf_class s4(0.0,iPrec);
 n=uIters;
 for(i=1;i<uIters;i++)
 {
  p=D1[i-1];
  s1=s1+i*p;
  s2=s2+i;
  s3=s3+p;
  s4=s4+i*i;
 }
 c=(n*n*(n+1)*(2*n+1))/6.0;
 d=(n*(n+1))/2.0;
 e=d*d;
 f=c-e; // denominador
 c=n*s1;
```

```
 d=s2*s3;
 e=c-d; // numerador
 m=e/f;
 a=s3*s4;
 c=s2*s1;
 b=(a-c)/f;
 p=m*n+b;
 return p;
}
```

Como se puede observar fácilmente, este código es prácticamente igual al usado en el primer ajuste en la función `AjusteMC()`, solo que, en esta ocasión se ajustan las diferencias $\delta_{1,k-1}$. El resultado se suma al valor de $p_{1,k}$ para obtener una segunda aproximación del n-ésimo primo, según la expresión:

$$p_{2,k} = p_{1,k} + \delta_{1,k-1}$$

De otra parte, en la función `RunAjusteMC()` hemos agregado el siguiente fragmento de código para cumplir con el cometido propuesto:

```
// realiza el primer ajuste de las diferencias
cout<<"\n";
cout<<"AJUSTE DE PRIMERAS DIFERECNIAS:\n";
mpf_class p2(0.0,iPrec);
mpf_class d1(0.0,iPrec);
q=5;
for(i=2;i<uIters;i++)
{
 pv=NextPrime(q);
 d1=AjusteDif(i,0);
 p2=P1[i-2]+d1;
 P2[i-2]=p2;
 d=q-p2;
 cout<<i<<" : "<<q<<" : ";
 gmp_printf("%.5Ff : %.5Ff\n",p2,d);
 q=pv;
}
```

Al ejecutar este nuevo entorno de código, se obtiene el siguiente resultado:

```
AJUSTE DE PRIMERAS DIFERECNIAS:
2 : 5 : 4.00000 : 1.00000
3 : 7 : 6.44444 : 0.55556
4 : 11 : 10.66667 : 0.33333
5 : 13 : 13.71667 : -0.71667
6 : 17 : 17.02222 : -0.02222
7 : 19 : 19.95714 : -0.95714
8 : 23 : 23.04881 : -0.04881
9 : 29 : 27.74985 : 1.25015
10 : 31 : 32.09259 : -1.09259
11 : 37 : 36.50456 : 0.49544
12 : 41 : 41.34640 : -0.34640
13 : 43 : 45.25855 : -2.25855
14 : 47 : 48.71487 : -1.71487
15 : 53 : 52.89465 : 0.10535
16 : 59 : 57.95473 : 1.04527
17 : 61 : 62.62260 : -1.62260
18 : 67 : 67.12237 : -0.12237
19 : 71 : 71.82991 : -0.82991
20 : 73 : 75.91628 : -2.91628
21 : 79 : 79.98126 : -0.98126
22 : 83 : 84.32684 : -1.32684
23 : 89 : 88.90789 : 0.09211
24 : 97 : 94.29872 : 2.70128
25 : 101 : 99.94421 : 1.05579
26 : 103 : 104.89997 : -1.89997
27 : 107 : 109.34293 : -2.34293
28 : 109 : 113.36558 : -4.36558
29 : 113 : 117.07991 : -4.07991
30 : 127 : 122.13542 : 4.86458
31 : 131 : 128.17393 : 2.82607
32 : 137 : 134.07632 : 2.92368
33 : 139 : 139.60475 : -0.60475
34 : 149 : 145.32301 : 3.67699
35 : 151 : 151.13945 : -0.13945
36 : 157 : 156.63882 : 0.36118
37 : 163 : 162.28080 : 0.71920
38 : 167 : 167.82842 : -0.82842
```

```
39 : 173 : 173.30535 : -0.30535
40 : 179 : 178.91065 : 0.08935
41 : 181 : 184.23231 : -3.23231
42 : 191 : 189.71124 : 1.28876
43 : 193 : 195.28774 : -2.28774
44 : 197 : 200.43450 : -3.43450
45 : 199 : 205.20343 : -6.20343
46 : 211 : 210.35030 : 0.64970
47 : 223 : 216.63823 : 6.36177
48 : 227 : 223.24748 : 3.75252
49 : 229 : 229.34422 : -0.34422
50 : 233 : 234.99796 : -1.99796
51 : 239 : 240.57030 : -1.57030
```

Nuevamente, el resultado obtenido es, de lejos, superior a su similar de [1]. Y no solo eso, la aproximación es tan buena que $p_{2,k}$ difiere muy poco de su valor real, más sin embargo, todavía hay espacio para un tercer ajuste.

8 Tercer ajuste $P_{3,k}$

Para este tercer ajuste, introducimos un criterio diferente, es decir, no hacemos mínimos cuadrados y en su lugar aproximamos p_k mediante la siguiente expresión, que proviene del análisis de los resultados obtenidos en el segundo ajuste:

$$p_{3,k} = p_{k-1}\frac{p_{2,k-2}}{p_{2,k-1}}$$

Para ello introducimos el siguiente fragmento de código en la función que recoge los ajustes consecutivos, `RunAjusteMC()`:

```
// realiza un segundo ajuste de diferencias
cout<<"\n";
cout<<"AJUSTE DE SEGUNDAS DIFERECNIAS:\n";
q=5;
mpf_class rel(0.0,iPrec);
mpf_class p3(0.0,iPrec);
for(i=2;i<uIters-1;i++)
{
 rel=P2[i-2]/P2[i-1];
```

```
 p3=rel*q;
 P3[i-2]=p3;
 d=q-p3;
 D2[i-2]=d;
 pv=NextPrime(q);
 cout<<i<<" : "<<q<<" : ";
 gmp_printf("%.5Ff : %.5Ff\n",p3,d);
 q=pv;
}
```

Ejecutando este nuevo entorno se consigue la salida:

```
AJUSTE DE SEGUNDAS DIFERECNIAS:
2 : 5 : 3.10345 : 1.89655
3 : 7 : 4.22917 : 2.77083
4 : 11 : 8.55407 : 2.44593
5 : 13 : 10.47552 : 2.52448
6 : 17 : 14.49996 : 2.50004
7 : 19 : 16.45142 : 2.54858
8 : 23 : 19.10362 : 3.89638
9 : 29 : 25.07574 : 3.92426
10 : 31 : 27.25331 : 3.74669
11 : 37 : 32.66714 : 4.33286
12 : 41 : 37.45597 : 3.54403
13 : 43 : 39.94915 : 3.05085
14 : 47 : 43.28602 : 3.71398
15 : 53 : 48.37252 : 4.62748
16 : 59 : 54.60216 : 4.39784
17 : 61 : 56.91066 : 4.08934
18 : 67 : 62.60900 : 4.39100
19 : 71 : 67.17826 : 3.82174
20 : 73 : 69.28984 : 3.71016
21 : 79 : 74.92893 : 4.07107
22 : 83 : 78.72336 : 4.27664
23 : 89 : 83.91208 : 5.08792
24 : 97 : 91.52082 : 5.47918
25 : 101 : 96.22849 : 4.77151
26 : 103 : 98.81477 : 4.18523
27 : 107 : 103.20322 : 3.79678
```

```
28 : 109 : 105.54201 : 3.45799
29 : 113 : 108.32262 : 4.67738
30 : 127 : 121.01680 : 5.98320
31 : 131 : 125.23304 : 5.76696
32 : 137 : 131.57472 : 5.42528
33 : 139 : 133.53055 : 5.46945
34 : 149 : 143.26589 : 5.73411
35 : 151 : 145.69860 : 5.30140
36 : 157 : 151.54162 : 5.45838
37 : 163 : 157.61198 : 5.38802
38 : 167 : 161.72234 : 5.27766
39 : 173 : 167.57988 : 5.42012
40 : 179 : 173.82947 : 5.17053
41 : 181 : 175.77265 : 5.22735
42 : 191 : 185.54594 : 5.45406
43 : 193 : 188.04414 : 4.95586
44 : 197 : 192.42172 : 4.57828
45 : 199 : 194.13084 : 4.86916
46 : 211 : 204.87573 : 6.12427
47 : 223 : 216.39807 : 6.60193
48 : 227 : 220.96558 : 6.03442
49 : 229 : 223.49056 : 5.50944
50 : 233 : 227.60301 : 5.39699
```

Analizando la salida, vemos que, a diferencia del ajuste anterior, volvemos a tener una serie de diferencias que se comportan en forma aproximadamente lineal, lo cual nos permite conjeturar que un nuevo ajuste por mínimos cuadrados podría funcionar.

9 Cuarto ajuste $P_{4,k}$

En este ajuste, volvemos a los mínimos cuadrados y para ello usamos la función `AjusteDif` pero cambiando la matriz que provee las diferencias, esto es, la línea de código

```
p=D1[i-1];
```

se sustituye por

```
if(iOption==0)p=D1[i-1];
else p=D2[i-1];
```

y la rutina se ejecuta con la opción 1 para que se ejecuta la segunda línea que contiene las diferencias que intervienen en este ajuste.

También, agregamos el siguiente fragmento de código a la función que resume las novedades en cada ajuste, `RunAjusteMC()`:

```
cout<<"\n";
cout<<"AJUSTE DE TERCERAS DIFERECNIAS:\n";
mpf_class p4(0.0,iPrec);
mpf_class d3(0.0,iPrec);
mpz_class pa;
q=5;
char *pStr;
pStr=new char[1000];
for(i=2;i<uIters-1;i++)
{
 pv=NextPrime(q);
 d3=AjusteDif(i,1);
 p4=P3[i-2]+d3;
 pa=p4;
 if(pa%2==0)p4=p4-1;
 P4[i-2]=p4;
 d=p4-q;
 cout<<i<<" : "<<q<<" : ";
 gmp_printf("%.5Ff : %.5Ff\n",p4,d);
 p=q;
 gmp_sprintf(pStr,"%d : %.0Ff : %.5Ff : %.5Ff\n",i,p,p4,d);
 str+=pStr;
 q=pv;
}
```

Con estas modificaciones, se consigue la siguiente salida:

```
AJUSTE DE TERCERAS DIFERECNIAS:
2 : 5 : 5.89655 : 0.89655
3 : 7 : 7.55580 : 0.55580
4 : 11 : 11.38542 : 0.38542
```

```
5 : 13 : 13.08258 : 0.08258
6 : 17 : 17.92884 : 0.92884
7 : 19 : 19.77896 : 0.77896
8 : 23 : 23.02295 : 0.02295
9 : 29 : 29.35203 : 0.35203
10 : 31 : 31.67287 : 0.67287
11 : 37 : 37.38138 : 0.38138
12 : 41 : 41.10241 : 0.10241
13 : 43 : 43.38191 : 0.38191
14 : 47 : 47.73808 : 0.73808
15 : 53 : 53.07548 : 0.07548
16 : 59 : 59.43736 : 0.43736
17 : 61 : 61.77214 : 0.77214
18 : 67 : 67.55308 : 0.55308
19 : 71 : 71.06497 : 0.06497
20 : 73 : 73.10242 : 0.10242
21 : 79 : 79.74464 : 0.74464
22 : 83 : 83.57581 : 0.57581
23 : 89 : 87.93383 : -1.06617
24 : 97 : 95.74972 : -1.25028
25 : 101 : 101.51969 : 0.51969
26 : 103 : 103.06709 : 0.06709
27 : 107 : 107.36172 : 0.36172
28 : 109 : 109.56783 : 0.56783
29 : 113 : 113.39808 : 0.39808
30 : 127 : 125.30858 : -1.69142
31 : 131 : 129.68728 : -1.31272
32 : 137 : 137.12936 : 0.12936
33 : 139 : 139.17872 : 0.17872
34 : 149 : 149.02758 : 0.02758
35 : 151 : 151.51151 : 0.51151
36 : 157 : 157.41678 : 0.41678
37 : 163 : 163.53453 : 0.53453
38 : 167 : 167.67497 : 0.67497
39 : 173 : 173.57325 : 0.57325
40 : 179 : 179.83387 : 0.83387
41 : 181 : 181.79153 : 0.79153
42 : 191 : 191.59862 : 0.59862
```

```
43 : 193 : 193.08051 : 0.08051
44 : 197 : 197.40777 : 0.40777
45 : 199 : 199.09543 : 0.09543
46 : 211 : 209.92878 : -1.07122
47 : 223 : 221.57285 : -1.42715
48 : 227 : 227.20515 : 0.20515
49 : 229 : 229.74665 : 0.74665
50 : 233 : 233.86471 : 0.86471
```

Los resultados obtenidos son plenamente satisfactorios, como se puede apreciar fácilmente por inspección. Es decir, hemos llegado a una relación funcional aceptable para aproximar el n-ésimo primo mediante la expresión:

$$p_{4,k} = p_{3,k} + \delta_{3,k-1} \tag{8}$$

También, escrito como:

$$p_k = p_{3,k} + \delta_{3,k-1}$$

dado que ya no hace falta realizar más ajustes.

Un comentario especial merecen algunos valores que presentan diferencias negativas, esto es, donde $\delta_{3,k-1} \approx -1$. Estos valores son muy escasos y curiosamente solo aparecen antes del primo 500, es decir, antes de computar p_{500}. En efecto, después de computar $p_{1000000}$ y analizar los datos se verifica que no aparecen tales valores después del mencionado valor.

10 Primer millón de primos

Me complace presentar en esta sección el primer millón de números primos aproximado por una relación funcional, o si lo prefiere, obtenido a través de una fórmula para el n-ésimo primo (ecuación 8), basada en los $n-1$ primos anteriores. **No tenemos conocimiento de un cálculo de esta naturaleza en la literatura científica internacional**.

Desde luego, no puedo presentar aquí tal cantidad de primos pero, para que no alberguen dudas, presentamos segmentos del listado producido por el software presentado en esta investigación.

```
2 : 5 : 5.89655 : 0.89655
3 : 7 : 7.55580 : 0.55580
4 : 11 : 11.38542 : 0.38542
```

```
5 : 13 : 13.08258 : 0.08258
6 : 17 : 17.92884 : 0.92884
7 : 19 : 19.77896 : 0.77896
8 : 23 : 23.02295 : 0.02295
9 : 29 : 29.35203 : 0.35203
10 : 31 : 31.67287 : 0.67287
11 : 37 : 37.38138 : 0.38138
12 : 41 : 41.10241 : 0.10241
13 : 43 : 43.38191 : 0.38191
14 : 47 : 47.73808 : 0.73808
15 : 53 : 53.07548 : 0.07548
16 : 59 : 59.43736 : 0.43736
17 : 61 : 61.77214 : 0.77214
18 : 67 : 67.55308 : 0.55308
19 : 71 : 71.06497 : 0.06497
 .
 .
 .
49994 : 611903 : 611903.56979 : 0.56979
49995 : 611921 : 611921.56990 : 0.56990
49996 : 611927 : 611927.57074 : 0.57074
49997 : 611939 : 611939.57098 : 0.57098
49998 : 611951 : 611951.57202 : 0.57202
49999 : 611953 : 611953.57392 : 0.57392
50000 : 611957 : 611957.57497 : 0.57497
50001 : 611969 : 611969.57552 : 0.57552
50002 : 611977 : 611977.57585 : 0.57585
50003 : 611993 : 611993.57616 : 0.57616
50004 : 611999 : 611999.57700 : 0.57700
50005 : 612011 : 612011.57724 : 0.57724
50006 : 612023 : 612023.57732 : 0.57732
50007 : 612037 : 612037.57799 : 0.57799
50008 : 612041 : 612041.57984 : 0.57984
50009 : 612043 : 612043.58157 : 0.58157
50010 : 612049 : 612049.58242 : 0.58242
 .
 .
 .
```

```
577659 : 8603467 : 8603467.54797 : 0.54797
577660 : 8603471 : 8603471.54814 : 0.54814
577661 : 8603477 : 8603477.54826 : 0.54826
577662 : 8603489 : 8603489.54827 : 0.54827
577663 : 8603509 : 8603509.54826 : 0.54826
577664 : 8603521 : 8603521.54832 : 0.54832
577665 : 8603533 : 8603533.54815 : 0.54815
577666 : 8603579 : 8603579.54791 : 0.54791
577667 : 8603591 : 8603591.54804 : 0.54804
577668 : 8603593 : 8603593.54825 : 0.54825
577669 : 8603597 : 8603597.54842 : 0.54842
577670 : 8603603 : 8603603.54856 : 0.54856
577671 : 8603611 : 8603611.54843 : 0.54843
577672 : 8603657 : 8603657.54823 : 0.54823
 .
 .
 .
774985 : 11787121 : 11787121.55062 : 0.55062
774986 : 11787133 : 11787133.55060 : 0.55060
774987 : 11787157 : 11787157.55052 : 0.55052
774988 : 11787179 : 11787179.55053 : 0.55053
774989 : 11787187 : 11787187.55049 : 0.55049
774990 : 11787221 : 11787221.55042 : 0.55042
774991 : 11787229 : 11787229.55041 : 0.55041
774992 : 11787257 : 11787257.55026 : 0.55026
774993 : 11787287 : 11787287.55020 : 0.55020
774994 : 11787299 : 11787299.55030 : 0.55030
774995 : 11787301 : 11787301.55041 : 0.55041
774996 : 11787313 : 11787313.55049 : 0.55049
774997 : 11787319 : 11787319.55063 : 0.55063
774998 : 11787323 : 11787323.55075 : 0.55075
774999 : 11787331 : 11787331.55086 : 0.55086
 .
 .
 .
999965 : 15485411 : 15485411.52436 : 0.52436
999966 : 15485429 : 15485429.52437 : 0.52437
999967 : 15485441 : 15485441.52443 : 0.52443
```

```
999968 : 15485447 : 15485447.52446 : 0.52446
999969 : 15485471 : 15485471.52448 : 0.52448
999970 : 15485473 : 15485473.52452 : 0.52452
999971 : 15485497 : 15485497.52439 : 0.52439
999972 : 15485537 : 15485537.52433 : 0.52433
999973 : 15485539 : 15485539.52445 : 0.52445
999974 : 15485543 : 15485543.52456 : 0.52456
999975 : 15485549 : 15485549.52465 : 0.52465
999976 : 15485557 : 15485557.52472 : 0.52472
999977 : 15485567 : 15485567.52476 : 0.52476
999978 : 15485581 : 15485581.52473 : 0.52473
999979 : 15485609 : 15485609.52473 : 0.52473
999980 : 15485611 : 15485611.52483 : 0.52483
999981 : 15485621 : 15485621.52481 : 0.52481
999982 : 15485651 : 15485651.52480 : 0.52480
999983 : 15485653 : 15485653.52488 : 0.52488
999984 : 15485669 : 15485669.52491 : 0.52491
999985 : 15485677 : 15485677.52497 : 0.52497
999986 : 15485689 : 15485689.52498 : 0.52498
999987 : 15485711 : 15485711.52491 : 0.52491
999988 : 15485737 : 15485737.52489 : 0.52489
999989 : 15485747 : 15485747.52493 : 0.52493
999990 : 15485761 : 15485761.52496 : 0.52496
999991 : 15485773 : 15485773.52501 : 0.52501
999992 : 15485783 : 15485783.52504 : 0.52504
999993 : 15485801 : 15485801.52508 : 0.52508
999994 : 15485807 : 15485807.52508 : 0.52508
999995 : 15485837 : 15485837.52505 : 0.52505
999996 : 15485843 : 15485843.52515 : 0.52515
999997 : 15485849 : 15485849.52523 : 0.52523
999998 : 15485857 : 15485857.52532 : 0.52532
999999 : 15485863 : 15485963.52517 : 0.52517
```

Para quienes deseen conocer el listado completo de estos primos, lo pueden solicitar por la vía del correo electrónico, el archivo completo ocupa 41.4 MB de espacio en disco.

References

[1] Fandiño, A. Ramón. *Apuntes sobre una relación entre números primos.* Boletín de Matemáticas. Vol. XIV, Nos 1, 2, y 3. 1980. pp 127-144.

[2] Hotmath. *Recta que mejor se ajusta (Método de mínimos cuadrados).* https://www.varsitytutors.com/hotmath/hotmath_help/spanish/topics-/line-of-best-fit. 2018.

[3] MiProfe.com. *Mínimos cuadrados.* https://miprofe-com.cdn.ampproject.org/i/s/miprofe.com. 2018.

[4] Torre La Vega. *Ajuste por Mínimos Cuadrados.* Escuela Politécnica de Ingeniería de Minas y Energía . 2018. 13 p.

[5] Useche, H. *Un Millón de Precisiones Entorno a φ.* Cálculo del número áureo con múltiple precisión . 2018. 47 p.

[6] Useche, H. *Un millón de precisiones entorno a e.* Cálculo del número de Euler con múltiple precisión . 2018. 25 p.

[7] Useche, H. *Un millón de precisiones entorno a π.* Cálculo de π con múltiple precisión. 2018. 29 p.

[8] Wikipedia. *Mínimos cuadrados.* http/es.m.wikipedia.org/wiki/Minimos-_cuadrados.

[9] Wikipedia. *Formula for primes.* http://en.m.wikipedia.org/wiki/For-mula_for_primes.

[10] Zotkin, A. *Una fórmula que calcula el n-ésimo número primo.* https://tardigrados.wordpress.com/2013/02/25/una-formula-que-calcula-el-n-esimo-numero-primo/