UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

19|20

Trabajo fin de grado

Algoritmos para calcular el número π

Antonio Martín Masuda

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Algoritmos para calcular el número π

Autor: Antonio Martín Masuda
Tutor: Fernando Díez Rubio

junio 2020

Antonio Martín Masuda
Algoritmos para calcular el número π

Antonio Martín Masuda
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

La historia detrás del número π es inmensa y difícilmente abarcable. Se conoce desde hace milenios y aún así, sigue estando presente en actuales investigaciones. Muchos matemáticos se han fascinado porque aparece hasta en los ámbitos más inesperados. Este trabajo se centra en examinar los problemas y las fórmulas más importantes relacionadas con esta constante.

En primer lugar, se analizan, desde una perspectiva histórica, algunas de las ecuaciones, y los algoritmos que de ellas se derivan, más utilizadas para su cálculo. Entre ellas se encuentran el método original de Arquímedes, el de Newton con la llegada del cálculo, las fórmulas tipo Machin, el de la media aritmético-geométrica de Gauss, las identidades de Ramanujan y el algoritmo cuártico de los hermanos Borwein. El objetivo es encontrar el más eficiente y, para ello, se realiza un estudio de sus tiempos de ejecución en función del número de cifras correctas.

Una de las cuestiones que mayor inquietud provocó el número π el siglo pasado fue si existía una manera de hallar el dígito que ocupaba una cierta posición tras el punto decimal, sin tener que calcularlo desde el principio. En un comienzo, se creía que no era posible, hasta que se encontró una solución para el logaritmo neperiano de dos. En el tercer capítulo se presenta el algoritmo de búsqueda de relaciones enteras dado un vector de números reales (algoritmo PSLQ) y su influencia en el descubrimiento de la fórmula BBP, que resolvía el problema. Posteriormente, Fabrice Bellard daría con una igualdad similar, aunque más eficiente.

A pesar de haber sido analizada durante tantos años, aún se desconocen algunas de sus propiedades. En particular, se cree que sus dígitos en cualquier base generan una secuencia normal y aleatoria. Un número irracional es normal si todas las palabras de k cifras aparecen con la misma frecuencia, para cualquier longitud finita k. El estudio de estas cualidades se realiza desde un enfoque estadístico. Tras generar una cantidad suficiente de decimales gracias a los algoritmos anteriores, se efectúan una serie de test para comprobar si las hipótesis de normalidad y aleatoriedad son probables.

Palabras clave

C, Python, Jupyter, número pi, PSLQ, BBP, algoritmos, números aleatorios

Abstract

The history behind the number π is immense and cannot be fully addressed. It has been known since the ancient civilizations and still, it is present in current investigations. Many mathematicians have been fascinated because it appears even in the most unexpected fields. This project focuses on examining the most critical problems and formulas related to this constant.

First of all, we will analyze, from a historical point of view, some of the equations and the algorithms these produce to calculate π. Among them: the original method invented by Archimedes, the one created by Newton from his calculus, the Machin-type formulas, Gauss' arithmetic-geometric mean, the works of Ramanujan and the quartic algorithm of the Borwein brothers. The main objective is to find the most efficient and, to do so, an study of their execution times is conducted in terms of the number of correct digits.

One of the most inquisitive problems that put π in the spotlight of some researchers in the last century was to find the digit occupying the nth position after the decimal point, without the need of calculating all the previous ones. At the beginning, it was not clear this was even possible, until it was solved for the natural logarithm of two. In the third chapter we present the PSLQ integer relationship algorithm in a real number vector and its influence in the discovery of the BBP formula, which answered the question for π. Afterwards, Fabrice Bellard would come up with a similar identity, but more efficient.

Although this number has been studied for so long, we do not know some of his properties yet. In particular, it is believed that π is normal and that its digits generate a sequence of random numbers. An irrational number is said to be normal If, in any base, the words of fixed length appear equally distributed. The study of these qualities is performed with a statistical approach. After generating a big enough quantity of decimals through the previous algorithms, a series of tests are run over them to check if the hypothesis of normality and randomness are probable.

Keywords

C, Python, Jupyter, pi number, PSLQ, BBP, algorithms, randomness tests

ÍNDICE

Listas

Lista de algoritmos

Lista de códigos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

El número π, definido como la proporción entre la longitud de una circunferencia y su diámetro, es una de las constantes matemáticas más antiguas conocidas y más examinadas en la historia. Durante siglos, los matemáticos han descubierto una gran variedad de métodos para generar mejores aproximaciones. A pesar de todo, aun a día de hoy, hay algunas cuestiones sin resolver que provocan que siga siendo objeto de estudio.

Este Trabajo Fin de Grado analiza la evolución de los algoritmos creados para calcularlo, en términos de eficiencia y velocidad de convergencia, así como algunos de los problemas que los originaron.

1.1. Motivación

Son varias las propiedades conocidas sobre el número π como, por ejemplo, su irracionalidad (tiene infinitos decimales sin ningún patrón que se repita indefinidamente) y su trascendencia (no es raíz de ningún polinomio con coeficientes enteros). Sin embargo, hay otras aún desconocidas, que se conjeturan que son ciertas, pero de las que todavía no existe demostración. Entre éstas, se encuentran, por ejemplo, la aleatoriedad o la normalidad de sus decimales en cualquier base b. Al ser irracional, se cree que los dígitos generan una secuencia sin fin de números aparentemente aleatorios (aleatoriedad) y que las cifras aparecen con una frecuencia límite de $1/b$ (normalidad).

Con el objetivo de responder a la pregunta de si el número π es normal o no, en las últimas décadas se ha intentado generar el mayor número de decimales posibles, para poder analizarlos estadísticamente. Por este motivo, conviene encontrar un algoritmo con un orden alto de convergencia y a la vez computacionalmente eficiente.

No obstante, para tratar el asunto de la normalidad, no es necesario calcular los dígitos desde el principio. Si se pudieran hallar a partir de una cierta posición, se podrían analizar las cifras generadas sucesivamente, sin tener que computar todas las previas. Esta cuestión fue resuelta a finales del siglo pasado con una metodología de gran relevancia.

1.2. Objetivos

El documento está basado en tres objetivos claramente diferenciables pero conectados, de manera que cada uno es el fundamento sobre el que se apoyan los siguientes:

1.– **Encontrar un algoritmo eficaz a la hora de calcular los decimales de π**

El objetivo inicial es entender la importancia del número π en el ámbito de las matemáticas, sus aplicaciones y su evolución a lo largo de la historia. Esto ayudará a discernir los algoritmos más relevantes para su cálculo y poder escoger entre ellos el más eficaz.

2.– **Calcular el n-ésimo dígito de π, sin necesidad de hallar las cifras anteriores**

Los algoritmos mencionados en el primer apartado, obtienen una aproximación de π de manera secuencial. Es decir, en cada iteración se consiguen nuevos decimales correctos en orden posicional. No obstante, surge la duda de si se pueden conocer los dígitos de π a partir de una posición dada, sin tener que encontrar todas las cifras previas. El objetivo consiste en encontrar un método que resuelva esta cuestión y comprobar su eficiencia.

3.– **Estudiar la normalidad de π estadísticamente**

Una vez descubiertos algoritmos que puedan generar una gran cantidad de dígitos de π en un tiempo razonable, podemos analizar la secuencia de números generada. El fin es verificar si es factible que esta sucesión sea aleatoria.

1.3. Estructura del documento

El documento se divide en tres capítulos, además de la introducción y de las conclusiones. Cada uno de ellos se corresponde con uno de los objetivos del trabajo indicados anteriormente:

1.– **Algoritmos de π**

En el segundo capítulo se analizan una serie de algoritmos con un enfoque más histórico. De todos los métodos que han surgido a lo largo de los siglos, se han escogido algunos de los más utilizados y más representativos de cada época, para compararlos posteriormente.

2.– **Cálculo del n-ésimo dígito**

En el tercer capítulo se describe uno de los algoritmos más importantes del siglo XX, denominado PSLQ, a través del cual, se consiguió una fórmula para calcular el n-ésimo dígito del número π en base hexadecimal o en binario, sin hallar las cifras previas. Posteriormente, se expone un segundo algoritmo, similar al anterior pero mejorado, descubierto por Fabrice Bellard.

3.– **Análisis estadístico**

En el cuarto capítulo se analiza la secuencia de cifras en la expansión de π en diferentes bases: binario, decimal y hexadecimal, con el fin de observar si ésta se comporta de manera aleatoria. Para ello, se realizan una serie de tests estadísticos que debe satisfacer para no descartar esta hipótesis.

Al final del documento, hemos añadido varios apéndices con el propósito de añadir información relevante sobre la codificación e implementación de los algoritmos de los dos primeros apartados y de los tests estadísticos ejecutados en el tercero. Además, incluyen algunas demostraciones matemáticas, que no se necesitan para comprender el cuerpo del documento, pero complementan la información.

2

Breve historia de los algoritmos de π

Por su definición, intrínsecamente geométrica, el número π aparece de forma natural en este ámbito, como por ejemplo, en ciertas fórmulas para hallar el área comprendida por una curva cerrada o el volumen contenido dentro de una superficie. Estas ecuaciones se pueden obtener mediante integrales, por lo que, indirectamente, está muy ligado al cálculo infinitesimal.

Una segunda definición de π proviene del radián, una unidad de medida de ángulos. Un radián comprende el ángulo que habría que moverse para recorrer justo la longitud del radio sobre la circunferencia. Es decir, 2π radianes se corresponde con la circunferencia completa y, por tanto, equivale a $360°$. Por este motivo, su uso se amplía a más dominios, como el cálculo trigonométrico, el cálculo diferencial o la variable compleja. En especial, destaca una de las expresiones más importantes por relacionar cinco constantes fundamentales de áreas matemáticas muy dispares, conocida como la identidad de Euler: $e^{\imath\pi} + 1 = 0$.

Sin embargo, también se manifiesta en sitios inesperados. Un ejemplo es la función zeta de Riemann, muy estudiada en análisis matemático. Ésta se define como sigue: $\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$. Su valor en los números naturales pares es un múltiplo de π^s. En particular, el problema de Basilea se centraba en el caso en el que s vale dos. Leonhard Euler lo resolvió por primera vez, obteniendo el siguiente resultado: $\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$.

Estos ejemplos son una pequeña representación de la enorme cantidad de aplicaciones del número π. Uno de los más prolíferos es la física, donde aparece en innumerables ocasiones: desde la física cuántica (cálculo de la energía de un fotón) hasta la astrofísica (leyes de Kepler), pasando por la física clásica (período de un péndulo) e incluso en electromagnetismo (ecuaciones de Maxwell). Asimismo, se utiliza en ambientes fuera de la ciencia, como puede ser en arquitectura y dibujo técnico, de nuevo, por su relación con la geometría.

Esta breve exposición de algunos de los usos del número π da una idea de su gran relevancia y de los motivos por los que, desde sus orígenes, muchos matemáticos han dedicado su tiempo a estudiarlo. La historia de esta constante es larga y cada vez más densa. Ha pasado por muchas y muy variadas épocas, cada una con una forma de pensar distinta que ha dejado huella en los respectivos algoritmos, tratando de obtener mejores aproximaciones.

En este sentido, se libra una gran batalla, especialmente intensificada en las últimas décadas y que continúa a día de hoy, por batir el récord de dígitos calculados. Actualmente, lo ostenta la informática Emma Haruka, con más de 31 billones de dígitos obtenidos en marzo del año pasado, según *Guinness World Records*. No obstante, utilizar π con cuarenta cifras correctas es suficiente para hallar la longitud de la circunferencia máxima que podría haber alcanzado el universo, si éste se hubiera generado a partir de un punto hace 20 billones de años y expandido a la velocidad de la luz en todas direcciones, con un error del tamaño de un átomo de hidrógeno, según Borwein [1].

Entonces, surge la pregunta de para qué conseguir más decimales, sabiendo además que nunca vamos a llegar al final. Podría parecer una lucha inútil o innecesaria, pero nada más lejos de la realidad. Éste ha sido uno de los principales impulsores del cálculo computacional, ayudando a probar nuevos métodos, software y hardware relacionados con esta área. De hecho, es interesante contemplar cómo la evolución de las matemáticas y de la tecnología y la búsqueda incesante de una mejor aproximación de π, se han ayudado mutuamente para sus respectivos desarrollos.

En este capítulo se presentan algunos de los algoritmos [2], junto con su contexto histórico, que han permitido estos avances. Se presentan en orden cronológico, desde sus comienzos en la Edad Antigua hasta la actualidad. Los correspondientes códigos implementados son una mera traducción de los pseudocódigos tal y como se exponen y se ubican en el apéndice A.

2.1. Algoritmo de Arquímedes (250 A.C.)

La Grecia clásica se caracterizó por haber sido el epicentro cultural y de avance científico de la época. Allí se reunieron una cantidad innumerable de maestros filosóficos, físicos o matemáticos. Uno de los más destacados fue Arquímedes, quien se dedicó a estudiar diversos problemas, entre los que se encuentra la aproximación del número π. Fue el primero en encontrar un algoritmo iterativo que lo calcula con un error tan pequeño como queramos. De hecho, tuvieron que pasar cerca de dos milenios para mejorar la eficiencia de este método.

En aquel momento, la geometría era la rama de las matemáticas que reinaba sobre el resto. Dirigía la forma de pensar, quizás por la belleza contenida en sus relaciones, muy representadas en sus construcciones arquitectónicas. Arquímedes no fue una excepción y planteó el problema como se indica en la figura 2.1. Su idea fue aproximar la longitud de una circunferencia de diámetro uno, que vale π, mediante el perímetro de los polígonos regulares inscrito y circunscrito con $k_n = 3 * 2^n$ lados cada uno. Por el teorema de Tales, la longitud de los segmentos $\overline{AB}$ y $\overline{CD}$ coinciden con el de $\overline{ST}$ y $\overline{MN}$, respectivamente. Por ser el radio de la circunferencia exterior uno, éstos últimos resultan ser el seno y tangente del ángulo α, respectivamente. A partir de aquí, obtenemos una fórmula para hallar los perímetros circunscrito e inscrito:

$$a_n = k_n \tan \frac{\pi}{k_n}, \quad b_n = k_n \operatorname{sen} \frac{\pi}{k_n}.$$

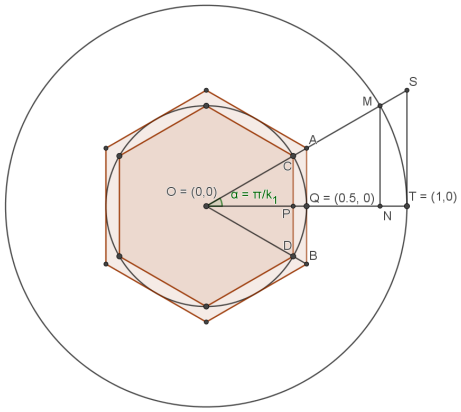

Figura 2.1: Ilustración del planteamiento de Arquímedes

Las relaciones trigonométricas del ángulo mitad nos serán de utilidad para obtener la iteración: $\operatorname{sen} x = 2\operatorname{sen}\frac{x}{2}\cos\frac{x}{2}$ y $\tan\frac{x}{2} = \frac{\operatorname{sen} x}{1+\cos x}$. De ellas se deducen las fórmulas iterativas del algoritmo de Arquímedes:

$$\frac{2a_nb_n}{a_n+b_n} = 2k_n\frac{\tan\frac{\pi}{k_n}\operatorname{sen}\frac{\pi}{k_n}}{\tan\frac{\pi}{k_n}+\operatorname{sen}\frac{\pi}{k_n}} = 2k_n\frac{\operatorname{sen}\frac{\pi}{k_n}}{1+\cos\frac{\pi}{k_n}} = 2k_n\tan\frac{\pi}{2k_n} = a_{n+1} \tag{2.1}$$

$$\sqrt{a_{n+1}b_n} = \sqrt{2k_n\tan\frac{\pi}{2k_n}k_n\operatorname{sen}\frac{\pi}{k_n}} = \sqrt{2k_n\tan\frac{\pi}{2k_n}2k_n\operatorname{sen}\frac{\pi}{2k_n}\cos\frac{\pi}{2k_n}} = 2k_n\operatorname{sen}\frac{\pi}{2k_n} = b_{n+1}$$

Partiendo del caso inicial $n = 1$, que ilustra la figura superior y donde $a_1 = \sqrt{12}$ y $b_1 = 3$, tenemos el algoritmo iterativo cuyo pseudocódigo se muestra en la figura 2.1. Éste proporciona dos bits correctos en cada iteración (unos tres decimales cada cinco iteraciones). Este hecho se puede ver a través de los desarrollos de Taylor de la tangente, que es $x + O(x^3)$, y de la función $1 - \cos x$, que es $\frac{x^2}{2} + O(x^4)$, de donde se deduce que $\tan x < x$ y $1 - \cos x < \frac{x^2}{2}$:

$$a_n - b_n = k_n\left(\tan\frac{\pi}{k_n} - \operatorname{sen}\frac{\pi}{k_n}\right) = k_n\tan\frac{\pi}{k_n}\left(1-\cos\frac{\pi}{k_n}\right) < k_n\frac{\pi}{k_n}\frac{\pi^2}{2k_n^2} < \frac{\pi^3}{18*4^n}.$$

```
input : Un entero positivo p
output: El número pi calculado hasta una precisión p
1  inicializaciones;
2  a ← RaizCuadrada ( 12 );
3  b ← 3;
4  umbral ← 2^-p;
5  iteración;
6  while a − b > umbral do
7  |  a ← ( 2 × a × b ) / ( a + b );
8  |  b ← RaizCuadrada ( a × b );
9  end
10 return a;
```

Algoritmo 2.1: Algoritmo de Arquímedes

2.2. Algoritmo de Newton (1666)

Con la llegada del cálculo infinitesimal, desarrollado paralelamente entre Sir Isaac Newton y Gottfried Leibniz, se abrió todo un campo de las matemáticas. Este innovador enfoque permitía calcular áreas bajo curvas y desarrollos de funciones trigonométricas con mayor facilidad. A través de estos métodos, surgieron nuevas relaciones para el número π que superaban en eficiencia el de Arquímedes, entre las que se encuentra la siguiente:

$$\frac{\pi}{3} = \sum_{n=0}^{\infty} \frac{\binom{2n}{n}}{(2n+1)16^n} \tag{2.2}$$

Esta igualdad proviene del desarrollo de Taylor de la función arcoseno y del hecho de que el seno de $\frac{\pi}{3}$ es 0.5:

$$f(x) = \frac{\arcsin 2x}{2x} = \sum_{n=0}^{\infty} \frac{\binom{2n}{n}}{2n+1} x^{2n} \implies \frac{\pi}{3} = f(0{,}25) = \sum_{n=0}^{\infty} \frac{\binom{2n}{n}}{(2n+1)16^n}$$

Este algoritmo proporciona al menos dos bits correctos en cada iteración (aproximadamente tres decimales correctos cada cinco iteraciones). Este hecho se puede ver comparando las sumas parciales y usando la fórmula de Stirling:

$$s_n - s_{n-1} = \frac{\binom{2n}{n}}{(2n+1)16^n} < \frac{(2n)!}{(n!)^2 16^n} < \frac{(2n)^{2n+1/2}e^{2n}}{n^{2n+1}e^{2n}16^n} < \frac{2^{2n}}{16^n} = \frac{1}{4^n}$$

input : Un entero positivo p
output: El número pi calculado hasta una precisión p

```
1  inicializaciones;
2  umbral ← 2^-p;
3  suma_inicial ← 0;
4  suma_final ← 1;
5  potencia ← 1;
6  exponencial ← 1;
7  binomio ← 2;
8  impar ← 1;
9  iteración;
10 while suma_final − suma_inicial > umbral do
11     impar ← impar + 2;
12     exponencial ← exponencial × 16;
13     suma_inicial ← suma_final;
14     suma_final ← suma_final + binomio / ( impar × exponencial );
15     potencia ← potencia + impar;
16     binomio ← ( impar + 1 ) × impar × binomio / potencia;
17 end
18 return 3 × suma_final
```

Algoritmo 2.2: Algoritmo de Newton

El cálculo del coeficiente del binomio de Newton involucra factoriales, lo que es muy costoso computacionalmente. Por ello, es preferible representar la fórmula (2.2) de forma iterativa:

$$\frac{\pi}{3} = \sum_{n=0}^{\infty} \frac{b_n}{i_n e_n}, \text{ donde } b_n = \frac{2n(2n-1)b_{n-1}}{n^2} = \frac{(i_{n-1}+1)i_{n-1}b_{n-1}}{n^2}, \; i_n = i_{n-1}+2 \text{ y } e_n = 16e_{n-1}.$$

En el pseudocódigo, estas iteraciones se corresponden con las variables *binomio*, *impar* y *exponencial*. Además, para evitar elevar al cuadrado, que es más costoso especialmente al trabajar con una gran cantidad de decimales, se sustituye por esta igualdad: $p_n = n^2 = (n-1)^2 + 2n - 1 = p_{n-1} + i_{n-1}$. Esta última iteración está codificada mediante la variable *potencia*.

2.3. Algoritmo de Machin (1706)

Unos años más tarde, John Machin dedujo una nueva fórmula para el cálculo de π con la que obtuvo sus primeros 100 decimales. Ésta se basa en diferentes relaciones trigonométricas de la función arcotangente y su nexo con π, dado que $\frac{\pi}{4} = \arctan 1$:

$$\boxed{\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}} \tag{2.3}$$

Las igualdades que involucran una combinación lineal de diferentes valores de la arcotangente para computar π son conocidas usualmente como series de tipo Machin, en honor a su descubridor. Conseguir una de estas series es algo laborioso, pero una vez hallada, se puede comprobar su certeza con relativa facilidad. Dado un número complejo $z = x + yi$, su argumento $\arg z$ es $\arctan\frac{y}{x}$. Además, se cumple que $\arg z^n = n * \arg z$. Usando estas propiedades, obtenemos la siguiente identidad:

$$\frac{(5+i)^4}{239+i} = 2 + 2i$$

Igualando los argumentos de ambos miembros, se tiene la fórmula (2.3). La función arcotangente se puede aproximar mediante su serie de Taylor en cualquier punto del intervalo (-1,1):

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^k}{2n+1} x^{2n+1} \implies \arctan\frac{1}{m} = \sum_{n=0}^{\infty} \frac{(-1)^k m}{(2n+1)m^{2(n+1)}}$$

No obstante, se puede mejorar la convergencia de esta serie, calculando dos términos cada vez [3]:

$$\arctan\frac{1}{m} = \sum_{n=0}^{\infty} \frac{m[(4n+3)m^2 - (4n+1)]}{(16n^2+16n+3)m^{4(k+1)}}$$

El pseudocódigo de la función arcotangente muestra la implementación de este segundo método. Finalmente, codificar la ecuación (2.3) se simplifica bastante, dado que únicamente es necesario llamar al procedimiento anterior con los parámetros $\frac{1}{5}$ y $\frac{1}{239}$.

input : Un número *m* y un umbral *umbral*
output: La arcotangente del recíproco de m con una precisión indicada por umbral

inicializaciones;
m2 $\leftarrow m^2$;
m3 $\leftarrow m^3$;
m4 $\leftarrow m^4$;
potencia $\leftarrow$ m4;
auxiliar $\leftarrow 4 \times ($ m3 $- m)$;
op1 $\leftarrow 3 \times$ m3 $- m$;
op2 $\leftarrow 3$;
$suma_{inicial} \leftarrow 0$;
$suma_{final} \leftarrow$ op1 / (op2 $\times$ potencia);
i $\leftarrow 1$;
iteración;
while $suma_{final} - suma_{inicial} >$ umbral **do**
    op1 $\leftarrow$ op1 $+$ auxiliar;
    op2 $\leftarrow$ op2 $+ 32 \times$ i;
    potencia $\leftarrow$ potencia $\times$ m4;
    $suma_{inicial} \leftarrow suma_{final}$;
    $suma_{final} \leftarrow suma_{final} +$ op1 / (op2 $\times$ potencia);
    i $\leftarrow$ i $+ 1$;
end
return $suma_{final}$;

Algoritmo 2.3: Función arcotangente

input : Un entero positivo p
output: El número pi calculado hasta una precisión p

inicializaciones;
umbral $\leftarrow 2^{-p}$;
return $16 \times \texttt{arctan}($ *5,* umbral $) - 4 \times \texttt{arctan}($ *239,* umbral $)$

Algoritmo 2.4: Algoritmo de Machin

2.4. Algoritmo de Gauss (1800)

Carl Friedrich Gauss, conocido como el príncipe de las matemáticas, fue un prodigio desde muy joven. A pesar de haber contribuido a diferentes áreas, como el álgebra o la estadística, su campo de interés fue la aritmética. En particular, la fórmula de la que se origina el algoritmo en cuestión procede de la media aritmético geométrica, que él mismo definió. Dado $\overline{x} \in \mathbf{R}^n$, un vector de n componentes, se definen la media aritmética y la media geométrica así:

$$M_A(\overline{x}) = \frac{\sum_{j=1}^{n} x_j}{n}, \quad M_G(\overline{x}) = \sqrt[n]{\prod_{j=1}^{n} x_j}$$

La media aritmético geométrica de dos números ($M_{AG}(a,b)$) es el límite común de la secuencia generada al calcular las respectivas medias aritmética y geométrica del paso anterior, empezando en a y b:

$$\overline{x}_0 = (a,b),\ a_{n+1} = M_A(\overline{x}_n),\ b_{n+1} = M_G(\overline{x}_n),\ \overline{x}_{n+1} = (a_n, b_n),\ M_{AG}(a,b) = \lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$$

A comienzos del siglo XIX, se dio cuenta de que la media aritmético geométrica, empezando en $a = 1$ y $b = \sqrt{2}/2$, cumple la siguiente igualdad:

$$\sum_{k=1}^{\infty} 2^k (a_k^2 - b_k^2) = \frac{1}{2} - \frac{2}{\pi}\left[M_{AG}\left(1, \frac{\sqrt{2}}{2}\right)\right]^2 \tag{2.4}$$

Sorprendentemente, hubo que esperar a que Eugene Salamin y Richard Brent dieran el paso de despejar π de esta ecuación, para expresarla de manera más adecuada para su implementación:

$$\pi = \lim_{n\to\infty} \frac{2a_n^2}{0{,}5 - \sum_{k=1}^{\infty} 2^k (a_k^2 - b_k^2)}$$

Este algoritmo cuadrático es utilizado por algunas librerías de alta precisión, como las que analizaremos en la sección 3.1.

input : Un entero positivo p
output: El número pi calculado hasta una precisión p

```
1  inicializaciones;
2  i ← 1;
3  pow2 ← 2;
4  a ← 0,5 + 0,25 × RaizCuadrada ( 2 );
5  b ← RaizCuadrada ( RaizCuadrada ( 2 ) / 2 );
6  s ← 2 × RaizCuadrada ( a ) − RaizCuadrada ( b ));
7  res_inicial ← 4;
8  res_final ← 2 × RaizCuadrada ( a ) / ( 0,5 − s );
9  umbral ← 2^−p;
10 iteración;
11 while res_inicial − res_final > umbral do
12     i ← i + 1;
13     a, b ← ( a + b ) / 2, RaizCuadrada ( a × b );
14     a2 ← a ²;
15     pow2 ← pow2 × 2;
16     s ← s + pow2 × ( a2 − b ² );
17     res_inicial ← res_final;
18     res_final ← 2 × a2 / ( 0,5 − s );
19 end
20 return res_final;
```

Algoritmo 2.5: Algoritmo de Salamin-Brent

2.5. Algoritmo de Ramanujan-Chudnovsky (1914)

Con el paso del tiempo, la evolución del cálculo ha permitido descubrir increíbles fórmulas vinculadas a π. Uno de los grandes responsables de estos hallazgos en el siglo pasado fue Srinivasa Ramanujan. Nació en la India con un don para la teoría de números y el cálculo de series, que él mismo atribuía a su vida espiritual. Más tarde, se fue a Inglaterra, donde el matemático G. H. Hardy le ayudó a desarrollar su talento, sobre todo a la hora de demostrar más rigurosamente sus ideas. Uno de sus numerosos estudios se centró en las integrales elípticas, a partir de las cuales obtuvo exóticas familias de series para el recíproco de π, las cuales tienen el siguiente formato:

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} C_n z^n (an + b), \text{ donde } a,\, b \in \mathbf{R} \text{ y } C_n \text{ es una constante que depende de } n.$$

Continuando esta línea de investigación, los hermanos Chudnovsky (David y Gregory) lograron una fórmula que converge muy rápidamente:

$$\frac{1}{\pi} = \frac{12}{\sqrt{640320^3}} \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(545140134n + 13591409)}{(3n!)(n!)^3 640320^{3n}} \tag{2.5}$$

A partir de ésta, se genera un algoritmo que proporciona quince decimales tras cada iteración:

input : Un entero positivo p
output: El número pi calculado hasta una precisión p

```
1  inicializaciones;
2  i ← 1;
3  divisor ← −640320^3;
4  cubo ← divisor;
5  factor ← 558731543;
6  incremento ← 545140134;
7  auxiliar ← 18;
8  coeficiente ← 120;
9  res_inicial ← 13591409;
10 res_final ← res_inicial + coeficiente × factor / divisor;
11 umbral ← 2^−p;
12 iteración;
13 while abs( res_final − res_inicial ) > umbral do
14     i ← i + 1;
15     divisor ← divisor × cubo;
16     factor ← factor + incremento;
17     coeficiente ← coeficiente × ( auxiliar^3 − 16 × auxiliar ) / i^3 ;
18     auxiliar ← auxiliar + 12;
19     res_inicial ← res_final;
20     res_final ← res_final + factor × coeficiente / divisor;
21 end
22 return RaizCuadrada( − cubo ) / ( 12 × res_final )
```

Algoritmo 2.6: Algoritmo de Ramanujan-Chudnovsky

Este método ha sido utilizado en numerosas ocasiones para romper el récord de decimales calculados, entre ellos el último de Emma Haruka, por su gran eficiencia. El bucle se encarga del sumatorio como se explica a continuación:

- El coeficiente C_n es $\frac{(6n)!}{(n!)^3(3n)!}$. Para evitar recalcular los factoriales en cada paso, lo cual sería muy costoso, se utiliza esta iteración:

$$C_n = \frac{\prod_{i=1}^{6}(6n+i)}{(n+1)^3\prod_{i=1}^{3}(3n+i)}\frac{(6n)!}{(3n)!(n!)^3} = \frac{(12n+10)(12n+6)(12n+2)}{(n+1)^3}C_{n-1}$$

El cómputo del numerador se puede optimizar más todavía, añadiendo una variable auxiliar:

$$(12n+10)(12n+6)(12n+2) = (12n+6)^3 - 16(12n+6) = aux_n^3 - 16aux_n$$

- La exponencial $\left(\frac{-1}{640320^3}\right)^n$ y el factor $545140134n + 13591409$ se hallan iterativamente, de manera natural:

$$exp_n = \frac{-1}{640320^3}exp_{n-1}, \quad f_n = f_{n-1} + 545140134.$$

2.6. Algoritmo de Borwein-Borwein (1987)

Los hermanos Peter y Johnatan Borwein diseñaron un algoritmo cuártico que se aproxima a $1/\pi$, mejorando el orden cuadrático de Salamin y Brent (2.5). De hecho, en el apéndice B se demuestra que una iteración del primero equivale a dos del segundo [4], a pesar de que aparentemente son muy diferentes:

```
   input  : Un entero positivo p
   output: El número pi calculado hasta una precisión p
1  inicializaciones;
2  i ← 0;
3  s ← RaizCuadrada ( 2 ) − 1;
4  t_inicial ← 0;
5  t_final ← 6 − 4 × RaizCuadrada ( 2 );
6  umbral ← 2^−p;
7  iteración;
8  while abs( t_final − t_inicial ) > umbral do
9      auxiliar ← RaizCuadrada( RaizCuadrada( 1 − s^4 ) );
10     s ← ( 1 − auxiliar ) / ( 1 + auxiliar );
11     t_inicial ← t_final;
12     t_final ← t_final × ( 1 + s )^4 − 2^( 2 × i + 3 ) × s × ( 1 + s + s^2 );
13     i ← i + 1;
14 end
15 return 1 / t_final;
```

Algoritmo 2.7: Algoritmo de Borwein-Borwein

El proceso iterativo del algoritmo viene dado por las siguientes ecuaciones:

$$s_{n+1} = \frac{1-(1-s_n^4)^{1/4}}{1+(1-s_n^4)^{1/4}} \tag{2.6}$$

$$t_{n+1} = t_n(1+s_{n+1}^4) - 2^{2n+3}s_{n+1}(1+s_{n+1}+s_{n+1}^2)$$

Partiendo de $s_0 = \sqrt{2}-1$ y $t_0 = 6-4\sqrt{2}$, con mucha dificultad se puede demostrar que t_n tiende al recíproco de π, proporcionando cuatro veces más decimales en cada paso. El investigador David H. Bailey implementó este método en el supercomputador Cray-2 de la NASA cuando fue construido, con el fin de verificar que no hubiera fallos de software ni de hardware. Durante el cálculo, con el que al final obtuvo cerca de treinta millones de dígitos batiendo el récord del momento, se detectaron algunos errores menores que hicieron posible su perfeccionamiento.

2.7. Análisis de los algoritmos

Con el fin de encontrar cuál de estos algoritmos es más eficiente, se han realizado dos estudios: el primero sobre los tiempos de ejecución y el segundo sobre el número de iteraciones necesarias para converger a π con una cierta precisión. Además, éste último nos permitirá comprobar si la velocidad de convergencia teórica se reproduce en la práctica. Los programas ejecutados para representar las gráficas que se exponen a continuación se encuentran al final del apéndice A.

2.7.1. Análisis de tiempos

Se han medido los tiempos de ejecución de los seis algoritmos para valores de la precisión empezando en 5.000 y aumentándola en 5.000 unidades hasta llegar a 200.000. Además, para cada valor, se han ejecutado cinco veces, tomando el tiempo medio. De esta forma, se reduce el error aleatorio y el efecto de posibles anomalías.

La primera gráfica muestra los tiempos medios obtenidos en función de la precisión utilizada para realizar los cálculos. En ella se diferencian claramente los algoritmos de Arquímedes y de Newton del resto. Con el paso de los años, la velocidad de convergencia aumenta drásticamente. De hecho, la fórmula de Machin y la de Ramanujan-Chudnovsky hacen que el tiempo baje del minuto para la precisión más alta. Sin embargo, ninguno supera a los programas de los hermanos Borwein y de Gauss (AGM por el uso de la media aritmético-geométrica). Éstos son tan rápidos que apenas se distingue del eje X. Por este motivo, se han ejecutado aparte para observar su comportamiento más de cerca y utilizando precisiones más altas, llegando hasta orden siete. A pesar de que el algoritmo AGM es de orden cuadrático, mientras que el de los Borwein es de orden cuártico, la segunda gráfica revela tiempos muy próximos entre ambos. Ello se debe a que cada iteración es computacionalmente menos costosa en el primer caso.

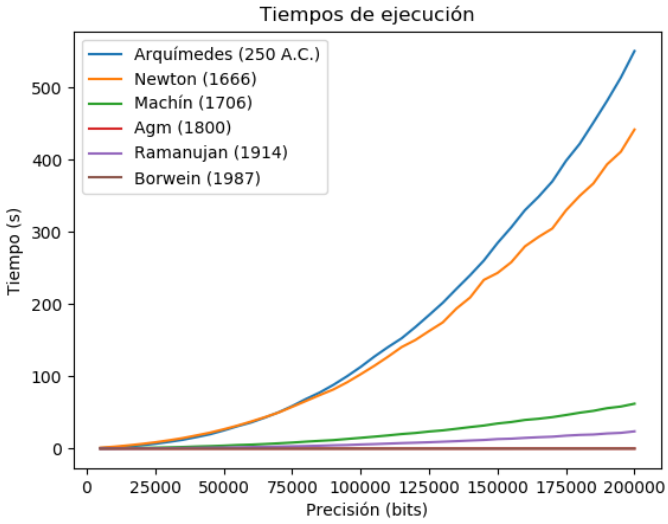

Figura 2.2: Tiempo de ejecución en segundos frente a la precisión de los cálculos.

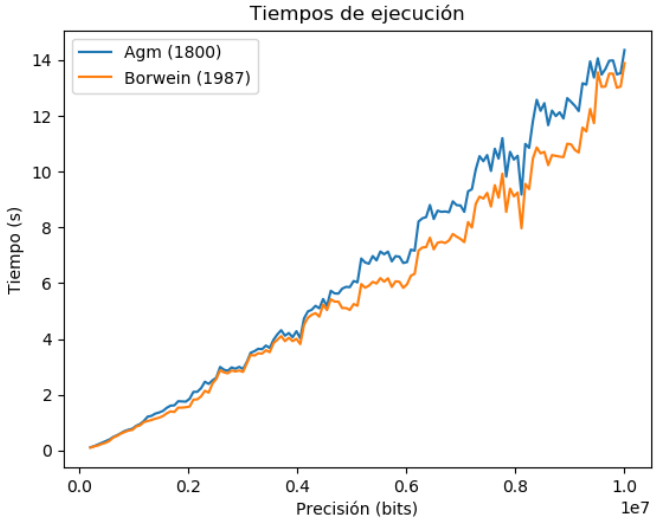

Figura 2.3: Comparación de los tiempos de ejecución entre el algoritmo de Gauss y el de los hermanos Borwein.

2.7.2. Análisis de iteraciones

Este estudio se ha realizado de manera similar al anterior y usando los mismos valores de la precisión. La tabla presenta la cantidad de iteraciones necesarias para hallar π correctamente hasta el bit indicado. Las gráficas muestran, por un lado, los datos para los seis algoritmos y, por otro, las iteraciones de los dos más rápidos donde se usan precisiones mayores para poder examinar su comportamiento detalladamente.

Precisión	Arquímedes	Newton	Machin	Gauss	Ramanujan	Borwein
10000	5000	4990	1077	13	213	7
20000	10000	9990	2153	14	425	7
30000	15000	14989	3230	14	638	7
40000	20000	19989	4306	15	850	8
50000	25000	24989	5383	15	1062	8
60000	30000	29988	6460	15	1275	8
70000	35000	34988	7536	15	1487	8
80000	40000	39988	8613	16	1699	8
90000	45000	44988	9690	16	1911	8
100000	50000	49988	10767	16	2124	8
110000	55000	54988	11843	16	2336	8
120000	60000	59988	12920	16	2548	8
130000	65000	64988	13997	16	2760	8
140000	70000	69988	15073	16	2973	8
150000	75000	74987	16150	17	3185	9
160000	80000	79987	17227	17	3397	9
170000	85000	84987	18303	17	3609	9
180000	90000	89987	19380	17	3822	9
190000	95000	94987	20457	17	4034	9
200000	100000	99987	21533	17	4246	9

Tabla 2.1: Datos obtenidos del número de iteraciones para alcanzar la correspondiente precisión en el cómputo de π.

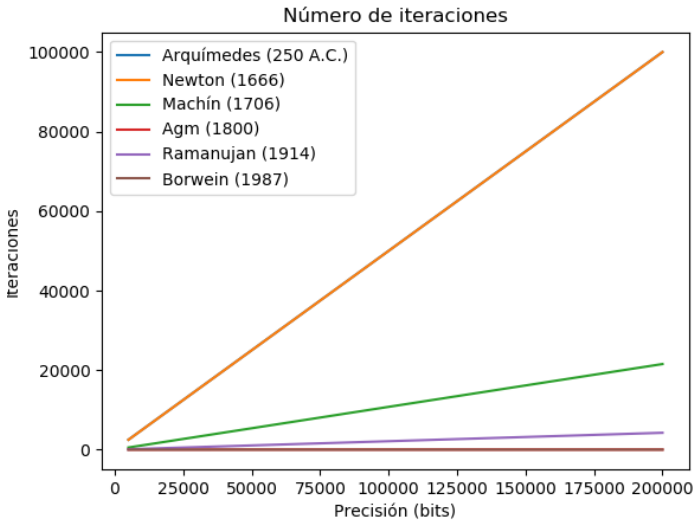

Figura 2.4: Número de iteraciones frente a la precisión de los cálculos de los seis algoritmos

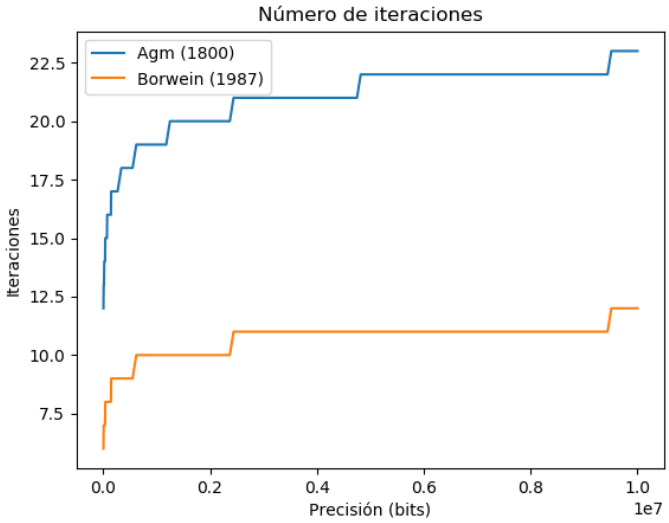

Figura 2.5: Comparación del número de iteraciones necesarias para alcanzar una determinada precisión entre el algoritmo de Gauss y el de los hermanos Borwein.

La primera gráfica exhibe un crecimiento lineal para los cuatro algoritmos más lentos. El de Arquímedes y el de Newton prácticamente se superponen, dado que el primero requiere de exactamente la mitad de iteraciones que la precisión, mientras que el segundo necesita una cantidad muy cercana, pero menor. Anteriormente, vimos que ambos tenían un orden de convergencia teórico de dos bits correctos por iteración, lo cual concuerda con esta información. En cambio, los programas creados a partir de la fórmula de Machin y de Ramanujan precisan de muchas menos iteraciones. El primero de ellos utiliza alrededor del $10{,}77\,\%$ con respecto a la precisión y el segundo tan sólo un $2{,}13\,\%$. Cantidades considerablemente menores que el $50\,\%$ de los dos primeros algoritmos.

En cuanto a los dos más rápidos, su crecimiento está lejos de ser lineal. En el caso del algoritmo de Gauss, cada número de iteraciones aparece en la tabla el doble de veces que el anterior, lo que indica un orden cuadrático. Además, este número duplica al de los hermanos Borwein como se esperaba. De hecho, esta afirmación se demuestra en el apéndice D. Por tanto, el orden de convergencia del segundo es cuártico.

3

Cálculo del n-ésimo dígito de π

3.1. Aspectos tecnológicos

El formato de doble precisión sólo permite el cálculo de π hasta unos pocos decimales. Por ello, los algoritmos requieren de una librería que permita realizar operaciones aritméticas con una cantidad grande de bits, para poder aproximarse con una precisión arbitrariamente alta. La librería escogida para este cometido es MPFR (Multiple Precision Floating-point Reliable library), codificada en C y basada en GMP (GNU Multiple Precision library), y su correspondiente interfaz para Python, el paquete bigfloat.

Los algoritmos del capítulo anterior se han implementado en Python 3, por su comodidad para realizar operaciones matemáticas entre datos numéricos de distintos tipos. En cambio, las de este capítulo se han programado en C, por ser un lenguaje de más bajo nivel y más adecuado para trabajar con bits.

Finalmente, los tests estadísticos del siguiente capítulo se han llevado a cabo en notebooks de Jupyter, mediante la aplicación Anaconda. Éstos permiten analizar datos y ejecutar cada test por separado en celdas diferentes, generando salidas visuales como tablas o gráficos.

3.2. Algoritmo PSLQ

Antes de comenzar con la exposición del método para obtener el n-ésimo dígito de π, vamos a exponer un algoritmo fundamental para la resolución de este problema: el algoritmo PSLQ. Considerado uno de los diez algoritmos más influyentes del siglo pasado por la revista *"Computing in science & engineering"* [5], fue desarrollado por H. R. P. Ferguson para detectar combinaciones lineales enteras de números reales. Es decir, dado un vector x formado por n números reales, busca un vector a de n enteros, tal que el producto escalar de ambos sea cero:

$$< a, x >= \sum_{i=0}^{\infty} a_i \times x_i = 0.$$

input : Un vector x de n componentes y un entero p
output: Un vector a de n componentes tal que $a \cdot x = 0$, si lo hubiera

inicializaciones;
M $\leftarrow 0$; $\gamma \leftarrow$ `Raiz2` (*4/3*); A, B $\leftarrow I_{n \times n}$; H $\leftarrow I_{n \times (n-1)}$; umbral $\leftarrow 10^{-p}$; over $\leftarrow 2^{np}$;
for $i \leftarrow 0$ **to** $n-1$ **do**
  s [i] $\leftarrow$ `Raiz2` ($\sum_{j=i}^{n-1} x[j]^2$); y [i] $\leftarrow$ x [i] / s [0]; s [i] $\leftarrow$ s [i] / s [0];
end
for $i \leftarrow 0$ **to** $n-1$ **do**
  if $i < n-2$ **then**
    H [ii] $\leftarrow$ s [j+1] / s [j];
  end
  for $j \leftarrow 0$ **to** $i-1$ **do**
    H [ij] $\leftarrow$ - y [i] y [j] / (s [j] s [j+1]);
  end
end
for $i \leftarrow 2$ **to** n **do**
  for $j \leftarrow i-1$ **to** 0 **do**
    t $\leftarrow$ `Nint` (H *[ij]* / H *[jj]*); y [j] $\leftarrow$ y [j] + t y [i];
    for $k \leftarrow 0$ **to** j **do**
      H [ik] $\leftarrow$ H [ik] – t H [jk];
    end
    for $k \leftarrow 0$ **to** $n-1$ **do**
      A [ik] $\leftarrow$ A [ik] – t A [jk]; B [kj] $\leftarrow$ B [kj] + t A [kj];
    end
  end
end
iteración;
while `Maximo` (*A*) < over $\wedge$ *1 /* $Maximo_i$ (`Abs` (*H [ii]*)) < 100 **do**
  m $\leftarrow i$ tal que $Maximo_i$ (γ^i `Abs` (H *[ii]*));
  `Swap` (*y [m], y [m+1]*); `Swap` (*A [fila-m], A [fila-(m+1)]*);
  `Swap` (*H [fila-m], H [fila-(m+1)]*); `Swap` (*B [col-m], B [col-(m+1)]*);
  if $m < n-2$ **then**
    t0 $\leftarrow$ `Raiz2` (H *[mm]*^{2} + H *[m,m+1]*^{2}); t1 $\leftarrow$ H [mm] / t0; t2 $\leftarrow$ H [m,m+1] / t0;
    for $i \leftarrow$ m **to** $n-1$ **do**
      t3 $\leftarrow$ H [im]; t4 $\leftarrow$ H [i,m+1]; H [im] $\leftarrow$ t1 t3 + t2 t4; H [i,m+1] $\leftarrow$ t1 t4 – t2 t3;
    end
  end
  for $i \leftarrow$ m + 1 **to** $n-1$ **do**
    for $j \leftarrow$ `Minimo` (*i-1,* m *+1*) **to** 0 **do**
      t $\leftarrow$ `Nint` (H *[ij]* / H *[jj]*); y [j] $\leftarrow$ y [j] + t y [i];
      for $k \leftarrow 0$ **to** j **do**
        H [ik] $\leftarrow$ H [ik] – t H [jk];
      end
      for $k \leftarrow 0$ **to** $n-1$ **do**
        A [ik] $\leftarrow$ A [ik] – t A [jk]; B [kj] $\leftarrow$ B [kj] + t A [kj];
      end
    end
  end
  if $Minimo_i$ (*y [i]*) < umbral **then**
    return B *[col-i]*;
  end
end

Algoritmo 3.1: Algoritmo PSLQ

Obviamente, esta ecuación tiene una solución trivial que es el vector nulo ($a = \overrightarrow{0}$). Además, si una de las componentes x_i es cero, habría un conjunto infinito de soluciones dado por $\{a \in \mathbf{R}^n : a_i \in \mathbf{R}, a_{j\neq i} = 0\}$. Por ello, se impone como condición extra que todos los elementos de x y al menos uno de a sean no nulos.

La fórmula anterior se satisface si y sólo si los datos a y x, vistos como vectores del espacio $\mathbf{R}^n$, son ortogonales (forman un ángulo de $90°$ entre ellos). No es complicado encontrar una recta perpendicular a otra dada. La dificultad radica en que la primera debe estar generada por un vector de enteros, lo cual no es siempre posible. Por ejemplo, tomando $x = (\pi, m)$ en el plano, con m un número natural cualquiera, $(-m, \pi)$ cumple la igualdad anterior, pero no es lo que buscamos. En este caso no hay solución posible, porque π tendría que ser racional.

Hay una gran cantidad de teoría algebraica detrás de este algoritmo. De hecho, la denominación PSLQ proviene del uso de un vector de suma-de-cuadrados parciales (Partial Sum-of-squares) y una factorización matricial ortogonal triangular inferior (LQ matrix factorization). En el pseudocódigo, el vector es la variable s y la matriz triangular H (de tamaño $n \times (n-1)$) se inicializa a la matriz identidad. Al igual que las matrices A y B, éstas de tamaño $n \times n$. Asimismo, el algoritmo requiere que el vector x tenga módulo uno. Por este motivo, al principio se crea el vector unitario y, fruto de dividir cada componente de x por su módulo. El algoritmo finaliza con éxito cuando una componente del vector y, pongamos y_i, es menor que un umbral proporcionado, en cuyo caso el vector a buscado se encuentra en la columna i-ésima de B. En cambio, si algún componente del vector A supera la precisión numérica, el algoritmo falla. Además, el valor $\frac{1}{\text{máx}_{0\leq i\leq n-2}|H_{ii}|}$ es una cota inferior del módulo de a, por lo que si se dispara es un indicio de que no existe solución.

3.3. Fórmula BBP

Durante el siglo XX, se creía que para conseguir el n-ésimo dígito de π, era necesario hallar todas las cifras previas. Esta sensación estaba generalizada sobre todas las constantes irracionales. Sin embargo, se descubrió la siguiente fórmula para el logaritmo natural de dos que arrojó luz sobre este problema:

$$\ln(2) = \sum_{j=1}^{\infty} \frac{1}{j2^j}$$

A partir de ella, se puede obtener el $(n+1)$-ésimo número del desarrollo en binario de $\ln(2)$. Denotando $\{x\}$ a la parte fraccionaria de x, el problema equivale a calcular $\{2^n \ln(2)\}$:

$$\{2^n \ln(2)\} = \left\{\sum_{j=1}^{n} \left\{\frac{2^{n-j}}{j}\right\}\right\} + \left\{\sum_{j=n+1}^{\infty} \frac{2^{n-j}}{j}\right\} = \left\{\sum_{j=1}^{n} \frac{2^{n-j} \bmod j}{j}\right\} + \left\{\sum_{j=n+1}^{\infty} \frac{2^{n-j}}{j}\right\}$$

En el primer paso, se ha utilizado la propiedad $\{x+y\} = \{\{x\}+\{y\}\}$ de manera recursiva y en el segundo, el hecho de que $\{x/y\} = (x \bmod y)/y$. El n-ésimo dígito se puede computar de manera eficiente, utilizando el algoritmo de la exponenciación modular para el numerador de la primera serie, el cual se detalla más adelante. En cuanto a la segunda serie, sólo es necesario sumar unos pocos términos para asegurarse de que el redondeo es correcto.

Este hallazgo abrió las puertas a la posibilidad de encontrar una fórmula similar para otras constantes matemáticas, que permitieran aplicar el mismo proceso para obtener la n-ésima cifra. Aquí es donde entra en juego el algoritmo PSLQ. Si éste se ejecutara con el vector $x = (\sum_{j=1}^{\infty} \frac{1}{j2^j}, \ln(2))$ con una precisión suficientemente alta, el algoritmo convergería al vector $a = (1, -1)$. En general, para obtener una solución de n componentes con un máximo de d dígitos cada una, se necesita trabajar con una precisión aritmética de (al menos) $n \times d$ dígitos.

La idea de David Bailey, Peter Borwein y Simon Plouffe fue realizar una exploración exhaustiva y ejecutar el algoritmo PSLQ con una gran cantidad de vectores conteniendo π. Finalmente, el esfuerzo tuvo su recompensa y dieron con la fórmula BBP, nombrada así en honor a sus descubridores [6]:

$$\sum_{j=0}^{\infty} \frac{1}{16^j}\left(\frac{4}{8j+1} - \frac{2}{8j+4} - \frac{1}{8j+5} - \frac{1}{8j+6}\right) - \pi = 0 \tag{3.1}$$

En efecto, si se ejecutase el algoritmo PSLQ, cuyo código se encuentra en el apéndice C junto con los de este capítulo, con el vector $x = (x_1, ..., x_8, \pi)$ con $x_i = \sum_{j=0}^{\infty} \frac{1}{16^j}\left(\frac{1}{8j+i}\right)$, obtendríamos como salida el vector $a = \pm(4, 0, 0, -2, -1, -1, 0, 0, -1)$. Es decir, daría los coeficientes de cada elemento del vector x de la ecuación anterior. Hay que tener en cuenta que PSLQ no es 100 % fiable. Simplemente proporciona una posible solución, que es más probable que sea correcta cuanto menor es el umbral. Por ello, se incluyen en el apéndice D las demostraciones de esta fórmula y de la encontrada por Bellard, que se detalla en la siguiente sección.

Con esta nueva identidad, se puede hallar el $(n+1)$-ésimo dígito de π en hexadecimal, siguiendo el mismo esquema que para el $\ln(2)$:

$$\{16^n\pi\} = \left\{4\left\{\sum_{j=0}^{\infty}\frac{16^{n-j}}{8j+1}\right\} - 2\left\{\sum_{j=0}^{\infty}\frac{16^{n-j}}{8j+4}\right\} - \left\{\sum_{j=0}^{\infty}\frac{16^{n-j}}{8j+5}\right\} - \left\{\sum_{j=0}^{\infty}\frac{16^{n-j}}{8j+6}\right\}\right\},$$

donde cada una de las series del lado derecho se calcula de la siguiente manera:

$$\left\{\sum_{j=0}^{\infty}\frac{16^{n-j}}{8j+i}\right\} \approx \left\{\left\{\sum_{j=0}^{n}\frac{16^{n-j} \bmod (8j+i)}{8j+i}\right\} + \sum_{j=n+1}^{N}\frac{16^{n-j}}{8j+i}\right\}$$

El dato N es un número ligeramente mayor que n y se utiliza para evitar errores de redondeo. El resultado de esta operación es el número π en base hexadecimal empezando desde la posición $n+1$. De nuevo, se ha introducido la exponenciación modular porque es una forma más eficiente

de elevar un número cuando el exponente es natural. Calcular el valor de a^b mediante el método usual requiere de b productos. En cambio, en la exponenciación binaria sólo hacen falta $\log_2(b)$ multiplicaciones. Además, se puede reducir el resultado de cada producto a través del módulo, evitando que crezca más allá de un cierto valor y permitiendo usar una precisión aritmética más baja, lo que agiliza la computación. Escribiendo el exponente b en base dos, hallar la potencia de un número a módulo m equivale a:

$$a^b \bmod m = a^{\sum_{j=0}^{n} c_j 2^j} \bmod m = \prod_{j=0}^{n} \left(a^{(2^j)}\right)^{c_j} \bmod m = \prod_{j=0}^{n} \left(a^{(2^j)} \bmod m\right)^{c_j} \bmod m.$$

Esta forma de calcular el resultado, recorriendo el exponente desde el bit menos significativo al más significativo, es la exponenciación binaria de derecha a izquierda. Esto significa que existe el algoritmo de izquierda a derecha, el cual proviene de la siguiente iteración:

$$a^b = a^{2*\lfloor b/2 \rfloor} * a^{c_0} = p_n, \text{ donde } p_k = a^{c_{n-k}} p_{k-1}^2 \text{ y } p_0 = 1.$$

```
input : Tres enteros positivos a, b y m
output: El resultado de elevar a con potencia b, módulo m
1  inicializaciones;
2  r ← 1;
3  iteración;
4  while b > 0 do
5      if (b & 1) > 0 then
6          r ← (r × b) % m;
7      end
8      b ← Int(b/2);
9      a ← (a × a) % m;
10 end
11 return r;
```

Algoritmo 3.2: Exponenciación binaria modular de derecha a izquierda

```
input : Tres enteros positivos a, b y m
output: El resultado de elevar a con potencia b = (c_n c_{n-1} ... c_0)_2, módulo m
1  inicializaciones;
2  r ← 1;
3  iteración;
4  for i ← n to 0 do
5      if (c_i > 0) then
6          r ← (r × b) % m;
7      end
8      r ← (r × r) % m;
9  end
10 return r;
```

Algoritmo 3.3: Exponenciación binaria modular de izquierda a derecha

Finalmente, encontrar el n-ésimo dígito se reduce a desarrollar las cuatro series de la fórmula BBP, donde cada término se calcula utilizando el algoritmo anterior:

```
input : Dos enteros positivos i, n
output: El resultado de la serie con denominador 8j + i hasta al menos n sumandos
1  inicializaciones;
2  N ← n + 100; r ← 0;
3  iteración;
4  for j ← i to n do
5  |  r ← r + ExpModular( 16, n − j, 8 j + i ) / ( 8 j + i );
6  end
7  r ← r − Int( r );
8  for j ← n + 1 to N do
9  |  r ← r + Power( 16, n − j ) / ( 8 j + i );
10 end
11 return r;
```

Algoritmo 3.4: Cálculo de las series de la fórmula BBP

```
input : Un entero positivo n
output: El (n+1)-ésimo dígito hexadecimal de π
1  inicializaciones;
2  s1 ← Serie( 1, n );
3  s2 ← Serie( 4, n );
4  s3 ← Serie( 5, n );
5  s4 ← Serie( 6, n );
6  r ← 4 × s1 − 2 × s2 − s3 − s4;
7  r ← ToHex( r − Int( r ) );
8  return r;
```

Algoritmo 3.5: Algoritmo de la fórmula BBP

3.4. Fórmula de Bellard

Poco tiempo después del descubrimiento de la fórmula BBP, Fabrice Bellard consiguió encontrar otra ecuación similar, pero que mejora la eficiencia del cálculo [7]:

$$\sum_{j=0}^{\infty} \frac{(-1)^j}{2^{10j}} \left(-\frac{2^{-1}}{4j+1} - \frac{2^{-6}}{4j+3} + \frac{2^2}{10j+1} - \frac{2^0}{10j+3} - \frac{2^{-4}}{10j+5} - \frac{2^{-4}}{10j+7} + \frac{2^{-6}}{10j+9} \right) - \pi = 0 \tag{3.2}$$

Repitiendo el proceso seguido con la fórmula BBP, el $(n+1)$-ésimo dígito hexadecimal de π se puede obtener calculando la parte entera de cada serie por separado [8]:

$$\{16^n \pi\} = -\{16^n S(4,1,-1,n)\} - \{16^n S(4,3,-6,n)\} - \{16^n S(10,1,2,n)\} - \{16^n S(10,3,0,n)\} - \{16^n S(10,5,-4,n)\} - \{16^n S(10,7,-4,n)\} - \{16^n S(10,9,-6,n)\}$$

Cada uno de los sumatorios es una función de cuatro parámetros:

$$\{16^n S(a,b,e,n)\} = \left\{ 2^{4n} \sum_{j=0}^{\infty} \frac{(-1)^j}{2^{10j}} \frac{2^e}{aj+b} \right\} = \left\{ \sum_{j=0}^{\infty} (-1)^j \frac{2^{4n+e-10j}}{aj+b} \right\}$$
$$\approx \left\{ \left\{ \sum_{j=0}^{\lfloor \frac{4n+e}{10} \rfloor} (-1)^j \frac{2^{4n+e-10j} \bmod (aj+b)}{aj+b} \right\} + \sum_{j=\lfloor \frac{4n+e}{10} \rfloor + 1}^{N} (-1)^j \frac{2^{4n+e-10j}}{aj+b} \right\}$$

De nuevo, N es un número que asegura la corrección del primer dígito hexadecimal. La diferencia principal con respecto a la fórmula BBP es que, a pesar de que hay que calcular más series, cada una de ellas tiene sólo alrededor de $\lfloor 4n/10 \rfloor$ sumandos, donde $\lfloor x \rfloor$ denota la parte entera del dato x. Esto provoca que este método sea, alrededor de un 43 % más rápido, según el propio Bellard.

Haciendo uso de la exponenciación modular descrita en la sección anterior, implementar el algoritmo de Bellard es muy similar al BBP:

```
   input : Cuatro números enteros a, b, e, n
   output: El resultado de la serie con numerador 2^(4n+e-10j) y denominador aj + b
1  inicializaciones;
2  index ← Int( ( 4 n + e ) / 10 ); n ← index + 100; signo ← 1; r ← 0;
3  iteración;
4  for j ← 0 to index do
5  |  r ← r + signo × ExpModular( 16, 4 n + e − 10j, a j + b ) / ( a j + b );
6  |  signo ← signo × ( −1 );
7  end
8  r ← r − Int( r );
9  for j ← index + 1 to N do
10 |  r ← r + signo × Power( 16, 4 n + e − 10j ) / ( a j + b );
11 |  signo ← signo × ( −1 );
12 end
13 return r;
```

Algoritmo 3.6: Cálculo de las series de la fórmula de Bellard

```
   input : Un entero positivo n
   output: El (n+1)-ésimo dígito hexadecimal de π
1  inicializaciones;
2  s1 ← Serie( 4, 1, -1, n ); s2 ← Serie( 4, 3, -6, n );
3  s3 ← Serie( 10, 1, 2, n ); s4 ← Serie( 10, 3, 0, n );
4  s5 ← Serie( 10, 5, -4, n ); s6 ← Serie( 10, 7, -4, n );
5  s7 ← Serie( 10, 9, -6, n );
6  r ← × s3 + s7 − s1 − s2 − s4 − s5 − s6;
7  r ← ToHex( r − Int( r ) );
8  return r;
```

Algoritmo 3.7: Algoritmo de la fórmula de Bellard

3.5. Comparación entre BBP y la fómula de Bellard

Anteriormente hemos mencionado que el cálculo del n-ésimo dígito de π en hexadecimal es aproximadamente un 43 % más rápido mediante la fórmula de Bellard frente a la fórmula BBP. En este apartado vamos a comprobar esta afirmación.

El experimento consiste en medir los tiempos de ejecución de sus respectivos códigos, los cuales se pueden ver en el apéndice C. El único parámetro que reciben es la posición del dígito que se quiere obtener. Como se explica en el apéndice, la implementación del algoritmo BBP es una optimización de la versión de David Bailey, quien comenta que proporciona la cifra correcta hasta una posición ligeramente superior a 10^7. Por esta razón, no hemos cronometrado las ejecuciones más allá de este valor.

Los tiempos se han determinado para posiciones empezando desde cero y aumentándolo en 50.000 (en total, 200 mediciones por algoritmo). De nuevo, cada una de ellas se ha repetido cinco veces para evitar errores. La gráfica siguiente muestra la media de los tiempos obtenidos en función de la posición del dígito:

Figura 3.1: Comparación de la eficiencia del algoritmo BBP y Bellard

El algoritmo de Bellard es visiblemente más rápido que BBP. Curiosamente, ambos tienen una complejidad temporal lineal porque las respectivas regresiones lineales se ajustan muy bien a los datos con coeficientes de determinación (R^2) próximos a uno. Esto significa que la proporción entre sus tiempos de ejecución es constante e igual al cociente de las pendientes:

$$\frac{T_{Bellard}}{T_{BBP}} \approx \frac{1{,}80768 * 10^{-6}}{1{,}07075 * 10^{-6}} \approx 59\,\%$$

Es decir, el algoritmo de Bellard es aproximadamente un 41 % más eficiente que BBP, lo cual se acerca a la afirmación que queríamos comprobar.

4

ESTUDIO DE LA NORMALIDAD DE π

Un número irracional es simplemente normal en una base b si las cifras entre 0 y $b-1$ están uniformemente distribuidas en su parte fraccionaria. Es decir, si se cumple el siguiente valor asintótico:

$$\lim_{n\to\infty} \frac{N(d,n)}{n} = \frac{1}{b}, \text{ para cada } d \in \{0,...,b-1\}.$$

La función N devuelve la cantidad de ocurrencias de cada dígito d en los primeros n dígitos decimales. La normalidad extiende esta definición a secuencias de k dígitos. Un número irracional es normal en una base b si para cada natural k las posibles cadenas de k cifras aparecen con una frecuencia límite b^{-k} [1]:

$$\lim_{n\to\infty} \frac{N(s,n)}{n} = b^{-k}, \text{ para todo } k \in \mathbf{N} \text{ y para cada secuencia } s \text{ de } k \text{ cifras.}$$

Esta propiedad es un intento de definir de manera formal la aleatoriedad de un número, aunque se queda algo corto. Una secuencia aleatoria debe satisfacer diversas cualidades, entre las que se encuentra la normalidad. No obstante, un número normal no necesariamente es aleatorio. Por ejemplo, la constante de Champernowne, cuya parte fraccionaria está formada por la concatenación de los números naturales en orden, es normal en base 10. Sin embargo, no es aleatorio, porque todos los dígitos situados en posiciones 10^n, para n natural, son unos.

En realidad, no hay ningún método que asegure si una secuencia es totalmente aleatoria o no. Pero sí podemos realizar una serie de test estadísticos que refuten o acepten la hipótesis de aleatoriedad con un determinado nivel de confianza. Es decir, si la secuencia pasa todos los test es altamente probable que sea aleatoria y, por tanto, normal.

Éste es el procedimiento que utilizaremos para analizar la normalidad del número π. Lógicamente, es imposible examinar la secuencia completa, pero estudiando sus primeras cifras en distintas bases lograremos decidir si la hipótesis de aleatoriedad es viable. En este capítulo se presentan la metodología del experimento y los resultados obtenidos de diversos test, los cuales se han seleccionado del libro *Seminumerical algorithms* de Knuth [9] y del libro *100 statistical tests* de Kanji [10]. Los códigos ejecutados y los datos producidos se encuentran en el apéndice E.

4.1. Obtención de los datos

Los test estadísticos se ejecutarán sobre los primeros dígitos de π en binario, decimal y hexadecimal. A modo de ejemplo, en el apéndice F se pueden ver sus primeras cifras en estas tres bases. En el segundo capítulo vimos que la fórmula de los hérmanos Borwein (algoritmo 2.7) es muy eficiente a la hora de computar π en base diez. Para los otros dos casos, usaremos la fórmula de Bellard (algoritmo 3.7). Este programa proporciona al menos ocho dígitos hexadecimales correctos a partir de la posición n, por lo que se concatenan los resultados de su ejecución para posiciones múltiplos de ocho. Analizando la gráfica 3.1, el tiempo necesario para obtener los primeros k hexadecimales mediante este procedimiento es aproximadamente un octavo del área bajo la curva desde cero hasta k:

$$T(k) = \frac{1}{8}\int_0^k 1{,}0705 * 10^{-6}x - 0{,}205dx = \frac{5{,}3525 * 10^{-7}k^2 - 0{,}205k}{8}$$

Probando con diferentes valores de k, vemos que para dos millones se necesitarían alrededor de dos días y medio, lo cual es una cantidad suficiente de dígitos para nuestro estudio y un tiempo razonable.

Tras generar las cifras en base 16, se escribieron en veinte ficheros distintos, cada uno con cien mil dígitos. Los motivos son manejar ficheros más ligeros para acceder a ellos más rápidamente y realizar los test de manera progresiva para comparar la evolución de los resultados.

A partir de estos ficheros, conseguir las primeras ocho millones de cifras binarias consiste en leer cada carácter hexadecimal de ellos y transformarlos en su equivalente secuencia binaria de cuatro dígitos, generando así otros veinte archivos, esta vez con 400 mil números:

Base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Base 2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Tabla 4.1: Las representaciones en binario, decimal y hexadecimal de cada número

Por último, sólo falta generar π en base diez. Una precisión de dos millones de cifras hexadecimales equivalen a aproximadamente 2.4 millones en decimal. Por tanto, se ejecuta el algoritmo de Borwein con una precisión de ocho millones de bits y se divide en trozos de 120 mil dígitos, cada uno de los cuales se escribe en un nuevo fichero.

Los siguientes test estadísticos se realizan sobre la expansión de π en las tres bases gradualmente. Se comienza analizando el respectivo primer fichero de los veinte de cada base. Después, en la segunda etapa se suman los resultados de los segundos archivos. Se repite este proceso hasta haber analizado las tres secuencias por separado. Esto nos permitirá ver la evolución de los resultados, esperando a que se hagan paulatinamente más uniformes.

4.2. Distribuciones de probabilidad

4.2.1. Distribución normal

La distribución de Gauss o normal es una de las más frecuentes en probabilidad. Se construye a partir de dos parámetros: la media y la varianza. La distribución estándar tiene media cero y varianza uno. No obstante, una variable aleatoria normal Z con datos genéricos μ y σ se puede estandarizar:

$$X = \frac{Z - \mu}{\sqrt{\sigma}} \sim \mathcal{N}(0, 1).$$

En un test estadístico basado en esta distribución, tras analizar los datos se obtiene un valor para Z. Después, hay que estandarizarlo mediante la fórmula anterior para poder aceptar o rechazar nuestra hipótesis:

$p = 0{,}90$	$p = 0{,}95$	$p = 0{,}975$	$p = 0{,}99$
$1{,}28$	$1{,}645$	$1{,}96$	$2{,}33$

Tabla 4.2: Valores de la distribución normal según el nivel de confianza p

Los valores indican que la probabilidad de que X sea superior a los mostrados es menor que $1 - p$. En la gráfica siguiente, sería el número a partir del cual, el área bajo la curva es 0.05. En nuestro caso, utilizaremos un nivel de confianza del 95 %, por lo que si el dato observado es mayor que $1{,}645$, se rechazará la hipótesis con un error menor del 5 %. El intervalo de confianza (IC) al p % es el intervalo del eje X centrado alrededor de la media que acumula el p % del área. Por ejemplo, el 95 % IC de la normal estándar sería $(-1{,}96, 1{,}96)$.

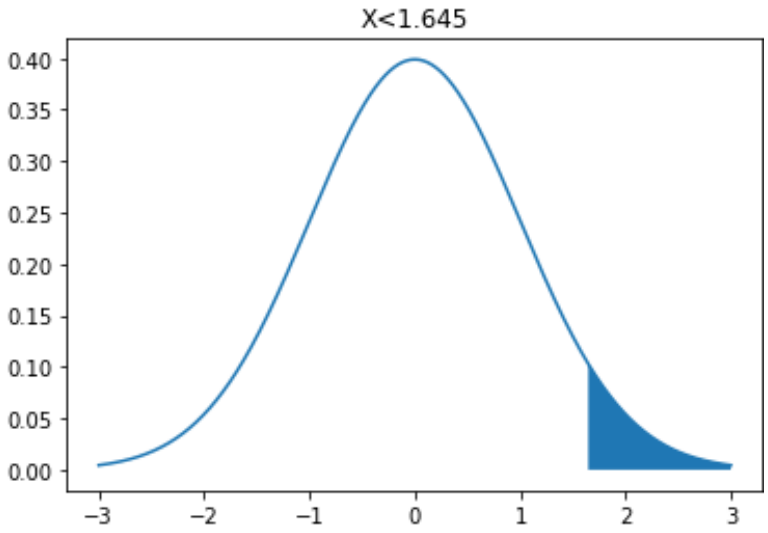

Figura 4.1: Función de densidad de la distribución normal

4.2.2. Distribución chi-cuadrado (χ^2)

La mayoría de los test estadísticos se basan en la distribución de Pearson, conocida como chi-cuadrado (χ^2). Dadas k variables aleatorias independientes que se comportan como una normal estándar $(\mathcal{N}(0,1))$, la suma de sus cuadrados se aproximan a una χ^2 con k grados de libertad:

$$X = \sum_{i=1}^{k} {Z_i}^2 \sim {\chi_k}^2, \text{ donde } Z_i \sim \mathcal{N}(0,1).$$

En nuestro caso, en cada test se realizará un determinado número n de observaciones, los cuales tienen una probabilidad p_i de pertenecer a la categoría i-ésima de las k posibles. Llamando Y_i a la variable aleatoria relativa a la cantidad de observaciones que han caído en la clase i, Z_i se puede definir en función de ella:

$$Z_i = \frac{Y_i - np_i}{\sqrt{np_i}} \sim \mathcal{N}(0,1).$$

Por tanto, la variable inicial, que hemos denotado como X, equivale a:

$$X = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i} \sim {\chi_{k-1}}^2 \tag{4.1}$$

Nótese que esta vez sólo hay $k-1$ grados de libertad. Esto se debe a que, al haber un número fijo n de observaciones, el valor de las $k-1$ primeras categorías determinan el valor de la k-ésima, puesto que la suma de todas ellas debe ser igual al total: $\sum_{i=1}^{k} Y_i = n$.

Una vez realizadas todas las observaciones, se computa el valor de X mediante la ecuación anterior. Este valor determinará si se acepta o se rechaza la hipótesis de aleatoriedad. Para ello, utilizaremos la siguiente tabla:

Grados de libertad (k)	Valor (${\chi_k}^2$)
1	3.84
2	5.99
3	7.815
4	9.49
5	11.07
9	16.92
15	25.00
99	123.23
255	293.25

Tabla 4.3: Valores de la distribución chi-cuadrado al 95 % de nivel de confianza

Al igual que la distribución normal, los valores indican que la probabilidad de que X sea superior es menor del 5 % y, en ese supuesto, se rechazaría la hipótesis. En este caso, la gráfica de la función de densidad no es simétrica y depende del parámetro k:

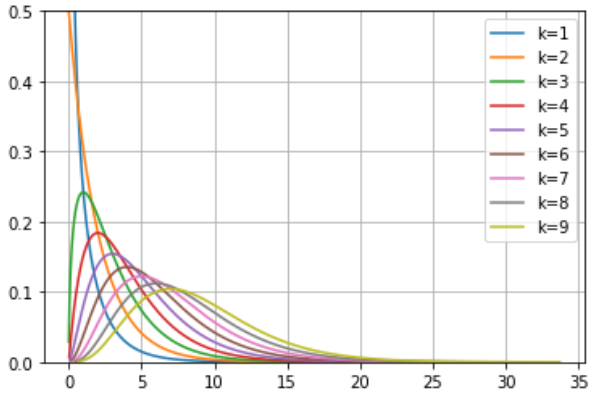

Figura 4.2: Función de densidad de la distribución chi-cuadrado

4.3. Test de frecuencias

Este test sirve para comprobar la normalidad simple definida al principio del capítulo. Se empieza contando el número de ocurrencias Y_i de cada posible cifra i entre 0 y $b-1$, donde b denota la base. Usando que la probabilidad p_i de cada categoría es idéntica para todas ellas e igual a $1/b$ y la cantidad de observaciones n es la cuantía de dígitos analizados tras cada etapa, se halla el valor de la función χ^2. Este dato se compara con el de la fila correspondiente a $b-1$ grados de libertad de la tabla 4.3. En la siguiente figura que muestra los resultados obtenidos, se destacan en rojo los valores superiores al de la tabla. En este caso concreto, esto significa que el test rechaza la hipótesis de aleatoriedad para los primeros 1.8 y 1.9 millones de cifras binarias. La columna "Núm. dígitos" indica la cantidad de hexadecimales analizados empezando siempre desde el primero.

Núm. dígitos	Base 2	Base 10	Base 16	Núm. dígitos	Base 2	Base 10	Base 16
100000	2.209	6.14699999	8.82111999	1100000	0.71524454	7.48689393	9.99095272
200000	0.2645	9.53841666	9.76464000	1200000	0.90480333	4.49202777	11.2173333
300000	0.17480333	9.90961111	14.5847466	1300000	0.91224692	6.29673076	10.0803938
400000	0.25921	7.430875	8.11512000	1400000	1.15388642	6.85	9.69792000
500000	0.1458	8.57029999	10.8159359	1500000	1.80621066	7.09441111	9.72290133
600000	0.20768166	9.32700000	8.84730666	1600000	2.3668225	7.38689583	10.68592
700000	1.07384142	5.17019047	10.7054628	1700000	2.74193	8.97390196	12.6399999
800000	1.48785125	5.55279166	10.36096	1800000	4.25656888	10.4174814	11.9038755
900000	0.88209	6.76362962	8.73208888	1900000	3.96580263	9.91735964	9.69306947
1000000	0.334084	6.74341666	8.430464	2000000	3.803282	10.0617916	11.473312

Figura 4.3: Resultados del test de frecuencias

4.4. Test serial

En el test anterior examinamos la frecuencia de cada dígito. En éste vamos a realizar el mismo proceso pero considerando parejas de cifras. Por tanto, la cantidad de observaciones se reduce a la mitad y el número de categorías se eleva al cuadrado. Es decir, hay b^2 posibilidades equiprobables, lo que implica que $p_i = 1/b^2$. La fórmula 4.1 produce los siguientes datos, los cuales se han comparado con respecto al valor de χ^2 con $b^2 - 1$ grados de libertad:

Núm. dígitos	Base 2	Base 10	Base 16	Núm. dígitos	Base 2	Base 10	Base 16
100000	2.66212	124.359999	244.229120	1100000	2.18479636	100.230000	239.060712
200000	0.72298	114.930000	259.527680	1200000	2.94598333	91.0672222	247.522986
300000	0.87828000	134.231111	242.095786	1300000	3.89659384	84.2256410	262.349193
400000	1.88900000	114.776666	219.13344	1400000	3.53618	87.3383333	265.623405
500000	2.257488	110.052666	246.309887	1500000	2.75681066	87.0555555	264.973311
600000	2.18319333	122.122777	252.434773	1600000	3.61072500	84.4893750	274.91456
700000	2.05770857	109.465714	251.985188	1700000	4.33985647	89.6690196	269.681543
800000	3.432195	97.97375	232.064	1800000	5.76935333	90.0179629	260.068693
900000	2.74578222	103.129629	240.75776	1900000	4.75130736	86.3905263	261.826829
1000000	1.891592	99.0023333	242.947072	2000000	4.58707000	87.6721666	266.057728

Figura 4.4: Resultados del test serial

4.5. Test de la mano de poker

Los dos test anteriores son muy útiles pero no suficientes para poder asegurar la aleatoriedad de una secuencia por ellos mismos. Por ejemplo, una secuencia que repita indefinidamente el periodo $00, 01, 02, ..., 97, 98, 99$ no es nada aleatoria. Sin embargo, los pasaría perfectamente, dado que cada cifra y cada pareja aparece exactamente el mismo número de veces. Por este motivo, también es importante examinar los dígitos por grupos en busca de algún patrón o relación interna.

En el juego de cartas del póker, se reparten cinco cartas a cada jugador y gana aquél que tenga una mejor combinación según su probabilidad: escalera real (5), escalera de color (5), póker (2), full (2), color (5), escalera (5), trío (3), doble pareja (3), pareja (4), carta más alta (5). El dato entre paréntesis indica la cantidad de cartas con distinto número que conforma dicha mano. Tomando cada dígito como una carta sin palo, cada jugada de póker (excluyendo las referentes al color de las cartas) debería aparecer con una frecuencia cercana a su probabilidad, si realmente fuese aleatoria.

No obstante, se puede simplificar este proceso contando cuántos dígitos distintos hay en cada grupo de cinco, en vez de enumerar cuántas veces aparece cada mano. Es decir, las combinaciones con el mismo número asociado se consideran equivalentes. Además, habría que añadir la

posibilidad de que se repita la misma cifra cinco veces, lo cual es imposible en el póker. Por otra parte, este planteamiento permite generalizar a grupos de cualquier tamaño.

En nuestro caso, hemos dividido la secuencia en tuplas de cuatro y cinco dígitos consecutivos. Por tanto, la cuantía de observaciones se reduce a la cuarta y quinta parte del número total de cifras, respectivamente. Cuanto mayor es el tamaño de los grupos, menor es el conjunto de datos disponible y los resultados pueden dejar de ser fiables, por lo que nos mantendremos en estos valores. Las categorías se corresponden con el número i de cifras distintas en la tupla, cuyas probabilidades p_i vienen dadas por:

$$p_i = \frac{b(b-1)...(b-r+1)}{b^k} \left\{ {k \atop i} \right\}, k = 4 \text{ ó } 5$$

Las llaves denotan el número de Stirling de segunda especie y toman los siguientes valores:

k \ i	1	2	3	4	5
4	1	7	6	1	
5	1	15	25	10	1

Tabla 4.4: Números de Stirling de segunda especie

Para este test en particular, la secuencia en base dos se ha excluido porque únicamente hay dos posibilidades, o todas las cifras son iguales o no. Además, la primera de ellas es muy poco probable en comparación con la otra, puesto que sólo ocurre una de cada 16 o 32 tuplas, en función de si éstas son de cuatro o cinco dígitos, respectivamente. En cuanto a las bases decimal y hexadecimal, con estos datos se calcula el valor de X con la fórmula 4.1 y se compara con el valor en la tabla correspondiente a $k-1$ grados de libertad:

Núm. dígitos	Base 10	Base 16	Núm. dígitos	Base 10	Base 16
100000	0.65363756	2.48365198	1100000	3.55572089	1.06667229
200000	3.51923500	4.87710694	1200000	3.80835354	0.63341214
300000	7.32535640	0.64628799	1300000	1.56725512	0.26555949
400000	10.9078400	0.40568404	1400000	1.92534328	0.65127703
500000	12.3372641	0.39256196	1500000	2.17946061	0.82591268
600000	11.5952950	0.68576784	1600000	1.42588527	1.47037585
700000	8.71482111	0.32940408	1700000	0.93275625	0.96429516
800000	6.40381668	0.26399445	1800000	1.06110070	1.16710507
900000	6.26843890	1.32599768	1900000	1.49516499	1.27887910
1000000	6.80002645	1.25147878	2000000	1.03483520	1.66737857

Figura 4.5: Resultados del test poker para 4-tuplas

Núm. dígitos	Base 10	Base 16	Núm. dígitos	Base 10	Base 16
100000	0.76051587	0.43384241	1100000	2.70652106	2.51591355
200000	2.22136243	3.08865106	1200000	3.00233686	3.56935775
300000	1.92927322	2.29185570	1300000	4.35887981	4.24717940
400000	1.71147073	2.40962928	1400000	2.82569326	3.30571491
500000	3.77192791	0.33973264	1500000	3.74277924	5.01625750
600000	3.42269896	0.55425698	1600000	4.19990182	4.71730443
700000	4.02892967	1.02807506	1700000	2.80782563	6.71133242
800000	4.81729015	1.55164682	1800000	1.59672527	6.48347121
900000	4.26865508	1.64490519	1900000	1.92306634	8.58931134
1000000	3.34573578	1.95304741	2000000	1.62516313	7.46208659

Figura 4.6: Resultados del test poker para 5-tuplas

4.6. Test de rachas

Una racha es una secuencia de números consecutivos que satisfacen una misma propiedad. En nuestro caso, nos fijaremos en dos condiciones: que todos los dígitos sean mayores o menores que la mediana y que formen una sucesión creciente o decreciente. Nótese que ambas cualidades son idénticas en la base binaria, dado que el número de rachas mayores/menores que la media coincide con la cantidad de veces que crece/decrece la secuencia.

Este test sirve para observar si existe algún patrón de crecimiento. Si hubiera una cantidad muy alta de rachas, indicaría que la serie alterna demasiadas veces como para ser aleatoria. En cambio, si el número de rachas es muy bajo, significa que las secuencias son demasiado largas y, de alguna manera, predecibles. Por tanto, lo ideal es que esté en un punto intermedio.

La variable aleatoria relativa al número de rachas se distribuye como una normal en ambos casos. Respecto al crecimiento, la media y la varianza valen:

$$\mu = \frac{2n-1}{3}, \quad \sigma = \frac{16n-29}{90}, \text{ donde n es el número de dígitos.}$$

En cuanto a la mediana, expresando el número de rachas mayores que ella como r_1 y las rachas menores como r_2, los respectivos valores son:

$$\mu = \frac{2r_1r_2}{r_1+r_2} + 1, \quad \sigma = \frac{2r_1r_2(2r_1r_2 - r_1 - r_2)}{(r_1+r_2)^2(r_1+r_2-1)}.$$

Tras contar las rachas en la secuencia de cifras del número π y estandarizar los datos conforme a las distribuciones anteriores, se comparan con el valor de la normal al $95\,\%$ $(1{,}645)$, produciendo los siguientes resultados:

Núm. dígitos	Base 10	Base 16	Núm. dígitos	Base 10	Base 16
100000	4.97518413	3.55753224	1100000	16.6789243	6.19154056
200000	6.60184779	3.71232742	1200000	16.9056787	6.73912610
300000	8.40640927	3.98661565	1300000	18.0805730	7.17505204
400000	9.85788355	3.88250879	1400000	18.5134443	7.39245073
500000	11.2942060	4.51127532	1500000	19.1849951	7.45291545
600000	12.2331706	4.93470866	1600000	20.2446340	7.59625430
700000	13.2561736	4.78031186	1700000	20.7925862	8.03278418
800000	14.0605565	4.79684106	1800000	21.3638154	8.35388157
900000	14.6211616	5.12667182	1900000	21.9233458	8.44192949
1000000	16.1355088	5.79250735	2000000	22.5638915	8.80619870

Figura 4.7: Resultados del test de rachas respecto al crecimiento

Núm. dígitos	Base 2	Base 10	Base 16	Núm. dígitos	Base 2	Base 10	Base 16
100000	0.62014107	0.58337690	1.89966884	1100000	0.32288292	1.67988197	0.18144360
200000	0.65934496	1.57592700	0.80863295	1200000	0.52357508	2.01535700	0.18392771
300000	0.46540486	1.93730520	0.41139781	1300000	0.98979872	1.62729691	0.35423924
400000	0.53105797	1.34524146	0.69303557	1400000	0.80494463	1.68090630	0.32115384
500000	0.08637015	1.50015426	0.95080110	1500000	0.58632388	1.78465613	0.06207553
600000	0.01949898	1.52801082	0.81345965	1600000	1.06712418	1.41745852	0.24043236
700000	0.30542521	1.91819379	1.34605997	1700000	0.90626853	1.50844972	0.42509890
800000	0.52183589	2.20259152	0.95061531	1800000	0.79370903	1.48896300	0.00731039
900000	0.40746907	2.35559710	0.35012105	1900000	0.60360899	1.48101172	0.35092381
1000000	0.18583299	1.79105410	0.44403483	2000000	0.73263256	1.18262619	0.12430329

Figura 4.8: Resultados del test de rachas respecto a la mediana

4.7. Test de autocorrelación

Por último, vamos a realizar el test de autocorrelación, el cual sirve para probar si existe alguna dependencia entre los dígitos salteados k unidades (lag k). En este caso, la variable aleatoria R_k es la autocovarianza entre el n-ésimo decimal y el situado en la posición $n+k$:

$$R_k = \frac{1}{n-k}\sum_{i=1}^{n-k}\left(U_i - \frac{1}{2}\right)\left(U_{i+k} - \frac{1}{2}\right) \tag{4.2}$$

En la ecuación, n denota el número de cifras y U_i es el i-ésimo dígito d_i normalizado en la base b: $\frac{d_i}{b-1}$. Esta variable se distribuye como una normal con media cero y varianza $\frac{1}{144(n-k)}$. Una covarianza no nula implica que hay dependencia, por lo que el cero debe estar contenido en el intervalo de confianza al 95 %: $\left(R_k - \frac{1{,}96}{12\sqrt{n-k}}, R_k + \frac{1{,}96}{12\sqrt{n-k}}\right)$. A continuación, se muestran los valores de R_k obtenidos y si su correspondiente IC contiene el cero (verde) o no (rojo):

Test autocorrelacion	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag 7	Lag 8	Lag 9	Lag 10
400000	-0.0002444	-0.0003525	1.94e-05	0.0007888	-0.0002269	0.0002925	0.0003244	0.00022	7.81e-05	4.5e-05
800000	0.0001841	-0.0003544	-0.0001928	0.000455	-0.0003541	7.81e-05	0.0003684	0.0001481	0.0001734	0.0004481
1200000	0.000106	-0.0002138	3.85e-05	0.0001596	-0.0001323	7.92e-05	0.0001473	0.0001288	-4.65e-05	0.0003321
1600000	0.0001048	-0.0002134	0.0001211	-6.63e-05	-0.0002089	1.09e-05	-7.23e-05	0.0001266	-9.58e-05	0.0002741
2000000	-1.54e-05	-0.0002488	0.0001729	-6.38e-05	-0.0002451	0.0001168	6.9e-06	0.000157	-9.14e-05	0.0002808
2400000	-3.2e-06	-0.0001935	0.0001584	-1.6e-05	-8.2e-05	0.00011	6.7e-05	0.000155	-5.22e-05	0.0001556
2800000	-4.56e-05	-0.0002534	8.15e-05	-2e-07	8.3e-06	-1.66e-05	5.04e-05	0.0001077	-5.72e-05	0.000175
3200000	-7.29e-05	-0.0002172	5.82e-05	-1.14e-05	6.65e-05	6.7e-06	4.68e-05	8.45e-05	-8.95e-05	0.0001888
3600000	5.37e-05	-0.0001643	7.85e-05	3.49e-05	2.88e-05	-5.33e-05	5.73e-05	0.0001199	-0.0001033	0.0002265
4000000	2.32e-05	-7.28e-05	0.0001288	-4e-06	3.67e-05	-3.36e-05	2.22e-05	0.0001486	-5.91e-05	0.000212
4400000	3.85e-05	-0.0001345	0.0001009	-2.3e-06	2.05e-05	-4.3e-06	4.68e-05	0.0001603	-6.94e-05	0.0001866
4800000	5.97e-05	-9.28e-05	4.79e-05	3.27e-05	-4.35e-05	3.52e-05	5.82e-05	9.22e-05	-0.0001261	0.000179
5200000	0.0001085	-4.13e-05	1e-05	4.15e-05	-0.0001127	5.56e-05	6.76e-05	4.75e-05	-0.0001446	0.0001524
5600000	8.5e-05	-4.46e-05	-1.93e-05	7.8e-06	-6.67e-05	5.59e-05	9.2e-05	6.3e-06	-7.35e-05	0.0001297
6000000	5.99e-05	-6.18e-05	-5.66e-05	-2.38e-05	-9.35e-05	8.16e-05	9.13e-05	3.89e-05	-3.91e-05	0.0001493
6400000	0.0001055	-8.53e-05	-5.06e-05	-6.09e-05	-4.34e-05	8.12e-05	7.03e-05	2.62e-05	-2.66e-05	0.0001008
6800000	8.69e-05	-8.98e-05	-4.34e-05	-5.44e-05	-6.34e-05	3.95e-05	6.86e-05	4.73e-05	-2.47e-05	0.0001059
7200000	7.41e-05	-6.15e-05	-6.22e-05	-5.49e-05	-4.84e-05	3.88e-05	4.77e-05	3.51e-05	1.31e-05	0.0001308
7600000	5.48e-05	-6.84e-05	-6.42e-05	-6.76e-05	-1.75e-05	3.95e-05	2.9e-05	3.89e-05	3.95e-05	0.0001489
8000000	6.48e-05	-3.66e-05	-7.4e-05	-0.0001209	-6.14e-05	3.63e-05	-1.62e-05	3.82e-05	2.65e-05	0.0001619

Figura 4.9: Resultados del test de autocorrelación sobre las cifras de π en binario

Test autocorrelacion	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag 7	Lag 8	Lag 9	Lag 10
120000	-0.0001287	0.0003197	4.97e-05	-0.0001974	-9.89e-05	-0.0001531	0.0001599	0.0001311	0.000235	-0.0001698
240000	-0.0001609	0.0004669	0.0001142	-1.23e-05	-0.0001281	-0.0001216	0.0002076	-0.0002033	0.0001547	-0.000161
360000	-8.51e-05	0.0004638	-6.32e-05	0.0001291	-0.0001298	-0.0001301	0.0002249	-0.0001191	0.0002024	-0.000238
480000	-3.68e-05	0.0003643	3.54e-05	0.0001395	-0.0002018	-2.73e-05	3.56e-05	-4.79e-05	0.0001594	-0.0001853
600000	-4.18e-05	0.0002186	-4.99e-05	8.13e-05	-0.0001388	-9.54e-05	9.73e-05	-3.13e-05	7.91e-05	-7.67e-05
720000	-1.86e-05	0.0001307	1.92e-05	0.0001203	-0.0001029	-0.0001081	0.0002126	-9.87e-05	2.35e-05	-0.0001537
840000	-5.72e-05	5.53e-05	-2.81e-05	0.0001049	-9.5e-05	-5.52e-05	0.0001739	-5.39e-05	5.08e-05	-0.0001767
960000	-0.0001074	3.25e-05	-5.82e-05	0.0001092	-3.66e-05	-3.52e-05	0.0001264	-7.58e-05	3.44e-05	-0.0002151
1080000	-0.0001691	-5.7e-06	-1.21e-05	9.03e-05	-2.48e-05	-8.98e-05	3.39e-05	-0.0001012	8.4e-06	-0.0002009
1200000	-0.0001257	-3e-06	-7.08e-05	2.02e-05	-3.32e-05	-0.0001139	-4.13e-05	-0.0001538	3.09e-05	-0.000164
1320000	-0.0001194	1.9e-05	-8.21e-05	1.2e-05	-8.7e-06	-0.0001268	-6.82e-05	-0.0001563	2.62e-05	-9.42e-05
1440000	-0.000147	4.86e-05	-5.66e-05	2.31e-05	1.7e-06	-9.58e-05	-9.35e-05	-0.0001425	-4.9e-06	-8.18e-05
1560000	-0.000127	2.8e-05	-9.05e-05	1.08e-05	4.2e-05	-7.92e-05	-8.17e-05	-0.0001182	1.4e-06	-8.4e-05
1680000	-0.0001131	1.47e-05	-7.47e-05	1.64e-05	1.3e-05	-9.3e-05	-6.14e-05	-0.000117	-2.52e-05	-9.37e-05
1800000	-0.000119	1.96e-05	-7.46e-05	-2.4e-06	2.41e-05	-9.99e-05	-6.21e-05	-0.0001241	-2.53e-05	-8.6e-05
1920000	-0.000109	1.15e-05	-7.6e-05	-2.79e-05	5.94e-05	-9e-05	-5.42e-05	-7.93e-05	-2.58e-05	-6.26e-05
2040000	-0.0001167	2.74e-05	-5.41e-05	-4.21e-05	9.52e-05	-0.0001002	-4.97e-05	-7.89e-05	-2.67e-05	-7.57e-05
2160000	-0.0001306	3.13e-05	-6.03e-05	-5.28e-05	0.0001067	-9.75e-05	-5.99e-05	-8.19e-05	-1.88e-05	-7.05e-05
2280000	-0.0001198	2.65e-05	-6.41e-05	-4.55e-05	9.32e-05	-9.72e-05	-3.65e-05	-3.95e-05	-2.5e-06	-3.55e-05
2400000	-0.0001063	3.98e-05	-5.83e-05	-5.87e-05	9e-05	-0.0001115	-3.96e-05	-2.51e-05	-9.2e-06	-2.62e-05

Figura 4.10: Resultados del test de autocorrelación sobre las cifras de π en decimal

Test autocorrelacion	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag 7	Lag 8	Lag 9	Lag 10
100000	0.0005183	4e-06	0.0003752	2.93e-05	0.000518	-0.0001666	0.0007299	-0.0001632	-4.9e-06	-6.64e-05
200000	6.85e-05	0.0001447	0.0003162	-8.14e-05	0.0002566	-5.19e-05	0.0003828	-6.38e-05	-0.0002745	-0.0001454
300000	1.55e-05	-2.11e-05	0.0001612	-0.0002125	0.0003532	-0.0001229	0.0001633	0.0001746	-0.0001615	9e-05
400000	-8.54e-05	-0.0001172	0.0001556	-0.0001176	0.0002832	-0.0002341	0.0002001	0.0001616	-8e-07	9e-05
500000	-2.58e-05	-7.48e-05	0.0001636	-0.0001321	0.0002347	-0.000142	0.0002123	0.0001406	2.05e-05	0.0001754
600000	3.3e-06	-8.85e-05	0.0001297	-0.000131	0.0001759	-0.0001137	8.16e-05	0.0001123	4.33e-05	0.0001874
700000	-6.23e-05	-8.9e-05	0.0001366	-0.000142	0.0001177	-9.59e-05	9.96e-05	0.0001173	6.92e-05	0.0001737
800000	-2.02e-05	-9.7e-05	0.0001759	-9.12e-05	8.3e-05	-7.16e-05	5.83e-05	3.89e-05	6.66e-05	0.0001901
900000	5.6e-06	-9.45e-05	0.0001469	-0.0001042	0.0001268	-5.83e-05	4.96e-05	3.86e-05	2.29e-05	0.0002046
1000000	1.79e-05	-5.83e-05	0.0001407	-0.0001444	8.1e-05	-2.26e-05	0.0001113	5.85e-05	3.22e-05	0.0001276
1100000	5.1e-06	-3.8e-05	0.0001609	-0.0001165	6.2e-05	-2.98e-05	9.7e-05	4.03e-05	1.46e-05	0.0001318
1200000	1.69e-05	-6.94e-05	0.0001769	-8.93e-05	6.48e-05	-7e-06	6.72e-05	7.59e-05	1.04e-05	8.21e-05
1300000	1.31e-05	-0.0001011	0.0001672	-0.0001167	7.52e-05	-3.09e-05	6.7e-05	5.78e-05	1.55e-05	9.74e-05
1400000	7.9e-06	-0.000103	0.000133	-0.0001154	5.89e-05	-3.84e-05	6.38e-05	4.17e-05	5e-06	6.48e-05
1500000	-3.31e-05	-8.74e-05	0.0001345	-0.000142	5.9e-05	-3.16e-05	8.83e-05	5.33e-05	3.84e-05	6.51e-05
1600000	-4.43e-05	-0.0001065	0.0001448	-0.0001074	4.97e-05	-1.14e-05	0.0001045	3.09e-05	2.18e-05	6.43e-05
1700000	-4.74e-05	-9.08e-05	0.0001352	-0.0001401	4.88e-05	-1.53e-05	9.78e-05	-2.6e-06	-3.1e-06	3.54e-05
1800000	-2.05e-05	-8.28e-05	0.0001289	-0.0001379	4.43e-05	-8.6e-06	9.26e-05	1.02e-05	1.49e-05	4.18e-05
1900000	-6.2e-06	-6.67e-05	0.0001041	-0.0001256	3.3e-05	-5.66e-05	6.86e-05	1.9e-05	1.07e-05	4.64e-05
2000000	-3.67e-05	-8.28e-05	8.76e-05	-0.0001071	5.75e-05	-3.85e-05	6.32e-05	1.95e-05	1.41e-05	6.73e-05

Figura 4.11: Resultados del test de autocorrelación sobre las cifras de π en hexadecimal

4.8. Discusión de los resultados

Los test más significativos para la hipótesis de normalidad son el de frecuencias y el serial porque son los que analizan la frecuencia con la que aparecen los dígitos individualmente y por parejas. El primero parece indicar que el número π es simplemente normal en las tres bases, al menos en sus primeras cifras. No se puede garantizar que este hecho se mantenga hasta el infinito porque para ello habría que considerar todos los decimales, pero no hay ningún motivo que apunte lo contrario, dado que prácticamente todas las celdas aparecen en verde, al igual que en el test serial.

En cuanto a la aleatoriedad, los resultados no son tan concluyentes. Tanto el test de la mano de poker como el test de rachas con respecto a la mediana aceptan la hipótesis, especialmente a partir del millón y medio de cifras hexadecimales. En cambio, el test de rachas con respecto al crecimiento la rechaza de manera contundente, al estar todas las celdas en rojo. Finalmente, el test de autocorrelación presenta muy buenos datos en decimal y en hexadecimal, pero en binario, parece haber una cierta dependencia entre las cifras en posiciones múltiplos de diez. Por tanto, el número π es probable que sea normal, pero para comprobar la aleatoriedad habría que analizar una mayor cantidad de dígitos.

5

Conclusión

El número π es una constante matemática conocida desde la Edad Antigua. Dada su relación con numerosos y diversos campos de las matemáticas, tiene una gran cantidad de aplicaciones y no sólo en el mundo de la ciencia. Es por ello que, a lo largo de la historia, muchas personas de la talla de Arquímedes, Newton o Gauss le han dedicado su tiempo a examinarlo, con el fin de obtener mejores y más exactas representaciones. De hecho, en las últimas décadas se ha ido superando el récord de mayor cantidad de dígitos computados de π.

Ha sido muy interesante, desde un punto de vista histórico, haber podido indagar sobre el origen y la evolución de los algoritmos para calcular π con una precisión arbitrariamente grande. En el capítulo 2, se presentan algunos de los métodos más utilizados para este propósito, empezando con el más primitivo, pero no menos creativo: el de Arquímedes, basado completamente en relaciones geométricas. Hubo que esperar hasta la llegada del cálculo por parte de Newton y Leibniz para tener una forma alternativa, aunque no mucho mejor. Ya en el siglo XVIII, Machin descubrió identidades involucrando la función arcotangente, una de las cuales superaba en gran medida la eficiencia de los dos anteriores. El último de los algoritmos lineales analizados, y el que más rápidamente converge de ellos, es el de Chudnovsky, generado a partir de los estudios de Ramanujan. Sin embargo, los más rápidos son el de Salamin-Brent, generado a partir de la fórmula de Gauss, y el de los hérmanos Borwein porque tienen un orden de convergencia cuadrático y cuártico, respectivamente. Sin embargo, a pesar de que dos iteraciones del primero equivalen a uno del segundo, prácticamente tardan lo mismo para una precisión fijada.

Otra manera de obtener las cifras de π es hallando el n-ésimo dígito directamente, sin tener que calcular todas las previas. Este problema fue muy costoso de resolver y hubo que recurrir a uno de los diez mejores algoritmos del siglo pasado: el algoritmo PSLQ. Éste busca una relación de números enteros dado un vector de reales. Su implementación, que se encuentra en el apéndice 3, fue probablemente el mayor reto, no sólo por la longitud del código, sino por toda la teoría algebraica detrás de él. Tras una búsqueda meticulosa, David Bailey, Peter Borwein y Simon Plouffe encontraron la fórmula BBP, denominada así en honor a sus descubridores. Posteriormente, Fabrice Bellard consiguió una fórmula de aspecto similar, pero más eficiente. Tras efectuar un nuevo análisis entre sus tiempos de ejecución, aunque ambos muestran una correlación lineal, el

segundo resultó ser cerca de un 41 % más rápido, dato cercano al 43 % que estipulaba el propio Bellard.

Finalmente, a partir de este último algoritmo y del de los hermanos Borwein, obtuvimos los primeros dos millones de dígitos de π en hexadecimal, que se convirtieron en los ocho primeros millones en binario tras el cambio de base, y 1.2 millones de decimales. Esto permitió estudiar una de las conjeturas sobre π de más actualidad con un enfoque estadístico. Se cree que esta constante es normal, es decir, todas las palabras de una determinada cantidad de cifras aparecen con la misma frecuencia. La aleatoriedad de la secuencia de decimales implicaría la normalidad, pero no al revés, por lo que se realizaron una serie de test estadísticos que pueden refutar la hipótesis de aleatoriedad con un 5 % de error. Aunque de por sí no aseguran que las cifras sean aleatorias, en general, los resultados son satisfactorios. En particular, los test más significativos con respecto a la normalidad son el de frecuencias y el serial, los cuales aparecen en verde prácticamente al completo, indicando que la hipótesis es cierta con gran probabilidad. No obstante, el límite de cifras es un condicionante a tener en cuenta, pues que un número tenga esta propiedad al comienzo, no implica que lo mantenga para siempre. Sin embargo, no existe ningún indicio de que vaya a cambiar esta tendencia, dado que la evolución cada 100.000 dígitos (hexadecimales) muestra una estabilidad, especialmente a partir del millón de cifras. En cuanto a la aleatoriedad, los datos no son concluyentes porque el test de rachas con respecto al crecimiento lo refuta, mientras que el resto lo aceptan en su mayoría. Para esclarecer este problema, habría que analizar una mayor cantidad de dígitos.

Este recorrido a través de la historia del número π, desde sus orígenes en la antigua Grecia hasta los problemas más actuales, permite un gran acercamiento a una de las constantes matemáticas más importantes. No obstante, a pesar de la antigüedad de este legendario número, tras miles de años de investigación seguimos desconociendo algunas de sus propiedades y seguirá estando presente en nuestro día a día.

Bibliografía

[1] D. H. Bailey and J. M. Borwein, *Pi: the next generation*. Springer International Publishing, first ed., 2016.

[2] J. G. Goyanes, "Historia de las fórmulas y algoritmos para π," *Gaceta de la Real Sociedad Matemática Española*, vol. 10, no. 1, pp. 159–178, 2007. (Descargar).

[3] D. Shanks and J. W. Wrench, Jr., "Calculation of π to 100,000 decimals," *Math. Comp.*, vol. 16, no. 77, pp. 76–99, 1962. (Descargar).

[4] L. Milla, "Easy proof of three recursive π-algorithms – einfacher beweis dreier rekursiver π-algorithmen," *arXiv e-prints*, 2019. (Descargar).

[5] D. H. Bailey, "Integer relation detection," *Computing in Science Engineering*, vol. 2, no. 1, pp. 24–28, 2000. (Ver abstract).

[6] D. H. Bailey, S. M. Plouffe, P. B. Borwein, and J. M. Borwein, "The quest for pi," *Mathematical Intelligencer*, vol. 19, pp. 50–56, 1997. (Descargar).

[7] F. Bellard, "A new formula to compute the nth binary digit of π," January 1997. (Descargar).

[8] D. Takahashi, "Computation of the 100 quadrillionth hexadecimal digit of π on a cluster of intel xeon phi processors.," *Parallel Computing*, vol. 75, pp. 1–10, 2018. (Descargar).

[9] D. E. Knuth, *The art of computer programming*, vol. 2 (Seminumerical algorithms). Addison-Wesley, 1969.

[10] G. K. Kanji, *100 statistical tests*, vol. 2 (Seminumerical algorithms). Sage publications, third ed., 2006.

Apéndices

A

Código:

Algoritmos para calcular π

En este apéndice se incluyen las implementaciones de los algoritmos presentados en el capítulo 2 y los códigos para generar las gráficas. Las funciones reciben como parámetro la precisión aritmética con la que se quieren realizar las operaciones y devuelve el cómputo del número π y la cantidad de iteraciones necesitadas.

A.1. Algoritmos

- Algoritmo de Arquímedes (250 A.C.)

Código A.1: Implementación del algoritmo de Arquímedes en Python.

```
1  import sys
2  from bigfloat import *
3
4  '''
5      Metodo de Arquimedes:
6
7      a_0 = 12**(0.5)
8      b_0 = 3
9      Iteracion:
10         a_(n+1) = (2 *a_n *b_n)/(a_n + b_n)
11         b_(n+1) = (a_n *b_n)**(0.5)
12 '''
13 def arquimedes(prec):
14     iteracion = 0
15     setcontext(precision(prec+100))
16     err = pow(2,-prec)
17     a = sqrt(12)
18     b = BigFloat(3)
19
20     while sub(a,b) > err:
21         a = div(2*a*b, a+b)
22         b = sqrt(a*b)
23         iteracion+=1
24
25     return a, iteracion
```

- Algoritmo de Newton (1666)

Código A.2: Implementación del algoritmo de Newton en Python.

```
1  import sys
2  from bigfloat import *
3
4  def newton(prec):
5      setcontext(precision(prec+100))
6      err = pow(2,-prec)
7      suma_ini = BigFloat(0)
8      suma_fin = BigFloat(1)
9      potencia = BigFloat(1)
10     exponencial = BigFloat(1)
11     binomio = BigFloat(2)
12     impar = 1
13
14     while sub(suma_fin, suma_ini) > err:
15         impar += 2
16         exponencial *= 16
17         suma_ini = suma_fin
18         suma_fin += div(binomio, impar*exponencial)
19         potencia += impar
20         binomio = (impar+1)*impar*div(binomio, potencia)
21
22     return 3*suma_fin, int((impar-1)/2)
```

- Algoritmo de Machin (1706)

Código A.3: Implementación del algoritmo de Machin en Python. Las dos llamadas a la función arcotangente se realizan en paralelo.

```
1  import sys
2  from bigfloat import *
3  from multiprocessing import Pool
4
5  '''
6      Optimizacion del calculo de la arcotangente
7  '''
8  def arcotan(m, err):
9      err = BigFloat(err)
10     m2 = sqr(m)
11     m3 = mul(m, m2)
12     m4 = sqr(m2)
13     pot = m4
14     aux = 4*sub(m3, m)
15     op1 = sub(3*m3, m)
16     op2 = BigFloat(3)
17     s_ini = BigFloat(0)
18     s_fin = div(op1, op2*pot)
19     k = 1
20     while sub(s_fin, s_ini) > err:
21         op1 += aux
22         op2 += 32*k
23         pot *= m4
24         s_ini = s_fin
25         s_fin += div(op1, op2*pot)
26         k += 1
27     return str(s_fin)
28
29 '''
30     Metodo de Machin:
31
32     pi = 16 *arcotan(0.2) -4 *arcotan(1/239)
33 '''
34 def machin(prec):
35     setcontext(precision(prec+100))
36     p = Pool(2)
37     err = str(pow(2,-prec))
38     results = [p.apply_async(arcotan, (m, err)) for m in [5, 239]]
39     return 16*BigFloat(results[0].get())-4*BigFloat(results[1].get())
```

Código A.4: Implementación del algoritmo de Machin en Python. Las dos llamadas a la función arcotangente se realizan en serie.

```
1  def machinSerie(prec):
2      setcontext(precision(prec+100))
3      err = str(pow(2,-prec))
4      res1, iters1 = arcotan(5, err)
5      res2, iters2 = arcotan(239, err)
6      return 16*res1-4*res2, iters1
```

- Algoritmo de Gauss (1800)

Código A.5: Implementación del algoritmo de Salamin-Brent a partir de la fórmula de Gauss.

```
1  import sys
2  from bigfloat import *
3
4  def agm(prec):
5      iterador = 1
6      setcontext(precision(prec+100))
7      err = pow(2,-prec)
8      a = div(1+sqrt(2)/2, 2)
9      b = sqrt(sqrt(2)/2)
10     s = 2*sub(sqr(a), sqr(b))
11     pow2 = BigFloat(2)
12     res_ini = BigFloat(4)
13     res_fin = div(2*sqr(a), 0.5-s)
14     while sub(res_ini, res_fin) > err:
15         iterador += 1
16         a, b = div(a+b, 2), sqrt(a*b)
17         a2 = sqr(a)
18         pow2 *= 2
19         s += pow2*sub(a2, sqr(b))
20         res_ini = res_fin
21         res_fin = div(2*a2, 0.5-s)
22     return res_fin, iterador
```

- Algoritmo de Ramanujan-Chudnovsky (1914)

Código A.6: Implementación del algoritmo de Ramanujan-Chudnovsky en Python.

```
1  import sys
2  from bigfloat import *
3
4  def ramanujan(prec):
5      iterador = 1
6      setcontext(precision(prec+100))
7      err = pow(2,-prec)
8      divisor = pow(-640320, 3)
9      cubo = divisor
10     factor = BigFloat(558731543)
11     inc = BigFloat(545140134)
12     aux = BigFloat(18)
13     coef = BigFloat(120)
14     res_ini = BigFloat(13591409)
15     res_fin = res_ini + (coef*div(factor, divisor))
16     while abs(sub(res_fin, res_ini)) > err:
17         iterador += 1
18         divisor, factor, aux, coef = divisor*cubo, factor+inc, aux+12, coef*div(pow(aux, 3) -16*aux,
               pow(iterador, 3))
19         res_ini = res_fin
20         res_fin = res_fin + div(mul(factor, coef), divisor)
21     return div(sqrt(-cubo), 12*res_fin), iterador
```

- Algoritmo de Borwein-Borwein (1987)

Código A.7: Implementación del algoritmo de los hermanos Borwein en Python.

```
import sys
from bigfloat import *

'''
    Metodo de cuartico de Borwein:

    s_0 = 2**(0.5) -1
    y_0 = 6 -4 *(2**(0.5))
    Iteracion:
        aux = (1 -s_n**4)**0.25
        s_(n+1) = (1- aux)/(1 + aux)
        y_(n+1) = y_n*(1 + s_(n+1)**4) -2**(2n+3) *s_(n+1) *(1 + s_(n+1) + s_(n+1)**2)

    pi = 1/y
'''
def borwein(prec):
    iteracion = 0
    setcontext(precision(prec+100))
    err = pow(2,-prec)
    s = sub(sqrt(2), 1)
    t_ini = BigFloat(0)
    t_fin = sub(6, 4*sqrt(2))
    while abs(sub(t_fin, t_ini)) > err:
        aux = sqrt(sqrt((1-pow(s, 4))))
        s = div(1-aux, 1+aux)
        t_ini = t_fin
        t_fin = t_ini*pow(1+s, 4)-pow(2, 2*iteracion+3)*s*(1+s+pow(s, 2))
        iteracion+=1
    return div(1, t_fin), iteracion
```

A.2. Análisis de los algoritmos

- Análisis de tiempos

Código A.8: Este código ejecuta cada uno de los algoritmos cinco veces con diferentes valores de precisión y genera la gráfica sobre la media de los tiempos de ejecución.

```
import matplotlib.pyplot as plt
import numpy as np
from timeit import timeit
from arquimedes import arquimedes
```

```
5  from newton import newton
6  from machin import machin
7  from agm import agm
8  from ramanujan import ramanujan
9  from borwein import borwein
10
11 min = 5000
12 max = 200000
13 inc = 40
14 reps = 3
15
16 x = np.linspace(min, max, inc)
17
18 arq = [0]*len(x)
19 new = [0]*len(x)
20 mac = [0]*len(x)
21 mag = [0]*len(x)
22 ram = [0]*len(x)
23 bor = [0]*len(x)
24
25 for rep in range(reps):
26     for i in range(len(x)):
27         print("rep:␣" + str(rep) + "␣prec:␣" + str(x[i]))
28         arq[i] += timeit(lambda: arquimedes(x[i]), number = 1)
29         new[i] += timeit(lambda: newton(x[i]), number = 1)
30         mac[i] += timeit(lambda: machin(x[i]), number = 1)
31         mag[i] += timeit(lambda: agm(x[i]), number = 1)
32         ram[i] += timeit(lambda: ramanujan(x[i]), number = 1)
33         bor[i] += timeit(lambda: borwein(x[i]), number = 1)
34
35
36 with open('tiempo_5_5000_200000.txt', 'a') as f:
37     f.write("Prec.\tArq.\tNew.\tMac.\tAGM\tRam.\tBor.\n")
38     [f.write(str(x[i])+"\t"+str(arq[i]/reps)+"\t"+str(new[i]/reps)+"\t"+str(mac[i]/reps)+ \
39         "\t"+str(mag[i]/reps)+"\t"+str(ram[i]/reps)+"\t"+str(bor[i]/reps)+"\n") for i in
              range(len(x))]
40
41 plt.plot(x, [arq[i]/reps for i in range(len(x))], label='Arquímedes␣(250␣A.C.)')
42 plt.plot(x, [new[i]/reps for i in range(len(x))], label='Newton␣(1666)')
43 plt.plot(x, [mac[i]/reps for i in range(len(x))], label='Machín␣(1706)')
44 plt.plot(x, [mag[i]/reps for i in range(len(x))], label='Agm␣(1800)')
45 plt.plot(x, [ram[i]/reps for i in range(len(x))], label='Ramanujan␣(1914)')
46 plt.plot(x, [bor[i]/reps for i in range(len(x))], label='Borwein␣(1987)')
47
48 plt.xlabel('Precisión␣(bits)')
49 plt.ylabel('Tiempo␣(s)')
50
51 plt.title("Tiempos␣de␣ejecución")
52
53 plt.legend()
54
55 plt.show()
```

Código A.9: Este código compara los tiempos de ejecución entre el algoritmo de Gauss y el de los hermanos Borwein siguiendo el mismo esquema.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from timeit import timeit
4  import sys
5  from agm import agm
6  from borwein import borwein
7
8  min = 200000
9  max = 10000000
10 inc = 141
11 reps = 5
12
13 x = np.linspace(min, max, inc)
14
15 mag = [0]*len(x)
16 bor = [0]*len(x)
17
18 for rep in range(reps):
19     for i in range(len(x)):
20         print("rep:␣" + str(rep) + "␣prec:␣" + str(x[i]))
21         mag[i] += timeit(lambda: agm(x[i]), number = 1)
22         bor[i] += timeit(lambda: borwein(x[i]), number = 1)
23
24 with open('tiempo_5_200000_10000000.txt', 'a') as f:
25     f.write("Prec.\tAGM\tBor.\n")
26     [f.write(str(x[i])+"\t"+str(mag[i]/reps)+"\t"+str(bor[i]/reps)+"\n") for i in range(len(x))]
27
28 plt.plot(x, [mag[i]/reps for i in range(len(x))], label='Agm␣(1800)')
29 plt.plot(x, [bor[i]/reps for i in range(len(x))], label='Borwein␣(1987)')
30
31 plt.xlabel('Precisión␣(bits)')
32 plt.ylabel('Tiempo␣(s)')
33
34 plt.title("Tiempos␣de␣ejecución")
35
36 plt.legend()
37
38 plt.show()
```

- Análisis de iteraciones

Código A.10: Este código ejecuta el algoritmo de Gauss y el de los hermanos Borwein con diferentes valores de precisión y genera la gráfica sobre el número de iteraciones.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from agm import agm
4  from borwein import borwein
5
6  min = 5000
7  max = 200000
8  inc = 40
9  y = np.linspace(min, max, inc)
10
11 min = 270000
12 max = 10000000
13 inc = 140
14 z = np.linspace(min, max, inc)
15
16 x = [*y, *z]
17
18 mag = [0]*len(x)
19 bor = [0]*len(x)
20
21 for i in range(len(x)):
22     print("prec: " + str(x[i]))
23     res, mag[i] = agm(x[i])
24     res, bor[i] = borwein(x[i])
25
26 with open('iters_5000_10000000.txt', 'w') as f:
27     f.write("Prec.\tAGM\tBor.\n")
28     [f.write(str(x[i])+"\t"+str(mag[i])+"\t"+str(bor[i])+"\n") for i in range(len(x))]
29
30 plt.plot(x, mag, label='Agm (1800)')
31 plt.plot(x, bor, label='Borwein (1987)')
32
33 plt.xlabel('Precisión (bits)')
34 plt.ylabel('Iteraciones')
35
36 plt.title("Número de iteraciones")
37 plt.legend()
38 plt.show()
```

Código A.11: Este código ejecuta los seis algoritmos con valores menores de precisión y genera la gráfica correspondiente sobre el número de iteraciones.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from arquimedes import arquimedes
4  from newton import newton
5  from machin_serie import machinSerie
6  from ramanujan import ramanujan
7
8  min = 5000
9  max = 200000
10 inc = 40
11 x = np.linspace(min, max, inc)
12
13 arq = [0]*len(x)
14 new = [0]*len(x)
15 mac = [0]*len(x)
16 ram = [0]*len(x)
17
18 for i in range(len(x)):
19     print("prec: " + str(x[i]))
20     res, arq[i] = arquimedes(x[i])
21     res, new[i] = newton(x[i])
22     res, mac[i] = machinSerie(x[i])
23     res, ram[i] = ramanujan(x[i])
24
25 with open('iters_5000_200000.txt', 'w') as f:
26     f.write("Prec.\tArq.\tNew.\tMac.\tAGM\tRam.\tBor.\n")
27     [f.write(str(x[i])+"\t"+str(arq[i])+"\t"+str(new[i])+"\t"+ \
28         str(mac[i])+"\t"+str(ram[i])+"\n") for i in range(len(x))]
29
30 plt.plot(x, arq, label='Arquímedes (250 A.C.)')
31 plt.plot(x, new, label='Newton (1666)')
32 plt.plot(x, mac, label='Machín (1706)')
33 plt.plot(x, ram, label='Ramanujan (1914)')
34
35 plt.xlabel('Precisión (bits)')
36 plt.ylabel('Iteraciones')
37
38 plt.title("Número de iteraciones")
39
40 plt.legend()
41
42 plt.show()
```

B

Algoritmos equivalentes

Dos algoritmos son equivalentes si producen la misma salida. En este apéndice vamos a ver que el resultado proporcionado por el algoritmo de los hermanos Borwein (2.7) tras la n-ésima iteración coincide con el del algoritmo Salamin-Brent (2.5) en la $2n$-ésima iteración. Por tanto, ambos son equivalentes, pero el primero duplica en orden de convergencia al segundo.

Primero, recordemos las iteraciones de ambos algoritmos:

	Salamin-Brent		Borwein-Borwein	
Inicialización	$a_0 = 1$	$b_0 = (\sqrt{2})^{-1}$	$y_0 = \sqrt{2} - 1$	$z_0 = 6 - 4\sqrt{2}$
Iteración	$a_n = \frac{a_{n-1}+b_{n-1}}{2}$ $b_n = \sqrt{a_{n-1}b_{n-1}}$ $c_n{}^2 = a_n{}^2 - b_n{}^2$		$y_n = \frac{1-\sqrt[4]{1-{y_{n-1}}^4}}{1+\sqrt[4]{1-{y_{n-1}}^4}}$ $z_n = z_{n-1}(1+y_n)^4 - 2 * 4^n y_n(1+y_n+{y_n}^2)$	
Salida	$S_N = \frac{(a_N+b_N)^2}{1-2\sum_{j=1}^{N} 2^j {c_j}^2}$		$B_N = \frac{1}{z_N}$	

Tabla B.1: Comparación entre el algortimo de Salamin-Brent y el de los hermanos Borwein

A continuación vamos a ver que $B_N = S_{2N}$:

Demostración. Para llegar al objetivo, necesitamos el siguiente resultado intermedio que probaremos por inducción sobre el índice n:

$$y_n = \sqrt{\frac{a_{2n}}{a_{2n+1}} - 1}$$

Caso base: la igualdad es cierta para $n = 0$:

$$\sqrt{\frac{a_0}{a_1} - 1} = \sqrt{\frac{1}{\frac{1+\frac{1}{\sqrt{2}}}{2}} - 1} = \sqrt{\frac{2\sqrt{2}}{\sqrt{2}+1} - 1} = \sqrt{3 - 2\sqrt{2}} = \sqrt{\left(\sqrt{2}-1\right)^2} = y_0$$

Paso inductivo: suponemos que la fórmula se cumple para todo $m < n$, entonces vamos a comprobar que se cumple para $m = n$.

Sea k_m el radicando, de forma que $y_m = \sqrt{k_{2m}}$. Realizando algunas operaciones, tenemos que:

$$k_m = \frac{a_m}{a_{m+1}} - 1 = \frac{a_m - \frac{a_m+b_m}{2}}{\frac{a_m+b_m}{2}} = \frac{a_m - b_m}{a_m + b_m}$$

Por un lado, elevando ambos miembros al cuadrado:

$${k_m}^2 = \left(\frac{a_m - b_m}{a_m + b_m}\right)^2 = \frac{(a_m + b_m)^2 - 4a_m b_m}{(a_m + b_m)^2} = 1 - \frac{(\sqrt{a_m b_m})^2}{\left(\frac{a_m+b_m}{2}\right)^2} = 1 - \frac{b_{m+1}^2}{a_{m+1}^2}$$

Es decir, $\sqrt{1 - {k_m}^2} = \frac{b_{m+1}}{a_{m+1}}$. Por otro lado, usando esta ecuación:

$$k_m = \frac{a_m - b_m}{a_m + b_m} = \frac{1 - b_m/a_m}{1 + b_m/a_m} = \frac{1 - \sqrt{1 - k_{m-1}^2}}{1 + \sqrt{1 - k_{m-1}^2}} \Rightarrow \sqrt{1 - k_{m-1}^2} = \frac{1 - k_m}{1 + k_m}$$

Despejamos k_{m-1} de esta última igualdad:

$$k_{m-1} = \sqrt{1 - \left(\frac{1 - k_m}{1 + k_m}\right)^2} = \sqrt{\frac{(1 + k_m^2) - (1 - k_m)^2}{(1 + k_m)^2}} = \frac{2\sqrt{k_m}}{1 + k_m}$$

Por inducción:

$$k_{2n-1} = \frac{1 - \sqrt{1 - k_{2n-2}^2}}{1 + \sqrt{1 - k_{2n-2}^2}} = \frac{1 - \sqrt{1 - y_{n-1}^4}}{1 + \sqrt{1 - y_{n-1}^4}}$$

Juntando las dos últimos identidades:

$$\sqrt{1 - y_{n-1}^4} = \frac{1 - k_{2n-1}}{1 + k_{2n-1}} = \frac{1 - \frac{2\sqrt{k_{2n}}}{1+k_{2n}}}{1 + \frac{2\sqrt{k_{2n}}}{1+k_{2n}}} = \frac{1 + k_{2n} - 2\sqrt{k_{2n}}}{1 + k_{2n} + 2\sqrt{k_{2n}}} = \frac{(1 - \sqrt{k_{2n}})^2}{(1 + \sqrt{k_{2n}})^2}$$

Finalmente, tomando la raíz cuadrada en ambos lados, obtenemos el resultado intermedio que buscábamos:

$$\sqrt[4]{1 - y_{n-1}^4} = \frac{1 - \sqrt{k_{2n}}}{1 + \sqrt{k_{2n}}} \Rightarrow \sqrt{k_{2n}} = \frac{1 - \sqrt[4]{1 - y_{n-1}^4}}{1 + \sqrt[4]{1 - y_{n-1}^4}} = y_n$$

La equivalencia la demostraremos también por inducción, siendo el caso base $N = 0$:

$$B_0 = \frac{1}{6 - 4\sqrt{2}} = \frac{6 + 4\sqrt{2}}{4} = \left(\frac{2 + \sqrt{2}}{2}\right)^2 = \left(1 + \frac{1}{\sqrt{2}}\right)^2 = (a_0 + b_0)^2 = S_0$$

Paso inductivo: suponemos que $1/S_{2m} = 1/B_m = z_m$ para todo $m < n$ y queremos ver que, de nuevo, se cumple para $m = n$.

Partimos desde el lado izquierdo:

$$\begin{aligned}\frac{1}{S_{2n}} &= \frac{1-2\sum_{j=1}^{2n} 2^j c_j^2}{(a_{2n}+b_{2n})^2} = \frac{1-2\sum_{j=1}^{2n-2} 2^j c_j^2}{(a_{2n}+b_{2n})^2} - \frac{2^{2n}c_{2n-1}^2}{(a_{2n}+b_{2n})^2} - \frac{2^{2n+1}c_{2n}^2}{(a_{2n}+b_{2n})^2} \\ &= \frac{(a_{2n-2}+b_{2n-2})^2}{(a_{2n}+b_{2n})^2}\frac{1}{S_{2n-2}} - \frac{2^{2n}c_{2n-1}^2}{(a_{2n}+b_{2n})^2} - \frac{2^{2n+1}c_{2n}^2}{(a_{2n}+b_{2n})^2} \\ &= \frac{(a_{2n-2}+b_{2n-2})^2}{(a_{2n}+b_{2n})^2} z_{n-1} - 2*4^n\left(\frac{a_{2n-1}^2-b_{2n-1}^2}{2(a_{2n}+b_{2n})^2} + \frac{a_{2n}^2-b_{2n}^2}{(a_{2n}+b_{2n})^2}\right)\end{aligned}$$

Si comparamos esta relación con la iteración de z_n, falta ver que el coeficiente de z_{n-1} es igual a $(1+y_n)^4$ y el coeficiente de 4^n coincide con $y_n(1+y_n+y_n^2)$. Para ello, necesitaremos estas tres identidades donde será útil el resultado intermedio anterior:

- $\frac{a_{2n}^2-b_{2n}^2}{(a_{2n}+b_{2n})^2} = y_n^2$

$$\frac{a_{2n}^2-b_{2n}^2}{(a_{2n}+b_{2n})^2} = \frac{a_{2n}-b_{2n}}{a_{2n}+b_{2n}} = \frac{2a_{2n}}{a_{2n}+b_{2n}} - \frac{a_{2n}+b_{2n}}{a_{2n}+b_{2n}} = \frac{a_{2n}}{a_{2n-1}} - 1 = y_n^2$$

- $\frac{a_{2n-1}^2-b_{2n-1}^2}{2(a_{2n}+b_{2n})^2} = y_n(1+y_n^2)$

A partir del punto anterior:

$$4y_n^2 = \frac{a_{2n}^2-b_{2n}^2}{\left(\frac{a_{2n}+b_{2n}}{2}\right)^2} = \frac{c_{2n}^2}{a_{n+1}^2} \Rightarrow y_n = \frac{c_{2n}}{2a_{2n+1}}$$

Por otra parte:

$$c_{2n} = \sqrt{a_{2n}^2-b_{2n}^2} = \sqrt{\frac{(a_{2n-1}+b_{2n-1})^2}{4} - a_{2n-1}b_{2n-1}} = \frac{a_{2n-1}-b_{2n-1}}{2}$$

Uniendo ambas ecuaciones:

$$\frac{a_{2n-1}-b_{2n-1}}{4a_{2n+1}} = \frac{c_{2n}}{2a_{2n+1}} = y_n$$

Finalmente:

$$\frac{a_{2n-1}^2-b_{2n-1}^2}{2(a_{2n}+b_{2n})^2} = \frac{a_{2n-1}+b_{2n-1}}{2}\frac{a_{2n-1}-b_{2n-1}}{(2a_{2n+1})^2} = \frac{a_{2n}}{a_{2n+1}}\frac{a_{2n-1}-b_{2n-1}}{4a_{2n+1}} = (1+y_n^2)y_n$$

- $\frac{(a_{2n-2}+b_{2n-2})^2}{(a_{2n}+b_{2n})^2} = (1+y_n)^4$

Usando las fórmulas anteriores:

$$(1+y_n^4) = 1+6y_n^2+4(y_n+y_n^3)+y_n^4 = 1+6\frac{a_{2n}^2-b_{2n}^2}{(a_{2n}+b_{2n})^2}+2\frac{a_{2n-1}^2-b_{2n-1}^2}{(a_{2n}+b_{2n})^2}+\frac{(a_{2n}^2-b_{2n}^2)^2}{(a_{2n}+b_{2n})^4}$$

Quitando los denominadores:

$$\begin{aligned}(a_{2n}+b_{2n})^2(1+y_n^4) &= (a_{2n}+b_{2n})^2+6(a_{2n}^2-b_{2n}^2)+2(a_{2n-1}^2-b_{2n-1}^2)+(a_{2n}-b_{2n})^2\\ &= 8a_{2n}^2-4b_{2n}^2+2(a_{2n-1}^2-b_{2n-1}^2)\\ &= 2(a_{2n-1}+b_{2n-1})^2-4a_{2n-1}b_{2n-1}+2(a_{2n-1}^2-b_{2n-1}^2)\\ &= 4a_{2n-1}^2=(a_{2n-2}+b_{2n-2})^2 \Rightarrow \frac{(a_{2n-2}+b_{2n-2})^2}{(a_{2n}+b_{2n})^2}=(1+y_n)^4\end{aligned}$$

Uniendo todas las igualdades, llegamos a nuestro objetivo:

$$\begin{aligned}\frac{1}{S_{2n}} &= \frac{(a_{2n-2}+b_{2n-2})^2}{(a_{2n}+b_{2n})^2}z_{n-1}-2*4^n\left(\frac{a_{2n-1}^2-b_{2n-1}^2}{2(a_{2n}+b_{2n})^2}+\frac{a_{2n}^2-b_{2n}^2}{(a_{2n}+b_{2n})^2}\right)\\ &= (1+y_n^4)z_{n-1}-2*4^n(y_n(1+y_n^2)+y_n^2)=(1+y_n^4)z_{n-1}-2*4^n y_n(1+y_n+y_n^2)=z_n\end{aligned}$$

□

C

Código:

Algoritmos PSLQ, BBP y Bellard

En este apéndice se muestran los códigos correspondientes a los algoritmos del capítulo 3.

- Algoritmo PSLQ

Código C.1: Este código implementa el algoritmo PSLQ. Recibe como parámetros la precisión, el umbral y el fichero con el tamaño del vector y una componente por línea.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <gmp.h>
5  #include <mpfr.h>
6
7  #define TAM 10000
8
9  /*
10     Implementación del algoritmo PSLQ, basado en el pseudocódigo de David H. Bailey,
11     para obtener la fórmula BBP. Se busca un vector de números enteros a_i tales que
12     sum [a_i*x_i] = 0, con algún a_i distinto de 0.
13     Pseudocódigo: http://www.cecm.sfu.ca/organics/papers/bailey/paper/html/node3.html
14 */
15 int main(int argc, char* argv[]){
16     short flag;
17     int i, j, k, maxIndex, minIndex, prec, umbral, tam, ret;
18     mpfr_t aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow;
19     mpfr_t *gamma_pow, *s, *x, *y;
20     mpfr_t **A, **B, **H;
21     char num[TAM];
22     FILE* f = NULL;
23
24     if(argc != 4){
25         printf("Se necesitan tres argumentos:\n");
26         printf("\tPrecision: indica el numero de digitos maximo de los coeficientes a buscar.\n");
27         printf("\tUmbral: condicion de parada -la relacion es menor que 10^(-umbral).\n");
28         printf("\tFichero: debe contener el tamaño del vector y sus componentes, uno por línea.\n");
29         return -1;
30     }
31
32     prec = atoi(argv[1]);
```

```
33      if(prec <= 0){
34          printf("La precision debe ser un numero positivo.\n");
35          return -1;
36      }
37
38      umbral = atoi(argv[2]);
39      if(umbral <= 0){
40          printf("El umbral debe ser un numero positivo.\n");
41          return -1;
42      }
43
44      f = fopen(argv[3], "r");
45      if(!f){
46          printf("No se pudo abrir el fichero.\n");
47          return -1;
48      }
49      fscanf(f, " %d", &tam);
50      if(tam <= 1){
51          printf("El tamaño del vector debe ser al menos dos.\n");
52          fclose(f);
53          return -1;
54      }
55
56      /* Reserva de memoria de variables y lectura del fichero dado */
57      mpfr_set_default_prec(tam*prec+50);
58      mpfr_inits2(tam*prec, aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
59      gamma_pow = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
60      if(!gamma_pow){
61          printf("Error reservando memoria.\n");
62          mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
63          fclose(f);
64          return -1;
65      }
66
67      s = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
68      if(!s){
69          printf("Error reservando memoria.\n");
70          free(gamma_pow);
71          mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
72          fclose(f);
73          return -1;
74      }
75
76      y = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
77      if(!y){
78          printf("Error reservando memoria.\n");
79          free(gamma_pow);
80          free(s);
81          mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
82          fclose(f);
83          return -1;
84      }
85
86      A = (mpfr_t**)malloc(tam*sizeof(mpfr_t*));
```

```
87      if(!A){
88          printf("Error␣reservando␣memoria.\n");
89          free(gamma_pow);
90          free(s);
91          free(y);
92          mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
93          fclose(f);
94          return -1;
95      }
96
97      B = (mpfr_t**)malloc(tam*sizeof(mpfr_t));
98      if(!B){
99          printf("Error␣reservando␣memoria.\n");
100         free(gamma_pow);
101         free(s);
102         free(y);
103         free(A);
104         mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
105         fclose(f);
106         return -1;
107     }
108
109     H = (mpfr_t**)malloc(tam*sizeof(mpfr_t));
110     if(!H){
111         printf("Error␣reservando␣memoria.\n");
112         free(gamma_pow);
113         free(s);
114         free(y);
115         free(A);
116         free(B);
117         mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
118         fclose(f);
119         return -1;
120     }
121
122     x = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
123     if(!x){
124         printf("Error␣reservando␣memoria.\n");
125         free(gamma_pow);
126         free(s);
127         free(y);
128         free(A);
129         free(B);
130         mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
131         fclose(f);
132         return -1;
133     }
134
135     for(i = 0; i < tam; i++){
136         ret = fscanf(f, "%s", num);
137         if(ret <= 0){
138             printf("Error␣leyendo␣el␣fichero:␣%d␣de␣%d␣elementos␣leídos\n", i-1, tam);
139             for(j = 0; j < i; j++){
140                 mpfr_clear(x[j]);
141             }
```

```
142                 free(x);
143                 free(gamma_pow);
144                 free(s);
145                 free(y);
146                 free(A);
147                 free(B);
148                 mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
149                 fclose(f);
150                 return -1;
151             }
152             mpfr_init2(x[i], tam*prec);
153             mpfr_set_str(x[i], num, 10, MPFR_RNDN);
154         }
155         fclose(f);
156 
157         for(i = 0; i < tam; i++){
158             A[i] = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
159             if(!A[i]){
160                 printf("Error␣reservando␣memoria.\n");
161                 for(j = i-1; j >= 0; j--){
162                     mpfr_clears(gamma_pow[j], s[j], y[j], NULL);
163                     free(A[j]);
164                     free(B[j]);
165                     free(H[j]);
166                 }
167                 for(i = 0; i < tam; i++){
168                     mpfr_clear(x[i]);
169                 }
170                 free(A);
171                 free(B);
172                 free(H);
173                 free(s);
174                 free(x);
175                 free(y);
176                 mpfr_free_cache();
177                 mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
178                 return -1;
179             }
180 
181             B[i] = (mpfr_t*)malloc(tam*sizeof(mpfr_t));
182             if(!B[i]){
183                 printf("Error␣reservando␣memoria.\n");
184                 free(A[i]);
185                 for(j = i-1; j >= 0; j--){
186                     mpfr_clears(gamma_pow[j], s[j], y[j], NULL);
187                     free(A[j]);
188                     free(B[j]);
189                     free(H[j]);
190                 }
191                 for(i = 0; i < tam; i++){
192                     mpfr_clear(x[i]);
193                 }
```

```
194                 free(A);
195                 free(B);
196                 free(H);
197                 free(s);
198                 free(x);
199                 free(y);
200                 mpfr_free_cache();
201                 mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
202                 return -1;
203             }
204 
205             H[i] = (mpfr_t*)malloc((tam-1)*sizeof(mpfr_t));
206             if(!H[i]){
207                 printf("Error reservando memoria.\n");
208                 free(A[i]);
209                 free(B[i]);
210                 for(j = i-1; j >= 0; j--){
211                     mpfr_clears(gamma_pow[j], s[j], y[j], NULL);
212                     free(A[j]);
213                     free(B[j]);
214                     free(H[j]);
215                 }
216                 for(i = 0; i < tam; i++){
217                     mpfr_clear(x[i]);
218                 }
219                 free(A);
220                 free(B);
221                 free(H);
222                 free(s);
223                 free(x);
224                 free(y);
225                 mpfr_free_cache();
226                 mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
227                 return -1;
228             }
229 
230             mpfr_inits2(tam*prec, gamma_pow[i], s[i], y[i], NULL);
231         }
232 
233         /* Paso 1: inicialización de las matrices A y B a la identidad */
234         for(i = 0; i < tam; i++){
235             for(j = 0; j < tam; j++){
236                 if(j == i){
237                     mpfr_init_set_ui(A[i][j], 1, MPFR_RNDN);
238                     mpfr_init_set_ui(B[i][j], 1, MPFR_RNDN);
239                 } else {
240                     mpfr_init_set_ui(A[i][j], 0, MPFR_RNDN);
241                     mpfr_init_set_ui(B[i][j], 0, MPFR_RNDN);
242                 }
243                 if(j != tam-1){
244                     mpfr_init2(H[i][j], tam*prec);
245                 }
246             }
247         }
```

```
248     /* Vector de potencias de gamma: g[0] = 1, g[1] = sqrt(4/3), g[i] = g[i]^i, 1 < i < tam */
249     mpfr_set_d(gamma_pow[1], 0.75, MPFR_RNDN);
250     mpfr_rec_sqrt(gamma_pow[1], gamma_pow[1], MPFR_RNDN);
251     mpfr_set_ui(gamma_pow[0], 1, MPFR_RNDN);
252     for(i = 2; i < tam; i++){
253         mpfr_mul(gamma_pow[i], gamma_pow[i-1], gamma_pow[1], MPFR_RNDN);
254     }
255     /* Paso 2: Cálculo del vector s: s[i] = sqrt(sum [x[j]^2], j=i a N) */
256     mpfr_mul(t, x[tam-1], x[tam-1], MPFR_RNDN);
257     mpfr_set(s[tam-1], x[tam-1], MPFR_RNDN);
258     for(i = tam-2; i > 0; i--){
259         mpfr_fma(t, x[i], x[i], t, MPFR_RNDN);
260         mpfr_sqrt(s[i], t, MPFR_RNDN);
261     }
262     mpfr_fma(t, x[0], x[0], t, MPFR_RNDN);
263     mpfr_sqrt(t, t, MPFR_RNDN);
264     /* t = s[0], s[k] = s[k] / t y y[k] = y[k] / t */
265     mpfr_set_ui(s[0], 1, MPFR_RNDN);
266     mpfr_div(y[0], x[0], t, MPFR_RNDN);
267     for(i = 1; i < tam; i++){
268         mpfr_div(y[i], x[i], t, MPFR_RNDN);
269         mpfr_div(s[i], s[i], t, MPFR_RNDN);
270     }
271     /* Paso 3: inicialización de la matriz H de tamaño tam x (tam-1) y triangular inferior */
272     for(i = 0; i < tam; i++){
273         /* H[i][j] = 0, para j > i */
274         for(j = i+1; j < tam-1; j++){
275             mpfr_set_ui(H[i][j], 0, MPFR_RNDN);
276         }
277         /* H[i][i] = s[i+1]/s[i] */
278         if(i != tam-1){
279             if(mpfr_regular_p(s[i])){
280                 mpfr_div(H[i][i], s[i+1], s[i], MPFR_RNDN);
281             } else {
282                 mpfr_set_ui(H[i][i], 0, MPFR_RNDN);
283             }
284         }
285         /* H[i][j] = -y[i]y[j]/(s[j]s[j+1]), para j < i */
286         for(j = 0; j < i; j++){
287             mpfr_mul(aux, s[j], s[j+1], MPFR_RNDN);
288             if(mpfr_regular_p(aux)){
289                 mpfr_mul(H[i][j], y[i], y[j], MPFR_RNDN);
290                 mpfr_div(H[i][j], H[i][j], aux, MPFR_RNDN);
291                 mpfr_neg(H[i][j], H[i][j], MPFR_RNDN);
292             } else {
293                 mpfr_set_ui(H[i][j], 0, MPFR_RNDN);
294             }
295         }
296     }
297     /* Paso 4: actualización de las matrices A, B, H y el vector y*/
298     for(i = 1; i < tam; i++){
299         for (j = i-1; j >= 0; j--){
300             if(mpfr_regular_p(H[j][j])){
301                 /* t = H[i][j]/H[j][j] redondeado al entero más próximo */
302                 mpfr_div(t, H[i][j], H[j][j], MPFR_RNDN);
303                 mpfr_rint_round(t, t, MPFR_RNDN);
```

```
304                 /* y[j] = y[j] + t*y[i] */
305                 mpfr_fma(y[j], t, y[i], y[j], MPFR_RNDN);
306                 for(k = 0; k <= j; k++){
307                     /* H[i][k] = H[i][k] -t*H[j][k] */
308                     mpfr_fmms(H[i][k], H[i][k], gamma_pow[0], t, H[j][k], MPFR_RNDN);
309                 }
310                 for(k = 0; k < tam; k++){
311                     /* A[i][k] = A[i][k] -t*A[j][k] */
312                     mpfr_fmms(A[i][k], A[i][k], gamma_pow[0], t, A[j][k], MPFR_RNDN);
313                     /* B[k][i] = B[k][i] + t*B[k][j] */
314                     mpfr_fma(B[k][j], t, B[k][i], B[k][j], MPFR_RNDN);
315                 }
316             }
317         }
318     }
319     /* Iteración -condiciones de parada:
320         a) Algún elemento de A supera la precisión (overflow -no se encuentra relación)
321         b) Algún elemento de y es menor que el umbral especificado (underflow -el vector a_i es la
                correspondiente columna de B)
322     */
323     flag = 1;
324     mpfr_ui_pow_ui(overflow, 2, tam*prec, MPFR_RNDN);
325     mpfr_ui_pow_ui(underflow, 10, (unsigned int)umbral, MPFR_RNDN);
326     mpfr_ui_div(underflow, 1, underflow, MPFR_RNDN);
327     while(flag){
328         /* Paso 1: valor de i tal que gamma^i*|H[i|[i] es máximo */
329         for(i = 0; i < tam-1; i++){
330             mpfr_abs(norma, H[i][i], MPFR_RNDN);
331             mpfr_mul(norma, norma, gamma_pow[i], MPFR_RNDN);
332             if((i == 0) || (mpfr_greater_p(norma, max))){
333                 maxIndex = i;
334                 mpfr_set(max, norma, MPFR_RNDN);
335             }
336         }
337
338         /* Paso 2: intercambio de los elementos m, m+1 de y, las filas m, m+1 de A y H y las filas m,
                m+1 de B */
339         mpfr_set(aux, y[maxIndex], MPFR_RNDN);
340         mpfr_set(y[maxIndex], y[maxIndex+1], MPFR_RNDN);
341         mpfr_set(y[maxIndex+1], aux, MPFR_RNDN);
342
343         for(i = 0; i < tam; i++){
344             mpfr_set(aux, A[maxIndex][i], MPFR_RNDN);
345             mpfr_set(A[maxIndex][i], A[maxIndex+1][i], MPFR_RNDN);
346             mpfr_set(A[maxIndex+1][i], aux, MPFR_RNDN);
347
348             if(i != tam-1){
349                 mpfr_set(aux, H[maxIndex][i], MPFR_RNDN);
350                 mpfr_set(H[maxIndex][i], H[maxIndex+1][i], MPFR_RNDN);
351                 mpfr_set(H[maxIndex+1][i], aux, MPFR_RNDN);
352             }
353
354             mpfr_set(aux, B[i][maxIndex], MPFR_RNDN);
355             mpfr_set(B[i][maxIndex], B[i][maxIndex+1], MPFR_RNDN);
356             mpfr_set(B[i][maxIndex+1], aux, MPFR_RNDN);
357         }
```

```
358         /* Paso 3: actualización de H */
359         if(maxIndex != tam-2){
360             /* t0 es la norma euclídea entre H[m][m] y H[m][m+1] */
361             mpfr_hypot(t0, H[maxIndex][maxIndex], H[maxIndex][maxIndex+1], MPFR_RNDN);
362             if(mpfr_regular_p(t0)){
363                 /* t1 = H[m][m]/t0 y t2 = H[m][m+1]/t0 */
364                 mpfr_div(t1, H[maxIndex][maxIndex], t0, MPFR_RNDN);
365                 mpfr_div(t2, H[maxIndex][maxIndex+1], t0, MPFR_RNDN);
366                 for(i = maxIndex; i < tam; i++){
367                     /* H[i][m] = t1*H[i][m] + t2*H[i][m+1] y H[i][m+1] = t1*H[i][m+1] -t2*H[i][m] */
368                     mpfr_set(t3, H[i][maxIndex], MPFR_RNDN);
369                     mpfr_set(t4, H[i][maxIndex+1], MPFR_RNDN);
370                     mpfr_fmma(H[i][maxIndex], t1, t3, t2, t4, MPFR_RNDN);
371                     mpfr_fmms(H[i][maxIndex+1], t1, t4, t2, t3, MPFR_RNDN);
372                 }
373             }
374         }
375         /* Paso 4: actualización de A, B, H y el vector y */
376         for(i = maxIndex+1; i < tam; i++){
377             for(j = ((i-1) < (maxIndex+1)) ? i-1 : maxIndex+1; j >= 0; j--){
378                 if(mpfr_regular_p(H[j][j])){
379                     /* t = H[i][j]/H[j][j] redondeado al entero más próximo */
380                     mpfr_div(t, H[i][j], H[j][j], MPFR_RNDN);
381                     mpfr_rint_round(t, t, MPFR_RNDN);
382                     /* y[j] = y[j] + t*y[i] */
383                     mpfr_fma(y[j], t, y[i], y[j], MPFR_RNDN);
384                     for(k = 0; k <= j; k++){
385                         /* H[i][k] = H[i][k] -t*H[j][k] */
386                         mpfr_fmms(H[i][k], H[i][k], gamma_pow[0], t, H[j][k], MPFR_RNDN);
387                     }
388                     for(k = 0; k < tam; k++){
389                         /* A[i][k] = A[i][k] -t*A[j][k] */
390                         mpfr_fmms(A[i][k], A[i][k], gamma_pow[0], t, A[j][k], MPFR_RNDN);
391                         /* B[k][i] = B[k][i] + t*B[k][j] */
392                         mpfr_fma(B[k][j], t, B[k][i], B[k][j], MPFR_RNDN);
393                     }
394                 }
395             }
396         }
397         /* Paso 5: cálculo de la norma -norma = 1/max_j |H_j|, donde |H_j| es la norma euclídea de
                la fila j */
398         /* Primero se halla max_j |H_j|^2*/
399         for(i = 0; i < tam; i++){
400             mpfr_mul(norma, H[i][0], H[i][0], MPFR_RNDN);
401             for(j = 1; j < tam-1; j++){
402                 mpfr_fma(norma, H[i][j], H[i][j], norma, MPFR_RNDN);
403             }
404             if((i == 0) || (mpfr_greaterequal_p(norma, max))){
405                 mpfr_set(max, norma, MPFR_RNDN);
406             }
407         }
408         /* norma = 1/sqrt(norma) */
409         mpfr_rec_sqrt(max, max, MPFR_RNDN);
```

```
410         /* COmprobación de que el máximo elemento de A no supera la preción (overflow) */
411         for(i = 0; (i < tam) && flag; i++){
412             for(j = 0; (j < tam) && flag; j++){
413                 if(mpfr_greaterequal_p(A[i][j], overflow)){
414                     flag = 0;
415                 }
416             }
417         }
418         /* En caso de overflow o que la norma sea demasiado grande, no se encuentra relación */
419         if(!flag || (mpfr_cmp_ui(max, 30) == 0)){
420             flag = 0;
421             printf("No se ha encontrado ninguna relación.\n");
422         }
423 
424         if(flag){
425             /* Se buscan los valores máximo y mínimo del vector y */
426             mpfr_abs(min, y[0], MPFR_RNDN);
427             mpfr_abs(max, y[0], MPFR_RNDN);
428             for(i = 1, minIndex = 0; i < tam; i++){
429                 mpfr_abs(aux, y[i], MPFR_RNDN);
430                 if(mpfr_less_p(aux, min)){
431                     minIndex = i;
432                     mpfr_set(min, aux, MPFR_RNDN);
433                 } else if(mpfr_greater_p(aux, max)){
434                     maxIndex = i;
435                     mpfr_set(max, aux, MPFR_RNDN);
436                 }
437             }
438             mpfr_div(max, min, max, MPFR_RNDN);
439             /* Si el mínimo de y o la proporción min/max es menor que el umbral, se ha encontrado
                    la relación */
440             if(mpfr_cmp(min, underflow) <= 0 || mpfr_cmp(max, underflow) <= 0){
441                 flag = 0;
442                 /* EL vector a buscado es la correspondiente columna de B */
443                 printf("\nRelación encontrada: [");
444                 for(i = 0; i < tam-1; i++){
445                     mpfr_out_str(stdout, 10, 6, B[i][minIndex], MPFR_RNDN);
446                     printf(", ");
447                 }
448                 mpfr_out_str(stdout, 10, 6, B[tam-1][minIndex], MPFR_RNDN);
449                 printf("]\n");
450             }
451         }
452     }
453     /* Liberación de memoria reservada */
454     for(i = 0; i < tam; i++){
455         mpfr_clears(gamma_pow[i], s[i], x[i], y[i], A[i][tam-1], B[i][tam-1], NULL);
456         for(j = 0; j < tam-1; j++){
457             mpfr_clears(A[i][j], B[i][j], H[i][j], NULL);
458         }
459         free(A[i]);
460         free(H[i]);
461         free(B[i]);
462     }
```

```
463     free(A);
464     free(B);
465     free(H);
466     free(gamma_pow);
467     free(s);
468     free(x);
469     free(y);
470     mpfr_clears(aux, norma, max, min, t, t0, t1, t2, t3, t4, overflow, underflow, NULL);
471     mpfr_free_cache();
472     return 0;
473 }
```

- Algoritmo BBP

Código C.2: Este código implementa el algoritmo BBP por David H. Bailey. Recibe como parámetro un entero positivo n e imprime por pantalla los dígitos hexadecimales de π desde $n + 1$.

```
1  /*
2      This program implements the BBP algorithm to generate a few hexadecimal
3      digits beginning immediately after a given position id, or in other words
4      beginning at position id + 1. On most systems using IEEE 64-bit floating-
5      point arithmetic, this code works correctly so long as d is less than
6      approximately 1.18 x 10^7. If 80-bit arithmetic can be employed, this limit
7      is significantly higher. Whatever arithmetic is used, results for a given
8      position id can be checked by repeating with id-1 or id+1, and verifying
9      that the hex digits perfectly overlap with an offset of one, except possibly
10     for a few trailing digits. The resulting fractions are typically accurate
11     to at least 11 decimal digits, and to at least 9 hex digits.
12 */
13
14 /* David H. Bailey 2006-09-08 */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <stdint.h>
19 #include <math.h>
20 #include <time.h>
21
22 #define NHX 16
23 #define eps 1e-30
24 #define ntp 26
25 #define POW2_64 18446744073709551616.0
```

Continuación del código: función exponenciación binaria modular (3.3). Recibe como argumentos la base b, el exponente e, el módulo m y devuelve b^e mod m. Se ha modificado el código original para que reciba un cuarto parámetro, la máxima potencia de dos menor que e.

```
26  /* Exponenciación modular: expm = 2^p mod (mod) */
27  uint64_t expm (uint64_t p, uint64_t mod, int pow2){
28    int i, size;
29    uint64_t res;
30    double potencia;
31
32    if(mod == 1)
33      return 0;
34
35    /* Si el exponente es pequeño, se calcula directamente */
36    if(p < 16)
37      return ((uint64_t)(pow(16., p))) % mod;
38
39    /* Si no, se usa el algoritmo de exponenciación binaria modular */
40    res = 1;
41    for (i = pow2; i >= 0; i--) {
42      res = (res*res) % mod;
43      if(p & (1 << i))
44        res <<= 4;
45        if(res >= mod)
46          res = res % mod;
47    }
48
49    return res;
50  }
```

Continuación del código: función que computa el valor de cada serie (3.4). Recibe como argumentos el índice del denominador b, la posición id y la máxima potencia de dos hasta id.

```
51  /* sum_k 16^(id-k)/(a*k+b) */
52  double series (int b, int id, int pow2){
53    uint64_t k, t;
54    double ak, p, q, r, s;
55
56    s = 0.;
57    ak = b;
58    p = id;
59
60  /* Suma hasta id */
61
62    for (k = 0; k <= id; k++, p--, ak += 8){
63      /* Se comprueba que la máxima potencia de dos sea menor o igual que p */
64      if(!((uint64_t) p & (1 << pow2))){
65        pow2--;
66      }
67      t = expm (p, ak, pow2);
68      s += (t / ak);
69    }
```

```
70    s -= (int) s;
71
72  /* Se calcunas algunos términos más de la serie con k >= id. */
73    for (k = id + 1; k <= id+100; k++, p--, ak += 8){
74      r = pow (16., p) / ak;
75      if (r < eps) break;
76      s += r;
77    }
78
79    return s;
80  }
```

Continuación del código: función que transforma la parte fraccionaria a hexadecimal. Recibe como argumentos el resultado de la fórmula BBP (3.1) y devuelve los dígitos hexadecimales.

```
81  /* Devuelve en chx, los primeros nhx dígitos hexadecimales de la parte fraccionarioa de x. */
82  void ihex (double x, int nhx, char chx[]){
83    int i;
84    double y;
85    char hx[] = "0123456789ABCDEF";
86
87    y = fabs (x);
88
89    for (i = 0; i < nhx; i++){
90      y = 16. *(y -floor (y));
91      chx[i] = hx[(int) y];
92    }
93  }
```

Continuación del código: función main que implementa el algoritmo BBP (3.5). También imprime el tiempo de ejecución y la parte fraccionaria en decimal.

```
94  void main(int argc, char* argv[])
95  {
96    int aux, i, id, max_pow2;
97    double pid, s1, s2, s3, s4, s5, s6, s7, time_spent;
98    char chx[NHX];
99    clock_t begin, end;
100
101   if(argc != 2) return;
102   begin = clock();
103 /* id es la posición a partir de la cual se obtienen los dígitos hexadecimales de pi. */
104   id = atoi(argv[1]);
105
106 /* max_pow2 es la potencia de dos más grande menor que id -para la exp. modular */
107   max_pow2 = 0;
108   aux = id;
109   while(aux >>= 1)
110     max_pow2++;
```

```
111     s1 = series (1, id, max_pow2);
112     s2 = series (4, id, max_pow2);
113     s3 = series (5, id, max_pow2);
114     s4 = series (6, id, max_pow2);
115
116     pid = 4*s1 -2*s2 -s3 -s4;
117     pid = pid -(int) pid + 1.;
118     ihex (pid, NHX, chx);
119
120       end = clock();
121       time_spent = (double)(end -begin) / CLOCKS_PER_SEC;
122
123     printf ("␣position␣=␣%i\n␣fraction␣=␣%.15f␣\n␣hex␣digits␣=␣␣%10.10s\n␣Tiempo␣=␣%f\n",
124     id, pid, chx, time_spent);
125     exit(0);
126   }
```

- Algoritmo de Bellard

  Se reutiliza el código del algoritmo BBP, cambiando la función que calcula el valor de las series y el main. Los prototipos de ambos métodos se mantienen.

```
1   /* sum_k 2^(4*id+power-10k)/(a*k+b) */
2   double series (int a, int b, int id, int power, int pow2){
3     short sgn = 1;
4     uint64_t k, t;
5     double ak, p, q, r, s;
6
7     s = 0.;
8     ak = b;
9     p = 4*id+power;
10    q = p/10;
11
12    /* Se ajusta el valor de la máxima potencia de dos para ser menor o igual que 4*id+power */
13    if((uint64_t) p & (1 << (pow2 + 1))){
14      pow2++;
15    } else if(!((uint64_t) p & (1 << pow2))){
16      pow2--;
17    }
18
19  /* Suma hasta floor(4*id+power/10) */
20
21    for (k = 0; k <= q; k++, p -= 10, ak += a, sgn *= (-1)){
22      /* Se comprueba que la máxima potencia de dos sea menor o igual que p */
23      if(!((uint64_t) p & (1 << pow2))){
24        pow2--;
25      }
26      t = expm (p, ak, pow2);
27      s += sgn*(t / ak);
28    }
```

```
29      s -= (int) s;
30
31  /* Se calcunas algunos términos más de la serie con k >= id. */
32    for (k = q + 1; k <= q+100; k++, p -= 10, ak += a, sgn *= (-1)){
33      r = pow (2., p) / ak;
34      if (r < eps) break;
35      s += sgn*r;
36    }
37
38    return s;
39  }
```

Código C.3: Función main que implementa el algoritmo de Bellard (3.7). También imprime el tiempo de ejecución y la parte fraccionaria en decimal.

```
1   void main(int argc, char* argv[])
2   {
3     int aux, i, id, max_pow2;
4     double pid, s1, s2, s3, s4, s5, s6, s7, time_spent;
5     char chx[NHX];
6       clock_t begin, end;
7
8     if(argc != 2) return;
9     begin = clock();
10
11    /* id es la posición a partir de la cual se obtienen los dígitos hexadecimales de pi. */
12    id = atoi(argv[1]);
13
14    /* max_pow2 es la máxima potencia de dos menor o igual que 4*id -para la exp. modular. */
15    max_pow2 = 2;
16    aux = id;
17    while(aux >>= 1)
18      max_pow2++;
19
20    s1 = series (4, 1, id, -1, max_pow2);
21    s2 = series (4, 3, id, -6, max_pow2);
22    s3 = series (10, 1, id, 2, max_pow2);
23    s4 = series (10, 3, id, 0, max_pow2);
24    s5 = series (10, 5, id, -4, max_pow2);
25    s6 = series (10, 7, id, -4, max_pow2);
26    s7 = series (10, 9, id, -6, max_pow2);
27
28    pid = s3 + s7 -s1 -s2 -s4 -s5 -s6;
29    pid = pid -(int) pid + 1.;
30    ihex (pid, NHX, chx);
31
32    end = clock();
33    time_spent = (double)(end -begin) / CLOCKS_PER_SEC;
34
35    printf (" position = %i\n fraction = %.15f \n hex digits = %10.10s\n Tiempo = %f\n",
36    id, pid, chx, time_spent);
37    exit(0);
38  }
```

D

Demostración: fórmulas BBP y Bellard

D.1. Demostración de la fórmula BBP

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right)$$

Demostración. La prueba se basa en la siguiente igualdad entre el sumatorio y la integral:

$$\sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{1}{8i+n}\right) = \sum_{i=0}^{\infty} \sqrt{2}^n \frac{x^{8i+n}}{8i+n}\Big|_0^{1/\sqrt{2}} = \sqrt{2}^n \sum_{i=0}^{\infty} \int_0^{1/\sqrt{2}} x^{8i+n-1} =$$

$$\sqrt{2}^n \int_0^{1/\sqrt{2}} \sum_{i=0}^{\infty} x^{8i+n-1} = \sqrt{2}^n \int_0^{1/\sqrt{2}} x^{n-1} \sum_{i=0}^{\infty} (x^8)^i = \sqrt{2}^n \int_0^{1/\sqrt{2}} \frac{x^{n-1}}{1-x^8}$$

Aplicando este resultado a la fórmula BBP:

$$\sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right) = \int_0^{1/\sqrt{2}} \frac{4\sqrt{2} - 8x^3 - 4\sqrt{2}x^4 - 8x^5}{1-x^8} dx \overset{y=\sqrt{2}x}{=}$$

$$16\int_0^1 \frac{4 - 2y^3 - y^4 - y^5}{16 - y^8} dy = 16\int_0^1 \frac{(y-1)(-y^4 - 2y^3 - 4y^2 - 4y - 4)}{(y^4 - 2y^3 + 4y - 4)(-y^4 - 2y^3 - 4y^2 - 4y - 4)} dy =$$

$$16\int_0^1 \frac{y-1}{y^4 - 2y^3 + 4y - 4} dy = 16\int_0^1 \frac{y-1}{(y^2-2)(y^2-2y+2)} = \int_0^1 \frac{4y}{y^2-2} + \int_0^1 \frac{8-4y}{y^2-2y+2} =$$

$$2ln(|y^2-2|)\Big|_0^1 - 2ln(|y^2-2y+2|)\Big|_0^1 + \int_0^1 \frac{4}{y^2-2y+2} = -2ln(2) + 2ln(2) + 4\int_0^1 \frac{1}{1+(y-1)^2} =$$

$$4arctan(y-1)\Big|_0^1 = -4arctan(-1) = \pi$$

□

D.2. Demostración de la fórmula de Bellard

$$\pi = \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{2^{10n}}\left(-\frac{2^5}{4n+1}-\frac{1}{4n+3}+\frac{2^8}{10n+1}-\frac{2^6}{10n+3}-\frac{2^2}{10n+5}-\frac{2^2}{10n+7}+\frac{1}{10n+9}\right)$$

Demostración. Antes de pasar a la demostración, se necesitan algunas propiedades de los números complejos y de la función arcotangente:

$$z = x + iy = re^{i\theta} \rightarrow ln(z) = lnr + i\theta \rightarrow Im(ln(z)) = \theta = arctan\left(\frac{x}{y}\right)$$

De aquí: $$arctan\left(\frac{1}{a-1}\right) = Im\left(ln\left(1+\frac{i-1}{a}\right)\right);$$

A partir del desarrollo de Taylor del logaritmo neperiano:

$$ln(1+u) = \sum_{n=0}^{\infty}\frac{(-1)^n}{n+1}u^{n+1} \rightarrow ln\left(1+\frac{i-1}{a}\right) = \sum_{n=0}^{\infty}\frac{(-1)^n}{n+1}\left(\frac{i-1}{a}\right)^{n+1} =$$

$$\sum_{n=0}^{\infty}\frac{(-1)^n}{n+1}\left(\frac{\sqrt{2}e^{3\pi i/4}}{a}\right)^{n+1} = \sum_{n=0}^{\infty}\frac{(-1)^n\sqrt{2}^{n+1}}{(n+1)a^{n+1}}\left(\cos\frac{3\pi(n+1)}{4}+i\sin\frac{3\pi(n+1)}{4}\right)$$

Tomando la parte imaginaria:

$$Im\left(ln\left(1+\frac{i-1}{a}\right)\right) = \sum_{n=0}^{\infty}\frac{(-1)^n\sqrt{2}^{n+1}}{(n+1)a^{n+1}}\left(\sin\frac{3\pi(n+1)}{4}\right) =$$

$$\sum_{n=0}^{\infty}\left(\frac{\sqrt{2}^{8n+1}}{(8n+1)a^{8n+1}}\frac{\sqrt{2}}{2}+\frac{\sqrt{2}^{8n+2}}{(8n+2)a^{8n+2}}+\frac{\sqrt{2}^{8n+3}}{(8n+3)a^{8n+3}}\frac{\sqrt{2}}{2}\right) -$$

$$\sum_{n=0}^{\infty}\left(\frac{\sqrt{2}^{8n+5}}{(8n+5)a^{8n+5}}\frac{\sqrt{2}}{2}+\frac{\sqrt{2}^{8n+6}}{(8n+6)a^{8n+6}}+\frac{\sqrt{2}^{8n+7}}{(8n+7)a^{8n+7}}\frac{\sqrt{2}}{2}\right) =$$

$$\sum_{n=0}^{\infty}\left(\frac{2^{4n}}{(8n+1)a^{8n+1}}+\frac{2^{4n+1}}{(8n+2)a^{8n+2}}+\frac{2^{4n+1}}{(8n+3)a^{8n+3}}\right) -$$

$$\sum_{n=0}^{\infty}\left(\frac{2^{4n+2}}{(8n+5)a^{8n+5}}+\frac{2^{4n+3}}{(8n+6)a^{8n+6}}+\frac{2^{4n+3}}{(8n+7)a^{8n+7}}\right) =$$

$$\sum_{n=0}^{\infty}\frac{(-1)^n2^{2n}}{a^{4n+3}}\left(\frac{a^2}{4n+1}+\frac{2a}{4n+2}+\frac{2}{4n+3}\right) = (*)$$

Por otra parte, el desarrollo de taylor de la arcotangente es:

$$arctan(x) = \sum_{n=0}^{\infty}\frac{(-1)^nx^{2n+1}}{2n+1} \Rightarrow arctan(\frac{1}{x}) = \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)x^{2n+1}} = (**)$$

Además, se tiene esta identidad entre π y la arcotangente:

$$\frac{\pi}{4} = 2arctan\left(\frac{1}{2}\right) - arctan\left(\frac{1}{7}\right)$$

Sustituyendo en $(*)$ $a = 8$ y en $(**)$ $x = 2$ y reordenando los términos:

$$\pi = 8\sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)2^{2n+1}} - 4\sum_{n=0}^{\infty}\frac{(-1)^n 2^{2n}}{8^{4n+3}}\left(\frac{64}{4n+1} + \frac{16}{4n+2} + \frac{2}{4n+3}\right) = \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{2^8}{10n+1}\right)$$

$$+ \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{-2^6}{10n+3}\right) + \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{2^4}{10n+5}\right) + \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{-2^2}{10n+7}\right)$$

$$+ \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{1}{10n+9}\right) + \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{1024^n}\left(\frac{-20}{10n+5}\right) + \frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{2^{10n}}\left(-\frac{2^5}{4n+1} - \frac{1}{4n+3}\right) =$$

$$\frac{1}{64}\sum_{n=0}^{\infty}\frac{(-1)^n}{2^{10n}}\left(-\frac{2^5}{4n+1} - \frac{1}{4n+3} + \frac{2^8}{10n+1} - \frac{2^6}{10n+3} - \frac{2^2}{10n+5} - \frac{2^2}{10n+7} + \frac{1}{10n+9}\right)$$

□

E

Codigo:

TEST ESTADÍSTICOS

En este apéndice se muestran los códigos ejecutados en un notebook de Jupyter para realizar los análisis estadísticos sobre la normalidad y aleatoriedad de los dígitos de π, cuyos resultados se han expuesto en el capítulo 4.

E.1. Procesamiento de los dígitos de π

Código E.1: Declaración de las variables que almacenarán los datos procesados de cada test.

```
# In[1]:


import numpy as np
import dill


# ## Recogida de datos

# In[2]:


# Variables para posterior análisis
# Hay 20 de cada contador porque se analiza cada 10^5 dígitos hexadecimales
# Equivalentes a 4*10^5 dígitos binarios y aprox. 1.2*10^5 dígitos decimales
    # chi_cuadrado: n de veces que aparece cada dígito
chi_cuadrado2 = np.zeros((20, 2), dtype=int)
chi_cuadrado10 = np.zeros((20, 10), dtype=int)
chi_cuadrado16 = np.zeros((20, 16), dtype=int)

    # serial: n de veces que aparece cada pareja (x[i], x[i+1])
serial2 = np.zeros((20, 2, 2), dtype=int)
serial10 = np.zeros((20, 10, 10), dtype=int)
serial16 = np.zeros((20, 16, 16), dtype=int)

    # rachas con respecto a la mediana: n de veces que la secuencia supera
        superiormente/inferiormente a la mediana
n_mayor2 = np.zeros(20, dtype=int)
```

```
28  n_menor2 = np.zeros(20, dtype=int)
29  n_rachas_mediana2 = np.zeros(20, dtype=int)
30
31  n_mayor10 = np.zeros(20, dtype=int)
32  n_menor10 = np.zeros(20, dtype=int)
33  n_rachas_mediana10 = np.zeros(20, dtype=int)
34
35  n_mayor16 = np.zeros(20, dtype=int)
36  n_menor16 = np.zeros(20, dtype=int)
37  n_rachas_mediana16 = np.zeros(20, dtype=int)
38
39      # rachas con respecto al crecimiento: n de veces que crece/decrece la secuencia y n de rachas
40  # Sólo se consideran bases 10 y 16, porque en binario equivale al test anterior
41  n_crece10 = np.zeros(20, dtype=int)
42  n_decrece10 = np.zeros(20, dtype=int)
43  n_rachas_creciente10 = np.zeros(20, dtype=int)
44
45  n_crece16 = np.zeros(20, dtype=int)
46  n_decrece16 = np.zeros(20, dtype=int)
47  n_rachas_creciente16 = np.zeros(20, dtype=int)
48
49      # autocorrelaciones: se almacena cada dígito normalizado menos la media (x[i]/base -0.5)
50  secuencia_normalizada2 = np.empty(8000000)
51  secuencia_normalizada10 = np.empty(2400000)
52  secuencia_normalizada16 = np.empty(2000000)
53
54      # poker: n de 4-tuplas que contienen 1, 2, 3 ó4 número(s) distinto(s)
55  n_distintos_4_10 = np.zeros((20, 4), dtype=int)
56  n_distintos_4_16 = np.zeros((20, 4), dtype=int)
57
58      # poker: n de 5-tuplas que contienen 1, 2, 3, 4 ó5 número(s) distinto(s)
59  n_distintos_5_10 = np.zeros((20, 5), dtype=int)
60  n_distintos_5_16 = np.zeros((20, 5), dtype=int)
```

Código E.2: Esta función procesa los dígitos de π dados el fichero, el número de dígitos del fichero, la base, la última racha con respecto a la mediana, con respecto al crecimiento y el último dígito del fichero anterior. Retorna las variables actualizadas.

```
1  # In[3]:
2
3
4  def datos(digitos, n_digitos, base, racha_actual_mediana=True, racha_actual_creciente=True,
       digito_previo=None):
5      # Inicialización de las variables
6      chi_cuadrado = np.zeros(base, dtype=int) # contador de apariciones de cada dígito
7      serial = np.zeros((base, base), dtype=int) # contador de apariciones de cada pareja de dígitos
8      mediana = (base-1)/2.0
9      n_mayor = 0 # n de dígitos superiores a la mediana
10     n_menor = 0 # n de dígitos inferiores a la mediana
11     n_rachas_mediana = 0 # n de rachas (dígitos consecutivos superiores o inferiores a la mediana)
```

```
12      secuencia_normalizada = np.empty(n_digitos) # dígitos normalizados para el test de
            autocorrelaciones
13      if(base > 2):
14          n_crece = 0 # n de veces que crece la secuencia (x[i] > x[i-1])
15          n_decrece = 0 # n de veces que decrece la secuencia (x[i] < x[i-1])
16          n_rachas_creciente = 1 # n de rachas (dígitos consecutivos cada uno mayor/menor que el
                anterior)
17          n_distintos_4 = np.zeros(4, dtype=int) # n de 4-tuplas con 1, 2, 3 ó4 dígitos iguales
18          n_distintos_5 = np.zeros(5, dtype=int) # n de 5-tuplas con 1, 2, 3, 4 ó5 dígitos iguales
19
20      x1 = int(digitos[0], base) # primer dígito
21      chi_cuadrado[x1] += 1 # se actualiza el correspondiente contador
22      secuencia_normalizada[0] = x1*1.0/(base-1.0) -0.5 # se normaliza y se guarda
23      if(base > 2):
24          n_distintos_4[len(set(digitos[0:4]))-1] += 1 # se cuenta el n de dígitos distintos de la primera
                4-tupla
25          n_distintos_5[len(set(digitos[0:5]))-1] += 1 # se cuenta el n de dígitos distintos de la primera
                5-tupla
26
27      # Se actualizan los contadores de la mediana
28      n_mayor += (x1 > mediana)
29      n_menor += (x1 < mediana)
30      # Si no hay dígito previo, se empieza en uno, pues es la racha actual. Si lo hay, se continua la
            anterior racha
31      n_rachas_mediana += (digito_previo is None) or ((x1 > mediana) ^ racha_actual_mediana)
32      racha_actual_mediana = (x1 > mediana) # True si x > mediana, False en caso contrario
33
34      if digito_previo and (base > 2):
35          n_rachas_creciente = 0
36          crece = (x1 >= digito_previo) # Se comprueba si este dígito es mayor que el anterior
37          n_crece += crece # Si lo fuera, la sucesión crece
38          n_decrece += not crece # Si no, la sucesión decrece
39          n_rachas_creciente += (crece ^ racha_actual_creciente) # Si la racha ha cambiado, se
                incrementa su contador
40          racha_actual_creciente = crece # Se actualiza la racha actual
41
42      # Se repite este mismo proceso para cada dígito
43      for i in range(1, n_digitos):
44          x2 = int(digitos[i], base)
45          chi_cuadrado[x2] += 1
46          secuencia_normalizada[i] = x2/(base-1.0) -0.5
47
48          if i % 2:
49              serial[x1, x2] += 1
50
51          if base > 2:
52              if not (i % 4):
53                  n_distintos_4[len(set(digitos[i:i+4]))-1] += 1
54              if not (i % 5):
55                  n_distintos_5[len(set(digitos[i:i+5]))-1] += 1
56
57              crece = (x2 >= x1)
58              n_crece += crece
59              n_decrece += not crece
60              n_rachas_creciente += (crece ^ racha_actual_creciente)
61              racha_actual_creciente = crece
```

```
62              mayor_mediana = (x2 > mediana)
63              n_mayor += mayor_mediana
64              n_menor += not mayor_mediana
65              n_rachas_mediana += (mayor_mediana ^ racha_actual_mediana)
66              racha_actual_mediana = mayor_mediana
67
68              x1 = x2
69
70          if base == 2:
71              return chi_cuadrado, serial, secuencia_normalizada, n_mayor, n_menor,
                    n_rachas_mediana, racha_actual_mediana, int(digitos[-1], 16)
72          else:
73              return chi_cuadrado, serial, secuencia_normalizada, n_distintos_4, n_distintos_5,
                    n_mayor, n_menor, n_rachas_mediana, n_crece, n_decrece, n_rachas_creciente,
                    racha_actual_mediana, racha_actual_creciente, int(digitos[-1], base)
```

Código E.3: Esta celda llama al método anterior para procesar cada uno de los 60 ficheros de manera independiente.

```
1   # In[4]:
2
3
4   # Variables auxiliares para guardar la última racha y el último dígito de cada fichero
5   r2m, r10m, r16m = True, True, True
6   r2c, r10c, r16c = True, True, True
7   d2, d10, d16 = None, None, None
8
9   # Análisis de la secuencia de dígitos de cada fichero individualmente
10  for i in range(20):
11      s = "./base2/" + str(400000*i+1) + "-" + str(400000*(i+1)) + ".txt"
12      with open(s, "r") as f:
13          digitos = f.read()
14          chi_cuadrado2[i], serial2[i], s2, n_mayor2[i], n_menor2[i], n_rachas_mediana2[i], r2m, d2 =
                datos(digitos, 400000, 2, r2m, r2c, d2)
15          if i == 0:
16              secuencia_normalizada2 = s2
17          else:
18              secuencia_normalizada2 = np.append(secuencia_normalizada2, s2)
19
20      s = "./base10/" + str(120000*i+1) + "-" + str(120000*(i+1)) + ".txt"
21      with open(s, "r") as f:
22          digitos = f.read()
23          chi_cuadrado10[i], serial10[i], s10, n_distintos_4_10[i], n_distintos_5_10[i], n_mayor10[i],
                n_menor10[i], n_rachas_mediana10[i], n_crece10[i], n_decrece10[i],
                n_rachas_creciente10[i], r10m, r10c, d10 = datos(digitos, 120000, 10, r10m, r10c, d10)
24          if i == 0:
25              secuencia_normalizada10 = s10
26          else:
27              secuencia_normalizada10 = np.append(secuencia_normalizada10, s10)
28
29      s = "./base16/" + str(100000*i+1) + "-" + str(100000*(i+1)) + ".txt"
30      with open(s, "r") as f:
31          digitos = f.read()
```

```
32          chi_cuadrado16[i], serial16[i], s16, n_distintos_4_16[i], n_distintos_5_16[i], n_mayor16[i],
                n_menor16[i], n_rachas_mediana16[i], n_crece16[i], n_decrece16[i],
                n_rachas_creciente16[i], r16m, r16c, d16 = datos(digitos, 100000, 16, r16m, r10c, d16)
33          if i == 0:
34              secuencia_normalizada16 = s16
35          else:
36              secuencia_normalizada16 = np.append(secuencia_normalizada16, s16)
```

Código E.4: A cada contador se le suma la información de los dígitos previos para tener los datos acumulados desde el primer dígito hasta el n-ésimo.

```
1   # In[5]:
2
3
4   # Cada contador se actualiza sumando los resultados de los dígitos previos
5   # para tener los datos desde el primer dígito hasta el n-ésimo
6   for i in range(1, 20):
7       chi_cuadrado2[i] += chi_cuadrado2[i-1]
8       chi_cuadrado10[i] += chi_cuadrado10[i-1]
9       chi_cuadrado16[i] += chi_cuadrado16[i-1]
10
11      serial2[i] += serial2[i-1]
12      serial10[i] += serial10[i-1]
13      serial16[i] += serial16[i-1]
14
15      n_distintos_4_10[i] += n_distintos_4_10[i-1]
16      n_distintos_4_16[i] += n_distintos_4_16[i-1]
17
18      n_distintos_5_10[i] += n_distintos_5_10[i-1]
19      n_distintos_5_16[i] += n_distintos_5_16[i-1]
20
21      n_mayor2[i] += n_mayor2[i-1]
22      n_menor2[i] += n_menor2[i-1]
23      n_rachas_mediana2[i] += n_rachas_mediana2[i-1]
24
25      n_mayor10[i] += n_mayor10[i-1]
26      n_menor10[i] += n_menor10[i-1]
27      n_rachas_mediana10[i] += n_rachas_mediana10[i-1]
28
29      n_mayor16[i] += n_mayor16[i-1]
30      n_menor16[i] += n_menor16[i-1]
31      n_rachas_mediana16[i] += n_rachas_mediana16[i-1]
32
33      n_crece10[i] += n_crece10[i-1]
34      n_decrece10[i] += n_decrece10[i-1]
35      n_rachas_creciente10[i] += n_rachas_creciente10[i-1]
36
37      n_crece16[i] += n_crece16[i-1]
38      n_decrece16[i] += n_decrece16[i-1]
39      n_rachas_creciente16[i] += n_rachas_creciente16[i-1]
```

```
40  dill.dump_session('Datos.db')
```

E.2. Análisis de los datos

Código E.5: Importación de los datos y de las librerías utilizadas para el análisis.

```
1  import numpy
2  import pandas as pd
3  import dill
4  dill.load_session('Datos.db') # Se recuperan las variables de la recogida de datos
```

- Distribución chi-cuadrado (χ^2)

Código E.6: Esta función calcula el valor de la distribución chi-cuadrado (χ^2) conforme a la ecuación 4.1. Recibe el número de observaciones, las observaciones, el número de categorías y sus probabilidades.

```
1  def chi_square_test(n, obs, n_obs, probs):
2      chi_square = 0.0
3      for i in range(n_obs):
4          chi_square += ((obs[i] -n*probs[i])**2) / (n*probs[i])
5      return chi_square
```

- Test de frecuencias

Código E.7: Esta celda analiza los datos relativos al test de frecuencias.

```
1  tabla_chi = [3.841, 16.92, 25]
2  chi = np.empty((10,8), dtype=np.dtype('U10'))
3  for j in range(2):
4      for i in range(10):
5          chi[i][4*j] = str(100000*(10*j+i+1))
6          chi[i][4*j+1] = str(chi_square_test(400000*(10*j+i+1), chi_cuadrado2[10*j+i], 2, 2*[1/2]))
7          chi[i][4*j+2] = str(chi_square_test(120000*(10*j+i+1), chi_cuadrado10[10*j+i], 10,
               10*[1/10]))
8          chi[i][4*j+3] = str(chi_square_test(100000*(10*j+i+1), chi_cuadrado16[10*j+i], 16,
               16*[1/16]))
9  df = pd.DataFrame(chi, columns=['Núm._dígitos', 'Base_2', 'Base_10', 'Base_16', 'Núm._dígitos_',
       'Base_2_', 'Base_10_', 'Base_16_'])
10 df.style.apply(lambda x: ["" if (i % 4) == 0 else ("background:_red" if float(x[i]) >
       tabla_chi[(i%4)-1] else "background:_lightgreen") for i in range(8)], axis = 1)
```

- Test serial

Código E.8: Esta celda analiza los datos relativos al test serial.

```
# ## Serial

# Este test comprueba la uniformidad de cada par de dígitos. Es decir, cada pareja de números debe aparecer N/(b^2) veces, donde N es el número de dígitos y b la base

# In[5]:


chi = np.empty((10,8), dtype=np.dtype('U10'))
s2 = np.zeros(20)
for j in range(2):
    s2 += [chi_square_test(200000*(i+1), serial2[i, j], 2, 2*[1/4]) for i in range(20)]

s10 = np.zeros(20)
for j in range(10):
    s10 += [chi_square_test(60000*(i+1), serial10[i, j], 10, 10*[1/100]) for i in range(20)]

s16 = np.zeros(20)
for j in range(16):
    s16 += [chi_square_test(50000*(i+1), serial16[i, j], 16, 16*[1/256]) for i in range(20)]

for j in range(2):
    for i in range(10):
        chi[i][4*j] = str(100000*(10*j+i+1))
        chi[i][4*j+1] = s2[10*j+i]
        chi[i][4*j+2] = s10[10*j+i]
        chi[i][4*j+3] = s16[10*j+i]

tabla_chi = [7.815, 123.23, 293.248]
df = pd.DataFrame(chi, columns=['Núm._dígitos', 'Base_2', 'Base_10', 'Base_16', 'Núm._dígitos_', 'Base_2_', 'Base_10_', 'Base_16_'])
df.style.apply(lambda x: ["" if (i%4) == 0 else ("background:_red" if float(x[i]) > tabla_chi[(i%4)-1] else "background:_lightgreen") for i in range(8)], axis = 1)
```

- Test de la mano de poker

Código E.9: Esta celda analiza los datos relativos al test poker en grupos de cuatro dígitos.

```
# ## Poker

# Este test comprueba que el número de k-tuplas que tiene d dígitos distintos se aproxime a lo estadísticamente esperado en una secuencia aleatoria

# ### 4-tuplas

# In[6]:


n_combinatorios = np.array([1, 7, 6, 1])
prob10 = np.zeros(4)
prob10[0] = 1.0
for i in range(1, 4):
    prob10[i] = prob10[i-1]*(10-i)
prob10 /= (10**3)
p10 = [chi_square_test(30000*(i+1), n_distintos_4_10[i], 4, prob10*n_combinatorios) for i in range(20)]

prob16 = np.zeros(4)
prob16[0] = 1.0
for i in range(1, 4):
    prob16[i] = prob16[i-1]*(16-i)
prob16 /= (16**3)
p16 = [chi_square_test(25000*(i+1), n_distintos_4_16[i], 4, prob16*n_combinatorios) for i in range(20)]

chi = np.empty((10,6), dtype=np.dtype('U10'))
for j in range(2):
    for i in range(10):
        chi[i][3*j] = str(100000*(10*j+i+1))
        chi[i][3*j+1] = p10[10*j+i]
        chi[i][3*j+2] = p16[10*j+i]

df = pd.DataFrame(chi, columns=['Núm. dígitos', 'Base 10', 'Base 16', 'Núm. dígitos ', 'Base 10 ', 'Base 16 '])
df.style.apply(lambda x: ["" if (i%3) == 0 else ("background: red" if float(x[i]) > 7.815 else "background: lightgreen") for i in range(6)], axis = 1)
```

Código E.10: Esta celda analiza los datos relativos al test poker en grupos de cinco dígitos.

```
# ### 5-tuplas

# In[7]:


n_combinatorios = np.array([1, 15, 25, 10, 1])
prob10 = np.zeros(5)
prob10[0] = 1.0
for i in range(1, 5):
    prob10[i] = prob10[i-1]*(10-i)
prob10 /= (10**4)
p10 = [chi_square_test(24000*(i+1), n_distintos_5_10[i], 5, prob10*n_combinatorios) for i in
    range(20)]

prob16 = np.zeros(5)
prob16[0] = 1.0
for i in range(1, 5):
    prob16[i] = prob16[i-1]*(16-i)
prob16 /= (16**4)
p16 = [chi_square_test(20000*(i+1), n_distintos_5_16[i], 5, prob16*n_combinatorios) for i in
    range(20)]

chi = np.empty((10,6), dtype=np.dtype('U10'))
for j in range(2):
    for i in range(10):
        chi[i][3*j] = str(100000*(10*j+i+1))
        chi[i][3*j+1] = p10[10*j+i]
        chi[i][3*j+2] = p16[10*j+i]

df = pd.DataFrame(chi, columns=['Núm. dígitos', 'Base 10', 'Base 16', 'Núm. dígitos ', 'Base 10 ',
    'Base 16 '])
df.style.apply(lambda x: ["" if (i%3) == 0 else ("background: red" if float(x[i]) > 9.488 else
    "background: lightgreen") for i in range(6)], axis = 1)
```

- Test de rachas

Código E.11: Esta celda analiza los datos relativos al test de rachas con respecto al crecimiento.

```
# ## Rachas

# ### Respecto al crecimiento

# Una secuencia aleatoria no debe tener patrones de crecimiento o decrecimiento. Es decir, debe estar en un punto intermedio entre ser monótona y alternar. Si no, se podría predecir que el siguiente dígito es mayor/menor que el anterior.

# In[8]:


N = np.array([120000*i for i in range(1, 21)])
media10 = (2.0 *N -1)/3
desviacion10 = np.sqrt((16*N-29) / 90)
z10 = np.absolute(n_rachas_creciente10 -media10) / desviacion10

N = N = np.array([100000*i for i in range(1, 21)])
media16 = (2.0 *N -1)/3
desviacion16 = np.sqrt((16*N-29) / 90)
z16 = np.absolute(n_rachas_creciente16 -media16) / desviacion16

chi = np.empty((10,6), dtype=np.dtype('U10'))
for j in range(2):
    for i in range(10):
        chi[i][3*j] = str(100000*(10*j+i+1))
        chi[i][3*j+1] = z10[10*j+i]
        chi[i][3*j+2] = z16[10*j+i]
df = pd.DataFrame(chi, columns=['Núm._dígitos', 'Base_10', 'Base_16', 'Núm._dígitos_', 'Base_10_', 'Base_16_'])
df.style.apply(lambda x: ["" if (i%3) == 0 else ("background:_red" if float(x[i]) > 1.645 else "background:_lightgreen") for i in range(6)], axis = 1)
```

Código E.12: Esta celda analiza los datos relativos al test de rachas con respecto a la mediana.

```
# ### Respecto a la mediana

# Al igual que con el crecimiento, tampoco se debe poder predecir si el siguiente número es mayor/menor que el valor medio. La secuencia no debe ser siempre mayor/menor o alternar un dígito mayor con otro menor.

# In[10]:


N = n_mayor2 + n_menor2
media2 = 1.0 + (2.0 *n_mayor2 *n_menor2)/N
varianza2 = (2.0 *(n_mayor2 / N) *(n_menor2 / N)) *((2.0 *n_mayor2 *n_menor2 -N)/(N-1))
z2 = np.absolute(n_rachas_mediana2 -media2) / np.sqrt(varianza2)

N = n_mayor10 + n_menor10
media10 = 1.0 + (2.0 *n_mayor10 *n_menor10)/N
varianza10 = (2.0 *(n_mayor10 / N) *(n_menor10 / N)) *((2.0 *n_mayor10 *n_menor10 -N)/(N-1))
z10 = np.absolute(n_rachas_mediana10 -media10) / np.sqrt(varianza10)

N = n_mayor16 + n_menor16
media16 = 1.0 + (2.0 *n_mayor16 *n_menor16)/N
varianza16 = (2.0 *(n_mayor16 / N) *(n_menor16 / N)) *((2.0 *n_mayor16 *n_menor16 -N)/(N-1))
z16 = np.absolute(n_rachas_mediana16 -media16) / np.sqrt(varianza16)

chi = np.empty((10,8), dtype=np.dtype('U10'))
for j in range(2):
    for i in range(10):
        chi[i][4*j] = str(100000*(10*j+i+1))
        chi[i][4*j+1] = z2[10*j+i]
        chi[i][4*j+2] = z10[10*j+i]
        chi[i][4*j+3] = z16[10*j+i]
df = pd.DataFrame(chi, columns=['Núm. dígitos', 'Base 2', 'Base 10', 'Base 16', 'Núm. dígitos ', 'Base 2 ', 'Base 10 ', 'Base 16 '])
df.style.apply(lambda x: ["" if (i%4) == 0 else ("background: red" if float(x[i]) > 1.645 else "background: lightgreen") for i in range(8)], axis = 1)
```

- Test de autocorrelación

Código E.13: Función que calcula la autocorrelación entre los dígitos n y $n + k$ (4.2). Recibe como argumentos el número de dígitos, el lag k y los dígitos normalizados.

```
1  def autocovarianza(n, k, digitos):
2      s = 0.0
3      for i in range(n-k):
4          s += digitos[i]*digitos[i+k]
5      return s/(n-k)
```

Esta celda calcula la autocorrelación entre los dígitos en binario, para lags entre 1 y 10.

```
1  r2 = [[autocovarianza(400000*(i+1), k, secuencia_normalizada2[:400000*(i+1)]) for k in range(1,11)]
       for i in range(20)]
2  chi = np.array(r2)
3  chi = np.round(chi, 7)
4  chi = np.array(chi, dtype=np.dtype('U10'))
5  df = pd.DataFrame({'Test autocorrelacion': [str(400000*(i+1)) for i in range(20)]})
6  df = pd.concat([df, pd.DataFrame(chi, columns=['Lag '+str(i+1) for i in range(10)])], axis=1)
7  df.style.apply(lambda x: ["" if i == 0 else ("background: red" if float(x[i]) -
       1.96/(12*((float(x[0])-i)**0.5)) > 0 or float(x[i]) + 1.96/(12*((float(x[0])-i)**0.5)) < 0 else
       "background: lightgreen") for i in range(11)], axis = 1)
```

Esta celda calcula la autocorrelación entre los dígitos en decimal, para lags entre 1 y 10.

```
1  r10 = [[autocovarianza(120000*(i+1), k, secuencia_normalizada10[:120000*(i+1)]) for k in range(1,
       11)] for i in range(20)]
2  chi = np.array(r10)
3  chi = np.round(chi, 7)
4  chi = np.array(chi, dtype=np.dtype('U10'))
5  df = pd.DataFrame({'Test autocorrelacion': [str(120000*(i+1)) for i in range(20)]})
6  df = pd.concat([df, pd.DataFrame(chi, columns=['Lag '+str(i+1) for i in range(10)])], axis=1)
7  df.style.apply(lambda x: ["" if i == 0 else ("background: red" if float(x[i]) -
       1.96/(12*((float(x[0])-i)**0.5)) > 0 or float(x[i]) + 1.96/(12*((float(x[0])-i)**0.5)) < 0 else
       "background: lightgreen") for i in range(11)], axis = 1)
```

Esta celda calcula la autocorrelación entre los dígitos en hexadecimal, para lags entre 1 y 10.

```
r16 = [[autocovarianza(100000*(i+1), k, secuencia_normalizada16[:100000*(i+1)]) for k in range(1, 11)] for i in range(20)]
chi = np.array(r16)
chi = np.round(chi, 7)
chi = np.array(chi, dtype=np.dtype('U10'))
df = pd.DataFrame({'Test autocorrelacion': [str(100000*(i+1)) for i in range(20)]})
df = pd.concat([df, pd.DataFrame(chi, columns=['Lag '+str(i+1) for i in range(10)])], axis=1)
df.style.apply(lambda x: ["" if i == 0 else ("background: red" if float(x[i]) - 1.96/(12*((100000*i)**0.5)) > 0 or float(x[i]) + 1.96/(12*((100000*i-i)**0.5)) < 0 else "background: lightgreen") for i in range(11)], axis = 1)
```

F

Primeros dígitos de π

F.1. Primeros dígitos de π en binario

$\pi = 11.00100100001111110110101010001000100001011010001100001000110100110001001100$
$011001100010100010111000000011011100000111001101000100101001000000100100111000000100$
$010001010011001111100110001110100000000100000101110111110101001100011101100010011100$
$110110010001001010001010010100000100001111001100011100011010000000100110111011100$
$111110010101000110011011001111001101001110100100001100011011001100000010101100001010$
$011011011111100100101111100010100001101110100111111100001001101010110110101101101010$
$100011100001001000101111001001000010110110101011101100110001001011110011111101100000$
$110111101000100110001000010111010011010011000110111111011010110101100001011111111111$
$010111001011011011110100000001101011011111101101111011100011100001101011111111011010$
$110101000100110011111101001011010111010011111001001000001000101111100010010110001111$
$111110011001001001001010000110011001010001111011001110010001011011001111011100001000$
$000000001111100101110001010000101100011101111110000010110011000110110100100100000011$
$011000011100010101011101001110011010011010010001011000111111101010001111110100100010$
$011001111010111111000001101100101010111010010001111011100101000111010110110010110000$
$011100011000101111001101010110001000001000010101010010101110111001111011010101001010$
$001000001110111000010010110100101100110110101100111000011000011010101001110010010100$
$101111001001100000000100111100010111010001101100000010001100101000011000001000010110$
$111000011001010010000010111100100011000101110001101101100111000111011111000111001111$
$100111011100101100000110000000111010000110000000111001101100100111100000111010001010$
$110110000000111101000101000111111011010111000101010111011111000001101111010011000101$
$001011001001110111100010101111001011111101101001010101011000000101110001100000011100$
$110010101010010010111110011101010100101010110101011100101000101011101001000100110000$
$011000100110001111101000000101000100000001010101110010100011100101101010001010101010$
$010110001000010110110101101001100110001011100001101000001000101000001111010001100110$
$101010000101010100100001101010111101111000111001011101001100100111011001111101110000$
$001010000010001011000110110111111011110000101010001010111010100111000101010111010111$
$010000011000001100011111011011001110010111000011111000010110100110111000011110010 \ldots$

F.2. Primeros dígitos de π en decimal

$\pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640628$
$620899862803482534211706798214808651328230664709384460955058223172535940812848111 74$
$502841027019385211055596446229489549303819644288109756659334461284756482337867831 65$
$271201909145648566923460348610454326648213393607260249141273724587006606315588174 88$
$152092096282925409171536436789259036001133053054882046652138414695194151160943305 72$
$703657595919530921861173819326117931051185480744623799627495673518857527248912279 38$
$183011949129833673362440656643086021394946395224737190702179860943702770539217176 29$
$317675238467481846766940513200056812714526356082778577134275778960917363717872146 84$
$409012249534301465495853710507922796892589235420199561121290219608640344181598136 29$
$774771309960518707211349999998372978049951059731732816096318595024459455346908302 64$
$252230825334468503526193118817101000313783875288658753320838142061717766914730359 82$
$534904287554687311595628638823537875937519577818577805321712268066130019278766111 95$
$909216420198938095257201065485863278865936153381827968230301952035301852968995773 62$
$259941389124972177528347913151557485724245415069595082953311686172785588907509838 17$
$546374649393192550604009277016711390098488240128583616035637076601047101819429555 96$
$198946767837449448255379774726847104047534646208046684259069491293313677028989152 10$
$475216205696602405803815019351125338243003558764024749647326391419927260426992279 67$
$823547816360093417216412199245863150302861829745557067498385054945885869269956909 27$
$210797509302955321165344987202755960236480665499119881834797753566369807426542527 86$
$255181841757467289097777279380008164706001614524919217321721477235014144197356854 81$
$613611573525521334757418494684385233239073941433345477624168625189835694855620992 19$
$222184272550254256887671790494601653466804988627232791786085784383827967976681454 10$
$095388378636095068006422512520511739298489608412848862694560424196528502221066118 63$
$067442786220391949450471237137869609563643719172874677646575739624138908658326459 95$
$813390478027590099465764078951269468398352595709825822620522489407726719478268482 60$
$147699090264013639443745530506820349625245174939965143142980919065925093722169646 15$
$157098583874105978859597729754989301617539284681382686838689427741559918559252459 53$
$959431049972524680845987273644695848653836736222626099124608051243884390451244136 54$
$976278079771569143599770012961608944169486855584840635342207222582848864815845602 85$
$060168427394522674676788952521385225499546667278239864565961163548862305774564980 35$
$593634568174324112515076069479451096596094025228879710893145669136867228748940560 10$
$150330861792868092087476091782493858900971490967598526136554978189312978482168299 89$
$487226588048575640142704775551323796414515237462343645428584447952658678210511413 54$
$735739523113427166102135969536231442952484937187110145765403590279934403742007310 57$
$853906219838744780847848968332144571386875194350643021845319104848100537061468067 49$
$192781911979399520614196634287544406437451237181921799983910159195618146751426 9\ldots$

F.3. Primeros dígitos de π en hexadecimal

$$\pi = 3.243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E6C894$$
$$52821E638D01377BE5466CF34E90C6CC0AC29B7C97C50DD3F84D5B5B54709179216D5D$$
$$98979FB1BD1310BA698DFB5AC2FFD72DBD01ADFB7B8E1AFED6A267E96BA7C904$$
$$5F12C7F9924A19947B3916CF70801F2E2858EFC16636920D871574E69A458FEA3F4933D7$$
$$E0D95748F728EB658718BCD5882154AEE7B54A41DC25A59B59C30D5392AF26013C5D1$$
$$B023286085F0CA417918B8DB38EF8E79DCB0603A180E6C9E0E8BB01E8A3ED71577C1$$
$$BD314B2778AF2FDA55605C60E65525F3AA55AB945748986263E8144055CA396A2AAB10B$$
$$6B4CC5C341141E8CEA15486AF7C72E993B3EE1411636FBC2A2BA9C55D741831F6CE5$$
$$C3E169B87931EAFD6BA336C24CF5C7A325381289586773B8F48986B4BB9AFC4BFE81B$$
$$6628219361D809CCFB21A991487CAC605DEC8032EF845D5DE98575B1DC262302EB651B$$
$$8823893E81D396ACC50F6D6FF383F442392E0B4482A484200469C8F04A9E1F9B5E21C66$$
$$842F6E96C9A670C9C61ABD388F06A51A0D2D8542F68960FA728AB5133A36EEF0B6C13$$
$$7A3BE4BA3BF0507EFB2A98A1F1651D39AF017666CA593E82430E888CEE8619456F9FB$$
$$47D84A5C33B8B5EBEE06F75D885C12073401A449F56C16AA64ED3AA62363F77061BFE$$
$$DF72429B023D37D0D724D00A1248DB0FEAD349F1C09B075372C980991B7B25D479D8F$$
$$6E8DEF7E3FE501AB6794C3B976CE0BD04C006BAC1A94FB6409F60C45E5C9EC2196A$$
$$246368FB6FAF3E6C53B51339B2EB3B52EC6F6DFC511F9B30952CCC814544AF5EBD0$$
$$9BEE3D004DE334AFD660F2807192E4BB3C0CBA85745C8740FD20B5F39B9D3FBDB5$$
$$579C0BD1A60320AD6A100C6402C7279679F25FEFB1FA3CC8EA5E9F8DB3222F83C751$$
$$6DFFD616B152F501EC8AD0552AB323DB5FAFD23876053317B483E00DF829E5C57BB$$
$$CA6F8CA01A87562EDF1769DBD542A8F6287EFFC3AC6732C68C4F5573695B27B0BBC$$
$$A58C8E1FFA35DB8F011A010FA3D98FD2183B84AFCB56C2DD1D35B9A53E479B6F84$$
$$565D28E49BC4BFB9790E1DDF2DAA4CB7E3362FB1341CEE4C6E8EF20CADA36774C$$
$$01D07E9EFE2BF11FB495DBDA4DAE909198EAAD8E716B93D5A0D08ED1D0AFC725$$
$$E08E3C5B2F8E7594B78FF6E2FBF2122B648888B812900DF01C4FAD5EA0688FC31CD$$
$$1CFF191B3A8C1AD2F2F2218BE0E1777EA752DFE8B021FA1E5A0CC0FB56F74E818A$$
$$CF3D6CE89E299B4A84FE0FD13E0B77CC43B81D2ADA8D9165FA2668095770593CC731$$
$$4211A1477E6AD206577B5FA86C75442F5FB9D35CFEBCDAF0C7B3E89A0D6411BD3AE$$
$$1E7E4900250E2D2071B35E226800BB57B8E0AF2464369BF009B91E5563911D59DFA6AA$$
$$78C14389D95A537F207D5BA202E5B9C5832603766295CFA911C819684E734A41B3472DCA$$
$$7B14A94A1B5100529A532915D60F573FBC9BC6E42B60A47681E6740008BA6FB5571BE9$$
$$1FF296EC6B2A0DD915B6636521E7B9F9B6FF34052EC585566453B02D5DA99F8FA108B$$
$$A47996E85076A4B7A70E9B5B32944DB75092EC4192623AD6EA6B049A7DF7D9CEE60B$$
$$88FEDB266ECAA8C71699A17FF5664526CC2B19EE1193602A575094C29A0591340E4183A$$
$$3E3F54989A5B429D656B8FE4D699F73FD6A1D29C07EFE830F54D2D38E6F0255DC14C$$
$$DD20868470EB266382E9C6021ECC5E09686B3F3EBAEFC93C9718146B6A70A1687F \ldots$$


UAM
UNIVERSIDAD AUTONOMA
DE MADRID