# Comparación de implementaciones Numéricas

**Raudel Alejandro Gómez Molina** RAUDELA.GOMEZ@ESTUDIANTES.MATCOM.UH.CU
*Grupo C211*

**Tutor(es):**
Msc. Fernando Raul Rodríguez Flores, *Facultad de Matemática y Computación, Universidad de La Habana*

## Resumen

Este trabajo tiene como objetivo proponer una implementación de una aritmética basada en las aritméticas de punto fijo que toma como base a las potencias de 10 (siempre y cuando dicha base sea soportada por la aritmética de enteros con que cuenta nuestra computadora). Además compararemos su desempeño en distintos escenarios comparándola con los resultados de otras aritméticas de coma flotante, punto fijo y fracciones racionales.

## Abstract

This paper aims to propose an implementation of an arithmetic based on fixed-point arithmetic that takes the powers of 10 as a base (as long as said base is supported by the integer arithmetic available to our computer). We will also compare its performance in different scenarios comparing it with the results of other floating point, fixed point and rational fractions arithmetic.

**Palabras Clave:** Implementaciones Numéricas, Aritméticas Computacionales.

## 1. Introducción

El sistema de coma flotante es una notación científica que usan las computadoras para representar números reales muy pequeños y a la vez muy grandes con relativa rapidez y eficiencia.

El modelo por el cual se basan estas aritméticas es el estándar IEEE 754, este formato comprende:

1. Representación de números finitos: cada número finito se caracteriza por una base $b$, un bit de signo $s$, una mantisa $c$ y un exponente $q$. Entonces un número $v$ se define como: $v = (-1)^s cb^q$.
2. Los números infinitos: $\pm\infty$.
3. Un valor no numérico: $NaN$ el cual indica que en algún proceso de cálculo realizado se efectuó una operación que no resultó en un número correctamente definido, por ejemplo una división por 0.

El IEEE 754 también establece las reglas básicas para operar con los números de la aritmética tal es el caso de las de la representación en memoria de los números, los formatos de precisión y los formatos de intercambio y redondeo.

Las aritméticas de punto fijo son un método para representar números reales, guardando solamente un número fijo de sus decimales. Esta representación se basa en la utilización de potencias negativas de una misma base para representar la parte fraccionaria de un número. Esta aritmética contrasta con la aritmética de punto flotante anteriormente descrita, ya que es mucho menos compleja de implementar, pero también menos efectiva. Dado que las antiguas calculadoras mecánicas contaban con este sistema, actualmente se utiliza el sistema de coma flotante en los modernos microprocesadores.

Las aritméticas abordadas anteriormente en muchas ocasiones no pueden representar exactamente los números que les proporcionamos ya que solo pueden guardar una parte finita del número, por lo que realmente operaremos con aproximaciones de estos números, lo que conlleva a la consideración de los errores de truncamiento y redondeo que se cometen durante este proceso.

Las aritméticas de fracciones racionales se basan en la representación de números racionales mediante una fracción racional que consta de un numerador y un denominador enteros. Dicha aritmética puede minimizar los errores de redondeo y truncamiento que se cometen en las aritméticas anteriormente descritas, pero en varias ocasiones su computo suele ser bastante más complejo que en las anteriores.

Otro de los puntos a considerar en los sistemas descritos anteriormente es que la mayoría de ellos trabajan en base 2, lo cual se debe a la implementación de los microprocesadores, entonces otro error que nos podemos plantear en este análisis es el intercambio entre bases, ya que comúnmente trabajamos en base 10 y nuestra computadora en base 2, este problema es el que trataremos de resolver con la aritmética propuesta en este trabajo.

## 2. Biblioteca BigNum

BigNum es una aritmética basada en las aritméticas de punto fijo, ya que cuenta con una cantidad fija de lugares decimales, la cual se puede cambiar dinámicamente desde el código una vez que se instancia. Utilizaremos como base las potencias de 10 (siempre y cuando dicha base sea representable en la aritmética de enteros con que cuenta nuestra computadora), para no cometer en los errores que se producen durante el intercambio de bases los cuales ilustraremos posteriormente.

```
def __init__(self, precision=6,
             ind_base10: int = 9):

    self.__precision = precision
    self.__base10 = 10 ** ind_base10
    self.__ind_base10 = ind_base10
```

Figura 1: Constructor de BigNum.

Para instanciar BigNum necesitaremos los parámetros **precision** el número de lugares decimales con respecto a nuestra base y **ind_base10** el exponente de la base 10 que tendrá la base de la aritmética. Notemos que la cantidad total de lugares decimales que puede representar nuestra aritmética es la multiplicación de **precision** y **ind_base10**, para los parámetros por defecto tenemos 54 lugares decimales.

### 2.1 Representación de los Números

Modelaremos los número de nuestra aritmetica mediante la clase **Numbers**:

```
def __init__(self,
             number: str|float|list,
             positive: bool,
             precision: int,
             ind_base10: int,
             base10: int):
```

Figura 2: Constructor de Numbers.

Los parámetros **number**, **positive**, **precision**, **ind_base10** y **base10** reciben la representación del número, el signo del número, la precisión, exponete de la base 10 y la base respectivamente. Notemos que la representación de los números puede estar dada por una variable del tipo **list**, **float** o **str**, en los dos últimos casos transformamos el dicho formato a una lista de enteros que finalmente será la representación en memoria del número. Cada posición de la lista representa el bit correspondiente a esa posición y almacena un número entero $x$ tal que $0 \leq x < base$.

### 2.2 Operaciones Básicas

Las operaciones básicas que podemos realizar con los números de la clase **Numbers** son la adición, substracción, multiplicación y división en el formato del sistema numérico posicional con que actualmente trabajamos. Como dichas operaciones se producen bit a bit y dado que nuestros bits son números enteros y conocemos que la aritmética de la computadora es exacta, aprovecharemos dicha ventaja y modificaremos la base de nuestra aritmética a potencias de 10 (que soporte la aritmética entera) para acelerar el computo con respecto a un sistema de numeración de base 10, ya que entonces para computar las operaciones entre números de igual cantidad de decimales en base 10 necesitaremos menos iteraciones bit a bit.

Para la adición y substracción utilizaremos el algoritmo clásico de suma y resta bit a bit llevando los respectivos acarreos.

#### 2.2.1 Multiplicación algortimo de Karatsuba

El algoritmo de Karatsuba consiste en reducir la multiplicación de dos números de $n$ dígitos a como máximo $3n^{\log_2 3} \approx 3n^{1.585}$ multiplicaciones de números de un solo dígito, lo cual mejora el algoritmo clásico de multiplicación que consta de $n^2$ operaciones de un solo dígito. Este algoritmo pertenece a la clase de los algoritmos de *Divide y Vencerás* utilizando el caso particular de la partición binaria.

A continuación, presentamos una descripción del algoritmo:

Sean $a$ y $b$ dos números de $n$ dígitos tal que:

$$\begin{aligned} &a = a_1 10^{\lfloor \frac{n}{2} \rfloor} + a_2 \wedge b = b_1 10^{\lfloor \frac{n}{2} \rfloor} + b_2 \\ &\Rightarrow ab = a_1 b_1 10^{2 \cdot \lfloor \frac{n}{2} \rfloor} + (a_1 b_2 + b_1 a_2) 10^{\lfloor \frac{n}{2} \rfloor} + a_2 b_2 \end{aligned} \tag{1}$$

Entonces tenemos que $a_1$ y $b_1$ poseen a lo sumo $\lceil \frac{n}{2} \rceil$ dígitos y $a_2$ y $b_2$ tienen $\lfloor \frac{n}{2} \rfloor$ dígitos, por lo que para realizar el computo de $ab$ necesitamos calcuar 4 multiplicaciones y una suma de a lo sumo $\lceil \frac{n}{2} \rceil$ dígitos.

Este algoritmo es particularmente muy eficiente en multiplicaciones de enteros grandes, por ejemplo para $n = 1024$, el costo del algoritmo sería $3^{10} = 59049$ comparado con el clásico que sería de $2^{20} = 1049756$ operaciones.

#### 2.2.2 División algortimo D

Ahora nos enfrentaremos a un problema más complejo que es la división de dos números. El algoritmo básico para la división consiste en mecanismo de prueba y error mediante el sistema de numeración posicional y el mismo tiene una complejidad de $dmn$, donde $d$ es una contante en la aritmética que viene dada por la base y $m$ y $n$ son la cantidad de dígitos del numerador y el denominador. Por tanto como anteriormente planteamos la idea de aumentar el tamaño de la base para que las operaciones fueran más eficientes aquí nos hemos encontrado con un problema que se agrava con dicha propuesta.

Para solucionar el problema descrito anteriormente plantearemos dos algoritmos como alternativa al algoritmo clásico el primero es el *Algoritmo de Newtón-Raphlson* y el segundo es el *Algoritmo D*.

Newtón-Raphlson es un algoritmo de división que pertenece a los algoritmos de división rápidos y que

actualmente se usa en los métodos de división de los microprocesadores modernos. Este algoritmo consiste en la aproximación del inverso del divisor mediante una serie de iteraciones.

Sean $x_0, x_1, \ldots, x_s$ aproximaciones del recíproco del divisor($d$) y $f(x) = dx - 1$:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \frac{\frac{1}{x_i} - d}{\frac{1}{x_i^2}} = x_i(2 - dx_i) \quad (2)$$

Como nuestro objetivo es brindar una aritmética lo más exacta posible no consideraremos este algoritmo para su implementación, pero a continuación abordamos un algoritmo menos eficiente, pero en el que no cometemos errores de aproximación fuera de los errores de truncamiento y redondeo.

El Algoritmo D se basa en la estimación del cociente que resulta de dividir un número de $n+1$ dígitos entre otro de $n$ dígitos. Esta estimación se realiza mediante la división del número formado por los dos primeros dígitos del dividendo entre el número formado por el primer dígito del divisor, a continuación mostraremos como acotar dicha estimación:

Sea $a$ un número de $n+1$ dígitos y $b$ un número de $n$ dígitos y $d$ la base de nuestra aritmética tales que:

$$a = \overline{x_{n+1}x_n \ldots x_1} \wedge b = \overline{y_n \ldots y_1}$$

Primero como $a - (\overline{x_{n+1}x_n})d^{n-1} = \overline{x_{n-1} \ldots x_1} \leq y_n d^{n-1}$ tenemos que:

$$\left\lfloor \frac{a}{y_n d^{n-1}} \right\rfloor = \left\lfloor \frac{\overline{x_{n+1}x_n} d^{n-1}}{y_n d^{n-1}} \right\rfloor \quad (3)$$

Ahora sea $q = \lfloor \frac{a}{b} \rfloor$ y $q' = \lfloor \frac{\overline{x_{n+1}x_n}}{y_n} \rfloor$, sabemos que $q \geq q'$ por lo demostrado anteriormente, por lo tanto trataremos de acotar el error que se comete en la estimación de $q'$:

$$\begin{aligned}
q - q' &= \left\lfloor \frac{a}{y_n d^{n-1}} \right\rfloor - \left\lfloor \frac{a}{y_n d^{n-1} + d^n - 1} \right\rfloor \\
q - q' &< \frac{a}{y_n d^{n-1}} - \frac{a}{y_n d^{n-1} + d^n - 1} + 1 \\
q - q' &< \frac{a(y_n d^{n-1} + d^{n-1} - 1 - y_n d^{n-1})}{y_n d^{n-1}(y_n d^{n-1} + d^{n-1} - 1)} + 1 \quad (4) \\
q - q' &< \frac{d^{n-1} - 1}{y_n d^{n-1}} \cdot \frac{a}{y_n d^{n-1} + d^{n-1} - 1} + 1 \\
q - q' &< \frac{1}{y_n} \cdot d + 1 = \frac{d}{y_n} + 1
\end{aligned}$$

Luego solo nos queda acotar $\frac{d}{y_n}$, mostremos que siempre podemos hacer transformaciones para que $\frac{d}{y_n} \leq 2$, para ello apoyémonos en el sistema decimal que es el que utilizamos en nuestra aritmética.

Por ejemplo si $y_n$ es 2 podemos multiplicar el numerador y el denominador por 3, al cual llamaremos factor de normalización, en el caso peor que el denominador sea $\overline{299\ldots} \cdot 3 = \overline{6\ldots}$, de la misma manera podemos encontrar un factor de normalización para los restantes dígitos. Para extender este ejemplo para la base $10^k$ simplemente multiplicamos por $10^x$ de tal manera que $y_n$ tenga exactamente $k$ dígitos y luego aplicamos el proceso descrito anteriormente.

De esta manera comprobamos que al realizar la estimación de nuestro cociente cometemos a lo sumo un error de 3 unidades, por lo que tendremos que hacer a lo sumo 3 comprobaciones para asegurarnos de que el resultado es correcto. Luego nuestro algoritmo realizará $3mn$ iteraciones en el caso peor al realizar la división de un número de $m$ dígitos entre otro de $n$ dígitos.

## 3. Biblioteca ArihmeticMath

Para proceder a la comparación de los desempeños de las aritméticas vamos a establecer una biblioteca con implementaciones de las principales funciones matemáticas. Como plantilla base para las aritméticas definiremos la clase **ArithmeticMath** que cuenta de los siguientes métodos:

```
@abstractmethod
def number1(self):
    pass

@abstractmethod
def number0(self):
    pass

@abstractmethod
def float_to_number(self,
                    number: float):
    pass
```

Figura 3: Métodos de ArithmeticMath.

Los métodos de la figura 3 tienen la función de definir el número 1 y el número 0 para cada aritmética, dado un **float** convertirlo al formato de la aritmética y convertir un número de la aritmética a entero, respectivamente.

```
def number_to_fraction(self, number):
    s = str(number).split('.')

    if len(s) == 1:
        return int(s[0]), 1

    return int(s[0] + s[1]),\
           int(add_zeros_right('1',\
           len(s[1])))
```

Figura 4: Método NumberToFraction.

Adicionalmente se encuentra predefinido el método de la Figura 4, el cual se encarga de convertir un número de la aritmética en una fracción racional, retornando 2 entero el numerador y el denominador en ese orden.

### 3.1 Funciones de ArihmeticMath

Como mencionamos anteriormente esta aritmética cuenta con métodos que permiten la implementación de las principales funciones matemáticas, a continuación, detallamos los recursos matemáticos utilizados en cada caso:

1. Raíz n-ésima: Para esta operación se trata de buscar un valor cercano mediante una potencia entera ($x_0$ aproximación inicial) y luego se aproxima mediante la siguiente relación de recurrencia:

$$x_{k+1} = \frac{1}{n}\left[(n-1)x_k + \frac{A}{x_k^{n-1}}\right] \tag{5}$$

   donde $x_i \approx \sqrt[n]{A}$.

2. Potencia: Para esta operación se busca la fracción que genera el exponente y luego se calcula la raíz y la potencia correspondiente.

3. Logaritmo en base $e$: Se aproxima mediante la serie de Taylor de la función $\ln(1-x)$, con $|x| \leq 1$:

$$\ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} \tag{6}$$

   si $|x| > 1$ se utiliza la siguiente identidad $\ln(\frac{1}{x}) = -\ln(x)$.

4. Logaritmo: Se aproxima mediante la identidad $\log_a b = \frac{\ln(a)}{\ln(b)}$, con el cálculo de los logaritmos en base $e$ correspondientes.

5. Seno: Se aproxima mediante la serie de Taylor de la función sen $x$:

$$\text{sen } x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} \tag{7}$$

6. Coseno: Se aproxima mediante la serie de Taylor de la función cos $x$:

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \tag{8}$$

7. Tangente y Cotangente: Se calculan mediante las identidades fundamentales: $\frac{\text{sen x}}{\cos \text{x}}$ y $\frac{\cos \text{x}}{\text{sen x}}$, respectivamente.

8. Arcoseno: Se aproxima mediante la serie de Taylor de la función arcsen $x$, para $|x| < 1$:

$$\text{arcsen } x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)}\, x^{2n+1} \tag{9}$$

9. Arcocoseno: Se aproxima mediante la identidad arccos $x = \frac{\pi}{2} - \text{arcsin } x$, con el cálculo del arcoseno correspondiente.

10. Arcotangente: Se aproxima mediante la serie de Taylor de la función arctan $x$, con $|x| \leq 1$, si $|x| > 1$ se utiliza la siguiente identidad arctan $x = \frac{\pi}{2} - \arctan(\frac{1}{x})$:

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}\, x^{2n+1} \tag{10}$$

11. Arcocotangente: Se aproxima mediante la identidad arccot $x = \frac{\pi}{2} - \arctan x$, con el cálculo de la arcotangente correspondiente.

12. Número $\pi$: Se aproxima mediante la serie la función arcsen $x$ evaluada en 0.5 que tiene como resultado $\frac{\pi}{6}$.

13. Número $e$: Se aproxima mediante la serie de Taylor de la función $e^x$, evaluada en 1:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{11}$$

Todos estos métodos anteriormente descritos tienen presentes errores numéricos de truncamiento, ya que la mayoría de ellos se basan en series numéricas o sucesiones recurrentes, los cuales son infinitos y solo se puede computar una parte de estos.

## 4. Comparando las Aritméticas

En esta sección contaremos con 3 aritméticas, cada una basadas en **ArithmeticMath**:

1. BigNum: Nuestra biblioteca con las especificaciones mencionadas anteriormente.

2. DecimalNum: Apoyada en el módulo **Decimal** de **Python**, para realizar las operaciones básicas.

3. FractionNum: Apoyada en el módulo **Fraction** de **Python**, para realizar las operaciones básicas.

### 4.1 Problemas y Limitaciones del Punto Flotante

Como mencionamos anteriormente en la introducción el sistema de coma flotante que se implementa en los microprocesadores de nuestra computadora es en base 2, el número $0.25 = \frac{2}{10} + \frac{5}{100} = \frac{25}{100} = \frac{1}{4}$ de ahí que se 0.25 en binario se exprese como 0.01.

Un problema que podemos observar a simple vista es que en muchas ocasiones la fracción decimal no se pueda expresar mediante un número finito de decimales, por ejemplo $\frac{1}{3} = 0.333333\ldots$, ahora, lo mismo ocurre en base 2, por ejemplo $0.4 = \frac{4}{10} = \frac{2}{5}$ que en binario seria $\frac{10}{101} = 0.0110011001100110\ldots$. Con el ejemplo analizado anteriormente se nos presenta el siguiente problema $0.4 = \frac{4}{10}$, sin embargo $\frac{10}{101} = 0.0110011001100110\ldots$, es decir para la misma fracción racional nos encontramos con que 0.4 es perfectamente representable en base 10 con un número finito de decimales pero en base 2 obtenemos

```
>>> format(0.4,'.17f')
'0.40000000000000002'
```

Figura 5: Representación de 0.4 como float.

```
>>> 0.4*3==1.2
False
>>> format(0.4*3,'.17f')
'1.20000000000000018'
>>> format(1.2,'.17f')
'1.19999999999999996'
```

Figura 6: Comparación de números flotantes.

$0.0110011001100110\ldots$ que es un número infinito periódico. Esto supone un importante problema ya que en la vida cotidiana fuera del mundo de la computación se trabaja en base 10, sin embargo, las computadoras y los sistemas computo están hechos en su mayoría para el procesamiento de datos de la vida real, lo cual trae consigo que verdaderamente se trabaje con una aproximación de estos datos y no con los números introducidos originalmente (Figura 5).

Otro problema que debemos de tener en cuenta a la hora de trabajar con estas aproximaciones es la comparación de los números obtenidos, por ejemplo, ilustremos la comparación mostrada en la Figura 6 donde podemos apreciar que en el sistema de coma flotante $0.4 \cdot 3$ es ligeramente superior a 1.2. Por este motivo a la hora de trabajar con los números flotantes en el diseño de nuestros algoritmos siempre debemos tener en cuenta este error y evitar hacer comparaciones del tipo $a == b$, donde $a$ y $b$ son números flotantes.

### 4.2 Análisis de la Comparación

En esta sección analizaremos los resultados de los cálculos obtenidos al probar las distintas aritméticas con las funciones implementadas en **ArihmeticMath**, en cuanto a precisión numérica y tiempo de ejecución, dichos resultados se muestran en la sección **Tablas y Valores Obtenidos**.

#### 4.2.1 Principales Funciones Matemáticas

En la Figura 8 podemos observar las aproximaciones para $\text{sen}(\pi)$, por nuestros conocimientos matemáticos podemos percatarnos que el resultado es 0, sin embargo, observamos que las 3 aproximaciones arrojan valores ligeramente mayores a 0, con más exactitud para **FractionNum** y una mayor velocidad para **DecimalNum**.

En la Figura 9 se comparan las aproximaciones para los números $\pi$ y $e$, respectivamente, con una mayor cantidad de cifras significativas para nuestra aritmética, aunque también con el tiempo más elevado, mientras **DecimalNum** sigue contando con el menor tiempo.

Pasamos a la Figura 10 donde comparamos el cálculo de $\ln(e)$, podemos percatarnos de que **FractionNum** es la única que nos proporciona el resultado preciso, mientras nuestra aritmética cuenta con más exactitud que **DecimalNum**.

Ahora procederemos a comparar potencias con índices decimales y raíces n-ésimas una tarea que puede ser computacionalmente más compleja, ya que como vimos anteriormente lleva el cálculo de potencias que pueden ser potencialmente más costosas que los factoriales utilizados en las series trigonométricas.

En la Figura 12 se muestras el cálculo de $\sqrt{2}$, donde se aprecia una mayor exactitud en cuanto a decimales para nuestra aritmética pero para **FractionNum** ha sido imposible realizar el computo dentro de los límites de tiempo concebidos.

En la Figura 13 se muestran los valores obtenidos para $\pi^{0.5}$, con mejor exactitud y velocidad para **DecimalNum**, al igual que en el análisis anterior ha sido imposible realizar los cálculos para **FractionNum**.

#### 4.2.2 Comparando números

En esta sección analizaremos algunos de los ejemplos planteados en la sección donde describimos las limitaciones y los problemas de la coma flotante.

| | $0.4 \cdot 3 == 1.2$ | $\sqrt{2}^3 == 2^{1.5}$ |
|---|---|---|
| BigNum | True | True |
| FractionNum | False | - |
| DecimalNum | False | True |
| | $1.5 \cdot 2 == 0.1 + 0.2$ | $6/10 == 0.6$ |
| BigNum | True | True |
| FractionNum | False | False |
| DecimalNum | False | False |

Figura 7: Comparando números.

En la Figura 7 podemos observar algunos de los ejemplos planteados en secciones anteriores, donde se demuestra la exactitud de nuestra biblioteca antes estos escenarios y los problemas del sistema de coma flotante.

### 4.3 Tablas y Valores Obtenidos

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | sen($\pi$) | 0.00000000000000023846264338327950288419718393539484659 | 0.0276847 |
| FractionNum | sen($\pi$) | 1.2246467991473532e-16 | 0.00103831 |
| DecimalNum | sen($\pi$) | 1.22464679915052570174645718 9E-16 | 6.19888e-05 |
| BigNum | sen(1) | 0.841470984807896506652502321630298999622563060798371067 | 0.0127552 |
| FractionNum | sen(1) | 0.8414709848078965 | 0.000446081 |
| DecimalNum | sen(1) | 0.841470984807896506652502 3216 | 5.31673e-05 |
| BigNum | cos(10) | -0.8390715290764492840940149626033131647956690 11492621151 | 0.0178885 |
| FractionNum | cos(10) | -0.8390715290764493 | 0.000496149 |
| DecimalNum | cos(10) | -0.8390715290764492840940147683 | 5.45979e-05 |
| BigNum | cos($\frac{\pi}{2}$) | 0.00000000000000001923132169163975144209858469968755 1724 | 0.0247827 |
| FractionNum | cos($\frac{\pi}{2}$) | 6.123233995736766e-17 | 0.00112844 |
| DecimalNum | cos($\frac{\pi}{2}$) | 6.123233995815319195807292141E-17 | 6.1512e-05 |

Figura 8: Cálculo de funciones trigonométricas.

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | $\pi$ | 3.14159265358979323846264338327950288419716939937510 5588 | 0.174473 |
| FractionNum | $\pi$ | 3.141592653589793 | 0.00294471 |
| DecimalNum | $\pi$ | 3.14159265358979323846264338 1 | 0.000238657 |
| BigNum | $e$ | 2.71828182845904523536028747135266249775724709369870 3317 | 0.0109069 |
| FractionNum | $e$ | 2.718281828459045 | 0.000371218 |
| DecimalNum | $e$ | 2.71828182845904523536028747 4 | 4.31538e-05 |

Figura 9: Cálculo de $\pi$ y $e$.

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | ln(34) | 3.52419384921878386701702406113107778059677205595807 5561 | 0.0921957 |
| FractionNum | ln(34) | 3.5241938492187836 | 0.00212646 |
| DecimalNum | ln(34) | 3.52419384921878386701702405 4 | 0.000117302 |
| BigNum | ln($e$) | 0.99999999999999991341578897108873753987036614796049 2922 | 0.0782981 |
| FractionNum | ln($e$) | 1.0 | 0.0125442 |
| DecimalNum | ln($e$) | 0.99999999999999994681762293 40 | 0.000138283 |

Figura 10: Cálculo de ln($x$).

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | atan(20) | 3.09163425786785047705263709624465800270589231895159 8412 | 0.293755 |
| FractionNum | atan(20) | 3.0916342578678506 | 0.155865 |
| DecimalNum | atan(20) | 3.09163425786785047705263709 4 | 0.0010519 |
| BigNum | asin(1) | 1.51435402843709682885517941913612288118566118025774 8239 | 0.130738 |
| FractionNum | asin(1) | 1.5143540284370969 | 0.00287724 |
| DecimalNum | asin(1) | 1.51435402843709682885517942 3 | 0.000198841 |

Figura 11: Cálculo de funciones trigonométricas inversas.

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | $\sqrt[12]{45}$ | 1.373307239498335812855818924624660567276647132735873217 | 0.305376 |
| FractionNum | $\sqrt[12]{45}$ | - | - |
| DecimalNum | $\sqrt[12]{45}$ | 1.373307239498335812855818925 | 0.000169277 |
| BigNum | $\sqrt{2}$ | 1.414213562373095048801688724209698078569671875376948073 | 0.0293512 |
| FractionNum | $\sqrt{2}$ | - | - |
| DecimalNum | $\sqrt{2}$ | 1.414213562373095048801688724 | 0.00041008 |

Figura 12: Cálculo de raíces n-ésimas.

| | Cálculo de: | Valor | Tiempo |
|---|---|---|---|
| BigNum | $\pi^{0.5}$ | 1.772453850905515960029097752286351896216738465831096675 | 0.0269032 |
| FractionNum | $\pi^{0.5}$ | - | - |
| DecimalNum | $\pi^{0.5}$ | 1.772453850905515992751519103 | 8.70228e-05 |
| BigNum | $3^{4.5}$ | 140.296115413079060775723153661975661722367225558640830762 | 0.0319946 |
| FractionNum | $3^{4.5}$ | - | - |
| DecimalNum | $3^{4.5}$ | 140.296115413079060775723154 0 | 9.46522e-05 |

Figura 13: Cálculo de potencias con índices no enteros.

### 4.4 Sacando lo máximo de nuestra aritmética

En esta sección mostraremos los resultados obtenidos en el desempeño de nuestra aritmética para tareas computacionalmente complejas, como la aproximación de $\pi$ y $e$ con varias cifras decimales y el cálculo de números grandes como los factoriales.

| Cifras Correctas | Tiempo |
|---|---|
| 100 | 1.357109 |
| 200 | 5.720520 |
| 500 | 30.96164 |

Figura 14: Aproximación de los decimales de $\pi$.

| Cifras Correctas | Tiempo |
|---|---|
| 100 | 0.047280 |
| 200 | 0.127560 |
| 500 | 0.887334 |

Figura 15: Aproximación de los decimales de $e$.

| Factorial | Tiempo |
|---|---|
| 100 | 0.027119 |
| 1000 | 3.804981 |
| 10000 | 457.1413 |

Figura 16: Cálculo de factoriales.

Además mostremos la utilización de nuestra aritmética en algoritmos iterativos para la resolución de ecuaciones lineales, para ello resolvamos el siguiente sistema de ecuaciones por el **Método de Jacobi** y el **Método de Gauss-Seidel**.

$$\left(\begin{array}{ccc|c} 9 & 3 & 4 & 1 \\ -3 & 7 & -3 & 1 \\ 2 & 2 & -10 & 1 \end{array}\right) \tag{12}$$

Obteniendo como resultado el siguiente vector para el **Método de Jacobi**:

$$\begin{pmatrix} 0.08246073298429319371727748691099476439790 \\ 0.15575916230366492146596858638743455497382 \\ -0.05235602094240837696335078534031413612565 \end{pmatrix}$$

con un tiempo de ejecución de 1.8970699310302734 segundos.

Obteniendo como resultado el siguiente vector para el **Método de Gauss-Seidel**:

$$\begin{pmatrix} 0.08246073298429319371727748691099476439790 \\ 0.15575916230366492146596858638743455497382 \\ -0.05235602094240837696335078534031413612565 \end{pmatrix}$$

con un tiempo de ejecución de 1.7851269245147705 segundos.

### 4.5 Finalizando Comparaciones

En las diferentes secciones abordadas de este tema, nos percatamos de que los resultados favorecen en cuanto a precisión favorecen a **BigNum** aunque por un estrecho margen. Sin embargo en el caso de los cálculos de expresiones decimales no finitas todas las aritméticas ofrecen comportamientos similares, con algunas ventajas para **FractionNum** como vimos en un ejemplo aunque vale la pena señalar que se realizó una división en todos los casos del numerador entre el divisor para poder comparar los resultados con las demás aritméticas, por lo cual no se explota todo el potencial de esta.

En cuanto a los tiempos de ejecución la aritmética más eficiente es **DecimalNum**, **FractionNum** en este sentido se vio incapacitada de realizar algunos cálculos, mientras que nuestra aritmética si bien fue la menos eficiente en la mayoría de los casos, se obtuvieron tiempos dentro de lo razonable para la ejecución de un algoritmo.

En la última sección analizada podemos observar la eficiencia de nuestra aritmética ante escenarios complejos desde el punto de vista computacional.

## 5. Conclusiones

En nuestro análisis del desempeño de la aritmética propuesta en este trabajo se vio reflejada la exactitud que esta posee a la hora de trabajar con datos de números en base 10. No obstante siempre tuvimos la desventaja de que la aritmética de punto flotante analizada fuera más rápida y eficiente y en par de ejemplos superando la exactitud de la nuestra. Por otro lado, observamos el comportamiento de la aritmética de fracciones racionales, si bien incapacitada de todo su potencial a la hora de compararse con las anteriores por los problemas antes explicados, siendo más precisa que las 2 anteriores en algunos casos, aunque incapacitada para realizar algunas tareas también especificadas anteriormente.

Cabe recalcar que la mayoría de estas comparaciones realizadas anteriormente en cuanto a la exactitud también se debe en gran medida a los métodos numéricos en los cuales se basa su cálculo, por lo que otro método numérico, fuera de los aquí abordados, pudiera ser más efectivo en uno u otro caso.

Por tanto, por todo lo anteriormente descrito podemos concluir que aunque la aritmética propuesta resuelve algunos de los problemas de precisión para los números en base 10, las aritméticas binarias de punto flotante actualmente son el método más efectivo para la realización de estas tareas computacionales.

## 6. Recomendaciones

Las comparaciones realizadas se basan en cálculos de números fuera de un contexto real, por tanto podemos sugerir una futura evaluación del desempeño de la aritmética propuesta en un marco práctico de la vida real.

Además también podemos mencionar que en la revisión bibliográfica encontramos propuestas y desarrollo de nuevos métodos para representar los números (que están siendo usados en el marco de la **Inteligencia Artificial** y el **Maching Learning**) lo cual serviría de objetivo para una futura investigación.

## Referencias

[1] Algoritmo para calcular la raíz n. URL: https://es.frwiki.wiki/wiki/Algorithme_de_calcul_de_la_racine_n-i%C3%A8me. Consultado en 19 de noviembre de 2022.

[2] Floating Point Arithmetic: Issues and Limitations URL: https://docs.python.org/3.9/tutorial/floatingpoint.html. Consultado en 19 de noviembre de 2022.

[3] Donald E. Knuth.(1997) The Art of Computer Programming. Chapter 4 – Arithmetic, (1997).

[4] Taylor Series. Wikipedia. URL: https://en.wikipedia.org/wiki/Taylor_series. Consultado en 19 de noviembre de 2022.