https://creativecommons.org/licenses/by/4.0/

CÁLCULO DE APROXIMACIÓN A LA RAÍZ CUADRADA CON EL MÉTODO BABILÓNICO USANDO TRES PARADIGMAS DE PROGRAMACIÓN

Calculation to reach the square root with the babylonic method using three programming paradigms

OMAR IVÁN TREJOS BURITICÁ[1], LUIS EDUARDO MUÑOZ GUERRERO[2]

Recibido: 22 de julio de 2022. Aceptado: 29 de julio de 2022

DOI: http://dx.doi.org/10.21017/rimci.2022.v9.n18.a117

Resumen

En el presente artículo se plantea una solución algorítmica al problema de aproximarse a la raíz cuadrada de un número dado, basándose en el método que utilizaron los babilonios y aprovechando los recursos computacionales modernos. Para implementar esta solución algorítmica, se ha acudido a la programación funcional a través del lenguaje Scheme entorno DrRacket versión 6.1, a la programación imperativa en lenguaje C++ entorno DevC++ versión 4.9.9.2 y a la programación orientada a objetos en lenguaje Java entorno Eclipse Java EE IDE for Web Developers Version: Luna Service Release 2 (4.4.2). La metodología utilizada se enmarca en la investigación científica de orden cuantitativo y tecnológico de carácter instrumental. El algoritmo presentado y la metodología utilizada se han compartido con los estudiantes de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira. Los resultados obtenidos han sido significativamente favorables en lo cuantitativo toda vez que se ha podido establecer un enlace entre lo teórico y lo práctico así como en lo tecnológico pues ha permitido promover entre los estudiantes de programación la idea de que cuando la lógica está clara, el paradigma y el lenguaje de programación automáticamente se aclaran. Con esta experiencia investigativa queda claro que es posible implementar soluciones antiguas a problemas tradicionales de las matemáticas sin perder el norte de su eficiencia y efectividad y así como encontrar diferencias y coincidencias entre diferentes paradigmas de programación.

Palabras clave: Algoritmo; programación funcional; programación imperativa; programación OO; raíz cuadrada.

Abstract

In this article an algorithmic solution to approximate the square root of a given number, based on the method used by the Babylonians, and using modern computational resources it arises. To implement this algorithmic solution, it has come to functional programming through the Scheme language environment DrRacket version 6.1, to imperative programming in C ++ environment DevC++ version 4.9.9.2 and object-oriented programming in Eclipse Java EE IDE Java environment for Web Developers Version: Moon Service Release 2 (4.4.2). The methodology is part of the quantitative scientific research and technological instrumental order. The algorithm presented and the methodology have been shared with students of Systems and Computer Engineering of the Technological University of Pereira. The results have been significantly favorable quantitatively since it has been able to establish a link between theory and practice as well as in technology as it has helped promote programming among students the idea that when the logic is clear, the paradigm programming language and automatically become clear. With this research experience it is possible to implement solutions to old traditional math problems without losing its northern and efficiency and effectiveness and to find differences and similarities between different programming paradigms.

Keywords: Algorithm; functional programming; imperative programming; object-oriented programming; square root.

1 Ingeniero de Sistemas, PhD en Ciencias de la Educación. Docente de planta, Universidad Tecnológica de Pereira. ORCID: https: //orcid.org/0000-0002-3751-6014 Correo electrónico: omartrejos@utp.edu.co

2 Ingeniero de Sistemas, PhD en Ciencias de la Educación. Docente de planta, Universidad Tecnológica de Pereira. ORCID: https: //orcid.org/0000-0002-9414-6187 Correo electrónico: lemunozg@utp.edu.co

I. Introducción

La resolución de problemas matemáticos en un proceso de formación de ingeniería de sistemas se usa como una estrategia de aprendizaje y enseñanza para darle sentido a los conocimientos adquiridos que se derivan de los contenidos curriculares. Desde este enfoque, el profesor puede combinar los conocimientos que se adquieren en otras asignaturas con los conocimientos propios de la programación de computadores a través de la implementación de soluciones que resuelvan problemas conocidos de las matemáticas. La idea que propone este artículo consiste en plantear un algoritmo que resuelva el problema de la aproximación a la raíz cuadrada de un número dado a partir del planteamiento de solución que propusieron los babilonios para que se afiance tanto el concepto matemático como la aplicación conceptual. En este sentido aprendizaje y didáctica pueden lograr, de una manera más sencilla, objetivos que se pueden lograr por caminos más complejos.

Con frecuencia encontramos que muchos de los conocimientos adquiridos en el aula pierden sentido y significado en poco tiempo, entre otras razones, por su falta de aplicación y de relación con conocimientos adquiridos en otras áreas. Todo conocimiento resulta ser significativo si se relaciona con conocimientos anteriores y ese es uno de los propósitos de este artículo, establecer una relación entre matemáticas y programación de computadores a través del sistema planteado por los babilonios para aproximarse a la raíz cuadrada de un número dado. Estas necesidades de aprendizaje se presentan con más frecuencia en las asignaturas teórico-prácticas como programación de computadores. La experiencia docente ha ido demostrando que las estrategias de enseñanza en áreas como las matemáticas enfatizan mucho más en el procedimiento que en la aplicación y por ello muchos conocimientos quedan ubicados en la memoria a corto plazo.

Estas estrategias traen dificultades al momento de acudir al pensamiento lógico como base para la resolución de problemas de las matemáticas y su respectiva implementación en lenguajes de programación a la luz de un determinado paradigma. Todo esto, en conjunto, dificultad la apropiación de conceptos y la toma de conciencia de los procesos, al menos en la medida esperada[1]. La problemática planteada pareciera evidenciar una concepción en la cual el estudiante, al momento de enfrentarse a un problema, confirma que el conocimiento ha sido aprendido de manera memorística a partir de sus reglas y conocimiento y no de manera lógica a partir de su aplicación.

Desde este enfoque, existe la posibilidad de que la relación entre matemática y programación de computadores posibilite un aprendizaje autónomo en ambas áreas apoyándose en las características cognitivas de cada estudiante, así como la responsabilidad frente al avance de su propio proceso de aprendizaje de forma que él mismo sea más activo y mucho más analítico. El concepto de interdisciplinariedad aparece, per se, dado que pocas veces, en los procesos curriculares de formación universitaria, se establecen nexos tan claros y directos entre dos áreas como los que se demuestran en este artículo entre las matemáticas y la programación de computadores. A partir de estos nexos entre áreas, se abren posibilidades para que los mismos estudiantes construyan conocimientos y significados con sentido de manera que lo aprendido pueda ubicarse, sobre una base lógica, en la memoria a largo o mediano plazo.

La pregunta de investigación que inspira este artículo gira alrededor del siguiente interrogante: ¿es posible encontrar soluciones algorítmicas, desde la programación de computadores, a problemas tradicionales de la matemática que posibiliten la interdisciplinariedad, la aplicación y el desarrollo lógico del pensamiento así como el fortalecimiento de los nexos entre esas dos áreas? La respuesta se resuelve a partir de su propia hipótesis: siempre es posible encontrar caminos que faciliten la relación interdisciplinaria entre diferentes áreas de manera que se fortalezca, para el estudiante, la autonomía, el significado con sentido y el conocimiento adquirido.

La presente investigación se ha realizado dentro del marco de trabajo que proveen las ciencias de la educación y se basa en las teorías del aprendizaje significativo formuladas por el Dr. David Paul Ausubel, teoría del aprendizaje por descubrimiento (Jerome Seymour Bruner y teoría pedagógica socio constructivista formulada por Lev Vygotsky, con una aplicación directa en el campo del aprendizaje de las matemáticas y su relación con la programación de computadores dentro del proceso de formación en Ingeniería de Sistemas y Computación.

A partir de estos enfoques se pretende que el estudiante sea capaz de conferirle sentido y significado, por sus propios medios, al conocimiento que adquiere y que sea capaz, asimismo, de encontrar caminos de aplicación desde la óptica de la tecnología computacional moderna y específicamente desde la programación de computadores.

Debe tenerse en cuenta que el sentido y el significado del conocimiento se establece a partir de las posibles relaciones que el estudiante pueda concebir entre los conocimientos previos y los nuevos conocimientos desde una óptica vinculante y crítica sobre la base de las posibles aplicaciones que encuentre en su propia formación académica y a la luz de sus capacidades y posibilidades de aprendizaje[2]. El presente artículo es uno de los productos resultantes del proyecto de investigación "Desarrollo de un modelo metodológico para la asignatura Programación I en Ingeniería de Sistemas basado en aprendizaje significativo, aprendizaje por descubrimiento y el modelo 4Q de preferencias de pensamiento" código 6-15-10 de la Vicerrectoría de Investigaciones, Innovación y Extensión de la Universidad Tecnológica de Pereira.

II. Marco Teórico

La relación entre matemáticas y programación de computadores posibilita el fortalecimiento del pensamiento lógico, el pensamiento algorítmico y la metodología para la resolución de problemas. El pensamiento lógico es el fundamento sobre el cual se han estructurado los cinco tipos de pensamiento matemático. Si bien la matemática por sí sola debiera promover este tipo de pensamientos, la práctica docente universitaria y la experiencia de los estudiantes ha ido demostrando que las estrategias que se utilizan en el desarrollo de las asignaturas pertinentes al área de las matemáticas enfatizan mucho más en los procedimientos que en la relación lógica del saber que subyace a dicha área. Por su parte el pensamiento lógico, que debiera ser el gran fundamento de las áreas de programación, no siempre lo es dado que en las asignaturas relativas a esta área se enfatiza más en lo instrumental y tecnológico que en lo lógico y por tanto se encuentran, al final de los cursos, estudiantes con un gran conocimiento de los lenguajes de programación pero con profundas dificultades para resolver de manera lógica un problema[3].

La práctica que se cristaliza a través del nexo entre matemáticas y computación son una muy interesante alternativa para que se promueva el pensamiento lógico toda vez que aquellas proveen problemas que pueden ser abordados de esta óptica y ésta posibilita la implementación de soluciones que, a partir de una lógica binaria computacional, permite resolver esas situaciones problémicas que la matemática brinda. Una de las dificultades que se presentan a nivel universitario es que poco diálogo académico existe entre los profesores de las áreas de matemáticas y de programación pero que, de darse esto, el gran beneficiado será el pensamiento lógico del estudiante por las razones ya planteadas[4].

En referencia al pensamiento algorítmico bien puede uno remontarse a los planteamientos realizados por Al Juarizmi hacia el año 700 d. C. en los cuales presentaba algunas soluciones a problemas matemáticos desde la óptica de los procedimientos generales lo cual abrió la puerta al álgebra y a la algoritmia, que precisamente lleva este nombre en honor al matemático árabe[5]. El pensamiento algorítmico implica un cierto orden en la concepción de una solución y, al menos, una metodología clara que posibilite su formulación y posterior implementación. El pensamiento algorítmico es una derivación del pensamiento lógico y, dado que es la base de la programación de computadores, puede inferirse que la programación es una derivación práctica de las matemáticas. Los procesos de formación universitaria en programas de ingeniería han llevado al área de programación de computadores a profundizar y fortalecer más el uso de las herramientas computacionales y de las tecnologías que a apropiar y asimilar la teoría matemática que subyace a ellas y que, de suceder así, permitiría que el estudiante capitalizara mucho más las herramientas computacionales que la tecnología moderna pone a su disposición.

Las metodologías para la resolución de problemas son un punto de unión entre el pensamiento lógico que se ha de promover desde las matemáticas y el pensamiento algorítmico que ha de fortalecerse desde la programación de computadores. Si bien podrían existir varias metodologías para resolver problemas, es claro que la algoritmia ha llevado a pensar que es posible decantar y sistematizar algunas de ellas a partir de unos conceptos claros y específicos. Kepner-Tregoe y Pólya son

dos autores que han propuesto metodologías efectivas para la resolución de problemas de forma que se puedan lograr objetivos por caminos cada vez más eficientes. De ese punto de encuentro entre las metodologías para resolver problemas y la programación de computadores han surgido los paradigmas que no son más que formas de adoptar una determinada visión para solucionar el problema.

Tres paradigmas han irrumpido en el mundo de la programación y se han convertido en el referente para la implementación de los respectivos lenguajes que los instrumentan. Estos paradigmas son el paradigma funcional, el paradigma imperativo y el paradigma orientado a objetos. En el paradigma funcional se destaca el concepto de función como elemento central y núcleo de trabajo, la estrategia "divide y vencerás" para simplificar una solución al punto de dividirla en pequeños problemas de forma que resolverlos, independiente, permitirá en conjunto encontrar la solución buscada y la recursividad que permite que una función se llame a sí misma como mecanismo para implementar procesos iterativos.

Por su parte la programación imperativa privilegia el concepto de estados y con ello plantea que un programa no es más que un conjunto de instrucciones que modifican los datos almacenados en diferentes variables en la memoria del computador y que, al final, uno o varios resultados almacenados resuelven el problema original. De acuerdo a esto, este paradigma se basa en el concepto de variable y, con este concepto, resalta la idea de los tipos de datos que son esquemas de almacenamiento que permiten guardar datos con diferentes características. El concepto de función, aplicado al paradigma imperativo, permite una concepción más simple, sencilla y manejable de los programas solución de manera que su mantenimiento sea mucho más eficiente en términos de tiempo.

En la programación orientada a objetos se privilegia el concepto de clases que son estructuras que permiten vincular atributos (que en la programación imperativa se llaman variables) y métodos (que equivalen a las funciones de la programación funcional y que también se pueden implementar en la programación imperativa). Esta unión entre atributos y métodos posibilita una interpretación del mundo mucho más aproximado a la realidad y, a partir del concepto de clases, permite una interacción entre ellas que considera las diferentes variantes que esa realidad provee.

III. Metodología

La raíz cuadrada de un número dado se define como el número que, al multiplicarlo por sí mismo, da como resultado el número original[6]. También puede definirse como el único número positivo que al multiplicarlo por él mismo origina el valor del primer número. La función raíz se representa con el símbolo que corresponde al signo de la radicación. La ubicación de un número n sobre el símbolo radical, indica que se quiere denotar la raíz n-ésima del número que se encuentre debajo de él. En tratándose de números negativos se puede hablar de dos raíces cuadradas imaginarias con fundamento en $\sqrt{-1}$.

Las raíces cuadradas son expresiones matemáticas que surgieron como consecuencia de los problemas geométricos como la longitud de la diagonal de un cuadrado determinado. De acuerdo con el papiro de Ahmes (1650 a. C.) ya los egipcios tenían conocimiento de este concepto y lo utilizaban en la medición de la tierra[7]. En la India antigua también se tuvo claro el concepto y fue tan antiguo como los Sulba Sutras, poemas sagrados con contexto matemático. El Baudhayana Sulba Sutra da fe de la aproximación a las raíces cuadradas y cúbicas[8].

Ariabhatta propuso una forma para hallar la raíz cuadrada incluyendo varios dígitos decimales[9]. Para los babilonios las raíces cuadradas se hallaban haciendo cálculos cíclicos aritméticos basados en la semisuma de los lados con lo cual se dejó la semilla para el concepto de convergencia y de método numérico[7]. La raíz cuadrada puede contarse como una de las primeras formas de desarrollo matemático frente a la cual los Pitagóricos tuvieron gran relevancia dado que ellos no creían en el concepto de números irracionales o inconmensurables (como era la raíz cuadrada de 2) puesto que no era expresable como cociente entre dos enteros[10] y que los llevó a enfrentarse a diferentes matemáticos de la época que no solo usaron sino que demostraron su existencia. Para el buen uso de los números reales negativos se acudió al con-

cepto de número imaginario y con ello se abrió el camino para los números complejos que permitió fortalecer el teorema fundamental del algebra.

El concepto de raíz cuadrada poco a poco fue ganando espacio en relación con los polinomios y también en lo que se refiere a la resolución de problemas de grados superiores a 2. El símbolo que representa la raíz cuadrada fue propuesto por Christoph Rudolff en 1525 y representa una r extendida de manera estética. Fue el matemático Leonhard Euler quien planteó como una forma de representar el factor componente de la raíz cuadrada de los números negativos[10].

La programación funcional se distingue porque prioriza el valor que se obtiene como resultado de invocar una función por encima del concepto de variables que no existen en este paradigma como se conciben en otros paradigmas[11]. En la programación funcional, como su nombre lo indica, el concepto de función es el alma del paradigma y se define como un pequeño código que logra, a su vez, un pequeño objetivo y que en conjunto con otras funciones, pueden satisfacer objetivos de gran calado en tratándose de programación de computadores. Una de las herramientas más poderosas que se usan en la programación funcional, y de la cual se hace gala en la solución que se plantea en este artículo, es la recursividad que se define como la propiedad que tiene una función de llamarse a sí misma lo cual establece una serie de características completamente diferentes a las que se describen cuando se invoca una función de manera formal[12]. El lenguaje Scheme utilizado en la implementación de la solución que se presenta en este artículo es un lenguaje híbrido que, aunque permite algunos pocos recursos de la programación imperativa, posibilita la implementación de soluciones funcionales a partir de los recursos y técnicas que el propio paradigma provee.

Por su parte la programación imperativa privilegia el concepto de máquina de estados para el desarrollo de un programa según la cual, un código no es más que un conjunto de instrucciones que se ocupan de modificar permanentemente los datos almacenados en las variables y de las cuales, al final, sale un valor o un conjunto de valores que satisfacen determinado problema[13]. La programación imperativa basa su modelo en la utilización, combinación y recombinación de tres estructuras que, teóricamente, permiten resolver cualquier problema computable. Estas tres estructuras son las secuencias, los condicionales y los ciclos y constituyen la base de la programación en sus primeros albores.

La programación imperativa es la que mejor se ha refinado con el tiempo y que ha permitido que otros modelos de programación inicien con un fundamento bastante sólido. A nivel de código, el modelo que propone la programación imperativa, puede mejorarse notoriamente incorporando el concepto de función que posibilita dividir el problema en pequeños problemas para poder conquistar el objetivo general del problema original[14]. El lenguaje C++ en su arista imperativas es un buen ejemplo de un lenguaje que, si bien soporta los elementos que se involucran en la programación orientada a objetos, posibilita todos los recursos que la programación imperativa sugiere[15].

La programación orientada a objetos es la base de un paradigma que intenta interpretar el mundo tal y cual como lo concebimos. En este paradigma el núcleo de trabajo es el concepto de objeto que se hace realidad a través de lo que se conoce como una clase que no es más que la conjunción entre atributos y métodos[16]. Los atributos corresponden a lo que en programación imperativa llamamos variables y los métodos corresponden a lo que en programación funcional llamamos funciones (con opción de retorno especificado por el programador)[17].

De esta forma podemos concebir la programación orientada a objetos como la mejor unión entre lo imperativo y lo funcional incorporando un concepto original como es el de objeto tal como se ha explicado en este párrafo. Este concepto origina algunos nuevos conceptos tales como herencia, polimorfismo o encapsulamiento que son propios del paradigma y que han dinamizado el mundo de la programación al punto de permitir un avance en la interacción de las nuevas tecnologías con el mundo moderno[17].

En este artículo se implementa en tres paradigmas de programación a través de lenguajes respectivos, el método babilónico para hallar la raíz cuadrada de un número dado. El procedimiento que los babilonios plantearon para hacerlo consiste

en multiplicar sucesivamente, y bajo un esquema establecido, dos números de forma que prontamente se encuentre la solución[18]. En términos generales se parte de dos números que inicialmente den como resultado el valor a buscar y progresivamente se va dividiendo el número al cual se le quiere hallar la raíz cuadrada entre la semisuma de los dos números dados. Iterar este procedimiento permite el hallazgo sencillo y simple de un valor que se aproxima a la raíz cuadrada y cuya aproximación dependerá del nivel de precisión que se le quiera conferir. Desde una óptica algorítmica[19], el procedimiento babilónico podría resumirse de la siguiente forma:

Algoritmo para aproximarse a la raíz cuadrada de un número basado en el método babilónico

i. *Definir valor n al cual se le quiere encontrar su raíz cuadrada*
ii. *Definir dos valores a y b cualesquiera cuyo producto sea igual a n*
iii. *Almacenar en la variable a, el valor de dividir n entre la semisuma entre a y b*
iv. *Hallar el nuevo valor para b*
v. *Verificar nivel de precisión con el valor hallado restando n – (a*b)*
vi. *En caso de que la precisión no se haya alcanzado, entonces volver al paso iii*

Realizando pruebas en los diferentes paradigmas se ha encontrado que, aunque el método babilonio especifica que el producto de los dos primeros números debe ser igual al número buscado, no es absolutamente necesario gracias al concepto de convergencia que proveen los métodos numéricos pues con el simple hecho de que dicho producto origine un valor menor que el número buscado, con eso es suficiente para que se pueda realizar una buena aproximación a la raíz cuadrada de un número dado. El concepto de convergencia implica que un método iterativo redundante siempre lleva a un único valor no disperso, lo cual se altamente conveniente en tratándose de una aproximación a la raíz cuadrada[20]. A partir del algoritmo presentado se han realizado tres implementaciones que se presentan a continuación, cada una de ellas codificada en un lenguaje de programación diferente. La primera implementación se realizó en lenguaje de programación Scheme tomando como base el paradigma de programación funcional[21].

```
; Función Cálculo Raíz Cuadrada - Método Babilónico

; Función que implementa el método babilonio para aproximarse a la raíz cuadrada
; babilonio: nombre de la función
; lado1, lado2: argumentos con los cuales se halla la raíz cuadrada
; precisión: nivel de exactitud de la aproximación
(define (babilonio num lado1 lado2 precision)    ; Definición de la función
; Si la diferencia entre los lados es menor que la precisión
      (if(< (- lado1 lado2) precision)
        lado1       ; retorne el valor de un lado
        (begin      ; Sino,
         ; Calcular de nuevo el lado1
          (set! lado1 (* (+ lado1 lado2) 0.5))
     ; Basado en el lado1, calcular el valor del lado2
          (set! lado2 (/ num lado1))
          ; Llamar recursivamente a la función
          (babilonio num lado1 lado2 precision))
        ) ; fin condicional
      )   ; fin función babilonio

; Función que recibe los datos del usuario
; ppal:    nombre de la función
; num:     número para calcularle la raíz cuadrada
; precisión: nivel de exactitud de la raíz cuadrada
; base:    argumento auxiliar
 (define (ppal num precision base)
 ; Llamado a la función
     (display "Número..>")    ; Título
     (set! num (read))            ; Leer valor de usuario
     (display "Precision-->")   ; Título
     (set! precision (read)) ; Leer nivel de exactitud
     (set! base (random num)); Guardar núm aleat
     (display "Raiz cuadrada-->")    ; Título
; Llamar función que muestra aprox de raíz cuad
     (set! base (babilonio num base(- base 1) precision))
      (display base)  ; Mostrar valor de retorno
      (newline)         ; Dejar línea en blanco
      (display "Prueba-->")    ; Título
      (display (* base base))
 ; Mostrar el resultado de elevar al cuadrado el
 ; valor recibido
      )    ; fin función ppal
    ; Llamado a la función ppal
(ppal 1 1 1)
```

Por su parte, la implementación realizada en lenguaje C++ aprovecha las características de la programación estructurada y las herramientas que el mismo lenguaje posibilita. El código corresponde al programa que se presenta a continuación.

```
/* Programa para aproximarse a la raíz cuadrada de un número usando el método babilonio */

// Inclusión de librerías
#include <iostream>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
using namespace std;

/* Función que se aproxima a la raíz cuadrada por el método babilónico */
double babilonio(double num, double lado1, double lado2, double precision)
{     // Inicio de la función
  int sw=0; // Variable local
  while(sw==0)
// Mientras no se encuentre la raíz cuadrada
  {      // Inicio del ciclo
    lado1 = (lado1 + lado2)*0.5;
// Almacenar la semisuma de los lados
     lado2 = num / lado1;
// Almacenar el valor del 2º lado
     if((lado1-lado2) < precision)
// Si se alcanzó la precisión esperada
       sw=1;
// cambiar valor de la variable respectiva
} // Fin ciclo
  return(lado1);
// Retornar valor de la raiz cuadrada
} // Fin función

// Función Principal
int main( )
{     // Inicio de la función principal
  double num, precision, aux;
// Declaración variables locales
  srand(time(NULL));
// Activación de números seudoaleatorios
  cout<<"Numero? ";
// Título
  cin>>num;
// Lee el número para calcular su raíz cuadrada
  cout<<"Precision? ";
// Título
  cin>>precision;
// Lee el nivel de precisión
  aux = rand()*num;
// Obtiene un valor aleatorio
  aux = babilonio(num, aux, aux-1, precision);
// Llamado a la función babilonio
  cout<<"Raiz Cuadrada-->"<<aux;
// Muestra el resultado
  cout<<"\nPrueba-->"<<aux*aux;
// Probar el resultado
  getche( );
// Espera para ver el resultado
} // Fin función principal
```

Finalmente se realizó una implementación simple en el lenguaje de programación Java desde una infante versión de la programación orientada a objetos con el ánimo de poder establecer algunos elementos que permitan comparar las tres implementaciones. El código correspondiente se presenta a continuación.

```
// Programa para aproximarse a la raíz
// cuadrada usando el método babilonio
import java.util.Scanner;

public class babilonio {
// Declaración Clase publica
public static void main(String[ ] args)//      Método principal
{                 // Inicio método principal
Scanner tc=new Scanner(System.in);
// Variable de I / O
   double num, precision, aux;    //          Variables locales
   System.out.print("Numero? ");    // Solicitud de dato
   num=tc.nextDouble();          // Lectura de número doble
   System.out.println("Precision? ");   // Solicitud de dato
   precision=tc.nextDouble();     //     Lectura     de número doble
   aux=Math.random()*num;      //          Almacena número seudoaleatorio
aux=babil(num, aux, aux-1, precision);  // Llamado a método babil
System.out.println("Raiz Cuadrada--> " + aux);   // Mostrar resultado
System.out.println("Prueba----> " + (aux*aux));   // Probar resultado
} // Fin método principal

// Método de apoyo babil
static double babil(double num, double lado1, double lado2, double precision)
{     // Inicio de método babil
  int sw=0; // Variable local
  while(sw==0)
// Mientras no se encuentra la raiz cuadrada
  {  // Inicio ciclo
    lado1 = (lado1 + lado2)*0.5;
// Calcular semisuma de los lados

     lado2 = num / lado1;
// Calcular valor del 2º lado
     if((lado1-lado2) < precision)
// Si se alcanzó el nivel de precisión
       sw=1;
// Cambiar valor del interruptor
     } // Fin del ciclo
  return(lado1);
// Retornar valor de la raíz cuadrada
} // Fin función babil
} // Fin Clase
```

IV. Resultados

En la tabla I se muestran los resultados obtenidos con los diferentes programas acorde con sus respectivas implementaciones y sobre la base de sus respectivos paradigmas.

Tabla I. Resultados obtenidos

Pdg	Leng	IDE	Ver	Resultado Obtenido
FUNCIONAL	SCHEME	DRRACKET	6.1	Caso: raíz cuadrada exacta Número..>25 Precision-->0.001 Raiz cuadrada-- >5.000000837813562 Prueba-- >25.000008378136318 Caso: raíz cuadrada real Número..>20 Precision-->0.001 Raiz cuadrada-- >4.4721402170657 Prueba-->20.00003812109645
IMPERATIVO	C++	DEVC++	4.9.2	Caso: raíz cuadrada exacta Numero? 25 Precision? 0.001 Raiz Cuadrada→ 5 Prueba → 25 Caso: raíz cuadrada real Numero? 20 Precision? 0.001 Raiz Cuadrada→ 4.47215 Prueba → 20.0001
ORIENTADO OBJ	JAVA	ECLIPSE LUNA	4.4.2	Caso: raíz cuadrada exacta Precision? 0,001 Raiz Cuadrada--> 5.000001275108964 Prueba -------> 25.00001275109127 Caso: raíz cuadrada real Precision? 0,001 Raiz Cuadrada--> 4.472136137421236 Prueba -------> 20.000001631628933

Fuente: el autor.

Como puede observarse, los resultados obtenidos en los tres lenguajes a partir de la implementación del método babilónico para hallar la raíz cuadrada demuestran tanto su efectividad como su simplicidad puesto que en cada paradigma se intentó interpretar el algoritmo inicial que se planteó sobre la base del procedimiento diseñado por los babilonios. Es notorio el nivel de precisión que se puede lograr con lenguajes de programación como Scheme o Java dado el modelo de manejo de memoria que provee su tecnología.

V. Discusión y análisis de resultados

Como puede verse, la implementación de un algoritmo tan antiguo como el de los babilonios para hallar la raíz cuadrada podría permitir encontrar caminos de gran eficiencia lógica y bajo consumo de máquina para la resolución del problema en mención. Si bien es cierto que el algoritmo es el mismo, se puede observar que cada una de las diferentes implementaciones acude a las características que el respectivo paradigma y su instanciación en un lenguaje, permiten. El algoritmo es bastante simple pero muy efectivo pues si se tiene en cuenta que no se necesario que el producto de los dos primeros número dé como resultado el valor al cual se le quiere encontrar su raíz cuadrada. Es suficiente con que dicho producto sea menor que el número dado.

Desde la óptica que provee el concepto de convergencia en los métodos numéricos se puede destacar el hecho de que cuando se construye un algoritmo paramétrico que se basa en un proceso convergente, el límite de dicho proceso es un número al cual se llega producto de la iteración de las operaciones[22]. Es claro que en tiempos de los babilonios este concepto de convergencia no era consciente pero un algoritmo como el que plantearon dejó las bases muy sólidas para que, entre otros algoritmos, se pudiera llegar posteriormente a encontrar métodos numéricos que resuelven problemas a partir de interacciones cíclicas algebraicas[23][24].

A nivel de código se puede notar que en el lenguaje de programación Scheme el fundamento para la implementación del proceso cíclico lo establece la recursividad, característica fuerte del paradigma funcional, con lo cual se intenta aprovechar en

máximo nivel las potencialidades de las tecnologías computacionales modernas a partir del uso de una variable que actúa como un interruptor. En la versión escrita de este mismo algoritmo en lenguaje C++ es de anotar que solo se aprovecha su arista imperativa, dado que ese era el objetivo del ejercicio, y se evidencia que la inclusión del concepto de función facilita no sólo la comprensión del programa (respaldado por su respectiva documentación) sino que simplifica tanto las pruebas como el ajuste y el eventual mantenimiento del programa[25] [26].

Por su parte, la implementación a partir del lenguaje de programación Java aprovecha el concepto de construcción de una clase y, dentro de ella, la implementación de una función principal así como de una función estática que es la que sirve de apoyo para la búsqueda del valor que se aproxima a la raíz cuadrada[27]. El factor de precisión se ha definido como un parámetro para que sea el mismo usuario el que establezca hasta donde quiere que llegue la aproximación. Debe advertirse que en Scheme y en Java dicho factor puede llegar a ser mucho más preciso que en lenguaje C++ en su arista imperativa, debido a que el C++ desde la perspectiva imperativa, está limitado por el concepto de tipo de variable para poder almacenar determinados datos[28][29].

VI. Conclusiones

Es posible encontrar en los métodos de solución antiguos, procedimientos óptimos que aprovechan conceptos modernos de computación, sin que se lo hubieran propuesto, para lograr resolver determinados problemas como la búsqueda y aproximación al valor de la raíz cuadrada de un número dado.

La comparación entre diferentes paradigmas posibilita un análisis mucho más exhaustivo de los algoritmos que se quieren implementar y, a partir de allí, permiten reconocer características que posibilitan la mejor escogencia de un lenguaje de programación al momento de implementarlo.

Es muy importante que cuando se implemente un algoritmo, se acuda al aprovechamiento de las características del paradigma que subyace al lenguaje de programación que se haya utilizado.

Si bien es cierto que la tecnología posibilita cada vez más opciones de implementación, también lo es que intentar implementar la solución de un problema sin entenderlo claramente podría implicar uso excesivo e ineficiente de tiempo y de recursos.

La historia de un algoritmo, su nacimiento, su evolución y su refinación terminan convirtiéndose en herramientas que, fácilmente, podrían permitir que encontremos soluciones óptimas de fácil implementación.

La relación entre programación y matemáticas permite conectar posibles soluciones implementables con problemas planteados que, desde la computación, pueden ser resueltos fácilmente.

Referencias

[1] H. Paz Penagos, «Aprendizaje autónomo y estilo cognitivo: diseño didáctico, metodología y evaluación,» Revista Educación en Ingeniería, vol. 9, nº 17, pp. 53-65, 2014.

[2] D. P. Ausubel, Sicología Educativa: Un punto de vista cognoscitivo, Ciudad de México: Trillas, 1986.

[3] L. E. Muñoz Guerrero, J. I. Rios Patiño y O. I. Trejos Buriticá, Programación Funcional con Scheme, Pereira: Calameo, 2014.

[4] P. Van Roy, Concepts, Techniques and Models of Computer Programming, Estocolmo: Université catholique de Louvain, 2008.

[5] J. Rey Pastor y J. Babini, Historia de la Matemática, Barcelona (España): Editorial Gedisa, 1985.

[6] I. Stewart, Historia de las Matemáticas en los últimos 10000 años, Madrid (España): Crítica, 2011.

[7] M. Kline, El pensamiento matemático de la antiguedad a nuestros días, Madrid: Alianza Editorial, 2012.

[8] C. Boyer, Historia de la Matemática, Madrid (España): Alianza Editorial, 2010.

[9] J. L. Chabert, A history of algorithms, Berlin (Germany): Springer, 2005.

[10] A. Rooney, Historia de las Matemáticas, Barcelona (España): La Biblioteca del Saber, 2009.

[11] P. Van Roy, Techniques and methods in programming computer, Louvaine: University Press, 2008.

[12] M. e. a. Felleisen, How to design Programs, Boston: MIT Press, 2006.

[13] O. I. Trejos Buriticá, Fundamentos de Programación, Pereira: Papiro, 2006.

[14] O. I. Trejos Buriticá, La Esencia de la Lógica de Programación, Pereira: Papiro, 2000.

[15] Deitel & Deitel, C++ Programming, New York: Prentice Hall, 2013.

[16] H. Schildt, The complete reference Java, 7th Ed. ed., New York, USA: McGraw Hill, 2009.

[17] H. Schildt, C++ Programming, Vancouver: McGraw Hill, 2010.

[18] J. E. Aguilar Morales, La Evaluación Educativa, Oaxaca: Asociación Oaxaqueña de Sicología, A. C., 2011.

[19] A. Attard, E. Di Ioio y K. Geven, Student Centered Learning. An insight into theory and practice, Bucarest: Lifelong learning programme - European Community, 2010.

[20] C. Bonwell y J. Eison, «Active Learning: Creating Excitement in the Classroom,» ASHEERIC Higher Education Report No 1, nº 1, 1991.

[21] J. S. Bruner, Actos de significado, Madrid: Alianza Editorial, 1991.

[22] J. S. Bruner, El proceso de la Educación, Ciudad de México: Editorial Hispanoamericana, 1963.

[23] F. Diaz Barriga, Estrategias docentes para un aprendizaje significativo, México: McGraw Hill, 2005.

[24] G. Small, El cerebro digital, Madird: Editorial Urano, 2011.

[25] O. I. Trejos Buriticá, Aprendizaje en Ingeniería: un problema de comunicación, Pereira (Colombia): Tesis Doctoral - Universidad Tecnológica de Pereira, 2012.

[26] O. I. Trejos Buriticá, Algoritmos Problemas Básicos, Pereira (Colombia): Papiro, 2008.

[27] S. McConell, Desarrollo de Proyectos Informáticos, New York, USA: McGraw Hill Companies, 2005.

[28] J. Medina, Los 12 principios del cerebro, Bogotá, Colombia: Grupo Editorial Norma, 2012.

[29] S. Aamodt y S. Wang, Entra en tu cerebro, Barcelona (España): Ediciones B, 2008.