ISSN 1870-4069

# Motor matemático OpenSource con inyección de dependencias dinámicas en Java

Edgar Hamlet Solano-Díaz, Francisco López-Orozco,
Rogelio Florencia-Juárez, Jesús Israel Hernández-Hernández

Instituto de Ingeniería y Tecnología,
México

Universidad Autónoma de Ciudad Juárez,
México

al154007@alumnos.uacj.mx,
{francisco.orozco,rogelio.florencia,israel.hernandez}@uacj.mx

**Resumen.** Este artículo presenta la implementación de una herramienta *OpenSource* tipo calculadora científica modular con el objetivo de facilitar el desarrollo de operaciones y funciones matemáticas en una misma aplicación con módulos altamente modificables, completamente personalizables y fáciles de compartir. El programa tiene su centro de ejecución en un motor matemático de operaciones dinámicas donde el usuario evita la necesidad de programar en lenguajes matemáticos especializados o desarrollar toda la lógica matemática de cero en un lenguaje conocido para el uso de una única función. El programa trae consigo todo lo necesario para la generación auto-descriptiva en lenguaje humano de cada operación, el proceso matemático correcto de realizar la jerarquía de operaciones y una ventana de visualización útil para interpretar resultados.

**Palabras clave:** Motor matemático, inyección de dependencias, programación modular, explicaciones matemáticas auto-generadas.

## Open-Source Mathematical Engine with Dynamic Dependency Injection in Java

**Abstract.** This article presents the implementation of an open-source, modular scientific calculator tool designed to facilitate the development of mathematical operations and functions within a single application. The tool is built with highly modifiable, fully customizable, and easily shareable modules. Its core execution relies on a mathematical engine of dynamic operations, allowing users to avoid programming in specialized mathematical languages or developing the entire mathematical logic from scratch in a known programming language just to use a single function. The program includes everything required for self-descriptive human-readable generation of each operation, ensures the correct

application of the order of operations, and provides a visualization window that helps users interpret results effectively.

**Keywords:** Mathematical engine, dependency injection, modular programming, auto-generated mathematical explanations.

## 1. Introducción

La computación aplicada a la resolución de matemática avanzada es un tema recurrente en la programación, este campo ha dado paso a herramientas altamente especializadas en la resolución de ecuaciones complejas y diseños abstractos. El desarrollo de estos programas ha convergido en la creación de diversos software auto-explicativos donde por cada operación el resultado es interpretado y explicado paso a paso la justificación de su operación o han surgido en la creación de nuevos lenguajes de programación altamente especializados en el desarrollo y visualización de abstracciones matemáticas especificas.

El proceso de uso y creación de este tipo de herramientas informáticas aplicadas a la matemática puede verse desde dos perspectivas.

El usuario final, estudiante o docente, que seguido ve en la problemática de las pocas herramientas didácticas disponibles en el mercado, con poca capacidad de personalización, recursos limitados para su uso o pago obligatorio para el completo acceso y poca facilidad para compartir o duplicar. El programador o investigador que desarrolló la herramienta final el cuál pasó por el proceso de capacitación en algún lenguaje matemático altamente especializado o el proceso de creación desde cero de todo un entorno matemáticamente correcto para el desarrollo de las funciones matemáticas a usar.

Estas posiciones se convierten en un ciclo auto-alimentado donde el elevado tiempo de desarrollo por la capacitación en materia de programación y matemática deja de lado la gran comunidad de programadores no especializados en temas de matemáticas o matemáticos no completamente familiarizados con la programación de alto nivel, esto haciendo las posibilidades de nuevas herramientas interactivas muy cerradas subiendo así los costos de uso y la continua sobre especialización del software.

## 2. Trabajos previos

Son conocidos programas de resolución de problemas paso a paso con explicaciones verbales[2] , programas de representación gráfica matemática[3], lenguajes de programación especializados de forma nativa[1] o algoritmos matemáticos de antemano optimizados para la resolución de ecuaciones jerárquicamente correctas [10].

### 2.1. Programas de resolución de problemas paso a paso:

Programa que en tiempo real describe por pasos y da justificación a sus operaciones con descripciones auto-generadas y gráficos explicativos.

Actualmente son populares ejemplos de aplicaciones web de este tipo que ofrecen la resolución paso a paso de forma parcial con detalles a cambio de una suscripción paga; igualmente no se encuentra disponible y popularizada un servicio o aplicación donde uno mismo como usuario pueda personalizar las funciones que usará con explicaciones paso a paso.

### 2.2. Programas de representación gráfica:

Programas de visualización con datos de entrada en diferentes unidades matemáticas; planos cartesianos, mapas 3D, tablas o diversos sistemas de coordenadas. En los ejemplos más populares de este tipo de programas se encuentra la complicación de no tener una forma de representar los datos personalizada o estandarizada ya que cada programa usa el formato que pueda de acuerdo a sus necesidades y no las del usuario.

### 2.3. Lenguajes de programación especializados en matemáticas:

Lenguajes de programación diseñados con el entorno matemático especializado. Las operaciones necesarias para su procesamiento, graficación e interpretación son funciones nativas del lenguaje. La problemática con este tipo de lenguajes radica en su nivel de especialización y toda la capacitación previa que debe tenerse elevando los costes de desarrollo final.

Finalmente se llega a la conclusión que no se ha encontrado un software popularizado con la capacidad de resoluciones matemáticas paso a paso con entradas variables y adaptables a los requerimientos del usuario, 100 % con salida de datos e interpretación matemática disponible y programada por la misma comunidad, teniendo una oportunidad de desarrollo y popularización entre comunidades de programadores y usuarios no programadores la idea desarrollada en este documento.

## 3. Metodología

El objetivo principal fue la creación de un motor matemático compatible con números enteros, fracciones, decimales, vectores, ángulos, entre otros; que permitiera fácilmente su implementación para realizar operaciones entre ellos. Además, que esto fuera posible de manera dinámica sin la necesidad de desarrollar múltiples motores para diferentes unidades u operaciones. En conjunto que se tuviera un apartado visual para representar cada tipo de unidad creada e inyectada. Todo complementado con un apartado para la explicación paso a paso de cada operación realizada de forma auto generada, con plantillas descritas por el programador del modulo a usar, y así hacer personalizada la experiencia de crear *"Cuadernos de Apuntes"* como un *"Todo en uno"*. Por ultimo el objetivo se buscaba la facilidad de incluir nuevos módulos útiles para la personalización de los apuntes del usuario como una pizarra virtual, una sección de notas y la opción de firmar cada documento creado con nombre y

título del apunte. Acorde a todas estas propuestas aunado a un desarrollo rápido siempre dependiente de las opiniones y accesibilidad del usuario la metodología implementada fue metodología ágil[4], donde en todo momento se consideró:

- Las personas y las interacciones antes que los procesos y las herramientas.
- El software en funcionamiento antes que la documentación exhaustiva.
- La colaboración con el cliente antes que la negociación contractual.
- La respuesta ante el cambio antes que el apego a un plan.

En todo el desarrollo puede verse estos principios fundamentales aplicados. Al momento de ser la librería expuesta en el presente documento consumida por sus propios desarrolladores y usuarios externos no directamente relacionados con el código fuente para crear sus propios módulos inyectables, cada parte fue sometida a máximas pruebas de funcionalidad y utilidad siendo demostrable esto en cada **commit**, **merge request** o modificación de código subido y registrado en la plataforma **github**, utilizada como herramienta para control de versiones y forma segura del cumplimiento de la metodología ágil planteada.

### 3.1. Planeación de lenguajes, técnicas y tecnologías

El lenguaje de desarrollo fue Java por su fácil exportación multiplataforma, amplio rango de usuarios[7], estandarización de enseñanza en la academia y bibliotecas de interfaces gráficas multiplataforma 100 % orientada a objetos. Para la interfaz gráfica se usó **Swing**[6] y **AWT**[5]; bibliotecas muy flexibles, útiles y reconocidas en el lenguaje permitiendo así que cualquier usuario con intensión de modificar o personalizar los módulos se encuentre con un entorno familiar, clásico y documentado. Para lograr un proyecto fácilmente modificable e intuitivo se usaron los patrones **SOLID**(**S**RP-**O**CP-**L**SP-**I**SP-**I**DP).

**SRP: S**ingle **R**esponsability **P**rinciple por la facilidad de desarrollo del proyecto y no especialización en ningún tema externo al modulo a programar.

**OCP:O**pen–**C**losed **P**rinciple A fin de que la lógica matemática compleja y la interfaz de interpretación de programación avanzada quedara oculto al usuario programador final.

**LSP:**(**L**iskov **S**ubstitution **P**rinciple) Para mantener coherente la herencia de las clases éstas son hiper especializadas siendo así más entendible al momento de heredar que atributos le proporcionaría.

**ISP:**(**I**nterface **S**egregation **P**rinciple) En este apartado se buscó fortalecer el mismo concepto de sub-modularizacion y super-especialización de cada objeto, evitando dejar métodos heredados sin sobrescribir o llamadas a métodos sin utilidad por herencias innecesarias.

**IDP**:(**D**ependency **I**nversion **P**rinciple) La principal metodología de diseño que dará todo su poder al motor matemático y su facilidad de interacción a nuevas unidades u operaciones reside en la inyección de dependencias que tiene como principio esta metodología técnica, la implementación de clases abstractas base las cuales serán las únicas variables que manejara directamente el motor creando un ambiente abstracto robusto y flexible[11].

# 4. Desarrollo

## 4.1. Base matemática del programa

El primer paso en el desarrollo del programa fue definir los objetos abstractos a usar como padres o plantillas genéricas para su posterior uso en el motor matemático. Las clases desarrolladas fueron:

- **Unidades Matemáticas:** Clase abstracta para definir un nuevo tipo de unidad en el programa, consiste en el dato que representa una forma cuantificable en las matemáticas y pueden por tanto desarrollar acciones como ser definido con sus variables específicas, ser visualizado en alguna forma gráfica, realizar operaciones matemáticas o ser resultado de alguna función matemática. Una unidad matemática está compuesta por:
  - Nombre del tipo de unidad (String del constructor).
  - Símbolo identificador (Char del constructor).
  - Nombre de la categoría que pertenece (String del constructor).
  - Un objeto que represente su valor y pueda ser retornado (Object primitivo del constructor).
  - Método que defina qué ventana gráfica necesita renderizar para ser creado y recibir los parámetros que usará. (método void abstracto heredado para crear un JFrame que represente la entrada de datos).
- **Operaciones Matemáticas:** Son el segundo tipo de dato en el programa, representan las operaciones a realizar entre dos o más unidades, son el núcleo dinámico donde al ser heredadas por el programador que deseé implementar sus propias operaciones le pedirá sobrescribir los métodos necesarios para su explicación, visualización y resolución. Una operación matemática está compuesta por:
  - Nombre del tipo de operación (String del constructor).
  - Símbolo identificador (Char del constructor).
  - Breve descripción de la operación (String del constructor).
  - Descripción detallada de la operación (String del constructor).
  - Prioridad según la jerarquía matemática (1:Sumas y Restas, 2: Multiplicaciones y Divisiones, 3:Potencias y raíces, 4:Paréntesis) (Int del constructor).
  - conLlave, booleano que representa si es una operación con paréntesis u operación-función. (Booleano del constructor) Ejemplo: ***vectorMagnitud(*****vecA+vecB+...VecN*****)**.
  - Método ***calcularOperacion()***, con N parámetros tipo UnidadMatematica de entrada y un retorno tipo **UnidadMatematica**. (Método abstracto heredado donde se implementara la lógica necesaria para resolver la operación).
  - Método definir ***TipoDeOperandoscorrectos()*** con salida tipo **String[]** que espera por retorno el "Nombre del tipo de unidad"que pueden usarse con este operando. (Método abstracto heredado donde se define si la entrada de datos pasada coincide con los datos esperados).

- **Funciones Matemáticas:** Intermedio entre la operaciones matemáticas y unidades matemáticas. Toma como entrada $N$ parámetros especificados por el usuario desarrollador. El tipo de los $N$ parámetros puede ser de cualquier objeto hijo de la clase unidad matemática. Su proceso de implementación consiste en sustituir métodos especificados por la plantilla padre y retornando otro objeto tipo unidad matemática el cual será en ultimo paso insertado en la operación general. Una función matemática está compuesta por:
  - Nombre del tipo de función (String del constructor).
  - Símbolo identificador (Char del constructor).
  - Descripción detallada de la función (String del constructor).
  - Método ***calcularOperacion()***, con $N$ parámetros tipo UnidadMatematica de entrada y un retorno tipo UnidadMatematica (Método abstracto con salida tipo UnidadMatematica donde se especifica la lógica de resolución de la función).
  - Método ***llamarFuncionMatematica()*** que defina qué ventana gráfica necesita renderizar para ser creado y recibir los parámetros que usará (método void abstracto heredado para crear un JFrame que represente la entrada de datos).

Operación-función no es igual a las funciones matemáticas, una función matemática solicita $N$ parámetros de $X$ tipo de datos, una operación con paréntesis solicita 1 sub-operación con resultado en un único tipo de dato: (funcion 1 no es igual a la operacion 2).

$$\boldsymbol{hipotenusa(CatetoA,CatetoB),} \tag{1}$$

$$\neq$$

$$(4\vec{a}x + 6\vec{a}y + 3\vec{a}z) - (2\vec{a}x + 1\vec{a}y + 6\vec{a}z). \tag{2}$$

Existen clases desarrolladas como **EstandarNparamsJDialog.java** creadas para ser utilizadas en los métodos que piden desarrollo de interfaces gráficas si el usuario no conoce las librerías **Swing** o **AWT**.

### 4.2. Base/motor matemático del programa

El concepto de motor matemático hace referencia a una clase encargada únicamente en la resolución de una operación general dividida en 4 fases de resolución matemática. El motor matemático debe contar con la capacidad de poder resolver operaciones según el orden de jerarquía, sin importar la longitud de la operación, sin importar los operandos ni los operadores y ser robusto al retornar errores y el causante de estos. Las técnicas de programación implementadas en este proceso principal del proyecto son algoritmos de recursividad al estilo **Divide and Conquer**, objetos tipo **ArrayList** para

aplicar los conceptos de **Stack** y la implementación de un flujo de resolución el cual consiste en 4 pasos de prioridad cada uno dependiente del anterior con cinco banderas de error las cuales representan un tipo de error diferente dependiendo el proceso que fue insatisfactorio.

Pasos del motor: Al instanciar la clase **OperacionGeneral** se solicita el valor **List<ObjetoMatematico>** el cual consiste en una lista de entrada que contendrá todos los objetos tipo **OperacionMatematica** y **UnidadMatematica** que conforman nuestra operación a resolver. Al crear un nuevo objeto tipo **OperacionMatematica** solicita por el constructor padre el tipo de orden jerárquico al que pertenece el cual será definitorio para el siguiente proceso.

- Orden de operación 4: En este primer orden se encontraran todos los objetos los cuales son operaciones con paréntesis, en esta categoría entran las operación-función encapsuladas en paréntesis (3+2) u operaciones que solicitan un parámetro de entrada y retornan una unidad matemática como el ejemplo 3:

$$\textbf{\textit{vecMagnitud(}}3\vec{a}x + 2\vec{a}y + 5\vec{a}z). \tag{3}$$

  Esto se guarda en el ArrayList **ObjetosMatematicosPrimerOrden** los resultados de las operaciones y pasando sin modificación cualquier unidad matemática u operación de orden jerárquico menor a 4. Una vez resuelto todo este primer proceso el resultado almacenado de esta nueva operación sin jerarquía 4 será pasado como entrada a la siguiente parte del flujo.
- Orden de operación menor a 4: En esta parte del proceso se hace uso del método ***resolverOrdenN()*** el cual es un método genérico para resolver todo tipo de operación menor a 4 o en otras palabras toda operación que no haga uso de paréntesis sino operandos de lado izquierdo y operandos de lado derecho. En esta parte del flujo la función es llamada 3 veces para la resolución de los ordenes de tipo 3 (potencias,raíces...), 2 (multiplicaciones,divisiones), 1 (sumas, restas, unidades sueltas finales).
  operación tipo 3:

$$5^2,$$

  operación tipo 2:

$$(4\vec{a}x + 6\vec{a}y + 3\vec{a}z) * 25,$$

  operación tipo 1:

$$(4\vec{a}x + 6\vec{a}y + 3\vec{a}z) - (2\vec{a}x + 1\vec{a}y + 6\vec{a}z),$$

  operación tipo Resultado:

$$(2\vec{a}x + 64\vec{a}y + 2\vec{a}z).$$

## 5. Interfaz gráfica

Una vez implementada la base lógica del programa fue necesario crear la interfaz gráfica. Ésta debía ser modular, personalizable y digerida para que

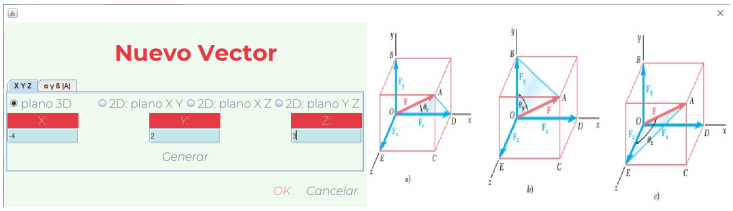



**Fig. 1.** Ejemplo de un panel de creación personalizado para vectores.

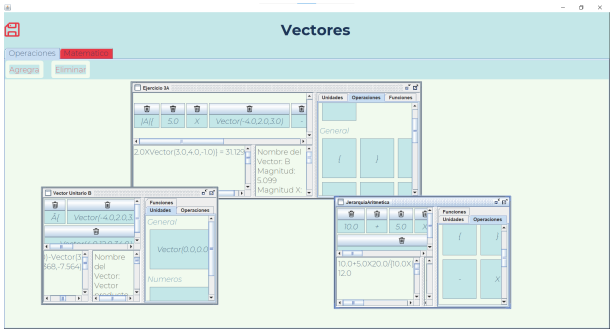



**Fig. 2.** Panel de operaciones con 3 operaciones generales resueltas.

cualquier usuario programador sin mucha experiencia en el lenguaje pueda modificarla sin necesidad de un profundo conocimiento de las bibliotecas **Swing** o **AWS**. Por ello, los procesos de renderizado son privados. Ejemplos de estos son: **OperacionesGenerales** y **ProcedimientosGenerales**. Sin embargo, las secciones relacionadas con la creación de nuevas **UnidadesMatematicas**, **OperacionesMatematicas** o **FuncionesMatematicas** son públicas y abstractas, lo que permite su posterior sobre-escritura a fin de personalizarlas. Esto último, si se tienen los conocimientos en el lenguaje (Fig.1), y si no es el caso, se pueden importar las clases base de acceso público, completamente compatibles con la interfaz y cumplen las necesidades genéricas de entrada y salida de datos de forma gráfica.

### 5.1. Panel de operaciones

Representa la principal parte gráfica del programa (Fig.2). Aquí se lleva el proceso de creación de nuevas y múltiples operaciones generales, selección de la operación, creación de la ecuación a resolver, resultados de la operación, entre otras funciones. Este panel de operaciones es un espacio de creación de sub-ventanas dentro del mismo programa las cuales se segmentan en las siguientes funcionalidades:

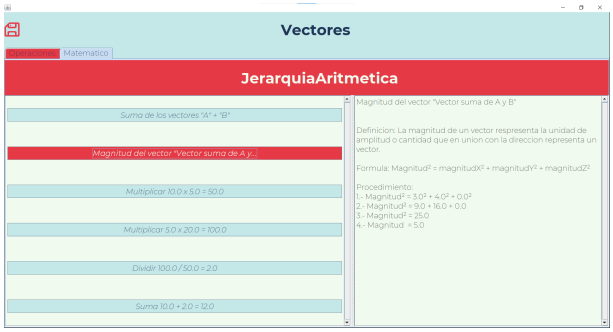



**Fig. 3.** Panel de procedimiento de una operación general llamada **"JerarquiaAritmetica"**.

| | |
|---|---|
| Panel Inspector | En él se encuentran las unidades, operaciones y funciones creadas en el programa divididas dependiendo su categoría matemática y subcategoria de tema especificado en el momento de creación. |
| Panel Operaciones | Panel con funcionalidad *Drag and Drop* el cual establece la operación matemática a resolver. Permite hacer modificaciones como el orden de la operación, remover objetos o seleccionar el objeto enfocado. |
| Panel Detalles | Representa el objeto matemático seleccionado desde el panel operaciones, aquí se da una breve descripción del objeto. En caso de no haber sido seleccionado ningún objeto será mostrado por defecto los detalles del resultado de la operación. |
| Panel Resultado | Muestra en una cadena (*string*) auto-generada el proceso matemático a realizar y su respectiva solución. |

### 5.2. Panel de procedimiento

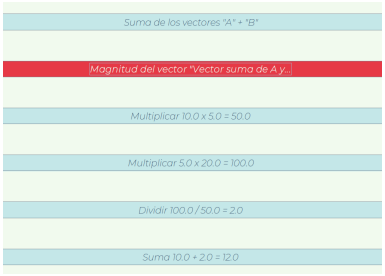
En él, se encuentra todo el procedimiento del motor matemático para llegar al resultado final de la operación general seleccionada en el panel de operaciones (Fig.3).

### 5.3. Paneles modulares genéricos

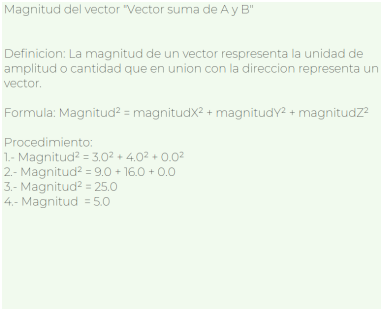
En esta categoría de paneles entran los extra de personalización como los paneles de Pizarra (Fig. 5a), notas (Fig. 5b) y graficación (Fig. 5c).

## 6. Sistema de apuntes

El sistema de apuntes consiste en la forma que el programa tiene para guardar los archivos generados y poderlos cargar desde su interfaz de entrada




**(a) Índice de procedimientos:** Permite seleccionar la operación que se visualizará en el panel derecho




**(b) Descripción:** Panel de descripción detallado con el texto auto-generado con base en la plantilla proporcionada por el programador.

**Fig. 4.** Paneles izquierdo y derecho de figura 3.

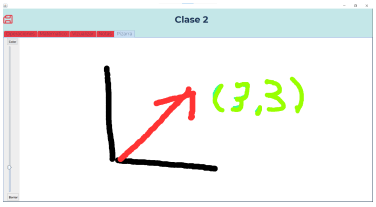
(Fig. 6). Esto tiene como objetivo facilitar la creación de material didáctico tanto para maestros o alumnos que quieren compartir la lección del día en un documento estandarizado con las operaciones realizadas, la calculadora que la hizo, explicación de cada procedimiento hecho, notas, algún dibujo y hasta una representación gráfica todo en un solo archivo.

## 7. Casos de prueba

Para el plan de pruebas se desarrollaron un total de 30 pruebas divididas entre casos de uso con vectores 3D, números reales, operaciones de diferentes tipos de unidad matemática en conjunto y operaciones con jerarquía de resolución. La metodología para la realización de pruebas se apegó a los principios **Big Bang integration testing**[9] siendo evaluado no únicamente la parte matemática sino la funcionalidad, uso del programa y el conjunto entero terminado de un ejecutable listo para la resolución y explicación de problemas matemáticos. Una vez que el motor logró resolver correctamente todas las operaciones, comprobar la funcionalidad de interfaz y la posterior explicación del procedimiento, fueron seleccionadas las más importantes con base en criterios de longitud, recursividad de operación (cantidad de sub-operaciones) y uso de diferentes unidades matemáticas inyectadas entre ellas, las principales fueron las siguientes:

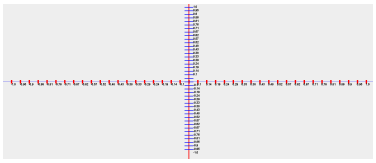
Operación con multiplicación de vectores por algún escalar para posteriormente poder realizar su resta (Op. 4), operación importante que implica el uso dinámico de 2 unidades inyectadas de forma independiente y la resolución por un orden de jerarquía. Fuente: Problema 13 primera parte inciso A libro "Métodos matemáticos para físicos"[8].

$$5{,}0 * (-4a\vec{x} + 2a\vec{y} + 3a\vec{z}) - 2{,}0 * (3a\vec{x} + 4a\vec{y} - 1a\vec{z}). \tag{4}$$




**(a)** Panel Pizarra.


**(b)** Panel Notas.



**(c)** Panel Gráficos

**Fig. 5.** Paneles modulares.

$$(-20\vec{ax} + 10\vec{ay} + 15\vec{az}) - (6\vec{ax} + 8\vec{ay} - 2\vec{az}),$$
$$(-26\vec{ax} + 2\vec{ay} + 17\vec{az}).$$

Este resultado es verificado en la Fig. 7.

Caso de pruebas con una operación tipo operación-función, la resolución de este problema implica el correcto funcionamiento de la jerarquía 4 y las jerarquías $N < 4$ en conjunto (Op.5). Fuente: Problema 13 primera parte inciso B libro "Métodos matemáticos para físicos"[8].

$$|5{,}0 * (-4\vec{ax} + 2\vec{ay} + 3\vec{az}) - 2{,}0 * (3\vec{ax} + 4\vec{ay} - 1\vec{az})|\,, \tag{5}$$
$$|(-20\vec{ax} + 10\vec{ay} + 15\vec{az}) - (6\vec{ax} + 8\vec{ay} - 2\vec{az})|\,,$$
$$|(-26\vec{ax} + 2\vec{ay} + 17\vec{az})|\,\sqrt{26^2 + 2^2 + 17^2} = 31{,}129.$$

La Fig. 8 muestra la verificación de este resultado.

Operación con uso de una alta capacidad de jerarquización y resolución de múltiples sub-operaciones en el orden correcto Op.6. Problema de caso útil para comprobar la salida del proceso matemático como fuente de estudio para el usuario no programador. Fuente: problema aleatorio generado específicamente para el motor.

$$10 + 5 * 20/10\,|(2\vec{ax} + 3\vec{ay}) + (\vec{ax} + \vec{ay})|\,, \tag{6}$$
$$10 + 100/10\,|(3\vec{ax} + 4\vec{ay}))|\,,$$
$$10 + 100/10 * (5),$$
$$10 + 100/50,$$
$$10 + 2 = 12.$$

Este resultado es comprobado con la Fig.10.

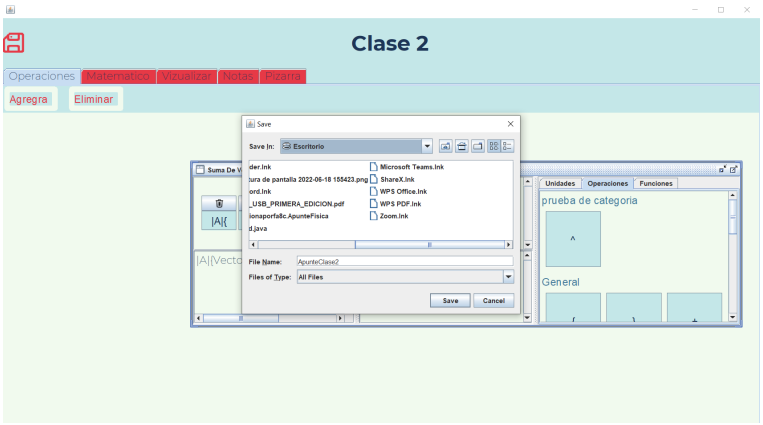



**Fig. 6.** Guardado de un nuevo apunte con todos los datos creados.


**Fig. 7.** Panel de operaciones de la ecuación 4.

## 8. Conclusiones

En este artículo se presentó el desarrollo de una biblioteca matemática para su fácil implementación o adaptación a cualquier usuario en la materia de programación o matemáticas, cumpliendo con las características solicitadas en las metas a cumplir, compatible con cualquier unidad desarrollada por usuarios programadores, reutilizable con módulos externos de otros desarrolladores, además modificable, e implementable para cualquier persona sin experiencia en el lenguaje. Posee una interfaz gráfica amigable a los usuarios finales. En la evaluación de la aplicación, se obtuvo un buen rendimiento en todos los problemas resueltos. El motor matemático con inyección de módulos externos e interfaz gráfica modular se encuentra lista en su primera versión para poder ser usada y distribuida en un ambiente de producción muy útil para las aulas de clase, alumnos autodidactas y grupos de estudio que buscan una

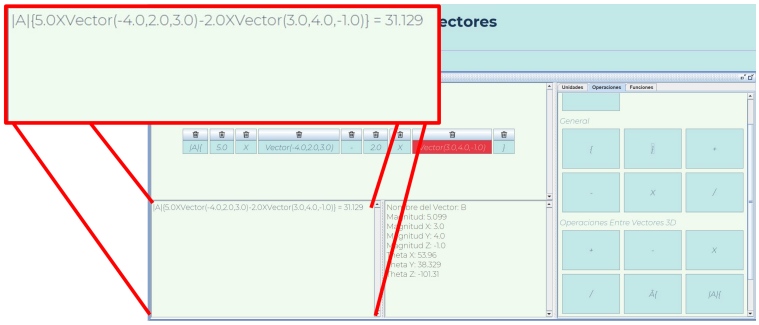



**Fig. 8.** Panel de operaciones de la ecuación 5.

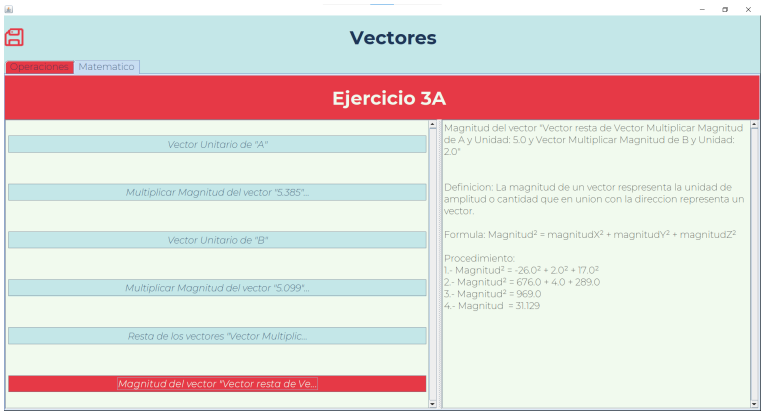



**Fig. 9.** Panel de procedimiento de la ecuación 5.

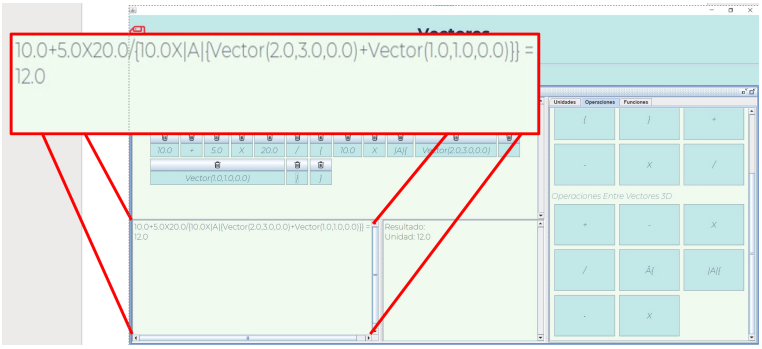



**Fig. 10.** Panel de operaciones de la ecuación 6.

rápida automatización de procesos manuales. El proyecto mostrado aquí aun se encuentra en su primera versión, pero se constata su gran potencial para popularizarse y albergar un diversidad de módulos que pueden ser creados por

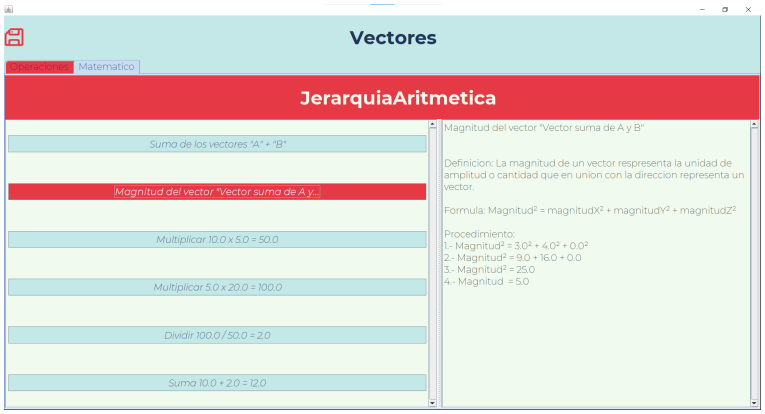



**Fig. 11.** Panel de procedimiento de la ecuación 6.

la comunidad de usuarios programadores. Esto permitiría su crecimiento y ser ofrecida como un sistema *Open Source*. Finalmente otra área de oportunidad que puede resultar atractivo entre desarrolladores interesados en implementar sus propios módulos y compartirlos a la comunidad seria migrar el proyecto de un ambiente de escritorio a una pagina web, adaptar las vistas mostradas en el presente documento a una forma mas acorde a los estándares de diseño de hoy en día y crear un sistema de desarrollo y red de difusión de módulos entre autores y usuarios no programadores. La lógica del proyecto se encuentra completamente preparada para crecer y su mayor potencial se verá alcanzado una vez que la cantidad de módulos creados pueda equiparar a los grandes programas de pago o lenguajes de programación privados explicados anteriormente en la sección 2. En términos de características, utilidades y funciones, el motor cumple con ser libre para su uso ilimitado, explicaciones paso a paso personalizadas por el usuario, entrada, salida o representación de datos según las necesidades y finalmente implementado en un lenguaje conocido y fácil de aprender como Java, representando así una gran ventaja en conjunto ante los trabajos citados en la sección 2.1, 2.2 y 2.3.

Repositorio con los códigos fuente, implementaciones y recursos gráficos listos para ser compilados, modificados, importados o ejecutados se encuentra en Github[1], igual que el motor matemático con inyección de dependencias dinámicas (Motomaticas).

## Referencias

1. Matlab, `https://www.mathworks.com/products/matlab.html`
2. Symbolab math solver - step by step calculator, `https://www.symbolab.com/`
3. The world's favorite, free math tools used by over 100 million students and teachers, `https://www.geogebra.org/`

[1] `https://github.com/hamletSolanoD/Motomaticas`

4. ¿qué es la metodología ágil?, https://www.redhat.com/es/devops/what-is-agile-methodology
5. Package nbsp;java.awt (Jun 2020), https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html
6. Package nbsp;javax.swing (Jun 2020), https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
7. Tiobe index (Jun 2022), https://www.tiobe.com/tiobe-index/
8. Arfken, G., Weber, H. J.: Mathematical methods for physicists. Harcourt Academic Press (2003)
9. Hanh, V. L., Akif, K., Traon, Y. L., Jézéque, J.-M.: Selecting an efficient oo integration testing strategy: an experimental comparison of actual strategies. In: European Conference on Object-Oriented Programming. pp. 381–401. Springer (2001)
10. Reingold, E. M.: A comment on the evaluation of Polish postfix expressions. The Computer Journal, vol. 24, no. 3, pp. 288–288 (01 1981) doi: 10.1093/comjnl/24.3.288
11. Yang, H. Y., Tempero, E., Melton, H.: An empirical study into use of dependency injection in java. In: 19th Australian Conference on Software Engineering (aswec 2008). pp. 239–247 (2008) doi: 10.1109/ASWEC.2008.4483212