Universidad Michoacana de San Nicolás de Hidalgo

Facultad de Ingeniería Eléctrica

"Implementación de algoritmos para realizar operaciones aritméticas básicas en Complex Fans"

TESIS

Para obtener el título de:
INGENIERO EN COMPUTACIÓN

Presenta:
Manuel Cipriano Hernández

Asesor:
Juan José Flores Romero
Doctor en Filosofía en Ciencias Computacionale

Morelia, Michoacán
Enero 2017

Agradecimientos

Agradezco a Dios por la vida y su gran amor hacía mí, y a su hijo Jesucristo por su misericordia y gracia.

A mi papá, Manuel, por su diligencia en ayudarnos a estudiar una carrera y por su ejemplo de trabajo arduo.

A mi mamá, Blanca, por enseñarme a usar mi tiempo en algo productivo y por su ejemplo de valentía y fortaleza para enfrentar los desafíos de la vida.

A mi hermana, Alma, por su cariño, su confianza hacía mi persona y por su ejemplo de bondad y de amor hacia las demás personas.

A mi cuñado, Roberto, por su ejemplo de valentía al enfrentar los desafíos de un padre joven y por su ejemplo de aprovechar los recursos con los que se cuenta.

A mi hermano, Agustín, por su ejemplo de lucha para hacer un sueño realidad, de leer libros y de aprender de diferentes temas.

A mi hermano, Francisco, por su ejemplo de diligencia en los estudios y de servicio a los demás.

A todos los compañeros de la facultad que me brindaron su ayuda durante el estudio de mi carrera.

A mis maestros y profesores desde mi niñez hasta mi licenciatura por sus enseñanzas y consejos.

A mi maestro de Seminario de Tesis, Ingeniero Ignacio Franco, por sus observaciones hacía mi tesis y por siempre recordarme trabajar en mi tesis.

A mi asesor de tesis, Doctor Juan José Flores, por su guía y consejos al término de mi carrera.

Dedicatoria

Este trabajo lo dedico a mis Padres por su esfuerzo en procurar que mis hermanos y yo recibiéramos una formación académica y profesional.

Índice general

Resumen

Un vector puede ser representado de dos maneras, en forma rectangular y en forma polar. Hay aplicaciones de vectores donde tiene mucho más sentido hablar de vectores en su forma polar que en su forma rectangular. Por ejemplo, en ingeniería eléctrica se usan los fasores, que son un tipo de vectores representados en forma polar (aunque no siempre son representados en forma polar), para resolver circuitos lineales en estado estable sinusoidal.

Cuando se trabaja con incertidumbre, un fasor en lugar de tener valores precisos tiene intervalos de valores para su magnitud y ángulo. Si dibujáramos este tipo de fasor, el objeto resultante sería un objeto semicircular, parecido a un abanico. En el artículo "Complex fans: A representation for vectors in polar form with interval attributes"del Doctor Juan José Flores Romero se llama a este objeto Complex Fan (en español, Abanico Complejo) y se proponen algoritmos para realizar operaciones aritméticas sobre ellas.

En esta tesis se implementaron los algoritmos propuestos en el artículo antes mencionado junto con las funciones adicionales para implementar dichos algoritmos. La implementación se hizo en Java y en Mathematica. Además se realizaron pruebas a las implementaciones para verificar su buen funcionamiento.

Palabras clave

Vectores, números complejos, fasores, intervalos, incertidumbre, abanicos complejos.

Abstract

A vector can be represented rectangularly and polarly. There are vector applications where it makes more sense to represent vectors in polar form rather than in its rectangular form, for example, in electrical engineering, phasors, are normally represented in polar form (although they are not always represented in polar form), are used to solve linear circuits in sinusoidal steady state.

When working with uncertainties, for example, when phasors instead of having a value for the magnitude and a value for angle has a range of values for the magnitude and a range of values for angle, if we drew this kind of phasor, the resulting object would be a semi-circular object, like a fan, in the article "Complex fans: A representation for vectors in polar form with interval attributes"from Doctor Juan José Flores call this object Complex Fan and algorithms are proposed to perform arithmetic operations on them.

In this thesis the algorithms proposed in the article referred along with additional functions to the algorithms can function were implemented. The implementation was done in Java and Mathematica. In addition tests were performed to verify the proper operation of implementations.

Keywords

Vectors, complex numbers, phasors, intervals, uncertainty, complex fans.

Índice de figuras

Índice de tablas

Índice de algoritmos

Glosario de términos

Mayúsculas

I_m	—	Magnitud del fasor I. Sección 1.1
I	—	Fasor. Sección 1.1
V, V_i, V_{ij}	—	Abanicos complejos.
V_m, V_α	—	Magnitud, Ángulo del abanico complejo V.
RLC	—	Resistencia, Inductancia y Capacitancia.
CA	—	Corriente Alterna.
Z	—	Impedancia. Sección 4.2.2.
R	—	Resistencia. Sección 4.2.2.
C	—	Capacitor. Sección 4.2.2.
L	—	Inductor. Sección 4.2.2.
I	—	Corriente. Sección 4.2.2.
V	—	Voltaje. Sección 4.2.2.

Minúsculas

$\overrightarrow{v}$	—	Vector.
v_x, v_y	—	Componentes ortogonales del vector v. Sección 1.1
v_m, v_α	—	Magnitud, ángulo del vector v. Sección 1.1
$i(t)$	—	Función dependiente del tiempo.
a, b, c, d	—	Magnitudes.
w	—	Velocidad angular.

Letras griegas

ϕ	—	Ángulo de fase. Sección 1.1
α_i	—	Ángulo.

Capítulo 1

Introducción

En esta tesis se presentan las implementaciones realizadas en Java y en Mathematica de los algoritmos propuestos en [Flores, 1998] para la aritmética de abanicos complejos (en inglés, complex fans). Se presentan, además, las diferentes pruebas que se realizaron a las implementaciones para verificar su buen funcionamiento, con sus respectivos resultados.

En este capítulo se define que es un Abanico Complejo (Complex Fan), se analiza brevemente los problemas donde se puede aplicar su uso, se informa brevemente de los antecedentes de este trabajo, se establecen los objetivos y se realiza la descripción de los capítulos que contiene esta tesis.

1.1. Fasores y Abanicos Complejos

Hay dos formas de representar un vector, en forma rectangular (Ecuación 1.1) y en forma polar (Ecuación 1.2). En ambos casos se necesitan dos números para expresar el vector. En la forma rectangular, esos dos números son los componentes del vector en el eje x y en el eje y. En la forma polar, un vector está expresado por su magnitud y su ángulo.

$$\overrightarrow{v} = (v_x, v_y) \tag{1.1}$$

$$\overrightarrow{v} = v_m \angle v_\alpha \tag{1.2}$$

Tradicionalmente en Ingeniería, los vectores son representados en la forma rectangular, porque las operaciones aritméticas son simples en esta representación. Sin embargo, hay algunas aplicaciones donde tiene más sentido hablar acerca de la magnitud y el ángulo de un vector, que de sus componentes ortogonales. Por ejemplo en Ingeniería Eléctrica, en la solución de circuitos lineales en estado estable senoidal, a menudo se habla de ángulos de fase y magnitudes de fasores.

Un fasor es un tipo de vector, una representación compleja abreviada de una tensión o corriente senoidal [William H. Hayt, 2007]. Por ejemplo, si se tiene la corriente senoidal real:

$$i(t) = I_m \cos(wt + \phi) \tag{1.3}$$

el fasor que lo representa es:

$$I = I_m \angle \phi \tag{1.4}$$

donde el fasor es el número complejo constante que contiene la magnitud y fase de la senoide.

Para simplificar la notación, los fasores se escriben habitualmente en forma polar. Los fasores también se utilizan en Óptica, Ingeniería de Telecomunicaciones y Acústica.

Dado un fasor, si en lugar de un valor real para su magnitud y uno para su ángulo, estos toman un rango de valores, que estén sobre un intervalo, es decir, valores con incertidumbre. El objeto resultante, en lugar de denotar un punto en el plano ahora se extiende sobre segmentos circulares, similares a un abanico, a esto es a lo que en [Flores, 1998] le llaman Abanico Complejo. Por ejemplo, el Abanico Complejo V de la Figura 1.1 es un fasor, el cual su magnitud abarca de a a b, y su ángulo abarca de α_1 a α_2.

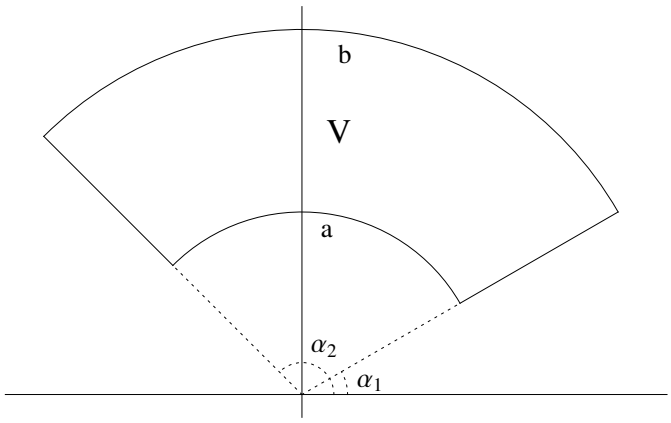


Figura 1.1: Un Abanico Complejo

El abanico complejo de la Figura 1.1 puede ser expresado como

$$V = V_m \angle V_\alpha \tag{1.5}$$

donde V_m representa la magnitud de V y V_α su ángulo, además $V_m = [a, b]$ y $V_\alpha = [\alpha_1, \alpha_2]$, de modo que también se puede representar de la siguiente manera

$$V = [a, b] \angle [\alpha_1, \alpha_2] \tag{1.6}$$

Nótese que si $a = b$ y $\alpha_1 = \alpha_2$, el abanico complejo se reduce a un fasor (es decir, un punto en el plano complejo).

Debido a que un fasor es un tipo de vector. Si se ve a un vector como el par magnitud-ángulo (veáse la Figura 1.2), un abanico complejo es el conjunto de todos los vectores del producto cartesiano de la magnitud y el ángulo del abanico complejo (veáse la Figura 1.3). Cabe mencionar que en la Figura 1.3 solo se muestran algunos ejemplos de vectores (en color gris) como elementos del abanico complejo dado (en color azul y con lineas punteadas), ya que todo vector que tenga una magnitud igual a ó este entre a y b, y su ángulo sea igual a ó este entre α_1 y α_2, pertenece al conjunto de vectores que representa el abanico complejo.

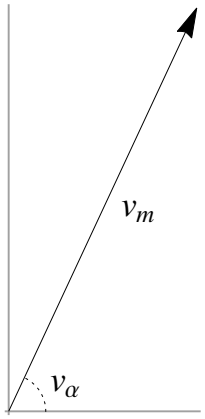


Figura 1.2: Un vector en su representación polar

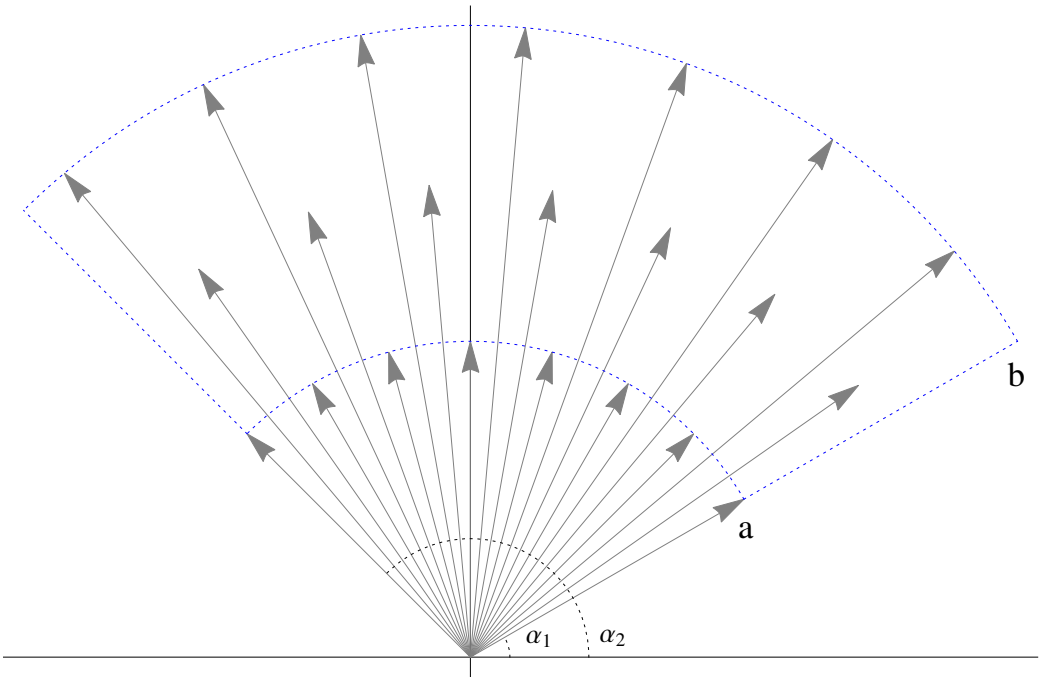
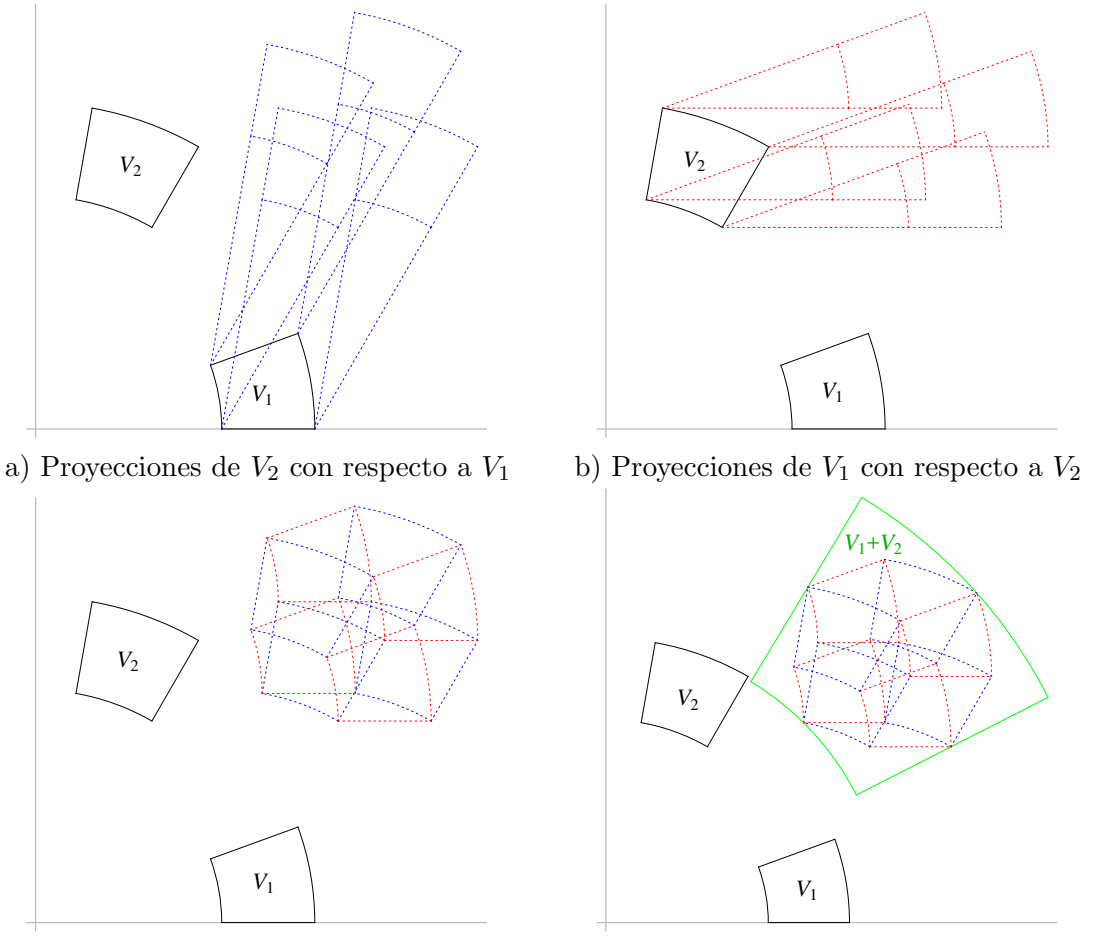


Figura 1.3: Un abanico complejo como un conjunto de vectores

1.2. Planteamiento del Problema

En la suma de dos vectores se usa la proyección del primer operando con respecto al segundo o viceversa, donde el resultado de la suma es el vector que va del origen al valor apuntado por el operando proyectado. Si se usan proyecciones en la suma de dos abanicos complejos, se pueden usar las esquinas de los abanicos complejos para proyectar un operando con respecto al otro, ya que esto da un indicio de los límites que tendrá el resultado de la operación, esto se puede observar de manera gráfica en la Figura 1.4, donde la Subfigura a) muestra las proyecciones del segundo operando (V_2) con respecto a las esquinas del primer operando (V_1), viceversa en la Subfigura b), en la Subfigura c) se pueden observar las proyecciones de V_1 con respecto a V_2 junto con las proyecciones de V_2 con respecto a V_1, donde además se puede notar que el resultado de la suma de los abanicos complejos tendría que englobar todos los abanicos complejos con líneas punteadas rojas y azules, lo cual da como resultado una forma geométrica irregular, esta forma queda fuera del dominio de los abanicos complejos, es decir, no se podría expresar de la forma $V = [a, b]\angle[\alpha_1, \alpha_2]$. El algoritmo que se implementó en esta tesis para la suma de dos abanicos complejos, nos regresa como resultado el abanico complejo mínimo que para este caso engloba los abanicos complejos con líneas punteadas como se muestra en la Subfigura d).



a) Proyecciones de V_2 con respecto a V_1

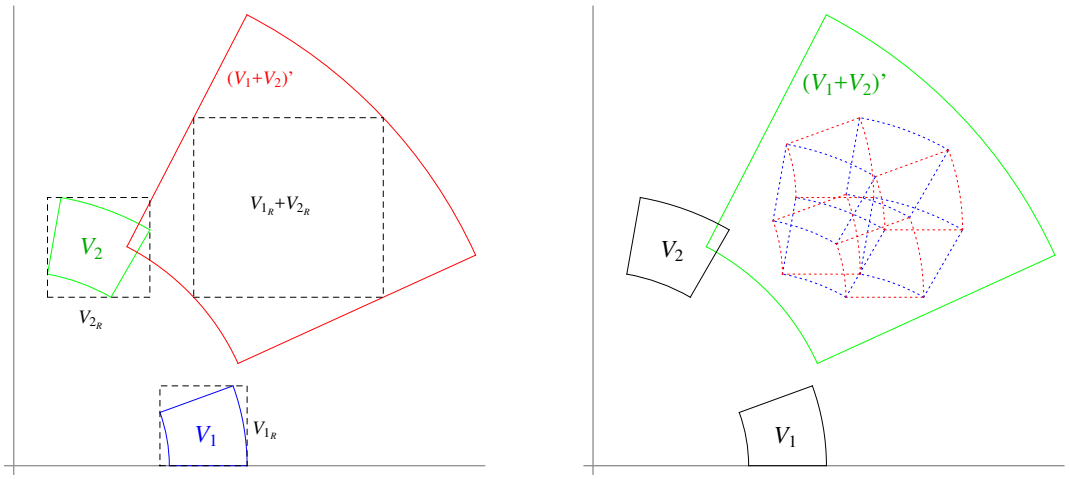
b) Proyecciones de V_1 con respecto a V_2

c) Sobreposición de Subfiguras a) y b)

d) Abanico complejo mínimo como resultado

Figura 1.4: Suma de dos abanicos complejos

Otra manera de sumar los abanicos complejos de la Figura 1.4, es usando su representación rectangular, como se puede observar en la Subfigura a) de la Figura 1.5, donde se puede notar que V_1 se convierte en V_{1_R}, V_2 en V_{2_R} y el resultado de su suma es $V_{1_R} + V_{2_R}$, al cambiar este resultado a la forma polar se convierte en $(V_1 + V_2)'$. Al dibujar el resultado $(V_1 + V_2)'$ junto con las proyecciones de V_2 con respecto a V_1 y viceversa, se puede notar que este resultado no es el abanico complejo mínimo que engloba los abanicos complejos con líneas punteadas rojas y azules.



b) Resultado en la forma rectangular a) Abanico complejo no mínimo como resultado

Figura 1.5: Imprecisión añadida por cambio de representación

En la Figura 1.6 se muestra la comparación de resultados, donde se puede observar que el resultado proveniente al cambiar de representación, $(V_1+V_2)'$, contiene muchos más resultados espurios que el resultado que se obtiene por el algoritmo implementado en esta tesis, $V_1 + V_2$.

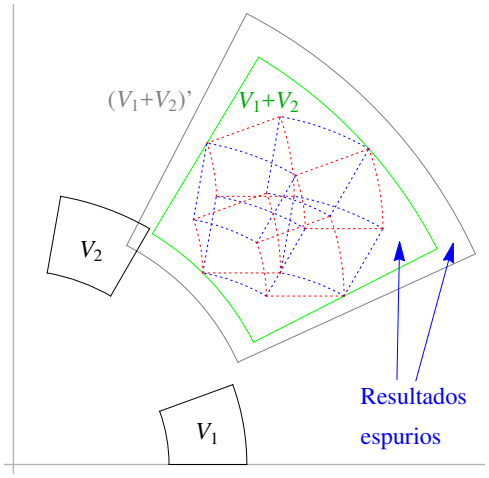


Figura 1.6: Comparación de resultados

La búsqueda de resultados mínimos, es decir, los abanicos complejos mínimos que contienen todos los resultados posibles, es lo que impulsó el desarrollo de los algoritmos implementados en esta tesis. Estos algoritmos trabajan sobre vectores en forma polar con atributos de intervalo (abanicos complejos). La posibilidad de que otras personas pudieran probar y usar

los algoritmos para sus propias investigaciones o trabajos impulsó el desarrollo de esta tesis.

Hay áreas dentro de la Física que se usan abundantemente los vectores, los abanicos complejos son una herramienta matemática que habilita estas áreas para tratar con problemas que incluyen incertidumbre o información incompleta. Por ejemplo, si se tiene el siguiente problema:
"Un carro es conducido hacia el Este una distancia de 50 millas, después 30 millas hacia el Norte y por último 25 millas en una dirección 30° grados del Norte hacia el Este. Dibujar el diagrama vectorial y determinar el desplazamiento total del carro desde su punto de partida."
se tienen los siguientes datos del problema:

$$\overrightarrow{a} = 50\angle 0^\circ$$
$$\overrightarrow{b} = 30\angle 90^\circ$$
$$\overrightarrow{c} = 25\angle 60^\circ$$

y se pide calcular el vector de desplazamiento total del carro, de acuerdo a los desplazamientos parciales anteriores, esto es sencillo de resolver, usando la forma rectangular de cada desplazamiento parcial y efectuando su suma, posteriormente cambiar a representación polar el resultado. Pero que pasa si se incluye el margen de error que cada medidor o sensor físico pueda poseer al realizar las mediciones, por ejemplo:
"¿Cuál será la posición final del carro si el odómetro del carro tiene un porcentaje de error de $\pm 2\,\%$ y la brújula tiene un margen de error de $\pm 3^\circ$ en cada medición?"
los datos del problema se convierten en:

$$\text{cfA} = [49, 51]\angle[-3^\circ, 3^\circ]$$
$$\text{cfB} = [29.4, 30.6]\angle[87^\circ, 93^\circ]$$
$$\text{cfC} = [24.5, 25.5]\angle[57^\circ, 63^\circ]$$

Este es un ejemplo de aplicación de los abanicos complejos dentro de la Física y se puede ver su análisis completo junto con los resultados obtenidos en la Sección 4.2.1 de esta tesis. Los abanicos complejos también se pueden aplicar a los números complejos (expresados en forma polar), si se deja que la magnitud y el ángulo estén sobre intervalos de valores en lugar de un solo valor para cada uno.

En [Flores, 1997] se presenta un marco de trabajo llamado Análisis Fasorial Cualitativo (en inglés, Qualitative Phasor Analysis), para el razonamiento sobre circuitos eléctricos lineales en estado estable senoidal. El proceso de razonamiento decae en un modelo basado en restricciones, derivado de la teoría electromagnética y generada automáticamente de la estructura del circuito. En este marco de trabajo los abanicos complejos proporcionan un mecanismo general para representar los fasores a diferentes niveles de precisión. El Análisis Fasorial Cualitativo es capaz de realizar las siguientes tareas de razonamiento: análisis, diseño de parámetros, diagnóstico, diseño de controladores y simplificación de la estructura. Toda estas tareas de razonamiento están basados en los resultados del análisis de un circuito, la cuales decaen en el cálculo de cantidades representas como abanicos complejos. No se presenta un ejemplo de aplicación de abanicos complejos sobre este tema, ya que es un tema bastante complejo, más bien se presenta un ejemplo básico de análisis de un circuito en Corriente Alterna (CA) en la sección 4.2.2, donde se presenta el circuito de la Figura 1.7, que contiene una Resistencia, un Capacitor y un Inductor, es decir, un circuito RLC. Se realiza el análisis

del circuito tomando en cuenta la tolerancia (margen de error) de los elementos pasivos, para la resistencia $\pm 5\,\%$, para el condensador $\pm 3\,\%$ y para el inductor $\pm 2\,\%$, obteniéndose así, con la ayuda de abanicos complejos, información acerca del comportamiento de las tensiones y corrientes en cada elemento del circuito.

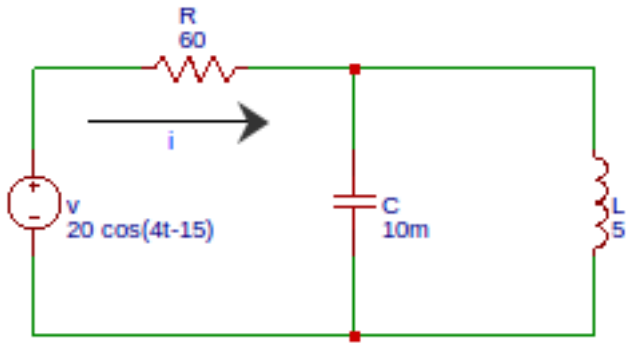


Figura 1.7: Circuito RLC en CA a analizar

1.3. Antecedentes

Los algoritmos a implementar en esta tesis, anteriormente ya habían sido implementados, en [Flores, 1998] se describe la implementación en el lenguaje de programación Common Lisp, usando Common Lisp Object Sytem (CLOS) para programación orientada a objetos.

1.4. Objetivos de la Tesis

Los algoritmos a implementar fueron propuestos en [Flores, 1998] y son para realizar operaciones aritméticas en abanicos complejos. Estos algoritmos implican operaciones adicionales, como descomponer un abanico complejo de acuerdo a su intersección con el plano cartesiano, unión de intervalos, intersección de intervalos, unión de abanicos complejos, una función general para la adición de 2 abanicos complejos que utilice los algoritmos de los 3 casos de adición, etc. Por lo tanto los objetivos de esta tesis son:

- Implementar los algoritmos.
- Realizar pruebas elementales a la implementación.
- Realizar pruebas de aplicaciones físicas a la implementación.

Se realizó una implementación en Java y otra en Mathematica. En Java se generaron clases para que puedan ser usadas ya sea por línea de comandos o dentro de otro proyecto con interfaces gráficas. En Mathematica se creó una biblioteca de funciones para que puedan ser cargadas en una Notebook o dentro de otra biblioteca.

1.5. Descripción de Capítulos

El Capítulo 2 presenta los algoritmos implementados, junto con las correcciones y observaciones que se encontraron durante la implementación.

El Capítulo 3 presenta la implementación en Java y en Mathematica.

El Capítulo 4 presenta pruebas elementales que se realizaron a las implementaciones con sus respectivos resultados, así como resultados de aplicaciones físicas.

El Capítulo 5 presenta las conclusiones de esta tesis y se sugiere algunos posibles trabajos futuros.

Capítulo 2

Aritmética de Abanicos Complejos

En este capítulo se presentan los algoritmos propuestos en [Flores, 1998] para la aritmética de abanicos complejos: Negación, Producto, División, Sustracción y Adición. Además se aprovecha la oportunidad para presentar algunas correcciones que se hicieron a los algoritmos de pequeños errores que se encontraron durante la prueba de los mismos mientras se hacía la implementación.

Dados los abanicos complejos:

$$V_1 = [a, b]\angle[\alpha_1, \alpha_2] \tag{2.1}$$

$$V_2 = [c, d]\angle[\alpha_3, \alpha_4] \tag{2.2}$$

Se pueden realizar operaciones aritméticas sobre ellas con los siguientes algoritmos.

2.1. Producto

El producto de dos abanicos complejos, $V = V_1 * V_2$, esta dado por la formula

$$V = ([a, b] * [c, d])\angle([\alpha_1, \alpha_2] + [\alpha_3, \alpha_4]) \tag{2.3}$$

2.2. División

La división de dos abanicos complejos, $V = V_1/V_2$, es análogo al producto

$$V = ([a, b]/[c, d])\angle([\alpha_1, \alpha_2] - [\alpha_3, \alpha_4]) \tag{2.4}$$

2.3. Negación

La negación de un abanico complejo, $V = -V_1$, es otro abanico complejo con la misma magnitud y cuyo ángulo es el complemento del ángulo del abanico original

$$V = [a, b]\angle([\alpha_1, \alpha_2] + [180, 180]) \tag{2.5}$$

Las operaciones aritméticas de Producto, División y Negación, utilizan aritmética de intervalos, la cual se muestra en la Tabla 2.1.

Tabla 2.1: Aritmética de intervalos.

Negación	$-(a,b) = (-b,-a)$
Adición	$(a,b) + (c,d) = (a+c, b+d)$
Producto	$(a,b) * (c,d) = (a*c, b*d)$
Sustracción	$(a,b) - (c,d) = (a-d, b-c)$
División	$(a,b)/(c,d) = (a/d, b/c)$ suponiendo que todas las magnitudes son positivas y que $0 \notin (c,d)$

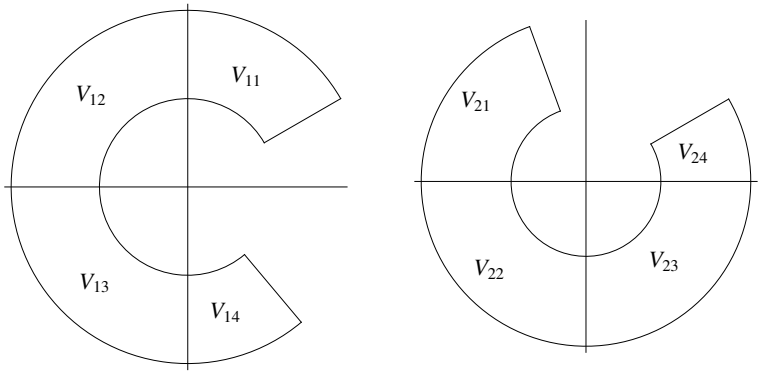
2.4. Sustracción

La sustracción, $V = V_1 - V_2$, se define en términos de la adición y la negación como sigue

$$V = V_1 + (-V_2) \tag{2.6}$$

2.5. Adición

Para la adición de dos abanicos complejos, se descomponen V_1 y V_2, en 4 abanicos complejos, es decir, en sus intersecciones con cada uno de los 4 cuadrantes del plano cartesiano como se muestra en la Figura 2.1, en la Subfigura a) se pueden observar las partes que conforman V_1 y en la Subfigura b) las partes de V_2; en la figura, la numeración de partes de cada abanico complejo no tiene nada que ver con el cuadrante en donde se localiza.



a) V_1 b) V_2

Figura 2.1: Abanicos complejos partidos

De la Figura 2.1 se observa que:

$$V_1 = V_{11} \cup V_{12} \cup V_{13} \cup V_{14} \tag{2.7}$$

$$V_2 = V_{21} \cup V_{22} \cup V_{23} \cup V_{24} \tag{2.8}$$

Al sumar V_1 y V_2,

$$V = V_1 + V_2$$
$$= (V_{11} \cup V_{12} \cup V_{13} \cup V_{14}) + (V_{21} \cup V_{22} \cup V_{23} \cup V_{24})$$

Además,

$$V_1 + V_2 = \{v + w | (v,w) \in V_1 \times V_2\} \tag{2.9}$$

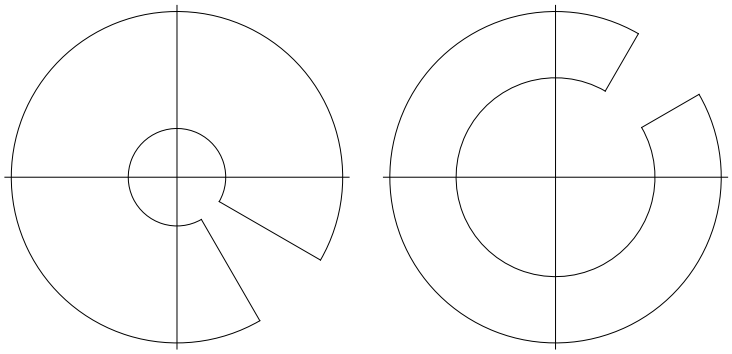
Por lo tanto,

$$V = \{v + w | (v, w) \in (V_{11} \cup V_{12} \cup V_{13} \cup V_{14}) \times (V_{21} \cup V_{22} \cup V_{23} \cup V_{24})\}$$

Y ya que el producto cartesiano distribuye sobre la unión,

$$\begin{aligned} V &= \{v + w | (v, w) \in (V_{11} \times V_{21} \cup V_{11} \times V_{22} \cup V_{11} \times V_{23} \cup ...)\} \\ &= \bigcup_{i,j=1...4} V_{1i} + V_{2j} \end{aligned} \tag{2.10}$$

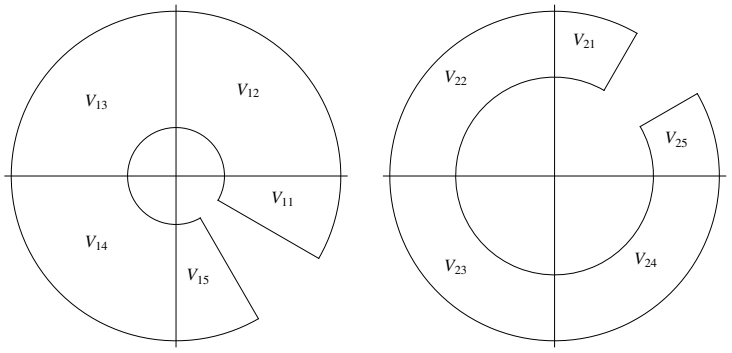
Toda la deducción anterior proviene de [Flores, 1998], en donde se establece que con solo cuatro partes se pueden representar todas las partes posibles de un abanico complejo. Durante la implementación se encontraron contraejemplos como los que se muestran en la Figura 2.2.



a) V_3 b) V_4

Figura 2.2: Contraejemplos al partir un abanico complejo

Al observar la Figura 2.2, se puede notar que en el abanico complejo de la Subfigura a) no se pueden representar las intersecciones con el cuarto cuadrante con solo un abanico complejo, como también las intersecciones con el primer cuadrante del abanico complejo de la Subfigura b). Las cuales suponen que con solo cuatro partes no son suficientes, sino que con cinco partes; cinco es el numero máximo de partes que puede haber (veáse la Figura 2.3).



a) V_1 b) V_2

Figura 2.3: Partes que puede tener un abanico complejo

Asumiendo lo anterior, la adición queda de la siguiente manera:

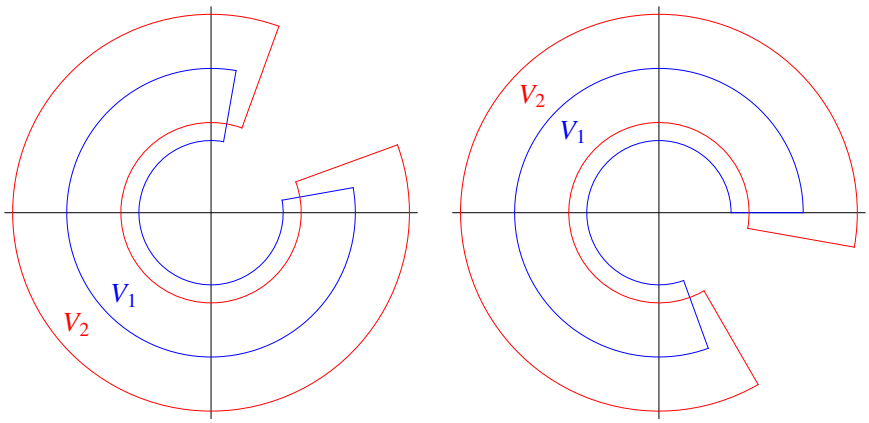
$$V = \bigcup_{i,j=1...5} V_{1i} + V_{2j} \tag{2.11}$$

Donde ahora,

$$V_1 = V_{11} \cup V_{12} \cup V_{13} \cup V_{14} \cup V_{15} \tag{2.12}$$

$$V_2 = V_{21} \cup V_{22} \cup V_{23} \cup V_{24} \cup V_{25} \tag{2.13}$$

Cabe mencionar que para la adición de dos abanicos complejos, en [Flores, 1998] se toma en cuenta que el primer operando de la adición siempre empieza sobre el eje X positivo, es decir, el primer extremo del intervalo de ángulo del primer abanico complejo es igual a cero, por lo tanto, si el primer operando no se encuentra sobre el eje X positivo, se debe aplicar una rotación a los dos operandos antes de realizar la adición, y cuando se obtenga el resultado de la adición, se corrige la rotación aplicada antes de la operación en el resultado de la operación. Por ejemplo, de la Figura 2.4, si se quiere sumar los abanicos complejos de la Subfigura a), se debe aplicar una rotación a los dos operandos, ya que el abanico complejo V_1 no empieza sobre el eje X positivo, para que queden como se muestran en la Subfigura b).



a) V_1 y V_2 originales b) V_1 y V_2 con rotación

Figura 2.4: Rotación de operandos antes de la adición

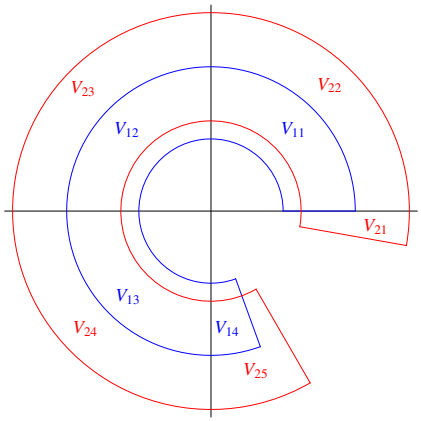


Figura 2.5: Operandos rotados y partidos

Tomando en cuenta la consideración anterior, el primer operando de una adición de dos abanicos complejos nunca tendrá cinco partes, pero el segundo operando puede tener hasta cinco partes como se puede observar en la Figura 2.5.

Cada suma parcial es efectuada con abanicos complejos que no se extienden más allá de 90 grados. Para ello se categorizó cada suma parcial en uno de tres casos: cuando los sumandos están en el mismo, adyacentes u opuestos cuadrantes.

Caso 1: Abanicos complejos en el mismo cuadrante. Con este algoritmo se considera el caso cuando ambos abanicos complejos están en el mismo cuadrante (véase la Figura 2.6), el análisis fue hecho para el primer cuadrante; para otros cuadrantes, solamente se aplica una rotación, la cual es corregida en el resultado de la operación. El algoritmo a utilizar para este caso se puede observar en el Algoritmo 1.

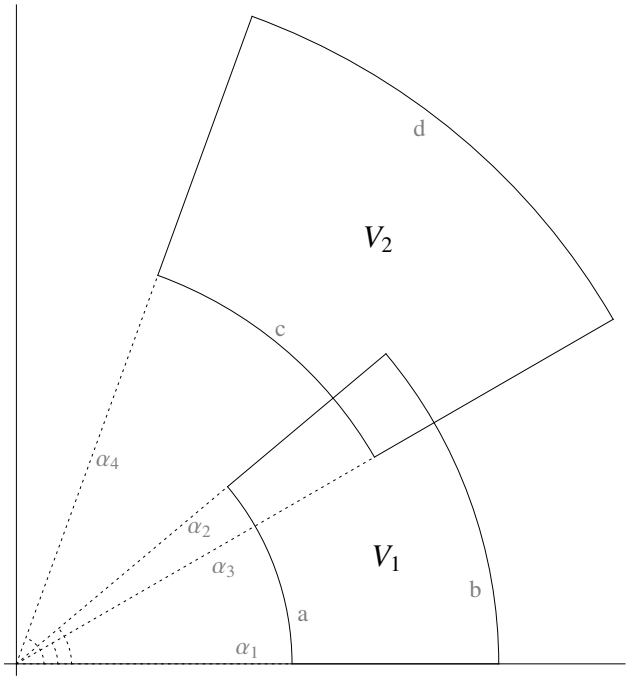


Figura 2.6: Adición caso 1

El Algoritmo 1 recibe dos abanicos complejos como parámetros, V_1 y V_2. El algoritmo regresa el resultado de la suma $V = V_1 + V_2$, resultando la magnitud de V como $V_m = [V_{m_{min}}, V_{m_{max}}]$ y su ángulo como $V_\alpha = [V_{\alpha_{min}}, V_{\alpha_{max}}]$. En las líneas del 1 al 6, se calculan los ángulos θ_{min} y θ_{max}, los cuales servirán para calcular $V_{m_{min}}$ y $V_{m_{max}}$ en las líneas 7 y 8. Por último entre las líneas 9 al 16 se calculan $V_{\alpha_{min}}$ y $V_{\alpha_{max}}$.

Para los algoritmos de los casos restantes, se hace mención de los números de esquina de cada operando. La numeración de esquinas que se utiliza se muestra en la Figura 2.7. Además, se mencionan las sumas entre esquinas i y j lo cual se indica por el punto ij, es decir, la suma del vector que va del origen a la esquina i con el vector del origen a j, resulta en el vector que va del origen al punto ij. Por ejemplo, la suma del punto 2 con el 5 resulta en el punto 25, como se muestra en la misma figura.

Algoritmo 1 AddCase1(V_1, V_2)

Require: $V_1 = [a, b]\angle[\alpha_1, \alpha_2]$ and $V_2 = [c, d]\angle[\alpha_3, \alpha_4]$
Ensure: $V = [V_{m_{min}}, V_{m_{max}}]\angle[V_{\alpha_{min}}, V_{\alpha_{max}}]$

if $\alpha_2 >= \alpha_3$ **then**
  $\theta_{min} = 0$
else
  $\theta_{min} = \alpha_2 - \alpha_3$
end if
$\theta_{max} = max(\alpha_4 - \alpha_1, \alpha_2 - \alpha_3)$
$V_{m_{min}} = \sqrt{a^2 + c^2 + 2ac\cos\theta_{max}}$
$V_{m_{max}} = \sqrt{b^2 + d^2 + 2bd\cos\theta_{min}}$
$V_{\alpha_{min}} = \tan^{-1}\left(\frac{b\sin\alpha_1 + c\sin\alpha_3}{b\cos\alpha_1 + c\cos\alpha_3}\right)$
if $\alpha_2 < \alpha_4$ **then**
  $V_{\alpha_{max}} = \tan^{-1}\left(\frac{a\sin\alpha_2 + d\sin\alpha_4}{a\cos\alpha_2 + d\cos\alpha_4}\right)$
else if $\alpha_2 > \alpha_4$ **then**
  $V_{\alpha_{max}} = \tan^{-1}\left(\frac{b\sin\alpha_2 + c\sin\alpha_4}{b\cos\alpha_2 + c\cos\alpha_4}\right)$
else if $\alpha_2 = \alpha_4$ **then**
  $V_{\alpha_{max}} = \alpha_2$
end if
return $([V_{m_{min}}, V_{m_{max}}]\angle[V_{\alpha_{min}}, V_{\alpha_{max}}])$

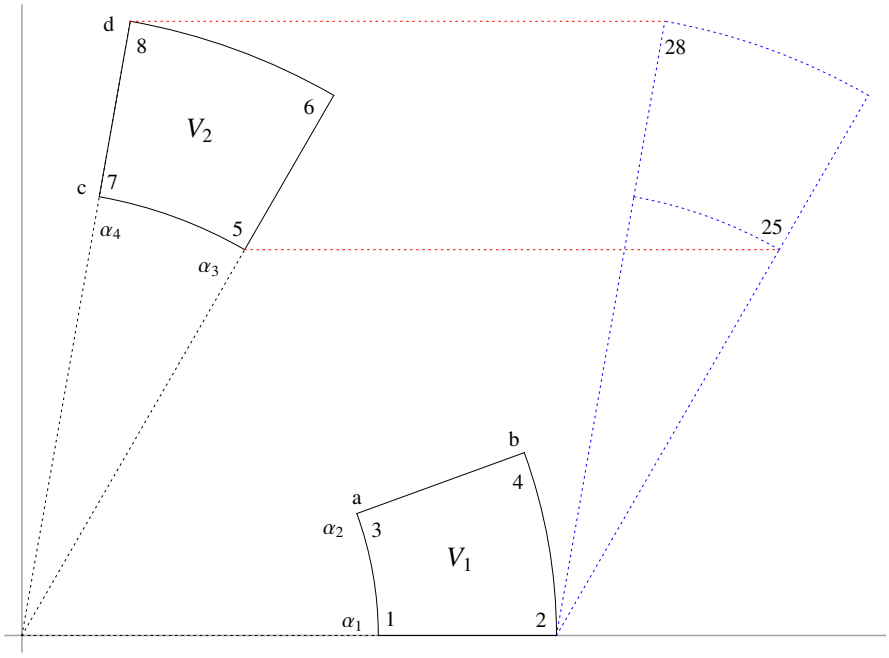


Figura 2.7: Numeración de esquinas de abanicos complejos

Caso 2: Abanicos complejos en cuadrantes adyacentes. Este caso supone que el abanico complejo V_1 se encuentra en el primer cuadrante y el abanico complejo V_2 en el segundo cuadrante (véase la Figura 2.8). Este caso se separa en dos partes un algoritmo para calcular la magnitud (véase el Algoritmo 2) y otro para calcular el ángulo (véase el Algoritmo 3).

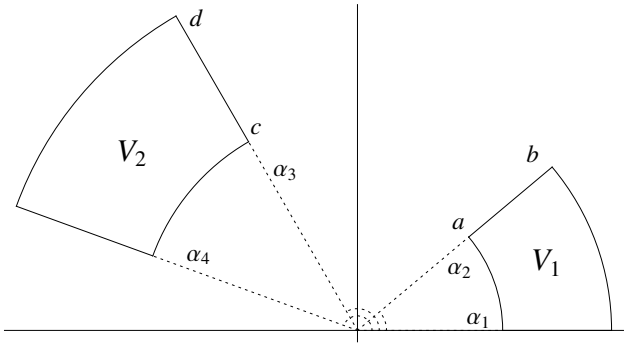


Figura 2.8: Adición caso 2

Algoritmo 2 MagnitudeCase2(V_1, V_2)

Require: $V_1 = [a, b]\angle[\alpha_1, \alpha_2]$ and $V_2 = [c, d]\angle[\alpha_3, \alpha_4]$
Ensure: $V_m = [V_{m_{min}}, V_{m_{max}}]$
1: $\theta_{min} = \alpha_3 - \alpha_2$
2: $\theta_{max} = \alpha_4 - \alpha_1$
3: **if** $\theta_{min} <= 90$ **then**
4: $\quad V_{m_{max}} = \sqrt{b^2 + d^2 + 2bd\cos\theta_{min}}$
5: **else**
6: $\quad V_{m_{max}} = max(p35_m, p36_m, p45_m, p46_m)$
7: **end if**
8: $I_x = V_{1_m} \cap (-V_{2_m}\cos\theta_{max})$
9: **if** $I_x \neq \varnothing$ **then**
10: $\quad x_m = I_{x_{min}}$
11: **else if** $a > -d\cos\theta_{max}$ **then**
12: $\quad x_m = a$
13: **else if** $b < -c\cos\theta_{max}$ **then**
14: $\quad x_m = b$
15: **end if**
16: $I_y = V_{2_m} \cap (-V_{1_m}\cos\theta_{max})$
17: **if** $I_y \neq \varnothing$ **then**
18: $\quad y_m = I_{y_{min}}$
19: **else if** $c > -b\cos\theta_{max}$ **then**
20: $\quad y_m = c$
21: **else if** $d < -a\cos\theta_{max}$ **then**
22: $\quad y_m = d$
23: **end if**
24: $V_{m_{min}} = \sqrt{x_m^2 + y_m^2 + 2x_m y_m\cos\theta_{max}}$
25: **return** $([V_{m_{min}}, V_{m_{max}}])$

El Algoritmo 2 recibe dos abanicos complejos como parámetros, V_1 y V_2. El algoritmo regresa el intervalo de magnitud V_m resultante para la suma $V = V_1 + V_2$. En las líneas 1 y 2, se calculan los ángulos θ_{min} y θ_{max}, los cuales servirán para calcular $V_{m_{min}}$ y $V_{m_{max}}$. Entre las líneas del 3 al 7, se calcula $V_{m_{max}}$, donde en la línea 6 se utilizan las magnitudes de los vectores que van del origen a los puntos 35, 36, 45 y 46 (véase la Figura 2.9). Dentro de las líneas del 8 al 23, se calculan algunas otras variables que se utilizarán para calcular $V_{m_{min}}$; en las líneas 8 y 16, V_{1_m} es el intervalo de magnitud de V_1 y V_{2_m} el de V_2. Por último, en la línea 24, se calcula $V_{m_{min}}$.

Algoritmo 3 AngleCase2(V_1, V_2)

Require: $V_1 = [a, b]\angle[\alpha_1, \alpha_2]$ and $V_2 = [c, d]\angle[\alpha_3, \alpha_4]$
Ensure: $V_\alpha = [V_{\alpha_{min}}, V_{\alpha_{max}}]$
1: $V_{\alpha_{min}} = min(p25_\alpha, p27_\alpha)$
2: $V_{\alpha_{max}} = max(p18_\alpha, p38_\alpha)$
3: **if** $V_{\alpha_{max}} < 90$ **then**
4: $\gamma = \alpha_2 + \sin^{-1}(d/a)$
5: $\phi = \gamma + 90$
6: **if** $\phi \cap V_{2_\alpha}$ **then**
7: $V_{\alpha_{max}} = \gamma$
8: **else if** $\phi < \alpha_3$ **then**
9: $V_{\alpha_{max}} = p36_\alpha$
10: **else if** $\phi > \alpha_4$ **then**
11: $V_{\alpha_{max}} = p38_\alpha$
12: **end if**
13: **end if**
14: **if** $V_{\alpha_{min}} > 90$ **then**
15: $\gamma = \alpha_3 - \sin^{-1}(b/c)$
16: $\phi = \gamma - 90$
17: **if** $\phi \cap V_{1_\alpha}$ **then**
18: $V_{\alpha_{min}} = \phi$
19: **else if** $\phi < \alpha_1$ **then**
20: $V_{\alpha_{min}} = p25_\alpha$
21: **else if** $\phi > \alpha_2$ **then**
22: $V_{\alpha_{min}} = p45_\alpha$
23: **end if**
24: **end if**
25: **return** $([V_{\alpha_{min}}, V_{\alpha_{max}}])$

El Algoritmo 3 recibe dos abanicos complejos como parámetros, V_1 y V_2. El algoritmo regresa el intervalo de ángulo V_α resultante para la suma $V = V_1 + V_2$. En la línea 1, se calcula $V_{\alpha_{min}}$, usando los ángulos de los vectores que van del origen a los puntos 25 y 27 (véase la Figura 2.9). En la línea 2, se calcula $V_{\alpha_{max}}$, usando los ángulos de los vectores que van del origen a los puntos 18 y 38. Entre las líneas del 3 al 13, se calculan las variables γ y ϕ, y se ajusta el valor de $V_{\alpha_{max}}$ de acuerdo a estas; en la línea 6, V_{2_α} es el intervalo de ángulo de V_2. Finalmente, entre las líneas del 14 al 24, se vuelven a calcular las variables γ y ϕ, y se ajusta el valor de $V_{\alpha_{min}}$ de acuerdo a estas; en la línea 17, V_{1_α} es el intervalo de ángulo de V_1.

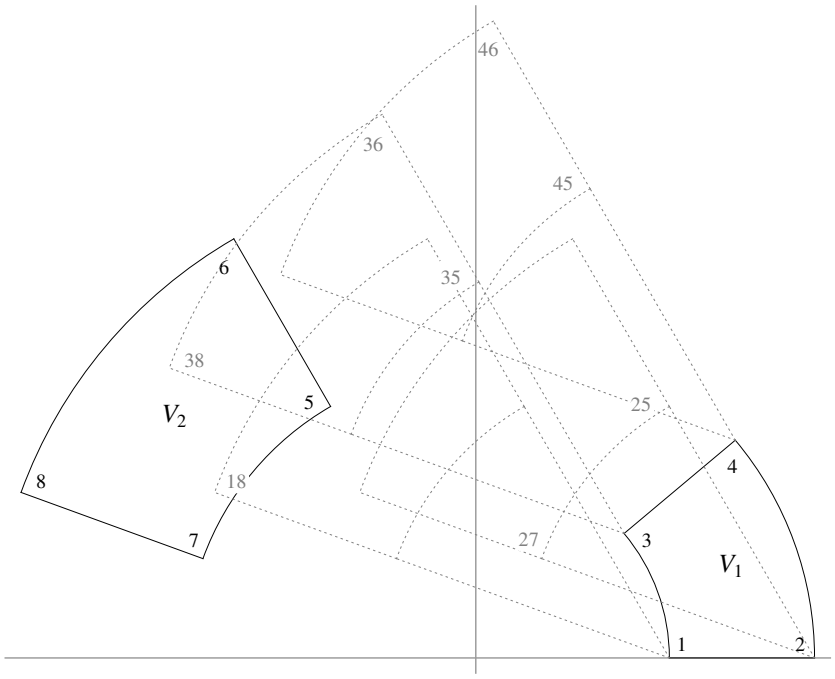


Figura 2.9: Numeración de esquinas, adición caso 2

Corrección al Algoritmo 3. En la línea 18, se cambió $V_{\alpha_{min}} = \phi$, por $V_{\alpha_{min}} = \gamma$, para ilustrar esta corrección al algoritmo, se muestra un ejemplo en la Figura 2.10, que utiliza esta corrección.

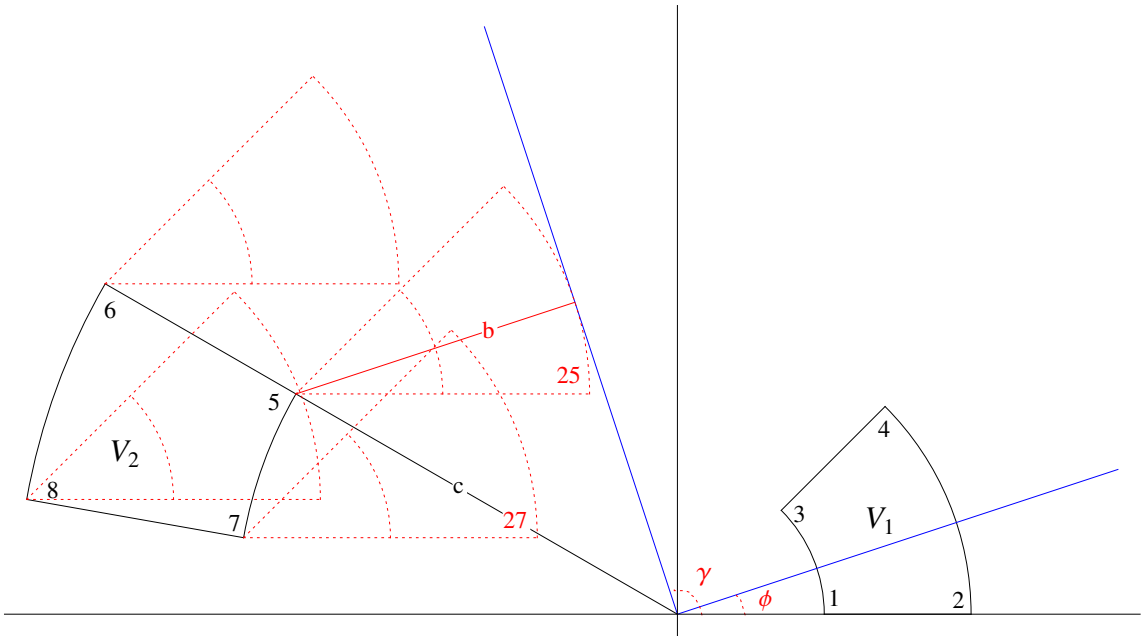


Figura 2.10: Ilustración para la corrección del Algoritmo 3

Al observar la Figura 2.10 y utilizando el Algoritmo 3, se observa que el ángulo del vector que va del origen al punto 25 es menor que el ángulo del vector que va del origen al punto

27, por lo tanto $V_{\alpha_{min}}$ toma como valor el ángulo del vector al punto 25. Posteriormente se cumple la cláusula $V_{\alpha_{min}} > 90$, por lo tanto se calculan γ y ϕ, los cuales se muestran en la figura. Después se cumple la cláusula $\phi \in V_{1_\alpha}$. Ya que el resultado cubrirá el área alrededor de donde están los abanicos complejos con líneas punteadas rojas, el valor correcto de $V_{\alpha_{min}}$ es γ y no ϕ. Para más información de porque esta deducción es correcta, redirijo al lector a [Flores, 1998], ya que allí se detalla la matemática usada para estos algoritmos.

Caso 3: Abanicos complejos en cuadrantes opuestos. Este caso supone que el abanico complejo V_1 está en el primer cuadrante y V_2 está en el tercero (véase la Figura 2.11). Para resolver este caso de adición, se usan dos algoritmos, uno para calcular el intervalo de magnitud (véase el Algoritmo 4) y el otro para calcular el intervalo de ángulo (véase el Algoritmo 5).

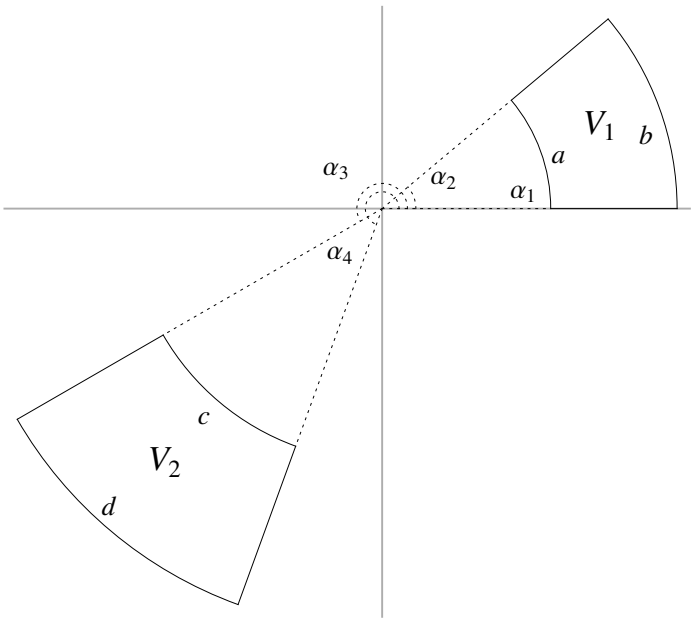


Figura 2.11: Adición caso 3

El Algoritmo 4 recibe dos abanicos complejos como parámetros, V_1 y V_2. El algoritmo regresa el intervalo de magnitud V_m resultante para la suma $V = V_1 + V_2$. En las líneas 1 y 2, se calculan los ángulos θ_1 y θ_2, los cuales servirán para calcular $V_{m_{max}}$. Entre las líneas del 3 al 7, se calcula $V_{m_{max}}$; en las líneas 4 y 6, se usan las magnitudes de los vectores que van del origen a los puntos 35, 36, 17, etc. (véase la Figura 2.12). Dentro de las líneas del 8 al 18, se calculan algunas otras variables que se utilizarán para calcular $V_{m_{min}}$; en la línea 8, V_{1_m} es el intervalo de magnitud de V_1 y V_{2_m} el de V_2. Por último, en la línea 19, se calcula $V_{m_{min}}$.

Observación para el Algoritmo 4. En la línea 19, en el cálculo de $V_{m_{min}}$ se usó el primer resultado, $\sqrt{x_m^2 + y_m^2 + 2x_m y_m}$, y no el resultado simplificado, $x_m - y_m$, ya que $\sqrt{a^2 + 2ab + b^2} = \sqrt{(a+b)^2} = (a+b)$, por lo tanto el resultado simplificado debería ser $x_m + y_m$.

Algoritmo 4 MagnitudeCase3(V_1, V_2)

Require: $V_1 = [a, b]\angle[\alpha_1, \alpha_2]$ and $V_2 = [c, d]\angle[\alpha_3, \alpha_4]$
Ensure: $V_m = [V_{m_{min}}, V_{m_{max}}]$
1: $\theta_1 = \alpha_3 - \alpha_2$
2: $\theta_2 = 360 - (\alpha_4 - \alpha_1)$
3: **if** $\theta_1 < \theta_2$ **then**
4: $\quad V_{m_{max}} = max(p35_m, p36_m, p45_m, p46_m)$
5: **else**
6: $\quad V_{m_{max}} = max(p17_m, p18_m, p27_m, p28_m)$
7: **end if**
8: $I = V_{1_m} \cap V_{2_m}$
9: **if** $I \neq \varnothing$ **then**
10: $\quad x_m = I_{min}$
11: $\quad y_m = I_{min}$
12: **else if** $a > d$ **then**
13: $\quad x_m = a$
14: $\quad y_m = d$
15: **else if** $b < c$ **then**
16: $\quad x_m = b$
17: $\quad y_m = c$
18: **end if**
19: $V_{m_{min}} = \sqrt{x_m^2 + y_m^2 + 2x_m y_m} = x_m - y_m$
20: **return** $([V_{m_{min}}, V_{m_{max}}])$

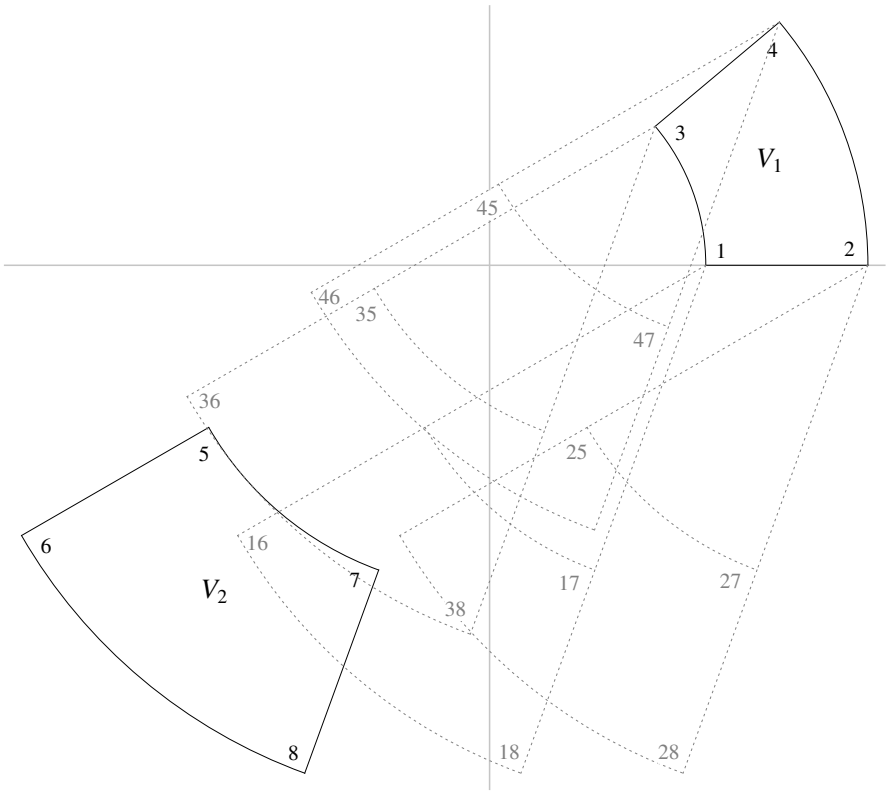


Figura 2.12: Numeración de esquinas, adición caso 3

Algoritmo 5 AngleCase3(V_1, V_2)

Require: $V_1 = [a, b]\angle[\alpha_1, \alpha_2]$ and $V_2 = [c, d]\angle[\alpha_3, \alpha_4]$
Ensure: $V_\alpha = [V_{\alpha_{min}}, V_{\alpha_{max}}]$
1: $\theta_{max} = 180$
2: MagDiff $= V_{1_m} - V_{2_m} \cos\theta_{max}$
3: **if** $0 \in$ MagDiff **then**
4: **return** $([0, 360])$
5: **else if** MagDiff > 0 **then**
6: $V_{\alpha_{max}} = p36_\alpha$
7: **if** $V_{\alpha_{max}} > 90$ **then**
8: $\gamma_1 = \sin^{-1}(d/a) + \alpha_2$
9: $\phi_1 = \gamma_1 + 90$
10: **if** $\phi_1 \in V_{2_\alpha}$ **then**
11: $V_{\alpha_{max}} = \gamma_1$
12: **else if** $\phi_1 < \alpha_3$ **then**
13: $V_{\alpha_{max}} = p36_\alpha$
14: **else if** $\phi_1 > \alpha_4$ **then**
15: $V_{\alpha_{max}} = p38_\alpha$
16: **end if**
17: **end if**
18: $V_{\alpha_{min}} = p18_\alpha$
19: **if** $V_{\alpha_{min}} > 270$ **then**
20: $\gamma_2 = 360 - \sin^{-1}(d/a)$
21: $\phi_2 = \gamma_2 - 90$
22: **if** $\phi_2 \in V_{2_\alpha}$ **then**
23: $V_{\alpha_{min}} = \gamma_2$
24: **else if** $\phi_2 < \alpha_3$ **then**
25: $V_{\alpha_{min}} = p16_\alpha$
26: **else if** $\phi_2 > \alpha_4$ **then**
27: $V_{\alpha_{min}} = p18_\alpha$
28: **end if**
29: **end if**
30: **else**
31: ... case MagDiff < 0 is symmetrical
32: **end if**
33: **return** $([V_{\alpha_{min}}, V_{\alpha_{max}}])$

El Algoritmo 5 recibe dos abanicos complejos como parámetros, V_1 y V_2. El algoritmo regresa el intervalo de ángulo V_α resultante para la suma $V = V_1 + V_2$. En la línea 1, se asigna el valor de 180 a θ_{max}. En la línea 2, se calcula la variable MagDiff. Posteriormente, en la línea 3, empieza una sentencia if ... else if ... else ..., y termina en la línea 32, el control de su flujo depende del valor de MagDiff; si se cumple la sentencia if de la línea 3, el algoritmo regresa un intervalo de ángulo que va de 0° a 360°; si se cumple la sentencia else if de la línea 5, entre las líneas del 6 al 17, se calcula $V_{\alpha_{max}}$ y se ajusta su valor si fuere necesario, y entre la 18 a la 29, se calcula $V_{\alpha_{min}}$ y se ajusta su valor si fuere necesario; si no se cumple ninguna de las sentencias anteriores, el algoritmo no especifica las operaciones a realizar en la sentencia else de la línea 30, para ello se dedujo la parte que falta, el resultado se muestra

en el Algoritmo 6. Para más información acerca de los desarrollos matemáticos usados para la deducción de estos algoritmos, se dirige al lector a [Flores, 1998].

Corrección para el Algoritmo 5. Se cambió la línea 2: MagDiff $= V_{1_m} - V_{2_m} \cos\theta_{max}$, por la siguiente línea: MagDiff $= V_{1_m} + V_{2_m} \cos\theta_{max}$.

Ya que $\cos\theta_{max} = \cos 180^\circ = -1$ y, $V_{2_m} \cos\theta_{max} = [c,d] * (-1) = [-d,-c]$ por lo tanto, $V_{1_m} - V_{2_m} \cos\theta_{max} = [a,b] - [-d,-c] = [a-(-c), b-(-d)] = [a+c, b+d]$, es decir, MagDiff $= [a+c, b+d]$. Para que MagDiff sea menor que 0, $a+c$ y $b+d$ deben ser menores que 0, pero como se explicará más adelante en la Sección 3.1.2, no se manejarán intervalos de magnitud con valores negativos, por lo tanto $a+c$ y $b+d$ siempre tomarían valores positivos, lo cual implicaría que nunca se cumpliría la condición MagDiff < 0 del algoritmo. Lo cual proporcionaría resultados incorrectos para algunos casos.

Ahora bien, si se usa la corrección: MagDiff $= V_{1_m} + V_{2_m} \cos\theta_{max} = [a,b] + [-d,-c] = [a+(-d), b+(-c)] = [a-d, b-c]$. En el caso de que $a-d < 0$ y $b-c < 0$, ahora si se podría entrar en la condición MagDiff < 0.

Algoritmo 6 Case MagDiff < 0 for AngleCase3

1: $V_{\alpha_{max}} = p27_\alpha$
2: **if** $V_{\alpha_{max}} > 270$ **then**
3: $\quad \gamma_1 = \sin^{-1}(b/c) + \alpha_4$
4: $\quad \phi_1 = \text{MODULO}_{360}(\gamma_1 + 90)$
5: $\quad$ **if** $\phi_1 \in V_{1_\alpha}$ **then**
6: $\quad\quad V_{\alpha_{max}} = \gamma_1$
7: $\quad$ **else if** $\phi_1 < \alpha_1$ **then**
8: $\quad\quad V_{\alpha_{max}} = p27_\alpha$
9: $\quad$ **else if** $\phi_1 > \alpha_2$ **then**
10: $\quad\quad V_{\alpha_{max}} = p47_\alpha$
11: $\quad$ **end if**
12: **end if**
13: $V_{\alpha_{min}} = p45_\alpha$
14: **if** $V_{\alpha_{min}} < 180$ **then**
15: $\quad \gamma_2 = \alpha_3 - \sin^{-1}(b/c)$
16: $\quad \phi_2 = \gamma_2 - 90$
17: $\quad$ **if** $\phi_2 \in V_{1_\alpha}$ **then**
18: $\quad\quad V_{\alpha_{min}} = \gamma_2$
19: $\quad$ **else if** $\phi_2 < \alpha_1$ **then**
20: $\quad\quad V_{\alpha_{min}} = p25_\alpha$
21: $\quad$ **else if** $\phi_2 > \alpha_2$ **then**
22: $\quad\quad V_{\alpha_{min}} = p45_\alpha$
23: $\quad$ **end if**
24: **end if**

Estos algoritmos fueron deducidos considerando todos los extremos de los abanicos complejos como cerrados. Si los abanicos complejos tienen extremos abiertos, se hace el cálculo como si fueran extremos cerrados. Así fué como se implementaron los algoritmos también. Para mejorar esto se puede tomar en consideración cuales son los puntos que producen el extremo resultante y luego verificar cuales de estos se tienen que incluir en el resultado final o no.

Los algoritmos anteriores garantizan completitud, es decir, todos los posibles resultados están incluidos en el resultado. Los algoritmos garantizan minimalidad en el sentido de que ya no hay un abanico complejo mas pequeño que engloba todos los resultados posibles de la adición. Pero aún cuando los resultados son los abanicos complejos mínimos, estos pueden contener resultados espurios (y los contienen la mayoría de las veces). Este problema es causado por la incertidumbre contenida en la representación, la cual se propaga a través de las operaciones aritméticas. Además es importante mencionar que la adición de abanicos complejos es conmutativa pero no es asociativa.

Para finalizar la unión de dos abanicos complejos se define de la siguiente manera:

$$V = V_1 \cup V_2 = [a, b] \cup [c, d] \angle [\alpha_1, \alpha_2] \cup [\alpha_3, \alpha_4] \tag{2.14}$$

2.6. Conclusiones

Se han presentado satisfactoriamente los algoritmos propuestos en [Flores, 1998] para la aritmética de abanicos complejos: Negación, Producto, División, Sustracción y Adición. El producto, la división y la negación de abanicos complejos, pueden ser expresados directamente en base a la aritmética de intervalos. El mayor problema radica en la adición de dos abanicos complejos, donde se necesita descomponer ambos operandos de acuerdo a sus intersecciones con el plano cartesiano y realizar las sumas parciales provenientes de las combinaciones de partes de cada operando. Las sumas parciales se dividen en tres casos de adición, dependiendo de la localización de los operandos en el plano cartesiano; el primer caso de adición es cuando los dos operandos se encuentran en el mismo cuadrante; el segundo caso de adición cuando el primero se encuentra en el primer cuadrante y el segundo en el segundo cuadrante; el tercer caso de adición cuando el primero se encuentra en el primer cuadrante y el segundo en el tercer cuadrante. Cualquier otro caso puede convertirse en alguno de los tres casos anteriores. Para obtener el resultado total se realiza la unión de los resultados parciales. La sustracción se define en base a la adición y la negación de abanicos complejos.

También se presentaron las correcciones de pequeños errores que se encontraron en los algoritmos.

Para que los algoritmos presentados puedan funcionar se requieren operaciones adicionales como la aritmética de intervalos, unión e intersección de intervalos, partición de abanicos complejos, unión de resultados parciales en la adición, una función general de adición que use los algoritmos de los tres casos de adición, etc.

Capítulo 3

Implementación

En este capítulo se presentan las implementaciones que se efectuaron en esta tesis. En la Sección 3.1 se presenta el diseño de objetos de acuerdo a los objetos localizados al analizar los algoritmos presentados en el capítulo anterior, junto con las consideraciones que se deben tomar en cuenta para la implementación de este sistema. Por las características del proyecto, ya que Java es orientada a objetos la traducción de los requerimientos será sencillo; pero como Mathematica no es orientado a objetos, sino orientado a la programación funcional, aprovecharemos este atributo de Mathematica para traducir nuestros requerimientos al diseño de patrones para representar cada uno de los diferentes objetos requeridos. En la Sección 3.2 se presenta la implementación hecha en Java, donde se describen brevemente las diferentes clases implementadas. En la Sección 3.3 se presenta una breve descripción de la implementación hecha en Mathematica.

3.1. Diseño de objetos

El diseño de objetos se efectuó de acuerdo a los objetos localizados al analizar los algoritmos presentados en el Capítulo 2. En Java los objetos se pueden crear al instanciar clases, pero Mathematica está orientado a la programación funcional y no orientado a objetos. Para solucionar esto, en Mathematica se pueden diseñar patrones para representar cada uno de los diferentes objetos que se presentarán.

3.1.1. Objeto Intervalo

Se necesita una clase que modele un Intervalo en Java, y en Mathematica se puede diseñar un patrón para representar un Intervalo. Este objeto tiene como atributos: el primer extremo, el segundo extremo y los límites de cada extremo; además se requerirían métodos para el manejo de intervalos como acceder a sus extremos y límites, asignar nuevos valores a sus extremos y límites, multiplicar un intervalo por una constante, verificar si un intervalo es vacío, unión de dos intervalos, intersección de dos intervalos, negación de un intervalo, adición de dos intervalos, producto entre dos intervalos, resta entre dos intervalos, división entre dos intervalos.

Un intervalo se considerará vacío cuando el valor de sus extremos sean iguales y los límites sean abiertos, o cuando el valor de sus extremos sean iguales con un límite abierto y el otro cerrado, he aquí algunos ejemplos:

$(0, 0)$ $\rightarrow$ Intervalo vacío
$(3, 3)$ $\rightarrow$ Intervalo vacío
$[6, 6)$ $\rightarrow$ Intervalo vacío
$(9, 9]$ $\rightarrow$ Intervalo vacío
$[12, 12]$ $\rightarrow$ Intervalo NO vacío

3.1.2. Objeto Intervalo de Magnitud

Ya que un Intervalo de Magnitud es un tipo de Intervalo, se puede modelar con la misma clase (en Java) que se use para modelar un Intervalo, o usando herencia. Es decir, que la clase Intervalo de Magnitud herede de la clase Intervalo, tomando en cuenta que las diferencias entre un intervalo y un intervalo de magnitud son las siguientes consideraciones.

Los algoritmos presentados en el Capítulo 2 trabajan con intervalos de magnitud con valores positivos para sus extremos, a continuación se muestran ejemplos de lo que esperarían los algoritmos y lo que no esperarían:

$(2, 9)$ $\rightarrow$ correcto
$[3, 11]$ $\rightarrow$ correcto
$(-2, -10)$ $\rightarrow$ incorrecto
$[-4, -12]$ $\rightarrow$ incorrecto

Además los algoritmos recibirían intervalos de magnitud normalizados, es decir, el valor del primer extremo debe ser menor o igual al valor del segundo extremo, a continuación algunos ejemplos:

$(8, 9)$ $\rightarrow$ correcto
$[13, 21]$ $\rightarrow$ correcto
$(22, 1)$ $\rightarrow$ incorrecto
$[24, 17]$ $\rightarrow$ incorrecto

3.1.3. Objeto Intervalo de Ángulo

Un Intervalo de Ángulo también es un tipo de Intervalo, se le puede modelar con una clase (en Java) que herede de la clase Intervalo, ya que se requiere tomar en cuenta las siguientes consideraciones y métodos adicionales que conciernen a los intervalos de ángulo.

Los algoritmos recibirán intervalos de ángulo normalizados, es decir, que el valor de sus extremos estén dentro del intervalo de [0,360]. Algunos ejemplos de lo que esperarían y lo que no esperarían se presentan a continuación:

$[90, 240]$ $\rightarrow$ correcto
$[300, 90]$ $\rightarrow$ correcto
$[450, 600]$ $\rightarrow$ incorrecto
$[-60, -270]$ $\rightarrow$ incorrecto

En el caso de los intervalos de ángulo solo se ocuparán la suma y la resta, por lo tanto se requerirá normalizar los resultados de la suma y la resta de dos intervalos de ángulo, es decir, que los resultados estén dentro del intervalo de [0, 360], por ejemplo:

$$\begin{aligned}
[90, 190] + [180, 180] &= [270, 10] && \rightarrow \text{correcto} \\
[35, 125] + [200, 315] &= [235, 80] && \rightarrow \text{correcto} \\
[90, 190] + [180, 180] &= [270, 370] && \rightarrow \text{incorrecto} \\
[35, 125] + [200, 315] &= [235, 440] && \rightarrow \text{incorrecto}
\end{aligned}$$

Un intervalo de ángulo, por ejemplo [15, 45], puede tener dos interpretaciones, el ángulo que va de 15 a 45 y el ángulo que va de 45 a 15 pasando por 0. Para este trabajo siempre se representarán los extremos en sentido contrario a las manecillas del reloj, por lo tanto en el ejemplo antes mencionado solo la primera interpretación es válida.

Además se necesitará una función para verificar si un intervalo de ángulo abarca los 360 grados, otra función para sumar 180 grados a un intervalo de ángulo, etc.

3.1.4. Objeto Abanico Complejo

Se requiere una clase (en Java) para modelar un objeto Abanico Complejo, que tiene como atributos un intervalo de magnitud y un intervalo de ángulo. Estos objetos deben contener métodos y funciones para su manejo, como asignar nuevos valores a sus atributos, obtener los valores de sus atributos y operaciones aritméticas. Dentro de la adición de dos abanicos complejos, se necesita implementar los algoritmos para los diferentes casos de sumas parciales, unión de resultados parciales, partir un abanico complejo, etc.

3.2. Implementación en Java

A continuación se presenta un resumen de las clases que se implementaron en Java para cubrir los requerimientos de la Sección 3.1. Para información acerca de los atributos, métodos, funciones, subclases, clases padre, constructores, etc., de cada clase se dirige al lector al Apéndice A.

3.2.1. Clase Interval

Esta clase se ha implementado para modelar un intervalo, implementa atributos, métodos y funciones para su manejo. Cabe mencionar que esta clase también sirve para modelar un intervalo de magnitud; se deja la obligación al usuario de ingresar intervalos de magnitud con valores positivos, no se promete al usuario que los resultados sean válidos si ingresa valores negativos; además esta clase implementa el método para normalizar un intervalo de magnitud como se especifica en la Sección 3.1.2.

3.2.2. Clase AngleInterval

Clase para modelar un intervalo de ángulo, implementa los métodos y las funciones para su manejo, hereda de la clase Interval. También dentro de esta clase se implementó el método para normalizar los intervalos de ángulo como se especifica en la Sección 3.1.3.

3.2.3. Clase ComplexFan

Esta clase modela un abanico complejo, implementa atributos, métodos y funciones para su manejo. En esta clase se implementan los algoritmos para la aritmética de abanicos complejos.

3.3. Implementación en Mathematica

Mathematica está basado en el lenguaje Wolfram. El lenguaje Wolfram es un lenguaje basado en conocimiento altamente desarrollado que unifica un gran rango de paradigmas de programación y usa su concepto único de programación simbólica para añadir un nuevo nivel de flexibilidad a la propia idea de la programación [Wolfram, 2016b].

Ya que Mathematica no es un lenguaje de programación orientado a objetos, se pueden utilizar varias opciones para traducir los requerimientos de la Sección 3.1 a este lenguaje, una de ellas es utilizar la facilidad J/Link, que permite cargar clases Java arbitrarias en el lenguaje Wolfram, entonces crear objetos, llamar métodos y acceder a atributos directamente desde el lenguaje Wolfram [Wolfram, 2016a]. Otra manera es usando paquetes que emulan la programación orientada a objetos, es decir, implementan la definición de clases, atributos, métodos y funciones, incluso herencia, paquetes hechos en Mathematica desarrollados por terceros [Exchange, 2016]. La última manera que se menciona en esta tesis, es la del diseño de patrones para emular la programación orientada a objetos, se puede consultar este tema a más profundidad en el articulo [Antonov, 2015].

Para este proyecto no se investigaron las dos primeras opciones de solución a profundidad, se usó la última opción, que es la del diseño de patrones para emular objetos, se usó esta opción por que es la única que se conocía al momento de la implementación, por lo tanto no se sabe si fue la opción más viable para este proyecto.

En resumen, para la implementación en Mathematica, se diseñaron y se implementaron patrones para representar los objetos requeridos; un patrón para representar los intervalos de magnitud y los intervalos de ángulo, `IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]`, y otro patrón para representar los abanicos complejos, `CF[mi_,ai_]`. Además se implementaron los métodos y las funciones requeridas para trabajar sobre los patrones; los métodos y las funciones que empiezan con las letras “IN” están diseñados para trabajar sobre intervalos de magnitud, los que empiezan con “AI” sobre intervalos de ángulo, y los que empiezan con “CF” sobre abanicos complejos. Para un resumen completo de los patrones, métodos y funciones implementadas se dirige al lector al Apéndice B.

3.4. Conclusiones

El diseño de objetos presentado en la Sección 3.1 se realizó en base a los algoritmos presentados en el Capítulo 2. Los objetos presentados en esta sección se modelaron usando clases en Java y usando patrones en Mathematica.

La implementación en Java consta de tres archivos “.java”. Interval.java que contiene la clase Interval, esta clase sirve para modelar un Intervalo de magnitud. AngleInterval.java que contiene la clase AngleInterval, esta clase sirve para modelar un intervalo de ángulo. ComplexFan.java que contiene la clase ComplexFan, esta clase sirve para modelar un abanico complejo.

La implementación en Mathematica consta de un archivo “.m”, ComplexFans.m, biblioteca que contiene todos los métodos y las funciones requeridas para usar abanicos complejos. Se implementaron dos patrones, el primero para modelar intervalos de magnitud y ángulo, y

el segundo para modelar un abanico complejo. Las primeras dos letras de los nombres de cada método o función indican para que objeto fueron diseñados, ya sea para un intervalo de magnitud, intervalo de ángulo ó abanico complejo.

Capítulo 4

Pruebas y Resultados

En este capítulo se presentan las pruebas realizadas a las implementaciones y los resultados obtenidos en las pruebas. En la Sección 4.1 se presentan las pruebas elementales realizadas a las implementaciones y los resultados de las pruebas. Estas pruebas son pruebas básicas con ejemplos arbitrarios para cada operación aritmética implementada. En la Sección 4.2 se presentarán las pruebas realizadas sobre aplicaciones físicas y los resultados de las pruebas.

Las pruebas se realizaron en una Laptop HP con el sistema operativo Ubuntu 14.04.4 LTS (Trusty Tahr), Mathematica 9 y Java versión 1.7.0_101.

4.1. Pruebas elementales

En esta sección se presenta una prueba básica por cada operación aritmética disponible, con excepción a la Adición en la que se presentan tres pruebas (por los tres casos de adición). Estas pruebas son de ejemplos arbitrarios y se realizan tanto en Java como en Mathematica. Además se presentan los resultados de las pruebas, las cuales se verificaron y se comprobó la validez de los algoritmos correspondientes.

4.1.1. Negación

Si se tiene el abanico complejo $V_0 = [3, 9]\angle[90, 180]$, se puede calcular la negación de V_0 de la siguiente manera.

Cálculo de la negación de V_0 en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Negacion-----------");
ComplexFan V0 = new ComplexFan(new Interval('[',3,9,']'), new
   AngleInterval('[',90,180,']'));
V0.print();
System.out.println("-------Resultado-------");
ComplexFan V0n = ComplexFan.negation(V0);
V0n.print();
...
```

Al compilar el archivo `PruebasComplexFan.java` y ejecutar el archivo `PruebasComplexFan`, se obtiene el siguiente resultado:

```
$ javac PruebasComplexFan.java
$ java PruebasComplexFan
-----------Prueba Negacion-----------
[3.0,9.0] ∠ [90.0,180.0]
-------Resultado-------
[3.0,9.0] ∠ [270.0,360.0]
```

Cálculo de la negación de V_0 en Mathematica. Primero se carga la biblioteca "ComplexFans.m" en una Notebook de Mathematica. Esta biblioteca es donde se encuentran las funciones que se implementaron para la aritmética de abanicos complejos:

```
In[1]:=  << (NotebookDirectory[] <> ``ComplexFans.m'')
```

Una vez cargada la biblioteca de funciones ya se pueden usar todas la funciones contenidas en esta.

A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo para calcular la negación de V_0:

```
In[2]:=  V0 = CFDefineShort[MI["[", 3, 9, "]"], AI["[", 90, 180, "]"]];

In[3]:=  CFPrint[V0]
Out[3]:= [3,9] ∠ [90,180]

In[4]:=  V0n = CFNegation[V0];

In[5]:=  CFPrint[V0n]
Out[5]:= [3,9] ∠ [270,360]

In[6]:=  CFNegationPlot[V0, V0n]
```

La instrucción número 6 de la lista anterior produce la gráfica de la Figura 4.1. En esta figura se puede observar a V_0 de color azul y a V_{0n} de rojo.

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{0n} = [3, 9]\angle[270, 360]$. Para comprobar este resultado se usa la Ecuación 2.5 tomando en cuenta que $V_0 = [3, 9]\angle[90, 180]$,

$$V_{0n} = -V_0 = [3, 9]\angle([90, 180] + [180, 180])$$

Para simplificar se usa la ecuación para la adición de intervalos de la Tabla 2.1,

$$V_{0n} = [3, 9]\angle[90 + 180, 180 + 180] = [3, 9]\angle[270, 360]$$

Al comparar ambos resultados podemos observar que los resultados son los mismos, por lo tanto los resultados obtenidos en Java y en Mathematica son correctos.

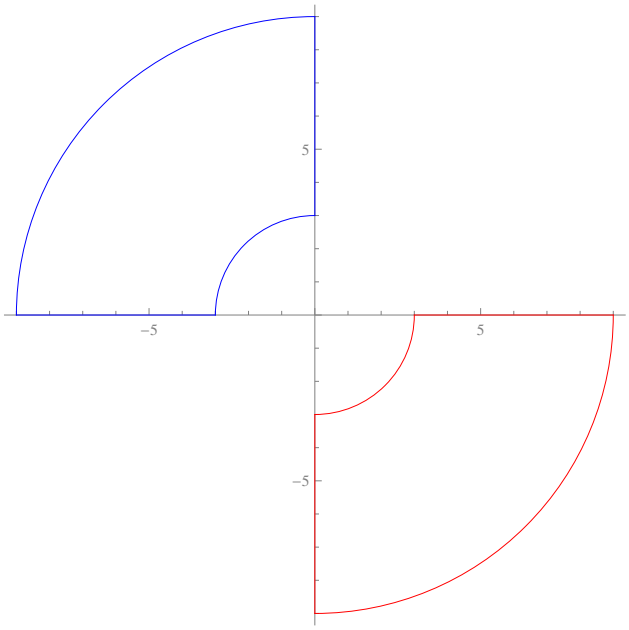


Figura 4.1: Negación de un abanico complejo

4.1.2. Producto

Si se tienen los abanicos complejos $V_1 = [4, 11]\angle[90, 180]$ y $V_2 = [7, 15]\angle[270, 300]$, se puede calcular el producto entre estos de la siguiente manera.

Cálculo de $V_{R1} = V_1 * V_2$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Producto-----------");
ComplexFan V1 = new ComplexFan(new Interval('[',4,11,']'), new
    AngleInterval('[',90,180,']'));
ComplexFan V2 = new ComplexFan(new Interval('[',7,15,']'), new
    AngleInterval('[',270,300,']'));
V1.print();
V2.print();
System.out.println("-------Resultado-------");
ComplexFan VR1 = ComplexFan.product(V1,V2);
VR1.print();
...
```

Después de compilar el archivo `PruebasComplexFan.java` y ejecutar el archivo `PruebasComplexFan`, se obtiene el resultado:

```
$ java PruebasComplexFan
-----------Prueba Producto-----------
[4.0,11.0] ∠ [90.0,180.0]
[7.0,15.0] ∠ [270.0,300.0]
-------Resultado-------
[28.0,165.0] ∠ [0.0,120.0]
```

Cálculo de $V_{R1} = V_1 * V_2$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[7]:=  V1 = CFDefineShort[MI["[", 4, 11, "]"], AI["[", 90, 180, "]"]];

In[8]:=  CFPrint[V1]
Out[8]:= [4,11] ∠ [90,180]

In[9]:=  V2 = CFDefineShort[MI["[", 7, 15, "]"], AI["[", 270, 300, "]"]];

In[10]:=  CFPrint[V2]
Out[10]:= [7,15] ∠ [270,300]

In[11]:=  VR1 = CFProduct[V1, V2];

In[12]:=  CFPrint[VR1]
Out[12]:= [28,165] ∠ [0,120]

Out[13]:= CFOperationPlot[V1, V2, VR1]
```

La instrucción número 13 de la lista anterior produce la gráfica de la Figura 4.2, donde se puede observar a V_1 de color azul, a V_2 de verde y a V_{R1} de rojo.

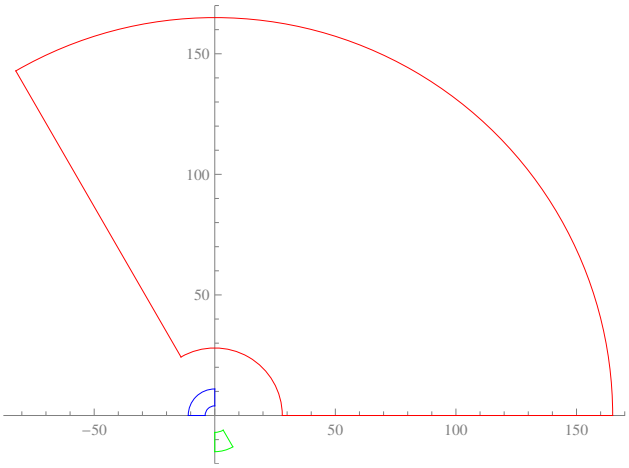


Figura 4.2: Producto entre dos abanicos complejos

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R1} = [28, 165]\angle[0, 120]$. Para comprobar este resultado se usa la Ecuación 2.3 tomando en cuenta que $V_1 = [4, 11]\angle[90, 180]$ y $V_2 = [7, 15]\angle[270, 300]$,

$$V_{R1} = V_1 * V_2 = ([4, 11] * [7, 15])\angle([90, 180] + [270, 300])$$

Para simplificar se usa la aritmética de intervalos definida en la Tabla 2.1,

$$V_{R1} = [4 * 7, 11 * 15]\angle[90 + 270, 180 + 300] = [28, 165]\angle[360, 480]$$

Resultado normalizado,

$$V_{R1} = [28, 165]\angle[0, 120]$$

Al comparar ambos resultados podemos observar que los resultados son los mismos, por lo tanto los resultados obtenidos en Java y en Mathematica son correctos.

4.1.3. División

Si se tienen los abanicos complejos $V_3 = [2,7]\angle[35,78]$ y $V_4 = [1,5]\angle[289,356]$, se puede calcular la división entre estos de la siguiente manera.

Cálculo de $V_{R2} = V_3/V_4$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Division-----------");
ComplexFan V3 = new ComplexFan(new Interval('[',2,7,']'), new
   AngleInterval('[',35,78,']'));
ComplexFan V4 = new ComplexFan(new Interval('[',1,5,']'), new
   AngleInterval('[',289,356,']'));
V3.print();
V4.print();
System.out.println("-------Resultado-------");
ComplexFan VR2 = ComplexFan.division(V3,V4);
VR2.print();
...
```

Después de compilar el archivo `PruebasComplexFan.java` y ejecutar el archivo `PruebasComplexFan`, se obtiene el resultado:

```
-----------Prueba Division-----------
[2.0,7.0] ∠ [35.0,78.0]
[1.0,5.0] ∠ [289.0,356.0]
-------Resultado-------
[0.4,7.0] ∠ [39.0,149.0]
```

Cálculo de $V_{R2} = V_3/V_4$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[14]:=  V3 = CFDefineShort[MI["[", 2, 7, "]"], AI["[", 35, 78, "]"]];

In[15]:=  CFPrint[V3]
Out[15]:= [2,7] ∠ [35,78]

In[16]:=  V4 = CFDefineShort[MI["[", 1, 5, "]"], AI["[", 289, 356, "]"]];

In[17]:=  CFPrint[V4]
Out[17]:= [1,5] ∠ [289,356]

In[18]:=  VR2 = CFDivision[V3, V4];

In[19]:=  CFPrint[VR2]
Out[19]:= [2/5,7] ∠ [39,149]

In[20]:=  CFOperationPlot[V3, V4, VR2]
```

La instrucción número 20 de la lista anterior produce la gráfica de la Figura 4.3, donde se puede observar a V_3 de color azul, a V_4 de verde y a V_{R2} de rojo.

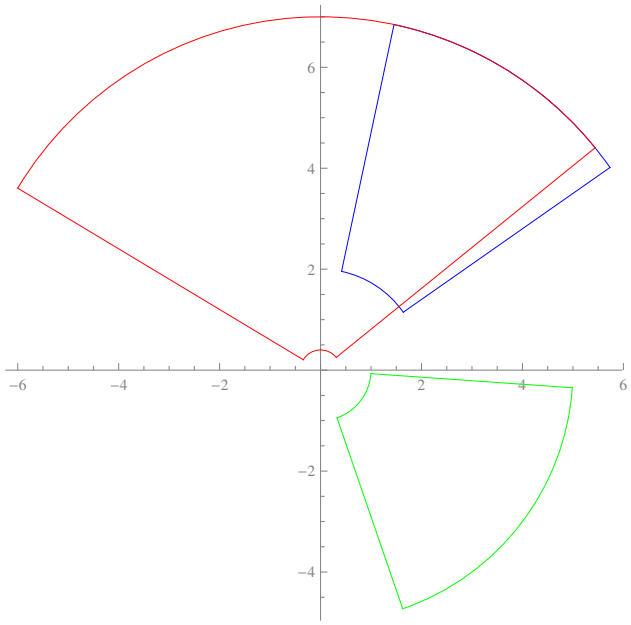


Figura 4.3: División entre dos abanicos complejos

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R2} = [0.4, 7]\angle[39, 149]$. Para comprobar este resultado se usa la Ecuación 2.4 tomando en cuenta que $V_3 = [2, 7]\angle[35, 78]$ y $V_4 = [1, 5]\angle[289, 356]$,

$$V_{R2} = V_3/V_4 = ([2, 7]/[1, 5])\angle([35, 78] - [289, 356])$$

Para simplificar se usa la aritmética de intervalos definida en la Tabla 2.1,

$$V_{R2} = [2/5, 7/1]\angle[35 - 356, 78 - 289] = [0.4, 7]\angle[-321, -211]$$

Resultado normalizado,

$$V_{R2} = [0.4, 7]\angle[39, 149]$$

Al comparar ambos resultados podemos observar que los resultados son los mismos, por lo tanto los resultados obtenidos en Java y en Mathematica son correctos.

4.1.4. Adición

Para la adición de dos abanicos complejos se presenta una prueba por cada caso de adición, en total tres pruebas por los tres casos de adición.

Adición caso 1. Si se tienen los abanicos complejos $V_5 = [14, 19]\angle[0, 44]$ y $V_6 = [11, 17]\angle[11, 55]$, se puede calcular la adición de estos. Ambos operandos se encuentran en el primer cuadrante, por lo tanto esta prueba cae en el caso 1.

Cálculo de $V_{R3} = V_5 + V_6$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Adicion 1-----------");
ComplexFan V5 = new ComplexFan(new Interval('[',14,19,']'), new
   AngleInterval('[',0,44,']'));
ComplexFan V6 = new ComplexFan(new Interval('[',11,17,']'), new
   AngleInterval('[',11,55,']'));
V5.print();
V6.print();
System.out.println("-------Resultado-------");
ComplexFan VR3 = ComplexFan.addition(V5,V6);
VR3.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
-----------Prueba Adicion 1-----------
[14.0,19.0] ∠ [0.0,44.0]
[11.0,17.0] ∠ [11.0,55.0]
-------Resultado-------
[22.21849550253397,36.0] ∠ [4.029134909709915,50.03388352288333]
```

Cálculo de $V_{R3} = V_5 + V_6$ en Mathematica A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[21]:=  V5 = CFDefineShort[MI["[", 14, 19, "]"], AI["[", 0, 44, "]"]];

In[22]:=  CFPrint[V5]
Out[22]:= [14,19] ∠ [0,44]

In[23]:=  V6 = CFDefineShort[MI["[", 11, 17, "]"], AI["[", 11, 55, "]"]];

In[24]:=  CFPrint[V6]
Out[24]:= [11,17] ∠ [11,55]

In[25]:=  VR3 = CFAddition[V5, V6];

In[26]:=  CFPrint[VR3]
Out[26]:= [22.2185,36.] ∠ [4.02913,50.0339]

In[27]:=  CFOperationPlot[V5, V6, VR3]
```

La instrucción número 27 de la lista anterior produce la gráfica de la Figura 4.4, donde se puede observar a V_5 de color azul, a V_6 de verde y a V_{R3} de rojo.

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R3} = [22.2185, 36.]\angle[4.02913, 50.0339]$. Ya que la adición de abanicos complejos no es tan sencilla, para comprobar el resultado se usa la función `CFAdditionPlot` de Mathematica, implementada para este propósito,

```
In[28]:=  CFAdditionPlot[V5, V6, VR3]
```

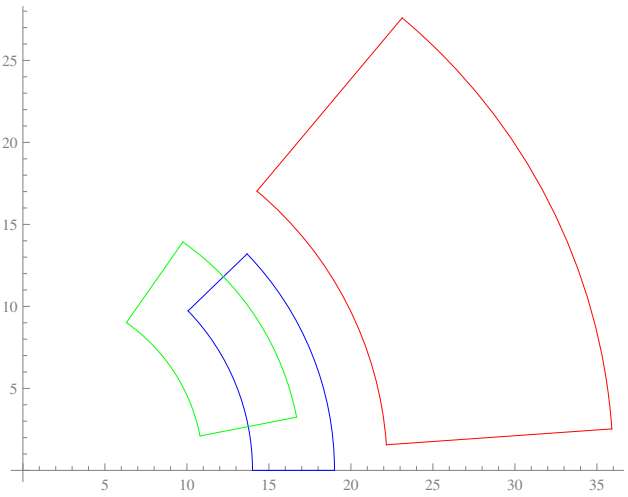
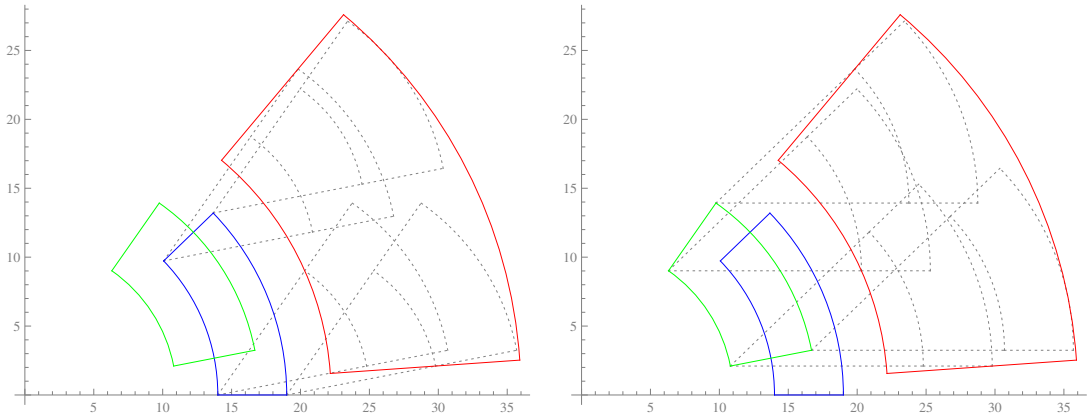


Figura 4.4: Adición caso 1 de dos abanicos complejos

La instrucción anterior produce las subfiguras de la Figura 4.5. Donde se puede observar que el resultado obtenido tanto en Java como en Mathematica es correcto, ya que es el abanico complejo mínimo que cubre todos los abanicos complejos con líneas punteadas grises.



a) Proyección de V_6 con respecto a V_5 b) Proyección de V_5 con respecto a V_6

Figura 4.5: Gráfica de V_{R3} con respecto a proyecciones

Adición caso 2. Si se tienen los abanicos complejos $V_7 = [14, 19]\angle[0, 44]$ y $V_8 = [11, 17]\angle[101, 145]$, se puede calcular la adición de estos. Ya que V_7 se encuentra en el primer cuadrante y V_8 en el segundo, esta prueba cae en el caso 2.

Cálculo de $V_{R4} = V_7 + V_8$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Addition 2-----------");
ComplexFan V7 = new ComplexFan(new Interval('[',14,19,']'), new
    AngleInterval('[',0,44,']'));
```

```
ComplexFan V8 = new ComplexFan(new Interval('[',11,17,']'), new
    AngleInterval('[',101,145,']'));
V7.print();
V8.print();
System.out.println("-------Resultado-------");
ComplexFan VR4 = ComplexFan.addition(V7,V8);
VR4.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
-----------Prueba Addition 2-----------
[14.0,19.0] ∠ [0.0,44.0]
[11.0,17.0] ∠ [101.0,145.0]
-------Resultado-------
[8.030070108914645,31.651805898237587] ∠ [32.27684091136668,101.19567552989061]
```

Cálculo de $V_{R4} = V_7 + V_8$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[29]:=  V7 = CFDefineShort[MI["[", 14, 19, "]"], AI["[", 0, 44, "]"]];

In[30]:=  CFPrint[V7]
Out[30]:= [14,19] ∠ [0,44]

In[31]:=  V8 = CFDefineShort[MI["[", 11, 17, "]"], AI["[", 101, 145, "]"]];

In[32]:=  CFPrint[V8]
Out[32]:= [11,17] ∠ [101,145]

In[33]:=  VR4 = CFAddition[V7, V8];

In[34]:=  CFPrint[VR4]
Out[34]:= [8.03007,31.6518] ∠ [32.2768,101.196]

In[35]:=  CFOperationPlot[V7, V8, VR4]
```

La instrucción número 35 de la lista anterior produce la gráfica de la Figura 4.6, donde se puede observar a V_7 de color azul, a V_8 de verde y a V_{R4} de rojo.

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R4} = [8.03007, 31.6518] \angle [32.2768, 101.196]$. Para comprobar el resultado se usa la función `CFAdditionPlot` de Mathematica, implementada para este propósito,

```
In[36]:=  CFAdditionPlot[V7, V8, VR4]
```

La instrucción anterior produce las subfiguras de la Figura 4.7. Donde se puede observar que el resultado obtenido tanto en Java como en Mathematica es correcto, ya que es el abanico complejo mínimo que cubre todos los abanicos complejos con líneas punteadas grises.

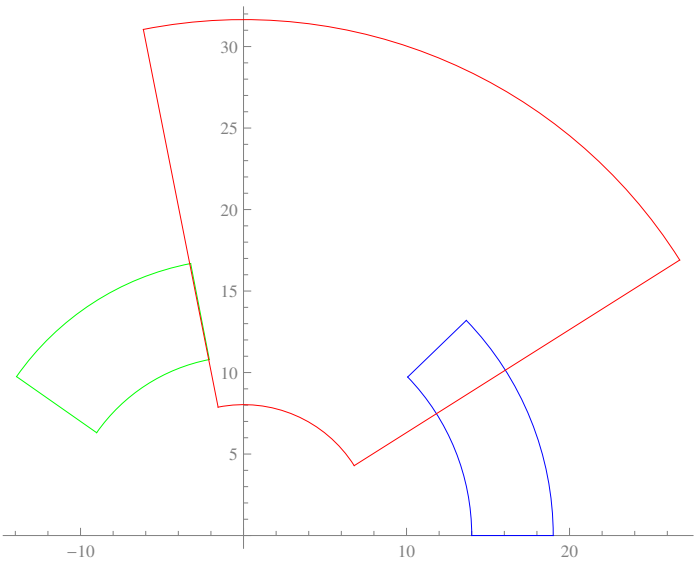
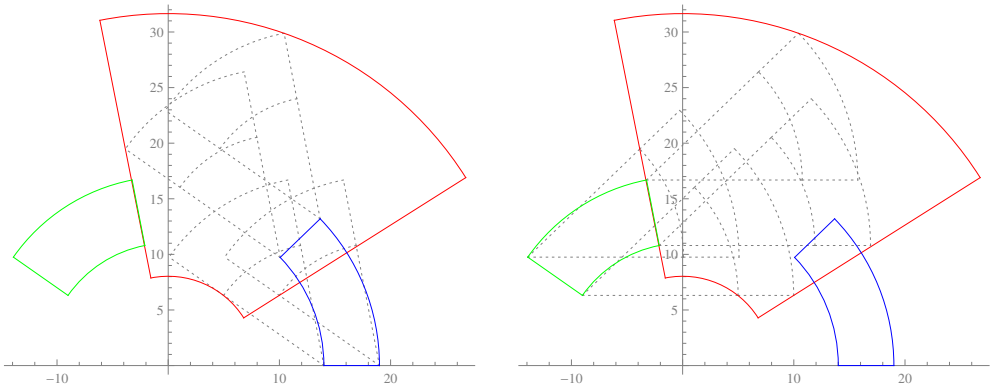


Figura 4.6: Adición caso 2 de dos abanicos complejos



a) Proyección de V_8 con respecto a V_7 b) Proyección de V_7 con respecto a V_8

Figura 4.7: Gráfica de V_{R4} con respecto a proyecciones

Adición caso 3. Si se tienen los abanicos complejos $V_9 = [14, 19]\angle[0, 44]$ y $V_{10} = [11, 17]\angle[191, 235]$, se puede calcular la adición de estos. Ya que V_9 se encuentra en el primer cuadrante y V_{10} en el tercero, esta prueba cae en el caso 3.

Cálculo de $V_{R5} = V_9 + V_{10}$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Addition 3-----------");
ComplexFan V9 = new ComplexFan(new Interval('[',14,19,']'), new
    AngleInterval('[',0,44,']'));
ComplexFan V10 = new ComplexFan(new Interval('[',11,17,']'), new
    AngleInterval('[',191,235,']'));
V9.print();
V10.print();
System.out.println("-------Resultado-------");
```

```
ComplexFan VR5 = ComplexFan.addition(V9,V10);
VR5.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
----------Prueba Addition 3----------
[14.0,19.0] ∠ [0.0,44.0]
[11.0,17.0] ∠ [191.0,235.0]
-------Resultado-------
[0.0,16.71734494820347] ∠ [0.0,360.0]
```

Cálculo de $V_{R5} = V_9 + V_{10}$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[37]:=  V9 = CFDefineShort[MI["[", 14, 19, "]"], AI["[", 0, 44, "]"]];

In[38]:=  CFPrint[V9]
Out[38]:= [14,19] ∠ [0,44]

In[39]:=  V10 = CFDefineShort[MI["[", 11, 17, "]"], AI["[", 191, 235, "]"]];

In[40]:=  CFPrint[V10]
Out[40]:= [11,17] ∠ [191,235]

In[41]:=  VR5 = CFAddition[V9, V10];

In[42]:=  CFPrint[VR5]
Out[42]:= [0,16.7173] ∠ [0,360]

In[43]:=  CFOperationPlot[V9, V10, VR5]
```

La instrucción número 43 de la lista anterior produce la gráfica de la Figura 4.8, donde se puede observar a V_9 de color azul, a V_{10} de verde y a V_{R5} de rojo.

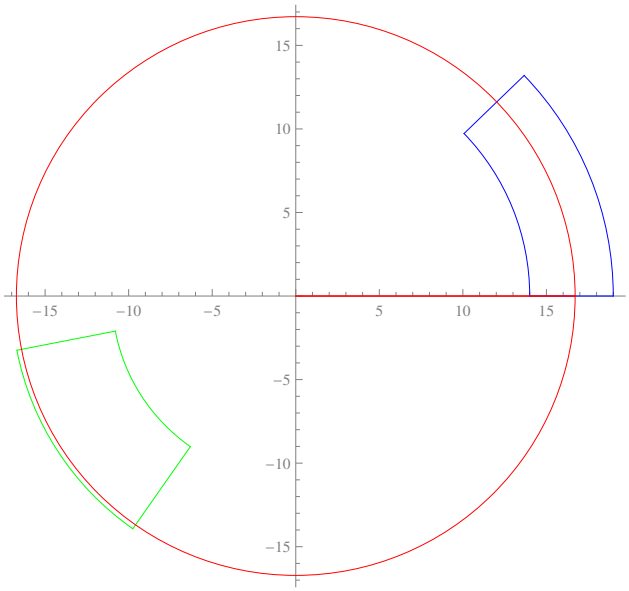
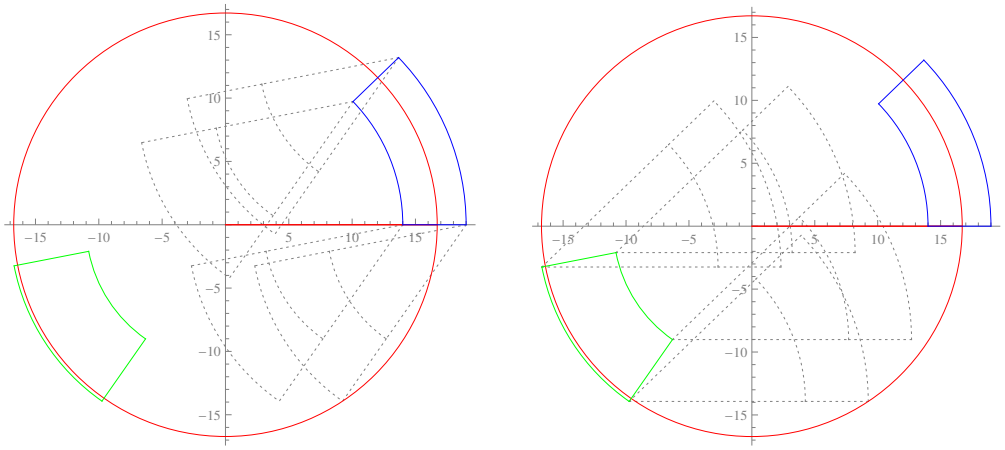


Figura 4.8: Adición caso 3 de dos abanicos complejos

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R5} = [0, 16.7173]\angle[0, 360]$. Para comprobar este resultado se usa la función `CFAdditionPlot` de Mathematica,

```
In[44]:=  CFAdditionPlot[V9, V10, VR5]
```

La instrucción anterior produce las subfiguras de la Figura 4.9. Donde se puede observar que el resultado obtenido tanto en Java como en Mathematica es correcto, ya que es el abanico complejo mínimo que cubre todos los abanicos complejos con líneas punteadas grises.



a) Proyección de V_{10} con respecto a V_9 b) Proyección de V_9 con respecto a V_{10}

Figura 4.9: Gráfica de V_{R5} con respecto a proyecciones

4.1.5. Sustracción

Si se tienen los abanicos complejos $V_{11} = [34, 40]\angle[123, 213]$ y $V_{12} = [24, 45]\angle[13, 99]$, se puede realizar la sustracción entre estos de la siguiente manera.

Cálculo de $V_{R6} = V_{11} - V_{12}$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Prueba Sustraccion----------");
ComplexFan V11 = new ComplexFan(new Interval('[',34,40,']'), new
    AngleInterval('[',123,213,']'));
ComplexFan V12 = new ComplexFan(new Interval('[',24,45,']'), new
    AngleInterval('[',13,99,']'));
V11.print();
V12.print();
System.out.println("-------Resultado-------");
ComplexFan VR6 = ComplexFan.subtraction(V11,V12);
VR6.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
----------Prueba Sustraccion----------
[34.0,40.0] ∠ [123.0,213.0]
[24.0,45.0] ∠ [13.0,99.0]
-------Resultado-------
[13.829045864577196,85.0] ∠ [148.0708700885612,251.16685288511707]
```

Cálculo de $V_{R6} = V_{11} - V_{12}$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[45]:=  V11 = CFDefineShort[MI["[", 34, 40, "]"], AI["[", 123, 213, "]"]];

In[46]:=  CFPrint[V11]
Out[46]:= [34,40] ∠ [123,213]

In[47]:=  V12 = CFDefineShort[MI["[", 24, 45, "]"], AI["[", 13, 99, "]"]];

In[48]:=  CFPrint[V12]
Out[48]:= [24,45] ∠ [13,99]

In[49]:=  VR6 = CFSubtraction[V11, V12];
Out[49]:= Less::nord: Invalid comparison with 200. -62.9458 i attempted. >>
Out[49]:= LessEqual::nord: Invalid comparison with 200. -62.9458 i attempted. >>

In[50]:=  CFPrint[VR6]
Out[50]:= [13.829,85.] ∠ [148.071,251.167]

In[51]:=  CFOperationPlot[V11, V12, VR6]
```

La instrucción número 51 de la lista anterior produce la gráfica de la Figura 4.10, donde se puede observar a V_{11} de color azul, a V_{12} de verde y a V_{R6} de rojo.

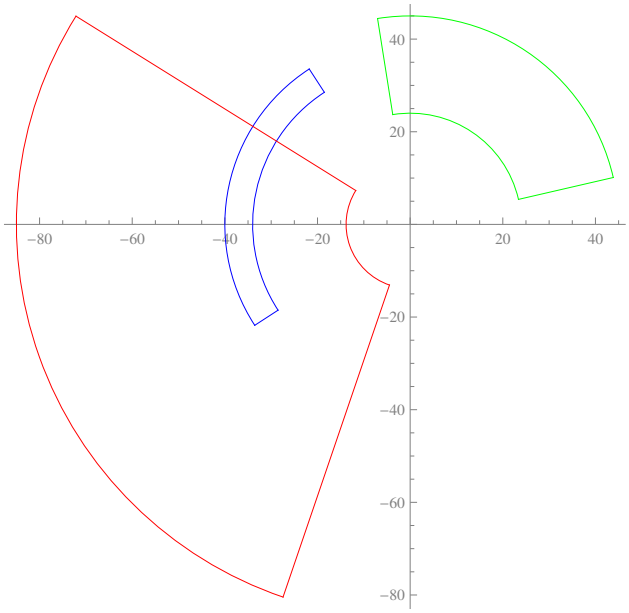
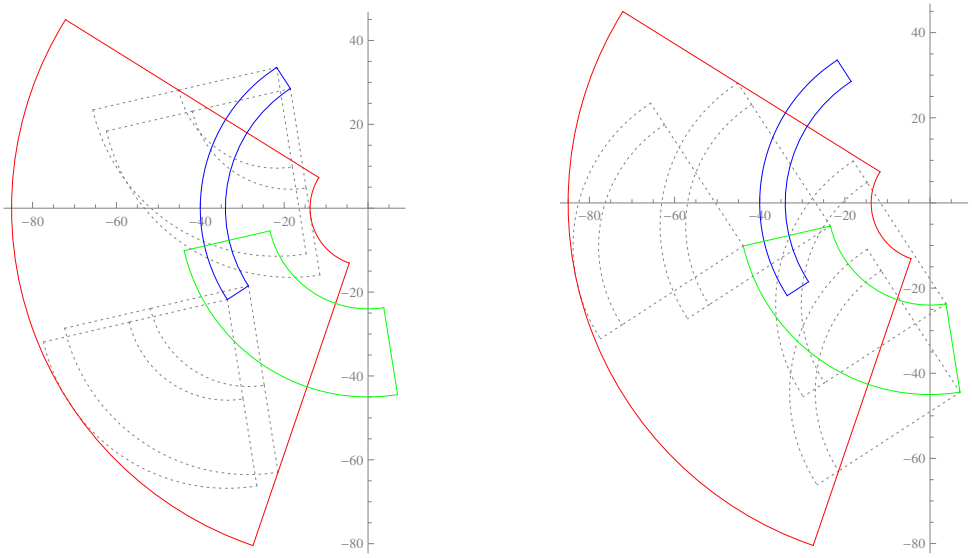


Figura 4.10: Sustracción de dos abanicos complejos

Comprobación de resultados. El resultado obtenido tanto en Java como en Mathematica es: $V_{R6} = [13.829, 85]\angle[148.071, 251.167]$. Ya que la sustracción depende de la adición y la negación, para comprobar el resultado se usa la función `CFSubtractionPlot` de Mathematica, implementada para este propósito,

```
In[52]:=  CFSubtractionPlot[V11, V12, VR6]
```

La instrucción anterior produce las subfiguras de la Figura 4.11. Donde se puede observar que el resultado obtenido tanto en Java como en Mathematica es correcto, ya que es el abanico complejo mínimo que cubre todos los abanicos complejos con líneas punteadas grises.



a) Proyección de $-V_{12}$ con respecto a V_{11} b) Proyección de V_{11} con respecto a $-V_{12}$

Figura 4.11: Gráfica de V_{R6} con respecto a proyecciones

4.2. Pruebas de aplicaciones físicas

En esta sección se presentan pruebas de aplicaciones físicas de la aritmética de abanicos complejos. En el primer caso, en el desplazamiento total de un carro y posteriormente, en el análisis de un circuito RLC en CA. Además se presentan los resultados de cada una de las diferentes pruebas.

4.2.1. Problema 1

"Un carro es conducido hacia el Este una distancia de 50 millas, después 30 millas hacia el Norte y por último 25 millas en una dirección 30° grados del Norte hacia el Este. Dibujar el diagrama vectorial y determinar el desplazamiento total del carro desde su punto de partida."

El problema anterior fue obtenido de [Robert Resnick, 1966], del cual se obtienen los siguientes datos:

$$\overrightarrow{a} = 50\angle 0°$$
$$\overrightarrow{b} = 30\angle 90°$$

$$\overrightarrow{c} = 25\angle 60°$$

Su resultado, es decir, el desplazamiento total del carro es:

$$\overrightarrow{r} = 81.08\angle 39.57°$$

En la Figura 4.12 se muestra el diagrama vectorial del problema, inclusive la solución del problema.

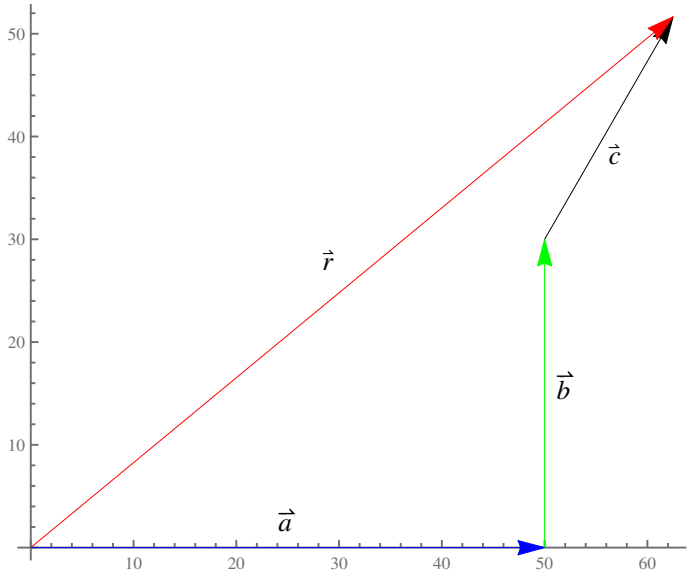


Figura 4.12: Diagrama vectorial

Prueba 1. ¿Que pasa si se busca la solución utilizando las implementaciones?

Los datos del Problema 1 se convierten en:

$$A = [50, 50]\angle[0°, 0°]$$

$$B = [30, 30]\angle[90°, 90°]$$

$$C = [25, 25]\angle[60°, 60°]$$

Cálculo de $R = A + B + C$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Suma de A, B y C-----------");
ComplexFan A = new ComplexFan(new Interval('[',50,50,']'), new
    AngleInterval('[',0,0,']'));
ComplexFan B = new ComplexFan(new Interval('[',30,30,']'), new
    AngleInterval('[',90,90,']'));
ComplexFan C = new ComplexFan(new Interval('[',25,25,']'), new
    AngleInterval('[',60,60,']'));
A.print();
B.print();
```

```
C.print();
System.out.println("-------Resultado-------");
ComplexFan R = ComplexFan.addition(A,B);
R = ComplexFan.addition(R,C);
R.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
$ java PruebasComplexFan
-----------Suma de A, B y C-----------
[50.0,50.0] ∠ [0.0,0.0]
[30.0,30.0] ∠ [90.0,90.0]
[25.0,25.0] ∠ [60.0,60.0]
-------Resultado-------
[81.08044218969614,81.08044218969614] ∠ [39.57067315625065,39.57067315625065]
```

Cálculo de $R = A+B+C$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[53]:=  fanA = CFDefineShort[MI["[", 50, 50, "]"], AI["[", 0, 0, "]"]];

In[54]:=  fanB = CFDefineShort[MI["[", 30, 30, "]"], AI["[", 90, 90, "]"]];

In[55]:=  fanC = CFDefineShort[MI["[", 25, 25, "]"], AI["[", 60, 60, "]"]];

In[56]:=  fanR = CFAddition[CFAddition[fanA, fanB], fanC];

In[57]:=  CFPrint[fanA]
Out[57]:= [50,50] ∠ [0,0]

In[58]:=  CFPrint[fanB]
Out[58]:= [30,30] ∠ [90,90]

In[59]:=  CFPrint[fanC]
Out[59]:= [25,25] ∠ [60,60]

In[60]:=  CFPrint[fanR]
Out[60]:= [81.0804,81.0804] ∠ [39.5707,39.5707]
```

Al comparar los resultados de las pruebas con el resultado del Problema 1, se puede notar que los resultados son correctos tanto en Java como en Mathematica. Estos resultados son los que se esperaban ya que si los algoritmos implementados trabajan con incertidumbre también deberían dar resultados correctos al trabajar sin incertidumbre.

Prueba 2. Ahora, ¿Cuál será la posición final del carro si el odómetro del carro tiene un porcentaje de error de $\pm 2\,\%$ y la brújula tiene un margen de error de $\pm 3^\circ$ en cada medición?

Entonces los datos del Problema 1 se convierten en:

$$cfA = [49, 51] \angle [-3^\circ, 3^\circ]$$

$$cfB = [29.4, 30.6] \angle [87^\circ, 93^\circ]$$

$$cfC = [24.5, 25.5] \angle [57^\circ, 63^\circ]$$

Cálculo de $cfR = cfA + cfB + cfC$ en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Suma de cfA, cfB y cfC-----------");
ComplexFan cfA = new ComplexFan(new Interval('[',49,51,']'), new
   AngleInterval('[',-3,3,']'));
ComplexFan cfB = new ComplexFan(new Interval('[',29.4,30.6,']'), new
   AngleInterval('[',87,93,']'));
ComplexFan cfC = new ComplexFan(new Interval('[',24.5,25.5,']'), new
   AngleInterval('[',57,63,']'));
cfA.print();
cfB.print();
cfC.print();
System.out.println("-------Resultado-------");
ComplexFan cfR = ComplexFan.addition(cfA,cfB);
cfR = ComplexFan.addition(cfR,cfC);
cfR.print();
...
```

Después de ejecutar el código se obtiene el resultado:

```
-----------Suma de cfA, cfB y cfC-----------
[49.0,51.0] ∠ [357.0,3.0]
[29.4,30.6] ∠ [87.0,93.0]
[24.5,25.5] ∠ [57.0,63.0]
-------Resultado-------
[75.64180677249529,86.32940098340137] ∠ [35.33081030119229,43.83103671030323]
```

Cálculo de $cfR = cfA + cfB + cfC$ en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[61]:=  cfA = CFDefineShort[MI["[", 49, 51, "]"], AI["[", -3, 3, "]"]];

In[62]:=  cfB = CFDefineShort[MI["[", 29.4, 30.6, "]"], AI["[", 87, 93, "]"]];

In[63]:=  cfC = CFDefineShort[MI["[", 24.5, 25.5, "]"], AI["[", 57, 63, "]"]];

In[64]:=  cfAB = CFAddition[cfA, cfB];

In[65]:=  cfR = CFAddition[cfAB, cfC];

In[66]:=  CFPrint[cfA]
Out[66]:= [49,51] ∠ [357,3]

In[67]:=  CFPrint[cfB]
Out[67]:= [29.4,30.6] ∠ [87,93]

In[68]:=  CFPrint[cfC]
Out[68]:= [24.5,25.5] ∠ [57,63]

In[69]:=  CFPrint[cfAB]
Out[69]:= [54.4444,62.158] ∠ [26.9622,34.9844]

In[70]:=  CFPrint[cfR]
Out[70]:= [75.6418,86.3294] ∠ [35.3308,43.831]

In[71]:=  CFPlot[{cfA, cfB, cfC, cfAB, cfR}]
```

La instrucción número 71 de la lista anterior produce la gráfica de la Figura 4.13, excepto las etiquetas con los nombres de los abanicos complejos, donde se pueden observar de manera gráfica a cfA, cfB, cfC, $cfAB$ y cfR.

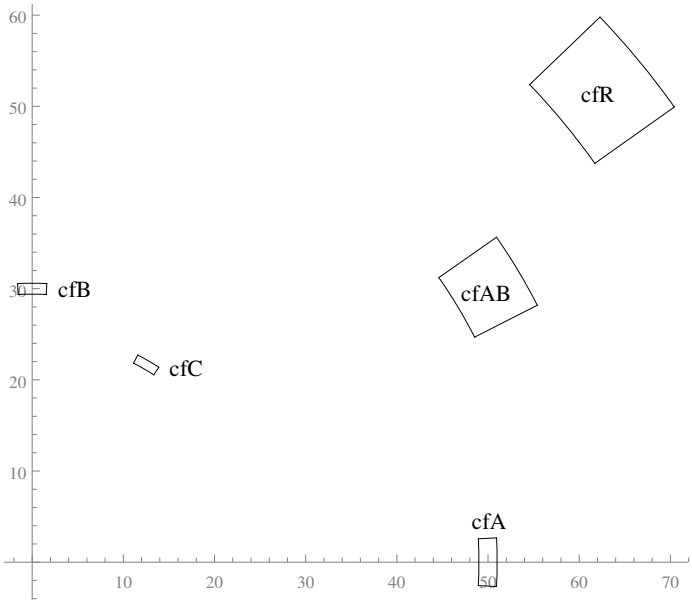


Figura 4.13: Resultados de la Prueba 2

Al comparar el resultado obtenido en Java con el resultado obtenido en Mathematica, se puede notar que los resultados son iguales.

Al analizar los resultados de las pruebas entonces se obtiene que el carro se encontrará a una distancia de entre 75.64 a 86.32 millas y a una dirección de entre 35.33° a 43.83° sobre el Este respecto a la posición inicial.

4.2.2. Problema 2

Análisis del circuito RLC en CA de la Figura 4.14, tomando en cuenta la tolerancia (margen de error) de los elementos pasivos. Para la resistencia $\pm 5\,\%$, para el condensador $\pm 3\,\%$ y para el inductor $\pm 2\,\%$.

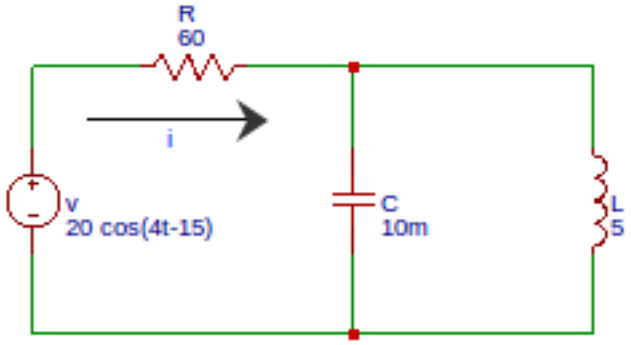


Figura 4.14: Circuito RLC en CA

Se obtienen los siguientes datos del problema:

$$R = [60 - 60 * 0.05, 60 + 60 * 0.05] = [57, 63]\Omega$$

$$C = [0.01 - 0.01 * 0.03, 0.01 + 0.01 * 0.03] = [0.0097, 0.0103]F$$

$$L = [5 - 5 * 0.02, 5 + 5 * 0.02] = [4.9, 5.1]H$$

Para el dominio de la Frecuencia se obtienen los siguientes datos.

Del voltaje fuente obtenemos w:

$$w = 4\frac{rad}{s}$$

El fasor para el voltaje fuente es:

$$V = 20\angle - 15°$$

Cálculo de Z_R

$$Z_R = R = [57, 63]\Omega$$

Cálculo de Z_C

$$Z_C = -\frac{j}{wC} = -\frac{j}{4 * [0.0097, 0.0103]} = -j\frac{[1, 1]}{[0.0388, 0.0412]} = -j * [24.2718, 25.7732]\Omega$$

Cálculo de Z_L

$$Z_L = jwL = j * 4 * [4.9, 5.1] = j * [19.6, 20.4]\Omega$$

En la Figura 4.15 se puede observar el circuito en el dominio de la Frecuencia.

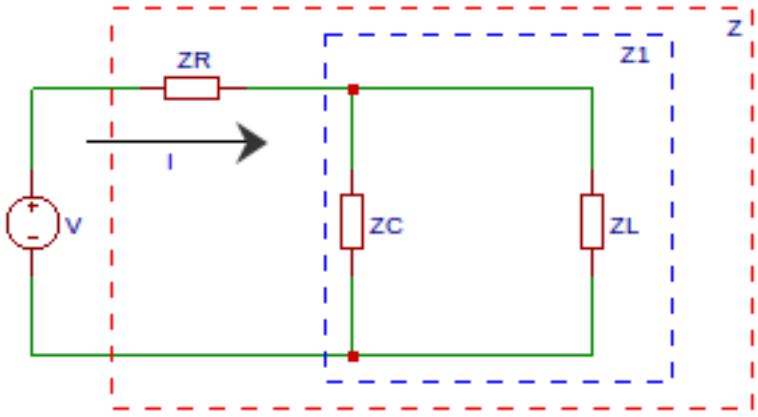


Figura 4.15: Circuito RLC en el dominio de la frecuencia

Conversión de datos al dominio Abanico Complejo

El voltaje fuente queda de la siguiente manera:

$$V = [20, 20]\angle[345°, 345°]$$

La resistencia queda de la siguiente forma:

$$Z_R = [57, 63]\angle[0°, 0°]$$

El condensador:

$$Z_C = [24.2718, 25.7732]\angle[270°, 270°]$$

Y por ultimo el inductor:

$$Z_L = [19.6, 20.4]\angle[90°, 90°]$$

De acuerdo a la Figura 4.15 se tiene que, Z_1 esta compuesta por dos impedancias en paralelo, es decir, por Z_C y Z_L:

$$Z_1 = \frac{Z_L Z_C}{Z_L + Z_C} \tag{4.1}$$

Z esta compuesta por Z_R y Z_1 en serie:

$$Z = Z_R + Z_1 \tag{4.2}$$

Por lo tanto:

$$I = \frac{V}{Z} \tag{4.3}$$

Los voltajes en las impedancias serían:

$$V_{Z_R} = IZ_R \tag{4.4}$$

$$V_{Z_1} = V_{Z_C} = V_{Z_L} = IZ_1 \tag{4.5}$$

Las corrientes en las impedancias están dadas por:

$$I_{Z_R} = I \tag{4.6}$$

$$I_{Z_C} = \frac{V_{Z_1}}{Z_C} \tag{4.7}$$

$$I_{Z_L} = \frac{V_{Z_1}}{Z_L} \tag{4.8}$$

Análisis del circuito en Java. A continuación se muestra parte del código del archivo `PruebasComplexFan.java`:

```
...
System.out.println("-----------Datos-----------");
ComplexFan V = new ComplexFan(new Interval('[',20,20,']'), new
   AngleInterval('[',345,345,']'));
ComplexFan ZR = new ComplexFan(new Interval('[',57,63,']'), new
   AngleInterval('[',0,0,']'));
ComplexFan ZC = new ComplexFan(new Interval('[',24.2718, 25.7732,']'),
    new AngleInterval('[',270,270,']'));
```

```
ComplexFan ZL = new ComplexFan(new Interval('[',19.6, 20.4,']'), new
    AngleInterval('[',90,90,']'));
System.out.println("V = "+V.toString());
System.out.println("ZR = "+ZR.toString());
System.out.println("ZC = "+ZC.toString());
System.out.println("ZL = "+ZL.toString());
System.out.println("-------Resultados-------");
ComplexFan Z1 = ComplexFan.division(ComplexFan.product(ZL,ZC),
    ComplexFan.addition(ZL,ZC));
ComplexFan Z = ComplexFan.addition(ZR,Z1);
ComplexFan I = ComplexFan.division(V,Z);
System.out.println("Z1 = "+Z1.toString());
System.out.println("Z = "+Z.toString());
System.out.println("I = "+I.toString());
ComplexFan VZ1 = ComplexFan.product(I,Z1);
ComplexFan VZR = ComplexFan.product(I,ZR);
ComplexFan VZC = new ComplexFan(VZ1);
ComplexFan VZL = new ComplexFan(VZ1);
System.out.println("VZR = "+VZR.toString());
System.out.println("VZ1 = VZC = VZL = "+VZ1.toString());
ComplexFan IZR = new ComplexFan(I);
ComplexFan IZC = ComplexFan.division(VZ1,ZC);
ComplexFan IZL = ComplexFan.division(VZ1,ZL);
System.out.println("IZR = I = "+IZR.toString());
System.out.println("IZC = "+IZC.toString());
System.out.println("IZL = "+IZL.toString());
...
```

Después de ejecutar el código se obtiene el resultado:

```
$ java PruebasComplexFan
-----------Datos-----------
V = [20.0,20.0] ∠ [345.0,345.0]
ZR = [57.0,63.0] ∠ [0.0,0.0]
ZC = [24.2718,25.7732] ∠ [270.0,270.0]
ZL = [19.6,20.4] ∠ [90.0,90.0]
-------Resultados-------
Z1 = [77.06331886217848,135.79556795289002] ∠ [90.0,90.00000000000006]
Z = [95.8527783324708,149.69781653600685] ∠ [50.733673091510134,67.22983947118684]
I = [0.13360248307422304,0.2086533155108855] ∠ [277.77016052881316,294.2663269084899]
VZR = [7.615341535230713,13.145158877185787] ∠ [277.77016052881316,294.2663269084899]
VZ1 = VZC = VZL = [10.295850753927652,28.334195485054256] ∠
    [7.7701605288131645,24.26632690848993]
IZR = I = [0.13360248307422304,0.2086533155108855] ∠
    [277.77016052881316,294.2663269084899]
IZC = [0.39947894533576167,1.167371001946879] ∠ [97.77016052881316,114.26632690848993]
IZL = [0.5046985663690026,1.445622218625217] ∠ [277.77016052881316,294.26632690848993]
```

Análisis del circuito en Mathematica. A continuación se muestran las instrucciones ejecutadas en la Notebook de trabajo, para obtener el resultado:

```
In[72]:=  V = CFDefineShort[MI["[", 20, 20, "]"], AI["[", -15, -15, "]"]];

In[73]:=  ZR = CFDefineShort[MI["[", 57, 63, "]"], AI["[", 0, 0, "]"]];

In[74]:=  ZC = CFDefineShort[MI["[", 24.2718, 25.7732, "]"],
   AI["[", -90, -90, "]"]];
```

```
In[75]:=  ZL = CFDefineShort[MI["[", 19.6, 20.4, "]"], AI["[", 90, 90, "]"]];

In[76]:=  CFPrint[V]
Out[76]:= [20,20] ∠ [345,345]

In[77]:=  CFPrint[ZR]
Out[77]:= [57,63] ∠ [0,0]

In[78]:=  CFPrint[ZC]
Out[78]:= [24.2718,25.7732] ∠ [270,270]

In[79]:=  CFPrint[ZL]
Out[79]:= [19.6,20.4] ∠ [90,90]

In[80]:=  Z1 = CFDivision[CFProduct[ZL, ZC], CFAddition[ZL, ZC]];

In[81]:=  Z = CFAddition[ZR, Z1];

In[82]:=  It = CFDivision[V, Z];

In[83]:=  VZR = CFProduct[It, ZR];

In[84]:=  VZ1 = CFProduct[It, Z1];

In[85]:=  VZC = VZ1;

In[86]:=  VZL = VZ1;

In[87]:=  CFPrint[Z]
Out[87]:= [95.8528,149.698] ∠ [50.7337,67.2298]

In[88]:=  CFPrint[It]
Out[88]:= [0.133602,0.208653] ∠ [277.77,294.266]

In[89]:=  CFPrint[Z1]
Out[89]:= [77.0633,135.796] ∠ [90.,90.]

In[90]:=  CFPrint[VZ1]
Out[90]:= [10.2959,28.3342] ∠ [7.77016,24.2663]

In[91]:=  CFPrint[VZR]
Out[91]:= [7.61534,13.1452] ∠ [277.77,294.266]

In[92]:=  CFPrint[VZC]
Out[92]:= [10.2959,28.3342] ∠ [7.77016,24.2663]

In[93]:=  CFPrint[VZL]
Out[93]:= [10.2959,28.3342] ∠ [7.77016,24.2663]

In[94]:=  IZR = It;

In[95]:=  IZC = CFDivision[VZ1, ZC];

In[96]:=  IZL = CFDivision[VZ1, ZL];

In[97]:=  CFPrint[IZR]
Out[97]:= [0.133602,0.208653] ∠ [277.77,294.266]

In[98]:=  CFPrint[IZC]
Out[98]:= [0.399479,1.16737] ∠ [97.7702,114.266]

In[99]:=  CFPrint[IZL]
Out[99]:= [0.504699,1.44562] ∠ [277.77,294.266]
```

Al observar los resultados obtenidos tanto en Java como en Mathematica, se obtienen las siguientes conclusiones.

Cuando la resistencia tenga un valor de entre 57Ω a 63Ω la corriente que circulará sobre este elemento se comportará de la siguiente manera:

$$I_{ZR} = [0.133602, 0.208653]\angle[277.77, 294.266] = [a, b]\angle[\alpha_1, \alpha_2] = [a, b]e^{j[\alpha_1, \alpha_2]}$$

Pasando la corriente al dominio del tiempo,

$$i_R(t) = Re[I_{ZR}e^{jwt}] = Re[[a, b]e^{j[\alpha_1, \alpha_2]}e^{jwt}]$$

$$i_R(t) = Re[[a, b]e^{j[\alpha_1, \alpha_2]+jwt}] = Re[[a, b]e^{j([\alpha_1, \alpha_2]+wt)}]$$

$$i_R(t) = [a, b]\cos([\alpha_1, \alpha_2] + wt) = [0.133602, 0.208653]\cos(wt + [277.77, 294.266])$$

donde se puede observar que la amplitud de la corriente será un valor de entre [0.133602, 0.208653] Amperes y el valor del ángulo de fase estará dentro del rango de $[277.77°, 294.266°]$.

Y el voltaje en la resistencia se comportará de la siguiente manera:

$$V_{ZR} = [7.61534, 13.1452]\angle[277.77, 294.266] = [c, d]\angle[\alpha_3, \alpha_4] = [c, d]e^{j[\alpha_3, \alpha_4]}$$

Pasando el voltaje al dominio del tiempo,

$$v_R(t) = Re[V_{ZR}e^{jwt}] = Re[[c, d]e^{j[\alpha_3, \alpha_4]}e^{jwt}]$$

$$v_R(t) = Re[[c, d]e^{j[\alpha_3, \alpha_4]+jwt}] = Re[[c, d]e^{j([\alpha_3, \alpha_4]+wt)}]$$

$$v_R(t) = [c, d]\cos([\alpha_3, \alpha_4] + wt) = [7.61534, 13.1452]\cos(wt + [277.77, 294.266])$$

donde se puede observar que la amplitud del voltaje será un valor de entre [7.61534, 13.1452] Volts y el valor del ángulo de fase estará dentro del rango de $[277.77°, 294.266°]$.

Cuando el Capacitor tenga un valor de entre $9.7mF$ a $10.3mF$ la corriente que circulará sobre este elemento se comportará de la siguiente manera:

$$I_{ZC} = [0.399479, 1.16737]\angle[97.7702, 114.266]$$

Pasando la corriente al dominio del tiempo,

$$i_C(t) = [0.399479, 1.16737]\cos(wt + [97.7702, 114.266])$$

donde se puede observar que la amplitud de la corriente será un valor de entre [0.399479, 1.16737] Amperes y el valor del ángulo de fase estará dentro del rango de $[97.7702°, 114.266°]$.

Y el voltaje en la Capacitancia se comportará de la siguiente manera:

$$V_{ZC} = [10.2959, 28.3342]\angle[7.77016, 24.2663]$$

Pasando el voltaje al dominio del tiempo,

$$v_C(t) = [10.2959, 28.3342]\cos(wt + [7.77016, 24.2663])$$

donde se puede observar que la amplitud del voltaje será un valor de entre [10.2959, 28.3342] Volts y el valor del ángulo de fase estará dentro del rango de $[7.77016°, 24.2663°]$.

Cuando el Inductor tenga un valor de entre $4.9H$ a $5.1H$ la corriente que circulará sobre este elemento se comportará de la siguiente manera:

$$I_{ZL} = [0.504699, 1.44562]\angle[277.77, 294.266]$$

Pasando la corriente al dominio del tiempo,

$$i_L(t) = [0.504699, 1.44562]\cos(wt + [277.77, 294.266])$$

donde se puede observar que la amplitud de la corriente será un valor de entre [0.504699, 1.44562] Amperes y el valor del ángulo de fase estará dentro del rango de $[277.77°, 294.266°]$.

Y el voltaje en la Inductancia se comportará de la siguiente manera:

$$V_{ZL} = [10.2959, 28.3342]\angle[7.77016, 24.2663]$$

Pasando el voltaje al dominio del tiempo,

$$v_Z(t) = [10.2959, 28.3342]\cos(wt + [7.77016, 24.2663])$$

donde se puede observar que la amplitud del voltaje será un valor de entre [10.2959, 28.3342] Volts y el valor del ángulo de fase estará dentro del rango de $[7.77016°, 24.2663°]$.

4.3. Conclusiones

Se han presentado diversas pruebas elementales para probar el funcionamiento correcto de cada uno de los algoritmos implementados. Estas pruebas se efectuaron tanto en Java como en Mathematica. Como se pudo ver en las comprobaciones que se presentaron, los resultados de las pruebas son correctos.

Además se presentaron pruebas de aplicaciones físicas con los que se utilizó cada una de las operaciones aritméticas para abanicos complejos que fueron implementadas. Los resultados de estas pruebas demuestran la utilidad de estos algoritmos implementados en aplicaciones físicas.

Los resultados satisfactorios de las pruebas presentadas en este capítulo y de las no documentadas aquí, sugieren que tanto la implementación en Java como en Mathematica funcionan correctamente. Son implementaciones estables.

Capítulo 5

Conclusiones

En esta tesis se presentó la implementación de los algoritmos propuestos en [Flores, 1998] para la aritmética de abanicos complejos junto con las funciones adicionales para implementar dichos algoritmos. La implementación se hizo en Java y en Mathematica. Además se presentaron diversas pruebas que se realizaron con sus respectivos resultados para verificar el buen funcionamiento de las implementaciones.

5.1. Conclusiones Generales

Se han presentado satisfactoriamente los requerimientos del sistema, en donde se describieron los Objetos: Intervalo, Intervalo de magnitud, Intervalo de ángulo y Abanico Complejo; además se realizó el análisis tanto de los requerimientos como de las consideraciones para la implementación del sistema.

Se realizaron satisfactoriamente las implementaciones de los algoritmos y las funciones adicionales para el buen funcionamiento de los algoritmos. Para Java se diseñaron y se implementaron las clases Interval para los intervalos de magnitud, AngleInterval para los intervalos de ángulo y ComplexFan para los abanicos complejos; dentro de las clases se implementaron los atributos de cada objeto, los métodos y las funciones para su manejo; todo esto para que los algoritmos para la aritmética de abanicos complejos funcionen en Java. En Mathematica se diseñaron y se implementaron dos patrones, uno para representar a los intervalos de magnitud y de ángulo, y otro para representar a los abanicos complejos; además de los métodos y las funciones para la manipulación de estos patrones; todo esto para que los algoritmos para la aritmética de abanicos complejos funcionen en Mathematica.

Se realizaron las pruebas elementales a las implementaciones satisfactoriamente, con excelentes resultados, con excepción a la prueba de la Sección 4.1.5, en la prueba que se realizó en Mathematica para la sustracción, donde Mathematica arrojó algunas advertencias. Se realizaron las pruebas de aplicaciones físicas de manera exitosa, ya que se demostró la utilidad de los algoritmos implementados en aplicaciones físicas.

Las implementaciones respondieron satisfactoriamente a todas las pruebas documentadas y no documentadas que se efectuaron sobre ellas por lo tanto actualmente se consideran implementaciones estables.

Por los lenguajes en las que se realizaron las implementaciones, en Java y en Mathematica, este trabajo tendrá una alta disponibilidad para las personas que deseen probar los algoritmos o usarlos como parte de sus proyectos.

Las implementación en Java producto de esta tesis se encuentra disponible en [machz1988, 2016a] y la de Mathematica en [machz1988, 2016b].

5.2. Trabajos Futuros

Corregir las advertencias que aparecen en la Sección 4.1.5, en la prueba que se realiza en Mathematica para la sustracción. Al parecer tiene que ver con las funciones nativas de Mathematica, Min[] y Max[] que no trabajan con números complejos.

Implementar soporte para poder realizar operaciones aritméticas en abanicos complejos, donde sus atributos, es decir, sus intervalos de magnitud y de ángulo, tengan límites abiertos. Ya que por el momento los algoritmos regresan resultados suponiendo que reciben intervalos con limites cerrados.

Desarrollar un programa tipo shell o uno con interfaces gráficas, en Java, para usar aritmética de abanicos complejos.

Migrar las implementaciones a otros lenguajes de programación, si se requiere, lo cual debería ser fácil con la ayuda de este trabajo.

Apéndice A

Detalles de la implementación en Java

A continuación se presentan las clases que se implementaron en Java para cubrir los requerimientos de la Sección 3.1. Además se presenta información acerca de atributos, subclases, clases padre, constructores, métodos y funciones relacionadas a cada clase.

A.1. class Interval

Subclases

- AngleInterval

Atributos

- firstExtreme
  - `private double firstExtreme`
  - el primer extremo del intervalo
- secondExtreme
  - `private double secondExtreme`
  - el segundo extremo del intervalo
- feIncluded
  - `private char feIncluded`
  - el límite para el primer extremo del intervalo
- seIncluded
  - `private char seIncluded`
  - el límite para el segundo extremo del intervalo

Constructores

- Interval
  - `public Interval()`
  - Constructor que inicializa un intervalo de magnitud vacío, es decir, $(0, 0)$.
- Interval
  - `public Interval(char fei, double fe, double se, char sei)`
  - Constructor que inicializa un intervalo de magnitud con los valores recibidos para sus atributos.
  - Parámetros:

    `fei` .- el límite del primer extremo

    `fe` .- el valor del primer extremo

    `se` .- el valor del segundo extremo

    `sei` .- el límite del segundo extremo
  - Detalles:

    Será obligación del usuario mandar valores para 'fei' y 'sei' válidos, es decir, para 'fei' los valores esperados son '(' ó '[' y para 'sei' son ')' ó ']'. Si el usuario manda valores diferentes a estos, no se promete un funcionamiento correcto del sistema.
- Interval
  - `public Interval(double fe, double se, char fei, char sei)`
  - Constructor que inicializa un intervalo de magnitud con los valores recibidos para sus atributos, en otro orden.
  - Parámetros:

    `fe` .- el valor del primer extremo

    `se` .- el valor del segundo extremo

    `fei` .- el límite del primer extremo

    `sei` .- el límite del segundo extremo
  - Detalles:

    Será obligación del usuario mandar valores para 'fei' y 'sei' válidos, es decir, para 'fei' los valores esperados son '(' ó '[' y para 'sei' son ')' ó ']'. Si el usuario manda valores diferentes a estos, no se promete un funcionamiento correcto del sistema.
- Interval
  - `public Interval(Interval in)`
  - Constructor para inicializar un intervalo de magnitud a partir de otro intervalo de magnitud.
  - Parámetros:

    `in` .- el intervalo de magnitud a ser copiado

Métodos

- setFirstExtreme
  - `public void setFirstExtreme(double fe)`
  - Función para asignar el valor del primer extremo.
  - Parámetros:

    `fe` .- el nuevo valor del primer extremo
- setSecondExtreme
  - `public void setSecondExtreme(double se)`
  - Función para asignar el valor del segundo extremo.
  - Parámetros:

    `se` .- el nuevo valor del segundo extremo
- setFEincluded
  - `public void setFEincluded(char fei)`
  - Función para asignar el límite del primer extremo.
  - Parámetros:

    `fei` .- el nuevo límite del primer extremo.
  - Detalles:

    Se dejará al usuario la responsabilidad de mandar un valor válido, ya sea '(' ó '['. No se prometerá un buen funcionamiento si el usuario no manda un valor válido.
- setSEincluded
  - `public void setSEincluded(char sei)`
  - Función para asignar el límite del segundo extremo.
  - Parámetros:

    `sei` .- el nuevo límite del segundo extremo.
  - Detalles:

    Se dejará al usuario la responsabilidad de mandar un valor válido, ya sea ')' ó ']'. No se prometerá un buen funcionamiento si el usuario no manda un valor válido.
- getFirstExtreme
  - `public double getFirstExtreme()`
  - Función para obtener el valor del primer extremo.
  - Regresa:

    el valor del primer extremo

- getSecondExtreme
  - `public double getSecondExtreme()`
  - Función para obtener el valor del segundo extremo.
  - Regresa:

    el valor del segundo extremo
- getFEincluded
  - `public char getFEincluded()`
  - Función para obtener el límite del primer extremo, es decir, regresa '(' o '['.
  - Regresa:

    el límite del primer extremo
- getSEincluded
  - `public char getSEincluded()`
  - Función para obtener el límite del segundo extremo, es decir, regresa ')' o ']'.
  - Regresa:

    el límite del segundo extremo
- normalize
  - `public void normalize()`
  - Función para normalizar un intervalo de magnitud, es decir, que el primer extremo sea menor o igual que el segundo extremo.
- byConstant
  - `public void byConstant(double k)`
  - Función para multiplicar un intervalo por una constante.
  - Parámetros:

    `k` .- el factor constante
  - Detalles:

    Cuando la función recibe una constante positiva, hace la siguiente operación: $[a, b] * k = [a * k, b * k]$. Pero cuando la constante es negativa, hace lo siguiente: $[c, d] * k = [d * k, c * k]$.
- isEmpty
  - `public boolean isEmpty()`
  - Función para verificar si un intervalo esta vacío.
  - Regresa:

    si el intervalo es vacío

- union
  - `public static Interval union(Interval in1, Interval in2)`
  - Función para calcular la unión de dos intervalos. Esta función primero verifica si existe intersección entre los dos intervalos que recibe, si lo hay realiza la operación de la unión, si no lo hay regresa un intervalo vacío.
  - Parámetros:

    `in1` .- el primer intervalo para la unión

    `in2` .- el segundo intervalo para la unión
  - Regresa:

    el resultado de la unión de intervalos
  - Detalles:

    Esta función verifica los diferentes casos que pueden surgir en la unión de dos intervalos, basándose en la posición de los extremos de cada intervalo para así encontrar la solución.
- intersection
  - `public static Interval intersection(Interval in1, Interval in2)`
  - Función para calcular la intersección de dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo para la intersección

    `in2` .- el segundo intervalo para la intersección
  - Regresa:

    el resultado de la intersección
  - Detalles:

    Esta función verifica los diferentes casos que pueden surgir al efectuar una intersección, basándose en la posición de cada extremo de cada intervalo.
- negation
  - `public static Interval negation(Interval in)`
  - Función para calcular la negación de un intervalo.
  - Parámetros:

    `in` .- el intervalo a negar
  - Regresa:

    el resultado de la negación
  - Detalles:

    Realiza la siguiente operación: $-[a, b] = [-b, -a]$.
- addition
  - `public static Interval addition(Interval in1, Interval in2)`
  - Función para sumar dos intervalos.

  - Parámetros:

    `in1` .- el primer intervalo de la suma
    `in2` .- el segundo intervalo de la suma

  - Regresa:

    el resultado de la suma

  - Detalles:

    Realiza la siguiente operación: $[a, b] + [c, d] = [a + c, b + d]$

- product

  - `public static Interval product(Interval in1, Interval in2)`
  - Función para multiplicar dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo del producto
    `in2` .- el segundo intervalo del producto

  - Regresa:

    el resultado del producto

  - Detalles:

    Realiza la siguiente operación: $[a, b] * [c, d] = [a * c, b * d]$

- subtraction

  - `public static Interval subtraction(Interval in1, Interval in2)`
  - Función para restar dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo para la resta
    `in2` .- el segundo intervalo para la resta

  - Regresa:

    el resultado de la resta

  - Detalles:

    Realiza la siguiente operación: $[a, b]–[c, d] = [a - d, b - c]$

- division

  - `public static Interval division(Interval in1, Interval in2)`
  - Función para dividir dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo de la división
    `in2` .- el segundo intervalo de la división

  - Regresa:

    el resultado de la división

  - Detalles:

    Realiza la siguiente operación: $[a, b]/[c, d] = [a/d, b/c]$. En caso de que 'd' sea cero o 'c' sea cero, imprime un mensaje de "División por cero!" y se sale del sistema.

- print
  - `public void print()`
  - Función para imprimir la cadena de caracteres que representa un intervalo.
- toString
  - `public java.lang.String toString()`
  - Función que regresa la representación de un intervalo en cadena de caracteres.
  - Sobrescribe:

    `toString` de la clase `java.lang.Object`
  - Regresa:

    la cadena de caracteres que representa el intervalo

A.2. class AngleInterval

Clase padre

- Interval

Atributos heredados de la clase Interval

- `firstExtreme, secondExtreme, feIncluded, seIncluded`

Métodos heredados de la clase Interval

- `addition, byConstant, division, getFEincluded, getFirstExtreme, getSecondExtreme, getSEincluded, intersection, isEmpty, negation, print, product, setFEincluded, setFirstExtreme, setSecondExtreme, setSEincluded, subtraction, toString, union`

Constructores

- AngleInterval
  - `public AngleInterval()`
  - Constructor que inicializa el primer y el segundo extremo en 0.0, el primer límite en '(' y el segundo límite en ')', es decir, un intervalo vacío. Utiliza el constructor de la clase padre.
- AngleInterval
  - `public AngleInterval(char feIncluded, double firstExtreme, double secondExtreme, char seIncluded)`

  - Constructor para inicializar un intervalo de ángulo de acuerdo a los valores recibidos.
  - Parámetros:

    `feIncluded` .- el límite para el primer extremo

    `firstExtreme` .- el valor del primer extremo

    `secondExtreme` .- el valor del segundo extremo

    `seIncluded` .- el límite para el segundo extremo

- AngleInterval
  - `public AngleInterval(double firstExtreme, double secondExtreme, char feIncluded, char seIncluded)`
  - Constructor para inicializar un intervalo de ángulo de acuerdo a los valores recibidos, en otro orden.
  - Parámetros:

    `firstExtreme` .- el valor del primer extremo

    `secondExtreme` .- el valor del segundo extremo

    `feIncluded` .- el límite para el primer extremo

    `seIncluded` .- el límite para el segundo extremo

- AngleInterval
  - `public AngleInterval(AngleInterval ai)`
  - Constructor para inicializar un intervalo de ángulo a partir de otro intervalo de ángulo.
  - Parámetros

    `ai` .- el intervalo de ángulo a copiar

- AngleInterval
  - `public AngleInterval(Interval in)`
  - Constructor para inicializar un intervalo de ángulo a partir de un intervalo.
  - Parámetros

    `in` .- el intervalo a copiar

Métodos

- normalize
  - `public void normalize()`
  - Esta función normaliza los extremos del intervalo de ángulo.
  - Sobre escribe:

    `normalize` de la clase `Interval`

  - Detalles:

    Un intervalo de ángulo está normalizado si el valor de sus extremos está dentro del intervalo de [0,360]. Normaliza intervalos con valores negativos y valores arriba de 360 grados.

- modulo360
  - `public static double modulo360(double opd)`
  - Función para sacar módulo base 360 al valor que recibe.
  - Parámetros:

    `opd` .- el valor al cual se va a sacar módulo base 360
  - Regresa:

    el módulo base 360 del valor recibido
  - Detalles:

    Se estuvo probando la función `java.lang.Math.IEEEremainder(double f1, double f2)` para funciones que necesitan sacar modulo 360 como la función `normalize`, obteniéndose resultados ambiguos. Es por eso que se implementó esta función para sacar módulo 360 a un valor.

- addition
  - `public static AngleInterval addition(AngleInterval ai1, AngleInterval ai2)`
  - Función para sumar dos intervalos de ángulo.
  - Parámetros:

    `ai1` .- el primer operando de la suma

    `ai2` .- el segundo operando de la suma
  - Regresa:

    el resultado de la suma
  - Detalles:

    Esta función usa la función `addition` de la clase `Interval` para realizar la suma de los dos intervalos, después normaliza el resultado y devuelve el resultado.

- subtraction
  - `public static AngleInterval subtraction(AngleInterval ai1, AngleInterval ai2)`
  - Función para restar dos intervalos de ángulo.
  - Parámetros:

    `ai1` .- el primer operando de la resta

    `ai2` .- el segundo operando de la resta
  - Regresa:

    el resultado de la resta

  - Detalles:

    Esta función usa la función `subtraction` de la clase `Interval` para realizar la resta de los dos intervalos, después normaliza el resultado y lo devuelve.

- add180toAI
  - `public static AngleInterval add180toAI(AngleInterval ai)`
  - Esta función suma 180 grados al intervalo de ángulo que recibe. Depende de la función `addition`.
  - Parámetros:

    `ai` .- el intervalo de ángulo a utilizar
  - Regresa:

    el resultado de sumar 180 grados al intervalo de ángulo

- VerifyCase0to360
  - `public boolean VerifyCase0to360()`
  - Esta función verifica si un intervalo de ángulo abarca los 4 cuadrantes, es decir, es igual a [0, 360].
  - Regresa:

    si el intervalo de ángulo abarca los 360 grados

- verifyCase0to360ofList
  - `public static boolean verifyCase0to360ofList(AngleInterval[] lais)`
  - Función para verificar si un arreglo de intervalos de ángulo en conjunto abarcan los 360 grados.
  - Parámetros:

    `lais` .- el arreglo de intervalos de ángulo
  - Regresa:

    si el arreglo de intervalos, en conjunto abarcan los 360 grados

- unionAIs
  - `public static AngleInterval unionAIs(AngleInterval[] res)`
  - Función para unir un arreglo de intervalos de ángulo.
  - Parámetros:

    `res` .- el arreglo de intervalos de ángulo a unir
  - Regresa:

    el resultado de la unión
  - Detalles:

    Esta función espera recibir intervalos de ángulo que representan a cada cuadrante, por lo tanto, verifica adyacencias, conforme encuentra adyacencias va uniendo intervalos, por ejemplo:
    Unir los siguientes intervalos [0, 90), [90, 180) y [180, 270).

Como hay adyacencia entre [0, 90) y [90, 180), estos se unen dando como resultado [0, 180), este a su vez es adyacente con [180, 270), por lo tanto se unen dando como resultado [0, 270), como ya no quedan más intervalos se devuelve el resultado [0, 270).

A.3. class ComplexFan

Atributos

- magnitudeInterval
  - `private Interval magnitudeInterval`
  - el intervalo de magnitud
- angleInterval
  - `private AngleInterval angleInterval`
  - el intervalo de ángulo

Constructores

- ComplexFan
  - `public ComplexFan()`
  - Constructor que inicializa un abanico complejo con valores por defecto para sus atributos, es decir, el intervalo de magnitud y el intervalo de ángulo como vacíos.
- ComplexFan
  - `public ComplexFan(Interval mi, AngleInterval ai)`
  - Constructor que inicializa el intervalo de magnitud y el intervalo de ángulo de acuerdo con los intervalos que recibe; cabe recalcar que después de inicializar los atributos antes mencionados se les aplica la normalización de acuerdo a su tipo.
- ComplexFan
  - `public ComplexFan(ComplexFan cf)`
  - Constructor que inicializa un abanico complejo a partir de otro abanico complejo.
  - Parámetros:

    `cf` .- el abanico complejo a copiar

Métodos

- setMagnitudeInterval
  - `public void setMagnitudeInterval(Interval mi)`
  - Función para asignar un nuevo valor al atributo magnitudeInterval.
  - Parámetros:

    `mi` .- el nuevo valor para el intervalo de magnitud

- setAngleInterval
  - `public void setAngleInterval(AngleInterval ai)`
  - Función para asignar un nuevo valor al atributo angleInterval.
  - Parámetros:
    `ai` .- el nuevo valor para el intervalo de ángulo
- getMagnitudeInterval
  - `public Interval getMagnitudeInterval()`
  - Función para obtener el intervalo de magnitud.
  - Regresa:
    el valor actual del intervalo de magnitud
- getAngleInterval
  - `public AngleInterval getAngleInterval()`
  - Función para obtener el intervalo de ángulo.
  - Regresa:
    el valor actual del intervalo de ángulo
- negation
  - `public static ComplexFan negation(ComplexFan cf)`
  - Función para calcular la negación de un abanico complejo.
  - Parámetros:
    `cf` .- el abanico complejo del cual se va a calcular la negación
  - Regresa:
    el resultado de la negación
  - Detalles:
    Implementa el algoritmo para la negación de un abanico complejo.
- product
  - `public static ComplexFan product(ComplexFan cf1, ComplexFan cf2)`
  - Función para calcular el producto entre dos abanicos complejos.
  - Parámetros:
    `cf1` .- el primer operando para el producto
    `cf2` .- el segundo operando para el producto
  - Regresa:
    el resultado del producto
  - Detalles:
    Implementa el algoritmo para el producto entre dos abanicos complejos.

- division
  - `public static ComplexFan division(ComplexFan cf1, ComplexFan cf2)`
  - Función para calcular la división entre dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando de la división

    `cf2` .- el segundo operando de la división
  - Regresa:

    el resultado de la división
  - Detalles:

    Implementa el algoritmo para la división entre dos abanicos complejos.
- addition
  - `public static ComplexFan addition(ComplexFan cf1, ComplexFan cf2)`
  - Función general para calcular la suma de dos abanicos complejos. Depende de funciones como `unionOfResults`, `verifyCase`, `part`, `additionCase1`, `additionCase2`, `additionCase3`, etc.
  - Parámetros:

    `cf1` .- el primer operando de la suma

    `cf2` .- el segundo operando de la suma
  - Regresa:

    el resultado de la suma
  - Detalles:

    Implementa una función general para la suma de dos abanicos complejos, usando las consideraciones mencionados en la Sección 3.1.4 y los algoritmos de los 3 casos de adición en la Sección 2.5, siguiendo los siguientes pasos (véase también la Figura A.1):

    1. Si es necesario, se rotan los dos abanicos complejos de modo que el primer abanico complejo empiece sobre el eje X positivo.
    2. Se parten los dos abanicos complejos en partes de acuerdo a sus intersecciones con el plano cartesiano.
    3. Se va tomando cada parte del primer operando con cada parte del segundo operando mediante dos iteraciones anidadas, se verifica en que caso de adición cae cada suma parcial, es decir, la suma entre la parte en turno del primer operando con la parte en turno del segundo operando, se usa el algoritmo de acuerdo al caso para realizar la suma parcial. Cada resultado parcial se va guardando en un arreglo de resultados parciales.
    4. Al terminar las iteraciones del paso anterior, se unen los resultados parciales.
    5. Si se aplicó la rotación en el paso 1, se corrige la rotación aplicada al resultado total obtenido en el paso anterior.

6. Se devuelve el resultado.

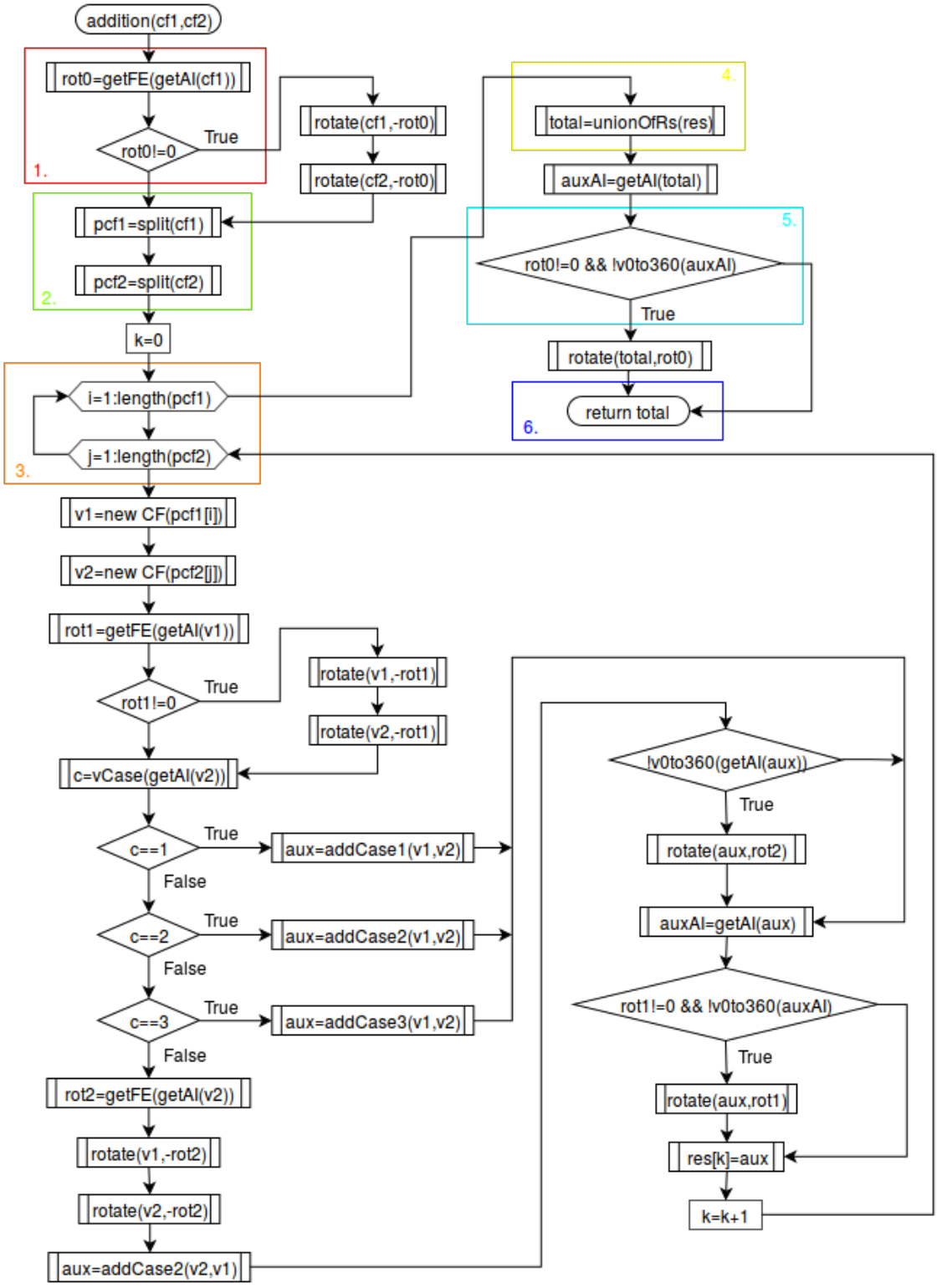


Figura A.1: Diagrama de flujo de la función addition

- subtraction
  - `public static ComplexFan subtraction(ComplexFan cf1, ComplexFan cf2)`
  - Función para calcular la resta entre dos abanicos complejos, esta función depende de la función `negation` y la función `addition`, es decir, se realiza la negación del segundo abanico complejo y el resultado se suma al primer abanico complejo.
  - Parámetros:

    `cf1` .- el primer operando de la resta

    `cf2` .- el segundo operando de la resta
  - Regresa:

    el resultado de la resta
- unionOfResults
  - `private static ComplexFan unionOfResults(ComplexFan []acf)`
  - Función para realizar la unión de un arreglo de abanicos complejos. Depende de las funciones `unionOfMIs` y `unionOfAIs`.
  - Parámetros:

    `acf` .- el arreglo de abanicos complejos
  - Regresa:

    el resultado de la unión
- unionOfAIs
  - `private static AngleInterval unionOfAIs(ComplexFan []acf)`
  - Función para realizar la unión de los intervalos de ángulo provenientes de un arreglo de abanicos complejos.
  - Parámetros:

    `acf` .- el arreglo de abanicos complejos
  - Regresa:

    el resultado de la unión de los intervalos de ángulo
  - Detalles:

    Esta función trabaja de acuerdo con los siguientes pasos (también véase la Figura A.2):

    1. Se saca el primer abanico complejo del arreglo de abanicos complejos y se parte este de acuerdo a su intersección con el plano cartesiano, por lo tanto, el primer abanico complejo se convierte en otro arreglo de abanicos complejos.
    2. Se entra en un ciclo infinito.
    3. Se entra en un ciclo que recorre el arreglo de abanicos complejos restante. Conforme se va recorriendo el arreglo, se toma el abanico complejo en turno y se parte de acuerdo a su intersección con el plano cartesiano. Ahora el abanico complejo en turno es un arreglo de abanicos complejos.

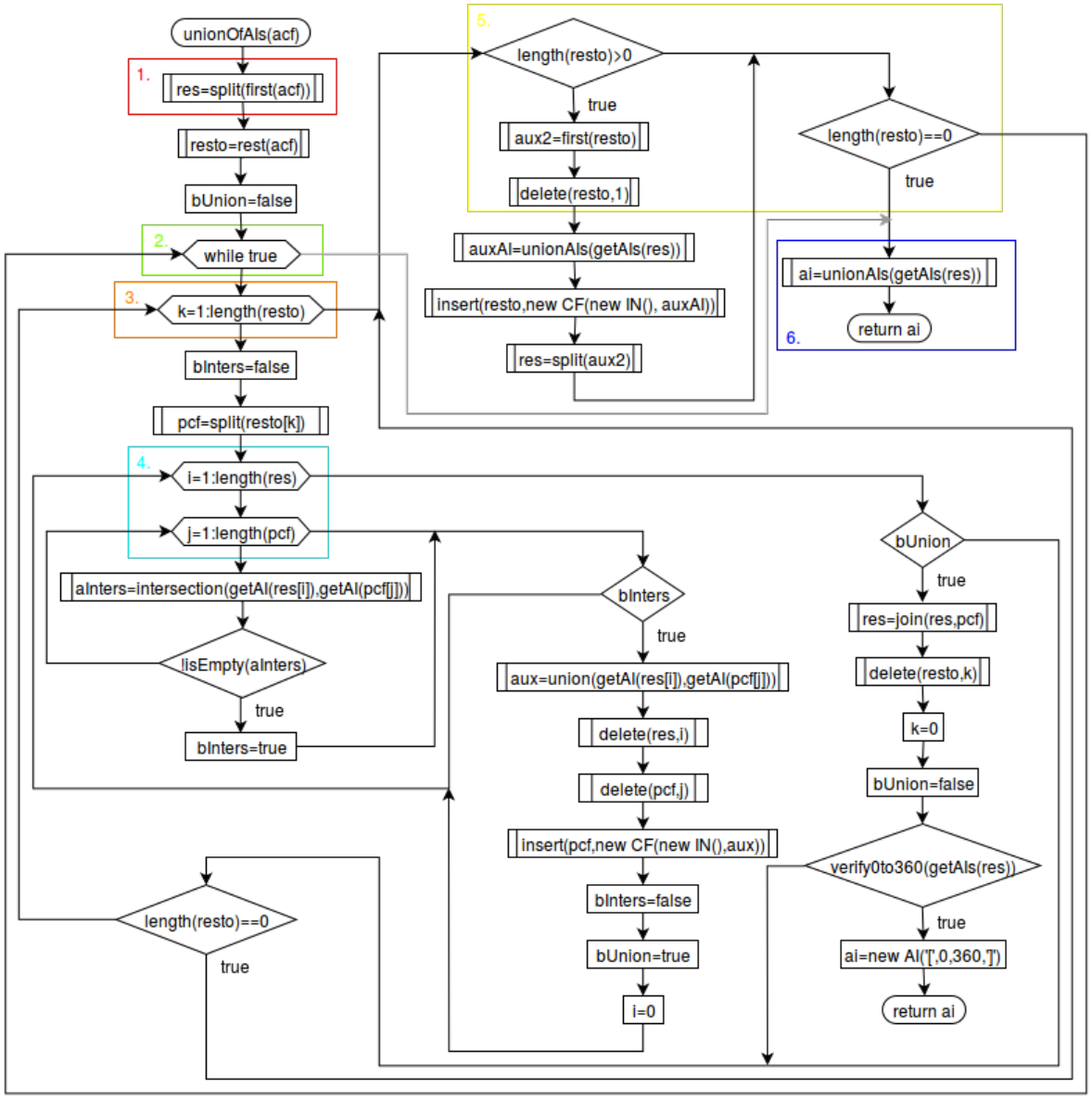


Figura A.2: Diagrama de flujo de la función unionOfAIs

4. Mediante dos ciclos anidados se realiza la unión de intervalos de ángulo a nivel cuadrante del primer abanico complejo y el abanico complejo en turno. Si se uso el abanico complejo en turno se elimina del arreglo de abanicos complejos y los resultados de las uniones parciales se van actualizando sobre el primer abanico complejo.
5. Cuando termina de recorrer el arreglo de abanicos complejos y este ya no contiene elementos se rompe el ciclo infinito. Si el arreglo todavía contiene elementos, se realiza la unión de partes del primer abanico complejo y el resultado se pone al final del arreglo de abanicos complejos, se toma el nuevo primer abanico complejo del arreglo de abanicos complejos, se parte el nuevo primer abanico complejo, por lo tanto ya es un arreglo de abanicos complejos y se repite el proceso a partir del paso 3, con el nuevo primer

abanico complejo y el nuevo arreglo de abanicos complejos restante.

6. Se realiza la unión de los resultados parciales y se devuelve el resultado final.

- unionOfMIs
  - `private static Interval unionOfMIs(Interval []mis)`
  - Función para unir un arreglo de intervalos de magnitud.
  - Parámetros:

    `mis` .- el arreglo de intervalos de magnitud
  - Regresa:

    el resultado de la unión de los intervalos de magnitud
  - Detalles:

    Esta función realiza este proceso siguiendo los siguientes pasos (también véase la Figura A.3):

    1. Saca el primer intervalo del arreglo.
    2. Entra en un ciclo infinito.
    3. Entra en un ciclo que recorre el arreglo restante. Si existe intersección entre el primer intervalo y el intervalo en turno realiza la unión de estos, guarda el resultado como el primer intervalo, elimina el intervalo en turno del arreglo y reinicia el ciclo.
    4. Si al terminar el ciclo del paso anterior todavía hay elementos en el arreglo, se mete el primer intervalo al final del arreglo, se saca el nuevo primer intervalo y se regresa al paso 3. De lo contrario, si ya no existen elementos en el arreglo se rompe el ciclo infinito.
    5. Se devuelve el resultado.

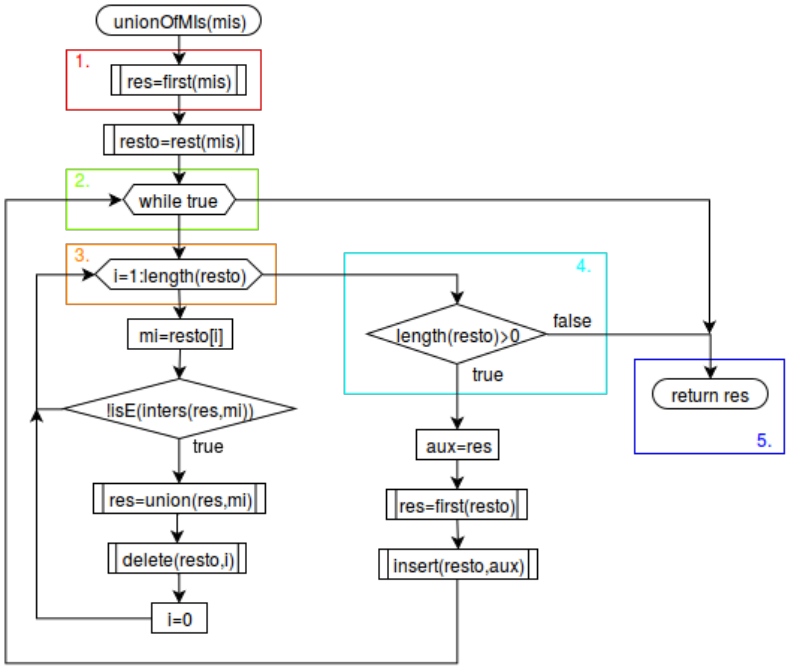


Figura A.3: Diagrama de flujo de la función unionOfMIs

- additionCase1
  - `private ComplexFan additionCase1(ComplexFan cf1, ComplexFan cf2)`
  - Función que implementa el algoritmo para el caso 1 para la adición de dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- additionCase2
  - `private ComplexFan additionCase2(ComplexFan cf1, ComplexFan cf2)`
  - Función para la suma de dos abanicos complejos caso 2. Depende de las funciones `magnitudeCase2` y `angleCase2`.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- magnitudeCase2
  - `private Interval magnitudeCase2(ComplexFan cf1, ComplexFan cf2)`
  - Función que implementa el algoritmo para calcular el intervalo de magnitud resultante en la suma de dos abanicos complejos caso 2.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de magnitud resultante
- angleCase2
  - `private AngleInterval angleCase2(ComplexFan cf1, ComplexFan cf2)`
  - Función que implementa el algoritmo para calcular el intervalo de ángulo resultante en la suma de dos abanicos complejos caso 2.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de ángulo resultante

- additionCase3
  - `private ComplexFan additionCase3(ComplexFan cf1, ComplexFan cf2)`
  - Función para la suma de dos abanicos complejos para el caso 3. Depende de las funciones `magnitudeCase3` y `angleCase3`.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- magnitudeCase3
  - `private Interval magnitudeCase3(ComplexFan cf1, ComplexFan cf2)`
  - Función que implementa el algoritmo para calcular el intervalo de magnitud resultante en la suma de dos abanicos complejos para el caso 3.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de magnitud resultante
- angleCase3
  - `private AngleInterval angleCase3(ComplexFan cf1, ComplexFan cf2)`
  - Función que implementa el algoritmo para calcular el intervalo de ángulo resultante en la suma de dos abanicos complejos para el caso 3.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de ángulo resultante
- angleFixer
  - `private double angleFixer(double x, double y)`
  - Función para corregir el ángulo resultante, al calcular el ángulo de un vector.
  - Parámetros:

    `x` .- la componente en x del vector

    `y` .- la componente en y del vector
  - Regresa:

    el ángulo resultante

- Detalles:

  Esta función es para corregir el cálculo del ángulo de un vector, mediante el uso de sus componentes, por ejemplo, dado los siguientes vectores, calculemos sus ángulos (en grados):

  $< 3, 4 >$ su ángulo sería $\theta = \tan^{-1}(4/3) = 49.85$

  $< -3, 4 >$ su ángulo sería $\theta = \tan^{-1}(4/-3) = -49.85$

  $< -3, -4 >$ su ángulo sería $\theta = \tan^{-1}(-4/-3) = 49.85$

  $< 3, -4 >$ su ángulo sería $\theta = \tan^{-1}(-4/3) = -49.85$

  Los resultados anteriores no nos sirven de mucho, ya que en nuestro sistema estamos trabajando con los 4 cuadrantes del plano cartesiano, entonces necesitamos valores normalizados de los ángulos resultantes, es decir, que los valores estén dentro de los 360 grados, para los ejemplos anteriores los valores que son útiles son los siguientes:

  $< 3, 4 >$ su ángulo sería $\theta = 49.85$ (primer cuadrante)

  $< -3, 4 >$ su ángulo sería $\theta = 180 - 49.85 = 130.15$ (segundo cuadrante)

  $< -3, -4 >$ su ángulo sería $\theta = 180 + 49.85 = 229.85$ (tercer cuadrante)

  $< 3, -4 >$ su ángulo sería $\theta = 360 - 49.85 = 310.15$ (cuarto cuadrante)

  Donde se toma en cuenta en que cuadrante se encuentra el vector para así regresar un resultado útil.

- verifyCase
  - `public int verifyCase(double alfa3, double alfa4)`
  - Función para verificar en que caso cae la suma de dos abanicos complejos.
  - Parámetros:

    `alfa3` .- el primer extremo del intervalo de ángulo del segundo abanico complejo

    `alfa4` .- el segundo extremo del intervalo de ángulo del segundo abanico complejo
  - Regresa:

    el numero de caso al cual cae la suma

- part
  - `public ComplexFan[] part()`
  - Esta función parte un abanico complejo de acuerdo a su intersección con el plano cartesiano.
  - Regresa:

    el arreglo de abanicos complejos (partes) que componen el abanico complejo
  - Detalles:

    Para solucionar este problema, se definen los limites de los cuadrantes, luego se va verificando en que cuadrantes se encuentran los extremos del intervalo de ángulo, dependiendo de que cuadrantes abarca el intervalo y el valor de sus extremos se devuelve el arreglo de abanicos complejos (partes).

Hay que tener en cuenta que con intervalos de ángulo de la forma $[180, 180)$, $(67, 67]$, $[349, 349)$, etc., estos se consideraran vacíos, por lo tanto, para partir abanicos complejos de las formas:
$[a, b]\angle[\alpha_1, \alpha_1)$
$[c, d]\angle(\alpha_2, \alpha_2]$
el resultado será
$(0, 0)\angle(0, 0)$
es decir, un abanico complejo vacío.

- isEmpty
  - `public boolean isEmpty()`
  - Esta función verifica si un abanico complejo esta vacío. Implementa las consideraciones presentadas en la Sección 3.1.1, donde se especifica cuando un intervalo es vacío, tomando en cuenta que un abanico complejo esta compuesto por dos intervalos.
  - Regresa:

    verdadero si el abanico complejo esta vacío, falso en caso contrario

- print
  - `public java.lang.String toString()`
  - Esto calcula la cadena de caracteres que representa a un abanico complejo.
  - Sobre escribe:

    `toString` de la clase `java.lang.Object`
  - Regresa:

    la cadena de caracteres que representa al abanico complejo

Apéndice B

Detalles de la implementación en Mathematica

A continuación se presentan los patrones, los métodos y las funciones que se implementaron en Mathematica para la aritmética de abanicos complejos.

B.1. Intervalos de magnitud

En esta sección se muestran el patrón que representará a los intervalos de magnitud, los métodos y las funciones que se implementaron para el manejo de intervalos de magnitud.

Patrón

- `IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]`
- Para que este patrón funcione las cabeceras `IN`, `FE`, `SE`, `FEI`, `SEI`, no deben estar definidas en el kernel en ejecución, ya que si hay funciones definidas con estos nombres, se ejecutarían las funciones definidas y esto causaría la desaparición del patrón.
- Detalles
  - `IN` indica que es un intervalo
  - `FE` indica que es el primer extremo
  - `SE` indica que es el segundo extremo
  - `FEI` indica que es el límite del primer extremo
  - `SEI` indica que es el límite del segundo extremo
  - `fe_` el valor del primer extremo
  - `se_` el valor del segundo extremo
  - `fei_` el valor del límite del primer extremo
  - `sei_` el valor del límite del segundo extremo

Resumen de métodos y funciones

- INDefine
  - `INDefine[options___]`
  - Esta función es para definir un nuevo intervalo de magnitud. Puede depender de la función `INOptions`.
  - Parámetros:

    `options___` .- indica que espera 0 o varias reglas.
  - Regresa:

    Patrón que representa un intervalo de magnitud
  - Detalles:

    Para que esta función trabaje necesita de 0 a 4 reglas, una regla para el primer extremo `firstExtreme->value`, una regla para el segundo extremo `secondExtreme->value`, la regla para especificar el límite para el primer extremo `feIncluded->value` (con '[' el límite es cerrado y con '(' el límite es abierto) y la regla para especificar el límite para el segundo extremo `seIncluded->value` (con ']' el límite es cerrado y con ')' el límite es abierto). Si no se especifica alguna de las reglas esta función usa valores por defecto para cada regla omitida con ayuda de la función `INOptions`; si no se especifica ninguna regla se inicializa un intervalo en $[0, 0]$, lo cual es un intervalo NO vacío, caso contrario a la implementación en Java. Será obligación del usuario mandar valores para `fei` y `sei` válidos, es decir, para `fei` los valores esperados son '(' o '[' y para `sei` son ')' o ']'. Si el usuario manda valores diferentes a estos, no se promete un funcionamiento correcto del sistema.
- INDefineShort
  - `INDefineShort[fei_,fe_,se_,sei_]`
  - Función para definir un nuevo intervalo de una manera más corta.
  - Parámetros:

    `fei` .- el límite del primer extremo.

    `fe` .- el primer extremo.

    `se` .- el segundo extremo.

    `sei` .- el límite del segundo extremo.
  - Regresa:

    Patrón que representa un intervalo
  - Detalles:

    Es obligación del usuario mandar valores para `fei` y `sei` válidos, es decir, para `fei` los valores esperados son '(' ó '[' y para `sei` son ')' o ']'. Si el usuario manda valores diferentes a estos, no se promete un funcionamiento correcto del sistema.

- INOptions
  - `INOptions[INDefine]`
  - Función auxiliar para la función `INDefine`.
  - Parámetros:

    `INDefine` .- nombre de variable sin definir
  - Regresa:

    las reglas con valores por defecto para los extremos y límites de un intervalo
  - Detalles:

    Esta función regresa 4 reglas con valores por defecto para los extremos y los límites de un intervalo; para el primer extremo `firstExtreme->0.0`, para el segundo extremo `secondExtremo->0.0`, para el límite para el primer extremo `feIncluded->``[''` y para el límite para el segundo extremo `seIncluded->``]''`; estas reglas pueden ser usadas en caso de que la función `INDefine` no reciba alguna de las 4 reglas necesarias.
- INNormalize
  - `INNormalize[in_]`
  - Función para normalizar un intervalo de magnitud, es decir, que el primer extremo sea menor o igual que el segundo extremo.
  - Parámetros:

    `in` .- el intervalo a normalizar
- INGetFE
  - `INGetFE[IN[FE[fe_], se_, fei_, sei_]]`
  - Función para obtener el valor del primer extremo.
  - Parámetros:

    `IN[FE[fe_], se_, fei_, sei_]` .- patrón que representa a un intervalo
  - Regresa:

    el valor actual del primer extremo
- INGetSE
  - `INGetSE[IN[fe_, SE[se_], fei_, sei_]]`
  - Función para obtener el valor del segundo extremo.
  - Parámetros:

    `IN[fe_, SE[se_], fei_, sei_]` .- patrón que representa un intervalo
  - Regresa:

    el valor actual del segundo extremo

- INGetFEI
  - `INGetFEI[IN[fe_, se_, FEI[fei_], sei_]]`
  - Función para obtener el límite del primer extremo.
  - Parámetros:

    `IN[fe_, se_, FEI[fei_], sei_]` .- patrón que representa un intervalo
  - Regresa:

    el valor actual del límite para el primer extremo
- INGetSEI
  - `INGetSEI[IN[fe_, se_, fei_, SEI[sei_]]]`
  - Función para obtener el límite del segundo extremo.
  - Parámetros:

    `IN[fe_, se_, fei_, SEI[sei_]]` .- patrón que representa un intervalo
  - Regresa:

    el valor actual del límite para el segundo extremo
- INSetFE
  - `INSetFE[int_, newfe_]`
  - Función para asignar el valor del primer extremo.
  - Parámetros:

    `int` .- el intervalo a modificar

    `newfe` .- el nuevo valor del primer extremo
- INSetSE
  - `INSetSE[int_, newse_]`
  - Función para asignar el valor del segundo extremo.
  - Parámetros:

    `int` .- el intervalo a modificar

    `newse` .- el nuevo valor del segundo extremo
- INSetFEI
  - `INSetFEI[int_, newfei_]`
  - Función para asignar el límite del primer extremo.
  - Parámetros:

    `int` .- el intervalo a modificar

    `newfei` - el nuevo límite del primer extremo
  - Detalles:

    Se dejará al usuario la responsabilidad de mandar un valor válido, ya sea '(' ó '[' para newfei. No se prometerá un buen funcionamiento si el usuario manda un valor no válido.

- INSetSEI
  - `INSetSEI[int_, newsei_]`
  - Función para asignar el límite del segundo extremo.
  - Parámetros:

    `int` .- el intervalo a modificar

    `newsei` .- el nuevo límite del segundo extremo
  - Detalles:

    Se dejará al usuario la responsabilidad de mandar un valor válido, ya sea ')' ó ']' para newsei. No se prometerá un buen funcionamiento si el usuario manda un valor no válido.
- INByConstant
  - `INByConstant[in_, k_]`
  - Función para multiplicar un intervalo por una constante.
  - Parámetros:

    `in` .- el intervalo a afectar

    `k` .- el factor constante
- INEmptyQ
  - `INEmptyQ[IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]]`
  - Función para verificar si un intervalo esta vacío.
  - Parámetros:

    `IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]` .- patrón que representa a un intervalo
  - Regresa:

    si el intervalo es vacío
- INIntersection
  - `INIntersection[in1_, in2_]`
  - Función para calcular la intersección de dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo para la intersección

    `in2` .- el segundo intervalo para la intersección
  - Regresa:

    el resultado de la intersección
- INUnionList
  - `INUnionList[mis_]`
  - Función para unir una lista de intervalos de magnitud.

  - Parámetros:

    `mis` .- la lista de intervalos de magnitud

  - Regresa:

    el resultado de la unión

- INNegation

  - `INNegation[in_]`
  - Función para calcular la negación de un intervalo.
  - Parámetros:

    `in` .- el intervalo a negar

  - Regresa:

    el resultado de la negación

- INAddition

  - `INAddition[IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]],`
    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]]`
  - Función para sumar dos intervalos.
  - Parámetros:

    `IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]]` .- el primer intervalo de la suma

    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]` - el segundo intervalo de la suma

  - Regresa:

    el resultado de la suma

- INProduct

  - `INProduct[IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]],`
    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]]`
  - Función para multiplicar dos intervalos.
  - Parámetros:

    `IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]]` .- el primer intervalo del producto

    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]` .- el segundo intervalo del producto

  - Regresa:

    el resultado del producto

- INSubtraction

  - `INSubtraction[IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]],`
    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]]`
  - Función para restar dos intervalos

  - Parámetros:

    `IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]]` .- el primer intervalo para la resta

    `IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]` .- el segundo intervalo para la resta

  - Regresa:

    el resultado de la resta

- INDivision
  - `INDivision[IN[FE[fe1_],SE[se1_],FEI[fei1_],SEI[sei1_]], IN[FE[fe2_],SE[se2_],FEI[fei2_],SEI[sei2_]]]`
  - Función para dividir dos intervalos.
  - Parámetros:

    `in1` .- el primer intervalo de la división

    `in2` .- el segundo intervalo de la división
  - Regresa:

    el resultado de la división
  - Detalles:

    En caso de que 'd' sea cero o 'c' sea cero, imprime un mensaje de 'División por cero!'.

- INPrint
  - `INPrint[in_]`
  - Función para imprimir la cadena de caracteres que representa un intervalo.
  - Parámetros:

    `in` - el intervalo a imprimir

- INStringForm
  - `INStringForm[IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]]`
  - Función que regresa la representación de un intervalo en cadena de caracteres.
  - Parámetros:

    `IN[FE[fe_],SE[se_],FEI[fei_],SEI[sei_]]` .- el intervalo a usar
  - Regresa:

    la cadena de caracteres que representa el intervalo

B.2. Intervalos de ángulo

A continuación se presentan los métodos y las funciones para el manejo de intervalos de ángulo. Cabe mencionar que se usará el mismo patrón para un intervalo de ángulo, que para un intervalo de magnitud, lo que los diferenciará son los métodos y las funciones que se apliquen sobre ellos, y los nombres de las variables que los representen.

Patrón

- Mismo que la de un intervalo de magnitud

Resumen de métodos y funciones

- AINormalize
  - `AINormalize[ai_]`
  - Esta función normaliza los extremos de un intervalo de ángulo.
  - Parámetros:

    `ai` .- el intervalo de ángulo a normalizar
- AIAddition
  - `AIAddition[ai1_, ai2_]`
  - Función para sumar dos intervalos de ángulo, depende de la función `INAddition`.
  - Parámetros:

    `ai1` .- el primer operando para la suma
    `ai2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- AISubtraction
  - `AISubtraction[ai1_, ai2_]`
  - Función para restar dos intervalos de ángulo, depende de la función `INSubtraction`.
  - Parámetros:

    `ai1` .- el primer operando de la resta
    `ai2` .- el segundo operando de la resta
  - Regresa:

    el resultado de la resta
- AIAdd180
  - `AIAdd180[ai_]`
  - Esta función suma 180 grados al intervalo de ángulo que recibe. Depende de la función `AIAddition`.
  - Parámetros:

    `ai` .- el intervalo de ángulo a utilizar
  - Regresa:

    el resultado de sumar 180 grados al intervalo de ángulo

- AIVerifyCase0360Q
  - `AIVerifyCase0360Q[ai_]`
  - Esta función verifica si un intervalo de ángulo abarca los 4 cuadrantes, es decir, es igual a [0, 360].
  - Parámetros:

    `ai` .- el intervalo de ángulo a utilizar
  - Regresa:

    si el intervalo de ángulo abarca los 360 grados
- AIUnionList
  - `AIUnionList[res_]`
  - Función para unir una lista de intervalos de ángulo.
  - Parámetros:

    `res` .- la lista de intervalos de ángulo a unir
  - Regresa:

    el resultado de la unión
- AIVerifyCase0360ListQ
  - `AIVerifyCaseO360ListQ[lais_]`
  - Función para verificar si una lista de intervalos de ángulo en conjunto abarcan los 360 grados.
  - Parámetros:

    `lais` .- la lista de intervalos de ángulo
  - Regresa:

    si la lista de intervalos en conjunto abarcan los 360 grados

B.3. Abanicos complejos

A continuación se muestran el patrón que representará a los abanicos complejos, los métodos y las funciones para el manejo de abanicos complejos.

Patrón

- `CF[mi_,ai_]`
- Para que este patrón funcione la cabecera `CF` no debe estar definida en el kernel en ejecución con alguna funcionalidad; ya que si hay una función definida con este nombre, se ejecutaría la función definida y esto causaría la desaparición del patrón.
- Detalles
  - `CF` indica que es un abanico complejo
  - `mi_` es el patrón del intervalo de magnitud
  - `ai_` es el patrón del intervalo de ángulo

Resumen de métodos y funciones

- CFDefine
  - `CFDefine[options___]`
  - Función para definir un abanico complejo, puede depender de la función `CFOptions`.
  - Parámetros:

    `options___` .- indica que puede recibir 0 o más reglas
  - Regresa:

    patrón que representa un abanico complejo
  - Detalles:

    Esta función puede ser llamada con 0 a 2 reglas, una regla de la forma `magnitudeInterval->value` para especificar el valor del intervalo de magnitud, otra regla de la forma `angleInterval->value` para especificar el valor que tendrá el intervalo de ángulo. Si no se especifica alguna regla, esta función usa un valor por defecto por cada regla omitida.
- CFDefineShort
  - `CFDefineShort[MI[fei1_,fe1_,se1_,sei1_],AI[fei2_,fe2_,se2_,sei2_]]`
  - Función para definir un abanico complejo de una manera más corta.
  - Parámetros:

    `fei1` .- el límite para el primer extremo del intervalo de magnitud

    `fe1` .- el primer extremo del intervalo de magnitud

    `se1` .- el segundo extremo del intervalo de magnitud

    `sei1` .- el límite para el segundo extremo del intervalo de magnitud

    `fei2` .- el límite para el primer extremo del intervalo de ángulo

    `fe2` .- el primer extremo del intervalo de ángulo

    `se2` .- el segundo extremo del intervalo de ángulo

    `sei2` .- el límite para el segundo extremo del intervalo de ángulo
  - Regresa:

    patrón que representa un abanico complejo
  - Detalles:

    Es obligación del usuario mandar valores válidos para fei1, sei1, fei2 y sei2. Para fei1 y fei2 es '(' ó '['. Para sei1 y sei2 es ')' ó ']'.
- CFOptions
  - `CFOptions[CFDefine]`
  - Función auxiliar para la función `CFDefine`.
  - Parámetros:

    `CFDefine` .- nombre de variable sin definir
  - Regresa:

    las reglas con valores por defecto para definir un abanico complejo

  - Detalles:

    Esta función regresa 2 reglas con valores por defecto, una para el intervalo de magnitud y otra para el intervalo de ángulo, que pueden ser usadas en caso de que la función `CFDefine` no reciba alguna de las 2 reglas necesarias.

- CFGetMI
  - `CFGetMI[CF[mi_,ai_]]`
  - Función para obtener el intervalo de magnitud.
  - Parámetros:

    `CF[mi_,ai_]` .- patrón que representa a un abanico complejo
  - Regresa:

    el valor actual del intervalo de magnitud
- CFGetAI
  - `CFGetAI[CF[mi_,ai_]]`
  - Función para obtener el intervalo de ángulo.
  - Parámetros:

    `CF[mi_,ai_]` .- patrón que representa a un abanico complejo
  - Regresa:

    el valor actual del intervalo de ángulo
- CFSetMI
  - `CFSetMI[cf_, newmi_]`
  - Función para asignar un nuevo valor al intervalo de magnitud.
  - Parámetros:

    `cf` .- el abanico complejo a modificar

    `newmi` .- el nuevo valor para el intervalo de magnitud
- CFSetAI
  - `CFSetAI[cf_, newai_]`
  - Función para asignar un nuevo valor al intervalo de ángulo.
  - Parámetros:

    `cf` .- el abanico complejo a modificar

    `newai` .- el nuevo valor para el intervalo de ángulo
- CFNegation
  - `CFNegation[cf_]`
  - Función para calcular la negación de un abanico complejo.
  - Parámetros:

    `cf` .- el abanico complejo del cual se va a calcular la negación

  - Regresa:

    el resultado de la negación

- CFProduct
  - `CFProduct[cf1_, cf2_]`
  - Función para calcular el producto entre dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando para el producto
    `cf2` .- el segundo operando para el producto
  - Regresa:

    el resultado del producto

- CFDivision
  - `CFDivision[cf1_, cf2_]`
  - Función para calcular la división entre dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando de la división
    `cf2` .- el segundo operando de la división
  - Regresa:

    el resultado de la división

- CFSubtraction
  - `CFSubtraction[cf1_, cf2_]`
  - Función para calcular la resta entre dos abanicos complejos, depende de la función `CFNegation` y de la función `CFAddition`.
  - Parámetros:

    `cf1` .- el primer operando de la resta
    `cf2` .- el segundo operando de la resta
  - Regresa:

    el resultado de la resta

- CFAddition
  - `CFAddition[cf1_, cf2_]`
  - Función para calcular la suma de dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando de la suma
    `cf2` .- el segundo operando de la suma
  - Regresa:

    el resultado de la suma

- CFUnionRes
  - `CFUnionRes[acf_]`
  - Función para realizar la unión de una lista de abanicos complejos. Depende de las funciones `INUnionList` y `CFUnionAIs`.
  - Parámetros:

    `acf` .- la lista de abanicos complejos
  - Regresa:

    el resultado de la unión
- CFUnionAIs
  - `CFUnionAIs[acf_]`
  - Función para realizar la unión de los intervalos de ángulo provenientes de una lista de abanicos complejos. Depende de la función `AIUnionList`.
  - Parámetros:

    `acf` .- la lista de abanicos complejos
  - Regresa:

    el resultado de la unión de los intervalos de ángulo
- CFVerifyCase
  - `CFVerifyCase[alfa3_, alfa4_]`
  - Función para verificar en que caso cae la suma de dos abanicos complejos.
  - Parámetros:

    `alfa3` .- el primer extremo del intervalo de ángulo del segundo abanico complejo

    `alfa4` .- el segundo extremo del intervalo de ángulo del segundo abanico complejo
  - Regresa:

    el número de caso al cual cae la suma
- CFPart
  - `CFPart[cf_]`
  - Esta función parte un abanico complejo de acuerdo a su intersección con el plano cartesiano.
  - Parámetros:

    `cf` .- el abanico complejo a partir
  - Regresa:

    la lista de partes que componen el abanico complejo

- CFAdditionCase1
  - `CFAdditionCase1[cf1_, cf2_]`
  - Función que implementa el algoritmo para el caso 1 para la adición de dos abanicos complejos.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- CFAdditionCase2
  - `CFAadditionCase2[cf1_, cf2_]`
  - Función para la suma de dos abanicos complejos caso 2. Depende de las funciones `CFMagnitudeCase2` y `CFAngleCase2`.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- CFMagnitudeCase2
  - `CFMagnitudeCase2[cf1_ cf2_]`
  - Función que implementa el algoritmo para calcular el intervalo de magnitud resultante en la suma de dos abanicos complejos para el caso 2.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de magnitud resultante
- CFAngleCase2
  - `CFAngleCase2[cf1_, cf2_]`
  - Función que implementa el algoritmo para calcular el intervalo de ángulo resultante en la suma de dos abanicos complejos para el caso 2.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de ángulo resultante

- CFAdditionCase3
  - `CFAdditionCase3[cf1_, cf2_]`
  - Función para la suma de dos abanicos complejos para el caso 3. Depende de las funciones `CFMagnitudeCase3` y `CFAngleCase3`.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el resultado de la suma
- CFMagnitudeCase3
  - `CFMagnitudeCase3[cf1_, cf2_]`
  - Función que implementa el algoritmo para calcular el intervalo de magnitud resultante en la suma de dos abanicos complejos para el caso 3.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de magnitud resultante
- CFAngleCase3
  - `CFAngleCase3[cf1_, cf2_]`
  - Función que implementa el algoritmo para calcular el intervalo de ángulo resultante en la suma de dos abanicos complejos para el caso 3.
  - Parámetros:

    `cf1` .- el primer operando para la suma

    `cf2` .- el segundo operando para la suma
  - Regresa:

    el intervalo de ángulo resultante
- CFAngleFixer
  - `CFAngleFixer[x_, y_]`
  - Función para corregir el ángulo resultante, al calcular el ángulo de un vector.
  - Parámetros:

    `x` .- la componente en x del vector

    `y` .- la componente en y del vector
  - Regresa:

    el ángulo resultante

- CFPrint
  - `CFPrint[cf_]`
  - Esto imprime la cadena de caracteres que representa a un abanico complejo.
  - Parámetros:

    `cf` .- el abanico complejo a imprimir
- CFStringForm
  - `CFStringForm[cf_]`
  - Esto calcula la cadena de caracteres que representa a un abanico complejo.
  - Parámetros:

    `cf` .- el abanico complejo a usar
  - Regresa:

    la cadena de caracteres que representa a un abanico complejo
- CFBeforePlot
  - `CFBeforePlot[cf_]`
  - Esta función prepara un abanico complejo para dibujarlo.
  - Parámetros:

    `cf` .- el abanico complejo a dibujar
  - Regresa:

    la lista de formas geométricas que se utilizaran para dibujar el abanico complejo
- CFPlot
  - `CFPlot[cf_]`
  - Función para dibujar una lista de abanicos complejos. Depende de la función `CFBeforePlot`.
  - Parámetros:

    `cf` .- la lista de abanicos complejos a dibujar
- CFAdditionPlot
  - `CFAdditionPlot[cf1_, cf2_, cfr_]`
  - Función para dibujar los operandos de la suma de dos abanicos complejos, junto con el resultado. Depende de la función `CFAdditionPlotAux`. Esta función es para comprobar el resultado de una adición de manera gráfica, ya que incluye proyecciones del primer operando con respecto al segundo y viceversa.
  - Parámetros:

    `cf1` .- el primer operando

    `cf2` .- el segundo operando

    `cfr` .- el resultado de la suma

- CFAdditionPlotAux
  - `CFAdditionPlotAux[p_, cf_]`
  - Función auxiliar para la función `CFAdditionPlot`. Calcula la lista de formas geométricas que representarán a un abanico complejo con respecto al punto que recibe.
  - Parámetros:
    `p` .- el punto con respecto al que se quiere dibujar un abanico complejo
    `cf` .- el abanico complejo a dibujar
- CFProjectionPlot
  - `CFProjectionPlot[cf1_, cf2_]`
  - Función para dibujar la proyección de un abanico complejo con respecto a otro y viceversa.
  - Parámetros:
    `cf1` .- el primer abanico complejo
    `cf2` .- el segundo abanico complejo
- CFSubtractionPlot
  - `CFSubtractionPlot[cf1_, cf2_, cfr_]`
  - Función para dibujar los operandos de una resta de dos abanicos complejos, junto con el resultado. Depende de la función `CFAdditionPlot`. Esta función es para comprobar el resultado de una sustracción de manera gráfica, ya que incluye proyecciones del primer operando con respecto al negado del segundo y viceversa.
  - Parámetros:
    `cf1` .- el primer operando de la resta
    `cf2` .- el segundo operando de la resta
    `cfr` .- el resultado de la resta
- CFNegationPlot
  - `CFNegationPlot[cf_, cfr_]`
  - Función para dibujar un abanico complejo y el resultado de su negación. Gráfica a `cf` de color azul y a `cfr` de rojo. Depende de la función `CFBeforePlot`.
  - Parámetros:
    `cf` .- el abanico complejo original
    `cfr` .- el resultado de la negación de `cf`
- CFOperationPlot
  - `CFOperationPlot[cf1_, cf2_, cfr_]`
  - Función para dibujar dos abanicos complejos y el resultado de la operación realizada entre estos. Gráfica a `cf1` de color azul, a `cf2` de verde y a `cfr` de rojo. Depende de la función `CFBeforePlot`.
  - Parámetros:
    `cf1` .- el primer operando
    `cf2` .- el segundo operando
    `cfr` .- el resultado de la operación entre estos

Bibliografía

[Antonov, 2015] Antonov, A. (2015). Implementation of object-oriented programming design patterns in mathematica. *GitHub,WordPress.com*.

[Exchange, 2016] Exchange, S. (2016). *How can I implement object oriented programming in Mathematica?* mathematica.stackexchange.com.

[Flores, 1997] Flores, J. J. (1997). Reasoning about linear circuits in sinusoidal steady state. *Ph. D. thesis, University of Oregon.*

[Flores, 1998] Flores, J. J. (1998). Complex fans: A representation for vectors in polar form with interval attributes. *Asociation for Computing Machinery Inc.*

[machz1988, 2016a] machz1988 (2016a). *ComplexFans*. https://github.com/machz1988/ComplexFans.

[machz1988, 2016b] machz1988 (2016b). *ComplexFansMathematica*. https://github.com/machz1988/ComplexFansMathematica.

[Robert Resnick, 1966] Robert Resnick, D. H. (1966). *Physics Part I.* John Wiley & Sons, Inc., 2 edition.

[William H. Hayt, 2007] William H. Hayt, Jr., J. E. K. S. M. D. (2007). *Análisis de circuitos en ingeniería*. McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V., 7 edition.

[Wolfram, 2016a] Wolfram (2016a). *Introduction to J/Link*. reference.wolfram.com.

[Wolfram, 2016b] Wolfram (2016b). *Language Overview*. reference.wolfram.com.