Facultad
de
Ciencias

Continuación homotópica para la resolución de sistemas de polinomios

(*Homotopy continuation for solving polynomial systems*)

Trabajo de Fin de Grado
para acceder al
Grado en Matemáticas

Autor: Jaime Vía Santoveña.
Director: Carlos Beltrán Álvarez.
Junio - 2025

Resumen

Este Trabajo de Fin de Grado está dedicado a la explicación del método de continuación homotópica, haciendo especial énfasis en su aplicación a la resolución de sistemas de ecuaciones polinomiales cuadrados.

En primer lugar se definen los sistemas de polinomios y se proporcionan diversas aplicaciones actuales de los mismos. Seguidamente se introducen los fundamentos teóricos de la continuación homotópica, detallando en especial el método de seguimiento de caminos o *path-tracking* con homotopías de combinación convexa. Se expone un algoritmo general para su desarrollo.

A continuación se presenta un programa informático desarrollado por el autor, diseñado para poder modelar cualquier sistema de ecuaciones polinomiales y probar los diversos métodos de seguimiento de caminos. Este desarrollo práctico permite una experimentación controlada así como una evaluación de los métodos usando los datos que se obtienen.

Finalmente se discute la eficiencia y el éxito de cuatro algoritmos de seguimiento de caminos específicos, los cuales son probados y comparados mediante el uso de varios sistemas predefinidos.

Palabras Clave: Continuación homotópica, sistemas de ecuaciones polinomiales, seguimiento de caminos, homotopía, algorítmica, métodos numéricos

Abstract

This Final Project is dedicated to explaining the homotopy continuation method, with a special emphasis on its application to solving square polynomial equation systems.

Firstly, polynomial systems are defined, and various current applications of these systems are provided. Next, the theoretical foundations of homotopy continuation are introduced, detailing in particular the path-tracking method with convex combination homotopies. A general algorithm for its development is also presented.

Then, a computer program developed by the author is provided. This program is designed to model any polynomial system of equations, and to test various path-tracking methods. This practical development allows for controlled experimentation and an evaluation of the methods using the data obtained.

Finally, the efficiency and success of four specific path-tracking algorithms are discussed. These algorithms are tested and compared using a collection of predefined systems.

Key Words: Homotopy continuation, systems of polynomial equations, path-tracking, homotopy, algorithmics, numerical methods

Índice general

Capítulo 1

Introducción

En este primer capítulo se realizará una breve introducción a los sistemas de polinomios, su historia y definición formal, así como una breve aproximación al propio método numérico de continuación homotópica.

1.1. Sistemas de polinomios: historia y usos actuales

Los polinomios siempre han sido una parte recurrente en nuestras vidas, ya que gran parte de los problemas que nos afectan a diario pueden modelarse y resolverse como la solución de un sistema $F(\mathbf{x}) = 0$, donde $F = (p_1, \ldots, p_m)$ es un conjunto de polinomios en $\mathbf{x} = (x_1, \ldots, x_n)$.
Pongamos el siguiente ejemplo de un sistema lineal con polinomios de grado 1:

Ejemplo 1.1.0.1 *:*
- *3 manojos de paja de arroz de alta calidad, 2 manojos de paja de arroz de calidad media y 1 manojo de paja de arroz de baja calidad producen 39 unidades de arroz.*
- *2 manojos de paja de arroz de alta calidad, 3 manojos de paja de arroz de calidad media y 1 manojo de paja de arroz de baja calidad producen 34 unidades de arroz.*
- *1 manojo de paja de arroz de alta calidad, 2 manojos de paja de arroz de calidad media y 3 manojos de paja de arroz de baja calidad producen 26 unidades de arroz.*
- *¿Cuántas unidades de arroz pueden producir respectivamente la paja de arroz de alta, media y baja calidad?*

$$\begin{cases} 3a + 2m + b = 39 \\ 2a + 3m + b = 34 \\ a + 2m + 3b = 26 \end{cases}$$

Tanto este problema como su solución aparecen en el libro *Los Nueve Capítulos sobre Arte Matemático* (173 d. C.), en el llamado capítulo del *Fangcheng* [1]. Éste plantea diversos sistemas de ecuaciones lineales, y los resuelve utilizando un método indistinguible al ahora conocido como método de Gauss, formalizado en *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium* (1809) (Teoría del movimiento de los cuerpos celestes que giran alrededor del Sol en secciones cóni-

cas [2]).
Pongamos otro ejemplo, pero esta vez de un sistema de polinomios de grado superior a uno:

Ejemplo 1.1.0.2 *:*
- *He cuadrado la diferencia entre la longitud y la anchura.*
- *He sustraído [esta cantidad] de la superficie: 8,2.*
- *He sumado la longitud y la anchura: 50.*

$$\begin{cases} la - (l-a)^2 = 8.2 \\ l + a = 50 \end{cases}$$

Este sistema se encuentra en el segundo problema de la tablilla YBC 6504, escrita en cuneiforme, y que data de entre el 1800-1600 a. C. Fue traducida por Jens Høyrup quien también logró descifrar que las soluciones propuestas eran de tipo geométrico [3].

Hoy en día, muchos problemas matemáticos y de ingeniería requieren resolver sistemas de ecuaciones polinomiales. Aunque la geometría algebraica o la teoría de Galois han permitido entender mejor su estructura teórica (como el número y tipo de soluciones o sus propiedades geométricas), en la práctica muchos sistemas no lineales complejos suelen presentar dificultades debido a su tamaño o la imposibilidad de su resolución simbólica. Un ejemplo clásico es la imposibilidad de hallar raíces exactas de polinomios univariables de grado ≥ 5 mediante radicales (por el Teorema de Abel-Ruffini).

Sin embargo, aunque no siempre se puedan expresar fácilmente las soluciones de forma analítica, sí se pueden localizar numéricamente y estudiar su comportamiento. Entre los métodos más robustos para esto destaca el método de **continuación homotópica**, que transforma gradualmente un sistema simple (con soluciones conocidas) en el sistema objetivo mediante una deformación continua (homotopía).
Algunos casos de uso que se beneficiarían de este método serían por ejemplo:

- **Robótica:** Para calcular la posición en el espacio del efector de un robot, así como calcular qué parámetros hace falta modificar para desplazarlo de un sitio a otro se tienen que resolver sistemas de ecuaciones polinomiales no lineales [4][5]. Uno de los casos más estudiados son los de estabilización de un robot del tipo plataforma de Stewart [6].

- **Medicina:** Para estimar el estado de una enfermedad infecciosa [7] o para el doblado de proteínas (ya que normalmente las interacciones entre átomos y moléculas se pueden describir mediante funciones de energia potencial polinomiales). Un proyecto que se podría beneficiar de estos métodos sería el proyecto de computación distribuida "*Folding@home*"[1] que identifica las estructuras moleculares de diversas proteínas.

- **Optimización:** Para problemas de economía, biología, física... La optimización de problemas no lineales aparece de forma recurrente en nuestras vidas [8]. Un ejemplo directamente polinomial puede ser la maximización de la utilidad del consumidor con restricciones presupuestarias. En [9] se puede ver un ejemplo sencillo.

[1] https://foldingathome.org/

- **Identificación de métodos no lineales:** Mediante la entrada y salida de diversos tipos de métodos no lineales (como por ejemplo los de una red neuronal) se puede intentar aproximar el modelo utilizado mediante el uso de polinomios [10].

A pesar de esto, especialmente para sistemas con infinitas soluciones, los métodos numéricos pueden fallar. En esos casos, técnicas de álgebra computacional, como el algoritmo de Buchberger [11] para calcular bases de Gröbner, permiten:

- Reducir el sistema a una forma triangular.
- Identificar variedades irreducibles (Punto 1.2.4).
- Distinguir las soluciones aisladas de variedades de dimensión superior.

Es por ello que, en la vida real, métodos de cálculo simbólico y numérico se usan conjuntamente para la resolución de problemas de manera mucho más eficiente. No obstante por regla general los métodos simbólicos son mucho más lentos que los numéricos.

Figura 1.1: Sistema Ampelmann de estabilización que usa una plataforma de Stewart (los brazos se pueden identificar, en negro). Imagen por Kees Torn.

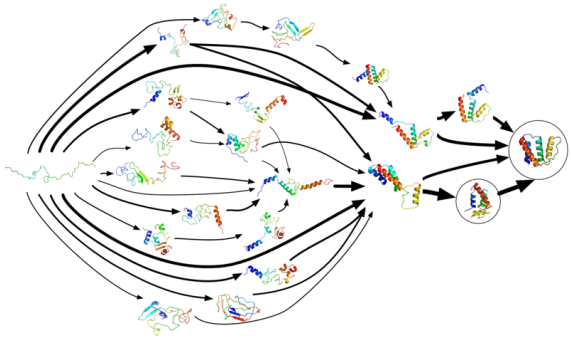

Figura 1.2: Doblado de proteína ACBP gracias a Folding@home. Imagen por Vincent Voelz.

1.2. Monomios, polinomios y sistemas

Para garantizar una implementación rigurosa en los programas desarrollados en este trabajo, es necesario formalizar los conceptos básicos de monomio, polinomio,

sistemas de polinomios y variedades desde una perspectiva **operacional** y utilitarista (enfocada en su representación y manipulación algorítmica). Se trabajará sobre $\mathbb{C}$.

1.2.1. Monomios

Definición 1.2.1.1 (Monomio) *Un monomio m en n variables sobre $\mathbb{C}$ es una expresión algebraica de la forma:*

$$m(x_1, x_2, \ldots, x_n) = a \cdot x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}$$

donde:

- $a \in \mathbb{C}$ *es el coeficiente.*
- $x_1, \ldots, x_n$ *son las variables algebraicas.*
- $\alpha_1, \ldots, \alpha_n \in \mathbb{N}$ *son los exponentes correspondientes a las variables.*

Definición 1.2.1.2 (Grado total de un monomio) *El grado total de un monomio se corresponde a la suma de sus exponentes:*

$$deg(m) = \alpha_1 + \cdots + \alpha_n$$

A partir de ahora, se usará la notación $\mathbf{x} = (x_1, \ldots, x_n)$.

1.2.2. Polinomios

Definición 1.2.2.1 (Polinomio) *Un polinomio en n variables sobre $\mathbb{C}$ es una combinación lineal finita de monomios en dichas variables. Se expresa por:*

$$P(\mathbf{x}) = \sum_{k=1}^{N} m_k(\mathbf{x})$$

con $m_k(x_1, \ldots, x_n)$ un monomio como los antes definidos y N el número de términos del polinomio.

Definición 1.2.2.2 (Grado total de un polinomio) *El grado total de un polinomio, a partir de ahora grado, es el máximo de los grados de los monomios que lo componen.*

$$deg(P) = máx\{deg(m_k) : k \in \{1, \ldots, N\}\}$$

Definición 1.2.2.3 ($\mathbb{C}[\mathbf{x}]$) *El anillo $\mathbb{C}[\mathbf{x}] = \mathbb{C}[x_1, \ldots, x_n]$ representa el conjunto de todos los polinomios de múltiples variables con coeficientes en $\mathbb{C}$ y en $\mathbf{x} = x_1, \ldots, x_n$. Formalmente:*

$$\mathbb{C}[\mathbf{x}] = \left\{ \sum^{N} m(\mathbf{x}) \mid N \in \mathbb{N} \right\}$$

con $m(\mathbf{x})$ monomios en $\mathbf{x}$ de coeficientes complejos.

Definición 1.2.2.4 (Propiedades de los polinomios) *Si* $P \in \mathbb{C}[\mathbf{x}]$ *y* $Q \in \mathbb{C}[\mathbf{x}]$*, entonces:*

- $-P \in \mathbb{C}[\mathbf{x}]$
- $P+Q \in \mathbb{C}[\mathbf{x}]$
- $PQ \in \mathbb{C}[\mathbf{x}]$

Estas propiedades propias del anillo conmutativo serán útiles para la realización del método de continuación homotópica.

1.2.3. Sistemas

Definición 1.2.3.1 (Sistema de polinomios) *Un sistema de polinomios en* n *variables sobre* $\mathbb{C}$ *es un conjunto finito de* m *polinomios en* $\mathbb{C}[\boldsymbol{x}]$ *de la manera:*

$$F(\mathbf{x}) = \begin{cases} P_1(\mathbf{x}) \\ P_2(\mathbf{x}) \\ \dots \\ P_m(\mathbf{x}) \end{cases}$$

donde cada $P_i(\mathbf{x})$ *con* $i \in \{1, \dots, m\}$ *pertenece a* $\mathbb{C}[\boldsymbol{x}]$*. Si* $m = n$*, se dice que el sistema es cuadrado. Si además tiene solución única, compatible determinado. Por otro lado, se denomina homogéneo, si todos los monomios que conforman los polinomios del sistema tienen el mismo grado.*
El sistema también se puede interpretar como una función vectorial multivariable del tipo $F(\mathbf{x}) = (P_1(\mathbf{x}), \dots, P_m(\mathbf{x}))$

1.2.4. Conjunto de soluciones de un sistema

Las soluciones de un polinomio con una variable son ampliamente conocidas, y se pueden representar de manera numérica con una lista de puntos aproximados. Pero,

Una variable	**Varias variables**
1 polinomio	m polinomios
1 variable	n variables
Raíces son puntos	Raíces son puntos, curvas, superficies...
Raíces con multiplicidad	Raíces con multiplicidad
Factorización: $c \cdot \prod_i (x - a_i)^{\mu_i}$	Descomposición en factores irreducibles

Cuadro 1.1: Diferencias entre sistemas de polinomios

como se ilustra en el Cuadro 1.1, la situación es más compleja para los sistemas de polinomios multivariables. Se introduce entonces el concepto de variedades.

Definición 1.2.4.1 *Dado un sistema de polinomios F, su variedad algebraica es el conjunto de puntos donde todos los polinomios se anulan.*

$$V(F) = \{\boldsymbol{a} \in \mathbb{C}^n | f(\boldsymbol{a}) = 0, \forall f \in F\}.$$

Una variedad es irreducible si no puede escribirse como unión $V = V_1 \cup V_2$ *de variedades propias con* $V_1 \cap V_2 = \emptyset$ *y* $V_1, V_2 \neq V$

Se utilizará la notación $\{\mathbf{x} : F(\mathbf{x}) = 0\}$, $V(F)$ y $F^{-1}(0)$ indistintamente a partir de ahora.

Para el siguiente teorema se definirá el espacio proyectivo de los complejos, $\mathbb{CP}^n = \{\text{Rectas complejas que pasan por el origen en } \mathbb{C}^{n+1}\}$, de una forma apta para lo que representa este trabajo. En resumidas cuentas es una herramienta que permite trabajar con puntos en el infinito.

Definición 1.2.4.2 ($\mathbb{CP}^n$) *Se puede definir el* ***espacio proyectivo complejo*** *como todas las direcciones en $\mathbb{C}^{n+1}$ tal que la última coordenada, z_{n+1} sirve para representar los puntos en el infinito. Formalmente:*

$$\mathbb{CP}^n = \{[z_1, \ldots, z_n, z_{n+1}] \mid (z_1, \ldots, z_n, z_{n+1}) \in \mathbb{C}^{n+1} \backslash \{\boldsymbol{0}\}\} / \sim,$$

donde:

- $[z_1, \ldots, z_n, z_{n+1}] \sim [\lambda z_1, \ldots, \lambda z_n, \lambda z_{n+1}]$ *para todo* $\lambda \in \mathbb{C}\backslash\{0\}$.

con la convención de que:

- *Si $z_{n+1} \neq 0$, (normalizar para que sea 1) el punto es afín y corresponde a un punto en $\mathbb{C}^n$.*
- *Si $z_{n+1} = 0$, el punto está en el infinito.*

Se mostrará un ejemplo para ver de qué se habla:

Ejemplo 1.2.4.3 *Se considera el siguiente sistema en $\mathbb{C}^2$:*

$$F(x, y) = \begin{cases} x = 1 \\ x = 2 \end{cases}$$

Aparentemente el sistema no cuenta con soluciones ya que definen planos paralelos entre sí, pero se puede normalizar en $\mathbb{CP}^2$ definiendo $x = \frac{X}{Z}$ e $y = \frac{Y}{Z}$ con $Z \neq 0$. El sistema entonces se podría escribir de la manera:

$$\begin{cases} X = Z \\ X = 2Z \end{cases}$$

en el que se pueden distinguir los casos: $Z \neq 0$, en el que nos encontramos el mismo problema de antes, y $Z = 0$, las soluciones en el infinito.
En este caso, $X = 0$ también, por lo que sólo varía la Y, y se tiene que el sistema tiene solución en $[0, 1, 0]$ (recta del infinito/del horizonte).

Teorema 1.2.4.4 (Teorema de Bézout) *Para un sistema cuadrado de n ecuaciones polinomiales en n variables, donde los polinomios tienen grados $d_1, \ldots, d_n$ respectivamente, el número de soluciones en $\mathbb{CP}^n$ (contando multiplicidades) está acotado por el producto de los grados $\prod_{i=1}^{n} d_i$, siempre que las ecuaciones no tengan componentes comunes (la variedad es de dimensión 0).*

Hay que tener un especial cuidado a la hora de aplicar el Teorema 1.2.4.4, como se ilustrará en los siguientes ejemplos:

Ejemplo 1.2.4.5 (Componentes comunes) *Se considera el siguiente sistema:*

$$F(x,y) = \begin{cases} x^3 - y^3 \\ x^2 - y^2 \end{cases}$$

Según Bézout parece tener 6 soluciones $deg(x^3 - y^3) \times deg(x^2 - y^2) = 6$*, pero factorizando ambos:* $x^3 - y^3 = (x-y)(x^2 + xy + y^2)$ *así como* $x^2 - y^2 = (x-y)(x+y)$*, cuentan con una componente común de dimensión 1, que es su solución:* $x = y$*. La solución que resta es* $(0,0)$*, al sustituir la otra solución del polinomio inferior,* $y = -x$*, en el polinomio superior, que ya está en la recta* $x = y$*.*

Ejemplo 1.2.4.6 (Multiplicidades) *Se considera el siguiente sistema:*

$$F(x,y) = \begin{cases} (x-1)^2 + (y-1)^2 \\ (x-1)^3 \end{cases}$$

El sistema cuenta con una solución única $(1,1)$ *con multiplicidad 2 para el primer polinomio, y 3 para el segundo. La multiplicidad total según Bézout es 6, a pesar de que hay un único punto solución.*

Estos ejemplos advierten sobre los peligros de utilizar métodos numéricos o algebraicos sin pensar sobre el propio problema.

1.3. Introducción al método de continuación homotópica

El método de continuación homotópica es una técnica numérica que sirve para resolver sistemas de ecuaciones (en nuestro caso polinomiales y cuadrados) siempre que cumplan ciertas condiciones relativamente laxas. Es un método mucho más seguro que otros como Newton-Raphson ya que a diferencia de éste no necesita una buena aproximación inicial, sino que lo hace de forma global "deformando el sistema" gracias a la equivalencia homotópica de sus polinomios.

1.3.1. Equivalencia homotópica de polinomios de mismo grado

Para que el método funcione, se debe garantizar que exista una homotopía entre 2 polinomios del mismo grado y coeficientes complejos. En este apartado se da una justificación de por qué esto es así.

Sea $\mathbb{C}[x_1, \ldots, x_k]$ el anillo de los polinomios en k variables, y grado total $\leq n$. Su espacio de coeficientes es $\mathbb{C}^N$, con $N = \binom{n+k}{k}$. Esto se debe a que el número de coeficientes de un polinomio con las características antes mencionadas es a lo sumo N.

Este valor de N se obtiene al resolver un problema de **permutaciones con repetición**, a menudo visualizado con el método de "puntos y barras" (o bolas y urnas) [12]. Cada "punto" representa una unidad de exponente. Para un monomio,

se distribuyen estos puntos entre las k variables $x_1, \ldots, x_k$. Sin embargo, como el grado total puede ser menor o igual a n, se introduce una "variable de holgura" para "gastar" los puntos que no se utilizan en las variables. Entonces hay k variables más la variable de holgura, lo que suma un total de $k+1$ "posiciones" o "urnas" donde los exponentes pueden ir.

Se puede visualizar de la siguiente manera: Cada variable de las $k+1$ es una urna, y las unidades de los exponentes, las "bolas". Entonces, se necesitará ver cuántas maneras distintas hay de meter las n bolas en las $k+1$ urnas. Se puede representar mediante puntos y barras, con los espacios a la izquierda y derecha de esas k barras simbolizando las urnas. Por ejemplo, si hay 3 urnas se representaría de la manera: `URNA|URNA|URNA`. De la misma manera, las "bolas" serían los puntos. Un caso para 5 bolas y 3 urnas: $\cdot\cdot|\cdot|\cdot\cdot$ que representa que se han metido 2 en la primera, 1 en la segunda y las 2 restantes en la última. El monomio x^2y en un polinomio de 2 variables y grado total 4 se simbolizaría de la manera $\cdot\cdot|\cdot|\cdot$.
Generalizando: todos los casos que hay para k variables y grado n, es decir, N, surgen de reordenar esas k barras y n puntos.

$$N = \frac{(n+k)!}{n!k!} = \binom{n+k}{k}$$

Dado que $\mathbb{C}^N$ es contráctil, existe un camino que une a cualesquiera 2 polinomios de grado total $\leq n$, llamado **homotopía**. Esta homotopía puede ser una combinación convexa de ambos.

Si se restringen los polinomios a tener un grado total fijo n, la situación cambia, ya que el monomio de mayor grado es siempre distinto de 0, por lo que el espacio de coeficientes ya no es $\mathbb{C}^N$, sino $\mathbb{C}\backslash\{0\} \times \mathbb{C}^{N-1}$, que no es contráctil, debido a que $\mathbb{C}\backslash\{0\}$ es homotópicamente equivalente a la circunferencia unidad. A pesar de eso, los complejos sin el origen son un espacio conexo por caminos, por lo que $\mathbb{C}\backslash\{0\} \times \mathbb{C}^{N-1}$ lo es, y en consecuencia el espacio de coeficientes de los polinomios también. Nótese que si los polinomios tienen coeficientes sobre $\mathbb{R}$, el espacio $\mathbb{R}\backslash\{0\}$ deja de ser conexo, causando diversos problemas, pero en el contexto del método de continuación homotópica se puede generalizar y tomar todos los polinomios con coeficientes en los complejos.

1.3.2. Construcción de homotopía

Dado un sistema de polinomios $F(\mathbf{x})$ del que se quiera hallar $V(F)$, se construye otro sistema más simple $G(\mathbf{x})$ con soluciones conocidas, tal que el grado de sus polinomios coincida. Es decir, si el primer polinomio del sistema F tiene grado 5, el grado del primer polinomio "simple" del sistema G será 5 y así sucesivamente.
Con estos dos sistemas, se construye una homotopía, que normalmente consiste en la combinación convexa de ambos:

$$H(\mathbf{x}, t) = (1-t)F(\mathbf{x}) + tG(\mathbf{x})$$

De este modo para $t = 1$ se tiene el sistema fácil, y para $t = 0$ el sistema objetivo. Esta decisión se toma ya que según el estándar IEEE 754 de números de coma flotante [13], la densidad de números representables es mayor cerca del cero.
Obviamente hay muchos más tipos de homotopía entre polinomios, pero esta suele funcionar bien y es más que suficiente. Es la que se usará en este trabajo.

1.3.3. Seguimiento de caminos

En **geometría diferencial** y **topología**, el concepto de **homotopía de caminos** es fundamental, ya que permite entender cuándo dos caminos [2] en un espacio topológico son esencialmente el mismo, es decir, si uno se puede deformar de forma continua en el otro manteniendo sus puntos inicial y final fijos. Esta deformación se logra a través de una función de homotopía, que gracias a un factor continuo (normalmente $t \in [0, 1]$) crea una familia de caminos intermedios conectando el inicial y final. Es de gran utilidad para la clasificación de espacios topológicos, pero en este trabajo se indagará en la propia homotopía.

Tal y como en las homotopías entre caminos de geometría se tenían que encontrar funciones que "llevasen un camino a otro" el **método de continuación homotópica** busca conectar un problema "fácil" de resolver con otro más "difícil", o de interés. Esto se hace construyendo una familia continua de problemas intermedios, parametrizados por un factor continuo $t \in [0, 1]$.

Las soluciones a estos problemas intermedios forman **caminos de soluciones** en el espacio de las variables, y la clave es "seguir" cada camino desde el punto de partida hasta la solución deseada. Cada punto en este "camino de soluciones" representa una solución válida para el problema en un instante t dado.

Encontrar explícitamente la función que define el camino de soluciones puede ser extremadamente complejo (mucho más que resolver el problema inicial) o incluso inviable debido a la naturaleza no lineal de los sistemas involucrados. Por ello, se recurre a métodos numéricos de **seguimiento de caminos**, como el método de *path-tracking*. Este enfoque computacional no busca la expresión analítica de la homotopía, sino que aproxima el camino de soluciones avanzando paso a paso a lo largo de él. En la práctica, esto se logra mediante una combinación de un predictor (que estima el siguiente punto en el camino) y un corrector (que refina esa estimación para asegurar que se mantenga sobre la trayectoria de soluciones), que se explicarán más adelante.

Los caminos de los que habla este trabajo no son algo abstracto, sino que se pueden visualizar como se verá en el Capítulo 3 o en el siguiente ejemplo simple:

Ejemplo 1.3.3.1 *Se quieren encontrar las raíces del polinomio* $x^2 - 5$ *(son* $\pm\sqrt{5}$*). Para ello se parte de un polinomio más simple,* $x^2 - 1$*, cuyas raíces (conocidas) son* ± 1 *y se construye la homotopía:*

$$H(x, t) = (1 - t)(x^2 - 5) + t(x^2 - 1) = x^2 + 4t - 5$$

Se resuelve para x *dependiendo de* t *igualando la homotopía a 0, y se obtienen 2 caminos de soluciones, válidos* $\forall t \in [0, 1]$*, que son:* $x_1(t) = \sqrt{5 - 4t}$ *y* $x_2(t) = -\sqrt{5 - 4t}$*. En la Figura 1.3 se pueden visualizar.*

Obviamente para $t = 0$ y para $t = 1$ se obtienen las soluciones esperadas, pero no siempre es tan fácil "despejar" la x, especialmente cuando trabajamos con polinomios con varias variables. Es por eso que el método se encarga de aproximar $\mathbf{x}(t)$ de manera numérica.

[2] Aplicación continua en X espacio topológico de la forma $\gamma : [0, 1] \to X$ con $\gamma(0)$ el punto inicial y $\gamma(1)$ el punto final. [14]

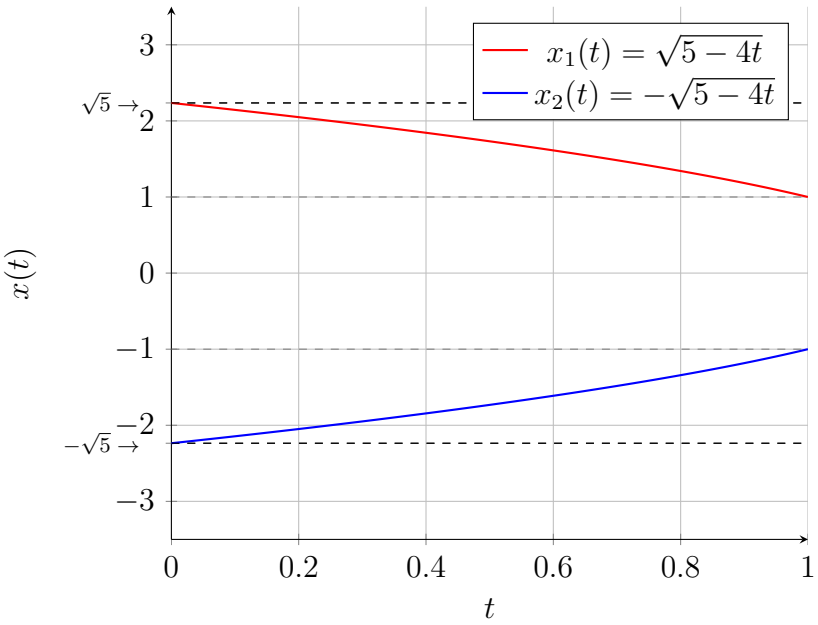

Figura 1.3: Caminos de la homotopía para $x^2 - 5 = 0$.

Predictor de Euler

Creada la homotopía $H(\mathbf{x}, t) = (1-t)F(\mathbf{x}) + tG(\mathbf{x})$, lo que se pretende es seguir la curva de solución de cada solución, desde ahora $x(t)$, definida por $H(x(t), t) = 0$. Para ello, se necesita primeramente encontrar qué dirección seguirá la curva ($\frac{dx(t)}{dt}$). Para hallarlo se derivará la homotopía usando la regla de la cadena:

$$\frac{d}{dt}H(x(t),t) = \frac{\partial H}{\partial x}(x(t),t) \cdot \frac{dx(t)}{dt} + \frac{\partial H}{\partial t}(x(t),t) = 0$$

Que, igualando términos, se tiene que:

$$J_{H(x(t),t)} \cdot \frac{dx(t)}{dt} = -\frac{\partial H}{\partial t}(x(t),t)$$

siendo $J_{H(x(t),t)} = \frac{\partial H}{\partial x}(x(t),t)$ el jacobiano del sistema respecto a las variables, y con $\frac{\partial H}{\partial t}(\mathbf{x}, t) = -F(\mathbf{x}) + G(\mathbf{x})$.
Se resuelve para $w = \frac{dx(t)}{dt}$, y se usará más adelante para dar el paso predictivo $x_{pred} = x(t+h) \approx x + h \cdot w$ con h un paso definido en base a las circunstancias del problema.

Corrector de Newton

Una vez se ha dado un paso con Euler o cualquier otro predictor (se puede no dar el paso directamente, ya que el corrector actúa las veces de predictor), puede ocurrir que la solución dada para ese t esté algo lejos de la curva de soluciones real. Es por eso que se intenta acercar de nuevo con los métodos correctores, y uno de los más importantes es el de Newton.

En vez de trabajar como antes con $x(t)$, que es la solución exacta de la curva de soluciones, se usará la notación v que es el vector de soluciones "aproximado" para cada paso de la homotopía (Es decir, un vector que evaluado en la homotopía fijado t, se acerca lo suficientemente al vector nulo $H(v, t) \approx 0$). También se definirá el "paso" con el que avanzará la homotopía como Δt (recordar que se recorrerá t de 1

a 0). Se asume que el jacobiano es no singular para cualquier punto de la curva de soluciones que se quiera seguir. El paso corrector de Newton transforma v en $\hat{v}$.

$$\hat{v} = v - J^{-1}_{H(v,t-\Delta t)} \cdot H(v, t - \Delta t),$$

donde de acuerdo con la notación elegida y poniendo $\hat{t} = t - \Delta t$:

- $J^{-1}_{H(v,\hat{t})} = (\hat{t} \cdot J_{G(\mathbf{x})}(v) + (1 - \hat{t}) \cdot J_{F(\mathbf{x})}(v))^{-1}$
- $H(v, \hat{t}) = \hat{t}G(v) + (1 - \hat{t})F(v)$

Luego, se realizará una reasignación de valor $v \leftarrow \hat{v}$ y se repetirá durante un número de iteraciones, o hasta que $\|v - \hat{v}\| < \varepsilon$ con una norma determinada (en el programa de este trabajo será la euclídea) y un ε o umbral preestablecido. El procedimiento completo se expone en la siguiente explicación:

MÉTODO DE SEGUIMIENTO DE CAMINOS EN PASO VARIABLE

Objetivo: Encontrar una secuencia de puntos (t_i, v_i) que aproximen un camino de solución de la homotopía cuadrada $H(\mathbf{x}, t) = 0$ dado un punto inicial v_0 tal que $H(v_0, 1) \approx 0$.

Entrada:

- Homotopía entre sistemas cuadrados de la misma dimensión $H(\mathbf{x}, t) = 0$
- Punto inicial v_0 tal que $H(v_0, 1) \approx 0$
- Valor inicial del parámetro de homotopía $t = 1$
- Tamaño del paso inicial $\Delta t_{inicial} = 0.1$ por defecto

Salida:

- Secuencia de puntos (t_i, v_i) que aproximan la curva de homotopía.

Procedimiento:

1. **Inicialización:**
   - Establecer $t\_actual \leftarrow 1$
   - Establecer $v\_actual \leftarrow v_0$
   - Establecer $\Delta t \leftarrow \Delta t_{inicial}$
   - Inicializar lista de soluciones $S \leftarrow []$
2. **Bucle principal:**
   - Mientras $t\_actual > 0$ o Δt mayor que un valor mínimo:
     a) **Paso predictor (Opcional):** Estimar una nueva solución *v_pred* para un nuevo valor de t utilizando por ejemplo un paso de Euler.
     b) **Paso corrector:** Inicializar $t\_nuevo \leftarrow t\_actual - \Delta t$. Inicializar $v\_corrector \leftarrow v\_pred$ o si no se ha realizado el paso del predictor, v_actual. Luego actualizar *v_corrector* realizando 4 (por defecto) iteraciones de un método corrector como Newton-Raphson, o cortarlas si $H(v\_corrector, t\_nuevo) \approx 0$.

c) **Evaluación de la solución:** Se evalúa $||H(v\_corrector, t\_nuevo)||$ con la norma elegida (de normal la euclídea).
* Si ≈ 0: *Exitoso*. Se actualiza la t con $t\_actual = t\_nuevo \leftarrow t\_actual - \Delta t \cdot \lambda$. Se guarda la solución aproximada $S = S \cup \{(t\_actual, v\_corrector)\}$
* Si no: *No exitoso*. No se guarda la solución, no se actualiza la t y se reintenta para la misma t y $\Delta t \leftarrow \frac{\Delta t}{\lambda}$

3. **Refinamiento final (Opcional):**
   - Cuando t_actual es próximo a 0, se puede aplicar un método de refinamiento adicional para pulir la solución en $t = 0$
4. **Retorno:**
   - Devolver la secuencia de soluciones S

Notas:

- λ es el factor de escalado y determina cómo se ajusta el tamaño del paso Δt. Normalmente se escoge entre 1.1 y 2.
- El valor mínimo para Δt detiene el bucle para evitar pasos excesivamente pequeños y una recursión infinita.
- El criterio de que $||H(v\_corrector, t\_nuevo)||$ sea suficientemente pequeño será del tipo $||H(v\_corrector, t\_nuevo)||_2 < umbral$ con un umbral definido. En el trabajo se utilizará 10^{-6} que ha demostrado dar buenos resultados en la práctica. Aún así si el método de Newton falla, no falla por culpa del umbral si no por otros motivos.
- Los pasos de predictor y corrector pueden cambiar, pero en el trabajo se utilizará el predictor de Euler y el corrector de Newton-Raphson.
- Para un seguimiento de caminos de paso fijo, saltarse el paso 2.c) de evaluación de la solución y reintentar con $t = t - \Delta t$

Capítulo 2

Introducción al programa de continuación homotópica

En esta sección se presentará el programa, parte principal del TFG. Se justificará el lenguaje de programación y estructura utilizada, y se explicarán las partes principales que lo componen.

2.1. Justificación de estructura y lenguaje de programación

Para el desarrollo y procesamiento de polinomios en este trabajo, se ha seleccionado el lenguaje de programación *Python*. Esta elección se fundamenta en las siguientes características:

Orientado a objetos

En primer lugar, Python es un lenguaje **orientado a objetos**, lo que permite representar elementos matemáticos mediante clases (polinomios, matrices, vectores...) por lo que se tienen las siguientes ventajas:

- **Abstracción intuitiva:** Los objetos pueden modelar entidades matemáticas y operaciones (incluso mediante sobrecarga de operadores) de una forma natural. Por ejemplo `polinomio.grado()`, `vector.norma()` ó `matriz@vector`.
- **Código organizado:** *Python* permite que las operaciones y entidades se encapsulen en clases, y cada una separada en distintos archivos. No se tienen por qué usar *strings* o listas para almacenar con lo que se trabajará.
- **Extensible:** *Python* permite el uso de interfaces y abstracción. El desarrollo de nuevos elementos para el código es trivial y no requiere reescribir todo de nuevo.

Integración con librerías

Al ser *Python* uno de los lenguajes más utilizados en el ámbito científico, cuenta con diversas bibliotecas bastante veteranas con las que se puede optimizar el código, como *SymPy* para el cálculo simbólico, *NumPy* para el cálculo numérico, o *Cython* para optimizar fragmentos del código al correrlos nativamente en *C*. Además, tiene un graficador de funciones, *Matplotlib*, bastante potente.

Objetivos de diseño de software

El programa de este TFG escrito en *Python* prioriza los siguientes aspectos y convenciones:

- **Claridad:** El código de las clases principales será legible, y autodocumentado, siguiendo convenciones estándar y evitando estructuras complejas o ilegibles. Se quiere que el usuario pueda interpretar el código.
- **Accesibilidad:** Al ser legible, se facilitan las colaboraciones y modificaciones externas. Es de código abierto.
- **Desarrollo simple:** *Python* permite construir tests y casos de estudio de forma fácil. Mismo caso para su expansión.

Limitaciones

Python es ideal para el fin didáctico de este trabajo debido a las razones antes presentadas, pero tiene una gran limitación: su rendimiento. Al ser un lenguaje interpretado y con tipado dinámico, es menos eficiente que otras alternativas compiladas o de más bajo nivel, como *C, C++* o *Fortran*. Por lo tanto, para sistemas de alta demanda computacional o de estándar industrial, se recomienda elegir otros lenguajes de programación, o si se quieren tener las ventajas anteriores, integrar *Python* con módulos optimizados para lenguajes como los anteriores, como *Cython* o *ctypes*.

2.2. Representación de polinomios en programas

Antes de empezar a realizar cualquier tipo de acercamiento al método de continuación homotópica, se deben definir de forma precisa cómo se representan los sistemas, polinomios y homotopías en un lenguaje que sea claro, tanto como para el usuario como para el ordenador, y con el que resulte sencillo programar. Se empezará con los bloques básicos:

2.2.1. Monomios

Para la representación de monomios de n variables, el bloque básico sobre el que se construye el programa, se utilizará un *array* de tamaño $n+1$ compuesto por:

$$[\alpha_1, \ldots, \alpha_n, a]$$

en el que se empleará la notación antes definida en la Definición 1.2.1.1. Los exponentes $\alpha_1, \ldots, \alpha_n \in \mathbb{N}$ serán enteros (0 y positivos), mientras que el coeficiente a será un número complejo de coma flotante para obtener la máxima precisión a la hora de hacer los cálculos.

Se deberá prestar especial atención a que los exponentes conserven un orden basado en el orden que llevan las propias variables. Este es un dato muy importante la hora de considerar hacer el programa, y es por eso que se usa un *array*, a diferencia de otras estructuras de datos.

Ejemplo 2.2.1.1 *El siguiente monomio* $m(x) = -3x^3y$ *sobre las variables* x, y, z *tendrá una representación en array de tipo* $[3, 1, 0, -3]$.

2.2.2. Polinomios

Los polinomios se han definido con anterioridad como una combinación lineal de monomios, por lo que la representación que se usará consistirá en un $2-array$ o $array$ de 2 dimensiones, cuyos elementos serán los $arrays$ que representan a cada monomio: $[m_1, \ldots, m_N]$.

Ejemplo 2.2.2.1 *El polinomio* $P(x) = -3x^3y^2z - 2yz + 4x + 7$ *sobre las variables* x, y, z *tendrá una representación en array* $[[3, 2, 1, -3], [0, 1, 1, -2], [1, 0, 0, 4], [0, 0, 0, 7]]$, *cuyos elementos cumplirán las mismas reglas que las antes definidas.*

2.2.3. Sistemas de polinomios

La representación de un sistema de polinomios, siguiendo con la lógica anterior, consistirá en un $3-array$, cuyos elementos son polinomios ($2-arrays$). De la forma: $[P_1, \ldots, P_m]$.

Ejemplo 2.2.3.1 *El sistema* $F(x) = \{x^2 - y^2, x^2 - y\}$ *estará representado por:* $[[[2, 0, 1], [0, 2, -1]], [[2, 0, 1], [0, 1, -1]]]$

2.2.4. Soluciones aisladas

Las soluciones de los polinomios se expresarán como vectores o listas representando los puntos aislados solución del sistema. Consisten en un *array* del tamaño igual al número de variables.

2.2.5. Polinomios como *strings*

En el contexto de este trabajo, los polinomios que se admitirán en los programas se representarán por parte del usuario no como las estructuras antes definidas sino mediante *strings*. Esto se hace para facilitar la introducción de los datos al emplear un lenguaje más natural. Los *strings* cumplirán con las siguientes reglas sintácticas:

- **Separador de términos:** Los términos del polinomio estarán separados por operadores binarios de suma ($+$) o resta ($-$). No incluir $+-$ ni paréntesis entre operadores salvo en el caso de los complejos.
- **Complejos:** Los coeficientes complejos se escribirán siempre de la manera ($a+bj$). Nótese el uso de paréntesis y la j como unidad imaginaria, sin asteriscos para multiplicarla. Los complejos siempre irán precedidos por un signo $+$.
- **Separador decimal:** La coma (,) se usará exclusivamente para separar polinomios en los sistemas. Por lo tanto, el punto (.) se usará como separador decimal en los coeficientes.
- **Estructura de cada término:** Cada monomio constará de un producto de coeficientes, variables y potencias, unidos por asteriscos ($*$). Las potencias se indicarán con el acento circunflejo ($\wedge$) seguido del exponente, sin paréntesis. Tampoco se admiten operaciones implícitas, y los únicos paréntesis se utilizarán con los números complejos.

Ejemplo 2.2.5.1 *Para una mejor visualización se proporciona como ejemplo el sistema* $\begin{cases} 0.5x^2 + y^3 - ixy^2 - 1 \\ -x^2 - (2 - 3i) \end{cases}$ *que según nuestra notación se representaría como un string "0.5*x^2+y^3+(-j)*x*y^2-1 , -x^2+(-2+3j)"*

Estas convenciones tienen como fin simplificar la visualización de los polinomios y centrar el programa en el método de la continuación homotópica.

Se pueden escribir los polinomios de muchas formas distintas según las diversas necesidades. Por ejemplo, la notación que se usa en este trabajo es la expandida, que tiene como fin ser fácil de procesar y leer para un humano y así comprender el funcionamiento del método, pero se podrían usar notaciones factorizadas, más compactas, y que permitirían anular términos (a pesar de que hay polinomios que no se pueden factorizar), o "SLPs" (*straight-line programs*), que consiste en guardar los polinomios como una secuencia de operaciones. (Muy fácil de entender para un ordenador ya que sigue la estructura *stack*, y muy eficiente ya que por ejemplo si se calcula x^2 no se tiene que volver a calcular). Es la notación usada en los programas profesionales como *PHCpack* ó *HomotopyContinuation.jl*.

Como se acaba de decir se trabajará con la notación expandida ya que a fin de cuentas pasar de unos a otros sería trabajar con expresiones de registro que ofuscan en exceso los métodos.

2.3. Estructura del programa: clases y funcionalidades

Como se ha justificado en el apartado anterior este programa pretende ser modular y orientado a objetos, para que cada concepto matemático pueda manipularse de forma precisa, y para que sea eficiente y escalable.

Se detallarán las clases usando fragmentos del código y explicando los métodos más relevantes. El programa documentado y en su totalidad se podrá encontrar en el Anexo.

2.3.1. Clases fundamentales

Son los bloques básicos sobre los que se modela la totalidad del programa y definen las estructuras de datos que se utilizarán.

Monomio

Representa los monomios $a \cdot x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ como *arrays* de la forma antes definida $[\alpha_1, \ldots, \alpha_n, a]$.

```
def __init__(self, expresion: str, variables: List[str]):
    """
    Constructor de la clase Monomio

    Args:
        expresion (str): Expresión formateada (p.e. -2*x*y^2)
        variables (List[str]): Lista de variables sobre la que se
```

```
            construye el monomio (p.e. ["x","y","z"])
    """
    self._variables = variables
    self._array = self._parsear_monomio(expresion.replace(" ",""))
```

La función `_parsear_monomio(str)` recoge el monomio formateado según la convención 2.2.5 y lo trocea, enviando cada parte a una posición del *array*. Es importante definir las variables debido a que por ejemplo, `2*x` con las variables $[x, y, z]$ se representa $[1, 0, 0, 2]$.

Cuenta con métodos de **sobrecarga de operadores**. Estos métodos predefinidos por *Python* permiten asignarles nuevas funciones a los operadores binarios y unarios convencionales para que trabajen con los nuevos objetos creados. En la clase Monomio se usan `__mul__(float|complex)` y `__rmul__(float|complex)` que permiten la multiplicación del monomio por un escalar a ambos lados usando el operador estrella `*`. Es necesario definirlos ya que sólo se busca escalar el coeficiente, no los exponentes.

```
def __mul__(self, escalar: float|complex) -> "Monomio":
    """
    Sobrecarga de método de multiplicación por la izquierda por escalar.
    Multiplica el coeficiente.

    Args:
        escalar (float|complex): Número por el que se quiere multiplicar.
    Returns:
        Monomio: Monomio con el coeficiente multiplicado por el escalar.
    """
    # Se cogen las variables y se añade el coeficiente escalado al final.
    nuevo_array = self._array[:-1] + [self.coeficiente * escalar]
    # Creo un nuevo monomio sin llamar al constructor
    nuevo_monomio = Monomio.__new__(Monomio)
    # Modifico su atributo array
    nuevo_monomio._array = nuevo_array
    # Modifico su atributo variables
    nuevo_monomio._variables = self._variables
    return nuevo_monomio
```

El método `__rmul__(float|complex)` funciona de la misma manera.

Aparte de la sobrecarga de operadores, un monomio cuenta con funciones que le son específicas, como hallar el grado (sumar los exponentes), así como la evaluación en un punto `evaluar(Vector)` o la diferenciación dada una variable o posición de variable, útil para el jacobiano. Se exponen las dos últimas:

```
def evaluar(self, valores: Vector) -> float|complex:
    """
    Dado un float o complejo, evalua el polinomio según los valores de
    variables dados.

    Args:
```

```
        valores (List[Union[float, complex]]): Valores sobre los que se
            evaluará el monomio
    Raises:
        ValueError: Dimensión incorrecta para la evaluación.
    Returns:
        float|complex: Monomio evaluado
    """
    # Pasa a lista el vector (para usar el zip)
    if isinstance(valores, Vector):
        valores = valores.tolist()
    # Comprueba las dimensiones de vector
    if len(valores) != len(self.variables):
        raise ValueError("Dimensión incorrecta para evaluar")
    # Multiplica el coeficiente por los elementos del vector potenciados.
    monomio_evaluado = self.coeficiente
    for exponente, valor in zip(self.exponentes, valores):
        monomio_evaluado *= valor**exponente

    return monomio_evaluado
```

```
def diferenciar(self, variable: str|int) -> "Monomio":
    """
    Diferencia monomio según la variable o posición dada, y devuelve el
    monomio diferenciado

    Args:
        variable (str | int): Variable o posición (del array de variables)
            dada para diferenciar el monomio sobre la misma
    Raises:
        ValueError: No es variable válida
        ValueError: Índice mayor que len(variables)
        ValueError: Variable no es string ni int
    Returns:
        Monomio: Monomio diferenciado.
    """
    # Variable numérica o explícita. Lanza excepciones.
    if isinstance(variable, str):
        if variable not in self._variables:
            raise ValueError(f"{variable} no es una variable válida.")
        indice = self._variables.index(variable)
    elif isinstance(variable, int):
        if not (0 <= variable < len(self._variables)):
            raise ValueError(f"Índice {variable} se sale de las variables.")
        indice = variable
    else:
        raise ValueError(f"{variable} no es ni string ni int.")
    # Aísla exponentes y coeficiente
    exponentes = self._array[:-1].copy()
```

```
    coeficiente = self._array[-1]
    # Se centra en la variable a diferenciar
    exponente = exponentes[indice]
    # Multiplica coeficiente por exponente y resta 1 al mismo.
    coeficiente_nuevo = coeficiente*exponente
    exponentes[indice] = exponente-1
    # Crea nuevo monomio y asigna atributos
    monomio_nuevo = Monomio.__new__(Monomio)
    monomio_nuevo._variables = self._variables
    monomio_nuevo._array = exponentes + [coeficiente_nuevo]

    return monomio_nuevo
```

Polinomio

Representa los polinomios (suma de monomios) $m_1(\mathbf{x}) + \cdots + m_k(\mathbf{x})$ con $\mathbf{x} = (x_1, \ldots, x_n)$ como un *array* de monomios (*arrays*) de la forma antes definida en el Punto 2.2.2: $[[\alpha_1, \ldots, \alpha_n, a], [\beta_1, \ldots, \beta_n, b], \ldots]$.

```
def __init__(self, expresion: str, variables: List[str]):
    """
    Constructor de la clase Polinomio.

    Args:
        expresion (str): Expresión del polinomio según las convenciones
            de la documentación. P.e. (-j)*x+2*y^2
        variables (List[str]): Lista de variables (en orden) dadas como
            un string. El polinomio se construye sobre estas.
    """
    self._variables = variables
    self._monomios = self._parsear_polinomio(expresion.replace(" ",""))
```

Lo que hace la función `_parsear_polinomio(str)` aquí es crear una lista vacía, separar los monomios según los mases y menos, y llamar al constructor de Monomio para cada fragmento. Añade esos objetos a la lista vacía.

Cuenta con métodos de sobrecarga de operadores para la multiplicación por escalar `*` que funcionan gracias a la propiedad distributiva, escalando cada monomio y añadiéndolo de nuevo a la lista, así como para la suma (`__add__(Polinomio)`) y la resta (`__sub__(Polinomio)`) (no incluídos en la clase Monomio debido a la dificultad para manejar los casos de coeficiente nulo) que suma los coeficientes si tienen los mismos exponentes.

```
def __add__(self, other:"Polinomio") -> "Polinomio":
    """
    Sobrecarga de método de suma para polinomios. Elimina los monomios con
    coeficiente 0.

    Args:
        other (Polinomio): Polinomio que se desea sumar
```

```
        Raises:
            ValueError: Si los polinomios están definidos con variables
                distintas.
        Returns:
            Polinomio: Polinomio sumado con other.
        """
        # Comprueba dimensiones.
        if self._variables != other._variables:
            raise ValueError("No se pueden sumar polinomios con distintas
            ↪  variables.")
        # Combina monomios.
        lista_monomios = self._monomios + other._monomios
        # Diccionario para agrupar por exponentes
        resultado = dict()
        for monomio in lista_monomios:
            exponentes = tuple(monomio.exponentes)
            coeficiente = monomio.coeficiente
            # Si la lista de exponentes está, se suma el coeficiente al valor.
            if exponentes in resultado:
                resultado[exponentes] += coeficiente
            # Si no está, se añade como par llave-valor.
            else:
                resultado[exponentes] = coeficiente
        # Construye los monomios
        nuevos_monomios = []
        for exponentes, coeficiente in resultado.items():
            if coeficiente != 0:
                array = list(exponentes) + [coeficiente]
                # Monomio sin constructor
                monomio = Monomio.__new__(Monomio)
                monomio._array = array
                monomio._variables = self._variables
                nuevos_monomios.append(monomio)
        # Se crea el Polinomio sin constructor
        polinomio = Polinomio.__new__(Polinomio)
        polinomio._variables = self._variables
        polinomio._monomios = nuevos_monomios
        return polinomio
```

`__sub__(Polinomio)` funciona de forma más elegante utilizando los métodos ya definidos: `return self + (-1 * other)`.

Los métodos propios como `evaluar(Vector)`, `grado()` o `diferenciar(str|int)` llaman a los métodos homónimos de la clase Monomio, y los suman o añaden de nuevo a la lista de monomios.

Sistema

Representa un sistema de polinomios $F(\mathbf{x}) = p_1(\mathbf{x}), \ldots, p_m(\mathbf{x})$ con $p_i(\mathbf{x})$ polinomios, como un *array* de polinomios de la forma antes definida en el Punto 2.2.3.

```python
def __init__(self, expresion: str, variables: Optional[List[str]] = None):
    """
    Constructor de la clase sistema

    Args:
        expresion (str): Polinomios formateados y separados por una coma.
        variables (Optional[List[str]], optional): Lista de variables
            del sistema. Defaults to None, es decir, las busca dado el
            propio string.
    """
    self._expresion = expresion.replace(" ","")
    self._variables = variables
    self._polinomios = []
    # Guarda las soluciones válidas que se encuentren en la homotopía.
    self._soluciones_propuestas: List[Vector] = []

    if self._variables is None:
        self._variables = self._leer_variables()

    self._parsear_sistema()
```

En el constructor se pueden apreciar los métodos `_leer_variables()` que como su propio nombre indica, encuentra los caracteres alfanuméricos de la expresión, los ordena de menor a mayor, y los añade a la lista de variables, así como `_parsear_sistema()` que separa la expresión según las comas y los parsea llamando a Polinomio.

Cuenta con la misma sobrecarga de operadores que Polinomio (`*`, `+` y `-`) así como un método de evaluación que crea un vector de tamaño la cantidad de ecuaciones del sistema, y que consiste en la evaluación de cada polinomio. Cabe destacar que a la hora de registrar soluciones hay un comprobador que constata que ninguna solución que se vaya a introducir esté ya en la lista de ceros, con un error establecido. Aún así a la hora de realizar el método de continuación homotópica, al usuario se le expondrán las soluciones ya registradas y la que se va a registrar, para que sea él mismo quien decida si se quiere guardar la nueva solución a la que ha llegado el método.

2.3.2. Clases algebraicas

Estas clases tienen como fin representar elementos de álgebra lineal para poder trabajar con ellos tal y como se ve en la Sección 1.3

Vector

Representa un vector n-dimensional en $\mathbb{C}^n$, útil para los cálculos intermedios del método de homotopía.

```python
def __init__(self, componentes: List[Union[float, complex]]):
    """
    Constructor del vector.
    El vector se construye desde una lista de elementos que pueden ser
```

```
    float o complejos.

    Args:
        componentes (List[Union[float, complex]]): Lista que define al
            vector. Se usa una lista ya que importa el valor posicional.
    """
    self._componentes = tuple(componentes)
```

La clase cuenta con una sobrecarga de métodos para la multiplicación con escalar, la suma y la resta (`*`, `+` y `-`), así como métodos más específicos que permiten trabajar con Vector como si fuera una lista: `__len__()` que devuelve la longitud del vector, junto con `__getitem__()` para poder acceder a los valores posicionales del vector (usando `[i]` siendo `i` la posición empezando en 0). Se guarda como una tupla para ser *hashable* y comparable.

Los métodos más destacables son sin embargo son la norma y el producto vectorial usual entre vectores (usa el operador `@`).

```
def __matmul__(self, other: "Vector") -> Union[float, complex]:
    """
    Producto escalar usual entre vectores.

    Args:
        other (Vector): Vector con el que se realiza el producto escalar.
    Raises:
        ValueError: Error de dimensión
    Returns:
        Union[float, complex]: Producto escalar de los 2 vectores.
    """
    if len(self) != len(other):
        raise ValueError("Los vectores deben tener la misma dimensión")
    return sum(a * b for a, b in zip(self._componentes, other._componentes))
```

```
def norma(self) -> float:
    """
    Norma euclídea del vector. ||·||_2

    Returns:
        float: Norma euclídea del vector.
    """
    return math.sqrt(sum(abs(x)**2 for x in self._componentes))
```

MatrizNumerica

Representa una matriz de tamaño $n \times m$ con componentes en $\mathbb{C}$, útil para los cálculos intermedios del método de homotopía.

```python
def __init__(self, valores: List[List[Union[float, complex]]]):
    """
    Constructor que toma una lista de listas. Por lo tanto siendo A una matriz
    se tiene que A[i][j] busca en la lista que ocupa la posición i el elemento
    j, o lo que es lo mismo, a_ij en la notación matemática.

    Args:
        valores (List[List[Union[float, complex]]]): Lista de listas con los
            elementos que componen la matriz.
    Raises:
        ValueError: Error en el caso de que las filas no tengan la misma
            longitud entre ellas.
    """
    if not valores or not all(len(fila) == len(valores[0]) for fila in
    ↪  valores):
        raise ValueError("Todas las filas deben tener la misma longitud.")
    self._valores = valores
    # Se definen ahora para no tener que llamar a self,:valores todo el rato
    self._filas = len(valores)
    self._columnas = len(valores[0])
```

La matriz se representa al estilo *Python* con un *2-array* consistente en las filas y columnas.

Entre los métodos a destacar se encuentran `__matmul__(MatrizNumerica)` que multiplica la matriz por un vector columna situado a su derecha, y el método `inversa()` que halla la inversa multiplicativa de la matriz por pivoteo con el fin de eliminar al máximo los errores numéricos. No se incluye el segundo al ser uno de los algoritmos más implementados en métodos numéricos.

```python
def __matmul__(self, vector: Vector) -> Vector:
    """
    Operador @ que multiplica una matriz por un vector a su derecha.

    Args:
        vector (Vector): Vector que se va a multiplicar.
    Raises:
        ValueError: Error de dimensión
    Returns:
        Vector: Producto de matriz y vector
    """
    # Comprueba dimensión vector
    if self._columnas != len(vector):
            raise ValueError("Dimensiones incompatibles para multiplicación
            ↪  entre matriz y vector.")
    # Lo devuelve en el formato Vector
    resultado = []
    for i in range(self._filas):
        # Fijada la fila multiplica la misma por cada elemento del vector.
        suma = sum(self._valores[i][j] * vector[j] for j in
        ↪  range(self._columnas))
```

```
            resultado.append(suma)
        return Vector(resultado)
```

2.3.3. Clases para métodos numéricos

El propio programa se centra en estas clases, ya que no son bloques básicos. Se trabajará con ellas más adelante y serán una parte importante del TFG.

Jacobiano

Guarda una matriz jacobiana proveniente de un sistema interpretado como la Definición 1.2.3.1.

```
def __init__(self, matriz: List[List[Polinomio]], variables: List[str]):
    """
    Constructor de la matriz, almacena la matriz lista de polinomios y las
    variables de los polinomios.

    Args:
        matriz (List[List[Polinomio]]): Matriz de polinomios.
        variables (List[str]): Variables de los polinomios para evitar fallos
            dimensionales.
    """
    self._matriz = matriz
    self._variables = variables
```

Cuenta con la estructura de MatrizNumerica (2.3.2) pero en vez de almacenar valores de $\mathbb{C}$ almacena polinomios.

Es importante denotar que el constructor funciona de una manera algo diferente y que se "sucede" de un objeto de la clase Sistema.

```
@classmethod
def desde_sistema(cls, sistema: Sistema) -> "Jacobiano":
    """
    Constructor real del jacobiano. Se llama de la manera
    ↪  Jacobiano.desde_sistema()
    y construye el jacobiano en base al sistema como si fuera una función
    de C^n en C^n.

    Args:
        sistema (Sistema): Sistema del que se va a construir el jacobiano.
    Returns:
        Jacobiano: Jacobiano como matriz de polinomios.
    """
    # Inicializa la matriz como lista vacía.
    matriz = []
    # Para cada polinomio del sistema
    for polinomio in sistema.polinomios:
        fila = []
```

```
        # Lo diferencia por cada una de sus variables y lo añade a la fila.
        for variable in sistema.variables:
            fila.append(polinomio.diferenciar(variable))
        matriz.append(fila)
    # Llama al constructor a la hora de devolver.
    return cls(matriz, sistema.variables)
```

Como método importante se tiene `evaluar(Vector)` que como su nombre indica evalúa los polinomios derivados del jacobiano en un punto.

```
def evaluar(self, valores: Vector) -> "MatrizNumerica":
    """
    Jacobiano evaluado (evaluar cada polinomio).

    Args:
        valores (Vector): Punto en el que evaluar el jacobiano.
    Returns:
        MatrizNumerica: Jacobiano evaluado.
    """
    # Se da en formato matriz numérica
    matriz_evaluada = []
    for fila in self._matriz:
        fila_evaluada = []
        for polinomio in fila:
            # Añade el polinomio diferenciado evaluado en el punto
            fila_evaluada.append(polinomio.evaluar(valores))
        matriz_evaluada.append(fila_evaluada)
    # Crea el objeto MatrizNumerica
    return MatrizNumerica(matriz_evaluada)
```

Homotopia

Es la clase más importante del programa, en ella se ejecutan los pasos de homotopía y se modifican los diversos parámetros que alteran el transcurso de la misma.

```
def __init__(self, g:  Sistema, f: Sistema, delta_t: float = 0.1):
    """
    Constructor de la homotopía. Se usará por defecto una de tipo
    combinación convexa: t*g+(1-t)*f que va de t=1 a t=0

    Args:
        g (Sistema): Sistema "fácil".
        f (Sistema): Sistema "objetivo" a resolver
        delta_t (float, optional): Paso de homotopía. Puede ser variable
            dependiendo de la ejecución del método . Defaults to 0.1.
    """
    # Esencial
    self._f  = f
    self._jacobiano_f = Jacobiano.desde_sistema(f)
    self._g = g
```

```
        self._jacobiano_g = Jacobiano.desde_sistema(g)
        self._delta_t = delta_t
        self._t = 1.0
        self._trayectoria = []

        # Variación del paso
        self._delta_t_max = 0.2
        self._delta_t_min = 1e-5
        self._factor_escalado = 2
        self._umbral = 1e-6

        self.reset()

    def reset(self, t: float = 1.0, delta_t: float = None, v_inicial: Vector =
    ↪  None):
        """
        Método que "resetea" la homotopía. Se ha tomado la decisión de diseño
        debido a que la homotopía se puede reutilizar (calcular menos jacobianos),
        aunque ofusca un poco el establecimiento de homotopías de forma manual.
        Permite distribuir cada camino en un núcleo del procesador en casos
        más complicados, usando el mismo objeto de homotopía.

        Args:
            t (float, optional): Parámetro de homotopía. Poder cambiarlo es útil
                para seguir caminos con bifurcaciones o divergentes.
                Defaults to 1.0.
            delta_t (float, optional): Paso de homotopía. Si no ha funcionado el
                paso predeterminado (0.1) se puede cambiar al gusto. El propio
                método se encarga de escalarlo en base al progreso de la
                homotopía. Defaults to None.
            v_inicial (Vector, optional): Vector solución inicial que determina
                el camino de solución que se seguirá. Defaults to None.
        """
        self._t = t
        if delta_t is not None:
            self._delta_t = delta_t
        # Si se proporciona una solución aproximada, se usa.
        self._solucion_aproximada = Vector(v_inicial) if v_inicial else None
        # Se resetea la trayectoria de cada camino al resetear la homotopía.
        self._trayectoria = []
        if v_inicial is not None:
            self._trayectoria.append((t, Vector(v_inicial)))
```

Las implementaciones del método de reseteo se podrán ver claras en el apartado 2.3.3

Cada camino de la homotopía entre los 2 sistemas se llama desde `homotopia_corrector(Vector)` y `homotopia_predictor_corrector(Vector)` (para el caso de paso variable que es el más interesante). Se mostrará el primero debido a la similitud de ambos.

```
def homotopia_corrector(self, v_inicial_dado: Vector = None) ->
↪  Tuple[List[Tuple[float, Vector]],int]:
    """
    Homotopía paso variable que usa únicamente el paso del corrector de
    ↪  Newton.

    Args:
        v_inicial_dado (Vector, optional): Se puede definir el vector inicial
            desd el que se empieza si no se ha hecho ya. Defaults to None.
    Returns:
        Tuple[List[Tuple[float, Vector]], int]: Lista de tuplas consistentes
            en el vector aproximado por cada t, y el int el número de
            ↪  iteraciones
            a las que se ha llamado al paso
    """
    if v_inicial_dado is not None:
        self.reset(v_inicial = v_inicial_dado)
    # Se asegura de que la trayectoria funcione si no hay un v inicial
    elif not self._trayectoria:
        self._trayectoria.append((self._t, Vector(self._solucion_aproximada)))
    # Inicia contador de iteraciones.
    contador = 0
    # Minetras que el t sea mayor que 0:
    while self._t > 0:
        # Guarda el resultado del paso.
        v = self._paso_homotopia_corrector()
        contador += 1
    # Para que se calcule lo más cerca del 0 posible. Refinamiento.
    if self._t > 1e-12:
        self._delta_t = self._t
        try:
            v = self._paso_homotopia_predictor_corrector()
            self._trayectoria.append((0.0, v))
        except CaminoFallidoError as e:
            # Ya se ha guardado la trayectoria hasta el fallo
            pass
    return self._trayectoria, contador
```

Obviamente en el método del predictor y corrector se llama al paso propio al método de predicción y corrección. Se muestran ambos:

```
def construir_homotopia(self, t: float) -> "Sistema":
    """
    Construye la homotopía convexa t*g + (1-t)*f

    Args:
        t (float): Parámetro de homotopía actual.
    Returns:
        Sistema: Sistema h_t(x).
    """
```

```
    return self._g*t + self._f*(1-t)
```

```
def _paso_homotopia_corrector(self) -> Vector:
    """
    Paso de homotopía con corrector de Newton y paso variable.

    Raises:
        ArithmeticError: El valor del paso es demasiado pequeño.
    Returns:
        Vector: Vector solución aproximada.
    """
    # Se guardan para cambiarlos luego o no
    t_actual = self._t
    delta_t_actual = self._delta_t
    # Ajuste para no pasar de t=0
    if t_actual - delta_t_actual < 0:
        delta_t_actual = t_actual
    # t decrementado
    t_nuevo = t_actual - delta_t_actual
    # Construye la homotopía
    h_t = self.construir_homotopia(t_nuevo)
    # Coge el valor de v aprox del paso anterior
    v_gorro = copy.deepcopy(self._solucion_aproximada)
    # Corrector de Newton (4 iteraciones)
    for _ in range(4):
        h_t_evaluada = h_t.evaluar(v_gorro)
        # Si es menor de 1e-12 se da por válido
        if h_t_evaluada.norma() < 1e-12:
            break
        jacobiano = self._jacobiano_g.evaluar(v_gorro) * t_nuevo +
        ↪  self._jacobiano_f.evaluar(v_gorro) * (1 - t_nuevo)
        v_gorro -= jacobiano.inversa() @ h_t_evaluada
    # Verificar convergencia
    if h_t.evaluar(v_gorro).norma() < self._umbral:
        # Paso exitoso: aumentar delta_t para el próximo paso
        nuevo_delta_t = min(delta_t_actual * self._factor_escalado,
        ↪  self._delta_t_max)
        # "Guarda" los parámetros de forma exitosa
        self._t = t_nuevo
        self._delta_t = nuevo_delta_t
        self._solucion_aproximada = v_gorro
        # Se guarda el punto exitoso
        self._trayectoria.append((self._t, Vector(self._solucion_aproximada)))
        return v_gorro
    else:
        # Paso fallido: reducir delta_t y reintentar
        nuevo_delta_t = max(delta_t_actual / self._factor_escalado,
        ↪  self._delta_t_min)
```

```
        # Si al reducir delta t se pasa del mínimo, error.
        if nuevo_delta_t <= self._delta_t_min:
            raise CaminoFallidoError(f"|X| No converge. \nPunto: {v_gorro}
            ↪  \nt: {self._t}", self._trayectoria)
        # Reducir paso para la próxima iteración
        self._delta_t = nuevo_delta_t
        return v_gorro
```

```
def _paso_homotopia_predictor_corrector(self) -> Vector:
    """
    Paso de homotopía con predictor-corrector y paso variable adaptativo.
    Utiliza:
    1. Predictor de Euler para estimar la siguiente solución
    2. Corrector de Newton para refinar la solución
    3. Ajuste automático del paso (delta_t) basado en la convergencia
     Raises:
        ArithmeticError: El valor del paso es demasiado pequeño.
     Returns:
        Vector: Vector solución aproximada.
    """
    # Inicia la fase de predictor copiando solucion aproximada.
    v_actual = copy.deepcopy(self._solucion_aproximada)
    # Calcular Jacobiano y derivada según t.
    J = self._jacobiano_g.evaluar(v_actual) * self._t +
    ↪  self._jacobiano_f.evaluar(v_actual) * (1 - self._t)
    g_menos_f = self._g.evaluar(v_actual) - self._f.evaluar(v_actual)
    # Dirección de predictor (dx/dt = -(J)^-1 * (g - f))
    dx_dt = (-1) * J.inversa() @ g_menos_f
    # t decrementado
    t_nuevo = self._t - self._delta_t
    # Ajuste para no pasar de t=0
    if t_nuevo < 0:
        t_nuevo = 0
        self._delta_t = self._t
    # V predecido
    v_pred = v_actual + self._delta_t * dx_dt
    # Inicia fase de corrector evaluando la homotopía.
    h_t = self.construir_homotopia(t_nuevo)
    # 4 iteraciones de Newton al igual que antes
    for _ in range(4):
        h_eval = h_t.evaluar(v_pred)
        if h_eval.norma() < 1e-12:
            break
        J_corr = self._jacobiano_g.evaluar(v_pred) * t_nuevo +
        ↪  self._jacobiano_f.evaluar(v_pred) * (1 - t_nuevo)
        v_pred -= J_corr.inversa() @ h_eval
     # Verifica convergencia y ajuste de paso
    if h_t.evaluar(v_pred).norma() < self._umbral:
```

```
        # Paso exitoso: Aumenta delta t sin superar el máximo
        nuevo_delta_t = min(self._delta_t * self._factor_escalado,
        ↪  self._delta_t_max)
        # Se guardan en atributos de clase y se devuelve T
        self._t = t_nuevo
        self._delta_t = nuevo_delta_t
        self._solucion_aproximada = v_pred
        self._trayectoria.append((self._t, Vector(self._solucion_aproximada)))
        return v_pred
    else:
        # Paso fallido: Reduce delta t sin bajar del mínimo
        nuevo_delta_t = max(self._delta_t / self._factor_escalado,
        ↪  self._delta_t_min)
        # Si delta t es muy pequeño, error.
        if nuevo_delta_t <= self._delta_t_min:
            raise CaminoFallidoError(f"|X| No converge. \nPunto: {v_pred}\nt:
            ↪  {self._t}",
                                     self._trayectoria)
        # No actualizamos t ni la solución. Reducimos paso y F
        self._delta_t = nuevo_delta_t
        return v_pred
```

La notación es relativamente intuitiva y se ha sacado directamente del Punto 1.3.3. Los métodos de paso fijo esencialmente se saltan el paso de comprobar si está cerca o no (no varían el paso) por lo que no saltan errores y siempre dan un resultado y una trayectoria al acabar las iteraciones.

Respecto al error anterior, debe guardar la trayectoria, para poder dar las soluciones en casos divergentes:

```
class CaminoFallidoError(ArithmeticError):
    """
    Excepción lanzada cuando el camino de homotopía falla,
    almacenando la trayectoria recorrida hasta el momento.
    """
    def __init__(self, mensaje, trayectoria):
        super().__init__(mensaje)
        self.trayectoria = trayectoria
```

BancoPruebas

Clase que actúa, como su propio nombre indica, como un banco de pruebas en el que se pueden crear sistemas fáciles y generar sus soluciones, obtener las gráficas del seguimiento de caminos, manejar errores, recopilar soluciones y crear casos límite y de registro del rendimiento.

```
def __init__(self, descripcion: str):
    """
    Constructor del banco de pruebas. Guarda una descripción para
    ↪  identificarlo,
    un timestamp para situarlo temporalmente, y un conjunto de sistemas que
```

```
    se han probado.

    Args:
        descripcion (str): Descripción del banco de pruebas.
    """
    self._descripcion = descripcion
    self._timestamp = datetime.now()
    self._sistemas_probados = set()
```

De los métodos más importantes que implementa esta clase es el de la creación de un sistema fácil y aleatorizado aprovechándose del factor de que los sistemas que se tratan son cuadrados.

Definición 2.3.3.1 (Sistema fácil aleatorizado) *Dado un sistema cuadrado F en $\mathbf{x} = (x_1, \ldots, x_n)$ y con una sucesión de grados $(d_1, \ldots, d_n)$, su sistema fácil aleatorizado será un sistema de la forma:*

$$G(\mathbf{x}) = \begin{cases} x_1^{d1} - a_1 \\ x_2^{d2} - a_2 \\ \ldots \\ x_n^{dn} - a_n \end{cases}$$

con $\{a_1, \ldots, a_n\}$ números complejos tal que $|a_i| = 1$ para todo $i \in \{1, \ldots, n\}$.

Se escogen estos sistemas ya que respetan los grados del sistema objetivo y sus soluciones son combinaciones de las raíces de la unidad girada una fase apropiada.

```
def sistema_facil(self,
                  sistema_objetivo: Sistema) -> "Sistema":
    """
    Método que crea un sistema "fácil" de forma aleatoria dado el sistema
    objetivo. Este sistema será de la forma:
    [x1^d1-a1,x2^d2-a2,...] con x1, x2... las variables del sistema objetivo,
    d1,d2... los grados de cada polinomio del sistema objetivo, y a1, a2...
    números complejos de módulo 1.

    Raises:
        ValueError: Error de dimensión
    Returns:
        "Sistema": Devuelve el sistemas habiendo registrado ya los
            posibles valores iniciales que determinan los caminos a seguir
            (es decir, las soluciones del sistema que se acaba de crear,
            según bezout)
    """
    # Comprueba que el sistema es cuadrado
    if len(sistema_objetivo.polinomios) != len(sistema_objetivo.variables):
        raise ValueError("El sistema debe ser cuadrado.")
    # Guarda los grados del sistema objetivo
    grados = []
    for pol in sistema_objetivo.polinomios:
        grados.append(pol.grado())
    # Raíces enésimas de la unidad (desplazada por el theta)
```

```
    def hallar_raices(grado, theta) -> List[complex]:
        lista_raices_unidad_polinomio = []
        for i in range(grado):
            lista_raices_unidad_polinomio.append(cmath.exp(
            ↪  ((i*2*math.pi+theta)/grado)*1j))
        return lista_raices_unidad_polinomio
    # Crea el sistema fácil
    variables = sistema_objetivo.variables
    ecuaciones = []
    # Soluciones de cada polinomio
    soluciones = []
    for i, d in enumerate(grados):
        # Aleatoriza el complejo
        theta = random.uniform(0, 2*math.pi)
        complejo = cmath.exp(1j * theta)
        # Crea la ecuación
        eq = f"{variables[i]}^{d}+{-complejo}"
        # Añade la ecuación y las raíces de la ecuación
        ecuaciones.append(eq)
        soluciones.append(hallar_raices(d, theta))
    # Crea el sistema con la expresión
    expresion = ",".join(ecuaciones)
    sistema = Sistema(expresion, variables)
    # Asigna las soluciones del sistema realizando un producto cartesiano
    sistema._soluciones_propuestas =
    ↪  self.generar_soluciones_iniciales_recursivo(soluciones)
    return sistema
```

La función `generar_soluciones_iniciales_recursivo(List[List[complex]])` lo que hace es realizar un producto cartesiano entre las soluciones de cada polinomio para hallar cada solución del sistema. Por ejemplo si se tiene un sistema con polinomios $x-2$ e y^2-1, el primer polinomio tiene como solución 2, mientras que el segundo, ± 1. Las soluciones del sistema serán $(2,1)$ y $(2,-1)$.

Como se ha dicho antes la clase cuenta con un método para graficar los caminos del seguimiento de caminos:

```
@staticmethod
def graficar_ruta(lista_raices: List[List[Tuple[float, Vector]]]):
    """
    Método que grafica una ruta dada una lista de raíces del tipo tupla
    de floats (la t) y vector (la propia solución aproximada).
    Si son soluciones complejas se dibuja el plano complejo para cada
    ↪  variable.
    Si son soluciones reales se dibuja la ruta de cada variable contra la
    t dada.

    Args:
        lista_raices (List[List[Tuple[float, Vector]]]): Tupla de floats (la
        ↪  t) y vector
```

```
            (la propia solución aproximada).
    """
    # Si no hay soluciones
    if not lista_raices:
        print("Lista vacía.")
        return
    # Valores de la t
    t_values = [t for t, _ in lista_raices]
    # Número de gráficas a imprimir.
    num_variables = len(lista_raices[0][1])
    # Transponer la lista de vectores para obtener listas por variable
    variables_por_indice = list(zip(*(vector for _, vector in lista_raices)))
    # Detectar si son complejos
    es_complejo = any(isinstance(v, complex) for v in lista_raices[0][1])
    # Definición de la manera en la que se muestran las gráficas
    fig, ejes = plt.subplots(nrows=1, ncols=num_variables, figsize=(5 *
    ↪  num_variables, 4))
    # Para hacer iterable aunque solo haya 1 variable
    if num_variables == 1:
        ejes = [ejes]
    # Para cada variable
    for i in range(num_variables):
        var = variables_por_indice[i]
        eje = ejes[i]
        # Separa el complejo en real e imaginario
        if es_complejo:
            re = [z.real for z in var]
            im = [z.imag for z in var]
            eje.plot(re, im, marker='o')
            eje.set_xlabel(f"Re(x{i+1})")
            eje.set_ylabel(f"Im(x{i+1})")
            eje.set_title(f"Variable x{i+1} (compleja)")
        else:
            # Usa la t como eje x
            eje.plot(t_values, var, marker='o')
            eje.set_xlabel("t")
            eje.set_ylabel(f"x{i+1}")
            eje.set_title(f"Variable x{i+1} (real)")
    # Muestra gráficas
    plt.tight_layout()
    plt.show()
```

Y finalmente los métodos que asisten para hacer la homotopía:

```
def probar_homotopia_sistema_aleatorio_con_grafica(self, sistema_objetivo:
↪  Sistema, sistema_inicio: Optional[Sistema] = None, paso: float = 0.1,
↪  tipo: str = "c"):
    """
    Método que gestiona las raíces del sistema fácil y objetivo y crea la
    propia homotopía. Permite elegir entre tipos de homotopía (ampliable)
```

```
    Args:
        sistema_objetivo (Sistema): Sistema que se quiere resolver
        sistema_inicio (Optional[Sistema], optional): Sistema desde el que
            se empieza. Si no se proporciona ninguno entonces crea un
            sistema fácil aleatorizado. Defaults to None.
        paso (float, optional): Paso inicial. Defaults to 0.1.
        tipo (str, optional): Tipo de método a seguir entre el corrector (c)
            , el predictor corrector (pc), corrector paso fijo (cpf) o el
            predictor corrector paso fijo (pcpf). Defaults to "c".
    """
    # Sistema fácil aleatorizado.
    if sistema_inicio is None:
        sistema_inicio = self.sistema_facil(sistema_objetivo)
    # Se imprime el sistema
    print("\n=== PRUEBA DEL SISTEMA GENERADO ===")
    print("Sistema fácil generado:", sistema_inicio.expresion)
    print("Soluciones iniciales:", sistema_inicio.soluciones_propuestas)
    # Se registran las soluciones iniciales
    soluciones_iniciales = sistema_inicio.soluciones_propuestas
    # Si no hay soluciones iniciales pide que se añadan.
    if len(soluciones_iniciales) == 0:
        print("|X| El sistema inicial no cuenta con ninguna solución
        ↪  inicial.")
        print("|?| Para añadir, usar el método
        ↪  Sistema.registrar_solucion_propuesta")
        return
    # Crea la homotopía
    hom = Homotopia(sistema_inicio, sistema_objetivo, paso)
    # Distinción de casos y prueba con cada solución
    match tipo:
        case "c":
            print("\n\n|?| Se inicia la homotopía con el método del
            ↪  corrector.")
            for solucion_inicial in soluciones_iniciales:
                self.probar_con_homotopia_dado_raiz_corrector(hom,
                ↪  solucion_inicial)
        case "pc":
            # --- IGUAL PERO CON PC ---
        case "cpf":
            # --- IGUAL PERO CON CPF ---
        case "pcpf":
            # --- IGUAL PERO CON PCPF ---
        case _:
            # --- IGUAL PERO CON C ---
    # Añade el sistema probado a la clase
    self.anhadir_sistema(sistema_objetivo)
    print("\n|OK| Se han seguido todos los caminos. Método acabado.")
    return
```

```
def probar_con_homotopia_dado_raiz_corrector(self, homotopia: Homotopia,
↪  sol_aprox: Vector):
    """
    Sigue un camino de la homotopía.

    Args:
        homotopia (Homotopia): Homotopía que se utiliza.
        sol_aprox (Vector): v inicial con el que se empieza el seguimiento.
    """
    # Intenta realizar la homotopía y guarda iteraciones y soluciones
    print(f"\n \n|?| Probando con solución inicial: {sol_aprox}")
    try:
        # Intenta realizar la homotopía con el corrector
        lista_t_y_vectores, iteraciones =
        ↪  homotopia.homotopia_corrector(sol_aprox)
        # Si lo logra no salta error, se muestra el resultado
        print("|OK| Homotopía completada, resultado final: ",
              lista_t_y_vectores[-1][1],"\nSe tienen soluciones: ",
              homotopia.sistema_objetivo.soluciones_propuestas)
        print("|?| El método ha usado ",iteraciones," iteraciones.")
        trayectoria_final = lista_t_y_vectores
        # Se pregunta si se quiere registrar la solución
        while True:
            respuesta = input(
                "=== ¿Quieres registrar la solución? === \n(s/n):
                ↪  ").strip().lower()
            if respuesta in ('s', 'n'):
                break
            print("|X| Ingresa 's' para sí o 'n' para no.")
        if respuesta == 's' and trayectoria_final:
            homotopia.sistema_objetivo.registrar
            ↪  _solucion_propuesta(trayectoria_final[-1][1])
            print("Se tiene soluciones:
            ↪  ",homotopia.sistema_objetivo.soluciones_propuestas)
    # Si falla registra la trayectoria de todas maneras
    except CaminoFallidoError as e:
        print("|X| Camino de solución ha fallado")
        print(f"|?| Mensaje: {e}")
        trayectoria_final = e.trayectoria
        traceback.print_exc()
    # Se pregunta de todas maneras si se quiere visualizar la gráfica
    while True:
        respuesta = input(
            "=== ¿Quieres ver la gráfica de seguimiento de caminos? ===
            ↪  \n(s/n): ").strip().lower()
        if respuesta in ('s', 'n'):
            break
        print("|X| Ingresa 's' para sí o 'n' para no.")
    if respuesta == 's':
        self.graficar_ruta(trayectoria_final)
```

```
    return

def probar_con_homotopia_dado_raiz_corrector_paso_fijo(self, hom: Homotopia,
↪   sol_aprox: Vector, paso: float):
    """
    Prueba a realizar un camino de solución usando el método del paso fijo
    dado un vector inicial.

    Args:
        hom (Homotopia): Homotopía sobre la que se realiza el paso fijo
        sol_aprox (Vector): Solución con la que se empieza
        paso (float): Paso a seguir. Determina número de iteraciones.
    """
    print(f"\n\n|?| Probando con solución inicial: {sol_aprox}")
    try:
        # Intenta realizar el método de paso fijo. Ver punto 1.3.3. Igual que
        ↪  el de paso variable salvo la parte de evaluación del sistema
        trayectoria, iteraciones =
        ↪  hom.homotopia_paso_fijo_corrector(v_inicial_dado=sol_aprox,
        ↪  delta_t_fijo=paso)
        # Método finalizado, muestra solución
        print("|OK| Homotopía predictor-corrector con paso fijo completada.
        ↪  Resultado final:", trayectoria[-1][1])
        print("|?| El método ha usado ", iteraciones, " iteraciones.")
        # Pregunta por la gráfica
        # --- COMO EN EL ANTERIOR ---
        # Pregunta si se quiere registrar la solución
        # --- COMO EN EL ANTERIOR ---
    # Errores varios.
    except ValueError as e:
        print(f"|X| Error al iniciar la homotopía: {e}")
    except ArithmeticError as e:
        print(f"|X| Error durante la homotopía: {e}")
        traceback.print_exc()
    return
```

La clase también cuenta con métodos más directos que no preguntan si se quieren guardar las soluciones ni si se quieren registrar, pero se omiten debido a que se puede discernir cómo son utilizando los métodos antes expuestos.

Capítulo 3

Prueba del método y conclusiones

En este capítulo se probará el método con diversos sistemas de soluciones conocidas y con sistemas que pueden llegar a dar fallos, se comparará el método predictor-corrector con el método corrector (así como con sus variantes de paso fijo), y finalmente se intentará determinar si el éxito a la hora de hallar las soluciones del sistema depende del sistema inicial escogido.

Para el desarrollo de estas pruebas se partirán de sistemas fáciles aleatorizados según la Definición 2.3.3.1, generados por las funciones de la clase BancoPruebas del Punto 2.3.3.
La notación que se usará de ahora en adelante:

- SAF: Sistema fácil aleatorizado.
- C: Corrector.
- PC: Predictor-corrector.
- CPF: Corrector paso fijo.
- PCPF: Predictor-corrector paso fijo.
- Residual: Norma euclídea del vector resultante de la evaluación del sistema objetivo en la solución obtenida

3.1. Sistema tipo de 3 variables

Se empieza con el siguiente sistema académico de 3 variables y con coeficientes en los números complejos para ver cómo funciona el método:

$$\begin{cases} x^3 - 3xy^2 + 2z = 0.5i \\ ix^2 + y^2 = 4 \\ z^3 - 5xz + (2 - 3.14i)y = -3 \end{cases}$$

3.1.1. Análisis por soluciones

En este análisis se genera un SAF y se utilizan los 4 métodos disponibles (C, PC, CPF, PCPF) para analizar las raíces una a una. Es útil para ver los puntos en los que falla el seguimiento de caminos ($\times$ denota el inicio mientras que $\star$ la solución a la que se llega). Según Bézout habría 18 soluciones, pero se pondrán una selección de las mismas en las que algún método falle. Se puede ver completo en el cuaderno

de *Jupyter* de nombre *sistematipo.ipynb*, en el Anexo.

Solución 1:

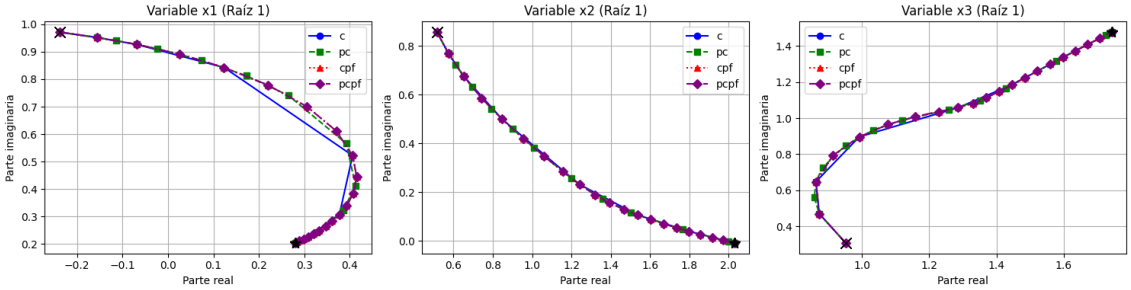

Figura 3.1: Primera solución

Método	Convergió	Iteraciones	Residual	Tiempo (s)	Solución
C	Sí	9	3.35e-14	0,0054	((0.279...+0.204...i), (2.028...-0.01...i), (1.742...+1.476...i))
PC	Sí	22	3.44e-15	0.0066	((0.279...+0.204...i), (2.028...-0.01...i), (1.742...+1.476...i))
CPF	Sí	20	7.30e-12	0.0045	((0,279...+0,204...i), (2.028...-0.01...i), (1.742...+1.476...i))
PCPF	Sí	20	2.29e-09	0.0052	((0.279...+0.204...i), (2.028...-0.01...i), (1.742...+1.476...i))

En esta solución se puede ver el caso más común entre los caminos que llegan "al mismo punto". El método C de forma constante proporciona el menor número de iteraciones (en comparación con PC) y es el que más se acerca. PC, a cambio de más iteraciones y más procesamiento (ya que tiene que calcular el predictor de Euler) suele proporcionar algo más de exactitud, mientras que en los de paso fijo, CPF ofrece una alta estabilidad en sistemas que dan pocos problemas, velocidad y unos pasos finales poco exactos. PCPF suele ser más inestable e inexacto a pesar de llegar siempre a soluciones.

Solución 12:

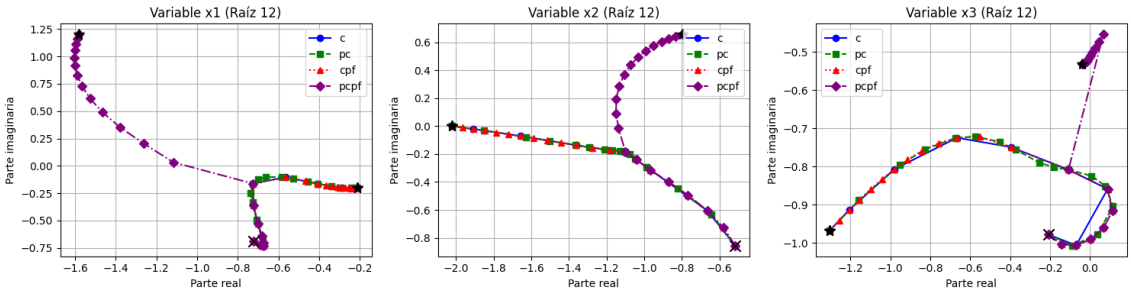

Figura 3.2: Decimosegunda solución

Método	Convergió	Iteraciones	Residual	Tiempo (s)	Solución
C	Sí	11	1.42e-15	0.0023	((-0.211...-0.2...i), (-2.021...+0.001...i), (-1.303...-0.967...i))
PC	Sí	29	2.72e-10	0.0092	((-0.211...-0.2...i), (-2.021...+0.001...i), (-1.303...-0.967...i))
CPF	Sí	20	1.50e-11	0.0036	((-0.211...-0.2...i), (-2.021...+0.001...i), (-1.303...-0.967...i))
PCPF	Sí	20	8.40e-10	0.0072	((-1.58...+1.202...i), (-0.797...+0.659...i), (-0.04...-0.532...i))

En este camino de soluciones se vuelve a comprobar la supremacía del método C al proporcionar el menor residual y el menor número de iteraciones. PC y CPF llegan a la misma solución pero con más iteraciones y mayor residual. Seguramente si se modificase el Δt para CPF tardaría menos llegando al mismo punto, pero para eso hay que saber cómo son las soluciones. PCPF por el contrario llega, pero no a la misma solución que el resto, sino a otra completamente diferente (la décima, no descubre nada nuevo). Denota la inestabilidad del método.

3.1.2. Análisis general

En este análisis, por el contrario, se generan 1000 SAFs y se miden los tiempos, iteraciones y errores medios. Aún así, debido al amplio número de SAFs es excepcionalmente complicado discernir si las soluciones han tenido una distribución uniforme discreta o si por el contrario los métodos han llegado constantemente a la misma. De todas maneras, debido a la continuidad del camino de soluciones y al Punto 3.3 es más probable que fallen que que lleguen a una solución "lejana" al camino establecido (salvo los métodos de paso fijo que "saltan" mucho más entre caminos).

```
=== C ===
- Soluciones válidas: 18000/18000
- Tiempo promedio: 0.0047s
- Residual máximo: 9.55e-07
- Residual promedio: 2.16e-09
- Desviación estándar residual: 2.78e-08

=== PC ===
- Soluciones válidas: 17998/18000
- Tiempo promedio: 0.0119s
- Residual máximo: 1.00e-06
- Residual promedio: 4.37e-08
- Desviación estándar residual: 1.43e-07

=== CPF ===
- Soluciones válidas: 18000/18000
- Tiempo promedio: 0.0206s
- Residual máximo: 9.98e-07
- Residual promedio: 1.14e-07
- Desviación estándar residual: 1.50e-07

=== PCPF ===
- Soluciones válidas: 18000/18000
- Tiempo promedio: 0.0590s
- Residual máximo: 9.98e-07
- Residual promedio: 1.68e-07
- Desviación estándar residual: 2.12e-07
```

3.2. Sistema simétrico de 4 variables

Se probará con un sistema que ofrece una inestabilidad numérica importante, y que muestra cómo no siempre se deben escoger métodos numéricos para la resolución

de ciertos problemas (es un caso perfecto para resolver por métodos simbólicos):

$$\begin{cases} x+y+z+w=10 \\ xy+xz+xw+yz+yw+zw=35 \\ xyz+xyw+xzw+yzw=50 \\ xyzw=24 \end{cases}$$

Éste es un sistema simétrico de 24 soluciones, es decir, un sistema en el que se pueden intercambiar las variables y tener la misma solución, que en este caso son las permutaciones de las coordenadas del vector $(1,2,3,4)$.

3.2.1. Análisis por soluciones

Se usará la metodología del apartado anterior, y se pondrán ciertos caminos a destacar. Se puede encontrar el análisis completo en *sistemasimetrico.ipynb*

Solución 13:

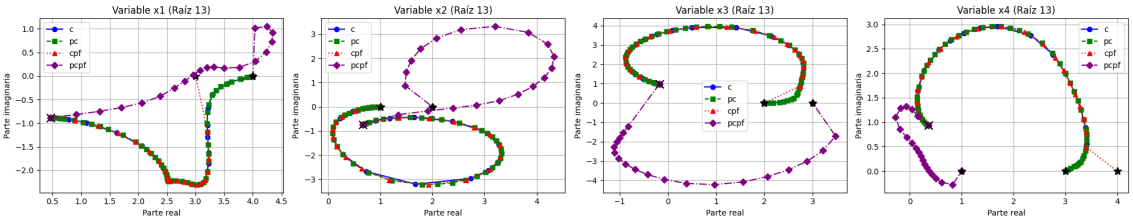

Figura 3.3: Decimotercera solución

Método	Convergió	Iteraciones	Residual	Tiempo (s)	Solución
C	Sí	55	7.47e-11	0.0231	≈ (4,1,2,3)
PC	Sí	106	2.52e-10	0.0547	≈ (4,1,2,3)
CPF	Sí	20	6.19e-04	0.0087	≈ (3,1,2,4)
PCPF	Sí	20	4.48e-11	0.0117	≈ (4,2,3,1)

En esta gráfica y en las adjuntas se puede ver la tendencia de que tanto C como PC suelen seguir los mismos caminos y llegar a los mismos puntos (y como se verá más adelante, divergir a la vez). CPF también lo intenta (no siempre lo hace como se verá en la siguiente solución), pero al final suele divergir bastante del resto, y en muchas ocasiones (como aquí) acercarse de una forma mucho menos exacta a la solución buscada. PCPF por el contrario casi siempre suele converger pero sigue caminos de una manera mucho más errática, lo que resulta en soluciones bastante más alejadas.

Solución 1:

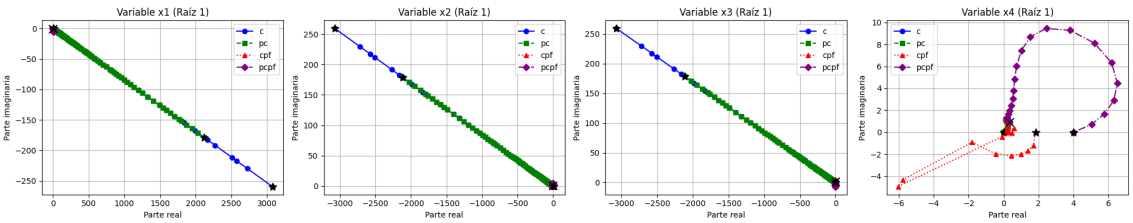

Figura 3.4: Primera solución

Método	Convergió	Iteraciones	Residual	Tiempo (s)	Solución
C	No	-1	Inf	0.0790	X
PC	No	-1	Inf	0.1415	X
CPF	No	20	1.25	0.0086	$\approx$ (1,4,3,2)
PCPF	Sí	20	1.32e-09	0.0159	$\approx$ (3,2,1,4)

En los casos en los que no converge se puede comprobar que C, al no tener que calcular ningún predictor, tarda menos en "darse cuenta" que PC. Respecto a los métodos de paso fijo, como se ha visto ya con anterioridad, CPF es muy errático, y a pesar de que se puede llegar a intuir la solución, su residual es demasiado alto como para darse por válido. PCPF sigue un camino más "normal" y llega a una solución que, aunque puede haber sido alcanzada en más ocasiones, cuenta con un residual relativamente pequeño.

3.2.2. Análisis general

Se sigue con una muestra de 1000 SAF:

```
=== C ===                                    === CPF ===
- Soluciones válidas: 17916/24000            - Soluciones válidas: 14170/24000
- Tiempo promedio: 0.0722s                   - Tiempo promedio: 0.0610s
- Residual máximo: 9.96e-07                  - Residual máximo: 1.00e-01
- Residual promedio: 2.20e-08                - Residual promedio: 9.78e-03
- Desviación estándar residual: 1.03e-07     - Desviación estándar residual: 2.19e-02

=== PC ===                                   === PCPF ===
- Soluciones válidas: 17643/24000            - Soluciones válidas: 23965/24000
- Tiempo promedio: 0.1919s                   - Tiempo promedio: 0.0884s
- Residual máximo: 9.96e-07                  - Residual máximo: 9.41e-02
- Residual promedio: 3.98e-08                - Residual promedio: 5.13e-05
- Desviación estándar residual: 1.31e-07     - Desviación estándar residual: 1.69e-03
```

3.3. Sistema de caminos que se intersecan

Se dejan los sistemas aleatorios para crear una homotopía que da problemas de forma intencionada. Se parte del sistema:

$$\begin{cases} x^2 - 2.5x = -1 \\ y^2 - 2.5y = -1 \end{cases}$$

de soluciones $(0.5, 2), (0.5, 0.5), (2, 0.5), (2, 2)$ y se acaba en el sistema:

$$\begin{cases} x^2 - 1.5x = -1 \\ y^2 - 1.5y = -1 \end{cases}$$

Por lo tanto, se forma la homotopía convexa

$$\begin{cases} x^2 - 1.5x - tx + 1 \\ y^2 - 1.5y - ty + 1 \end{cases}$$

Que al llegar a $t = 0.5$ resulta en las ecuaciones $x^2 - 2x + 1 = (x-1)^2$ (lo mismo con la variable y). Por lo tanto, todos los caminos llegan al mismo punto solución $(1, 1)$

en $t = 0.5$ y no saben continuar. Ninguno de los métodos es capaz de resolver esto, pero se ha considerado que la gráfica de *path-tracking* resulta bastante interesante:

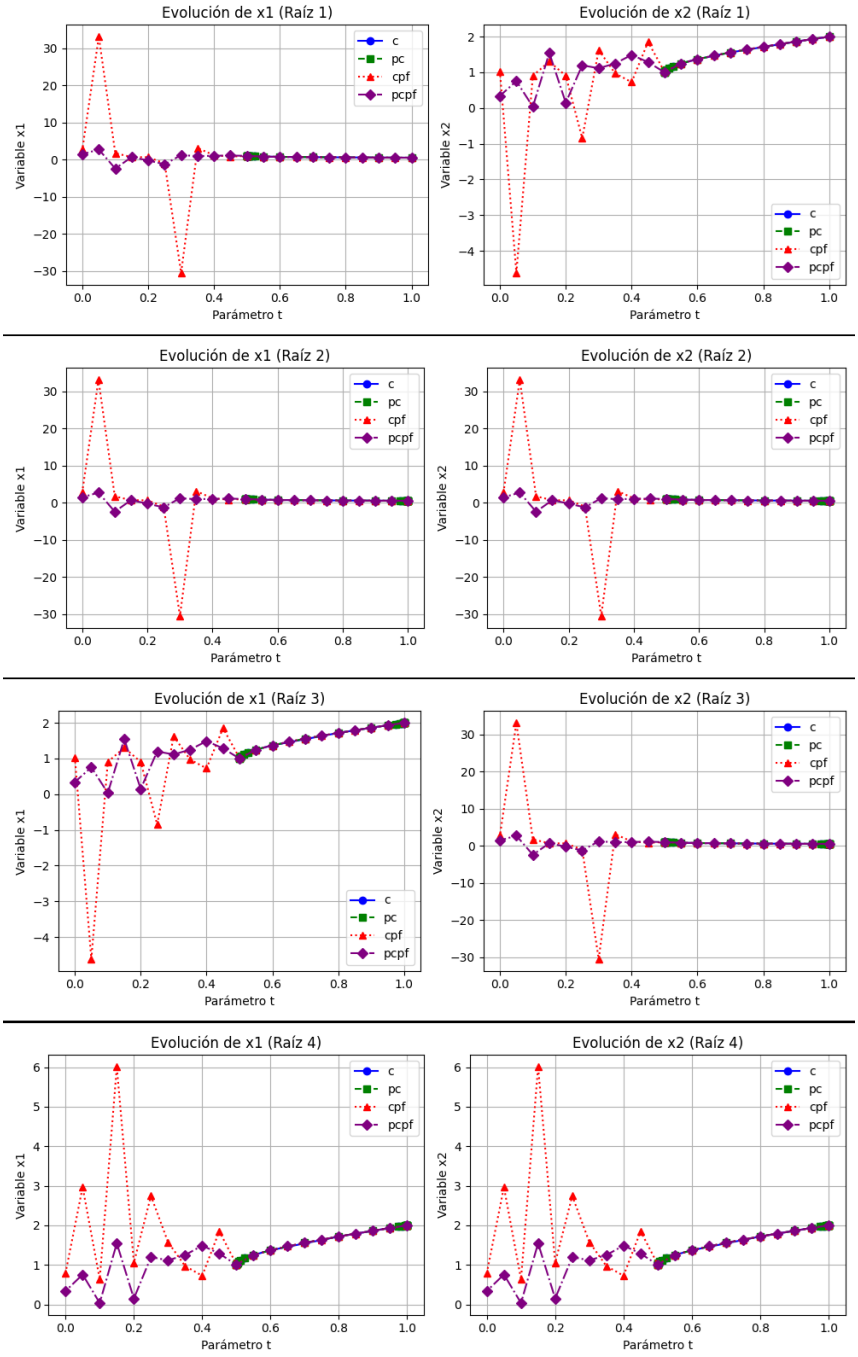

Figura 3.5: Gráficas de seguimiento de caminos para las 4 raíces

Se puede ver claramente que para $t = 0.5$ los métodos de paso variable escogen un Δt cada vez más y más pequeño hasta que el programa los para, mientras que en los de paso fijo los caminos se tornan irregulares ya que saltan entre distintas rutas, y no llegan a alcanzar ninguna solución para las 20 iteraciones propuestas (En otros experimentos con más iteraciones no llega a variar mucho el resultado). Una solución simple podría ser multiplicar el sistema objetivo por una constante compleja de módulo 1 para "rotarlo" en el espacio y forzar a que siga una trayectoria distinta.

```
=== RAÍZ 1 ===
--------------------------------------------------------------------------------------
Método              | Convergió  | Iteraciones  | Residual     | Tiempo (s) | Solución
--------------------------------------------------------------------------------------
Corrector           | NO         | -1           | Inf          | 0.0053     | [0.9993854118629083, 1.0006206916526947]
Predictor-Corrector | NO         | -1           | Inf          | 0.0080     | [0.9993479470531738, 1.0006581564624526]
Corrector PF        | NO         | 20           | 4.26e+00     | 0.0028     | [2.698103917016075, 1.0228991447602271]
Predictor-Corrector PF | NO      | 20           | 1.10e+00     | 0.0046     | [1.4425829217500419, 0.34141326597681576]

¿Desea graficar esta raíz? (s/n): s

=== RAÍZ 2 ===
--------------------------------------------------------------------------------------
Método              | Convergió  | Iteraciones  | Residual     | Tiempo (s) | Solución
--------------------------------------------------------------------------------------
Corrector           | NO         | -1           | Inf          | 0.0071     | [0.999780525167298, 0.999780525167298]
Predictor-Corrector | NO         | -1           | Inf          | 0.0077     | [0.999672189329945, 0.999672189329945]
Corrector PF        | NO         | 20           | 5.99e+00     | 0.0026     | [2.698103917016075, 2.698103917016075]
Predictor-Corrector PF | NO      | 20           | 1.30e+00     | 0.0059     | [1.4425829217500419, 1.4425829217500419]

¿Desea graficar esta raíz? (s/n): s

=== RAÍZ 3 ===
--------------------------------------------------------------------------------------
Método              | Convergió  | Iteraciones  | Residual     | Tiempo (s) | Solución
--------------------------------------------------------------------------------------
Corrector           | NO         | -1           | Inf          | 0.0063     | [1.0002204146958817, 0.999780525167298]
Predictor-Corrector | NO         | -1           | Inf          | 0.0089     | [1.0003294458917158, 0.9996721893302134]
Corrector PF        | NO         | 20           | 4.26e+00     | 0.0025     | [1.0228991447602271, 2.698103917016075]
Predictor-Corrector PF | NO      | 20           | 1.10e+00     | 0.0044     | [0.34141326597681576, 1.4425829217500419]

¿Desea graficar esta raíz? (s/n): s

=== RAÍZ 4 ===
--------------------------------------------------------------------------------------
Método              | Convergió  | Iteraciones  | Residual     | Tiempo (s) | Solución
--------------------------------------------------------------------------------------
Corrector           | NO         | -1           | Inf          | 0.0059     | [1.0002204146958817, 1.0002204146958817]
Predictor-Corrector | NO         | -1           | Inf          | 0.0082     | [1.0003294458917158, 1.0003294458917158]
Corrector PF        | NO         | 20           | 6.22e-01     | 0.0023     | [0.798572668895618, 0.798572668895618]
Predictor-Corrector PF | NO      | 20           | 8.55e-01     | 0.0043     | [0.34141326597681576, 0.34141326597681576]
```

Figura 3.6: Tablas de resultado para las 4 raíces

3.4. Sistema intensivo de 10 variables

Se ha escogido este sistema debido a la cantidad de recursos que consume y a la poca eficiencia del método a la hora que aumentan variables. Se expondrá únicamente el análisis general tal y como se ha hecho anteriormente (aunque con solo 25 SAFs debido al coste temporal)

El sistema en cuestión es el siguiente (con $j = \sqrt{-1}$):

$$\begin{cases} a + b + (2 - 3j)c - d & = 0 \\ a^2 - b + d - f & = 0 \\ b^2 + c - d + (1 + 0.5j)g & = 0 \\ c^2 - a - h & = 0 \\ d + f - g + (3j)h & = 0 \\ f^2 - h + i - k & = 0 \\ g^2 + i - l + (0.7j)a & = 0 \\ h^2 - k + l - b & = 0 \\ i^2 + k - a + (4 + j)c & = 0 \\ k^2 - l + b - d & = 0 \end{cases}$$

Al hacer el análisis general se obtiene:

```
=== C ===                                          === CPF ===
- Soluciones válidas: 3211/6400                    - Soluciones válidas: 6111/6400
- Tiempo promedio: 0.2071s                         - Tiempo promedio: 0.2066s
- Residual máximo: 9.98e-07                        - Residual máximo: 9.78e-02
- Residual promedio: 6.12e-08                      - Residual promedio: 7.39e-04
- Desviación estándar residual: 1.62e-07           - Desviación estándar residual: 5.66e-03

=== PC ===                                         === PCPF ===
- Soluciones válidas: 3198/6400                    - Soluciones válidas: 6400/6400
- Tiempo promedio: 0.2720s                         - Tiempo promedio: 0.3034s
- Residual máximo: 9.88e-07                        - Residual máximo: 9.88e-07
- Residual promedio: 6.76e-08                      - Residual promedio: 8.84e-08
- Desviación estándar residual: 1.72e-07           - Desviación estándar residual: 1.92e-07
```

A pesar de que el sistema en sí no es muy complejo, cuenta con un total de 256 soluciones, y debido al elevado número de variables calcular cada jacobiano y vector predictor tiene un coste temporal mayor. Para el cálculo de los 25 sistemas en los 4 métodos, el ordenador ha tomado un total de 8 horas. Se puede comprobar en *sistema10.ipynb*

3.5. Análisis de las raíces de la unidad

Con este pequeño experimento se quiere intentar ver si, dado un sistema objetivo "fácil" como uno compuesto por polinomios de la forma $X^3 - 1$ con $X \in \{x, y, z, t\}$, si se escogen sistemas de partida aleatorios de la siguiente forma:

$$A \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

con A una matriz 4×4 cualquiera con coeficientes en $\mathbb{C}$, y se parte de ellos (con vector inicial $(0, 0, 0, 0)^t$), ¿se dará preferencia a algunas raíces del sistema objetivo sobre otras? ¿Se alcanzarán todas?

Para realizar el experimento se intentarán seguir 5000 caminos distintos mediante el método C, al ser el que da los mejores resultados. Se escogerá el sistema "A" con coeficientes complejos aleatorios, cuyas partes real e imaginaria oscilan entre -10 y 10 con el fin de no alejarse en en exceso de los caminos a seguir. Se expone la frecuencia de las 81 soluciones en la Figura 3.8

3.6. Conclusiones

En este trabajo se han definido los sistemas de ecuaciones polinomiales, explicado el método de continuación homotópica, y se ha creado un programa que actúa como una *suite* expansible para trabajar con sistemas.

Dentro del programa se han desarrollado los métodos de *path-tracking* C, PC, CPF y PCPF. Se han visto sus puntos fuertes y sus diferencias, y según los resultados se ha determinado que **los métodos más estables son los de paso variable** (independientemente del sistema inicial escogido), y que dentro de ellos el **método C** que utiliza solo iteraciones del método de Newton es significativamente más preciso y rápido que el que también utiliza el predictor de Euler.

Se ha podido ver que **el aumento de variables en el problema produce un aumento de tiempo bastante significativo**, no sólo por el (generalmente) mayor número de caminos a seguir, sino también por el cálculo de jacobianos, evaluaciones, y multiplicaciones que se deben realizar. Además, los caminos suelen ser más "tergiversados" lo cual no ayuda a la velocidad de resolución ya que cuanto más difícil de seguir sea el camino más variaciones de paso e iteraciones hay.

¿Qué ventajas tiene la continuación homotópica frente a otros métodos? Primeramente, es **global y relativamente estable**. Al ser un proceso que emula una función continua, es difícil que todo falle "estrepitosamente", y si eso ocurre, se puede elegir entre una gran cantidad de sistemas iniciales con soluciones conocidas para reintentarlo. A pesar de esto, **a veces conviene usar otros métodos más simples como el de Newton**. En ocasiones solo se busca la rápidez, y no siempre se quieren seguir todas las soluciones, sino que con una basta. Se pone como ejemplo hallar la posición de un brazo robótico. Obviamente hay muchísimas soluciones del sistema que no son aptas (servomotores en ángulos imposibles, soluciones en el plano de los complejos...), pero sabiendo la posición anterior se puede utilizar el método de Newton para calcular la nueva, que muy probablemente converja a lo buscado, y **de forma mucho más rápida y eficiente**, **aunque no siempre más estable**, ya que el tratamiento de errores en continuación homotópica es bastante robusto. En este trabajo este punto no se ha desarrollado en exceso, pero en el libro de World Scientific "*The Numerical Solution of Systems of Polynomials Arising in Engineering and Science*" por Andrew J Sommese y Charles W Wampler publicado en marzo del año 2005 se recogen una gran cantidad de maneras en las que se pueden evitar los fallos.

Puesto que se considera que el programa puede tener bastante valor a la hora de enseñar asignaturas de cálculo numérico o de mostrar el método de seguimiento de caminos, se añaden algunas cosas a mejorar en el programa que afectan tanto a su usabilidad como al rendimiento:

- **Limpieza del código:** Debido a las muchas revisiones que ha habido, puede haber algunas variables o procesos arcaicos. Hace falta también mejorar y estandarizar los comentarios especialmente en la zona de *tests*.
- **Control de versiones:** Introducción de *Git* para poder editar el programa de forma simultánea y se puedan discutir los cambios.
- ***UI* o *CLI*:** *UI* o interfaz gráfica para el usuario, con la que no haría falta tocar ningún parámetro del código. También un *CLI* o interfaz en línea de comandos para utilizar la aplicación en un entorno seguro y utilizando la sintaxis que proporciona el propio programa.
- **Optimización:** Mejora del código eliminando procesos redundantes o usando estructuras de datos más rápidas. Pasar la parte del código más intensiva a *C* y utilizar *Numpy* para las operaciones entre matrices. Manejo de errores y cambio entre homotopías en el caso de error.
- **Multithreading:** El rendimiento es bastante pobre al ejecutarse un camino a la vez, y utilizando únicamente la *CPU*. Se debería poder trabajar con un núcleo por camino para que si no converge se pueda probar a seguir otro de forma inmediata. También utilizar la *GPU* al ser más útil para este tipo de cálculos concurrentes. Ver Figura 3.7

- **Uso de I.A.:** La introducción de una inteligencia artificial que sepa trabajar con texto puede resultar muy útil al ser capaz de procesar los sistemas introducidos por el usuario (usando la notación que use), así como poder formatear los datos obtenidos de la manera que se le especifique. También para ser capaz de identificar soluciones divergentes mucho antes.

Respecto al rendimiento, para realizar las pruebas se ha utilizado *Python* 3.13.3 en un ordenador con procesador *Intel Core i5-1235U* y *16GB* de memoria *RAM*. Se destina el 20 % del uso del procesador a *Python*, y $1GB$ de memoria (normalmente se usa entre el 15 % y $300MB$ respectivamente).

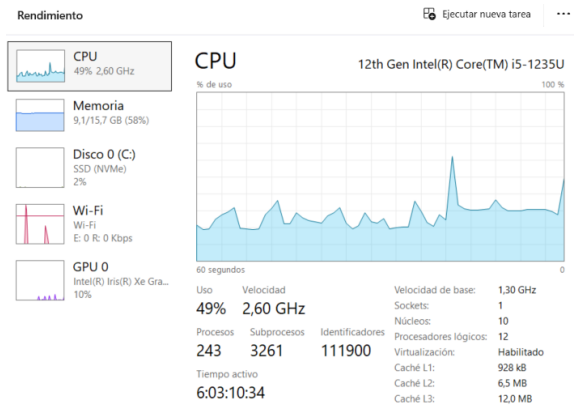

Figura 3.7: Actualmente no se usa la GPU

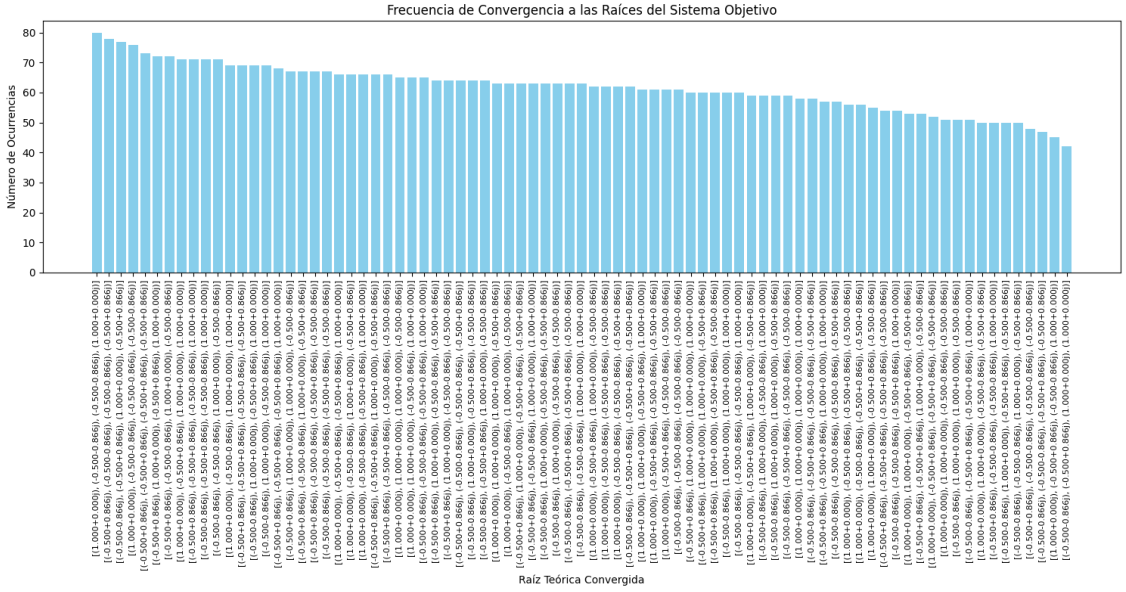

Figura 3.8: Las 81 soluciones del sistema ordenadas por frecuencia

Bibliografía

[1] Kangshen Shen, John N. Crossley y Anthony W. Lun. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford: Oxford University Press, 1999.

[2] Carl Friedrich Gauss. *Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections*. Trad. por Charles Henry Davis. Traducción al inglés de la obra original de 1809. Boston: Little, Brown y Company, 1857.

[3] Piedad Yuste Leciñena. "Ecuaciones cuadráticas y procedimientos algorítmicos: Diofanto y las matemáticas en Mesopotamia". En: *Theoria: An International Journal for Theory, History and Foundations of Science* 23.62 (2008). Citado en p. 228, págs. 219-244. ISSN: 0495-4548.

[4] Wikipedia Contributors. *Robot kinematics*. 2024. URL: https://en.wikipedia.org/wiki/Robot_kinematics (visitado 23-05-2025).

[5] M. Steglehner. *Flexible solutions of polynomial systems and their applications in robotics*. Whitepaper. Maplesoft, 2014.

[6] Wikipedia Contributors. *Stewart platform*. 2024. URL: https://en.wikipedia.org/wiki/Stewart_platform (visitado 23-05-2025).

[7] M. Hernandez-Gonzalez. "Chapter 7 - Polynomial state estimation in infectious diseases". En: *Feedback Control for Personalized Medicine*. Ed. por Esteban A. Hernandez-Vargas. Academic Press, 2022, págs. 129-149. ISBN: 978-0-323-90171-0. DOI: https://doi.org/10.1016/B978-0-32-390171-0.00016-0. URL: https://www.sciencedirect.com/science/article/pii/B9780323901710000160.

[8] Jorge Nocedal y Stephen J. Wright. *Numerical Optimization*. 2nd. New York, NY: Springer, 2006.

[9] Data 88E: Economic Models and Analytics Contributors. *Budget Constraints*. Textbook: Economic Models and Analytics. 2024. URL: https://data88e.org/textbook/content/05-utility/budget-constraints.html (visitado 23-05-2025).

[10] Wikipedia Contributors. *Nonlinear system identification*. 2024. URL: https://en.wikipedia.org/wiki/Nonlinear_system_identification (visitado 23-05-2025).

[11] Wikipedia Contributors. *Buchberger's algorithm*. 2024. URL: https://en.wikipedia.org/wiki/Buchberger%27s_algorithm (visitado 23-05-2025).

[12] Wikipedia. *Combinaciones con repetición*. Consultado el 20 de mayo de 2025. Ver sección "Cálculo del número de combinaciones con repetición". 2025. URL: https://es.wikipedia.org/wiki/Combinaciones_con_repetici%C3%B3n (visitado 20-05-2025).

[13] Institute of Electrical and Electronics Engineers. *IEEE Standard for Floating-Point Arithmetic*. Inf. téc. 754-2019. New York, NY, USA: IEEE, 2019.

[14] Wikipedia contributors. *Camino (topología)*. 2024. URL: https://es.wikipedia.org/w/index.php?title=Camino_(topolog%C3%ADa)&oldid=160275822 (visitado 21-05-2025).

Anexo

- **Programa completo:** https://github.com/j5v3s/Continuacion. Descargar, descomprimir, acceder a la carpeta con un editor de texto tipo *Visual Studio Code* y leer el archivo *readme.md* para iniciarlo. Importante tener *Python* $>$ 3.8 instalado, así como el gestor de paquetes *pip*.
- **sistematipo.ipynb:** GitHub. Si se quiere jugar con el sistema, meter el archivo en la carpeta *Tester* del programa. Mismo para los otros cuadernos de *Jupyter*.
- **sistemasimetrico.ipynb:** GitHub
- **sistema10.ipynb:** GitHub