TRABAJO FIN DE GRADO

Raíces de Polinomios

TRABAJO FIN DE GRADO PARA LA OBTENCIÓN DEL TÍTULO DE **GRADUADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES**

SEPTIEMBRE **2025**

Carlos Chacón Sánchez

DIRECTOR DEL TRABAJO FIN DE GRADO:
Ana Soledad Meroño Moreno

INDUSTRIALES
ETSII | UPM

POLITÉCNICA

"Somos lo que hacemos repetidamente.
La excelencia, entonces, no es un acto, sino un hábito."

— Aristóteles

RESUMEN

El presente TFT (Trabajo Fin de Titulación) del Grado en Ingeniería en Tecnologías Industriales, realiza un estudio en profundidad sobre algunos de los métodos iterativos más comunes utilizados en la industria y las matemáticas para el cálculo de raíces de polinomios.

El trabajo comienza con una introducción (1. Introducción), declaración de objetivos del trabajo (2. Objetivos) y descripción de la metodología seguida (3. Metodología). Estos tres apartados en su conjunto, establecen la base para la comprensión y entendimiento del trabajo, de los fines de este, y de su proceso de elaboración.

El apartado 4. Introducción a los métodos numéricos iterativos da una explicación del tipo de método a estudiar en este trabajo (métodos iterativos) y muestra aspectos relevantes al funcionamiento de estos, como los criterios de parada.

La sección 5. Método de Newton introduce este método e incluye el desarrollo matemático completo para deducir a partir de este sus algoritmos derivados, que son, el Método de Newton para Raíces Múltiples y el Método de Schröder. Seguidamente, se realizan experimentos que comparan, prueban y buscan errores en los códigos propuestos que replican estos métodos.

La parte 6. Problemas mal condicionados introduce la importancia de los problemas mal condicionados y su relación con el control de perturbaciones en los coeficientes de los polinomios, así como la relevancia de estas últimas. Posteriormente, se introducen los conceptos de número de condición absoluto y relativo. Estos son medidas matemáticas clásicas del mal condicionamiento de un problema.

Para respaldar estos conceptos, se llevan a cabo en este mismo apartado experimentos con casos bien y mal condicionados, el ejemplo más relevante de estos últimos es el Polinomio Wilkinson.

A continuación, se desarrolla y define cómo se puede preservar la multiplicidad al perturbar un polinomio haciendo uso de las variedades peyorativas y estructuras de multiplicidad.

Este es el punto de inflexión del trabajo ya que utilizando estos conceptos se puede definir la iteración de Gauss-Newton, que es especialmente útil para resolver problemas mal condicionados, y el número de condición peyorativo, que cuantifica la posibilidad de resolver problemas mal condicionados con esta iteración. También se incluyen las definiciones y ejemplos de cálculo de variedades peyorativas, números de condición peyorativos y estructuras de multiplicidad.

En el apartado 7. Algoritmos de cálculo para problemas mal condicionados se da el desarrollo matemático completo hasta llegar al Algoritmo I de Zhonggang Zeng y sus condiciones de aplicación. Además se incluyen varios ejemplos de cálculo de raíces ante polinomios de distinta índole.

Finalmente, en la sección 8. Comparativa de algoritmos para raíces múltiples se realizan distintos experimentos que muestran las ventajas y desventajas de cada método con el fin de ver cual es la sinergia entre los tres métodos para raíces múltiples y ver como se complementan entre sí.

En el apartado de 9. Resultados y Discusión se realiza el análisis necesario de los resultados obtenidos en los ejercicios para dar lugar a las conclusiones del trabajo (10.Conclusiones), de las cuales se incluye un resumen a continuación:

- **Relación entre el orden de convergencia y el número de iteraciones necesarias para convergencia (velocidad de convergencia).** Un método con mayor orden converge más rápido que uno con menor orden a misma constate de error asintótica y mismo problema.

- **Relación entre cálculo analítico y numérico de convergencia.** Los resultados de los cálculos de convergencia analíticos no se cumplen necesariamente cuando se introducen perturbaciones en los polinomios.

- **Influencia de la iteración inicial en la convergencia de los métodos.** El posicionamiento de esta puede determinar que el Método de Schröder y el Algoritmo I converjan o no. Así como provocar que el Método de Newton en raíces simples converja a una raíz no deseada.

- **Influencia de la iteración inicial z_0 en la precisión de las aproximaciones a las raíces dadas por el Algoritmo I de Zhonggang Zeng.** La elección de esta puede determinar una mayor o menor precisión en las aproximaciones a las raíces. Puede ser conveniente alejarla de la raíz en casos de número de condición peyorativo bajo y acercarla a la raíz en el caso contrario.

▪ **Convergencia exclusiva de un método.** Los tres métodos de raíces múltiples tienen casos en los que únicamente uno de ellos converge a la solución. Por ello queda demostrado que los tres algoritmos propuesto se complementan entre sí en cuanto a abordar un mayor número de casos se refiere.

Al final del trabajo se incluyen varios apartados relacionados con la normativa, que aportan información de interés relativa al trabajo y al marco ético, legal, social, económico y medioambiental.

Palabras clave: Raíces de polinomios, Derivada, Convergencia, Orden de convergencia, Algoritmo I de Zhonggang Zeng, Método de Newton, Método de Newton para Raíces Múltiples, Método de Schröder, Método iterativo, Algoritmo, Número de condición relativo, Número de condición absoluto, Número de condición peyorativo, Error.

Códigos UNESCO:

1201.13 - Polinomios

1202.09 - Funciones de una variable compleja

1203.23 - Lenguajes de programación

1206.01 - Construcción de algoritmos

1206.03 - Análisis de errores

1206.08 - Métodos iterativos

Índice

ÍNDICE DE FIGURAS

Índice de figuras

1. INTRODUCCIÓN

Este Trabajo Fin de Titulación, tiene como objeto presentar de una forma matemáticamente rigurosa algunos de los métodos numéricos más usados en la resolución de problemas en cálculo de raíces de polinomios. Para su posterior aplicación, discusión y comparación mediante códigos elaborados en el software MATLAB, con el objetivo de contribuir a la resolución de este tipo de casos de estudio en el ámbito de la ingeniería y las matemáticas.

Durante el desarrollo del trabajo se han tomado teoremas, definiciones, demostraciones, propiedades y corolarios de diversos libros y otros recursos literarios relacionados con la temática del trabajo, con el fin de respaldar y dar una demostración completa de los métodos utilizados y sus propiedades más relevantes.

Además, se presentan los códigos elaborados con sus correspondientes salidas de MATLAB que muestran su correcta ejecución y los resultados obtenidos. Cuando se comentan los resultados obtenidos, se hace referencia a la convergencia con oraciones como "el método converge", de ahora en adelante se sobreentiende que con este tipo de frases el alumno se refiere a una convergencia correcta hacia la raíz del polinomio, salvo que se mencione lo contrario.

A modo introductorio, se presentan los siguientes resultados esenciales en el ámbito relativo a este trabajo, los cuales establecen la base para todos los desarrollos que se realizarán posteriormente:

Teorema 1.1. Teorema Fundamental del Álgebra [28]. Todo polinomio complejo tiene alguna raíz.

Si p es un polinomio de grado $n > 1$, tiene alguna raíz $z_1 \in \mathbb{C}$; entonces $p(x) = (x - z_1)q(x)$, donde q es un polinomio de grado $n - 1$ que también tendrá alguna raíz $z_2 \in \mathbb{C}$; se puede aplicar el teorema recursivamente hasta llegar a un polinomio de grado 0 (es decir, constante), lo que permite concluir el siguiente resultado:

Corolario 1.1 [28]. Todo polinomio complejo de grado n puede escribirse como producto de n factores lineales, es decir:

$$p(x) = \lambda(x - z_1)(x - z_2)...(x - z_n) \tag{1.1}$$

en donde $\lambda \in \mathbb{C}$ es el coeficiente principal de p y $z_1, z_2, ..., z_n \in \mathbb{C}$ son sus raíces.

Para proceder a la lectura y comprensión del trabajo, se debe tener en cuenta que todos los métodos utilizados en este trabajo parten de una iteración inicial que se supone conocida dado que, como los polinomios están factorizados, es sencillo escogerla cerca de la raíz. En casos reales de estudio, no se suelen conocer las raíces del polinomio, por lo que para escoger el punto de partida del algoritmo puede ser conveniente realizar algunas iteraciones del Método de la Bisección. Este algoritmo se encuentra descrito en detalle en el Anexo A. Método de la Bisección.

Por último, cabe destacar que todo el trabajo se encuentra referenciado con hipervínculos internos para facilitar la navegación del documento. Por ello, cualquier sección, teorema, propiedad, corolario, definición, fuente bibliográfica o ejemplo mencionado en el trabajo, tiene incluido un enlace en su texto que lleva directamente al elemento mencionado.

2. OBJETIVOS

El cálculo de raíces de polinomios es uno de los problemas matemáticos más antiguos y estudiados, la investigación al respecto de este tópico comenzó aproximadamente en el 2000 A.C., por parte de los Babilonios (dato tomado de la fuente [27]), y, aunque a lo largo de toda la historia se han hecho grandes avances para resolver numerosos casos, a día de hoy hay polinomios cuyas raíces no se han logrado calcular con un error asumible.

El primer objetivo de este trabajo es poner en conjunto la teoría matemática más relevante relativa al Método de Newton, Método de Newton para Raíces Múltiples, Método de Schröder y Algoritmo I de Zhonggang Zeng.

La siguiente meta de este documento es comprobar mediante ejemplos prácticos, ya sean ejemplos de cálculo analítico o casos abordados mediante códigos de MATLAB, si los desarrollos teóricos propuestos se verifican en todos los casos o si por el contrario hay excepciones a estos. Además, se analiza como se comportan ciertos polinomios ante los métodos.

Por último, en base a los dos puntos mencionados anteriormente, el trabajo concluye con una serie de ejercicios en los que se muestra ante que tipo de polinomios son más efectivos los algoritmos, con el fin de servir como referencia a la hora de escoger un método para hallar las raíces de un polinomio dado.

3. METODOLOGÍA

Una vez escogidos por parte del tutor y el alumno los métodos a estudiar, se procedió al desarrollo del cuerpo del trabajo de la forma en la que se explica a continuación.

El trabajo consta principalmente de tres formatos de contenido: desarrollos matemáticos teóricos, ejemplos prácticos de cálculo analítico y ejemplos prácticos de cálculo con MATLAB. Estos tres tipos de contenido se han escogido en base a la presentación de la referencia bibliográfica [27], que se ha usado como guía en la elaboración de este TFT.

La metodología de elaboración de este trabajo ha sido entonces, tomar algunos de los conceptos de esta presentación, desarrollarlos de una forma notablemente más extensa e incluir nuevos apartados que se han creído convenientes de forma que el tópico de estudio quede explicado de manera más completa.

Para el desarrollo de cada sección, se consultaron las fuentes expuestas en la bibliografía como ayuda para la realización de los desarrollos teóricos y posteriormente se realizaron los ejercicios correspondientes que complementan dichos desarrollos.

4. INTRODUCCIÓN A LOS MÉTODOS NUMÉRICOS ITERATIVOS

Todos los métodos utilizados en este trabajo son métodos numéricos iterativos. Un método numérico iterativo es un algoritmo que parte de una iteración inicial dada y actualiza esta en cada iteración en base a un criterio que varía dependiendo del método y el valor en la iteración anterior, hasta encontrar una solución (raíz) suficientemente cercana a la exacta del problema que se está tratando.

Estos métodos funcionan en un bucle que, cada vez que se ejecuta, aplica el paso (variación entre una iteración y su consecutiva) del método. Para ahorrar potencia computacional y que el programa no realice el máximo número de iteraciones que puede llevar a cabo, todo programa que replique uno de estos algoritmos debe tener establecido un criterio de parada que le indique cuando está lo suficientemente cerca de la solución esperada.

4.1. Criterios de parada

Los criterios de parada más comunes son, para una cantidad $\epsilon > 0$ (tolerancia), siendo p_n el valor de la iteración enésima del método y p_{n-1} el valor de la iteración anterior a p_n:

Por error absoluto entre iteraciones consecutivas [2]:

$$|p_n - p_{n-1}| < \epsilon \tag{4.1}$$

Por error relativo entre iteraciones consecutivas [2]:

$$\frac{|p_n - p_{n-1}|}{|p_n|} < \epsilon; p_n \neq 0 \tag{4.2}$$

Por evaluación de la función en el valor de la iteración [2]:

$$|f(p_n)| < \epsilon \tag{4.3}$$

En este trabajo se utiliza principalmente el criterio descrito en la ecuación (4.1) dado que el de error relativo puede fallar si p_n es una cantidad cercana a cero y el de evaluación de función puede requerir demasiada potencia computacional cuando trabajamos con ciertas funciones complicadas, como puede ser un polinomio de grado alto.

Sin embargo, se debe tener en cuenta que al usar este criterio se puede detener la ejecución del método aún no habiendo llegado a la solución deseada, ya que se puede dar el caso en que el método avance muy poco entre dos iteraciones y consecuentemente se detenga.

En lo que respecta a este trabajo, esto no será un problema. Dado el carácter experimental de los ejercicios realizados, se conocen las raíces de los polinomios antes de la realización de estos, por lo que se puede verificar si el resultado obtenido es correcto o no.

En un caso de estudio en el que no se conozcan las raíces será conveniente aplicar el criterio de evaluación de la función únicamente para la última iteración, de forma que si este valor de la función es prácticamente nulo, la raíz obtenida será correcta.

5. MÉTODO DE NEWTON

5.1. Definición del Método de Newton

El método de Newton, también conocido como método de Newton-Raphson, es uno de los métodos más utilizados y efectivos en lo que a encontrar raíces de polinomios se refiere. Se define de la siguiente forma:

Definición 5.1. Método de Newton [2]. Supóngase que $f \in C^2$ en $[a, b]$. Sea $x^* \in [a, b]$ tal que $f'(x^*) \neq 0$ y $|p - x^*| < \epsilon$ (siendo p la raíz que se desea hallar), donde ϵ es una cantidad baja, del orden, por ejemplo de 10^{-10}. Considérese el primer polinomio de Taylor de $f(x)$ expandido en torno a x^*:

$$f(x) = f(x^*) + (x - x^*)f'(x^*) + \frac{(x - x^*)^2}{2}f''(\xi(x)); \tag{5.1}$$

donde $\xi \in (p, x^*)$ o $\xi \in (x^*, p)$. Como $f(p) = 0$, la ecuación (5.1) evaluada en $x = p$:

$$0 = f(x^*) + (p - x^*)f'(x^*) + \frac{(p - x^*)^2}{2}f''(\xi(p)); \tag{5.2}$$

Como bien se ha definido anteriormente, $|p - x^*|$ es una cantidad baja, consecuentemente, $(p - x^*)^2$ será una cantidad aún menor, por lo que se puede despreciar el término que contiene $(p - x^*)^2$, obteniendo así:

$$0 \approx f(x^*) + (p - x^*)f'(x^*) \tag{5.3}$$

Y, consecuentemente:

$$p \approx x^* - \frac{f(x^*)}{f'(x^*)} \tag{5.4}$$

La ecuación (5.4) es la base para el método de Newton, que comienza con una aproximación p_0 y genera la serie $(p_n)_{n\in\mathbb{N}}$ mediante:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} \tag{5.5}$$

Nótese que este algoritmo está diseñado para raíces simples. Más adelante se mostrará como adaptarlo para raíces múltiples.

5.2. Orden de convergencia en esquemas iterativos

En los esquemas iterativos, se necesita una forma de cuantificar con que velocidad converge el método, es decir, como de rápido llega el algoritmo a la solución. Esta forma de cuantificar es el orden convergencia, que se define a continuación.

Definición 5.2. Orden de convergencia [2]. Supóngase que $(p_n)_{n\in\mathbb{N}}$ es una serie que converge a p, con $p_n \neq p$; $\forall n \in \mathbf{N}$. Si la constantes α y λ positivas existen tal que:

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^\alpha} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^\alpha} = \lambda \tag{5.6}$$

Entonces $(p_n)_{n\in\mathbb{N}}$ converge a p con orden α con constante de error asintótica λ.

Dos de los casos más relevantes a este trabajo son la convergencia lineal($\alpha = 1$) y la convergencia cuadrática ($\alpha = 2$).

Ejemplo 5.1. Demostración del orden de convergencia cuadrático del Método de Newton para raíces simples

Partiendo de la iteración del Método de Newton:

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)} \tag{5.7}$$

Sea p la raíz exacta de $f(x)$, es decir, $f(p) = 0$. Se define el error en la iteración n como:

$$e_n = p_n - p \tag{5.8}$$

Para demostrar el orden de convergencia de Newton-Raphson, se utiliza la definición descrita en la expresión (5.6). Se quiere demostrar que para $\alpha = 2$ se obtiene $\lambda > 0$, que es la condición para que el método tenga convergencia cuadrática.

Expandiendo $f(p_n)$ en una serie de Taylor alrededor de p, se tiene:

$$f(p_n) = f(p + e_n) = f(p) + f'(p)e_n + \frac{f''(p)}{2}e_n^2 + O(e_n^3) \tag{5.9}$$

Dado que $f(p) = 0$:

$$f(p_n) = f'(p)e_n + \frac{f''(p)}{2}e_n^2 + O(e_n^3) \tag{5.10}$$

De manera similar, se expande $f'(p_n)$ en una serie de Taylor alrededor de p:

$$f'(p_n) = f'(p) + f''(p)e_n + O(e_n^2) \tag{5.11}$$

Sustituyendo estas expresiones en la ecuación de Newton:

$$p_{n+1} = p_n - \frac{f'(p)e_n + \frac{f''(p)}{2}e_n^2 + O(e_n^3)}{f'(p) + f''(p)e_n + O(e_n^2)} = p_n - e_n\frac{1 + \frac{f''(p)}{2f'(p)}e_n + O(e_n^2)}{1 + \frac{f''(p)}{f'(p)}e_n + O(e_n^2)} \tag{5.12}$$

Nótese que en el numerador de la expresión anterior se ha sacado factor común e_n, esto es posible gracias a que la dependencia de e_n^3 de la función $O(e_n^3)$ quiere decir que este término será proporcional a potencias de e_n^3. También cabe resaltar que, los términos de error quedan multiplicados por $1/f'(p)$.

Teniendo esto en cuenta y que cualquier cociente de la forma $\frac{1+a}{1+b} \approx (1+a)\frac{1}{1+b} \overset{\text{Taylor}}{\approx} (1+a)(1-b) \approx 1 + a - b - ab \approx 1 + a - b$; siempre y cuando $|a| \ll 1; |b| \ll 1$, la ecuación queda:

$$p_{n+1} = p_n - e_n\left(1 - \frac{f''(p)}{2f'(p)}e_n + O(e_n^2)\right) = p_n - e_n + \frac{f''(p)}{2f'(p)}e_n^2 + O(e_n^3). \tag{5.13}$$

Por lo tanto, el error en la iteración siguiente es:

$$e_{n+1} = \frac{f''(p)}{2f'(p)}e_n^2 + O(e_n^3). \tag{5.14}$$

Tomando el límite:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^2} = \left|\frac{f''(p)}{2f'(p)}\right| > 0 \tag{5.15}$$

Esto demuestra que el método de Newton-Raphson tiene orden de convergencia 2 o convergencia cuadrática.

El siguiente ejemplo analiza si un método iterativo de convergencia cuadrática converge más rápidamente que uno de orden de convergencia lineal:

Ejemplo 5.2 [2].

Supóngase que $(p_n)_{n\in\mathbb{N}}$ converge linealmente a 0 tal que:

$$\lim_{n\to\infty} \frac{|p_{n+1}|}{|p_n|} = 0.5 \tag{5.16}$$

Supóngase también, que $(p_n^*)_{n\in\mathbb{N}}$ converge cuadráticamente a 0 tal que:

$$\lim_{n\to\infty} \frac{|p_{n+1}^*|}{|p_n^*|^2} = 0.5 \tag{5.17}$$

Por simplicidad se toma:

$$\frac{|p_{n+1}|}{|p_n|} \approx 0.5 \tag{5.18}$$

$$\frac{|p_{n+1}^*|}{|p_n^*|^2} \approx 0.5 \tag{5.19}$$

Para el caso de convergencia lineal se tiene:

$$|p_n - 0| = |p_n| \approx 0.5|p_{n-1}| \approx ... \approx (0.5)^n|p_0| \tag{5.20}$$

Y para el cuadrático:

$$|p_n^* - 0| = |p_n^*| \approx 0.5|p_{n-1}^*|^2 \approx 0.5[0.5|p_{n-2}^*|^2]^2 = 0.5^3|p_{n-2}^*|^4 \approx ... \approx (0.5)^{2^n-1}|p_0^*|^{2^n} \tag{5.21}$$

Si se toma, por ejemplo, $|p_0| = |p_0^*| = 1$, para todos los valores de n se obtienen valores mayores de $|p_n|$ que de $|p_n^*|$.

Por ejemplo, para n=5 se obtiene $|p_n| = 3.125 * 10^{-2}$ y $|p_n^*| = 4.657 * 10^{-10}$, que demuestra claramente que el método con mayor orden de convergencia converge a la solución con un menor número de iteraciones ante una misma raíz, iteración inicial y constante de error asintótica.

5.3. Raíces múltiples en el Método de Newton

Hasta ahora, en los apartados previos se ha asumido que la solución de los métodos eran raíces de multiplicidad 1. Este no será siempre el caso, dado que se pueden encontrar multiplicidades mayores cuando se calculen las raíces de un polinomio. Por ello, en este apartado se estudian estos casos partiendo de la siguiente definición.

Definición 5.3 [2]. Una solución p de $f(x) = 0$ es un cero de multiplicidad m de f si se puede escribir, para $x \neq p$, $f(x) = (x-p)^m g(x)$, donde $\lim_{x\to p} g(x) \neq 0$.

Teorema 5.1 [2]. $f \in C^1[a,b]$ tiene un cero de multiplicidad 1 en $p \in (a,b)$ si y solo si $f(p) = 0$ y $f'(p) \neq 0$.

Este resultado es muy útil como apoyo para enunciar el siguiente teorema.

Teorema 5.2 [2]. $f \in \mathbf{C}^m[a,b]$ tiene un cero de multiplicidad m en $p \in (a,b)$ si y solo si $f(p) = f'(p) = f''(p) = ... = f^{m-1)}(p) = 0$ y $f^{m)}(p) \neq 0$.

Teniendo en cuenta todos los resultados anteriores, a continuación se estudiará como lidiar con polinomios de raíces múltiples en el Método de Newton.

5.3.1. Método de Newton para Raíces Múltiples

Para los casos en los que se conoce la multiplicidad, se puede expresar la derivada como:

$$f'(x) = m(x-p)^{m-1}g(x) + (x-p)^m g'(x); m > 1, \tag{5.22}$$

Teniendo en cuenta la ecuación (5.22), evaluando $f(x)/f'(x)$ se obtiene:

$$\frac{f(x)}{f'(x)} \approx \frac{(x-p)^m}{m(x-p)^{m-1}} = \frac{1}{m}(x-p) \tag{5.23}$$

Nótese que el término del denominador $(x-p)^m g'(x)$ se desprecia ya que cerca de la raíz este será notablemente menor al tener una mayor potencia de $(x-p)$. Por lo que:

$$p_n = p_{n-1} - m\frac{f(p_{n-1})}{f'(p_{n-1})} \tag{5.24}$$

5.3.2. Método de Schröder

Para la deducción de este método (tomada de la referencia bibliográfica [2]), en primer lugar se define $\mu(x) = \frac{f(x)}{f'(x)}$. Si p es un cero de multiplicidad m, entonces $f(x) = (x-p)^m g(x)$, por ello:

$$\mu(x) = \frac{(x-p)^m g(x)}{m(x-p)^{m-1}g(x) + (x-p)^m g'(x)} = \frac{(x-p)g(x)}{mg(x) + (x-p)g'(x)} \tag{5.25}$$

Es claro ver que como $g(p) \neq 0$, $\mu(x)$ tiene un cero de multiplicidad 1 en p. Ahora, aplicando el método de Newton a $\mu(x)$ se obtiene:

$$q(x) = x - \frac{\mu(x)}{\mu'(x)} = x - \frac{f(x)f'(x)}{[f'(x)]^2 - f(x)f''(x)} \tag{5.26}$$

Por la forma en la que se ha definido $\mu(x)$, esta tendrá un cero simple en la raíz múltiple p de $f(x)$. Por lo que la iteración queda:

$$p_n = p_{n-1} - \frac{f(p_{n-1})f'(p_{n-1})}{[f'(p_{n-1})]^2 - f(p_{n-1})f''(p_{n-1})} \tag{5.27}$$

Esta fórmula es comúnmente conocida como el Método de Schröder, que suele ser útil al tratar con raíces múltiples de las cuales no se conoce la multiplicidad.

Ejemplo 5.3. Demostración del orden de convergencia cuadrático del Método de Newton para Raíces Múltiples y Método de Schröder [1].

Para la demostración del orden de convergencia cuadrático del Método de Newton para Raíces Múltiples y Método de Schröder se introduce la siguiente propiedad.

Propiedad 5.1 [1]. Si $\phi \in C^{\gamma+1}(I)$ siendo I un entorno adecuado de p y un entero $\gamma \geq 0$, y si $\phi^{(i)}(p) = 0$ para $0 \leq i \leq \gamma$ y $\phi^{(\gamma+1)}(p) \neq 0$, entonces el método de punto fijo con función de iteración ϕ tiene orden de convergencia $\gamma + 1$ y:

$$\lim_{n\to\infty} \frac{p_{n+1} - p}{(p_n - p)^{\gamma+1}} = \frac{\phi^{(\gamma+1)}(p)}{(\gamma+1)!}, \qquad p \geq 0, \tag{5.28}$$

Demostración [1]. La expansión en serie de Taylor de ϕ en torno a $x = p$ es:

$$p_{n+1} - p = \sum_{i=0}^{\gamma} \frac{\phi^{(i)}(p)}{i!}(p_n - p)^i + \frac{\phi^{(\gamma+1)}(\eta)}{(\gamma+1)!}(p_n - p)^{\gamma+1}, \tag{5.29}$$

Para un cierto η entre p_n y p. De aquí se deduce, usando que $\phi^{(i)} = 0$; y tomando $\lim_{n\to\infty}$:

$$\lim_{n\to\infty} \frac{p_{n+1} - p}{(p_n - p)^{\gamma+1}} = \lim_{n\to\infty} \frac{\phi^{(\gamma+1)}(\eta)}{(\gamma+1)!} = \frac{\phi^{(\gamma+1)}(p)}{(\gamma+1)!}. \tag{5.30}$$

Nótese que de acuerdo con la notación utilizada en este trabajo $\alpha = \gamma + 1$.

El Método de Schröder y el Método de Newton para Raíces Múltiples son métodos iterativos de punto fijo (para más información acerca de esto, se recomienda consultar la referencia bibliográfica [2]). Por ello se puede aplicar esta propiedad a las funciones de iteración ϕ particularizadas para cada polinomio. Siempre y cuando cada caso concreto cumpla adecuadamente las condiciones de esta propiedad, se puede asegurar que ambos métodos tienen orden de convergencia cuadrático [**?**] de acuerdo con la siguiente expresión:

$$\lim_{n\to\infty} \frac{p_{n+1} - p}{(p_n - p)^2} = \frac{\phi''(p)}{2!} \tag{5.31}$$

Siendo $\phi(x)$ para el Método de Schröder:

$$\phi(x) = x - \frac{f(x)f'(x)}{[f'(x)]^2 - f(x)f''(x)} \tag{5.32}$$

Y para el Método de Newton para Raíces Múltiples:

$$\phi(x) = x - m\frac{f(x)}{f'(x)} \tag{5.33}$$

5.4. Fallos en los algoritmos para raíces múltiples derivados del Método de Newton

Ejemplo 5.4.

En este ejemplo se estudia el comportamiento del polinomio $p(x) = (x-2)^7(x-3)(x-4)$ (polinomio tomado de la referencia bibliográfica [27]). El siguiente código aplica el Método de Newton para Raíces Múltiples a $p(x)$:

```
1
2  %Método de Newton para Raíces Múltiples
3
4  %Declaración de variable x simbólica y del polinomio
5  syms x
6  p(x)=(x-2)^7*(x-3)*(x-4)
7
8  %Se trabaja con el polinomio en forma expandida
9  q=expand(p)
10
11 %Derivada del polinomio expandido
12 dq=diff(q)
13
14 %Se desea estudiar la convergencia del Método de Newton para Raíces
15 %Múltiples para la raíz a=2
16
17 m=7; % multiplicidad
18 a=2; % raíz exacta
19
20 % Transformación del polinomio y de su derivada de funciones simbólicas
21 % a funciones numéricas de MATLAB
22 Q=matlabFunction(q)
23 DQ=matlabFunction(dq)
24
25 x0=1.999 %valor inicial cerca de a
26
27 E=[] % vector de errores
28 N=100 % número de máximo iteraciones
29
30 for i=1:N
31     i
32     xi=x0-m.*Q(x0)./DQ(x0) %Newton modificado
33
34     E(i)=a-xi; % actualización del vector de errores
35     x0=xi;
36
37 end
38
39 %Gráfica de errores relativos por iteración:
40
41 plot(1:N,abs(E)./a)
```

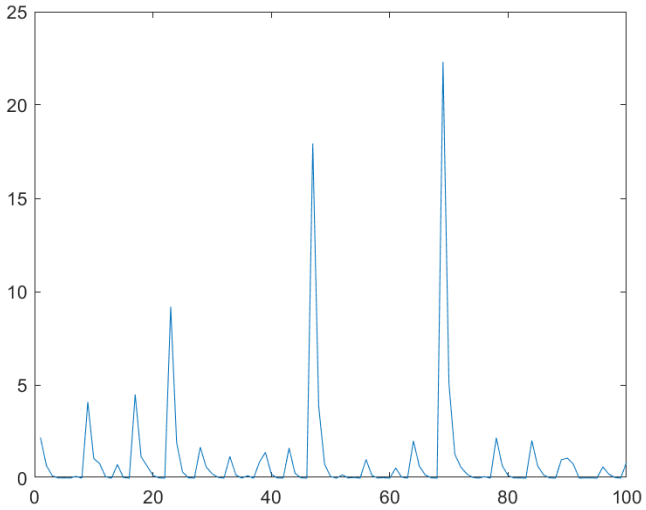

Figura 5.1: Errores relativos en las iteraciones aplicando el Método de Newton para raíces Múltiples en 100 iteraciones a $p(x) = (x-2)^7(x-3)(x-4)$

Si se cambia el valor de iteraciones a `N=200`:

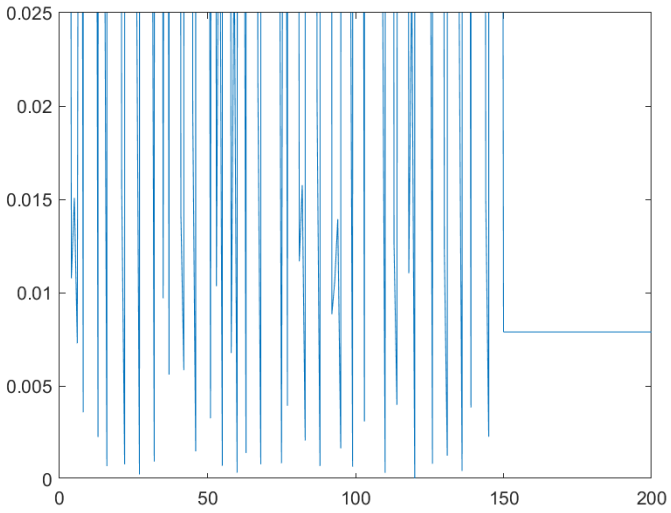

Figura 5.2: Errores relativos en las iteraciones aplicando el Método de Newton para raíces Múltiples en 200 iteraciones a $p(x) = (x-2)^7(x-3)(x-4)$

Se observa que en primer lugar el error oscila y que después se mantiene constante, por lo que el algoritmo no converge a la solución.

A continuación se estudia que ocurre al aplicar el Método de Schröder al mismo polinomio $p(x)$ mediante el siguiente código:

```
% Método de Schröder

% Declaración de variable x simbólica, del polinomio y de su primera y
%segunda derivada
syms x
f_sym = (x - 2)^7 * (x - 3) * (x - 4);
f1_sym = diff(f_sym, x);
```

```
f2_sym = diff(f1_sym, x);

% Transformación a funciones numéricas de MATLAB
f = matlabFunction(f_sym);
f1 = matlabFunction(f1_sym);
f2 = matlabFunction(f2_sym);

x0 = 1.999;             % iteración inicial
tol = 1e-10;            % tolerancia
max_iter = 100;         % número máximo de iteraciones

for i = 1:max_iter
    fx = f(x0);
    f1x = f1(x0);
    f2x = f2(x0);

    denom = f1x^2 - fx * f2x;

    x1 = x0 - (fx * f1x) / denom; % Método de Schröder

    if abs(x1 - x0) < tol % Criterio de parada
        fprintf('Convergencia alcanzada en %d iteraciones\n', i);
        fprintf('Raiz encontrada: %.12f\n', x1);
        break
    end

    x0 = x1;

    if i == max_iter
        fprintf('No se alcanzo la convergencia en %d iteraciones\n',
            ↪ max_iter);
        fprintf('Aproximacion final: %.12f\n', x1);
    end
end
```

```
Convergencia alcanzada en 3 iteraciones
Raiz encontrada: 2.000000000000
```

Figura 5.3: Resultado de aplicar el Método de Schröder al polinomio $p(x) = (x-2)^7(x-3)(x-4)$

En este caso el Método de Schröder si converge.

Ejemplo 5.5.

El Método de Newton en raíces simples, ante ciertos polinomios, como $p(x) = (x-1)(x-2)...(x-10)$, calcula otra raíz del polinomio diferente a la deseada cuando se escoge el punto de partida del algoritmo cerca de una solución. El siguiente programa de MATLAB muestra esto:

```
%Método de Newton

coeffs = poly(1:10);  % genera coeficientes de (x-1)(x-2)...(x-10) a
                      % partir de sus raíces

%Construcción del polinomio y su derivada
f = @(x) polyval(coeffs, x);
df = @(x) polyval(polyder(coeffs), x);

```

```
x0 = 5.6;  %iteración inicial
tol = 1e-14; %tolerancia
maxIter = 20; %número máximo de iteraciones
x = x0;
errors = []; %vector de errores

fprintf('\nNewton CLASICO - raices simples pero alto grado (NO CONVERGE)
    ↪ :\n');
for i = 1:maxIter
    fx = f(x);
    dfx = df(x);

    x_new = x - fx / dfx; % Método de Newton

    err = min(abs(x_new - (1:10))); % error mínimo con respecto a
                                    % raíces exactas
    errors(end+1) = err;

    fprintf('Iter %2d: x = %.15f, Error a raiz mas cercana = %.2e\n', i,
        ↪  x_new, err);

    if err < tol % criterio de parada
        break;
    end
    x = x_new;
end
```

```
Newton CLASICO - raíces simples pero alto grado (NO CONVERGE):
Iter  1: x = 6.619480620741639, Error a raíz más cercana = 3.81e-01
Iter  2: x = 7.894927586017596, Error a raíz más cercana = 1.05e-01
Iter  3: x = 8.017914703415128, Error a raíz más cercana = 1.79e-02
Iter  4: x = 8.000328717637476, Error a raíz más cercana = 3.29e-04
Iter  5: x = 8.000000117449909, Error a raíz más cercana = 1.17e-07
Iter  6: x = 8.000000000315291, Error a raíz más cercana = 3.15e-10
Iter  7: x = 8.000000000305867, Error a raíz más cercana = 3.06e-10
Iter  8: x = 8.000000000196151, Error a raíz más cercana = 1.96e-10
Iter  9: x = 7.999999999569264, Error a raíz más cercana = 4.31e-10
Iter 10: x = 8.000000000009980, Error a raíz más cercana = 9.98e-12
Iter 11: x = 8.000000000048091, Error a raíz más cercana = 4.81e-11
Iter 12: x = 7.999999999778950, Error a raíz más cercana = 2.21e-10
Iter 13: x = 8.000000000187143, Error a raíz más cercana = 1.87e-10
Iter 14: x = 7.999999999957592, Error a raíz más cercana = 4.24e-11
Iter 15: x = 7.999999999796597, Error a raíz más cercana = 2.03e-10
Iter 16: x = 8.000000000038021, Error a raíz más cercana = 3.80e-11
Iter 17: x = 7.999999999936573, Error a raíz más cercana = 6.34e-11
Iter 18: x = 8.000000000461736, Error a raíz más cercana = 4.62e-10
Iter 19: x = 7.999999999939160, Error a raíz más cercana = 6.08e-11
Iter 20: x = 7.999999999921420, Error a raíz más cercana = 7.86e-11
```

Figura 5.4: Resultados de aplicar el Método de Newton para raíces simples al polinomio $p(x) = (x-1)(x-2)...(x-10)$

El método converge a $x = 8$ en lugar de $x = 6$.

Ejemplo 5.6.

En este ejemplo se muestra como se comporta el método de Schröder ante un polinomio con multiplicidades altas como $p(x) = (x-2)^9$.

```
% Método de Schröder

% Declarar simbólicamente la funcion y sus derivadas
syms x
f_sym = (x - 2)^9;
f1_sym = diff(f_sym, x);
f2_sym = diff(f1_sym, x);

% Transformación a funciones numéricas de MATLAB
f = matlabFunction(f_sym);
f1 = matlabFunction(f1_sym);
f2 = matlabFunction(f2_sym);

x0 = 1.4;               % iteración inicial
tol = 1e-10;            % tolerancia
max_iter = 10000;         % número maximo de iteraciones

for i = 1:max_iter
    fx = f(x0);
    f1x = f1(x0);
    f2x = f2(x0);

    denom = f1x^2 - fx * f2x;

    x1 = x0 - (fx * f1x) / denom; % Método de Schröder

    if abs(x1 - x0) < tol
        fprintf('Convergencia alcanzada en %d iteraciones\n', i);
        fprintf('Raiz encontrada: %.12f\n', x1);
        break
    end

    x0 = x1;

    if i == max_iter
        fprintf('No se alcanzo la convergencia en %d iteraciones\n',
            ↪ max_iter);
        fprintf('Aproximacion final: %.12f\n', x1);
    end
end
```

```
Convergencia alcanzada en 2 iteraciones
Raiz encontrada: 2.000000000000
```

Figura 5.5: Resultados de aplicar el Método de Schröder a $p(x) = (x-2)^9$ con $x_0 = 1.4$

Si se edita el código para que $x_0 = 1.8$, se obtiene:

```
No se alcanzo la convergencia en 10000 iteraciones
Aproximacion final: NaN
```

Figura 5.6: Resultados de aplicar el Método de Schröder a $p(x) = (x-2)^9$ con $x_0 = 1.8$

La convergencia del método depende claramente de la iteración inicial.

6. PROBLEMAS MAL CONDICIONADOS

6.1. Introducción a los problemas mal condicionados

Un problema mal condicionado es aquel en el cual pequeñas perturbaciones en los datos de entrada producen grandes variaciones en la solución. Esto significa que el problema es numéricamente inestable y difícil de resolver con precisión mediante métodos computacionales. Una buena forma de entender que es un problema mal condicionado es seguir el razonamiento del siguiente ejemplo.

Ejemplo 6.1 [27].

Sea $f(x) = (x-2)^3$. Si se aplica a la función la iteración del Método de Newton:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{(x_0-2)^3}{3(x_0-2)^2} = x_0 - \frac{(x_0-2)}{3} \tag{6.1}$$

Es claro ver que:

$$(x_1 - 2) = \frac{2}{3}(x_0 - 2) \tag{6.2}$$

Y que:

$$(x_2 - 2) = \frac{2}{3}(x_1 - 2) = (\frac{2}{3})^2(x_0 - 2) \tag{6.3}$$

Generalizando la expresión para la iteración k-ésima y, haciendo k tender a infinito:

$$\lim_{k\to\infty}(x_k - 2) = \lim_{k\to\infty}(\frac{2}{3})^k(x_0 - 2) = 0 \tag{6.4}$$

Si bien analíticamente este resultado es correcto, a continuación se demuestra que numéricamente no se llega al mismo resultado.

Un algoritmo numérico encuentra una solución exacta a un problema ligeramente perturbado del original. En este caso, el polinomio perturbado sería (en su forma expandida):

$$f(x) = (x-2)^3 + \mu(x-2)^2 + \eta(x-2) + \epsilon \tag{6.5}$$

El cociente$f(x)/f'(x)$queda:

$$\frac{(x-2)^3 + \mu(x-2)^2 + \eta(x-2) + \epsilon}{3(x-2)^2 + \mu'(x-2) + \eta'} \tag{6.6}$$

Cuando el algoritmo esté cerca de hallar la solución, la expresión anterior se simplificará a:

$$\frac{f(x)}{f'(x)} \approx \frac{\epsilon}{\eta'} \tag{6.7}$$

El valor de los parámetros de perturbación variará dependiendo del caso, por lo que este cociente será un número ciertamente aleatorio. Por ello, hallar las raíces de este polinomio con el Método de Newton es un problema mal condicionado dada su dificultad para resolverlo numéricamente con este algoritmo.

6.2. Importancia del control de perturbaciones

Si bien todo lo redactado anteriormente matemáticamente tiene sentido, es normal que surja la siguiente pregunta: ¿Por qué perturbar un polinomio si esto se aleja de obtener una solución del problema original? Esta pregunta tiene tres principales respuestas en el ámbito de la ingeniería.

En primer lugar, en un problema de ingeniería real, no se conocen con exactitud los coeficientes de un polinomio que modela un determinado proceso, por ello, es muy importante saber como se comporta el polinomio ante pequeñas perturbaciones, para así saber con que precisión se deben ajustar los coeficientes experimentalmente.

Además, los softwares de cálculo numérico, como MATLAB, trabajan en punto flotante, lo que puede introducir errores de redondeo, pudiendo devolver así el algoritmo soluciones incorrectas si el polinomio ha sido excesivamente perturbado por el software.

Por último, saber como de sensible es un polinomio a pequeñas perturbaciones permite mejorar los métodos numéricos actuales de forma que se adapten mejor a funciones con determinadas características, brindando así resultados más precisos.

Estos tres enfoques anteriores muestran claramente que el estudio de perturbaciones en polinomios ya conocidos es crucial a la hora de utilizar métodos numéricos en la resolución de problemas reales.

6.3. Número de condición

Cualquiera de los problemas relativos a este trabajo se puede expresar como una función f: $X \to Y$ donde X e Y son espacios normados de datos y soluciones, respectivamente. Consecuentemente, todas las normas utilizadas en el siguiente razonamiento, son las de dichos espacios.

Para entender como lidiar con problemas mal condicionados, se debe conocer el número de condición del problema. Hay dos tipos de número de condición, el absoluto y el relativo.

Definición 6.1. Número de condición absoluto [8].

$$\hat{\kappa} = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|} = \sup_{\delta x} \frac{\|\delta f\|}{\|\delta x\|} \approx \|J(x)\| \tag{6.8}$$

Donde δx es una pequeña perturbación en x, $\delta f = f(x + \delta x) - f(x)$ y $J(x)$ es la matriz Jacobiana de f. Nótese que la segunda expresión es solo válida si δf y δx son de magnitud infinitesimal y la tercera es válida si f es diferenciable.

Definición 6.2. Número de condición relativo [8].

$$\kappa = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|/\|f(x)\|}{\|\delta x\|/\|x\|} = \sup_{\delta x} \frac{\|\delta f\|/\|f(x)\|}{\|\delta x\|/\|x\|} \approx \frac{\|J(x)\|}{\|f(x)\|/\|x\|} \tag{6.9}$$

Tanto la segunda como la segunda como la tercera expresión se pueden usar bajo las mismas condiciones mencionadas anteriormente.

Para tener una referencia, un problema bien condicionado debe tener un número de condición relativo $\kappa \leq 10^2$ y un problema mal condicionado tendrá $\kappa \geq 10^6$, aproximadamente. [8].

En la práctica, aplicar estas definiciones puede ser complicado, por lo que en los casos contemplados en este trabajo (polinomios), se calculará el número de condición estudiando cómo una pequeña perturbación en uno de los coeficientes del polinomio (a_i) afecta a una raíz x_j.

Podemos expresar el polinomio de la siguiente forma:

$$p(x) = \sum_{k=0}^{n} a_k x^k \tag{6.10}$$

Si sustituimos x_j (raíz) en la expresión anterior:

$$p(x_j) = \sum_{k=0}^{n} a_k x_j^k = 0 \tag{6.11}$$

Diferenciando implícitamente la ecuación (6.11) con respecto a un cierto coeficiente a_i:

$$\frac{d}{da_i} p(x_j) = \frac{\partial p}{\partial a_i} + \frac{\partial p}{\partial x_j} \cdot \frac{dx_j}{da_i} = 0 \tag{6.12}$$

Calculamos cada término:

$$\frac{\partial}{\partial a_i} p(x_j) = x_j^i, \quad \frac{\partial}{\partial x_j} p(x_j) = p'(x_j) \tag{6.13}$$

Sustituyendo en la ecuación (6.12):

$$x_j^i + p'(x_j) \cdot \frac{dx_j}{da_i} = 0 \tag{6.14}$$

Despejamos $\frac{dx_j}{da_i}$:

$$\frac{dx_j}{da_i} = -\frac{x_j^i}{p'(x_j)} \tag{6.15}$$

Ante una pequeña perturbación δa_i, el cambio en la raíz x_j es entonces, aproximadamente:

$$\delta x_j \approx -\frac{x_j^i}{p'(x_j)} \cdot \delta a_i \tag{6.16}$$

Haciendo uso de la definición de número de condición relativo dada en la expresión (6.9) se concluye que:

$$\kappa = \left| \frac{\delta x_j}{x_j} \Big/ \frac{\delta a_i}{a_i} \right| = \left| -\frac{a_i x_j^{i-1}}{p'(x_j)} \right| = \frac{a_i x_j^{i-1}}{p'(x_j)} \tag{6.17}$$

Para comprender esta ecuación es conveniente aclarar que en la definición de κ (ecuación (6.9)), f hace referencia a la salida que se quiere estudiar y x a la variable independiente de entrada. Esto no corresponde con la nomenclatura utilizada en la ecuación (6.17) puesto que la salida es la raíz x_j y la entrada es a_i.

También cabe destacar que un mismo polinomio tiene varios números de condición relativos siempre y cuando tenga varias raíces y varios coeficientes. Cuanto más altos sean varios números de condición, peor condicionado estará el problema.

Por último, esta expresión muestra claramente que cualquier problema de cálculo de raíces múltiples en polinomios es mal condicionado, ya que en dicho problema $p'(x_j) = 0$ y por ello $\kappa = \infty$.

Ejemplo 6.2 [8].

Este es un ejemplo simple de cálculo con la definición original de número de condición relativo (ecuación (6.9)) para la función $\sqrt{x}$ con $x > 0$. La Jacobiana de $f : x \mapsto \sqrt{x}$ es su derivada $J = f' = \frac{1}{2\sqrt{x}}$, entonces, tenemos:

$$\kappa = \frac{\|J\|}{\|f(x)\|/\|x\|} = \frac{\frac{1}{2\sqrt{x}}}{\frac{\sqrt{x}}{x}} = \frac{1}{2} \tag{6.18}$$

Dado que $\kappa \leq 10^2$, el problema está bien condicionado.

Ejemplo 6.3. Cálculo del número de condición relativo y mal condicionamiento del Polinomio Wilkinson [8].

Considérese el polinomio Wilkinson, que se define como $p(x) = \prod_{i=1}^{20}(x-i) = a_o + a_1 x + ... + x^{20}$. Este polinomio es más sensible entorno a su raíz $x = 15$ por las variaciones en su coeficiente $a_{15} \approx 1.67 * 10^9$. Su número de condicón relativo de acuerdo con la ecuación (6.17) será:

$$\kappa \approx \frac{1.67 \times 10^9 \cdot 15^{14}}{5!14!} \approx 5.1 \times 10^{13} \tag{6.19}$$

Con el siguiente código de MATLAB se estudia la estabilidad del Polinomio Wilkinson frente al Método de Newton al introducir una perturbación aleatoria del orden de 10^{-7} en uno de los coeficientes:

```
% Polinomio Wilkinson: Raíces exactas vs Raíces calculadas con el
% Método de Newton tras perturbar el polinomio

%Definición de los polinomios en forma simbólica
syms x
p_sym   = prod(x - (1:20));              % polinomio exacto
coeffs_p  = double(sym2poly(p_sym));     % vector de coeficientes
                                         % del polinomio exacto
```

```
9
10 eps_rel = 1e-7;                              % perturbación relativa
11
12 coeffs_pe = coeffs_p;
13 coeffs_pe(2) = coeffs_pe(2)*(1 + eps_rel); % altera coef de x^19
14
15 % Evaluar p_eps y su derivada
16 pe  = @(z) polyval(coeffs_pe , z);
17 dpe = @(z) polyval(polyder(coeffs_pe), z);
18
19 % Raices exactas del polinomio original
20 raices_p = 1:20;
21
22 maxIter = 50; % número máximo de iteraciones
23 tol = 1e-12; % tolerancia
24
25 raices_pe = NaN(1,20); %vector para almacenar raíces perturbadas
26
27 % Se aplica el Método de Newton al polinomio perturbado
28 for k = 1:20
29     x0 = k + 0.25; % iteración inicial
30     xk = x0;
31     for it = 1:maxIter
32
33         fx  = pe(xk);
34         dfx = dpe(xk);
35         if abs(dfx) < eps, break, end % evita que el método falle por
               ↪ derivadas cercanas a 0
36         xk = xk - fx/dfx; % Método de Newton
37         if abs(fx) < tol, break, end
38
39     end
40     raices_pe(k) = xk;
41
42 end
43
44 fprintf(' Raiz  |  valor teorico        |  p_eps(x)=0 (Newton)      |
       ↪ error absoluto \n');
45 fprintf('-------|------------------------|-----------------
46 ----------|-----------------\n');
47
48 for k = 1:20
49     err = abs(raices_pe(k) - raices_p(k));
50     fprintf(' %3d   | %23.16e | %23.16e | %13.6e \n', ...
51             k, raices_p(k), raices_pe(k), err);
52 end
```

Raiz	valor teorico	p_eps(x)=0 (Newton)	error absoluto
1	1.0000000000000000e+00	2.3820209657518852e+01	2.282021e+01
2	2.0000000000000000e+00	9.9999999999999778e-01	1.000000e+00
3	3.0000000000000000e+00	2.9999999998666573e+00	1.333427e-10
4	4.0000000000000000e+00	4.0000000459664236e+00	4.596642e-08
5	5.0000000000000000e+00	4.9999871482448741e+00	1.285176e-05
6	6.0000000000000000e+00	6.0012294953575296e+00	1.229495e-03
7	7.0000000000000000e+00	6.9542579932082482e+00	4.574201e-02
8	8.0000000000000000e+00	6.9542610671705472e+00	1.045739e+00
9	9.0000000000000000e+00	6.9542592096302043e+00	2.045741e+00
10	1.0000000000000000e+01	6.9542560946910088e+00	3.045744e+00
11	1.1000000000000000e+01	6.0012300301360124e+00	4.998770e+00
12	1.2000000000000000e+01	6.9542610573355930e+00	5.045739e+00
13	1.3000000000000000e+01	6.9542558070337348e+00	6.045744e+00
14	1.4000000000000000e+01	6.9542614243371181e+00	7.045739e+00
15	1.5000000000000000e+01	6.9542530817381927e+00	8.045747e+00
16	1.6000000000000000e+01	6.0012305830678185e+00	9.998769e+00
17	1.7000000000000000e+01	6.0012290857282711e+00	1.099877e+01
18	1.8000000000000000e+01	6.0012298876988712e+00	1.199877e+01
19	1.9000000000000000e+01	6.0012313128039159e+00	1.299877e+01
20	2.0000000000000000e+01	6.0012311883386023e+00	1.399877e+01

Figura 6.1: Diferencia entre Raíces del polinomio Wilkinson original y las calculadas por el Método de Newton del polinomio perturbado

El problema es claramente inestable ante la perturbación.

El siguiente código también estudia la estabilidad de este problema calculando las raíces perturbadas mediante el comando `roots()`, que utiliza métodos relacionados con álgebra lineal como la construcción de la matriz compañera y el cálculo de sus autovalores [34]. Este tipo de métodos no son objeto de estudio de este trabajo, si se desea conocer más información se recomienda consultar la fuente bibliográfica [34]. Aun así, el resultado obtenido es de alto interés para el presente trabajo:

```
1
2  % Polinomio Wilkinson: Raíces exactas vs Raíces calculadas con el
3  % comando roots() tras perturbar el polinomio
4
5  n = 20; % número de raíces
6
7  p = poly(1:n);  % coeficientes del polinomio de Wilkinson
8
9  num_trials = 100; % número de simulaciones aleatorias
10
11 roots_perturbed = []; % matriz de raíces perturbadas
12
13 eps = 1e-10; % perturbación
14
15 for i = 1:num_trials
16
17     r = randn(1, n+1); % vector de valores aleatorios para cada
18                        % simulación
19     p_pert = p .* (1 + eps * r); % perturbar el polinomio original
20
21     % Calcular las raíces del polinomio perturbado con roots()
22     roots_p = roots(p_pert);
23
```

```
    roots_perturbed = [roots_perturbed; roots_p.'];
end

% Representar
figure;
hold on;

original_roots = 1:n;
plot(real(original_roots), imag(original_roots), 'k.', 'MarkerSize', 20)
    ↪ ;

plot(real(roots_perturbed), imag(roots_perturbed), 'k.', 'MarkerSize',
    ↪ 5);

xlabel('Real');
ylabel('Imaginario');
title('Sensibilidad de Raices del Polinomio Wilkinson');
axis equal;
grid on
```

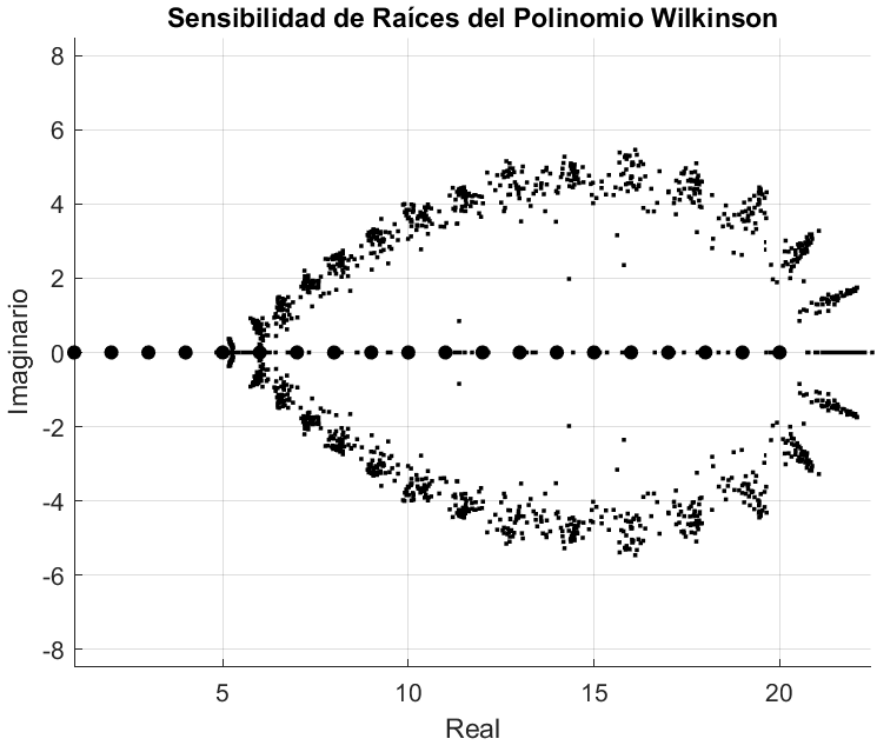

Figura 6.2: Raíces del Polinomio Wilkinson perturbadas y exactas en el plano complejo

Se observa claramente la inestabilidad del problema dada la dispersión de las raíces calculadas (puntos de menor tamaño) frente a las raíces exactas (puntos de mayor tamaño).

6.4. Perturbaciones para preservar la multiplicidad: Iteración de Gauss-Newton

Realmente, el cálculo numérico de raíces múltiples de polinomios, no tiene que ser necesariamente un problema mal condicionado. En 1972, William Kahan descubrió que los problemas de raíces múltiples no son sensibles a cualquier perturbación, sino a perturbaciones arbitrarias pero que si, estas perturbaciones preservan la multiplicidad, el problema permanece inalterado [27]. A continuación se muestra un caso en el que se perturba un polinomio preservando su multiplicidad:

Ejemplo 6.4 [27].

Para $p(x) = x^2 + bx + c$ existe una raíz doble $x = -\frac{b}{2}$ si $b^2 - 4c = 0$. Si se incluye una perturbación arbitraria, el polinomio perturbado es, por ejemplo:

$$\tilde{p}(x) = x^2 + (b+\delta)x + (c+\varepsilon) \tag{6.20}$$

Las raíces perturbadas son:

$$\tilde{x} = \frac{-(b+\delta) \pm \sqrt{(b+\delta)^2 - 4(c+\varepsilon)}}{2} \tag{6.21}$$

Nótese que la raíz cuadrada amplifica el error. Entonces, la perturbación que preserva la multiplicidad será cualquiera que cumpla la siguiente igualdad:

$$(b+\delta)^2 - 4(c+\varepsilon) = 0 \tag{6.22}$$

El ejemplo anterior es un caso bastante simple. Normalmente los polinomios que se encuentran en la práctica serán más complejos.

Con el fin de realizar perturbaciones que preservaran la multiplicidad, Kahan definió las variedades peyorativas [27]. Una variedad peyorativa es un conjunto que forman los polinomios con una misma estructura de multiplicidad, esto es, polinomios con mismo número de raíces y misma multiplicidad en cada una de ellas.

Para un polinomio de la forma $p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_0$ se define la estructura de multiplicidad y variedad peyorativa como:

Definición 6.3. Estructura de multiplicidad [22]. Es un vector ordenado de enteros positivos $l = [l_1, ..., l_m]$ tal que $l_1 + ... + l_m = n$, siendo l_m la multiplicidad de la raíz m-ésima.

Definición 6.4. Variedad peyorativa [22]. Para un determinado l se define su variedad peyorativa como el conjunto de vectores $\Pi_l = \{G_l(z) \mid \text{z es de tamaño m}\}$, donde $z = [\zeta_1, ..., \zeta_m]^T$ es el vector de las raíces (distintas) del polinomio y $G_l(z) = [g_{n-1}(z), g_{n-2}(z), ..., g_0(z)]^T$ es el operador de coeficientes asociado con l.

Esto se puede resolver como un sistema de la forma $G_l(z) = a = [a_{n-1}, a_{n-2}, ..., a_0]^T$;. Esto es un sistema sobredeterminado ya que hay m raíces diferentes que hallar y un polinomio de grado n, por ello se tienen m incógnitas y n ecuaciones, siendo $n > m$ (salvo que $l = [1, 1, ..., 1]$). Por lo tanto, se resolverá este sistema como un problema de mínimos cuadrados.

Ejemplo 6.5 [22].

Este es un ejemplo simple de cálculo de una variedad peyorativa. Sea un polinomio de estructura de multiplicidad $l = [1, 2]$, este se puede escribir como $p(x) = (x - \zeta_1)(x - \zeta_2)^2 = x^3 + (-\zeta_1 - 2\zeta_2)x^2 + (2\zeta_1\zeta_2 + \zeta_2^2)x + (-\zeta_1\zeta_2^2)$. Entonces:

$$G_{[1,2]}(z) = \begin{bmatrix} -\zeta_1 - 2\zeta_2 \\ 2\zeta_1\zeta_2 + \zeta_2^2 \\ -\zeta_1\zeta_2^2 \end{bmatrix}, \quad z = \begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix} \tag{6.23}$$

Los vectores $G_{[1,2]}(z)$ para todo z forman la variedad peyorativa $\Pi_{[1,2]}$.

Los pesos usados en este desarrollo son definidos por Zhonggang Zeng de la siguiente forma $\omega_j = \text{mín}\left\{1, \frac{1}{|a_j|}\right\}$, para $j = 0, \ldots, n-1$, ya que estos minimizan el error introducido en el polinomio para todos los coeficientes mayores que 1 [22].

Estos pesos se incluyen en el problema mediante la matriz $W = diag(\omega_{n-1}, \omega_{n-2}, ..., \omega_0)$. En muchas ocasiones, Zeng hace uso de los pesos unidad, es decir, $w_j = 1$ para $j = 0, \ldots, n-1$, que minimizan el error general en los coeficientes [17]. Para cualquier vector v de tamaño n se define su norma-2 ponderada como [22]:

$$||v||_W = ||Wv||_2 = \sqrt{\sum_{j=0}^{n-1} \omega_j^2 v_j^2} \tag{6.24}$$

Teniendo todo esto en cuenta, se plantea el problema de mínimos cuadrados como [22]:

$$\min_{z \in \mathbb{C}^m} \|G_l(z) - a\|_W^2 = \min_{z \in \mathbb{C}^m} \|W(G_l(z) - a)\|_2^2 = \min_{z \in \mathbb{C}^m} \left\{ \sum_{j=0}^{n-1} \omega_j^2 \left| G_j(z) - a_j \right|^2 \right\} \tag{6.25}$$

Sea $J(z)$ la Jacobiana de $G_l(z)$ y $\hat{J}(z) = WJ(z)$. Se quiere encontrar entonces un mínimo local de $F(z) = W(G_l(z) - a)$, para ello buscamos un $\tilde{z} \in \mathbb{C}^m$, denominado solución peyorativa, tal que [22]:

$$\hat{J}(\tilde{z})^H F(\tilde{z}) = \left[WJ(\tilde{z})\right]^H W \left[G_l(\tilde{z}) - a\right] = J(\tilde{z})^H W^2 \left[G_l(\tilde{z}) - a\right] = 0 \tag{6.26}$$

Nótese que A^H representa la traspuesta conjugada A y que esta condición proviene de la necesidad de gradiente nulo para que $\tilde{z}$ sea un mínimo local.

Teorema 6.1 [22]. Sea $G_l : \mathbb{C}^m \to \mathbb{C}^n$ el operador de coeficientes asociado con una estructura de multiplicidad $l = [l_1, ..., l_m]$. Entonces la Jacobiana $J(z)$ de $G_l(z)$ es de rango completo si y solo si todas las raíces $(\zeta_1, ..., \zeta_m)$ son distintas.

Demostración. Para la demostración de este teorema se recomienda consultar la referencia bibliográfica [22]. Además, es claro ver que este teorema sí que se verifica en el caso de estudio de este trabajo, puesto que en la Definición 6.4 se especifica que en el vector de raíces z todas las componentes son distintas.

Teniendo en cuenta este teorema, se puede afirmar que el sistema $G_l(z) = a$ no es singular, dado que la matriz $J(z)$ tiene rango completo y consecuentemente es invertible. Por ello, se puede definir la iteración de Gauss-Newton sobre la variedad peyorativa Π_l de la siguiente forma.

Definición 6.5. Iteración de Gauss Newton [22].

$$z_{k+1} = z_k - \left[J(z_k)_W^+\right] \left[G_l(z_k) - a\right], \quad (k = 0, 1, \ldots) \tag{6.27}$$

Donde $J(z_k)_W^+ = \left[J(z_k)^H W^2 J(z_k)\right]^{-1} J(z_k)^H W^2$

Teorema 6.2 [22]. Sea $\tilde{z} = (\tilde{\zeta_1}, \ldots, \tilde{\zeta_m})$ una solución peyorativa de $p \sim a$, asociada a una estructura de multiplicidad l y una matriz de pesos W. Supóngase que $\tilde{\zeta_1}, \tilde{\zeta_2}, \ldots, \tilde{\zeta_m}$ son distintas. Entonces existe un número $\epsilon > 0$ tal que, si $\|a - G_l(\tilde{z})\|_W < \epsilon$ y $\|z_0 - \tilde{z}\|_2 < \epsilon$, entonces la Iteración de Gauss-Newton está bien definida y converge a la raíz peyorativa $\tilde{z}$ con un orden de convergencia al menos lineal. Si además $a = G_l(\tilde{z})$, entonces la convergencia es cuadrática.

Demostración. Para la demostración de este teorema se recomienda consultar la referencia bibliográfica [22].

6.5. Número de condición peyorativo.

En general, un número de condición es el número más pequeño κ que satisface la siguiente expresión: [Error solución] $\leq [\kappa] \times$ [Error datos] + h.o.t. [27]. Donde el error en los datos es la perturbación introducida en los coeficientes del polinomio. El término h.o.t. (higher-order terms) representa los términos de mayor orden en el error de la solución, es decir, tiene la forma $O(\delta x)$. Al ser la perturbación prácticamente nula y estar elevada a potencias altas, de ahora en adelante se desprecia este término.

Teniendo esto en cuenta, la expresión anterior queda:

$$[\text{Error solución}] \leq [\kappa] \times [\text{Error datos}] \tag{6.28}$$

Definición 6.5. Número de condición peyorativo [27].

$$\|\hat{z} - z\|_2 \leq \frac{1}{\sigma_{min}} \|\hat{a} - a\|_2 \tag{6.29}$$

Donde $\hat{z}$ es una solución perturbada que preserva la multiplicidad de p y $\hat{a} = G(\hat{z})$. De la ecuación (6.28) es claro deducir que $\kappa = \frac{1}{\sigma_{min}}$. Este valor recibe el nombre de número de condición peyorativo y guarda relación con la posibilidad de resolver un problema de raíces de polinomios mediante la iteración de Gauss-Newton. Para entender de donde procede esta expresión, se introduce el siguiente teorema:

Teorema 6.3 [1]. Sea $\sigma_1(A)$ el mayor valor singular (o autovalor elevado al cuadrado) de A. Entonces:

$$\|A\|_2 = \sqrt{\rho(A^H A)} = \sqrt{\rho(AA^H)} = \sigma_1(A) \tag{6.30}$$

Donde A es una matriz cuadrada y $\rho(AA^H)$ es el radio espectral de la matriz AA^H. En particular, si A es hermitiana (o real y simétrica), entonces:

$$\|A\|_2 = \rho(A) \tag{6.31}$$

Mientras que, si A es unitaria:

$$\|A\|_2 = 1 \tag{6.32}$$

Demostración. Para esta demostración se hace uso de las expresiones: $U^H A^H AU = \text{diag}(\mu_1, \ldots, \mu_n)$; $\|A\|_2 \overset{\text{Definición}}{=} \sup_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2}$; $\langle Ax, y\rangle = \langle x, A^H y\rangle$; que han sido tomadas de la referencia bibliográfica [1].

Como $A^H A$ es una matriz hermítica, existe una matriz unitaria U tal que:

$$U^H A^H AU = \text{diag}(\mu_1, \ldots, \mu_n), \tag{6.33}$$

donde μ_i son los valores propios positivos de $A^H A$. Sea $y = U^H x$, como U es unitaria, $x = Uy$ entonces:

$$\|A\|_2 \overset{\text{Definición}}{=} \sup_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sup_{x\neq 0} \sqrt{\frac{\langle Ax, Ax\rangle}{\langle x, x\rangle}} \overset{\langle Ax,y\rangle = \langle x, A^H y\rangle}{=} \sup_{x\neq 0} \sqrt{\frac{\langle A^H Ax, x\rangle}{\langle x, x\rangle}}, \tag{6.34}$$

$$\overset{x=Uy}{=} \sup_{y\neq 0} \sqrt{\frac{\langle A^H AUy, Uy\rangle}{\langle Uy, Uy\rangle}} \overset{\langle Ax,y\rangle = \langle x, A^H y\rangle}{=} \sup_{y\neq 0} \sqrt{\frac{\langle U^H A^H AUy, y\rangle}{\langle U^H Uy, y\rangle}}, \tag{6.35}$$

$$= \sup_{y\neq 0} \sqrt{\frac{\langle U^H A^H AUy, y\rangle}{\langle y, y\rangle}} \overset{(6.33)}{=} \sup_{y\neq 0} \sqrt{\frac{\sum_{i=1}^n \mu_i |y_i|^2}{\sum_{i=1}^n |y_i|^2}} = \sqrt{\max_{1\leq i\leq n} |\mu_i|}, \tag{6.36}$$

Lo cual prueba la igualdad del teorema, gracias a la definición de valor singular.

Para continuar con la deducción del número de condición peyorativo se continua desarrollando la expresión:

$$\sqrt{\max_{1\leq i\leq n}|\mu_i|} \geq \sqrt{\min_{1\leq i\leq n}|\mu_i|} \overset{\mu_i\geq 0}{=} \sigma_{min} \tag{6.37}$$

Se sabe que [22]:

$$\hat{a} - a = G_l(\hat{z}) - G_l(z) = J(z)(\hat{z} - z) + O(\|\hat{z} - z\|^2) \tag{6.38}$$

Nótese que el término $O(\|\hat{z} - z\|^2)$ es equivalente al término h.o.t. y, consecuentemente, se depreciará.

De acuerdo con el Teorema 6.1, si las todas las raíces del polinomio $(\zeta_1, ..., \zeta_m)$ son distintas, $J(z)$ es de rango completo (y al ser W de rango completo $WJ(z)$ también lo será). Teniendo esto en cuenta, la expresión (6.38) se puede escribir como [22]:

$$\|W(\hat{a} - a)\|_2 = \|[WJ(z)](\hat{z} - z)\|_2 \tag{6.39}$$

Utilizando las expresiones (6.36), (6.37), (6.39) se obtiene que [22]:

$$\|\hat{a} - a\|_W = \|[WJ(z)](\hat{z} - z)\|_2 \geq \sigma_{\text{mín}} \|\hat{z} - z\|_2 \tag{6.40}$$

Por ser todos los pesos $w_j \leq 1$:

$$\|\hat{a} - a\|_2 \geq \sigma_{\text{mín}} \|\hat{z} - z\|_2 \tag{6.41}$$

De donde se deduce directamente la expresión (6.29). De ahora en adelante, para referirse al número de condición peyorativo asociado a una estructura de multiplicidad l y matriz de peso W, se usa la nomenclatura $\kappa_{l,W}$.

Ejemplo 6.6.

Este es un ejemplo de cálculo de este número de condición para diferentes multiplicidades en un polinomio de estructura similar. Para este caso se ha escogido un polinomio de la forma $p(x) = (x + 1)^{l_1}(x - 1)^{l_2}(x - 2)^{l_3}$ (polinomios tomados de la fuente bibliográfica [27]). A continuación se muestra un código de MATLAB el cual hace variar el vector l entre los siguientes valores: $l = [1, 1, 1]$; $l = [1, 2, 3]$; $l = [10, 20, 30]$; $l = [100, 200, 300]$; y calcula el número de condición peyorativo:

```
% Cálculo del número de condición peyorativo

% Declaración de las raíces exactas como vector
r = [-1, 1, 2];

% Matriz de multiplicidades para los diferentes casos
L = [  1   1   1;
       1   2   3;
      10  20  30;
     100 200 300 ];

fprintf('l1\tl2\tl3\tNumero de Condicion Peyorativo\n');
fprintf('-------------------------------------\n');

% Iteración sobre cada fila de la matriz L
for fila = 1:size(L,1)

    l = L(fila, :); % multiplicidades del caso sobre el que se está
    % iterando
    n_total = sum(l); % grado del polinomio

    % Bucle para construcción del polinomio
    p = 1;
```

```
    for j = 1:3
        for i = 1:l(j)
            p = conv(p, [1, -r(j)]);
        end
    end

    a = p(2:end); % coeficientes (excepto el primero, que es =1)
    n = length(a);

    % Declaración de la Jacobiana
    J = zeros(n, 3);
    delta = 1e-6; % perturbación

    for j = 1:3
        r_pert = r;
        r_pert(j) = r_pert(j) + delta; % perturbación en la raiz j

        % Bucle para construcción del polinomio perturbado
        p_pert = 1;
        for k = 1:3
            for i = 1:l(k)
                p_pert = conv(p_pert, [1, -r_pert(k)]);
            end
        end

        a_pert = p_pert(2:end); % exclusión del coeficiente principal

        % Derivada aproximada por diferencias finitas
        J(:, j) = (a_pert - a)' / delta;
    end

    w = min(1, 1 ./ abs(a)); % cálculo de los pesos
    W = diag(w); % matriz de pesos W

    JW =  W*J;

    sigma_min = min(svd(JW)); % cálculo del valor singular minimo

    k = 1 / sigma_min; % cálculo del número de condicion peyorativo

    fprintf('%d\t%d\t%d\t%.4f\n', l(1), l(2), l(3), k);
end
```

```
>> SensibilidadMultiplicidad
l1   l2   l3   Numero de Condicion Peyorativo
-------------------------------------
1    1    1    3.1500
1    2    3    2.0324
10   20   30   0.0733
100  200  300  0.0005
```

Figura 6.3: Número de Condición Peyorativo de los diferentes casos

Se observa que, a pesar de ser polinomios con raíces múltiples e incluso multiplicidades altas en algunos de los casos, los números de condición peyorativos son bajos.

7. ALGORITMOS DE CÁLCULO PARA PROBLEMAS MAL CONDICIONADOS

7.1. Algoritmo I de Zhonggang Zeng

En este apartado se desarrolla un método iterativo basado en la ecuación (6.27), comúnmente conocido como el Algoritmo I de Zhonggang Zeng. Para comprender este método se describe su pseudocódigo a continuación:

Algoritmo EVALG [22].

Input: Enteros m, n; vector $z = (z_1, \ldots, z_m)^T$; multiplicidades $l = [l_1, \ldots, l_m]$
Output: Vector $G_l(z) \in \mathbb{C}^m$
$s = (1)$
for $i = 1$ **to** m **do**
  for $k = 1$ **to** l_i **do**
    $s = \text{conv}(s, (1, -z_i))$
  end
end
for $j = 1$ **to** n **do**
  $g_{n-j}(z) =$ componente $(j+1)$ de s
end

Algoritmo EVALJ [22].

Input: Enteros m, n; vector $z = (z_1, \ldots, z_m)^T$; multiplicidades $l = [l_1, \ldots, l_m]$
Output: Jacobiana $J(z) \in \mathbb{C}^{n \times m}$
$u = \prod (x - z_j)^{l_j - 1}$
for $j = 1$ **to** m **do**
  $s = -l_j u$;
  for $r = 1$ **to** m **do**
    if $r \neq j$ **then**
      $s = \text{conv}(s, (1, -z_r))$
    end
  end
  columna j de $J(z) = s$
end

Algoritmo PEJROOT (Algoritmo I) [22].

Input: Enteros m, n; vector $a \in \mathbb{C}^n$; matriz de pesos W; iteración inicial z_0; multiplicidades l; tolerancia de error τ
Output: Raíces $z = (\zeta_1, \ldots, \zeta_m)$ o mensaje de error
for $k = 0, 1, \ldots$ **do**
  Calcular $G_l(z_k)$ y $J(z_k)$ usando EVALG y EVALJ ;
  Resolver en mínimos cuadrados: $[WJ(z_k)](\Delta z_k) = W[G_l(z_k) - a]$;
  $z_{k+1} = z_k - \Delta z_k, \quad \delta_k = \|\Delta z_k\|_2$;
  if $k \geq 1$ **then**
    if $\delta_k \geq \delta_{k-1}$ **then**
      detener, mostrar mensaje de error
    end
    else if $\frac{\delta_k^2}{\delta_{k-1} - \delta_k} < \tau$ **then**
      detener, salida: $z = z_{k+1}$
    end
  end
end

Nótese que para el cálculo del paso de la iteración Zeng no utiliza idénticamente la iteración descrita en la ecuación (6.27), sino que emplea una versión simplificada por razones de eficiencia computacional y estabilidad numérica.

En base a este pseudocódigo se ha elaborado un código de MATLAB el cual aplica este método en una forma similar a la descrita en el pseudocódigo. Para ello se han programado tres funciones y un archivo principal, el cual debe ser editado en función del polinomio con el que se quiera trabajar. A continuación se muestra el código de la función evalg y evalj, que corresponde a las partes del pseudocódigo con los mismos nombres.

```
function s = evalg(z, l)
    m = length(z); % m corresponde al número de raíces distintas
    s = 1; % inicialización del polinomio a 1
    for i = 1:m
        for k = 1:l(i)
            s = conv(s, [1, -z(i)]); % construcción del polinomio
                                     % mediante convoluciones
        end
    end
    s = s(:); % coeficientes como vector columna
end
```

```
function J = evalj(z, l, n)
    m = length(z); % m corresponde al número de raíces distintas
    l_menos1 = max(l - 1, 0); % vector auxiliar para el cáculo de la
                              % Jacobiana que evita multiplicidades
                              % negativas

    u = evalg(z, l_menos1); % polinomio con disminución unitaria en
                            % todas sus multiplicidades (exponentes)
                            % para el cáculo de la Jacobiana

    J = zeros(n, m); % declaración de la Jacobiana

    for j = 1:m
        s = -l(j) * u;
        for q = 1:m
            if q ~= j
```

```
                s = conv(s, [1, -z(q)]); % construcción de la derivada
                                         % mediante convoluciones
            end
        end
        s = s(:);
        if length(s) < n
            s = [zeros(n - length(s), 1); s]; % ceros en primeras
                                              % posiciones del vector
                                              % s en caso de que sea
                                              % necesario para que
                                              % tenga n componentes
        end
        J(:, j) = s(end - n + 1:end); % guardar vector s en Jacobiana
    end
end
```

```
function [z, exito] = pejroot_auto(a, W, z0, l, tol)
    max_iter = 100; % número de iteraciones
    z = z0(:); % inicilización vector solución con iteración inicial
    m = length(z); % m corresponde al número de raíces distintas
    n = length(a); % n corresponde al grado del polinomio
    exito = false; % inicialización de la variable exito como falsa
    delta_prev = inf; % inicialización de la variable delta_prev como
                      % infinito para evitar errores

    for k = 0:max_iter
        fprintf('Iteración %d:\n', k);
        disp(z.');

        % Evaluar G(z)
        s = evalg(z, l); %devuelve el polinomio
        s = s(:);
        if length(s) < n
            s = [zeros(n - length(s), 1); s]; % asegurar dimensión n
        end
        G = s(end - n + 1:end); % cálculo de G en esta función en lugar
                                % de en evalg para ahorrar potencia
                                % computacional


        % Evaluar Jacobiana
        Jmat = evalj(z, l, n);

        % Resolver sistema de mínimos cuadrados con regularización
        A = W * Jmat;
        b = W * (G - a(:));
        lambda = 1e-8; % regularización pequeña
        % Iteración de Gauss-Newton original con regularización
        dz = (A' * A + lambda * eye(size(A, 2))) \ (A' * b);

        z = z - dz;
        delta_k = norm(dz);

        if k >= 1
            if delta_k > 1.1 * delta_prev  % criterio de parada laxo
                disp('Fallo: no hay convergencia (incremento en delta)')
                    ↪ ;
                return;
```

```
            % Criterio de parada:
            elseif abs((delta_k^2 / (delta_prev - delta_k))) < tol
                exito = true;
                return;
            end
        end
        delta_prev = delta_k; % actualización de delta_prev
    end
end
```

Ejemplo 7.1.

Para ejecutar el Algoritmo I, se hace uso del archivo de MATLAB AlgoritmoI.m, en el cual se deben introducir las raíces del polinomio y su correspondiente estructura de multiplicidad (vectores *z_verdadero* y *l_verdadero*, respectivamente). Este archivo utiliza las funciones descritas anteriormente y contiene el código que se muestra a continuación. Las variables se han ajustado para calcular las raíces de uno de los casos mostrados en el Ejemplo 6.6:

```
% Raices y estructura de multiplicidad
z_verdadero = [-1;1;2];
l_verdadero = [10;20;30];

% Pequeña perturbación para inicializar cerca de la raíz
rng(0); % semilla para reproducibilidad
z0 = z_verdadero + 1e-1*randn(size(z_verdadero));
l = l_verdadero;

% Coeficientes objetivo
a = evalg(z_verdadero, l_verdadero);

% Matriz de pesos unidad
W = eye(length(a));

tol = 1e-10; % tolerancia

% Llamada al método
[z, exito] = pejroot_auto(a, W, z0, l, tol);

if exito
    disp('Raices encontradas:');
    disp(z.');
else
    disp('El metodo no convergio');
end
```

```
>> AlgoritmoI
Iteración 0:
  -0.946233286045390   1.183388501459509   1.774115313899635

Iteración 1:
  -1.040185700756367   0.508675743331050   2.386573194097219

Iteración 2:
  -0.636806601508860   0.265165419913177   2.509484614462944

Fallo: no hay convergencia (incremento en delta)
El metodo no convergio
```

Figura 7.1: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x+1)^{10}(x-1)^{20}(x-2)^{30}$

El método no converge. Si `z0=z_verdadero + 1e-2*randn(size(z_verdadero))`:

```
>> AlgoritmoI
Iteración 0:
  -0.994623328604539   1.018338850145951   1.977411531389964

Iteración 1:
  -1.000305903156209   0.998007600007320   2.001638932775836

Iteración 2:
  -0.999999102405407   0.999999051756526   1.999997424677804

Iteración 3:
  -1.000000000033856   1.000000000155659   1.999999999913948

Raices encontradas:
  -1.000000000000182   1.000000000000998   1.999999999999027
```

Figura 7.2: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x+1)^{10}(x-1)^{20}(x-2)^{30}$ por segunda vez

El método converge correctamente. Si se asigna `l_verdadero=[100;200;300]`:

```
Iteración 998:
   NaN   NaN   NaN

Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate. RCOND = NaN.
> In pejroot_auto (line 28)
In AlgoritmoI (line 20)

Iteración 999:
   NaN   NaN   NaN

Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate. RCOND = NaN.
> In pejroot_auto (line 28)
In AlgoritmoI (line 20)

Iteración 1000:
   NaN   NaN   NaN

Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate. RCOND = NaN.
> In pejroot_auto (line 28)
In AlgoritmoI (line 20)

El metodo no convergio
```

Figura 7.3: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x+1)^{100}(x-1)^{200}(x-2)^{300}$

Nuevamente, el método no converge.

A pesar de que este algoritmo es robusto frente muchos polinomios mal condicionados, únicamente se puede asegurar que dará una solución correcta bajo ciertas condiciones.

Teorema 7.1 [22]. Para una determinada estructura de multiplicidad $l = [l_1, l_2, ..., l_m]$, sea el polinomio $\hat{p} \sim \hat{b}$ una aproximación de $p \sim b$ con raíces perturbadas y raíces $\hat{z}$ y z, respectivamente , correspondientes a l y la matriz de pesos W. Supóngase que las componentes de z son distintas mientras $\left\|G_l(z) - \hat{b}\right\|_W$ alcance un mínimo local en $\hat{z}$. Si $||b - \hat{b}||_W$ y $||G_l(z) - b||_W$ son suficientemente pequeños entonces:

$$\|z - \hat{z}\|_2 \leq 2\kappa_{l,W}(z)\left(\|G_l(z) - b\|_W + \left\|b - \hat{b}\right\|_W\right) + \overbrace{h.o.t.}^{\approx 0} \tag{7.1}$$

Para la demostración de este teorema se requiere hacer uso de desigualdad triangular:

Demostración [22].

De la expresión (6.29) se sabe que:

$$\|z - \hat{z}\|_2 \leq \kappa_{l,W}(z)\,\|G_l(z) - G_l(\hat{z})\|_W \tag{7.2}$$

$$\leq \kappa_{l,W}(z)\left(\|G_l(z)-b\|_W+\|b-\hat{b}\|_W+\|G_l(\hat{z})-\hat{b}\|_W\right) \tag{7.3}$$

Como $\|G_l(\hat{z})-\hat{b}\|_W$ es un mínimo local:

$$\|G_l(\hat{z})-\hat{b}\|_W \leq \|G_l(z)-\hat{b}\|_W \ \leq \|G_l(z)-b\|_W+\|b-\hat{b}\|_W \tag{7.4}$$

De aquí es trivial demostrar con las expresiones (7.3) y (7.4) el resultado del Teorema 7.1

Este teorema muestra que las raíces peyorativas pueden ser estables a pesar de que las raíces exactas sean sensibles siempre y cuando $\kappa_{l,W}$ no sea demasiado grande. Para un polinomio p con estructura de multiplicidad l, se puede estimar el error (en base a este resultado) de sus raíces múltiples calculadas a partir de su aproximación $\hat{p}$ [22].

Las raíces exactas del polinomio perturbado $\hat{p}$ son, normalmente, simples y lejanas de las raíces múltiples de p. Sin embargo, por el siguiente corolario, las raíces múltiples $\hat{z}$ de un polinomio $\hat{p}$ con estructura de multiplicidad l pueden ser una aproximación precisa de las raíces múltiples z de p [22].

Corolario 7.1 [22].

Bajo las condiciones del Teorema 7.1, si z es el vector de raíces exactas de p con estructura de multiplicidad l, entonces:

$$\|z-\hat{z}\|_2 \leq 2\kappa_{l,W}(z)\left\|b-\hat{b}\right\|_W + \underbrace{h.o.t.}_{\approx 0} \tag{7.5}$$

Demostración [22]. Como z es exacto, $\|G_l(z)-b\|_W=0$, si introducimos esta expresión en la ecuación (7.1) la demostración es directa.

A continuación se presentan algunos ejemplos para diferentes polinomios, que son problemas mal condicionados por tener un número de condición relativo elevado, pero se pueden resolver con el Algoritmo I obteniendo una buena precisión por tener un número de condición peyorativo bajo. Para el cálculo del número de condición y las raíces en la misma ejecución de MATLAB, se combina el código del Algoritmo I con el del Ejemplo 7.1 y se añade el cálculo del número de condición relativo:

```
% Cálculo de raíces y número de condición

r =[]; % raíces exactas
l =[]; % multiplicidades

% Construcción del polinomio
p = 1;
for j = 1:length(r)
    for i = 1:l(j)
        p = conv(p, [1, -r(j)]);
    end
end
a_full = p(2:end).'; % coeficientes excepto el principal (que es 1)
n = length(a_full);  % grado del polinomio

fprintf('Número de condición peyorativo para r y l dados:\n');

% Cálculo del número de condición peyorativo

J = zeros(n, length(r)); % inicialización de la Jacobiana
delta = 1e-6; % perturbación

for j = 1:length(r)
    r_pert = r;
    r_pert(j) = r_pert(j) + delta; % perturba raíces exactas

    % Polinomio perturbado
```

```
    p_pert = 1;
    for k = 1:length(r)
        for i = 1:l(k)
            p_pert = conv(p_pert, [1, -r_pert(k)]); % construcción del
                                                    % polinomio
                                                    % perturbado
        end
    end
    a_pert = p_pert(2:end).'; % coeficientes excepto el principal (que
                              % es 1)

    % Ajustar tamaño para evitar errores
    if length(a_pert) > n
        a_pert = a_pert(end-n+1:end);
    elseif length(a_pert) < n
        a_pert = [zeros(n - length(a_pert), 1); a_pert];
    end
    % Derivada aproximada por diferencias finitas
    J(:, j) = (a_pert - a_full) / delta;
end

w = min(1, 1 ./ abs(a_full)); % cálculo de pesos
W = diag(w); % matriz de pesos
JW = W * J;
sigma_min = min(svd(JW)); % cálculo del valor singular mínimo
kappa_peyorativo = 1 / sigma_min; % número de condición peyorativo

fprintf('kappa peyorativo = %.4e\n', kappa_peyorativo);

% Cálculo del número de condición relativo

% Derivada de p(x)
dp = polyder(p);
a = flipud(a_full); % invertir para alinear con potencias x^0, x^1,
                    % ..., x^{n-1}

kappa_max = 0; % inicializar número de condición relativo máximo

for j = 1:length(r)
    xj = r(j);
    dp_xj = polyval(dp, xj); % evaluar derivada en xj

    if abs(dp_xj) < 1e-14
        kappa_j = Inf; % Número de condición relativo infinito para
                       % derivadas nulas
    else
        kappa_vals = abs(a .* xj.^(0:n-1).'); % cálculo de numeradores
        kappa_j = max(kappa_vals) / abs(dp_xj); % cálculo número de
                                                % condición relativo
    end

    kappa_max = max(kappa_max, kappa_j); % número de condición más
                                         % alto
end

fprintf('kappa clásico = %.4e\n\n', kappa_max);
z0 = []; % iteración inicial

```

```
% Matriz peso identidad para eficiencia computacional
W_method = eye(n);
tol = 1e-10; % tolerancia

% Ejecutar Algoritmo I
[z, exito] = pejroot_auto(a_full, W_method, z0, 1, tol);

if exito
    fprintf('Raíces encontradas:\n');
    disp(z.');
else
    disp('El método no convergió.');
end
```

Ejemplo 7.2.

Para realizar este ejemplo, se ejecuta el código anterior para el polinomio $p(x) = (x-1)^{100}$. Por ello los vectores se inicializan como `r=[1];` y `l=[100];`. La iteración inicial viene dada por `z0 = r + 1e-1*randn(size(r));`

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 1.7191e-03
kappa clásico = 4.2354e+14

Iteración 0:
   1.044937762316685

Iteración 1:
   1.026743694601220

Iteración 2:
   1.011860684832719

Iteración 3:
   1.002882334097622

Iteración 4:
   1.000196044173907

Iteración 5:
   1.000000948109155

Iteración 6:
   1.000000000022248

Raíces encontradas:
     1
```

Figura 7.4: Salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con z_0 con una perturbación del orden de 10^{-1}

Si se hace `z0 = r + 1e-2*randn(size(r))`:

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 1.7191e-03
kappa clásico = 4.2354e+14

Iteración 0:
   1.005361570799259

Iteración 1:
   1.000651685587503

Iteración 2:
   1.000010397393657

Iteración 3:
   1.000000002675153

Raíces encontradas:
   1.000000000000000
```

Figura 7.5: Salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con z_0 con una perturbación del orden de 10^{-2}

Con `z0 = r + 1e-3*randn(size(r))`:

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 1.7191e-03
kappa clásico = 4.2354e+14

Iteración 0:
   1.000897888425985

Iteración 1:
   1.000019656853034

Iteración 2:
   1.000000009560051

Raíces encontradas:
   1.000000000000002
```

Figura 7.6: Salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con z_0 con una perturbación del orden de 10^{-3}

Se observa que cuanto más cerca se encuentre la iteración inicial de las raíces, más rápido converge el método y peor precisión se obtiene. Ahora supóngase que no se conoce la raíz y se empieza a ejecutar el algoritmo desde un punto lejano, como puede ser `z0=10`:

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 1.7191e-03
kappa clásico = 4.2354e+14

Iteración 0:
    10

Iteración 1:
   9.890305748116379

Iteración 2:
   9.781705649871991

Iteración 3:
   9.674188793037640

Iteración 4:
   9.567744374203112

Iteración 5:
   9.462361697692575
```

Figura 7.7: Comienzo de la salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con $z_0 = 10$

Por razones de practicidad no se incluyen los resultados de todas las iteraciones pero si de algunas iteraciones intermedias y finales con el fin de mostrar que el algoritmo converge de forma uniforme a la solución sin estancarse.

```
Iteración 58:
   5.151267371227561

Iteración 59:
   5.089939125750896

Iteración 60:
   5.029222667506487

Iteración 61:
   4.969111895195832

Iteración 62:
   4.909600768385484

Iteración 63:
   4.850683306900458

Iteración 64:
   4.792353590223690

Iteración 65:
   4.734605756901492
```

Figura 7.8: Iteraciones intermedias de la salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con $z_0 = 10$

```
Iteración 168:
   1.045710423345008

Iteración 169:
   1.027428282802171

Iteración 170:
   1.012362683429831

Iteración 171:
   1.003107833259710

Iteración 172:
   1.000227081531827

Iteración 173:
   1.000001271419139

Iteración 174:
   1.000000000040008

Raíces encontradas:
     1
```

Figura 7.9: Iteraciones finales de la salida de MATLAB para el polinomio $p(x) = (x-1)^{100}$ con $z_0 = 10$

El método converge correctamente.

Ejemplo 7.3.

En este caso se estudia el polinomio $p(x) = (x - 0.9)^{18}(x - 1.0)^{10}(x - 1.1)^{16}$. Para ello en el código propuesto los vectores toman los valores `r=[0.9,1.0,1.1];` y `l=[18,10,16];`. Las iteraciones iniciales en las diferentes ejecuciones son idénticas a las del Ejemplo 7.2.

```
Iteración 27:
 -21.576761140244415  12.318680254529353   4.473991248074908

Iteración 28:
 -21.004146959309526  11.486176155299372   4.517074816378928

Iteración 29:
 -20.443179767976261  10.684022691546362   4.567815430893413

Iteración 30:
 -19.893864052881348   9.905864955740340   4.628537008303799

Iteración 31:
 -19.356145953407495   9.143654533164716   4.702806551356226

Iteración 32:
 -18.829901675261929   8.385872679388282   4.796554878275666

Iteración 33:
 -18.314911979961735   7.613091437879823   4.920884360335629

Iteración 34:
 -17.810801062128704   6.783727394789492   5.101062552957659

Fallo: no hay convergencia (incremento en delta)
El método no convergió.
```

Figura 7.10: Salida de MATLAB para el polinomio $p(x) = (x - 0.9)^{18}(x - 1.0)^{10}(x - 1.1)^{16}$ con z_0 con una perturbación del orden de 10^{-1}

El método no converge y se observa un número de condición peyorativo algo más alto que en el Ejemplo 7.2.

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 6.0379e+01
kappa clásico = 5.8083e+15

Iteración 0:
   0.897449448201192   1.001644040733187   1.107477340288081

Iteración 1:
   0.897700057374347   1.011429683105483   1.095445419088120

Iteración 2:
   0.899862513866852   0.999548550576457   1.100436846421089

Iteración 3:
   0.900001548684829   0.999991485448626   1.100003582350956

Iteración 4:
   0.899999999923158   0.999999999787872   1.100000000219047

Raíces encontradas:
   0.899999999999947   1.000000000000212   1.099999999999926
```

Figura 7.11: Salida de MATLAB para el polinomio $p(x) = (x - 0.9)^{18}(x - 1.0)^{10}(x - 1.1)^{16}$ con z_0 con una perturbación del orden de 10^{-2}

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 6.0379e+01
kappa clásico = 5.8083e+15

Iteración 0:
   0.899726953050597   1.001576300146546   1.099519062848221

Iteración 1:
   0.899998800258487   0.999994503285619   1.100004782140377

Iteración 2:
   0.900000000162014   0.999999999040458   1.100000000417760

Raíces encontradas:
   0.900000000000021   0.999999999999920   1.100000000000026
```

Figura 7.12: Salida de MATLAB para el polinomio $p(x) = (x - 0.9)^{18}(x - 1.0)^{10}(x - 1.1)^{16}$ con z_0 con una perturbación del orden de 10^{-3}

Si se toma una iteración inicial más cercana si que hay convergencia. Además, cuanto más cerca de la raíz se inicializa el algoritmo, menor error se obtiene.

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 6.0379e+01
kappa clásico = 5.8083e+15

Iteración 0:
    10    10    10

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  8.720219e-18.
> In pejroot_auto (line 28)
In CalculoRaicesYNumeroCondicion (line 87)

Iteración 1:
   9.392810056009884  10.000000000000000  10.000000000000000

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  3.138163e-17.
> In pejroot_auto (line 28)
In CalculoRaicesYNumeroCondicion (line 87)

Fallo: no hay convergencia (incremento en delta)
El método no convergió.
```

Figura 7.13: Iteraciones finales de la salida de MATLAB para el polinomio $p(x) = (x-0.9)^{18}(x-1.0)^{10}(x-1.1)^{16}$ con $z_0 = [10, 10, 10]$;

Usando `z0=[10,10,10];` no se observa convergencia.

Ejemplo 7.4.

A continuación se estudia como se comporta el Polinomio Wilkinson $p(x) = (x-1)(x-2)...(x-20)$ ante el Algoritmo I. Los valores de z_0 no son iguales a los anteriores (se especifican en el pie de cada figura). En este caso `r= [1,2,3,4,5,...,16,17,18,19,20];` y `l = [1,1,1,1,1,...,1,1,1,1,1];`:

```
>> CalculoRaicesYNumeroCondicion
Número de condición peyorativo para r y l dados:
kappa peyorativo = 3.6526e+10
kappa clásico = 1.4314e+14

Iteración 0:
  Columns 1 through 8

   0.998943027254040   1.999715859045374   2.999913309717541   3.998530604925514   5.000192182244871   5.999177706723710   6.999905759412040   8.000336213340955

  Columns 9 through 16

   8.999095345940754   9.999711743638795  11.000350062757533  11.998164140857495  13.001035975908245  14.002424461144939  15.000959400509409  15.999684228004991

  Columns 17 through 20

  17.000428622679859  17.998964015221485  19.001877865460497  20.000940704403352

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  1.809804e-19.
> In pejroot_auto (line 28)
In CalculoRaicesYNumeroCondicion (line 87)
```

Figura 7.14: Iteraciones iniciales de la salida de MATLAB para el Polinomio Wilkinson con z_0 con una perturbación del orden de 10^{-3}

```
Iteración 2:
  Columns 1 through 8

   1.000000000479085   1.999999602918325   3.000018342045076   3.999848158696074   5.000086475869812   6.001293030032177   6.999373242094617   7.995996852264717

  Columns 9 through 16

   8.999810404419055  10.001315307844067  11.011979354138495  11.996589317969228  12.994476426732598  13.999217216915719  14.992914668609984  16.005887869822331

  Columns 17 through 20

  16.997624545350675  18.000411997662674  19.006038168733436  19.997182807768169

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  2.231435e-20.
> In pejroot_auto (line 28)
In CalculoRaicesYNumeroCondicion (line 87)

Fallo: no hay convergencia (incremento en delta)
El método no convergió.
```


Figura 7.15: Iteraciones finales de la salida de MATLAB para el Polinomio Wilkinson con z_0 con una perturbación del orden de 10^{-3}

```
Iteración 2:
  Columns 1 through 8

   1.000000000000007   1.999999999996583   3.000000000083305   3.999999999452037   5.000000001065937   6.000000000276382   6.999999998682964   7.999999998340209

  Columns 9 through 16

   9.000000001613108  10.000000000280544  11.000000002453449  11.999999998568255  12.999999998573445  14.000000000914914  15.000000000505363  16.000000004248349

  Columns 17 through 20

  16.999999995461103  17.999999995656573  18.999999998827768  20.000000005028799

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  5.065277e-20.
> In pejroot_auto (line 28)
In CalculoRaicesYNumeroCondicion (line 87)

Fallo: no hay convergencia (incremento en delta)
El método no convergió.
```

Figura 7.16: Iteraciones finales de la salida de MATLAB para el Polinomio Wilkinson con z_0 con una perturbación del orden de 10^{-10}

El método no converge en ninguno de los casos, pero si se acerca a las soluciones correctamente. El número de condición relativo y peyorativo son notablemente elevados.

8. COMPARATIVA DE ALGORITMOS PARA RAÍCES MÚLTIPLES

El cometido de este apartado es mostrar una comparativa entre los tres algoritmos propuestos para el cálculo de raíces en polinomios con raíces múltiples. Para ello, se muestran ejemplos con la intención de resaltar las fortalezas y debilidades de cada método. Los códigos utilizados, que se adaptan para cada caso concreto (el valor asignado a cada correspondiente variable se concreta en cada ejemplo), son:

1) Método de Newton para Raíces Múltiples:

```
% Método de Newton para Raíces Múltiples

tic % iniciar temporizador

syms x % declaración de x como variable simbólica
p(x) =; % polinomio
q = expand(p); % expandir el polinomio
dq = diff(q); % derivada del polinomio

m =;                        %vector de multiplicidades
x0 =;              % iteración inicial para cada raíz
N = 1000;                           % máximo de iteraciones
tol = 1e-10;                        % tolerancia
E = zeros(length(x0), N);          % vector de errores

Q = matlabFunction(q); % polinomio expandido como función de MATLAB
DQ = matlabFunction(dq); % derivada como función de MATLAB

for k = 1:N
    % Método de Newton para Raíces Múltiples en formato vectorial:
    xi = x0 - m .* Q(x0) ./ DQ(x0);
    E(:,k) = abs(xi - x0);                  % almacenar errores

    % Criterio de parada:
    if all(E(:,k) < tol)
        fprintf('Convergencia alcanzada en %d iteraciones.\n', k);
        fprintf('Aproximaciones finales:\n');
        disp(xi');
        break
    end
    x0 = xi; % actualización
end

% Si el método no converge:
if k == N && any(E(:,k) >= tol)
    fprintf('Al menos una raíz no converge en %d iteraciones.', N);
end

tiempo_de_ejecucion=toc %tiempo de ejecución
```

2) Método de Schröder:

```
% Método de Schröder

tic % iniciar temporizador

syms x % declaración de x como variable simbólica
f_sym =; %polinomio
f1_sym = diff(f_sym, x); % primera derivada
f2_sym = diff(f1_sym, x); % segunda derivada

% Conversión a funciones numéricas
f = matlabFunction(f_sym);
f1 = matlabFunction(f1_sym);
f2 = matlabFunction(f2_sym);

x0 =;               % iteración inicial
tol = 1e-10;            % tolerancia
max_iter = 1000;         % número máximo de iteraciones

% Método de Schröder
for i = 1:max_iter
    fx = f(x0);
    f1x = f1(x0);
    f2x = f2(x0);

    denom = f1x.^2 - fx .* f2x;

    x1 = x0 - (fx .* f1x) ./ denom; % iteración del Método de Schröder
    % Criterio de parada:
    if abs(x1 - x0) < tol
        fprintf('Convergencia alcanzada en %d iteraciones\n', i);
        fprintf('Raiz encontrada: %.12f\n', x1);
        break
    end

    x0 = x1;

    if i == max_iter
        fprintf('No se alcanzo la convergencia en %d iteraciones\n',
            ↪ max_iter);
        fprintf('Aproximacion final: %.12f\n', x1);
    end
end

tiempo_de_ejecucion=toc % tiempo de ejecución
```

3) Algoritmo I:

```
tic  % iniciar temporizador

z_verdadero=; % raíces
l_verdadero=; % estructura de multiplicidad

rng(0); % semilla para reproducibilidad
z0 = z_verdadero + 1e-1*randn(size(z_verdadero)); % iteración inicial
l = l_verdadero;

% Coeficientes objetivo
a = evalg(z_verdadero, l_verdadero);

W = eye(length(a)); % Matriz de pesos identidad

tol = 1e-10; % tolerancia

% Llamada al método
[z, exito] = pejroot_auto(a, W, z0, l, tol);

if exito
    disp('Raices encontradas:');
    disp(z.');
else
    disp('El metodo no convergio');
end

tiempo_de_ejecucion=toc % tiempo de ejecución
```

4) Se hace uso también del código del Ejemplo 7.2 para el cálculo de los números de condición.

8.1. Casos óptimos para el Método de Newton para Raíces Múltiples

Para el código del Algoritmo I se toma `z_verdadero = [1;1.0000001]`, `l_verdadero = [1;1]` y `z0 = z_verdadero + 1e-1*randn(size(z_verdadero))`. Para el Método de Schröder, $f\_sym = (x-1).\hat{}2. * (x-1.0000001).\hat{}2$ y `x0 = [0.9;1.1];`. Y por último, en el Método de Newton en Raíces Múltiples $p(x) = (x-1).\hat{}2. * (x-1.0000001).\hat{}2$, `m=[2;2];` y `x0 = [0.9;1.1];`.

```
Convergencia alcanzada en 77 iteraciones.
Aproximaciones finales:
   0.999845518813586   0.999927011263583


tiempo_de_ejecucion =

   0.078160500000000
```

Figura 8.1: Salida de MATLAB al aplicar el Método de Newton para Raíces Múltiples a $p(x) = (x-1)^2 * (x-1.0000001)^2$

```
Convergencia alcanzada en 2 iteraciones
Raiz encontrada: 1.000000050000
Raiz encontrada: 1.000000050000

tiempo_de_ejecucion =

   0.053770300000000
```

Figura 8.2: Salida de MATLAB al aplicar el Método de Schröder a $p(x) = (x-1)^2 * (x-1.0000001)^2$

```
Iteración 999:
   1.000001177867740   0.999998922132895

Iteración 1000:
   1.000001177295597   0.999998922705037

El metodo no convergio

tiempo_de_ejecucion =

   0.079225300000000
```

Figura 8.3: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x-1)^2 * (x-1.0000001)^2$

```
Número de condición peyorativo para r y l dados:
kappa peyorativo = 1.8595e+07
kappa clásico = Inf
```

Figura 8.4: Números de condición de $p(x) = (x-1)^2 * (x-1.0000001)^2$

El mejor resultado se obtiene con el Método de Newton para Raíces Múltiples a pesar de tener un número de condición relativo y peyorativo elevado.

8.2. Casos óptimos para el Método de Schröder

Para el código del Algoritmo I se toma `z_verdadero = [1;1.0000001;2;2.0000001;3;3.0000001]`, `l_verdadero = [1;1;1;1;1;1]` y `z0 = z_verdadero + 1e-1*randn(size(z_verdadero))`. Para el Método de Schröder, $f\_sym = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$ y `x0 = [0.9;1.1;1.9;2.1;2.9;3.1];`. Y por último, en el Método de Newton en Raíces Múltiples $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$, `m=[1;1;1;1;1;1];` y `x0 = [0.9;1.1;1.9;2.1;2.9;3.1];`.

```
Al menos una raíz no converge en 1000 iteraciones.
tiempo_de_ejecucion =

   0.109834900000000
```

Figura 8.5: Salida de MATLAB al aplicar el Método de Newton para Raíces Múltiples a $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$

```
Convergencia alcanzada en 9 iteraciones
Raiz encontrada: 1.000000100000
Raiz encontrada: 1.000000100000
Raiz encontrada: 2.000000000000
Raiz encontrada: 2.000000100000
Raiz encontrada: 3.000000000000
Raiz encontrada: 3.000000000000

tiempo_de_ejecucion =

   0.068329800000000
```

Figura 8.6: Salida de MATLAB al aplicar el Método de Schröder a $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$

```
Iteración 998:
   1.000014048069031   0.999986053365438   2.000136878347415   1.999863266090196   3.000181127255003   2.999818927341639

Iteración 999:
   1.000014040936440   0.999986060496578   2.000136809467525   1.999863334925370   3.000181036076646   2.999819018565691

Iteración 1000:
   1.000014033941041   0.999986067490540   2.000136740638065   1.999863403710167   3.000180945079628   2.999819109608340

El metodo no convergio

tiempo_de_ejecucion =

   0.228510700000000
```

Figura 8.7: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$

El mejor resultado se obtiene con el Método de Schröder tanto en el tiempo de ejecución como en la convergencia hacia las raíces. Aún así este resultado no es correcto ya que el método converge a raíces incorrectas en la primera y sexta solución.

Tras varios intentos, se selecciona el siguiente vector `x0 = [0.99999;1.1;1.9;2.1;2.9;3.0000011];` con el que se obtiene el la siguiente salida de MATLAB:

```
Convergencia alcanzada en 11 iteraciones
Raiz encontrada: 1.000000000000
Raiz encontrada: 1.000000100000
Raiz encontrada: 2.000000000000
Raiz encontrada: 2.000000100000
Raiz encontrada: 3.000000000000
Raiz encontrada: 3.000000100000

tiempo_de_ejecucion =

   0.074720400000000
```

Figura 8.8: Salida de MATLAB al aplicar el Método de Schröder a $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$ con el vector x_0 actualizado

```
Número de condición peyorativo para r y l dados:
kappa peyorativo = 2.3184e+10
kappa clásico = 1.1745e+10
```

Figura 8.9: Números de condición de $p(x) = (x-1)*(x-1.0000001)*(x-2)*(x-2.0000001)*(x-3)*(x-3.0000001)$

Escogiendo meticulosamente x_0, el Método de Schröder converge, a pesar de tener el problema números de condición altos.

8.3. Casos óptimos para el Algoritmo I

Para el código del Algoritmo I se tomó `z_verdadero = [1;3;-2]`, `l_verdadero = [10;15;10]` y `z0 = z_verdadero + 1e-1*randn(size(z_verdadero))`. Para el Método de Schröder, $f\_sym = (x-1).\hat{}10. * (x-3).\hat{}15. * (x+2).\hat{}10$ y `x0 = [0.9;2.9;-1.9];`. Y por último, en el Método de Newton en Raíces Múltiples $p(x) = (x-1).\hat{}10. * (x-3).\hat{}15. * (x+2).\hat{}10$, `m=[10;15;10]` y `x0 = [0.9;2.9;-1.9];`.

```
Al menos una raíz no converge en 1000 iteraciones.
tiempo_de_ejecucion =

    0.103799800000000
```

Figura 8.10: Salida de MATLAB al aplicar el Método de Newton para Raíces Múltiples a $p(x) = (x-1)^{10} * (x-3)^{15} * (x+2)^{10}$

```
No se alcanzo la convergencia en 1000 iteraciones
Aproximacion final: NaN
Aproximacion final: NaN
Aproximacion final: NaN

tiempo_de_ejecucion =

    0.054449100000000
```

Figura 8.11: Salida de MATLAB al aplicar el Método de Schröder a $p(x) = (x-1)^{10} * (x-3)^{15} * (x+2)^{10}$

```
Iteración 4:
   1.003097208446631   2.990833740343314  -2.003218171857609

Iteración 5:
   0.999994655571467   3.000002050329634  -1.999987083191145

Iteración 6:
   1.000000000775494   2.999999998617652  -2.000000000952312

Raices encontradas:
   0.999999999999993   3.000000000000012  -1.999999999999998


tiempo_de_ejecucion =

   0.004955200000000
```

Figura 8.12: Salida de MATLAB al aplicar el Algoritmo I a $p(x) = (x-1)^{10} * (x-3)^{15} * (x+2)^{10}$

El mejor resultado se obtiene con el Algoritmo I.

```
Número de condición peyorativo para r y l dados:
kappa peyorativo = 3.8471e-02
kappa clásico = Inf
```

Figura 8.13: Números de condición de $p(x) = (x-1)^{10} * (x-3)^{15} * (x+2)^{10}$

El Algoritmo I converge con facilidad y precisión. En este caso el número de condición peyorativo es bajo pero el relativo es alto.

9. RESULTADOS Y DISCUSIÓN

En este apartado se realizará una discusión sobre los resultados obtenidos en lo ejemplos más relevantes del presente trabajo. Nótese que este aparatado únicamente explica y expone las ideas del alumno relativas a los resultados obtenidos, si se desean conocer las conclusiones de esta disertación se debe consultar el apartado 10. Conclusiones.

El Ejemplo 5.2 muestra como, a misma constante de error asintótica e iteración inicial, un método iterativo con orden de convergencia mayor converge a una misma solución en un menor número de iteraciones que otro método de menor orden. Sin embargo, para que este resultado sea concluyente y poder asegurar que los métodos de mayor orden son más eficientes a nivel de gasto computacional, se deben estudiar las tres suposiciones realizadas en este experimento (misma iteración inicial, raíz y constante de error asintótica).

Para hacer una comparativa justa de dos métodos, deben evaluarse ante un mismo problema, por lo que la raíz a la que convergen sí será la misma.

Además, se suele dar también la misma iteración inicial puesto que, como se explica al final del apartado 1. Introducción, en la práctica es algo más difícil conseguir una iteración inicial correcta, por lo que conseguir varias suele ser aún más complicado o en algunos casos prácticamente imposible.

Por último, la constante de error asintótica si que varía dependiendo del método e incluso de la demostración matemática que se haya llevado a cabo para probar el orden de convergencia.

El ejercicio del Ejemplo 5.4 trata el caso del polinomio $p(x) = (x-2)^7(x-3)(x-4)$; donde el Método de Newton para Raíces Múltiples no converge mientras que el Método de Schröder sí, ante una misma $x_0 = 1.999$ y raíz $x = 2$.

La principal diferencia entre estos métodos a nivel numérico es su paso. Mientras que Newton para Raíces Múltiples hace uso del valor del polinomio, su derivada y la multiplicidad de la raíz a calcular, Schröder hace uso del valor del polinomio y su primera y segunda derivada.

Además, el Método de Schröder puede, en ocasiones, permanecer estable ante derivadas o segundas derivadas prácticamente nulas, ya que la resta $[f'(x)]^2 - f(x)f''(x)$ de su denominador no tiene por que ser un número cercano a 0 necesariamente. Por el contrario, el Método de Newton para Raíces Múltiples se suele volver inestable en cualquier punto donde la primera derivada sea baja.

En último lugar, el numerador del Método de Schröder incluye el producto del polinomio y su derivada, frente al de Newton que solo incluye el polinomio, esto combinado con las diferencias mencionadas en los denominadores, hace que haya casos como este, en el que el Método de Schröder evita estancamientos y oscilaciones (ver Figura 5.1 y Figura 5.2) dada la diversificación de factores que influyen en su paso.

En el Ejemplo 5.5, se aprecia que el Método de Newton en raíces simples no converge a $x = 6$ a pesar de que el método se inicialice cerca de esta ($x_0 = 5.6$).

La primera ejecución del código realizada en el Ejemplo 5.6 con $x_0 = 1.4$ converge correctamente, pero la segunda llevada a cabo con $x_0 = 1.8$ no.

El cálculo analítico de convergencia del caso de estudio del Ejemplo 6.1 ($f(x) = (x-2)^3$) no muestra el mismo resultado que el cálculo analítico realizado con su versión perturbada expandida. Esto último se asemeja notablemente a la manera en la que MATLAB lidia con los polinomios en algoritmos iterativos.

El Ejemplo 6.3 muestra un cálculo aproximado del elevado número de condición relativo del Polinomio Wilkinson ($p(x) = (x-1)(x-2)...(x-20)$). El primer código de MATLAB, que perturba el polinomio y posteriormente aplica el Método de Newton, obtiene unos resultados muy alejados de las raíces exactas.

Posteriormente, se realiza un experimento similar a este, salvo que en lugar de calcular una sola vez las raíces con el Método de Newton, se calculan 100 veces con el comando `roots()` y se representan en la Figura 6.2. También se observa dispersión e imprecisión de las raíces calculadas frente a las originales.

La primera ejecución del Algoritmo I con el polinomio $p(x) = (x+1)^{10}(x-1)^{20}(x-2)^{30}$ en el Ejemplo 7.1 no converge. Sin embargo, cuando se inicializa el algoritmo más cerca de las raíces, el método converge con un error bajo.

Por el contrario, el Algoritmo I no converge para el polinomio $p(x) = (x+1)^{100}(x-1)^{200}(x-2)^{300}$ independientemente de la iteración inicial z_0 que se utilice en la ejecución.

Esto puede deberse a una inestabilidad numérica producida por el uso de la iteración original de Gauss-Newton en lugar de emplear la iteración simplificada que propone Zeng en Pseudocódigo del Algoritmo I.

Emplear la matriz de pesos que minimiza el error para coeficientes del polinomio mayores que 1, en lugar de la matriz de pesos unitaria, también puede ayudar a solucionar este fallo de convergencia.

Las salidas de MATLAB del Ejemplo 7.2 (ver Figura 7.4, Figura 7.5, Figura 7.6, Figura 7.7, Figura 7.8 y Figura 7.9) muestran que, cuanto más lejos se encuentra la iteración inicial de las raíces, más iteraciones se realizan y mejor precisión se obtiene.

Si se observan las iteraciones individualmente, se aprecia que en las ejecuciones de iteraciones iniciales más cercanas, el Algoritmo I converge directamente a la solución con menos pasos que los casos con z_0 más alejadas.

La introducción de estos pasos adicionales, hace que MATLAB evite cancelaciones numéricas y errores de redondeo, dado que no opera con decimales tan cercanos a 0. Es por ello que las ejecuciones con iteraciones iniciales más alejadas presentan un mayor número de iteraciones y una mejor precisión.

En el Ejemplo 7.3 ocurre lo contrario al caso anterior. Una iteración inicial más cercana a las raíces proporciona un resultado con menor error.

Esto se debe a que al ser el número de condición peyorativo 4 órdenes de magnitud mayor que en el Ejemplo 7.2 ($\kappa_{l,W} = 1.719 * 10^{-2}$ frente a $\kappa_{l,W} = 6.0379 * 10^{1}$), la matriz está significativamente más cerca de ser singular (determinante nulo) que en el caso previo. Cuando esto ocurre, la inversión de la matriz $J(z_k)^H W^2 J(z_k)$ realizada en la iteración de Gauss-Newton (6.27) se vuelve más inestable.

Por ello, se anula el efecto de cancelación numérica y error de redondeo explicado anteriormente como consecuencia de esta ligera desestabilización numérica en el algoritmo. Además, dada esta inestabilidad parcial, cuantas más iteraciones se realicen, más veces se invierte la matriz y por ello peor puede ser la solución obtenida.

En este mismo ejemplo se observa que la primera ejecución del código no converge, mientras que las siguientes, que tienen z_0 más cercano a las raíces, sí que convergen.

El Ejemplo 7.4 muestra como el Algoritmo I no puede dar una solución precisa para el Polinomio Wilkinson dado su mal condicionamiento. Aunque es cierto que dando una tolerancia muy laxa el algoritmo puede converger.

La Sección 8.1. Casos óptimos para el Método de Newton para Raíces Múltiples estudia el polinomio $p(x) = (x-1)^2(x-1.0000001)^2$, este caso tiene un número de condición relativo y un número de condición peyorativo muy elevado (ver Figura 8.4).

En primera instancia, el polinomio puede parecer complicado de resolver con los métodos presentados en este trabajo. Sin embargo, dando una iteración inicial con una precisión de apenas 10^{-1} ($x_0 = [0.9; 1.1]$), se obtiene una aproximación de las raíces con un error asumible mediante el Método de Newton para Raíces Múltiples.

Por el contrario, el Método de Schröder y el Algoritmo I no convergen correctamente. El primero, queda estancado rápidamente en una iteración que verifica el criterio de parada, pero no es una aproximación precisa de la raíz. El segundo, al tener un número condición peyorativo tan elevado, no converge correctamente debido a que la matriz WJ es prácticamente singular.

Es probable que en este caso concreto, conocer la multiplicidad de las raíces y hacer uso de ello en la iteración de Newton sea beneficioso y un factor diferencial frente a los otros métodos.

En la primera ejecución de los tres métodos del apartado 8.2. Casos óptimos para el Método de Schröder ninguno converge correctamente. Al tratarse de un problema tan mal condicionado ($p(x) = (x-1)(x-1.0000001)(x-2)(x-2.0000001)(x-3)(x-3.0000001)$) (ver Figura 8.9) el Método de Newton para Raíces Múltiples y el Algoritmo I fracasan a la hora de hallar una solución.

En cuanto al Método de Schröder, se logra la convergencia con $x_0 = [0.99999; 1.1; 1.9; 2.1; 2.9; 3.0000011]$ tras probar diversos valores. Este método resulta ventajoso frente a otros en este caso dada la diversificación de valores influyentes y la estructura de su paso.

Por último, el apartado Sección 8.3. Casos óptimos para el Algoritmo I analiza el comportamiento de los algoritmos ante el polinomio $p(x) = (x-1)^{10} * (x-3)^{15} * (x+2)^{10}$. Al ser un problema con un número de condición relativo alto y un número de condición peyorativo cercano a 0, el Algoritmo I da el mejor y único resultado correcto.

10. CONCLUSIONES

En base al análisis realizado en el apartado 9. Resultados y Discusión, este apartado presenta las conclusiones más relevantes en base a los experimentos realizados y sus resultados.

Nótese que las afirmaciones a continuación en relación a los códigos propuestos de MATLAB, se realizan en vista de los resultados obtenidos de los códigos elaborados por el alumno con ayuda del tutor. Otras versiones de códigos que repliquen los mismos métodos pueden obtener soluciones diferentes.

- **Relación entre el orden de convergencia y el número de iteraciones necesarias para convergencia (velocidad de convergencia).** En la comparación de dos métodos iterativos ante un mismo problema de raíces de polinomios, el método de mayor orden convergerá antes que el de menor orden a misma constate de error asintótica. En caso de que esta constante no coincida, se debe analizar la velocidad convergencia en una forma similar al Ejemplo 5.2.

- **Ventajas del Método de Schröder frente al Método de Newton para Raíces Múltiples.** La forma en la que el paso del Método de Schröder opera con el polinomio y la primera y segunda derivada, evita, en ocasiones, inestabilidad en puntos en los que el Método de Newton se desestabiliza. Esto, puede ocasionar que el Método de Schröder converja en casos en los que el Método de Newton para Raíces Múltiples no.

- **Fallo de convergencia en el Método de Newton en raíces simples.** El Método de Newton puede converger a raíces simples lejanas a su iteración inicial cuando se pretende calcular una raíz cercana a esta.

- **Influencia de la iteración inicial x_0 en la convergencia del Método de Schröder.** La iteración inicial seleccionada puede ocasionar que el Método de Schröder converja o no. Así como dar lugar a que el algoritmo se estanque en iteraciones que verifiquen el criterio de parada pero no sean aproximaciones precisas de las raíces.

- **Relación entre cálculo analítico y numérico de convergencia.** Los resultados de los cálculos de convergencia analíticos no se cumplen necesariamente cuando se introducen perturbaciones en los polinomios. Los métodos iterativos programados en softwares de cálculo numérico como MATLAB, encuentran una solución exacta a un polinomio perturbado del problema original, por ello varios de los ejemplos de este trabajo que teóricamente deben converger no lo hacen.

- **Inestabilidad del Polinomio Wilkinson.** Ninguno de los códigos propuestos para el Método de Newton, Algoritmo I y uso del comando `roots()`, logra dar una aproximación precisa de las raíces del Polinomio Wilkinson. Esto se debe a su mal condicionamiento e inestabilidad ante los métodos propuestos. Sin embargo, en caso de poder asumir un error más grande e incrementar el valor del parámetro de tolerancia, el Algoritmo I puede dar una solución.

- **Influencia de la iteración inicial z_0 en la convergencia del Algoritmo I de Zhonggang Zeng.** El valor de z_0 del que parten los códigos propuestos del Algoritmo I puede determinar si el método converge o no.

- **Influencia de la iteración inicial z_0 en la precisión de las aproximaciones a las raíces dadas por el Algoritmo I de Zhonggang Zeng.** Seleccionar una iteración inicial alejada de la raíz puede incrementar el número de iteraciones y minimizar el error en problemas con un número de condición peyorativo bajo (cercano a 0).

En casos con número de condición peyorativo lejano a 0, seleccionar una z_0 cercana a la raíz puede mejorar la precisión.

- **Casos atípicos de convergencia del Método de Newton para Raíces Múltiples.** El Método de Newton puede converger en problemas mal condicionados en los que el Método de Schröder y Algoritmo I no lo hagan. Un ejemplo de estos casos son polinomios de raíces múltiples cercanas entre sí de baja multiplicidad conocida. El conocimiento de la multiplicidad de las raíces puede ser ventajoso frente a otros algoritmos en estos problemas.

- **Casos atípicos de convergencia del Método de Schröder.** Existen casos en los que el Método de Schröder da aproximaciones precisas de las raíces mientras que el resto de algoritmos fracasan. Un tipo de problema en el que esto ocurre es un polinomio con raíces simples y cercanas entre sí.

Además, cuenta con la ventaja, frente al resto de métodos, de que no es necesario conocer la multiplicidad de las raíces para su uso. Así como su estructura de paso y diversificación de valores influyentes en este.

Sin embargo, para que el método converja correctamente a todas las raíces, es posible que se deban seleccionar algunas componentes del vector de iteraciones iniciales x_0 con una desviación de 10^{-5} o menor respecto de las raíces originales. Esto suele ser complicado de hacer por selección arbitraria o el Método de la Bisección.

▪ **Convergencia del Algoritmo I de Zhonggang Zeng en polinomios con número de condición Peyorativo Bajo.** El Algoritmo I converge en la mayoría de los casos con número de condición relativo alto y número de condición peyorativo bajo. Es habitual que ante estos problemas, el Algoritmo I sea el único método de los propuestos que proporcione una solución válida.

Para los casos en los que esto no ocurre, se sugiere usar la matriz que minimiza el error de coeficientes del polinomio mayores que 1, en lugar de la matriz de pesos unitaria, y la iteración del Algoritmo I simplificada en lugar de la original.

▪ **Sinergia entre algoritmos.** Los tres algoritmos propuestos para raíces múltiples se complementan entre sí abarcando numerosos casos a resolver. Convergiendo correctamente en varios casos uno de los métodos cuando los demás no.

11. LÍNEAS FUTURAS

El presente Trabajo Fin de Titulación tiene una principal línea de estudio que amplía naturalmente el trabajo realizado. Cuando se ejecuta cualquiera de los códigos relativos al Algoritmo I, se suponen conocidas las iteraciones iniciales y multiplicidades de las raíces.

En la práctica, puede ser complicado conocer estos elementos, por ello Zhonggang Zeng elaboró el Algoritmo II. Este algoritmo utiliza una metodología numérica denominada "numerical-GCD finder" (buscador numérico de máximo común divisor) para calcular la estructura de multiplicidad y la iteración inicial. Si se desea conocer más información sobre este método se recomienda consultar las fuentes bibliográficas [22] y [27].

La combinación del Algoritmo II con el Algoritmo I resulta muy útil a la hora de resolver problemas en raíces de polinomios de casos reales en el campo de la ingeniería y matemáticas. Esto se debe a que únicamente conociendo el polinomio, se puede, en muchos casos, calcular la estructura de multiplicidad e iteración inicial mediante el Algoritmo II para posteriormente hallar las raíces del polinomio con el Algoritmo I.

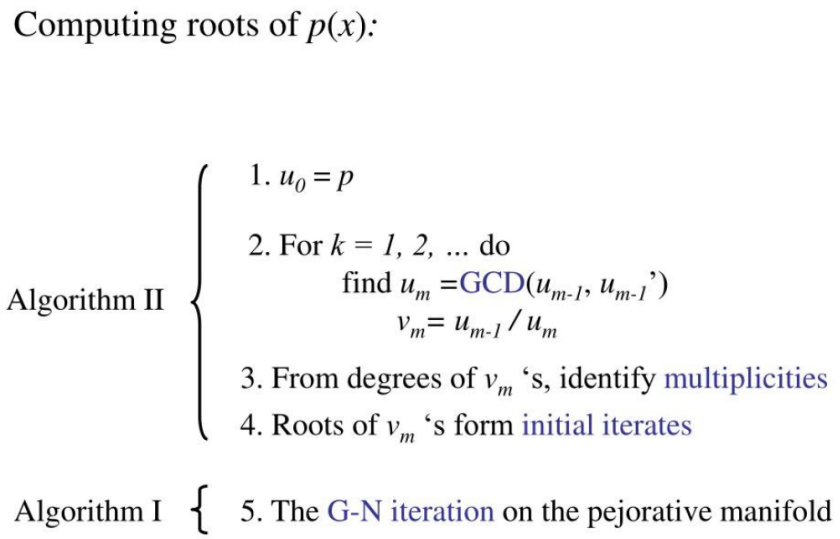

Figura 11.1: Combinación del Algoritmo II y Algoritmo I para el cálculo de raíces de un polinomio genérico $p(x)$ [27]

12. BIBLIOGRAFÍA

[1] A. Quarteroni, R. Sacco y F. Saleri. *Numerical Mathematics*. Springer, Nueva York, 2000.

[2] R. L. Burden y J. D. Faires. *Numerical Analysis*. Brooks/Cole, Pacific Grove, CA, 7ª edición, 2000.

[3] K. E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, Nueva York, 2ª edición, 2008.

[4] J. Stoer y R. Bulirsch. *Introduction to Numerical Analysis*. Springer, Nueva York, 3ª edición, 2002.

[5] G. Dahlquist y Å. Björck. *Numerical Methods*. Prentice-Hall, 1974.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling y B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3ª edición, 2007.

[7] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Filadelfia, 2ª edición, 2002.

[8] L. N. Trefethen y D. Bau. *Numerical Linear Algebra*. SIAM, Filadelfia, 1997.

[9] A. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Filadelfia, 1996.

[10] L. B. Rall. *Automatic Differentiation: Techniques and Applications*. Springer, 1981.

[11] M. T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, 2ª edición, 2002.

[12] J. M. Ortega y W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, 1970.

[13] J. E. Dennis y R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Filadelfia, 1996.

[14] L. V. Kantorovich y G. P. Akilov. *Functional Analysis in Normed Spaces*. Pergamon Press, 1964.

[15] J. F. Traub. *Iterative Methods for the Solution of Equations*. Prentice-Hall, 1964.

[16] E. Schröder. *Über unendlich viele Algorithmen zur Auflösung der Gleichungen*. *Mathematische Annalen*, 2(2):317–365, 1870.

[17] Z. Zeng. *Computing Multiple Roots of Inexact Polynomials*. *Mathematics of Computation*, 74:869–903, 2004. DOI: `https://doi.org/10.48550/arXiv.2301.07880`. Preprint: `https://arxiv.org/abs/2301.07880v1`.

[18] J. M. Müller. *Root multiplicity detection and Newton's method*. *Journal of Computational and Applied Mathematics*, 189(1-2):221–232, 2006.

[19] M. Gu y S. C. Eisenstat. *Efficient algorithms for computing a strong rank-revealing QR factorization*. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

[20] W. Gautschi. *Numerical Analysis: An Introduction*. Birkhäuser, 1997.

[21] OpenAI. *ChatGPT (GPT-5 language model)*. Disponible en: `https://openai.com/chatgpt`, 2025.

[22] J. M. McNamee. *Numerical Methods for Roots of Polynomials, Part I*. Elsevier (Studies in Computational Mathematics, 14), 2007.

[23] A. Greenbaum y T. P. Chartier. *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms*. Princeton University Press, Princeton, 2012.

[24] D. Kincaid y E. W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. American Mathematical Society (AMS), Providence, RI, 3ª edición, 2009.

[25] P. Bürgisser y F. Cucker. *Condition: The Geometry of Numerical Algorithms*. Springer, Berlín, 2013.

[26] W. Kahan. *Conserving Confluence Curbs Ill-Condition.* Informe técnico, University of California, Berkeley / Office of Naval Research, 1972.

[27] Z. Zeng. *The Tale of Polynomials.* Northeastern Illinois University, Presentación, 11 de noviembre de 2003.

[28] E. T. S. de Ingenieros Industriales, Universidad Politécnica de Madrid. *Apuntes de Álgebra (Curso 2022/2023).* Madrid, 2023.

[29] Encyclopædia Britannica. *Scientific notation: concise notation for large or small numbers.* Disponible en: `https://www.britannica.com/science/scientific-notation`.

[30] MathWorks. *format — Set output display format.* Disponible en: `https://www.mathworks.com/help/MATLAB/ref/format.html`.

[31] Escuela Técnica Superior de Ingenieros Industriales, Universidad Politécnica de Madrid. *Guía para la Elaboración y Defensa del TFG / TFM 2022.* Disponible en: `https://www.industriales.upm.es/wp-content/uploads/2022/12/Guia_TFG_TFM_2022.pdf`.

[32] Naciones Unidas. *Objetivos de Desarrollo Sostenible.* Disponible en: `https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/`.

[33] Real Academia Española (RAE), Asociación de Academias de la Lengua Española (ASALE). *Diccionario de la lengua española (DLE).* `https://dle.rae.es/`.

[34] MathWorks. *Methods used inside* `roots` *and* `solve` *functions to solve polynomial equations.* MATLAB Central, 2022. URL: `https://es.mathworks.com/MATLABcentral/answers/1895715-methods-used-inside-roots-and-solve-functions-to-solve-polynomial-equations`.

13. PLANIFICACIÓN TEMPORAL Y PRESUPUESTO

13.1. Planificación temporal

La siguiente tabla muestra las diferentes fases de la elaboración del trabajo y sus horizontes temporales:

Fase	Descripción	Fecha de inicio
1	Selección del trabajo	20/07/2024
2	Primera reunión con tutor	11/09/2024
3	Pausa temporal del trabajo por estancia en el extranjero	30/09/2024
4	Estudio, análisis de la bibliografía proporcionada y trabajo previo	16/12/2024
5	Elaboración de la Sección 4	15/01/2025
6	Reunión con tutor	06/02/2025
7	Elaboración de la Sección 5	07/02/2025
8	Reunión con tutor	21/02/2025
9	Elaboración de la Sección 6	17/03/2025
10	Reunión con tutor	01/04/2025
11	Reunión con tutor	04/04/2025
12	Reunión con tutor	09/04/2025
13	Elaboración de la Sección 7	17/04/2025
14	Reunión con tutor	23/04/2025
15	Reunión con tutor	30/04/2025
16	Revisión del trabajo hasta la fecha por tutor	06/05/2025
17	Reunión con tutor	09/05/2025
18	Corrección de errores encontrados en la revisión	10/05/2025
19	Elaboración de la Sección 8	01/07/2025
20	Reunión con tutor	07/07/2025
10	Elaboración de la Sección 1	08/07/2025
21	Elaboración de la Sección 2	09/07/2025
22	Elaboración de la Sección 3	10/07/2025
23	Elaboración de la Sección 12	11/07/2025
24	Elaboración de las Secciones 14 y 15	13/07/2025
25	Elaboración de las Sección 16	16/07/2025
26	Elaboración de las Secciones 17 y 18	17/07/2025
27	Elaboración de la Sección 19	19/07/2025
28	Elaboración de la Sección 9	31/07/2025
29	Elaboración de la Sección 10	03/08/2025
30	Inclusión de formato y portada	06/08/2025
31	Elaboración de la Sección 11	09/08/2025
32	Revisión del trabajo y mejora de calidad de contenido por parte del alumno	11/08/2025
33	Revisiones periódicas y corrección de errores	15/08/2025
34	Elaboración de la Sección 13	24/08/2025
35	Revisiones del trabajo completo	25/08/2025
36	Revisión final del trabajo con tutor	04/09/2025
37	Ensayo de defensa con tutor	15/09/2025

Cuadro 1: Tabla de Planificación Temporal

13.2. Presupuesto

Para el presupuesto del trabajo se tienen en cuenta los siguientes factores:

- Coste del equipo con el que se han realizado los experimentos y con el que se ha elaborado el documento, 500 €.

- Coste por hora trabajada de los tutores, se estiman unas 30 horas de revisión y tutorías a un sueldo estimado de 30 €/h.

- Coste por hora trabajada del alumno, se estiman unas 327 horas trabajadas de acuerdo con el número de ECTS del trabajo (12) a un coste de 6,25 €/h.

Concepto	Unidades	Coste por unidad	Coste total estimado
Ordenador portátil	1 ud	500 €/ud	500 €
Tutores	30 h	30 €/h	900 €
Alumno	327 h	6,25 €/h	2043,75 €
Coste total del TFT			3443,75 €

Cuadro 2: Tabla de presupuesto

14. EVALUACIÓN DE IMPACTOS: SOCIAL, ECONÓMICO Y MEDIOAMBIENTAL

A pesar de ser este un trabajo principalmente teórico y computacional, el impacto de este, repercute en diversos campos en los que calcular raíces de polinomios puede ser importante.

En cuanto al impacto social, este trabajo contribuye a la formación y capacitación de estudiantes, ingenieros y matemáticos en el ámbito teórico y computacional, lo que puede influir positivamente en el desarrollo profesional y la empleabilidad de estos. Además, este TFT tiene como meta una democratización del conocimiento, dado que pone a disposición de numerosos lectores ejemplificaciones de métodos iterativos sin depender de softwares a los que solo se puede obtener acceso bajo ciertas posiciones estudiantiles o profesionales o con licencias pagadas. Por último, hay un componente de contribución a la seguridad de la sociedad, ya que este campo de estudio se utiliza en sectores críticos como vibraciones, control de procesos y diseño de estructuras.

Otro de los fines de este trabajo es repercutir positivamente en la economía mediante el diseño optimizado que se ha realizado en los algoritmos, que reducen la potencia computacional requerida y el tiempo de cálculo. Este documento también contribuye a la realización de cálculos más precisos en el diseño y mantenimiento industrial, pudiendo esto evitar fallos costosos. Además, el objeto de este trabajo abarca múltiples sectores de la industria, como por ejemplo en análisis modal en la automoción o los modelos de redes eléctricas.

Por último, la reducción del tiempo de computación de los algoritmos mencionada anteriormente implica directamente una ligera disminución en el consumo eléctrico y en la huella de carbono. Sin embargo este no es el único impacto medioambiental de este TFT, pudiendo este contribuir al desarrollo de proyectos sostenibles en sectores como las energías renovables.

15. ANÁLISIS DE LOS ASPECTOS LEGALES Y ÉTICOS

El presente TFT pertenece al estudiante y a la Universidad Politécnica de Madrid, por lo que se requiere autorización para su publicación.

Durante la elaboración de este trabajo se han respetado todos los aspectos legales de derechos de autor, haciendo un uso correcto del material bibliográfico y citándolo adecuadamente de acuerdo con la *Guía para la Elaboración y Defensa del TFG / TFM 2022* de la Escuela Técnica Superior de Ingenieros Industriales de la Universidad Politécnica de Madrid [31].

Para el desarrollo y ejecución de los ejercicios se ha empleado MATLAB (Versión 9.13.0.2193358 (R2022b) Update 5), bajo licencia académica proporcionada por la Universidad Politécnica de Madrid a través de su suscripción institucional. Este software se ha utilizado únicamente con fines formativos y de investigación.

Todos los resultados de este trabajo han sido presentados de forma honesta sin ningún tipo de manipulación. Además, en base a estos se exponen las limitaciones y fortalezas de los métodos para evitar interpretaciones incorrectas que puedan llevar a un uso indebido de estos.

Por último, este documento pretende capacitar a otros de forma ética, evitando el plagio y fomentando la autoría original.

16. CONTRIBUCIÓN A LOS OBJETIVOS DE DESARROLLO SOSTENIBLE

Este apartado muestra como el presente TFT contribuye a los diferentes ODS (Objetivos de Desarrollo Sostenible). Para más información al respecto de estos consultar la fuente [32].

ODS 4. Educación de Calidad. El trabajo contribuye a la educación universitaria y al desarrollo de competencias STEM (Science, Technology, Engineering, and Mathematics).

Figura 16.1: ODS 4.

ODS 7. Energía asequible y no contaminante. Los métodos numéricos eficientes, como los presentados en este trabajo, contribuyen a la optimización y diseño en el campo de estudio de las energías renovables.

Figura 16.2: ODS 7.

ODS 9. Industria, innovación e infraestructura. Los algoritmos presentados en este documento ayudan a la mejora e innovación en procesos industriales.

Figura 16.3: ODS 9.

ODS 12. Producción y consumo responsables. El uso de algoritmos eficientes reduce el consumo de energía y recursos en cálculos extremadamente complejos.

Figura 16.4: ODS 12.

ODS 13. Acción por el clima. Por la misma razón que el ODS 12 hay una reducción en la huella de carbono.

Figura 16.5: ODS 13.

17. ABREVIATURAS, UNIDADES Y ACRÓNIMOS

No se han utilizado unidades ni acrónimos, únicamente se ha hecho uso de la notación científica como abreviatura, tanto a la hora de redactar el trabajo como en los códigos y salidas de MATLAB. Para más información al respecto, consultar las fuentes bibliográficas [29] y [30].

18. GLOSARIO

Todas las definiciones han sido tomadas del Diccionario de la lengua española de la Real Academia Española. [33].

Polinomio: Expresión compuesta por una suma finita de productos de variables y constantes.

Iteración: Acción y efecto de iterar.

Iterar: Repetir (volver a hacer lo que se había hecho).

Algoritmo: Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Derivada: Valor límite de la relación entre el incremento del valor de una función y el incremento de la variable independiente, cuando este tiende a cero.

Variable: Magnitud que puede tener un valor cualquiera de los comprendidos en un conjunto.

Tolerancia: Máxima diferencia que se tolera o admite entre el valor nominal y el valor real o efectivo en las características físicas y químicas de un material, pieza o producto.

Convergencia: Acción y efecto de converger.

Converger: Dicho de una sucesión o de una función: Aproximarse a un límite.

Límite: En una secuencia infinita de magnitudes, magnitud fija a la que se aproximan cada vez más los términos de la secuencia. Así, la secuencia de los números 2n/(n+1), siendo n la serie de los números naturales, tiene como límite el número 2.

19. ANEXO A. MÉTODO DE LA BISECCIÓN

El siguiente código muestra un código que aplica el Método de la Bisección. Este se puede adaptar fácilmente a cualquier función escogiendo adecuadamente los valores de f, x, a y b.

```
%Se define la función de la que se desea aproximar la iteración inicial
f = @(x) (x-1).*(x-2).*(x-3);
%Mallado de valores en el que se trabaja:
x=0:0.01:6;
% Para saber el intervalo [a,b] en el que evaluar la sucesión puede ser
% conveniente dibujar la función f(x) en un intervalo cualquiera para
% así ver donde se encuentran los ceros (raíces) de la función.

plot(x,f(x))

a=0;
b=4;

%Se construye la sucesión por el Método de la Bisección
an = a;
bn = b;

%Tolerancia delta y número de iteraciones n
delta = 1e-6;
n=22;

%Bucle para construir la sucesión
xn = (an+bn)*0.5;
for i=1:n
    if f(xn)==0;
        break;
    elseif f(an)*f(xn)<0
        bn=xn;
    else
        an=xn;
    end
   xn = (an+bn)*0.5;
end
% Si se desea más precisión se puede reducir el valor de delta.
% Se imprime por pantalla el valor encontardo de xn, en caso de que la
% función tenga varias raíces, se deben probar distintos valores de a,
% b, an y bn para así encontrar una iteración inicial para cada raíz.

% En este caso el método converge hacia la iteración inicial la raíz x=2
fprintf('El valor es: %d\n', xn)
```

```
>> Biseccion
El valor es: 2
```

Figura 19.1: Iteración inicial hallada por el Método de la Bisección.

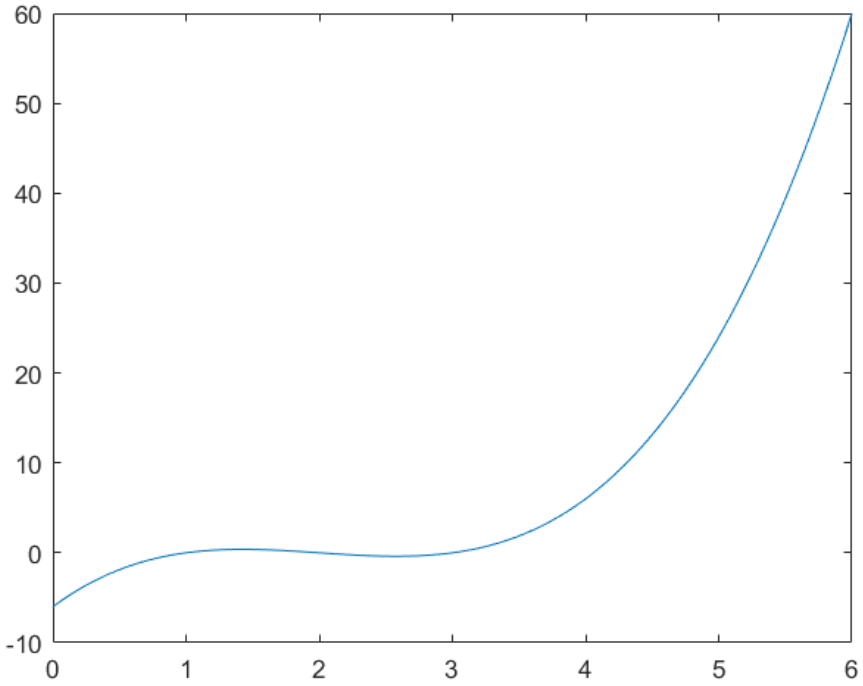

Figura 19.2: Figura generada por la línea plot(x,f(x))

POLITÉCNICA

Escuela Técnica Superior de Ingenieros Industriales
Universidad Politécnica de Madrid

José Gutiérrez Abascal, 2. 28006 Madrid
Tel.: 91 336 3060
info.industriales@upm.es

www.industriales.upm.es