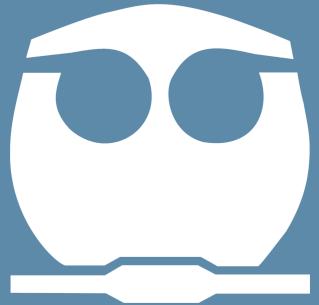


Métodos numéricos usando Python

con aplicaciones a la Ingeniería Química

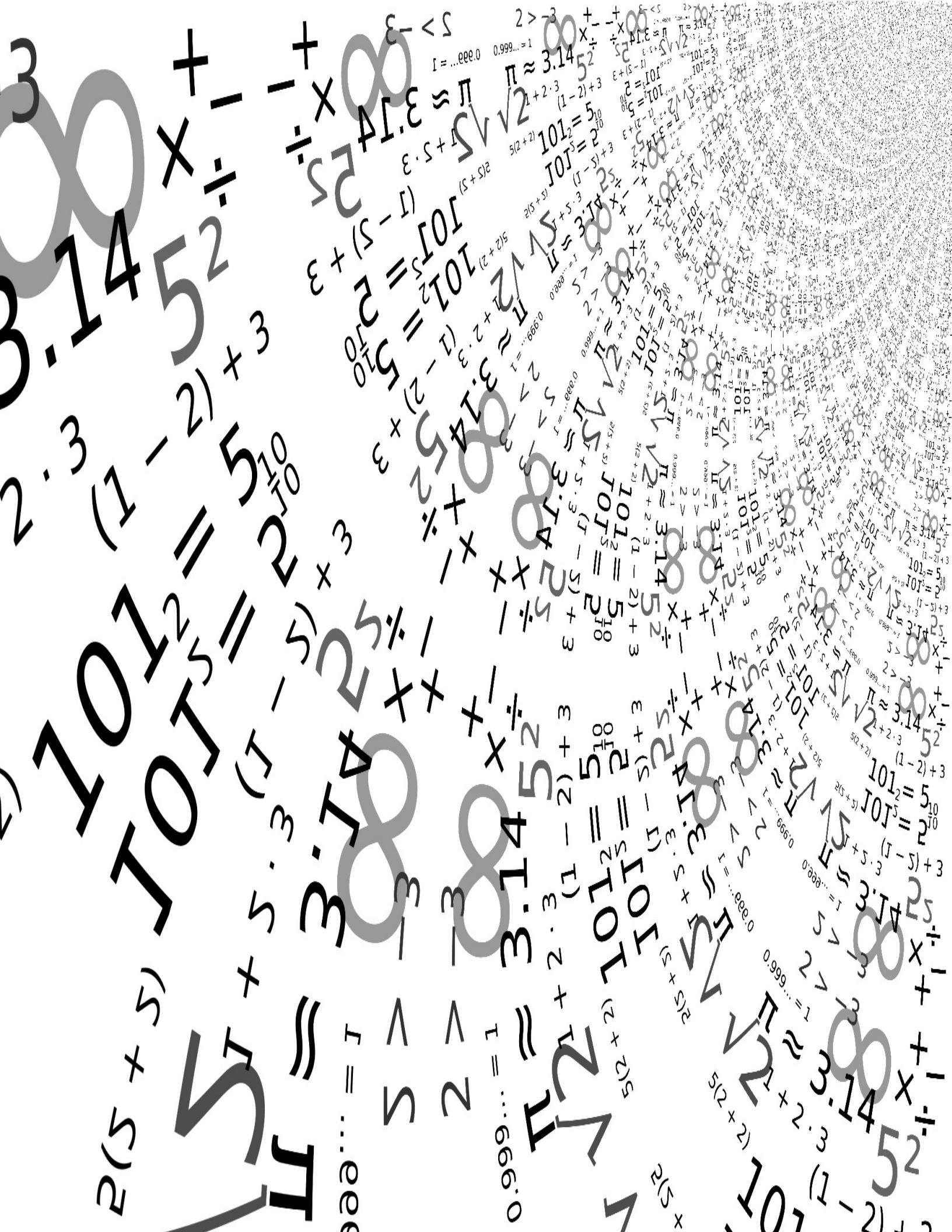
Juan Carlos Jiménez Bedolla



Universidad Nacional Autónoma de México
Facultad de Química

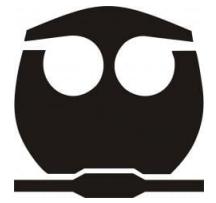
Métodos numéricos usando Python

con aplicaciones a la Ingeniería Química





UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE QUÍMICA
DEPARTAMENTO DE MATEMÁTICAS



Métodos numéricos usando Python

con aplicaciones a la Ingeniería Química

Juan Carlos Jiménez Bedolla

Primera edición 2022
Fecha de edición: 16 de febrero de 2022

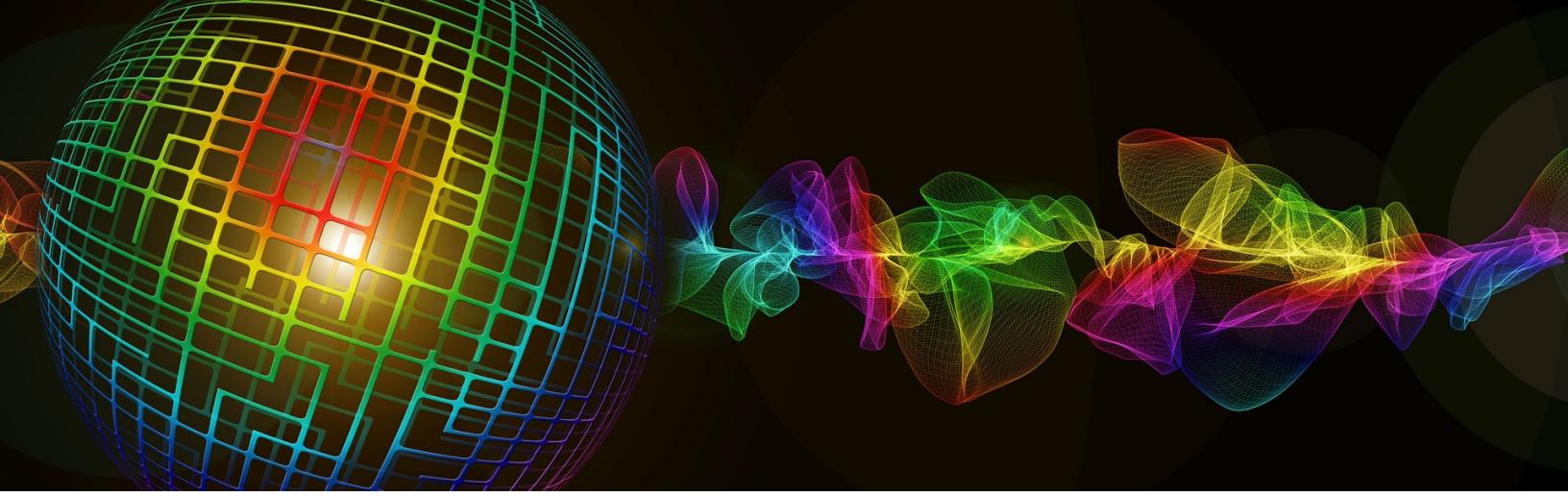
D.R. © 2022 UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Ciudad Universitaria, Alcaldía Coyoacán,
C.P. 04510, Ciudad de México.

ISBN:978-607-30-5826-1

Tamaño: 6.0 MB
Tipo de impresión: PDF
Tiraje: 1 (web)

“Prohibida la reproducción total o parcial por cualquier medio, sin la autorización escrita del titular de los derechos patrimoniales”.
Impreso y hecho en México

Publicación autorizada por el Comité Editorial de la Facultad de Química



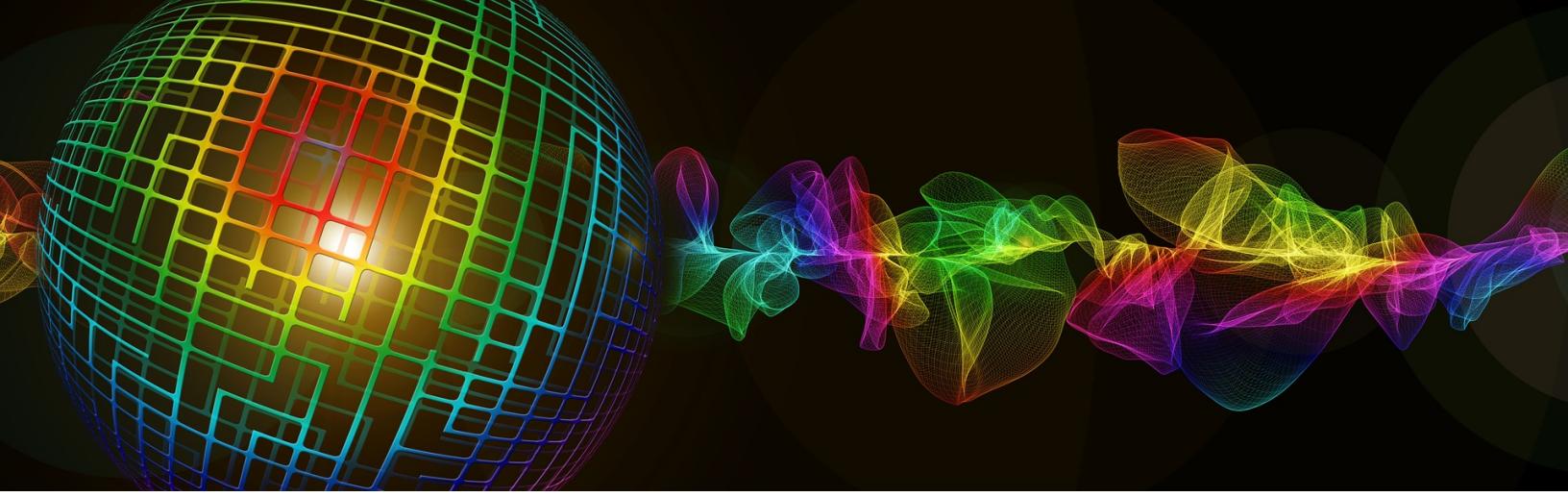
Índice general

Enlace para código fuente	I
Prólogo	V
1 Conceptos preliminares	1
1.1 ¿Qué son los métodos numéricos?	1
1.2 Sistemas de numeración	2
1.2.1 Sistema decimal	3
1.2.2 Sistema binario	3
1.2.3 Sistema octal	4
1.2.4 Sistema hexadecimal	4
1.3 Números fraccionarios	5
1.3.1 Sistema decimal	5
1.3.2 Sistema binario	5
1.3.3 Sistema octal	5
1.3.4 Sistema hexadecimal	6
1.4 Convertir de un sistema al sistema decimal	6
1.4.1 Números enteros	6
1.4.2 Números con fracciones	7
1.5 Cómo se almacenan los números en la memoria de la computadora	9
1.5.1 Notación científica	9
1.6 Errores	11
1.7 Serie de Taylor	13

2	Ajuste de curvas	25
2.1	Interpolación	26
2.1.1	Interpolación lineal	27
2.1.2	Interpolación cuadrática	30
2.1.3	Polinomio de interpolación de Newton	32
2.1.4	Polinomio de Lagrange	36
2.1.5	Algoritmo de Neville	40
2.1.6	Spline lineal	43
2.1.7	Spline cuadrático	44
2.1.8	Spline cúbico	46
2.1.9	Interpolación Nearest	49
2.1.10	Interpolación en dos dimensiones	52
2.2	Regresión por mínimos cuadrados	53
2.2.1	Lineal	54
2.2.2	Polinómica	57
2.3	No lineal	60
2.3.1	Exponencial	60
2.3.2	Potencial	62
2.3.3	Recíproco o hiperbólico	63
2.4	Múltiple	65
3	Ecuaciones no lineales	71
3.1	Búsqueda incremental	73
3.2	Métodos cerrados	75
3.2.1	Bisección, Bolzano o partición binaria	75
3.2.2	Regla falsa o regula falsi o falsa posición	78
3.3	Métodos abiertos	83
3.3.1	Newton-Raphson	83
3.3.2	Newton-Raphson modificado	88
3.3.3	Secante	93
3.3.4	Muller	97
3.3.5	Punto fijo	102
3.3.6	Wegstein	106
3.3.7	Raíces de polinomios	110
3.3.8	Raíces complejas	112
3.3.9	El fractal de Newton	116
4	Sistemas de ecuaciones lineales	123
4.1	Métodos directos	126
4.1.1	Eliminación de Gauss	126
4.1.2	Gauss-Jordan	130
4.1.3	Sistemas homogéneos	132
4.1.4	Sistemas rectangulares	134

4.1.5	Descomposición LU	136
4.1.6	Inversa multiplicación	139
4.1.7	Sistemas de ecuaciones lineales múltiples	142
4.1.8	Regla de Cramer	145
4.2	Métodos iterativos	146
4.2.1	Jacobi	146
4.2.2	Gauss-Seidel	150
4.2.3	Método SOR	154
5	Sistemas de ecuaciones no lineales	157
5.1	Punto fijo multivariable	157
5.2	Gauss-Seidel multivariable	159
5.3	Newton-Raphson multivariable	161
6	Diferenciación numérica	165
6.1	Diferenciación	165
6.1.1	Repasso de la serie de Taylor	165
6.1.2	Teorema del valor medio	165
6.1.3	Diferencias hacia adelante	166
6.1.4	Diferencias hacia atrás	166
6.1.5	Diferencias centradas	168
6.1.6	Diferencias de 2º orden	168
7	Integrales	173
7.1	Fórmulas de Newton-Cotes	173
7.1.1	Regla de los rectángulos	174
7.1.2	Regla de los trapecios	177
7.1.3	Regla de Simpson $\frac{1}{3}$	180
7.1.4	Regla de Simpson $\frac{3}{8}$	184
7.1.5	Integración de Romberg	189
7.1.6	Fórmulas cerradas de Newton-Cotes de grado superior	193
7.1.7	Fórmulas abiertas de Newton-Cotes	193
7.2	Cuadraturas	195
7.2.1	Gauss-Legendre	197
7.2.2	Cuadraturas con más puntos	200
7.3	Integrales múltiples	203
7.3.1	Integrales dobles y triples	203
7.4	Integral con intervalos desigualmente espaciados	206
7.5	Integrales impropias	206
7.6	Integrales de funciones con asíntotas verticales	209

8	Ecuaciones diferenciales ordinarias	211
8.1	Problemas de valor inicial	213
8.1.1	Método de Euler	213
8.1.2	Método de Euler modificado	216
8.1.3	Runge-Kutta de 4º orden	221
8.1.4	Sistemas de ecuaciones diferenciales de primer orden	225
8.2	Problemas de valor en la frontera	228
8.2.1	Método de disparo	230
8.2.2	Método de diferencias finitas	235
9	Ecuaciones diferenciales parciales	241
9.1	Ecuaciones diferenciales parciales elípticas	243
9.1.1	Métodos numéricos para las EDP elípticas	244
9.2	Ecuaciones diferenciales parciales parabólicas	252
9.2.1	Método de diferencias progresivas (<i>forward differences</i>)	253
9.2.2	Método de diferencias regresivas (<i>backward differences</i>)	255
9.2.3	Método de Crank-Nicholson	255
9.2.4	Método explícito	256
9.2.5	Método implícito	259
9.2.6	Método implícito de Crank-Nicolson	263
9.3	Ecuaciones diferenciales parciales hiperbólicas	266
	Bibliografía	268



Índice de programas

1.1	Convertir del sistema decimal a otros sistemas	7
1.2	Error de redondeo en Python	11
1.3	Serie de MacLaurin de $\cos(x)$	16
1.4	Serie de Taylor de $\ln(x)$	18
1.5	Serie de MacLaurin de e^x	20
1.6	Serie de MacLaurin de e^{ix}	22
2.1	Interpolación lineal vs regresión lineal	25
2.2	Interpolación lineal carbonato de sodio	29
2.3	Interpolación polinómica de Newton	33
2.4	Interpolación polinómica de Lagrange	39
2.5	Interpolación de Neville	41
2.6	Interpolación de spline cúbico	48
2.7	Interpolación Nearest	49
2.8	Interpolación con 6 métodos	51
2.9	Interpolación en 2 dimensiones	52
2.10	Regresión lineal	55
2.11	Regresión cuadrática	57
2.12	Regresión cúbica	58
2.13	Regresión exponencial	61
2.14	Regresión potencial	62
2.15	Regresión recíproca o hiperbólica	64
2.16	Regresión lineal múltiple	66
2.17	Regresión no lineal múltiple	68
3.1	Método gráfico	71
3.2	Método de bisección	76
3.3	Método de la regla falsa	81
3.4	Método de Newton-Raphson	85
3.5	Método de Newton-Raphson de <i>optimize</i>	87

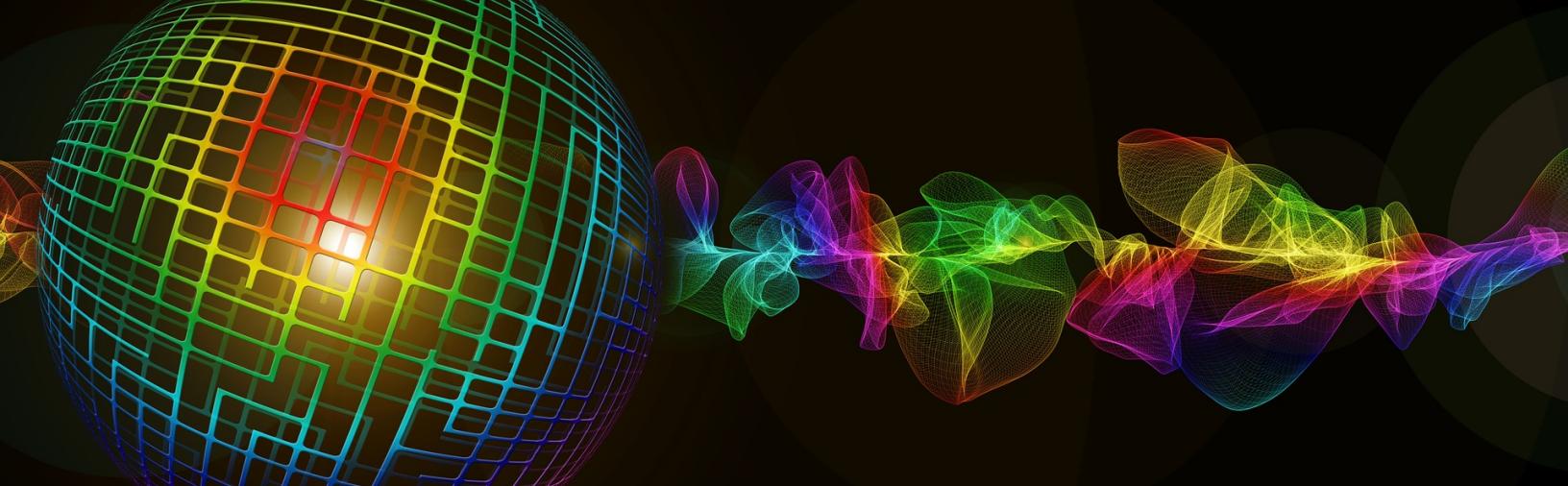
3.6	Método de Newton modificado	90
3.7	Método de Haley de <i>optimize</i>	92
3.8	Método de la secante	95
3.9	Método de Muller	101
3.10	Método del punto fijo	104
3.11	Método de Wegstein	108
3.12	Raíces de un polinomio por el método de Newton-Raphson	111
3.13	Raíces de un polinomio con la función <i>roots</i>	112
3.14	Método de Newton-Raphson complejos	113
3.15	Método de la secante complejos	114
3.16	Método de Muller complejos	115
3.17	Método de Newton-Raphson devuelve iteraciones	116
3.18	Método de Newton-Raphson, valor inicial vs iteraciones para $f(x) = x^2 - 2$	117
3.19	Método de Newton-Raphson, valor inicial vs iteraciones para $f(x) = x^3 - 1$	118
3.20	Método de Newton-Raphson, fractal para $f(x) = x^3 - 1$	119
4.1	Método de eliminación de Gauss	128
4.2	Método de eliminación de Gauss-Jordan	131
4.3	Método de reducción escalonada	133
4.4	Método de sistemas rectangulares	135
4.5	Método de descomposición LU	138
4.6	Método de inversa-multiplicación	141
4.7	Método de inversa-multiplicación múltiple	144
4.8	Regla de Cramer	145
4.9	Método de Jacobi	149
4.10	Método de Gauss-Seidel	151
4.11	Método de Gauss-Seidel sustituciones sucesivas	153
4.12	Método de SOR	154
5.1	Método del punto fijo multivariable	158
5.2	Método de Gauss-Seidel multivariable	160
5.3	Método de Newton-Raphson multivariable	163
6.1	Diferenciación numérica	171
7.1	Método de los rectángulos	175
7.2	Método de los trapezios	179
7.3	Método de Simpson $\frac{1}{3}$	182
7.4	Método de Simpson $\frac{3}{8}$	187
7.5	Método de Romberg	192
7.6	Cuadratura de Gauss-Legendre de 2 puntos	199
7.7	Cuadratura de Gauss-Legendre con n puntos	202
7.8	Integral doble	203
7.9	Integral triple	205
7.10	Integral con intervalos desigualmente espaciados	206
7.11	Integral impropia	207
7.12	Integral impropia π	208
7.13	Integral impropia 3	210
8.1	Euler con pasos	215
8.2	Euler modificado con pasos	217

8.3	Euler modificado mezclas	220
8.4	Método de Runge-Kutta de 4º orden	224
8.5	Sistema de ecuaciones diferenciales de primer orden	227
8.6	Problema de valores en la frontera: tiro 1	231
8.7	Problema de valores en la frontera: tiro 2	232
8.8	Problema de valores en la frontera: tiro 3	233
8.9	Problema de valores en la frontera: diferencias finitas	237
9.1	Solución de Laplace con condiciones Dirichlet	248
9.2	Solución de Laplace con condiciones Neumann-Dirichlet	251
9.3	Temperatura de la barra en el tiempo, método explícito	258
9.4	Temperatura de la barra en el tiempo, método implícito	261
9.5	Temperatura de la barra en el tiempo, método implícito de Crank-Nicolson	265

Todo el código fuente lo podrá encontrar en el siguiente enlace:

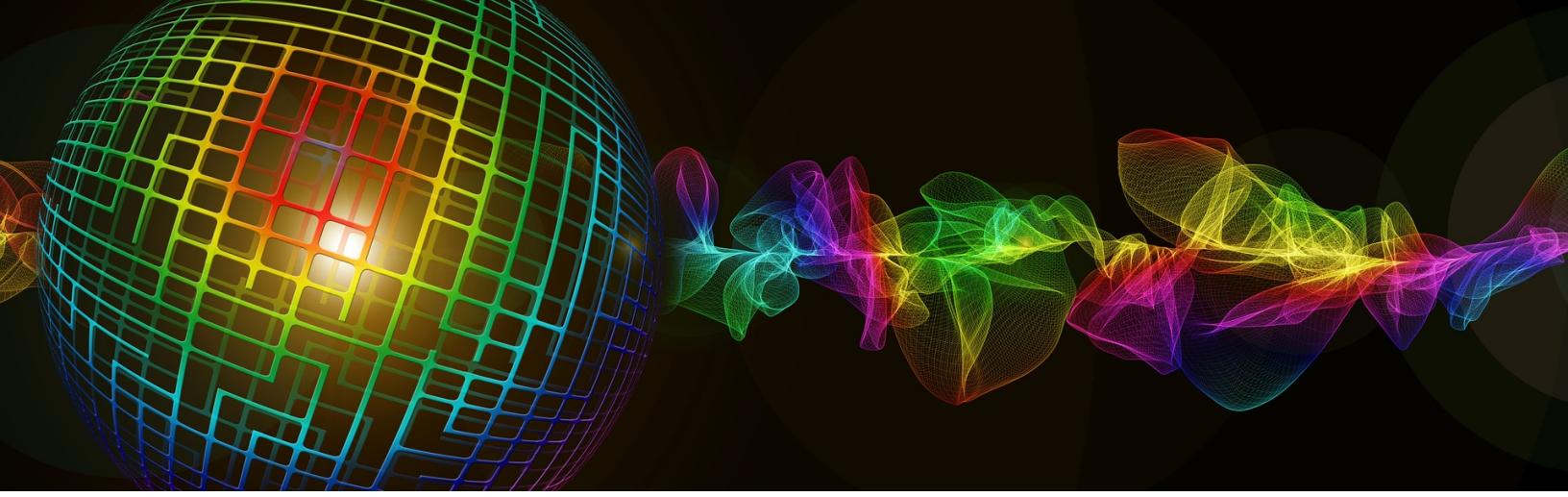
<https://github.com/jcjmenezb123/MNPython-Libro#readme>





Índice de algoritmos

1	Método de bisección	75
2	Método de regla falsa	79
3	Método de Newton-Raphson	85
4	Método de Newton modificado	90
5	Método de la secante	94
6	Método de Muller	100
7	Método del punto fijo	104
8	Método de Wegstein	107
9	Eliminación de Gauss	127
10	Eliminación de Gauss-Jordan	130



Prólogo

Los métodos numéricos son procedimientos matemáticos que se aplican para resolver diversos tipos de problemas como:

1. **Interpolación.** La toma de resultados en un laboratorio genera una tabla de datos (variable independiente x , variable dependiente $f(x)$), y se requiere conocer un valor que no se incluye en la tabla. Para encontrar un polinomio que toque todos los puntos tabulados, usamos técnicas de interpolación. El dato que se desea interpolar debe estar en el dominio de las x . El ajustar un polinomio que toque todos los puntos puede generar oscilaciones importantes y se pueden obtener resultados inesperados, fuera del contexto del problema, en estos casos se recomienda utilizar trazadores o splines, pueden ser lineales, cuadráticos o incluso cúbicos, esta técnica asegura que la oscilación entre los puntos no sea mayor del grado del trazador.
2. **Ecuaciones no lineales.** Es común escuchar la expresión “raíz de una ecuación”, el término raíz se deriva de los cuadrados perfectos de un número; por ejemplo, el cuadrado de 2 es 4, el cuadrado de 3 es 9; si se desea saber el origen del cuadrado perfecto de 4 es 2, es decir, la raíz de 4 es 2. Expresando esta idea en términos algebraicos sería $x^2 = 4$, donde 4 es el valor que conocemos y deseamos conocer su raíz x , por lo tanto, $f(x) = x^2 - 4 = 0$, es fácil deducir que el valor de x es 2, la respuesta es casi obvia, pero qué pasa con la raíz de 2, aquí se complica un poco el tema porque se debe encontrar un valor de x tal que su cuadrado sea 2, es decir, se desea encontrar la raíz de 2. Si usamos la calculadora para obtener la raíz de dos, nos damos cuenta de que es un número irracional, ¿qué técnica se puede usar para obtener este valor? Esta técnica debe ser fácil de implementar y localizar las raíces de una ecuación, incluso las raíces complejas. Si graficamos valores de x vs $x^2 - 2$ vemos una gráfica que cruza el eje x en dos puntos, éstas son las dos raíces reales de $f(x)$. La gráfica es una primera aproximación a la raíz de manera visual, pero los métodos no tienen ojos para observar dónde cruza $f(x)$, además las raíces complejas no cruzan el eje x . Aquí se utilizan otras técnicas que se pueden clasificar en métodos cerrados y abiertos, los métodos cerrados sólo localizan las raíces reales que cruzan el eje x , mientras que los métodos abiertos localizan las raíces que tocan o cruzan

el eje x y algunos de estos métodos también localizan raíces complejas, sólo si los valores iniciales propuestos son valores complejos.

3. **Sistemas de ecuaciones.** Los sistemas de ecuaciones representan las relaciones que tienen los elementos dentro de un sistema; por ejemplo, las concentraciones de una sustancia en cada uno de los platos en una torre de destilación, la cantidad de cada una de las sustancias que se requieren para fabricar distintos tipos de fertilizantes, las diferentes presiones parciales que tiene cada uno de los gases en una mezcla. En estos casos, un elemento se relaciona con el resto de cierta manera, la cual queda expresada en una ecuación y todo el sistema se expresa con un sistema de ecuaciones, las cuales pueden ser lineales o no lineales. Los métodos numéricos encuentran el conjunto solución, es decir, es la solución para la ecuación 1, pero también para la ecuación 2, y en general, para todo el sistema.
4. **Integrales.** Las integrales son una herramienta muy poderosa cuando se trata de sumar un total de pequeños eventos, las técnicas de integración analítica se pueden aplicar a un gran número de funciones, pero ¿qué hay de aquellas que no se pueden integrar porque son tan complicadas que no es posible aplicar ninguna de estas técnicas? Aquí es donde los métodos numéricos pueden aproximar polinomios de grado 1, 2, 3, los cuales son fáciles de integrar y aproximar el resultado. Hay otras técnicas que utilizan la cuadratura para llegar a esa aproximación, cualquiera de las dos siempre serán más fáciles que las técnicas analíticas, con el correspondiente costo de la aproximación.
5. **Ecuaciones diferenciales ordinarias.** Existe un grupo de ecuaciones que expresan la razón de cambio de una variable dependiente con respecto al cambio de una variable independiente, éstas son las ecuaciones diferenciales ordinarias; para resolverlas se requiere de un valor inicial, es decir, el estado inicial cuando la variable independiente marca el inicio y la variable dependiente marca el estado inicial, se requiere de este dato para encontrar la solución final, ya que sería imposible determinar el final si no se conoce el inicio; de otra manera, cualquier cosa podría ser el final. Aquí los métodos numéricos otra vez hacen aproximaciones para llegar a la solución.
6. **Ecuaciones diferenciales parciales.** Las ecuaciones que estudia un ingeniero químico involucran más de una variable independiente; por ejemplo, la temperatura (variable dependiente) cambia con respecto a lo largo (eje x , variable independiente) y ancho (eje y , variable independiente) de una placa metálica. La temperatura cambia en ambos sentidos y, por lo tanto, la expresión que relaciona las tres variables (una dependiente y dos independientes) es una ecuación diferencial parcial. Existen métodos analíticos para resolver este tipo de ecuaciones, pero los métodos numéricos son más sencillos de aplicar y dan una solución aproximada.
7. **Optimización.** En muchas áreas de la Ingeniería se busca optimizar recursos, ya sea materiales o humanos, maximizando los beneficios o minimizando los costos. Matemáticamente hablando, se trata de encontrar un máximo o mínimo de una función en un rango específico o general, hablamos de un máximo/mínimo local o global, pero cumpliendo una serie de restricciones. Aquí también los métodos numéricos hacen aproximaciones para encontrar estos puntos de interés.

Como se puede observar, los métodos numéricos son herramientas efectivas que llegan a una solución muy aproximada y se pueden aplicar a cualquier tipo de ecuaciones, no importa lo complejas o difíciles que sean.

En algunos libros se habla de “métodos numéricos para ingenieros” o “métodos numéricos para economistas” o “métodos numéricos para ciencias biológicas”, la verdad es que los métodos numéricos no son especiales para cada una de estas disciplinas, los métodos numéricos se pueden aplicar a cualquier rama de la Ingeniería y ciencias en general.

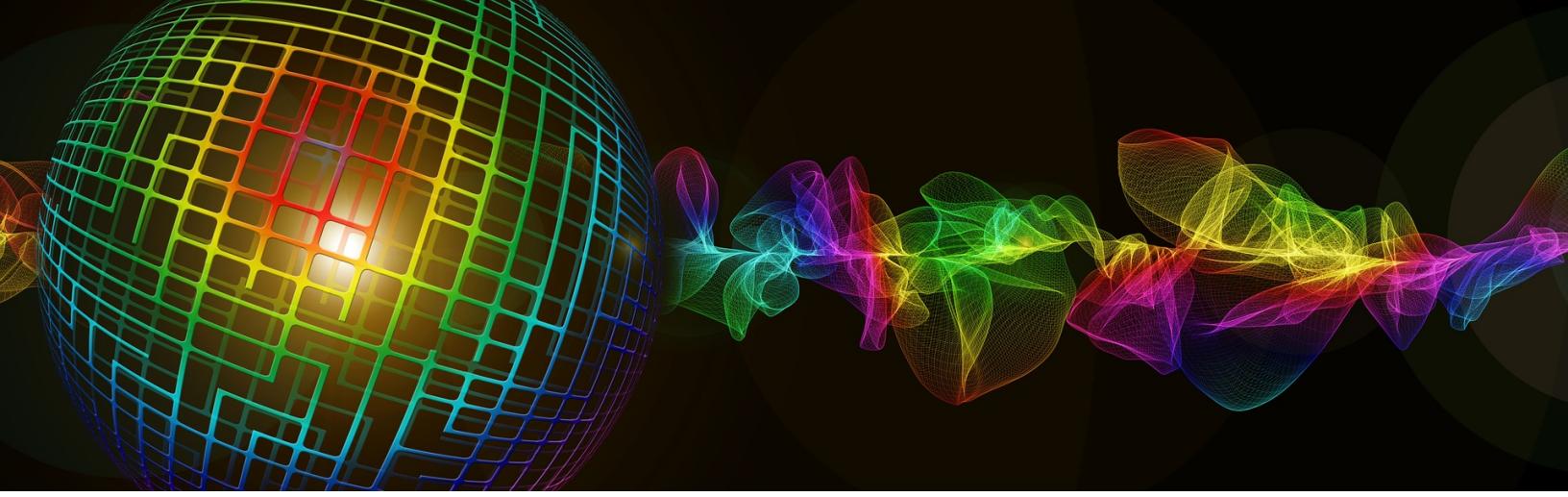
También encontrarán algunos títulos como: métodos numéricos con Java, C++, Fortran, Python, Matlab, Mathcad, Mathematica, Maple, Scilab, VBA, Pascal, Javascript, etc. Básicamente los métodos numéricos son algoritmos que se pueden programar en cualquier lenguaje de computadora, y como algoritmos son susceptibles de programarlos como cualquier otro algoritmo. No existe un Newton-Raphson para C++ o Java, el método de Newton-Raphson es el mismo en cualquier lenguaje, lo único que los diferencia son las reglas sintácticas y estructurales de cada lenguaje.

El lenguaje usado en el presente trabajo es Python, dado que es un lenguaje muy sencillo de aprender y cuenta con una gran lista de paquetes y módulos con funciones y clases orientadas a la solución de múltiples métodos.

El presente libro explica los métodos numéricos como algoritmos que resuelven un problema en particular y, además, expone la codificación en Python como herramienta para realizar los cálculos que, en muchos casos, son tediosos y con posibilidad de cometer errores humanos, es por eso que se sugiere el uso de un lenguaje computacional para resolver esta problemática.

Python es un lenguaje muy usado para el cálculo científico y la Ingeniería, gracias a los paquetes que incorpora, como lo son: *scipy*, *numpy*, *sympy*. Estos paquetes contienen funciones que resuelven muchos de los métodos numéricos, de tal manera que sólo se pasan los parámetros necesarios y devolverán el resultado correcto.

La comunidad de Python sigue incorporando nuevas funciones y mejorando las ya existentes. Puede decirse que Python es un lenguaje en constante evolución, además, abarca otros campos de la tecnología, como lo son: inteligencia artificial, redes neuronales, ciencia de datos, aplicaciones móviles, juegos, *bots*, etc.



1. Conceptos preliminares

1.1 ¿Qué son los métodos numéricos?

Los métodos numéricos son procedimientos matemáticos que se aplican para resolver un problema en específico. Los procedimientos implican tareas a realizar de manera repetitiva hasta cumplir con un criterio de aproximación y entonces se detienen los cálculos. Hacer tareas repetitivas resulta tedioso y, por otro lado, es probable cometer errores. Los ingenieros tenemos una herramienta que puede hacer tareas repetitivas de manera muy rápida y sin cometer errores, esa herramienta es la computadora.

Hoy día, la tecnología ha crecido exponencialmente y podemos tener una computadora al alcance de nuestras manos, tenemos teléfonos inteligentes, tabletas con acceso a internet con una gran capacidad de realizar cálculos a muy alta velocidad, con capacidades gráficas, para poder comunicar los resultados al otro lado del mundo.

Estos dispositivos se pueden programar para lograr hacer las tareas repetitivas, de tal manera que se obtengan los resultados siguiendo el algoritmo definido por el programador. Aquí se manejan dos conceptos primordiales para lograr que una computadora haga su trabajo.

Por un lado, se requiere programar. Programar a veces suena tan sencillo que cualquier persona que logra hacer un programa que calcule una suma de vectores se considera programador. Así era algunos años atrás, ahora se ha vuelto toda una disciplina de la Ingeniería, y no sólo se requiere un programa, a veces, es necesaria una base de datos, una arquitectura de computadoras y una red de comunicaciones, que en su conjunto requiere de muchas áreas de especialización de la Ingeniería de sistemas.

Por otro lado, se requiere el algoritmo que resuelva el problema. Aquí los métodos numéricos son exactamente eso, desde el punto de vista de los sistemas, los métodos numéricos son algoritmos precisos que logran llegar a un resultado de manera sistemática. En su mayoría, los métodos numéricos son técnicas que requieren hacer un conjunto de tareas para llegar al resultado cumpliendo un

criterio de convergencia. Aquí la programación es de mucha ayuda, ya que, si se hacen los mismos cálculos en papel y lápiz o, en el mejor de los casos, con calculadora y lápiz, resulta ser una tarea épica casi imposible.

Como comenté anteriormente, hoy se tienen muchos avances en la tecnología y, por ende, en los lenguajes de programación, los lenguajes se han vuelto más sencillos de entender y de programar. Un ingeniero químico tiene todas las habilidades que se requieren para crear un programa que resuelva sus necesidades de cálculo, su formación desarrolla una estructura lógica del pensamiento, por lo que logra adaptarse a la lógica de cualquier lenguaje de programación de manera casi inmediata. Por lo tanto, un ingeniero químico es capaz de crear programas de computadora que resuelvan problemas de diseños de equipos, producción en las plantas, optimización de recursos, etc. Todo lo que requiera la exigencia de la industria.

Definición 1.1.1 — Métodos numéricos. Los métodos numéricos son procedimientos matemáticos que resuelven, de una manera sencilla, problemas que involucran ecuaciones muy complicadas.

El ingeniero no puede invertir mucho tiempo en la solución del problema, el ingeniero requiere la solución lo antes posible para tomar decisiones importantes que hagan que la producción no se detenga o el diseño de la planta se termine cuanto antes. Entonces surge la necesidad de proveer al ingeniero de una herramienta de cálculo para obtener resultados lo más rápido posible.

Los métodos numéricos no llegan a una solución exacta en algunos casos, pero sí llegan a una solución lo suficientemente aceptable para tomarla como correcta. Para llegar a esa solución se requiere, a veces, que se hagan muchos cálculos para cumplir una condición de diseño dada, aquí otra vez se observa la necesidad de cálculos y si incorporamos la tecnología, entonces tenemos una herramienta poderosa que ayuda al ingeniero a realizar su trabajo.

Nos apoyamos en la ecuación de Niklaus Wirth¹ para plantear la solución de cualquier problema:

$$\text{Solución} = \text{métodos numéricos (algoritmos)} + \text{lenguaje de programación (programas)}$$

Se requiere que el ingeniero conozca los dos términos de la ecuación anterior. Entendemos, por lo tanto, que la solución se desprende de un algoritmo a prueba de fallas y de un programa en un lenguaje de programación que obtenga el resultado rápido y sin errores.

También está claro que el algoritmo deberá entenderse muy bien y que el programa no debe tener errores de lógica ni de sintaxis, para que todo salga como se espera.

1.2 Sistemas de numeración

Los números son una representación abstracta de elementos con los que interactuamos. El ser humano tiene esa capacidad de abstraer el mundo que lo rodea y representarlo con otras entidades para manejarlas fácilmente y los números son esas entidades.

¹Programas = Algoritmos + Estructuras de datos

Los sistemas de numeración constan de un conjunto de **símbolos** disponibles en el sistema (dígitos y letras), y las **reglas** que se aplican para construir cantidades grandes.

Definición 1.2.1 — Símbolos. Los símbolos o dígitos pueden ser 1, 2, 3, 4, 5, etc. y letras A, B, C, D.

Definición 1.2.2 — Reglas. Las reglas son básicamente la posición que guarda el dígito en la cantidad que se quiere representar, los dígitos que están a la derecha tienen una representación menor que los que están a la izquierda, ya que representan una cantidad mayor. Esto se aplica a las cantidades enteras.

La representación de una cantidad en un sistema de numeración cualquiera se construye con la siguiente fórmula:

$$d \times b^p$$

Donde d es cada dígito de la cantidad, b es la base numérica y p es la posición que guarda el dígito dentro de la cantidad, siendo 0 la primera posición de derecha a izquierda.

1.2.1 Sistema decimal

En un **Sistema decimal** la base es **10** y los dígitos válidos en una cantidad son {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Entonces la cantidad 328 en base 10 representa la cantidad

$$\begin{aligned} & 3 * 10^2 + 2 * 10^1 + 8 * 10^0 \\ & 3 * 100 + 2 * 10 + 8 * 1 \\ & 300 + 20 + 8 \\ & 328 \end{aligned}$$

El sistema decimal es el que usamos para representar cantidades y hacer cálculos. Fue adoptado por nosotros por la sencilla razón que tenemos 10 dedos y es práctico usarlos para contar. En esa época ya usábamos zapatos, de otra manera usaríamos el sistema 20.

1.2.2 Sistema binario

En un **sistema binario** la base es **2** y los dígitos válidos en una cantidad son {0, 1}. Esto quiere decir que sólo se pueden usar los dígitos 0 y 1, no es posible usar otros dígitos como 2, 3, 4, etc.

La posición de los dígitos se aplica de igual manera en el sistema binario, es decir que los dígitos se multiplican por la base elevada a la posición.

Por ejemplo, la cantidad 1101 en base 2 representa la cantidad

$$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

$$\begin{aligned}
 & 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 \\
 & 8 + 4 + 0 + 1 \\
 & 13
 \end{aligned}$$

1.2.3 Sistema octal

En un **sistema octal** la base es **8** y los dígitos válidos en una cantidad son $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Esto quiere decir que sólo se pueden usar los dígitos 0 al 7, no es posible usar otros dígitos como 8 o 9.

La posición de los dígitos se aplica de igual manera que en los sistemas anteriores, es decir, que los dígitos se multiplican por la base elevada a la posición.

Por ejemplo, la cantidad 32 en base 8 representa la cantidad

$$\begin{aligned}
 & 3 * 8^1 + 2 * 8^0 \\
 & 3 * 8 + 2 * 1 \\
 & 24 + 2 \\
 & 26
 \end{aligned}$$

1.2.4 Sistema hexadecimal

En un **sistema hexadecimal** su base es **16** y los dígitos válidos en una cantidad son $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.

En este caso, como ya no existen otros dígitos numéricos más que del 0 al 9, entonces los siguientes caracteres que se usan son los caracteres alfabéticos, de la **A** a la **F** para completar 16 caracteres en total.

La posición de los dígitos se aplica como sabemos, los dígitos se multiplican por la base elevada a la posición.

Por ejemplo, la cantidad 1A en base 16 representa la cantidad

$$\begin{aligned}
 & 1 * 16^1 + 10 * 16^0 \\
 & 1 * 16 + 10 * 1 \\
 & 16 + 10 \\
 & 26
 \end{aligned}$$

En este caso, como el carácter **A** no es posible multiplicarlo por la base, se toma su representación numérica 10, por lo tanto, se calcula como $10 * 16^0$ (16 es la base hexadecimal y 0 es la posición del dígito).

Así existen otros sistemas de numeración como el sistema ternario (base 3), el sistema cuaternario (base 4), sistema quinario (base 5), etc.

1.3 Números fraccionarios

Hasta ahora hemos revisado la representación numérica de números enteros, los números con fracciones tienen las mismas propiedades de los dígitos y la posición, sólo que en estos casos la posición es negativa. Empecemos revisando el sistema decimal.

1.3.1 Sistema decimal

Como sabemos, el **sistema decimal** tiene la base **10** y los dígitos son {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Sólo que la posición después del punto inicia con -1, luego sigue -2, etc.

La cantidad 0.537 en base 10 representa la cantidad

$$5 * 10^{-1} + 3 * 10^{-2} + 7 * 10^{-3}$$

$$5 * 0.1 + 3 * 0.01 + 7 * 0.001$$

$$0.5 + 0.03 + 0.007$$

$$0.537$$

1.3.2 Sistema binario

En el **sistema binario** también se representan cantidades con fracción y la posición de igual forma es negativa, se debe recordar que la base es **2**.

Por ejemplo, la cantidad 0.011 en base 2 representa la cantidad

$$0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3}$$

$$0 * 0.5 + 1 * 0.25 + 1 * 0.125$$

$$0 + 0.25 + 0.125$$

$$0.375$$

1.3.3 Sistema octal

En el **sistema octal** (base **8**), la posición de los dígitos se aplica de igual manera, es decir, los dígitos se multiplican por la base elevada a la posición negativa.

Por ejemplo, la cantidad 0.732 en base 8 representa la cantidad

$$7 * 8^{-1} + 3 * 8^{-2} + 2 * 8^{-3}$$

$$7 * 0.125 + 3 * 0.015625 + 2 * 0.001953125$$

$$0.875 + 0.046875 + 0.00390625$$

$$0.92578125$$

1.3.4 Sistema hexadecimal

En el **sistema hexadecimal** (base 16) los dígitos se multiplican por la base a la posición negativa. Hay que recordar que los caracteres A, B, C, D, E y F son las cantidades 10,11,12,13,14 y 15, respectivamente.

Por ejemplo, la cantidad 0.2B en base 16 representa la cantidad

$$2 * 16^{-1} + 11 * 16^{-2}$$

$$2 * 0.0625 + 11 * 0.00390625$$

$$0.125 + 0.04296875$$

$$0.16796875$$

1.4 Convertir de un sistema al sistema decimal

1.4.1 Números enteros

De acuerdo con la ecuación $d \times b^p$ convierte una cantidad de cualquier base a la base 10. Para hacer el proceso contrario entonces se debe dividir entre la base a la que se requiere convertir. Esta división debe ser una **división entera**.

Para hacer la conversión se hace en una columna donde se coloca el resultado de la división entera y en la siguiente columna se coloca el residuo de esa división entera.

■ **Ejemplo 1.1 — Convertir decimal a binario.** Convertir el número 26 que está en base 10 a base 2. Se toma la cantidad 26 y se hacen divisiones enteras entre la base a la que se quiere convertir, en este caso 2. Hasta que el resultado sea 0. Los dígitos de la columna del residuo son la cantidad convertida.

División entera	Residuo	Operación
26		Dividir 26 entre 2, resultado 13 residuo 0
13	0	Dividir 13 entre 2, resultado 6 residuo 1
6	1	Dividir 6 entre 2, resultado 3 residuo 0
3	0	Dividir 3 entre 2, resultado 1 residuo 1
1	1	Dividir 1 entre 2, resultado 0 residuo 1
0	1	Como 0 ya no es divisible entre 2, se detienen los cálculos

Cuadro 1.1. Convertir decimal a binario

La cantidad en binario es 11010

■ **Ejemplo 1.2 — Convertir decimal a octal.** Convertir el número 26 que está en base 10 a base 8. Se toma la cantidad 26 y se hacen divisiones enteras entre la base 8. Hasta que el resultado sea 0.

División Entera	Residuo	Operación
26		Dividir 26 entre 8, resultado 3 residuo 2
3	2	Dividir 3 entre 8, resultado 0 residuo 3
0	3	Como 0 ya no es divisible entre 8, se detienen los cálculos

Cuadro 1.2. Convertir decimal a octal

Los dígitos de la columna del residuo son la cantidad convertida.

La cantidad en octal es 32.

■ **Ejemplo 1.3 — Convertir decimal a hexadecimal.** Convertir el número 26 que está en base 10 a base 16. Se toma la cantidad 26 y se hacen divisiones enteras entre la base 16. Hasta que el resultado sea 0. Los dígitos de la columna del residuo son la cantidad convertida.

Div. entera	Residuo	Operación
26		Dividir 26 entre 16, resultado 1 residuo 10, como 10 no es válido en el sistema hexadecimal entonces se usa A
1	A	Dividir 1 entre 16, resultado 0 residuo 1
0	1	Como 0 ya no es divisible entre 16, se detienen los cálculos

Cuadro 1.3. Convertir decimal a hexadecimal

La cantidad en hexadecimal es 1A.

Programa 1.1. Convertir del sistema decimal a otros sistemas

```

1 numero=26
2 print(numero,' en base 10 es ',numero)
3 print(numero,' en base 2 es ',bin(numero))
4 print(numero,' en base 8 es ',oct(numero))
5 print(numero,' en base 16 es ',hex(numero))

```

```

26 en base 10 es 26
26 en base 2 es 0b11010
26 en base 8 es 0o32
26 en base 16 es 0x1a

```

1.4.2 Números con fracciones

De acuerdo con la ecuación $d \times b^{-p}$ convierte una cantidad de cualquier base a la base 10. Multiplicar por b^{-p} realmente es dividirlo entre la base. Para hacer el proceso contrario entonces se debe multiplicar por la base.

Para hacer la conversión se toma la parte fraccionaria y se multiplica por la base, del resultado se vuelve a tomar sólo la parte fraccionaria y se multiplica por la base, se repite el mismo procedimiento hasta que la parte fraccionaria es 0. Los dígitos que quedan en la parte entera de cada multiplicación son los dígitos de la cantidad en la nueva base.

■ **Ejemplo 1.4 — Convertir decimal a binario.** Convertir el número 0.1875 que está en base 10 a base 2. Se toma la parte fraccionaria 0.1875 y se multiplica por 2 (base a la que se quiere convertir), del resultado sólo se toma la parte fraccionaria y se multiplica por 2. Se repite el mismo procedimiento hasta que la parte fraccionaria sea 0. Los dígitos de la parte entera son la cantidad convertida.

Parte Fraccionaria	Resultado (parte entera)	Operación
0.1875	0	Multiplicar 0.1875 por 2, resultado 0.3750, Fracción 0.375
0.375	0	Multiplicar 0.375 por 2, resultado 0.750, Fracción 0.75
0.75	1	Multiplicar 0.75 por 2, resultado 1.5, Fracción 0.5
0.5	1	Multiplicar 0.5 por 2, resultado 1.0, Fracción 0
0	0	Aquí se detienen los cálculos ya que 0 por 2 es 0

Cuadro 1.4. Convertir decimal a binario

La cantidad en binario es 0.00110

■ **Ejemplo 1.5 — Convertir decimal a binario infinito.** Convertir el número 0.9 que está en base 10 a base 2.

La cantidad en binario es 0.1110011...

Parte Fraccionaria	Resultado	Operación
0.9	1.8	Multiplicar 0.9 por 2, resultado 1.8, Fracción 0.8
0.8	1.6	Multiplicar 0.8 por 2, resultado 1.6, Fracción 0.6
0.6	1.2	Multiplicar 0.6 por 2, resultado 1.2, Fracción 0.2
0.2	0.4	Multiplicar 0.2 por 2, resultado 0.4, Fracción 0.4
0.4	0.8	Multiplicar 0.4 por 2, resultado 0.8, Fracción 0.8
0.8	1.6	Multiplicar 0.8 por 2, resultado 1.6, Fracción 0.6
0.6	1.2	Multiplicar 0.6 por 2, resultado 1.2, Fracción 0.2
...	...	Se observa que se repiten los resultados

Cuadro 1.5. Convertir decimal a binario infinito

Observe que es ¡un número infinito de dígitos! Esto nos dice que un número exacto en una base numérica no necesariamente es exacto en otra base numérica. ■

1.5 Cómo se almacenan los números en la memoria de la computadora

Como sabemos, las computadoras trabajan con el sistema numérico **binario**. Por lo tanto, los números tendrán que ser convertidos a ese sistema para poder ser almacenados y procesados por una computadora.

El almacenamiento se hace sobre una localidad de memoria llamada **word**, la palabra de una computadora puede ser de 16, 32 o 64 bits². Para una palabra de 16 bits se entiende que cuenta con 16 espacios para almacenar los dígitos binarios de la cantidad. En el bit 0 se almacena el signo de la cantidad (0 = Positivo, 1 = Negativo).

De tal manera que si queremos saber cómo se almacena el número decimal 26 en una memoria de 16 bits, primero se convierte a binario 11010 (cuadro: 1.1), luego se coloca en las posiciones adecuadas.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0

En el Bit 0 se almacena 0 indicando que la cantidad es positiva, los dígitos se alinean a la izquierda llenando el resto de los espacios con 0.

Como es de esperarse, la cantidad más grande entera positiva que se puede almacenar en una palabra (word) de 16 bits es 01111111111111 que corresponde al número 32767.

1.5.1 Notación científica

En ciencias e ingeniería, se manejan cantidades muy grandes o pequeñas, para representarlas se utiliza la notación científica que se compone de la mantisa y la característica.

$$\text{mantisa} \times 10^{\text{característica}}$$

Definición 1.5.1 — Mantisa. La mantisa es la cantidad numérica expresada con el formato 9.999 indicando los dígitos más significativos.

Por ejemplo, para el número de Avogadro la mantisa es el número 6.022.

Definición 1.5.2 — Característica. La característica es la cantidad de posiciones que se desplaza el punto para que la mantisa quede expresada como se requiere.

Por ejemplo, para el número de Avogadro la característica es 23. Ambas cantidades se separan por la letra *e* o *E*. Entonces el número de Avogadro expresado en notación científica es:

²bit se deriva de las palabras **binary digit**

$$\text{Número de Avogadro} = 6.022\text{e}23$$

Para los números muy pequeños, como la carga del electrón, la característica es negativa, ya que el punto se desplaza a la izquierda.

$$\text{Carga del electrón} = -1.60\text{e}-19$$

Como el punto se puede desplazar a la derecha o izquierda, a estos números también se les llama *de coma flotante*. El formato de la mantisa 9.99 está en la forma normal, aunque la mantisa también podría expresarse como 99.9 ó 0.999, esto cambiaría el valor de la característica aumentando o disminuyendo en 1. Estas otras formas se llaman *formas subnormales*.

■ **Ejemplo 1.6 — Notación científica del número 123.** El número 123 se puede expresar de distintas maneras usando la notación científica.

0.123e3 Forma Subnormal

1.23e2 Forma Normal

12.3e1 Forma Subnormal

123e0 Forma Subnormal

1230e-1 Forma Subnormal

■ La primera forma subnormal 0.123e3 es la que utilizan las computadoras para almacenar los números de coma flotante. Claro que primero se convierten al sistema binario, tanto la mantisa como la característica y entonces se almacena en la palabra (word) de 16, 32 o 64 bits.

La palabra de 16 bits se divide en dos partes para poder almacenar los números de coma flotante, la primera parte es para almacenar la característica (primeros 7 dígitos) y la segunda parte para almacenar la mantisa (dígitos restantes). El signo de la mantisa se indica en el bit 0, el signo de la característica en el bit 1.

signo	característica							mantisa							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

■ **Ejemplo 1.7 — Almacenar el número -125.32.** Pasos a seguir:

1. Primero convertimos la parte entera 125 (el signo se indica en el bit 0 con el valor de 1).

1111101

2. Luego se convierte la parte fraccionaria 0.32.

0.010100011110101

3. Se juntan ambos números y se expresa en la forma subnormal. La característica también se convierte a binario para poder almacenarla en la memoria. Hay que recordar que la base del sistema numérico es 2.

$$-0.1111101010100011110101 \times 2^{111}$$

signo	característica							mantisa							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0

4. Se almacena en la palabra (*word*) de 16 bits usando los espacios para la característica y la mantisa.

En el ejemplo se observa que no es posible almacenar todos los dígitos en la palabra de 16 bits

0.11111010 10100011110101...

Para una computadora que tenga un sistema operativo de 32 o 64 bits, la mantisa es más grande para tratar de almacenar los más dígitos posibles y permitiendo mayor precisión.

Programa 1.2. Error de redondeo en Python

```
a=4.1
b=0.2
a-b
Out[1]: 3.8999999999999995
```

Se observa un error de redondeo al no ser capaz de almacenar el valor exacto de la resta.

1.6 Errores

Como se observó en la sección anterior, las cantidades expresadas en el sistema de numeración binaria no siempre se pueden expresar de una manera finita y, por otro lado, no se pueden almacenar completamente si es que el número de dígitos es más grande que el tamaño de la palabra de la computadora. Eliminar dígitos de una cantidad provoca un error de redondeo.

Definición 1.6.1 — Error. Un error se define simplemente como la diferencia entre el valor obtenido y el valor correcto.

Definición 1.6.2 — Error de redondeo. Los errores de redondeo se generan cuando se eliminan dígitos de una cantidad.

No confundir con la técnica de redondeo, el cual se utiliza para escribir una cantidad con algunos dígitos, por ejemplo 7.666 se redondea a 7.67 y 7.664 se redondea a 7.66. Redondeo a 2 dígitos.

Definición 1.6.3 — Error de truncamiento. Los errores de truncamiento se generan cuando se eliminan términos de una expresión, como por ejemplo la serie de Taylor que puede tener un número infinito de términos.

Dado que los métodos numéricos requieren usar tanto números como expresiones infinitas que representan a las ecuaciones a resolver, es inevitable caer en cualquiera de estos errores o en ambos. Se verá más adelante cómo reducir la afectación de estos errores en los cálculos.

Si en alguna expresión se involucran constantes como $\frac{1}{3}$ o el valor de π , no es posible usar todos los dígitos, si los manejamos de manera numérica, lo recomendable es manejárselos como expresiones y sólo hasta el momento de obtener la evaluación final, sustituir su valor numérico para evitar los errores de redondeo.

Los métodos numéricos requieren hacer muchos cálculos y si cada vez que se hace un cálculo se comete un error de redondeo, entonces al final se puede propagar el error y llevarnos a un resultado muy alejado de la realidad.

De la **definición 1.6.1** se encuentra que el error se calcula como:

Error	
	$E = \text{valor} - \text{real}$

(1.1)

Donde *valor* es el valor obtenido y *real* es el valor real. El signo del error depende si el valor es mayor o menor del real, en algunos casos lo que nos interesa es la magnitud entre los dos valores, para ello se calcula el error absoluto

Error absoluto	
	$EA = \text{valor} - \text{real} $

(1.2)

Entiéndase $||$ como el valor absoluto de.

Pensemos en un ejemplo sencillo. Supóngase que usted tiene una cuenta en el banco y de acuerdo con sus cargos y abonos usted debería tener \$100.00, pero el banco le reporta un saldo de \$90.00, entonces el banco tiene un error absoluto.

$$\text{error absoluto} = |90 - 100| = 10$$

El error absoluto es de \$10.00. ¿Esta cantidad puede ser alta o baja? Depende de la persona y su estado financiero. Pero suponga ahora que usted es un magnate que tiene millones y millones de pesos, entonces qué tan alto puede ser el error absoluto de \$10.00, supongo que contestará que es bajo, pero hace un momento consideró que era alto, ¡ah! Pero ahora es millonario y esa falta de \$10.00 no afecta para nada su economía. Entonces, ¿cómo considerar el error absoluto de 10? Debemos ponerlo en el contexto de lo relativo que es ese valor. El error relativo se divide entre el valor real para dimensionar el error de acuerdo con el contexto y entonces tenemos el error relativo.

Error relativo	
	$ER = \frac{ \text{valor} - \text{real} }{\text{real}}$

(1.3)

Ahora el error relativo para el mismo problema, considerando que nuestro saldo debería ser \$1,000,000.00

$$\text{errorRelativo} = \frac{|999,990 - 1,000,000|}{1,000,000} = 0.00001$$

El error relativo es de 0.00001, lo cual podría decirse que es un error pequeño. Expresemos ese error en porcentaje para acotarlo a una cantidad manejable.

Error relativo porcentual

$$ERP = \frac{|valor - real|}{real} * 100 \quad (1.4)$$

Para nuestro ejemplo el error relativo porcentual es

$$\text{errorRelativoPorcentual} = \frac{|999,990 - 1,000,000|}{1,000,000} * 100 = 0.001\%$$

Entonces el error relativo porcentual es de 0.001 % que nos da una idea más clara si el error es grande o pequeño.

■ **Ejemplo 1.8 — Error relativo porcentual.** Suponga que tiene el siguiente algoritmo de cálculo

$$a = 0.2145$$

$$b = 0.2144$$

$$c = 0.1000e5$$

$$x = (a - b)c$$

Con estos valores se obtiene $x = 1$, el cual consideramos como correcto. Sin embargo, supóngase que a fue calculada con un valor de 0.2146 (error absoluto 0.0001, error relativo de 0.00046 y ERP = 0.046 %). Usando este valor de a vuelva a efectuar el mismo algoritmo

$$a = 0.2146$$

$$b = 0.2144$$

$$c = 0.1000e5$$

$$x = (a - b)c$$

Se obtiene como respuesta $x = 2$. Un error del 0.046 % de pronto provoca un error del 100 %. Este error puede pasar desapercibido. ■

El error se hace grande porque el cálculo de $x = (a - b)c$ involucra cantidades pequeñas de a y b , pero como se multiplica por c , la diferencia pequeña se vuelve grande y provoca un error relativo porcentual del 100 %.

1.7 Serie de Taylor

Muchos métodos numéricos utilizan aproximaciones a las soluciones, estas aproximaciones se hacen con polinomios de distintos grados para que sean analíticos en un rango más amplio.

Las aproximaciones se hacen con polinomios ya que son más fáciles para derivar, integrar, encontrar sus raíces. En cambio, otras ecuaciones más complicadas tratados con la Matemática analítica no son tan dóciles.

La construcción del polinomio se basa en la serie de potencias de Newton

$$f(x) = C_0 + C_1(x - x_0) + C_2(x - x_0)^2 + C_3(x - x_0)^3 + C_4(x - x_0)^4 + \dots \quad (1.5)$$

Donde las C_i son constantes a determinar, x_0 es un punto inicial y x es el punto donde se desea evaluar la función. Observamos que se trata de una serie infinita.

Para determinar las C_i , hacemos $x = x_0$, entonces tenemos

$$\begin{aligned} f(x_0) &= C_0 + C_1(x_0 - x_0) + C_2(x_0 - x_0)^2 + C_3(x_0 - x_0)^3 + C_4(x_0 - x_0)^4 + \dots \\ f(x_0) &= C_0 \end{aligned}$$

Ya conocemos el valor de C_0 . Si $f(x)$ es igual a la serie en el punto inicial, entonces la derivada $f'(x)$ es igual a la primera derivada de la serie evaluada en el punto inicial x_0

$$f'(x) = C_1 + 2C_2(x - x_0) + 3C_3(x - x_0)^2 + 4C_4(x - x_0)^3 + \dots$$

Evaluando $x = x_0$

$$\begin{aligned} f'(x_0) &= C_1 + 2C_2(x_0 - x_0) + 3C_3(x_0 - x_0)^2 + 4C_4(x_0 - x_0)^3 + \dots \\ f'(x_0) &= C_1 \end{aligned}$$

Conocemos entonces el valor de C_1 . Si $f'(x)$ es igual a la primera derivada de la serie en el punto inicial, entonces la segunda derivada $f''(x)$ es igual a la segunda derivada de la serie evaluada en el punto inicial x_0

$$f''(x) = 2C_2 + 3 * 2C_3(x - x_0) + 4 * 3C_4(x - x_0)^2 + \dots$$

Evaluando $x = x_0$

$$\begin{aligned} f''(x_0) &= 2C_2 + 3 * 2C_3(x_0 - x_0) + 4 * 3C_4(x_0 - x_0)^2 + \dots \\ f''(x_0) &= 2C_2 \\ C_2 &= \frac{f''(x_0)}{2} \end{aligned}$$

Proseguimos de igual manera para la tercera derivada de $f'''(x)$ es igual a la tercera derivada de la serie en el punto inicial.

$$f'''(x) = 3 * 2 * 1C_3 + 4 * 3 * 2C_4(x - x_0) + \dots$$

Evaluando $x = x_0$

$$\begin{aligned} f'''(x_0) &= 3 * 2 * 1C_3 + 4 * 3 * 2C_4(x_0 - x_0) + \dots \\ f'''(x_0) &= 3 * 2 * 1C_3 \\ C_3 &= \frac{f'''(x_0)}{3!} \end{aligned}$$

Nos damos cuenta de que los valores de C_i tienen la forma

$$C_n = \frac{f^{(n)}(x_0)}{n!}$$

Sustituyendo las C_i en la ecuación 1.5 tenemos finalmente la **serie de Taylor**.

Expansión de la serie de Taylor

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f^{iv}(x_0)}{4!}(x - x_0)^4 + \dots$$

(1.6)

$$f(x) = \sum_{i=0}^n \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i \quad (1.7)$$

La serie de Taylor también es una serie infinita ($n = \infty$) para funciones infinitamente derivables, tales como $\sin(x)$, $\cos(x)$, e^x , $\ln(x)$, etc.

Definición 1.7.1 — Serie de Taylor. Mediante la serie de Taylor se aproxima $f(x)$ con un polinomio a partir del punto inicial x_0 . Si la función es finitamente derivable, entonces se puede trabajar con la serie de Taylor porque tendrá un número finito de términos; pero si la función es infinitamente derivable, se tendrá que truncar los últimos términos para poder trabajar con la serie.

Definición 1.7.2 — Serie de MacLaurin. La serie de MacLaurin es la serie de Taylor con $x_0 = 0$.

Definición 1.7.3 — Serie Truncada de Taylor. La Serie Truncada de Taylor se compone de los primeros n términos de la serie que se consideran para hacer la aproximación a la función.

Definición 1.7.4 — Residuo de la Serie Truncada de Taylor. El residuo de la Serie Truncada de Taylor son los últimos términos de la serie a partir de la cual se trunca.

La serie de Taylor de orden cero sólo incluye el primer término

$$f(x) = f(x_0)$$

La serie de Taylor de primer orden incluye los dos primeros términos

$$f(x) = f(x_0) + f'(x_0)(x - x_0)$$

La serie de Taylor de segundo orden incluye los tres primeros términos

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2$$

Así se va adicionando un término más a la serie. Recodar que x_0 es el punto inicial y x es un valor cercano a x_0 .

■ **Ejemplo 1.9 — Obtener la serie truncada de MacLaurin de $\cos(x)$.** Obtener la serie truncada de MacLaurin de sexto orden de

$$f(x) = \cos(x)$$

Como la serie es de sexto orden entonces se toman los primeros 7 términos de la serie y se evalúan en $x_0 = 0$, por lo tanto, la expresión estará en términos de x .

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f^{iv}(x_0)}{4!}(x - x_0)^4 \\ &\quad + \frac{f^v(x_0)}{5!}(x - x_0)^5 + \frac{f^{vi}(x_0)}{6!}(x - x_0)^6 \end{aligned}$$

Obtenemos cada derivada de $f(x)$ y la evaluamos en $x_0 = 0$

$$f(x_0) = \cos(0) = 1 \quad \text{Primer término 1}$$

$$f'(x_0) = \sin(0) = 0 \quad \text{Segundo término } 0(x - 0)$$

$$f''(x_0) = -\cos(0) = -1 \quad \text{Tercer término } \frac{-1}{2}(x - 0)^2$$

$$f'''(x_0) = -\sin(0) = 0 \quad \text{Cuarto término } \frac{0}{3!}(x - 0)^3$$

$$f^{iv}(x_0) = \cos(0) = 1 \quad \text{Quinto término } \frac{1}{4!}(x - 0)^4$$

$$f^v(x_0) = \sin(0) = 0 \quad \text{Sexto término } \frac{0}{5!}(x - 0)^5$$

$$f^{vi}(x_0) = -\cos(0) = -1 \quad \text{Séptimo término } \frac{-1}{6!}(x - 0)^6$$

Sustituimos el valor en cada término en la serie truncada

$$\begin{aligned} f(x) &= 1 + 0x + \frac{-1}{2!}x^2 + \frac{0}{3!}x^3 + \frac{1}{4!}x^4 \\ &\quad + \frac{0}{5!}x^5 + \frac{-1}{6!}x^6 \end{aligned}$$

Finalmente, la serie truncada de MacLaurin de $f(x) = \cos(x)$ de orden 6 es

$$f(x) = 1 - \frac{1}{2}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6$$

Programa 1.3. Serie de MacLaurin de $\cos(x)$

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
```

```

4
5 x0=0
6 n=7
7
8 x=sp.Symbol('x')
9 t=sp.cos(x).series(x,x0,n).remove0()
10 print(t)
11 t=sp.lambdify(x,t,'numpy')
12
13 #g=plt.figure()
14
15 xx=np.linspace(x0-5,x0+5,1000)
16 yc=np.cos(xx)
17 yt=t(xx)
18
19 plt.plot(xx,yc,xx,yt)
20 plt.legend(['cos(x)', 'Serie de Taylor orden 7'])
21 plt.title('Serie de Taylor de $f(x)=\cos(x)$')
22 plt.grid(True)
23
24 #g.savefig('seriet.pdf', bbox_inches='tight')

```

$-\frac{x^{12}}{720} + \frac{x^8}{48} - \frac{x^4}{2} + 1$

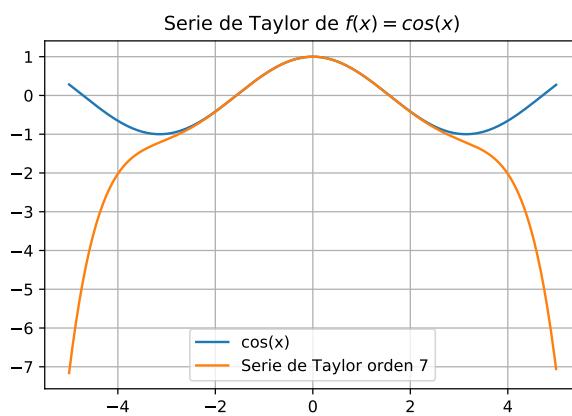


Figura 1.1. Gráfica de la Serie de MacLaurin de $\cos(x)$

Se observa que la serie y la función son idénticas cercanas al punto origen $x_0 = 0$, esto quiere decir que **la serie es analítica** alrededor de ese punto en el rango $[x_0 - r, x_0 + r]$, si x pertenece a ese rango, la aproximación es mejor que si nos alejamos, por ejemplo, en $x = 4$ la diferencia es notable.

La diferencia que se observa se debe al error de truncamiento que se genera al eliminar términos de la serie. La magnitud del error se puede calcular con el residuo de la serie.

Entonces tenemos dos formas para hacer que la serie sea analítica a la función en un rango mayor. Una es aumentar el número de términos en la serie, lo cual es complicado porque se debe obtener la derivada de $f(x)$ y no siempre es fácil de obtener. La segunda forma es que el punto x a evaluar no se aleje mucho del origen x_0 . Esta segunda forma es lo que hacen la mayoría de los métodos numéricos, por lo tanto, se requiere hacer incrementos de x lo suficientemente pequeños para no afectar el resultado.

■ **Ejemplo 1.10 — Obtener la serie truncada de Taylor de $\ln(x)$.** Obtener la serie truncada de Taylor de $f(x) = \ln(x)$ de cuarto orden con $x_0 = 1$.

Como la serie es de cuarto orden, entonces se toman los primeros 5 términos de la serie y se evalúan en $x_0 = 1$, por lo tanto, la expresión estará en términos de x .

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f^{iv}(x_0)}{4!}(x - x_0)^4$$

Obtenemos cada derivada de $f(x)$ y la evaluamos en $x_0 = 1$

$$f(x_0) = \ln(x_0) = 0 \quad \text{Primer término 0}$$

$$f'(x_0) = \frac{1}{x_0} = \frac{1}{1} \quad \text{Segundo término } 1(x - 1)$$

$$f''(x_0) = -\frac{1}{x_0^2} = -1 \quad \text{Tercer término } -\frac{1}{2}(x - 1)^2$$

$$f'''(x_0) = \frac{2}{x_0^3} = 2 \quad \text{Cuarto término } \frac{2}{3!}(x - 1)^3$$

$$f^{iv}(x_0) = -\frac{6}{x_0^4} = 6 \quad \text{Quinto término } -\frac{6}{4!}(x - 1)^4$$

Sustituimos el valor en cada término en la serie truncada

$$f(x) = 0 + (x - 1) + \frac{-1}{2}(x - 1)^2 + \frac{3}{3!}(x - 1)^3 + \frac{6}{4!}(x - 1)^4$$

Finalmente, la serie truncada de Taylor de $f(x) = \ln(x)$ de orden 4 es

$$f(x) = (x - 1) - \frac{1}{2}(x - 1)^2 + \frac{1}{3}(x - 1)^3 - \frac{1}{4}(x - 1)^4$$

Programa 1.4. Serie de Taylor de $\ln(x)$

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 x0=1
6 n=5
7
8 x=sp.Symbol('x')
9 t=sp.log(x).series(x,x0,n).remove0()
10 print(t)

```

```

11 t=sp.lambdify(x,t,'numpy')
12
13 g=plt.figure()
14
15 xx=np.linspace(x0-0.9,x0+2,100)
16 yc=np.log(xx)
17 yt=t(xx)
18
19 plt.plot(xx,yc,xx,yt)
20 plt.legend(['ln(x)', 'Serie de Taylor orden 4'])
21 plt.title('Serie de Taylor de $f(x)=\ln(x)$')
22 plt.grid(True)
23
24 g.savefig('serieln.pdf', bbox_inches='tight')

```

```
x - (x - 1)**4/4 + (x - 1)**3/3 - (x - 1)**2/2 - 1
```

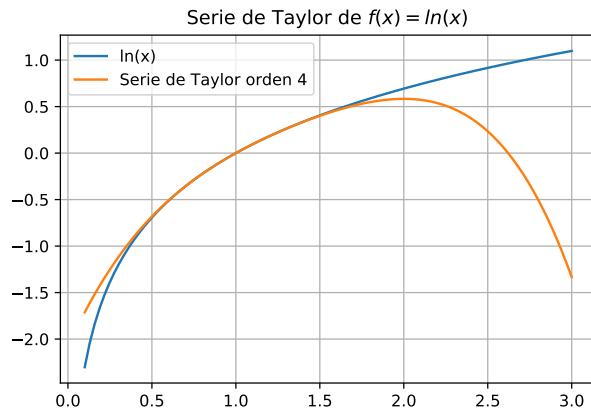


Figura 1.2. Gráfica de la Serie de Taylor de $\ln(x)$

■ **Ejemplo 1.11 — Obtener la serie truncada de MacLaurin de $f(x) = e^x$.**

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f^{iv}(x_0)}{4!}(x - x_0)^4 + \dots$$

Obtenemos cada derivada de $f(x)$ y la evaluamos en $x_0 = 0$

$$f(x_0) = e^{x_0} - 1 \quad \text{Primer término 1}$$

$$f'(x_0) = e^{x_0} - 1 \quad \text{Segundo término } 1(x-0)$$

$$f''(x_0) = e^{x_0} - 1 \quad \text{Tercer término } \frac{1}{2!}(x-0)^2$$

$$f'''(x_0) = e^{x_0} - 1 \quad \text{Cuarto término } \frac{1}{3!}(x-0)^3$$

$$f^n(x_0) = e^{x_0} - 1 \quad \text{Todos los términos restantes } \frac{1}{n!}(x-0)^n$$

Sustituimos el valor en cada término en la serie

$$f(x) = 1 + 1(x-0) + \frac{1}{2!}(x-0)^2 + \frac{1}{3!}(x-0)^3 + \frac{1}{4!}(x-0)^4 + \dots$$

Finalmente, la serie truncada de MacLaurin de $f(x) = e^x$ es

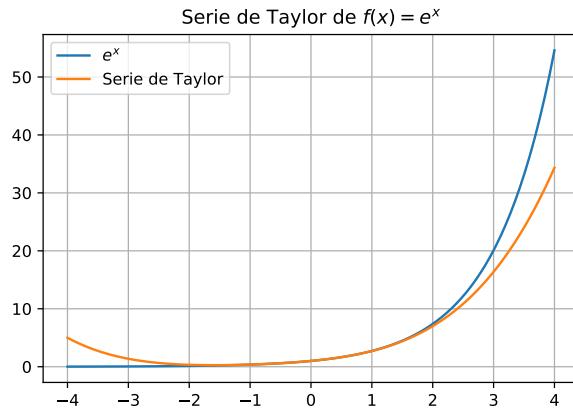
$$f(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Programa 1.5. Serie de MacLaurin de e^x

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 x0=0
6 n=5
7
8 x=sp.Symbol('x')
9 t=sp.exp(x).series(x,x0,n).remove0()
10 print(t)
11 t=sp.lambdify(x,t,'numpy')
12
13 g=plt.figure()
14
15 xx=np.linspace(x0-4,x0+4,100)
16 yc=np.exp(xx)
17 yt=t(xx)
18
19 plt.plot(xx,yc,xx,yt)
20 plt.legend(['$e^x$', 'Serie de Taylor'])
21 plt.title('Serie de Taylor de $f(x)=e^x$')
22 plt.grid(True)
23
24 g.savefig('serieex.pdf', bbox_inches='tight')
```

```
x**4/24 + x**3/6 + x**2/2 + x + 1
```

Figura 1.3. Gráfica de la Serie de MacLaurin de e^x

■ **Ejemplo 1.12 — Obtener la serie de MacLaurin de $f(x) = e^{ix}$.**

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f^{iv}(x_0)}{4!}(x - x_0)^4 + \dots$$

Obtenemos cada derivada de $f(x)$ y la evaluamos en $x_0 = 0$

$$f(x_0) = e^{ix_0} = 1 \quad \text{Primer término } 1$$

$$f'(x_0) = ie^{ix_0} = i \quad \text{Segundo término } ix$$

$$f''(x_0) = i^2 e^{ix_0} = i^2 \quad \text{Tercer término } \frac{i^2}{2!}(x - 0)^2$$

$$f'''(x_0) = i^3 e^{ix_0} = i^3 \quad \text{Cuarto término } \frac{i^3}{3!}(x - 0)^3$$

$$f^n(x_0) = i^n e^{ix_0} = i^n \quad \text{Todos los términos restantes } \frac{i^n}{n!}(x - 0)^n$$

Sustituimos el valor en cada término en la serie

$$f(x) = 1 + ix + \frac{i^2}{2!}x^2 + \frac{i^3}{3!}x^3 + \frac{i^4}{4!}x^4 + \frac{i^5}{5!}x^5 + \dots$$

Sabemos que $i^2 = -1$ y que $i^4 = 1$ y así sucesivamente, entonces

$$f(x) = 1 + ix - \frac{x^2}{2!} - \frac{ix^3}{3!} + \frac{x^4}{4!} + \frac{ix^5}{5!} + \dots$$

Agrupamos los términos de la siguiente manera

$$f(x) = \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots\right) + i \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots\right)$$

Observamos que cada término es una serie conocida, el primer término es $\cos(x)$ y el segundo es $\sin(x)$ multiplicado por i . Así que:

$$f(x) = \cos(x) + i\sin(x)$$

Si sustituimos $x = \pi$ entonces tenemos

$$e^{i\pi} = \cos(\pi) + i\sin(\pi)$$

$$e^{i\pi} = -1$$

$$e^{i\pi} + 1 = 0$$

Esta ecuación es la famosa **Fórmula de Euler**.

Programa 1.6. Serie de MacLaurin de e^{ix}

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 x0=0
6 n=8
7
8 x=sp.Symbol('x')
9 t=sp.series(sp.exp(1j*x),x,x0,n).remove0()
10 #print(t)
11 t=sp.lambdify(x,t,'numpy')
12
13 yt=t(np.pi)
14
15 print(yt)
16 xx=np.linspace(x0-4,x0+4,100)
17 yc=np.exp(1j*xx)
18 yt=t(xx)
19
20 g=plt.figure()
21 plt.plot(yc.real,yc.imag,yt.real,yt.imag)
22
23 plt.legend(['$e^{\{ix\}}$', 'Serie de Taylor'])
24 plt.title('Serie de Taylor de $f(x)=e^{\{ix\}}$')
25 plt.grid(True)
26 g.savefig('serieeix.pdf', bbox_inches='tight')
```

(-0.999999999999999 -8.947542198854151e-15j)

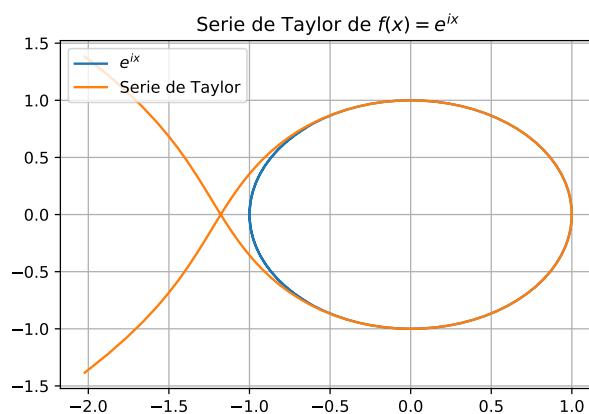
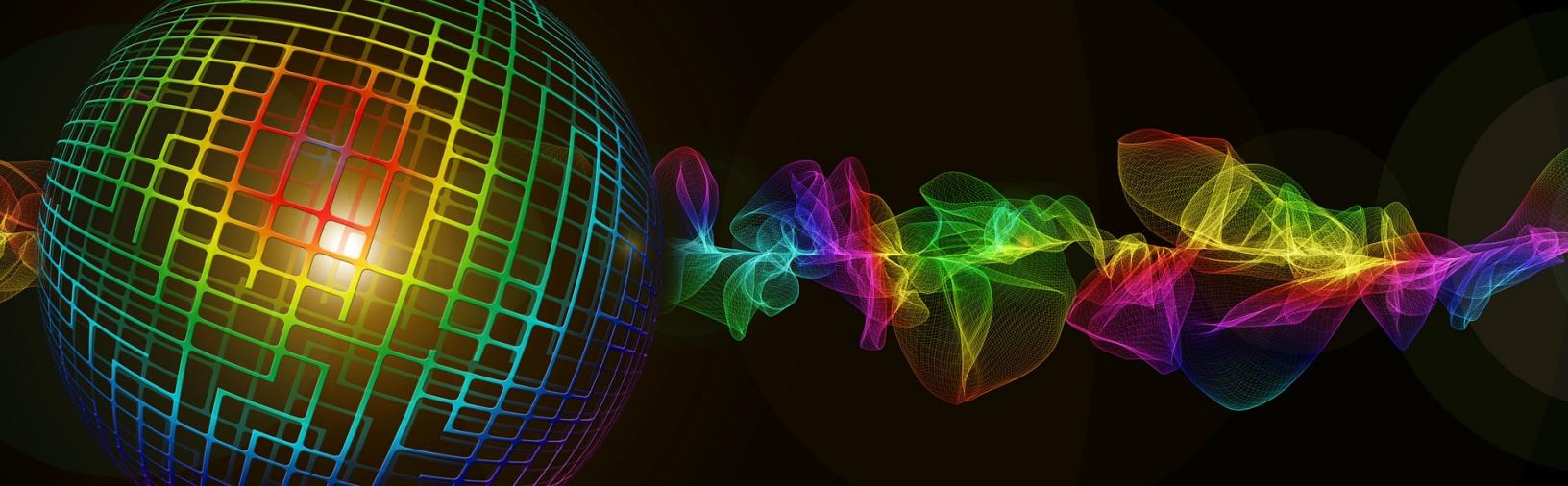


Figura 1.4. Gráfica de la Serie de MacLaurin de e^{ix}



2. Ajuste de curvas

El resultado de observaciones de un proceso de laboratorio o industrial genera una lista de datos que forman tablas.

x	x_0	x_1	x_2	\dots	x_n
y	y_0	y_1	y_2	\dots	y_n

Por ejemplo, la medición de la densidad del carbonato de sodio a distintas concentraciones a temperatura constante genera los siguientes datos.

%Concentración	4.1	12.2	20.3	28.2	38.1	45.2
Densidad	1.0276	1.1013	1.1801	1.2652	1.3480	1.4120

Los datos expresan la relación de la densidad (variable dependiente) y la concentración (variable independiente). Es decir que el valor de y depende del valor de x . Si deseamos conocer el valor de la variable dependiente para un valor de la variable independiente que no está en la tabla de datos, se requiere hacer una estimación.

La estimación puede lograrse de dos maneras, ajustando una función que toque todos los puntos de la tabla (interpolación) o ajustando una función que minimice las distancias del punto tabulado a la función (regresión).

La función de la interpolación pasa por cada punto de la tabla, la función de la regresión no necesariamente pasa lo más cerca de los puntos. Esto es, si sustituimos x_i en $f(x)$, el resultado es y_i para la interpolación y para la regresión obtenemos y_i corregida.

Programa 2.1. Interpolación lineal vs regresión lineal

```
1 import numpy as np
2 from scipy import interpolate
```

```

3 from scipy.optimize import curve_fit
4 import matplotlib.pyplot as plt
5
6 x=np.array([4.1,12.2,20.3,28.2,38.1,45.2])
7 y=np.array([1.0276,1.1013,1.1801,1.2652,1.3480,1.4120])
8
9 xfino=np.linspace(4.1,45.2,100)
10 yi=interpolate.interp1d(x,y,kind='linear')
11
12 def f(x,a,b):
13     return a+b*x
14
15 popt,=curve_fit(f,x,y)
16 yr=f(x,*popt)
17
18 plt.plot(x,y,'o',xfino,yi(xfino),'-',x,yr,'--')
19 plt.legend(['Datos','Interpolación lineal','Regresión lineal'])
20 plt.title('Interpolación lineal vs regresión lineal')
21 plt.grid(True)

```

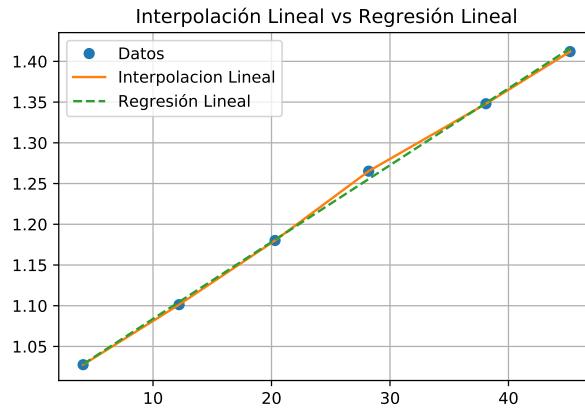


Figura 2.1. Interpolación lineal vs regresión lineal

2.1 Interpolación

El objetivo es ajustar una función que por combinaciones lineales logre pasar por todos los puntos.

Definición 2.1.1 — Interpolación. Las técnicas de interpolación ajustan un polinomio de menor grado que pasa por los puntos tabulados, esto quiere decir que, si sustituimos un valor de x_i de la tabla, obtenemos la correspondiente y_i de la tabla.

En general, la función tiene la forma:

$$f(x) = a_0g_0(x) + a_1g_1(x) + a_2g_2(x) + a_3g_3(x) + \dots + a_ng_n(x) \quad (2.1)$$

Donde las a_i son factores a determinar y las $g_i(x)$ son funciones de una familia en particular.

- Familia de monomios. Son los más utilizados ya que son fáciles de obtener y generan un polinomio. La desventaja es que los polinomios tienen comportamientos diferentes a los esperados, de acuerdo con los datos, cuando son de potencias altas:

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

$$f(x) = \sum_{i=0}^n a_i x^i$$

- Familia de funciones de Fourier: 1, $\sin(x)$, $\cos(x)$, $\sin(2x)$ que al combinarse linealmente como en la ecuación 2.1 tenemos:

$$f(x) = a_0 + \sum_{i=1}^n a_i \cos(ix) + b_i \sin(ix)$$

- Familia de funciones exponenciales: 1, e^x , e^{2x} , e^{3x} que al llevarla a la forma general de la ecuación 2.1 se tiene:

$$f(x) = a_0 + a_1 e^x + a_2 e^{2x} + \dots$$

$$f(x) = \sum_{i=0}^n a_i e^{ix}$$

La técnica de ajustar un polinomio a los datos es la más usada porque se adapta a muchos casos simples, las otras técnicas se usan para un conjunto de datos que tienen un comportamiento cíclico o exponencial.

2.1.1 Interpolación lineal

La interpolación lineal ajusta un polinomio de grado 1 a los datos tabulados. Usamos entonces la forma de una ecuación lineal

$$f(x) = a + bx \quad (2.2)$$

Para lograr una interpolación lineal (polinomio de grado 1), se requieren dos puntos de la tabla

x	x_0	x_1
y	y_0	y_1

Como indica la **definición 2.1.1** se debe encontrar un polinomio que pase por los puntos tabulados. Entonces, si sustituimos x_0 en $f(x)$ (Ecuación lineal 2.2) debemos obtener y_0 .

$$y_0 = a + bx_0 \quad (2.3)$$

Y, si sustituimos x_1 en la ecuación 2.2, entonces deberíamos obtener y_1

$$y_1 = a + bx_1 \quad (2.4)$$

Tenemos dos ecuaciones (2.3 y 2.4) y dos incógnitas (a y b), entonces procedemos a resolver el sistema de 2x2.

Despejamos a de ecuación 2.3

$$a = y_0 - bx_0$$

Sustituimos a en y_1

$$y_1 = y_0 - bx_0 + bx_1$$

$$y_1 = y_0 + b(x_1 - x_0)$$

$$b = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

Sustituimos a en ecuación 2.2

$$f(x) = (y_0 - bx_0) + bx$$

$$f(x) = y_0 + b(x - x_0)$$

Finalmente sustituimos b en la ecuación anterior

$$f(x) = y_0 + \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0) \quad (2.5)$$

La ecuación 2.5 se usa para hacer la interpolación lineal en el intervalo $x \in [x_0, x_1]$, x es el valor que se desea interpolar. En general, para un par de puntos $[x_i, y_i], [x_{i+1}, y_{i+1}]$ la ecuación sería

Interpolación lineal

$$f(x) = y_i + \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}(x - x_i), x \in [x_i, x_{i+1}] \quad (2.6)$$

■ **Ejemplo 2.1 — Interpolación lineal carbonato de sodio.** La medición de la densidad del carbonato de sodio a distintas concentraciones a temperatura constante genera los siguientes datos.

%Concentración	4.1	12.2	20.3	28.2	38.1	45.2
Densidad	1.0276	1.1013	1.1801	1.2652	1.3480	1.4120

Obtener la densidad del carbonato de sodio a una concentración del 15 %.

Observamos que la densidad del carbonato de sodio para una concentración del 15 % no existe en los datos tabulados, también observamos que los datos más cercanos son

%Concentración	x_i 12.2	x_{i+1} 20.3
Densidad	1.1013	1.1801

y_i	y_{i+1}
-------	-----------

Sustituimos los valores correspondientes en la ecuación 2.6 donde $x = 15$ (dato a interpolar)

$$f(x) = 1.1013 + \frac{(1.1801 - 1.1013)}{(20.3 - 12.2)}(x - 12.2)$$

Esta ecuación es válida solo para $x \in [12.2, 20.3]$ fuera de ese intervalo no estaríamos hablando de una interpolación. Sustituimos entonces el valor de $x = 15$

$$f(x) = 1.1013 + \frac{(1.1801 - 1.1013)}{(20.3 - 12.2)}(15 - 12.2) = 1.128539$$

Por lo tanto, de acuerdo con la interpolación lineal, la densidad del carbonato de sodio al 15% de concentración es 1.128539.

Programa 2.2. Interpolación lineal carbonato de sodio

```

1 import numpy as np
2 from scipy import interpolate
3 import matplotlib.pyplot as plt
4
5 x=np.array([4.1,12.2,20.3,28.2,38.1,45.2])
6 y=np.array([1.0276,1.1013,1.1801,1.2652,1.3480,1.4120])
7
8
9 xfino=np.linspace(4.1,45.2,100)
10 yi=interpolate.interp1d(x,y,kind='linear')
11
12 xi=15
13 yii=yi(xi)
14
15 plt.plot(x,y,'o',xfino,yi(xfino),'-',xi,yii,'sr')
16 plt.legend(['Datos','Interpolación lineal','Dato interpolado'])
17 plt.title('Interpolación lineal carbonato de sodio')
18 plt.xlabel('Concentración')
19 plt.ylabel('Densidad')
20 plt.text(xi, yii, '%C ' + str(xi) + ' Densidad ' + str(yii))
21
22 plt.grid(True)
23 plt.show()

```

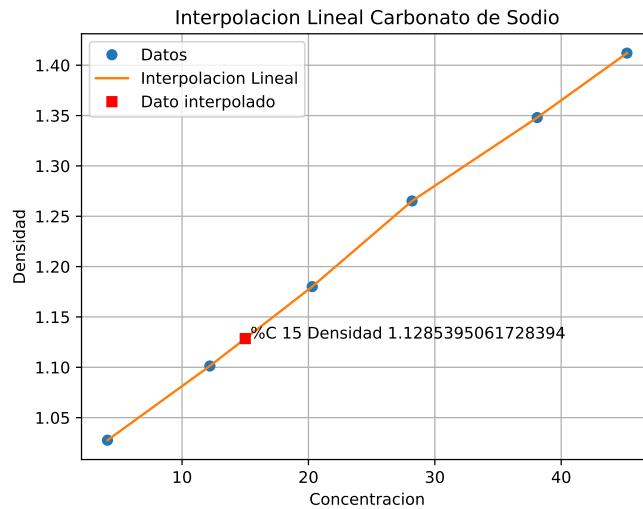


Figura 2.2. Interpolación lineal carbonato de sodio

Observamos que el dato intercalado forma parte de la línea recta que une los puntos $[12.2, 1.1013]$ y $[20.3, 1.1801]$ y el dato intercalado $15 \in [12.2, 20.3]$

■

En una interpolación lineal $y_i \leq y \leq y_{i+1}$.

2.1.2 Interpolación cuadrática

La interpolación lineal supone un comportamiento lineal entre y_i y y_{i+1} y esto no es necesariamente así, para los casos donde suponemos un comportamiento cuadrático se debe ajustar un polinomio de segundo grado que pase por los puntos. Si tomamos sólo dos puntos, entre ellos puede pasar un número infinito de polinomios de grado 2. La interpolación exige que el polinomio que une los puntos sea único y de menor grado, entonces se requieren 3 puntos para cumplir esta restricción.

x	x_0	x_1	x_2
y	y_0	y_1	y_2

Similar a la interpolación lineal donde ajustamos un polinomio de primer grado, ahora ajustamos un polinomio de segundo grado a los puntos tabulados, el polinomio debe pasar por los tres puntos, por lo que el polinomio es único.

$$f(x) = a + bx + cx^2 \quad (2.7)$$

Si sustituimos x_i en la ecuación 2.7 se debe obtener y_i

$$y_0 = a + bx_0 + cx_0^2$$

$$y_1 = a + bx_1 + cx_1^2$$

$$y_2 = a + bx_2 + cx_2^2$$

Tenemos un sistema de tres ecuaciones con tres incógnitas (a, b, c). La forma de la ecuación 2.7 nos lleva a resolver un sistema de 3x3, si pensamos en una interpolación cúbica tendríamos que resolver un sistema de 4x4, y si pensamos en una interpolación de grado n , entonces tendríamos que resolver un sistema $(n+1) \times (n+1)$, lo cual no es conveniente. Trabajemos con una ecuación similar a la ecuación 2.7 que tome en cuenta pasar por los tres puntos $(x_0, y_0), (x_1, y_1), (x_2, y_2)$

$$f(x) = a + b(x - x_0) + c(x - x_0)(x - x_1) \quad (2.8)$$

Debemos lograr que la ecuación 2.8 pase por los tres puntos, entonces si sustituimos x_0 debemos obtener y_0 .

$$y_0 = a + b(x_0 - x_0) + c(x_0 - x_0)(x_0 - x_1)$$

$$a = y_0$$

Si sustituimos x_1 debemos obtener y_1 , sabemos ya que $a = y_0$

$$y_1 = y_0 + b(x_1 - x_0) + c(x_1 - x_0)(x_1 - x_1)$$

$$b = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

Ya sabíamos los valores de estos términos a y b por la interpolación lineal, ahora sustituimos x_2 para obtener y_2 , sustituyendo también los valores obtenidos de a y b .

$$y_2 = y_0 + \frac{(y_1 - y_0)}{(x_1 - x_0)}(x_2 - x_0) + c(x_2 - x_0)(x_2 - x_1)$$

Despejando c ya tenemos los tres coeficientes que necesitamos para la ecuación 2.8, pero lo vamos a hacer de tal manera que los términos queden expresados al final como diferencias divididas.

Despejamos el último término donde se encuentra c

$$c(x_2 - x_0)(x_2 - x_1) = y_2 - y_0 - \frac{(y_1 - y_0)}{(x_1 - x_0)}(x_2 - x_0)$$

Sumamos un cero del lado derecho $-y_1 + y_1$

$$c(x_2 - x_0)(x_2 - x_1) = y_2 - y_1 + y_1 - y_0 - \frac{(y_1 - y_0)}{(x_1 - x_0)}(x_2 - x_0)$$

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} + \frac{(y_1 - y_0)}{(x_2 - x_1)} - \frac{(y_1 - y_0)}{(x_1 - x_0)} \frac{(x_2 - x_0)}{(x_2 - x_1)}$$

factorizamos $\frac{(y_1 - y_0)}{(x_2 - x_1)}$ en el segundo y tercer término

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} - \left(\frac{(y_1 - y_0)}{(x_2 - x_1)} \right) \left(\frac{(x_2 - x_0)}{(x_1 - x_0)} - 1 \right)$$

El término 1 lo sustituimos por otro $1 \frac{x_1 - x_0}{x_1 - x_0}$

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} - \left(\frac{(y_1 - y_0)}{(x_2 - x_1)} \right) \left(\frac{(x_2 - x_0)}{(x_1 - x_0)} - \frac{x_1 - x_0}{x_1 - x_0} \right)$$

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} - \left(\frac{(y_1 - y_0)}{(x_2 - x_1)} \right) \left(\frac{(x_2 - x_1 - x_0 + x_0)}{(x_1 - x_0)} \right)$$

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} - \left(\frac{(y_1 - y_0)}{(x_2 - x_1)} \right) \left(\frac{(x_2 - x_1)}{(x_1 - x_0)} \right)$$

$$c(x_2 - x_0) = \frac{(y_2 - y_1)}{(x_2 - x_1)} - \left(\frac{(y_1 - y_0)}{(x_1 - x_0)} \right)$$

$$c = \frac{\frac{(y_2 - y_1)}{(x_2 - x_1)} - \frac{(y_1 - y_0)}{(x_1 - x_0)}}{(x_2 - x_0)}$$

Por último, sustituimos a , b y c en la ecuación 2.8 para obtener la ecuación de la **interpolación cuadrática**.

Interpolación Cuadrática

$$f(x) = y_0 + \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0) + \frac{\frac{(y_2 - y_1)}{(x_2 - x_1)} - \frac{(y_1 - y_0)}{(x_1 - x_0)}}{(x_2 - x_0)}(x - x_0)(x - x_1), x \in [x_0, x_2] \quad (2.9)$$

Los dos primeros términos son iguales a la interpolación lineal, el tercer término es el término cuadrático que da la curvatura entre los puntos de la tabla. El segundo término expresa la pendiente del comportamiento lineal, que es la “derivada” expresada como una diferencia dividida, y el tercer término es una diferencia de diferencias, es decir, la segunda derivada. Es precisamente como está construida la serie de Taylor de la ecuación 1.7.

2.1.3 Polinomio de interpolación de Newton

Como observamos en la interpolación lineal (ecuación 2.2) y la interpolación cuadrática (ecuación 2.9) cada término que se va aumentando es una diferencia dividida de orden mayor. El polinomio de interpolación polinómica para n datos tiene la forma:

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots + a_n(x - x_0)(x - x_1)\dots(x - x_{n-1}) \quad (2.10)$$

Donde

$$a_0 = y_0$$

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0} = f[x_1, x_0]$$

$$a_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0} = f[x_2, x_1, x_0] \quad (2.11)$$

$$a_n = f[x_n, x_{n-1}, \dots, x_0]$$

sustituyendo las diferencias en la ecuación 2.10 obtenemos

Polinomio de interpolación de Newton

$$f(x) = y_0 + f[x_1, x_0](x - x_0) + f[x_2, x_1, x_0](x - x_0)(x - x_1) + \\ f[x_n, x_{n-1}, \dots, x_0](x - x_0)(x - x_1) \dots (x - x_{n-1}) \quad (2.12)$$

donde $f[x_1, x_0], f[x_2, x_1, x_0], f[x_n, x_{n-1}, \dots, x_0]$ son las diferencias divididas. Para calcular la segunda diferencia $\frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_1}$ se usa el valor de la primera diferencia $f[x_1, x_0]$ la cual fue calculada anteriormente, entonces para hacer un cálculo eficiente, necesitamos usar el resultado de la primera diferencia en la segunda diferencia para no volver a calcularlo. Se construye entonces la tabla de diferencias (Cuadro 2.1).

La tabla de diferencias contiene todos los valores para construir el polinomio de interpolación.

■ **Ejemplo 2.2 — Interpolación polinómica de Newton.** En temporada de lluvias se toma la profundidad del agua en una presa, obteniendo los siguientes datos:

día	1	4	8	13	18
profundidad	1.1	1.5	12.8	15.3	15.5

Calcular la profundidad del agua en el día 3.

Solución

Usemos la interpolación polinómica de Newton con todos los puntos de la tabla.

Programa 2.3. Interpolación polinómica de Newton

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x=np.array([1,4,8,13,18])
5 y=np.array([1.1,1.5,12.8,15.3,15.5])
6 n=x.size
7
8 t=np.zeros((n,n))
9 t[:,0]=y
10
11 #Genera la tabla de diferencias
12 for c in range(1,n):
13     for r in range(0,n-c):
14         t[r,c]=(t[r+1, c-1] - t[r, c-1]) / (x[r+c] - x[r])
15 print(t)
16
17 #calcula la interpolacion de xi
18 xi=3
19 xt = 1
20 yi = t[0, 0]

```

Cuadro 2.1. Tabla de diferencias

Ren	x	y	1er. Diferencia	2da. Diferencia	3er. Diferencia	4ta. Diferencia
1	x_0	y_0	$\frac{y_1 - y_0}{x_1 - x_0} = f[x_1, x_0]$	$\frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_1} = f[x_2, x_1, x_0]$	$\frac{f[x_3, x_2, x_1] - f[x_2, x_1, x_0]}{x_3 - x_0} = f[x_3, x_2, x_1, x_0]$	$\frac{f[x_4, x_3, x_2, x_1] - f[x_3, x_2, x_1, x_0]}{x_4 - x_0} = f[x_4, x_3, x_2, x_1, x_0]$
2	x_1	y_1	$\frac{y_2 - y_1}{x_2 - x_0} = f[x_2, x_1]$	$\frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} = f[x_3, x_2, x_1]$	$\frac{f[x_4, x_3, x_2] - f[x_3, x_2, x_1]}{x_4 - x_1} = f[x_4, x_3, x_2, x_1]$	
3	x_2	y_2	$\frac{y_3 - y_2}{x_3 - x_2} = f[x_3, x_2]$	$\frac{f[x_4, x_3] - f[x_3, x_2]}{x_4 - x_2} = f[x_4, x_3, x_2]$	$\frac{f[x_5, x_4, x_3] - f[x_4, x_3, x_2]}{x_5 - x_2} = f[x_5, x_4, x_3, x_2]$	
4	x_3	y_3	$\frac{y_4 - y_3}{x_4 - x_3} = f[x_4, x_3]$	$\frac{f[x_5, x_4] - f[x_4, x_3]}{x_5 - x_3} = f[x_5, x_4, x_3]$		
5	x_4	y_4	$\frac{y_5 - y_4}{x_5 - x_4} = f[x_5, x_4]$			
6	x_5	y_5				

```

21 for k in range(0,n-1):
22     xt = xt * (xi - x[k])
23     yi = yi + t[0, k + 1] * xt
24
25 print(yi)
26
27 plt.plot(x,y, 'o')
28 plt.plot(xi,yi, 'sr')
29 plt.text(xi+0.1,yi, ' Profundidad ' + str(yi))
30 plt.title('Profundidad del agua')
31 plt.legend(['Datos','Interpolación'])
32 plt.grid(True)
33 plt.show()

```

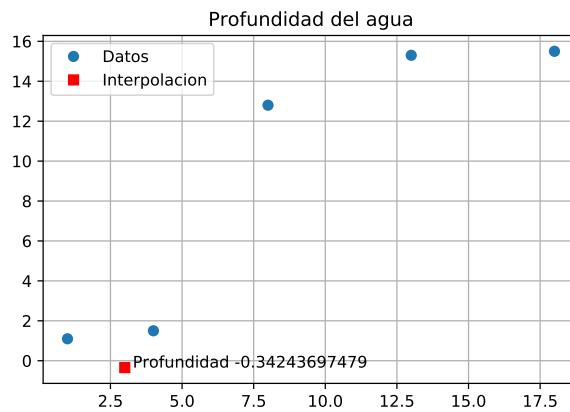


Figura 2.3. Interpolación polinómica de Newton

```

[[ 1.1000000e+00   1.3333333e-01   3.84523810e-01   -5.35714286e←
-02 4.04341737e-03]
 [ 1.5000000e+00   2.8250000e+00   -2.5833333e-01   1.51666667e←
-02 0.0000000e+00]
 [ 1.2800000e+01   5.0000000e-01   -4.6000000e-02   0.0000000e←
+00 0.0000000e+00]
 [ 1.5300000e+01   4.0000000e-02   0.0000000e+00   0.0000000e←
+00 0.0000000e+00]
 [ 1.5500000e+01   0.0000000e+00   0.0000000e+00   0.0000000e←
+00 0.0000000e+00]] -0.34243697479

```

La salida nos muestra el resultado de la interpolación, la profundidad es -0.3424, es obvio que no es el resultado esperado de acuerdo con el contexto del problema, la altura del agua no puede ser negativa; sin embargo, es el resultado de ajustar un polinomio con todos los datos de la tabla.

Si hacemos la interpolación lineal usando los mismos datos de la tabla de diferencias, como $\text{día}=3$, entonces $\text{día}\in[1,4]$, debemos usar la 1er diferencia para calcular la interpolación lineal con la ecuación 2.12.

$$f(x) = y_0 + f[x_1, x_0](x - x_0)$$

$$f(x) = 1.1 + 0.1333333(3 - 1) = 1.366666666$$

Que es un resultado de acuerdo con el contexto del problema. ■

Como observamos en el ejemplo anterior, no siempre es recomendable usar todos los datos de la tabla para la interpolación polinómica, en algunos casos puede dar resultados fuera de lo esperado, de acuerdo con el contexto del problema. El método no es incorrecto, el método se limita a ajustar un polinomio que pase por los puntos tabulados, el pasar por los puntos puede provocar oscilaciones entre los puntos si el polinomio es de un grado alto y puede generar resultados fuera del contexto. Es tarea del ingeniero seleccionar el método más adecuado al problema, como en este caso una interpolación lineal, finalmente, los métodos son herramientas que el ingeniero sabe cómo utilizarlas.

Utilizar un polinomio de grado alto con puntos equidistantes genera el problema de Runge que oscila en los extremos del polinomio.

2.1.4 Polinomio de Lagrange

La interpolación de Lagrange ajusta un polinomio a los datos, de tal manera que pasa por cada uno. Como lo precisan los métodos de interpolación, se trata de ajustar un polinomio de menor grado que toca todos los puntos y el polinomio es único. Como el polinomio es único, entonces tanto la interpolación de Newton como la interpolación de Lagrange obtienen el mismo polinomio, sólo que expresado de una manera diferente. Otra diferencia es que el polinomio de Lagrange obtiene el polinomio en un solo paso, la interpolación de Newton primero obtiene la tabla de diferencias para construir el polinomio de interpolación.

La forma del polinomio que usa la interpolación lineal de Lagrange entre dos puntos es la siguiente:

x	x_0	x_1
y	y_0	y_1

$$f(x) = a_0(x - x_1) + a_1(x - x_0) \quad (2.13)$$

Como en los casos anteriores, para asegurar que el polinomio pasa por los puntos tabulados, si sustituimos x_0 en la ecuación 2.13 debemos obtener y_0

$$y_0 = a_0(x_0 - x_1) + a_1(x_0 - x_0)$$

de aquí despejamos a_0

$$a_0 = \frac{y_0}{(x_0 - x_1)}$$

si sustituimos x_1 en la ecuación 2.13, debemos obtener y_1

$$y_1 = a_0(x_1 - x_1) + a_1(x_1 - x_0)$$

de aquí despejamos a_1

$$a_1 = \frac{y_1}{(x_1 - x_0)}$$

finalmente sustituimos a_0 y a_1 en la ecuación 2.13 para obtener

$$f(x) = \frac{(x - x_1)}{(x_0 - x_1)}y_0 + \frac{(x - x_0)}{(x_1 - x_0)}y_1 \quad (2.14)$$

En la interpolación cuadrática o de segundo grado se ajusta un polinomio de grado 2 que pasa por los tres puntos:

x	x_0	x_1	x_2
y	y_0	y_1	y_2

$$f(x) = a_0(x - x_1)(x - x_2) + a_1(x - x_0)(x - x_2) + a_2(x - x_0)(x - x_1) \quad (2.15)$$

Para asegurar que el polinomio pasa por los tres puntos tabulados, sustituimos x_0 en la ecuación 2.15, debemos obtener y_0

$$y_0 = a_0(x_0 - x_1)(x_0 - x_2) + a_1(x_0 - x_0)(x_0 - x_2) + a_2(x_0 - x_0)(x_0 - x_1)$$

de aquí despejamos a_0

$$a_0 = \frac{y_0}{(x_0 - x_1)(x_0 - x_2)}$$

si sustituimos x_1 en la ecuación 2.15, debemos obtener y_1

$$y_1 = a_0(x_1 - x_1)(x_1 - x_2) + a_1(x_1 - x_0)(x_1 - x_2) + a_2(x_1 - x_0)(x_1 - x_1)$$

de aquí despejamos a_1

$$a_1 = \frac{y_1}{(x_1 - x_0)(x_1 - x_2)}$$

Sustituimos x_2 en la ecuación 2.15, debemos obtener y_2

$$y_2 = a_0(x_2 - x_1)(x_2 - x_2) + a_1(x_2 - x_0)(x_2 - x_2) + a_2(x_2 - x_0)(x_2 - x_1)$$

de aquí despejamos a_2

$$a_2 = \frac{y_2}{(x_2 - x_0)(x_2 - x_1)}$$

finalmente, sustituimos a_0 , a_1 y a_2 en la ecuación 2.15 para obtener la ecuación de la interpolación de Lagrange de segundo grado.

$$f(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}y_2 \quad (2.16)$$

La construcción de la interpolación polinómica de Newton para n puntos, observando el comportamiento de la interpolación lineal y cuadrática anteriores (ecuaciones 2.13 y 2.15) podemos deducir que:

$$f(x) = L_0y_0 + L_1y_1 + L_2y_2 + \dots + L_ny_n \quad (2.17)$$

$$f(x) = \sum_{i=0}^{n-1} L_i y_i \quad (2.18)$$

Donde las L_i son los factores de Lagrange que se calculan como:

$$\prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{(x - x_j)}{(x_i - x_j)}$$

La ecuación de la interpolación polinómica de Lagrange es:

Interpolación polinómica de Lagrange

$$f(x) = \sum_{i=0}^{n-1} \left(\prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{(x - x_j)}{(x_i - x_j)} \right) y_i \quad (2.19)$$

Hay que recordar que, tanto la interpolación polinómica de Newton como la interpolación polinómica de Lagrange, ambas obtienen el mismo polinomio de interpolación, sólo que se expresan y se obtienen de maneras diferentes, pero algebraicamente son el mismo polinomio y, por lo tanto, se tienen los mismos resultados.

■ **Ejemplo 2.3 — Interpolación polinómica de Lagrange.** En temporada de lluvias se toma la profundidad del agua en una presa, obteniendo los siguientes datos:

día	1	4	8	15	18
profundidad	1.1	1.5	12.1	15.3	15.9

Calcular la profundidad del agua en el día 3.

Solución

Usemos la interpolación polinómica de Newton con todos los puntos de la tabla

Programa 2.4. Interpolación polinómica de Lagrange

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import interpolate
4
5 x=np.array([1,4,8,13,18])
6 y=np.array([1.1,1.5,12.8,15.3,15.5])
7 n=x.size
8 xi=3
9 yi=0
10 #Calcula los factores de Lagrange y hace la suma
11 for i in range(0,n):
12     producto = y[i]
13     for j in range(0,n):
14         if i != j:
15             producto = producto * (xi - x[j])/(x[i]-x[j]);
16     yi = yi + producto
17
18 print(yi)
19 f=interpolate.lagrange(x,y) #usando la funcion de Lagrange de ←
20     scipy
21 print(f(xi))
22
23 plt.plot(x,y,'o')
24 plt.plot(xi,yi,'sr')
25 plt.text(xi+0.1,yi, ' Profundidad ' + str(yi))
26 plt.title('Profundidad del agua')
27 plt.legend(['Datos','Interpolacion'])
28 plt.grid(True)
29 plt.show()

```

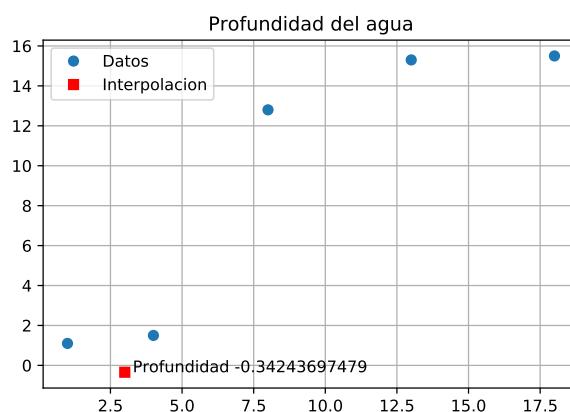


Figura 2.4. Interpolación polinómica de Lagrange

-0.34243697479

Era de esperarse que se obtuviera el mismo resultado con la interpolación de Newton, entonces debemos hacer la interpolación lineal de Lagrange, sólo se debe cambiar la línea 3 del código por n=2 y volver a correr el programa para obtener:

$$y_i = 1.3667$$

La cual es una estimación más real de acuerdo con el contexto del problema. ■

2.1.5 Algoritmo de Neville

El algoritmo de Neville se basa en la interpolación de Lagrange. Iniciemos con la interpolación lineal (ecuación 2.14)

$$F_{01}(x) = \frac{(x - x_1)}{(x_0 - x_1)} y_0 + \frac{(x - x_0)}{(x_1 - x_0)} y_1$$

La cual une los puntos

x	x_0	x_1
y	y_0	y_1

En una interpolación cuadrática entre los puntos

x	x_0	x_1	x_2
y	y_0	y_1	y_2

Se construyen dos interpolaciones lineales, una entre los puntos $[x_0, y_0], [x_1, y_1]$ que llamaremos F_{01} , y la otra entre los puntos $[x_1, y_1], [x_2, y_2]$ que llamaremos F_{12}

$$F_{01}(x) = \frac{(x - x_1)}{(x_0 - x_1)} y_0 + \frac{(x - x_0)}{(x_1 - x_0)} y_1$$

$$F_{12}(x) = \frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2$$

Ahora revisemos la interpolación cuadrática entre los tres puntos

$$F_{012}(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2$$

Por lo tanto, la interpolación cuadrática de Lagrange (ecuación 2.16) se expresa como

$$F_{012}(x) = \frac{(x - x_2)}{(x_0 - x_2)} \left(\frac{(x - x_1)}{(x_0 - x_1)} y_0 + \frac{(x - x_0)}{(x_1 - x_0)} y_1 \right) + \frac{(x - x_0)}{(x_2 - x_0)} \left(\frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2 \right)$$

$$F_{012}(x) = \frac{(x-x_2)}{(x_0-x_2)} F_{01}(x) + \frac{(x-x_0)}{(x_2-x_0)} F_{12}(x)$$

Para construir este algoritmo es conveniente usar un algoritmo recursivo, donde para obtener la interpolación de un polinomio de grado k , es necesario construir el polinomio de un grado $k-1$ y así sucesivamente.

$$\begin{array}{ll} x_0 & F_0 = G_{0,0} \\ x_1 & F_1 = G_{1,0} \quad F_{01} = G_{1,1} \\ x_2 & F_2 = G_{2,0} \quad F_{12} = G_{2,1} \quad F_{012} = G_{2,2} \\ x_3 & F_3 = G_{3,0} \quad F_{23} = G_{3,1} \quad F_{123} = G_{3,2} \quad F_{0123} = G_{3,3} \end{array}$$

Creando así una tabla triangular inferior, donde los valores son las interpolaciones anteriores, y más que hacer un algoritmo recursivo, se construye un algoritmo dinámico para evitar la saturación de la computadora.

Algoritmo de Neville

$$G_{i,j}(x) = \frac{(x-x_{i,j})G_{i,j}(x) - (x-x_i)G_{i-1,j-1}(x)}{(x_i-x_{i-j})} \quad (2.20)$$

Los cálculos continúan hasta hacer todas las interpolaciones con los datos de la tabla, pero si la diferencia entre una interpolación y la siguiente ya no es significativa, entonces se pueden detener los cálculos. Si se cumple que $|G_{i,i} - G_{i-1,j-1}| < \varepsilon$, donde ε es el valor de la tolerancia dada por el ingeniero.

■ **Ejemplo 2.4 — Interpolación de Neville.** La factorial se aplica a número naturales, no existe el concepto de la factorial de números como π . Con la ayuda de la interpolación de Neville, calcule el factorial hipotético de π , use los factoriales de 0 a 5 para hacer la interpolación.

n	1	2	3	4	5	
n!	1	2	6	24	120	

Solución

Usemos la interpolación de Neville con todos los puntos de la tabla

Programa 2.5. Interpolación de Neville

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x=np.array([0,1,2,3,4,5])
5 y=np.array([1,1,2,6,24,120])
6 n=x.size

```

```

7
8 G=np.zeros((n,n))
9 xi=np.pi
10 yi=0
11 #Calcula las interpolaciones y compara sus valores
12 G[:,0]=y
13 for i in range(1,n):
14     for j in range(1,i+1):
15         G[i,j]=((xi-x[i-j])*G[i,j-1]-(xi-x[i])*G[i-1,j-1])/(x[i]-x[i-j])
16         yi=G[i,i]
17         if abs(G[i,i]-G[i-1,i-1])<1e-5:
18             exit
19 print(G)
20
21 print(yi)
22
23 plt.plot(x,y,'o')
24 plt.plot(xi,yi,'sr')
25 plt.text(xi+0.1,yi, '$\pi$ ! ' + str(yi))
26 plt.title('Factorial n!')
27 plt.legend(['Factorial','Interpolación'])
28 plt.grid(True)
29 plt.show()

```

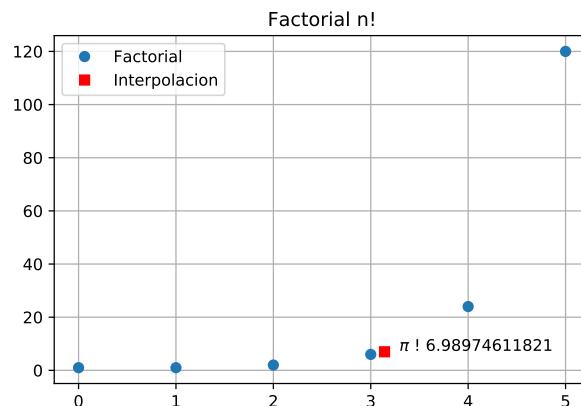


Figura 2.5. Interpolación de Neville

```

[[ 1.  0.          0.          0.          0.          0.          ←
  ]
 [ 1.  1.          0.          0.          0.          0.          ←
  ]
 [ 2.  3.14159265  4.36400587  0.          0.          0.          ←
  ]
 [ 6.  6.56637061  6.80883231  6.92422214  0.          0.          ←
  ]

```

```
[ 24.  8.54866776  7.69785855  7.44347633  7.33204343  0.      ↵
  ]
[120. -58.40710526  3.80844498  6.21781656  6.78726034  ↵
  6.98974612]]
6.98974611821
```

El valor hipotético de $\pi!$ es 6.98974611821

■

2.1.6 Spline lineal

Los métodos revisados hasta el momento ajustan un polinomio de menor grado que pasa por todos los puntos. Si los n puntos tienen una variación tal que puede generar un polinomio de grado $n - 1$, creando *oscilaciones* entre los puntos, entonces los puntos que se interpolan muestran un comportamiento distinto al resto.

El fenómeno de Runge es un problema que sucede cuando se usa interpolación polinómica con polinomios de grado alto. Lo descubrió Carle David Tolmé Runge, cuando exploraba el comportamiento de los errores al usar interpolación polinómica para aproximar determinadas funciones.

Considérese la función:

$$f(x) = \frac{1}{1 + 25x^2}$$

Runge descubrió que si se interpola esta función en puntos equidistantes entre -1 y 1, la interpolación resultante oscila hacia los extremos del intervalo.

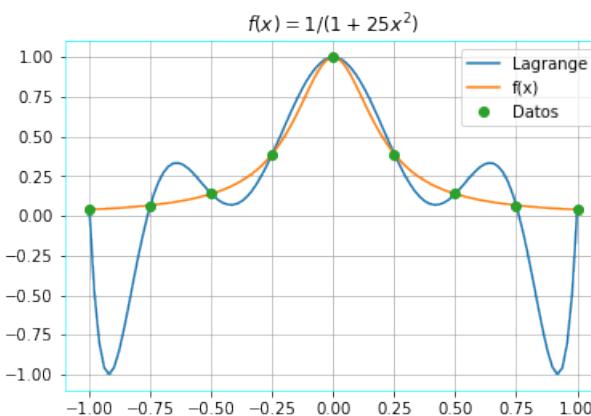


Figura 2.6. Problema de Runge

La oscilación se puede minimizar usando nodos de Chebyshev en lugar de equidistantes. En este caso, se garantiza que el error máximo disminuye al crecer el orden polinómico. El fenómeno demuestra que los polinomios de grado alto no son, en general, aptos para la interpolación. Este problema se puede evitar usando curvas spline.

Los trazadores o splines generan un polinomio de un grado menor entre los puntos, de tal manera que el comportamiento no debe rebasar el grado del trazador que se utiliza.

Definición 2.1.2 — Spline lineal. El spline lineal o trazador lineal ajusta un polinomio de grado 1 entre cada par de puntos, de tal manera que el comportamiento entre ellos no puede ser mayor del grado 1.

La expresión que define el trazador lineal es una función a trozos, donde se indica el polinomio de grado 1 que aplica en cada intervalo de $[x_i, x_{i+1}]$.

x	x_0	x_1	x_2	x_3	\dots	x_n
y	y_0	y_1	y_2	y_3	\dots	y_n

$$S(x) = \begin{cases} y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) & x \in [x_0, x_1] \\ y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) & x \in [x_1, x_2] \\ \vdots \\ y_{n-1} + \frac{y_n - y_{n-1}}{x_n - x_{n-1}}(x - x_{n-1}) & x \in [x_{n-1}, x_n] \end{cases}$$

Aplicar el trazador lineal es como aplicar la interpolación lineal (ecuación 2.6).

2.1.7 Spline cuadrático

El spline cuadrático o trazador cuadrático ajusta un polinomio de segundo grado entre cada par de puntos.

x	x_0	x_1	x_2	x_3	\dots	x_n
y	y_0	y_1	y_2	y_3	\dots	y_n

$$S_2(x) = \begin{cases} a_0 + b_0x + c_0x^2 & x \in [x_0, x_1] \\ a_1 + b_1x + c_1x^2 & x \in [x_1, x_2] \\ \vdots \\ a_n + b_nx + c_nx^2 & x \in [x_{n-1}, x_n] \end{cases}$$

De tal manera que se tiene un sistema con $3n$ incógnitas, a_i, b_i, c_i por cada ecuación. Como sólo se tienen n ecuaciones, no es posible todavía conocer los valores de a_i, b_i, c_i , se requieren de $3n$ ecuaciones.

Dado que los polinomios deben pasar por todos los puntos tabulados, quiere decir que si sustituimos x_0 en el primer polinomio se obtiene y_0 , y si sustituimos x_1 se obtiene y_1 , así para todos los puntos.

$$\begin{cases} a_0 + b_0x_0 + c_0x_0^2 = y_0 \\ a_0 + b_0x_1 + c_0x_1^2 = y_1 \\ a_1 + b_1x_1 + c_1x_1^2 = y_1 \\ a_1 + b_1x_2 + c_1x_2^2 = y_2 \\ \vdots \\ a_n + b_nx_{n-1} + c_nx_{n-1}^2 = y_{n-1} \\ a_n + b_nx_n + c_nx_n^2 = y_n \end{cases}$$

Se obtienen entonces $2n$ ecuaciones.

Uno de los requerimientos que se plantearon para los trazadores es que *suavizaran* las oscilaciones entre los puntos, la solución para los trazadores cuadráticos es proponer un polinomio de grado 2 entre cada par de puntos, como sabemos, hay infinidad de polinomios de grado 2 que unen dos puntos, se requiere de una condición más para hacer que los polinomios sean únicos. La condición que se usa es que la derivada del polinomio i -ésimo debe ser igual a la derivada del polinomio i -ésimo + 1. Esta solución, por un lado, asegura que la pendiente (derivada) del polinomio i -ésimo sea igual que la pendiente del siguiente polinomio, y no podrá suceder un comportamiento extraño entre cada par de puntos y, por otro lado, genera $n - 1$ ecuaciones más.

Derivamos entonces el trazador cuadrático de cada intervalo.

$$S'_2(x) = \begin{cases} b_0 + 2c_0x & x \in [x_0, x_1] \\ b_1 + 2c_1x & x \in [x_1, x_2] \\ \vdots \\ b_n + 2c_nx & x \in [x_{n-1}, x_n] \end{cases}$$

El primer polinomio $a_0 + b_0x + c_0x^2$ se une con el segundo polinomio $a_0 + b_0x + c_0x^2$ en el punto x_1 , el segundo se une con el tercero en x_2 , así sucesivamente hasta el penúltimo, que se une con el último en x_{n-1} para $n + 1$ puntos. Entonces, para asegurar que la derivada del primer polinomio se iguala a la del segundo, se evalúan ambas en el nodo que las comparte.

$$\begin{aligned} b_0 + 2c_0x_1 &= b_1 + 2c_1x_1 \\ b_1 + 2c_1x_2 &= b_2 + 2c_2x_2 \\ &\vdots \\ b_{n-1} + 2c_{n-1}x_{n-1} &= b_n + 2c_nx_{n-1} \end{aligned}$$

Se tienen ahora $n - 1$ ecuaciones para generar un total de $3n - 1$.

La derivada del primer polinomio no tiene manera de igualarse con una derivada anterior porque no existe un polinomio anterior, así que arbitrariamente decimos que la segunda derivada del primer polinomio es 0. Por lo tanto

$$2c_0 = 0$$

$$c_0 = 0$$

Se reduce entonces el sistema a $3n - 1$ ecuaciones. Esto nos dice que el primer polinomio se trata de un polinomio de primer grado, ya que su ecuación es $a_0 + b_0x$.

Se resuelve el sistema para obtener los coeficientes de las ecuaciones y conocer entonces los trazadores cuadráticos. Finalmente, se sustituye el valor de x ubicando a qué intervalo pertenece y usar ese polinomio para calcular la interpolación.

2.1.8 Spline cúbico

El spline cúbico o trazador cúbico ajusta un polinomio de tercer grado entre cada par de puntos.

x	x_0	x_1	x_2	x_3	\dots	x_n
y	y_0	y_1	y_2	y_3	\dots	y_n

$$S_3(x) = \begin{cases} a_0 + b_0x + c_0x^2 + d_0x^3 & x \in [x_0, x_1] \\ a_1 + b_1x + c_1x^2 + d_1x^3 & x \in [x_1, x_2] \\ \vdots \\ a_n + b_nx + c_nx^2 + d_nx^3 & x \in [x_{n-1}, x_n] \end{cases}$$

De tal manera que se tiene un sistema con $4n$ incógnitas, a_i, b_i, c_i, d_i por cada ecuación. Se deben determinar los valores de a_i, b_i, c_i, d_i , para construir el trazador, por lo que se requiere de $4n$ ecuaciones.

Dado que los polinomios deben pasar por todos los puntos tabulados, quiere decir que si sustituimos x_i en el polinomio correspondiente se obtiene y_i .

$$\begin{cases} a_0 + b_0x_0 + c_0x_0^2 + d_0x_0^3 = y_0 \\ a_0 + b_0x_1 + c_0x_1^2 + d_0x_1^3 = y_1 \\ a_1 + b_1x_1 + c_1x_1^2 + d_1x_1^3 = y_1 \\ a_1 + b_1x_2 + c_1x_2^2 + d_1x_2^3 = y_2 \\ \vdots \\ a_n + b_nx_{n-1} + c_nx_{n-1}^2 + d_{n-1}x_{n-1}^3 = y_{n-1} \\ a_n + b_nx_n + c_nx_n^2 + d_nx_n^3 = y_n \end{cases}$$

Se obtienen entonces $2n$ ecuaciones.

Se debe cumplir la condición: la primera derivada del polinomio i -ésimo debe ser igual a la derivada del polinomio i -ésimo + 1, evaluado en el nodo que comparten.

Derivamos entonces el trazador cúbico de cada intervalo.

$$S'_3(x) = \begin{cases} b_0 + 2c_0x + 3d_0x^2 & x \in [x_0, x_1] \\ b_1 + 2c_1x + 3d_1x^2 & x \in [x_1, x_2] \\ \vdots \\ b_n + 2c_nx + 3d_nx^2 & x \in [x_{n-1}, x_n] \end{cases}$$

Se igualan las derivadas en cada nodo:

$$\begin{aligned} b_0 + 2c_0x_1 &= b_1 + 2c_1x_1 \\ b_1 + 2c_1x_2 &= b_2 + 2c_2x_2 \\ &\vdots \\ b_{n-1} + 2c_{n-1}x_{n-1} &= b_n + 2c_nx_{n-1} \end{aligned}$$

Se tienen ahora $n - 1$ ecuaciones

Se debe cumplir también la condición: la segunda derivada del polinomio i -ésimo debe ser igual a la segunda derivada del polinomio i -ésimo + 1, evaluado en el nodo que comparten.

Obtenemos la segunda derivada del trazador cúbico de cada intervalo.

$$S''_3(x) = \begin{cases} 2c_0x + 6d_0x & x \in [x_0, x_1] \\ 2c_1x + 6d_1x & x \in [x_1, x_2] \\ \vdots \\ 2c_nx + 6d_nx & x \in [x_{n-1}, x_n] \end{cases}$$

Se igualan las derivadas en cada nodo:

$$\begin{aligned} 2c_0 + 6d_0x_1 &= 2c_1 + 6d_1x_1 \\ 2c_1 + 6d_1x_2 &= 2c_2 + 6d_2x_2 \\ &\vdots \\ 2c_{n-1} + 6d_{n-1}x_{n-1} &= 2c_n + 6d_nx_{n-1} \end{aligned}$$

Y se obtienen $n - 1$ ecuaciones, sólo se requieren 2 ecuaciones más para poder resolver el sistema. Bajo el entendimiento que el primer polinomio no tiene un polinomio que lo antecede y el último polinomio no tiene otro que precede, por lo tanto, la segunda derivada evaluada en x_0 es cero y la segunda derivada del último polinomio evaluado en x_n también es cero y se tienen las dos ecuaciones que hacían falta.

$$\begin{aligned} 2c_0 + 6d_0x_0 &= 0 \\ 2c_n + 6d_nx_n &= 0 \end{aligned}$$

En total se tienen $4n$ ecuaciones para resolver el sistema y obtener las $4n$ coeficientes que requieren los n polinomios.

Finalmente, se sustituye el valor de x ubicando a qué intervalo pertenece y usar ese polinomio para calcular la interpolación.

■ **Ejemplo 2.5 — Interpolación spline cúbico.** Usar la interpolación de spline cúbico en el problema de Runge.

Usamos la ecuación

$$f(x) = \frac{1}{1 + 25x^2}$$

para generar los datos

x	-1.0	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1.0
y	0.03846	0.06639	0.13793	0.39024	1	0.39024	0.13793	0.06639	0.03846

Solución

Usemos la interpolación de spline cúbico usando todos los puntos de la tabla

Programa 2.6. Interpolación de spline cúbico

```

1 import numpy as np
2 from scipy import interpolate
3 import matplotlib.pyplot as plt
4
5 x=np.array([-1.0,-0.75,-0.5,-0.25,0,0.25,0.5,0.75,1.0])
6 y=np.array([0.03846,0.06639,0.13793,0.39024,1,0.39024,\n
7             0.13793,0.06639,0.03846])
8
9 f=interpolate.interp1d(x,y,'cubic')
10
11 xs=np.linspace(-1,1,50)
12 ys=f(xs)
13 xi=0.85
14 yi=f(xi)
15
16 print(yi)
17
18 plt.plot(x,y,'o',label='Datos')
19 plt.plot(xi,yi,'sr',label='Interpolación')
20 plt.plot(xs,ys,'r:',label='Spline cúbico')
21 plt.text(xi+0.05,yi, ' interpolación ' + str(yi))
22 plt.title('f(x)=1/(1+25x^2)')
23 plt.legend()
24 plt.grid(True)
25 plt.show()
```

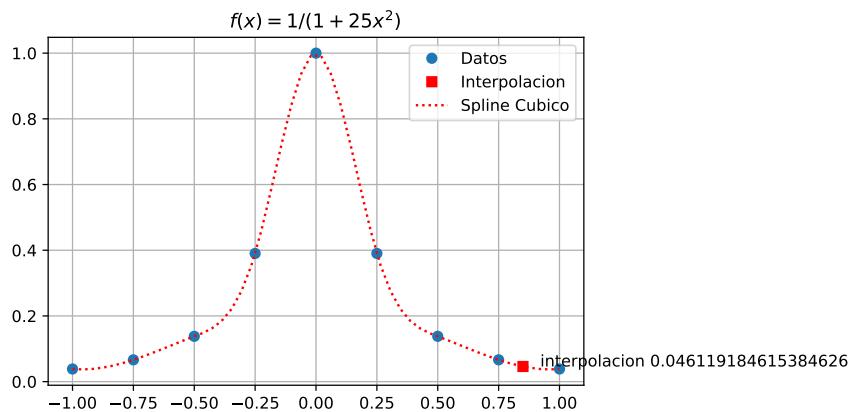


Figura 2.7. Interpolación de spline cúbico

0 . 0 4 6 1 1 9 1 8 4 6 1 5 3 8 4 6 2 6

■

2.1.9 Interpolación Nearest

La interpolación Nearest muestra como resultado la y_i más cercana a x .

x	x_0	x_1	x_2	x_3	...	x_n
y	y_0	y_1	y_2	y_3	...	y_n

Por ejemplo, si x es más cercano a x_0 el valor de la interpolación es y_0 , si x es más cercano a x_2 , el valor de la interpolación es y_2 .

■ **Ejemplo 2.6 — Interpolación Nearest.** Usar la interpolación Nearest en el problema de Runge.

Usamos la ecuación

$$f(x) = \frac{1}{1 + 25x^2}$$

para generar los datos

x	-1.0	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1.0
y	0.03846	0.06639	0.13793	0.39024	1	0.39024	0.13793	0.06639	0.03846

Solución

Usemos la interpolación Nearest usando todos los puntos de la tabla

Programa 2.7. Interpolación Nearest

```

1 import numpy as np
2 from scipy import interpolate
3 import matplotlib.pyplot as plt

```

```

4
5 x=np.array([-1.0,-0.75,-0.5,-0.25,0,0.25,0.5,0.75,1.0])
6 y=np.array([0.03846,0.06639,0.13793,0.39024,1,0.39024,\n
7 0.13793,0.06639,0.03846])
8
9 f=interpolate.interp1d(x,y,'nearest')
10
11 xs=np.linspace(-1,1,50)
12 ys=f(xs)
13 xi=0.85
14 yi=f(xi)
15
16 print(yi)
17
18 plt.plot(x,y,'o',label='Datos')
19 plt.plot(xi,yi,'sr',label='Interpolación')
20 plt.plot(xs,ys,'r:',label='Nearest')
21 plt.text(xi+0.05,yi, ' interpolación ' + str(yi))
22 plt.title('f(x)=1/(1+25 x^2)')
23 plt.legend()
24 plt.grid(True)
25 plt.show()

```

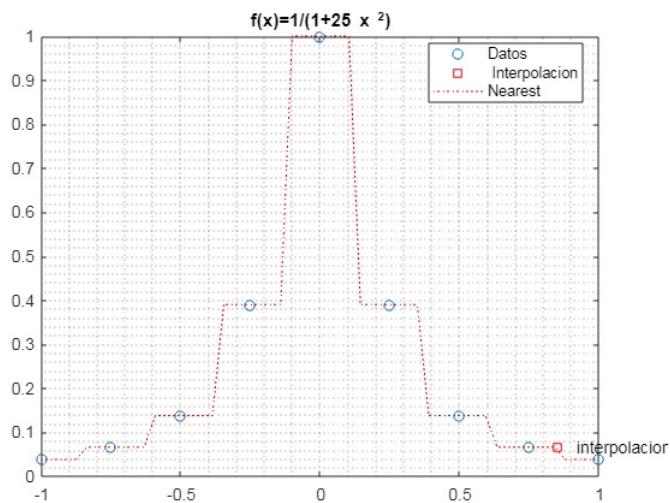


Figura 2.8. Interpolación Nearest

```

yi =
0.0664

```

■ **Ejemplo 2.7 — Interpolación de los 6 métodos.** Usar los 6 métodos de la interpolación en el problema de Runge.

Usamos la ecuación

$$f(x) = \frac{1}{1+25x^2}$$

para generar los datos

x	-1.0	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1.0
y	0.03846	0.06639	0.13793	0.39024	1	0.39024	0.13793	0.06639	0.03846

Solución

Usemos los 6 métodos de interpolación con todos los puntos de la tabla

Programa 2.8. Interpolación con 6 métodos

```

1 import numpy as np
2 from scipy import interpolate
3 import matplotlib.pyplot as plt
4
5 x=np.array([-1.0,-0.75,-0.5,-0.25,0,0.25,0.5,0.75,1.0])
6 y=np.array([0.03846,0.06639,0.13793,0.39024,1,0.39024,\n
7           0.13793,0.06639,0.03846])
8 tipos = ('nearest', 'zero', 'linear', 'slinear', 'quadratic', '←\n    cubic')
9
10 xs=np.linspace(-1,1,50)
11
12 for tipo in tipos:
13     f=interpolate.interp1d(x,y,kind=tipo)
14     ys=f(xs)
15     plt.plot(xs,ys,'-',label=tipo)
16
17 plt.plot(x,y,'o',label='Datos')
18 plt.title('f(x)=1/(1+25x^2)')
19 plt.legend()
20 plt.grid(True)
21 plt.show()
```

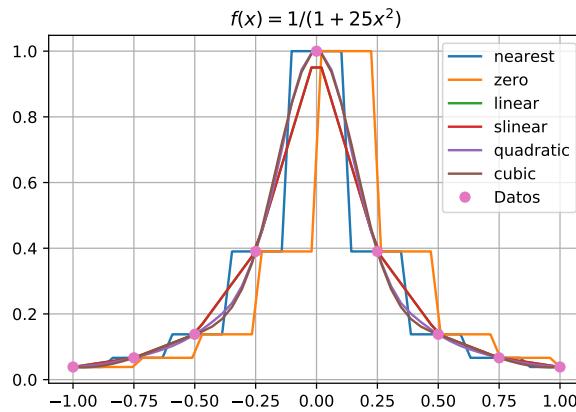


Figura 2.9. Interpolación con 6 métodos

2.1.10 Interpolación en dos dimensiones

La interpolación en dos dimensiones se aplica para casos donde se tiene dos variables independientes x e y , y una variable dependiente z . Las variables independientes son vectores de datos, mientras que la variable dependiente es una matriz de datos, un valor para cada dato de x e y .

$y \setminus x$	x_0	x_1	x_2	...	x_n
y_0	$z_{0,0}$	$z_{0,1}$	$z_{0,2}$...	$z_{0,n}$
y_1	$z_{1,0}$	$z_{1,1}$	$z_{1,2}$...	$z_{1,n}$
y_2	$z_{2,0}$	$z_{2,1}$	$z_{2,2}$...	$z_{2,n}$
\vdots					
y_m	$z_{m,0}$	$z_{m,1}$	$z_{m,2}$...	$z_{m,n}$

■ **Ejemplo 2.8 — Interpolación en 2 dimensiones.** La siguiente información muestra la entalpía de vapor supercalentado a distintas temperaturas y presiones, use la interpolación en dos dimensiones para estimar la entalpía a 420K y 190kPa.

Entalpía de vapor supercalentado

$Presión(kPa) \setminus Temp(K)$	300	350	400	450	500
150	3073.3	3174.7	3277.5	3381.7	3487.6
200	3072.1	3173.8	3276.7	3381.1	3487.0
250	3070.9	3172.8	3275.9	3380.4	3486.5
300	3069.7	3171.9	3275.2	3379.8	3486.0

Solución

Usemos la interpolación cúbica de dos dimensiones:

Programa 2.9. Interpolación en 2 dimensiones

```
1 import numpy as np
```

```
2 from scipy import interpolate
3 import matplotlib.pyplot as plt
4 from matplotlib import cm
5 from matplotlib.ticker import LinearLocator, FormatStrFormatter
6
7 x=np.array([300,350,400,450,500])
8 y=np.array([150,200,250,300])
9 z=np.array([[3073.3,3174.7,3277.5,3381.7,3487.6],
10           [3072.1,3173.8,3276.7,3381.1,3487.0],
11           [3070.9,3172.8,3275.9,3380.4,3486.5],
12           [3069.7,3171.9,3275.2,3379.8,3486.0]])
13
14 f=interpolate.interp2d(x,y,z,'cubic')
15 zi=f(420,190)
16
17 print(zi)
18 X, Y = np.meshgrid(x, y)
19
20 fig = plt.figure()
21 ax = fig.gca(projection='3d')
22 ax.view_init(10,35)
23 ax.set_xlabel('Temperatura (K)')
24 ax.set_ylabel('Presion (kPa)')
25 ax.set_zlabel('Entalpia (kJ/Kg)')
26 ax.set_title('Entalpia de vapor supercalentado')
27 surf=ax.plot_surface(X, Y, z, cmap=cm.coolwarm,
28                       linewidth=0, antialiased=False)
29
30 ax.set_zlim(3000, 3500)
31 ax.xaxis.set_major_locator(LinearLocator(10))
32 ax.xaxis.set_major_formatter(FormatStrFormatter('%.0f'))
33
34 fig.colorbar(surf, shrink=0.5, aspect=5)
35
36 plt.show()
37 #fig.savefig("interp2d.pdf", bbox_inches='tight')
```

```
[ 3318.426752]
```

La entalpía del vapor supercalentado a 420K y 190Kpa es de 3318.426752 kJ/kg ■

2.2 Regresión por mínimos cuadrados

Las técnicas de regresión se aplican a una lista de datos (o tabla de datos) donde el polinomio que se ajusta minimiza la distancia entre los puntos tabulados y el polinomio. Esto quiere decir que no toca los puntos necesariamente, calculando entonces un dato corregido de y_i para un valor x_i de la tabla.

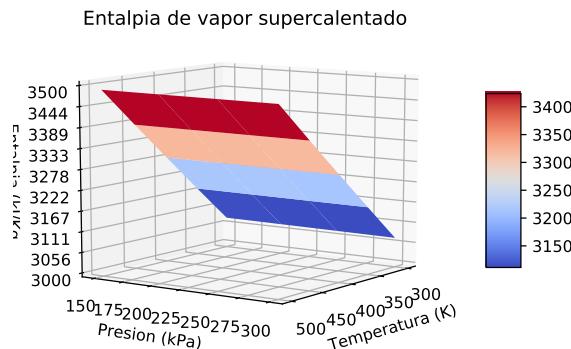


Figura 2.10. Interpolación en 2 dimensiones

2.2.1 Lineal

En la regresión lineal se ajusta un polinomio de grado 1, es decir, una línea recta.

$$y = mx + b$$

Donde m es la pendiente e indica la proporcionalidad que tiene la variable dependiente y con respecto a la variable independiente x . El valor de b indica la ordenada al origen. La regresión lineal obtiene estos valores con las ecuaciones:

Pendiente

$$m = \frac{n \sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2} \quad (2.21)$$

Ordenada al origen

$$b = \frac{\sum x^2 \sum y - \sum x \sum xy}{n \sum x^2 - (\sum x)^2} \quad (2.22)$$

Entonces para calcular m y b se deben hacer las sumas de x , y , xy y x^2 , como se indica en la siguiente tabla.

Se sustituyen en las ecuaciones 2.21 y 2.22 para construir la ecuación lineal, con el polinomio construido sustituimos los valores de x de la tabla para obtener las y corregidas. Como se conoce la ecuación de la línea recta que representa a los datos, también se puede sustituir un valor de x tal que $x \in [x_0, x_n]$ lo que nos daría el valor correspondiente de acuerdo con el comportamiento lineal.

x	y	xy	x^2	y^2
x_0	y_0	x_0y_0	x_0^2	y_0^2
x_1	y_1	x_1y_1	x_1^2	y_1^2
x_2	y_2	x_2y_2	x_2^2	y_2^2
\vdots				
x_n	y_n	x_ny_n	x_n^2	y_n^2
$\sum x_i$	$\sum y_i$	$\sum x_iy_i$	$\sum x_i^2$	$\sum y_i^2$
$(\sum x_i)^2$	$(\sum y_i)^2$			

Cuadro 2.2. Cálculo de cada término para la regresión lineal

La distancia de los puntos a la línea recta puede ser mucha (mucha dispersión) o poca (poca dispersión). La medida se obtiene con la siguiente fórmula:

Medida de dispersión

$$r = \frac{n \sum xy \sum x \sum y}{\sqrt{(n \sum x^2 - (\sum x)^2)(n \sum y^2 - (\sum y)^2)}} \quad (2.23)$$

$$r = \begin{cases} 1 & \text{Directamente proporcional exacto} \\ 0 < r < 1 & \text{Mucha dispersión} \\ 0 & \text{No existe relación} \\ -1 < r < 0 & \text{Mucha dispersión} \\ -1 & \text{Inversamente proporcional exacto} \end{cases}$$

■ **Ejemplo 2.9 — Regresión lineal.** En la fabricación de láminas de acero, el método para deformar acero a temperatura normal mantiene una relación inversa con la dureza de éste, ya que a medida que la deformación crece, se afecta la dureza del acero. Con el objetivo de conocer esta relación se tomaron los siguientes datos.

Deformación	6	9	11	13	22	26	28	33	35
Dureza	68	67	65	53	44	40	37	34	32

Solución

La deformación es la variable dependiente y , y la dureza es la variable independiente x .

Programa 2.10. Regresión lineal

```

1 import numpy as np
2 from scipy import stats
3 import matplotlib.pyplot as plt
4
```

```

5
6 x = np.array([6,9,11,13,22,26,28,33,35])
7 y = np.array([68,67,65,53,44,40,37,34,32])
8
9 m, b, r, p, std_err = stats.linregress(x, y)
10
11 print('Ordenada al origen ', b)
12 print('pendiente ', m)
13 print('coeficiente de correlacion ', r)
14
15 y_pred = m*x+b
16 print('datos corregidos ', y_pred, sep='\n')
17
18 #fig=plt.figure()
19 plt.plot(x,y,'o',label='Datos')
20 plt.plot(x,y_pred,'r:',label='Regresion lineal')
21 plt.title('Deformacion y Dureza')
22 plt.xlabel('Dureza')
23 plt.ylabel('Deformacion')
24 plt.legend()
25 plt.grid(True)
26 plt.show()
27 #fig.savefig("reglineal.pdf", bbox_inches='tight')

```

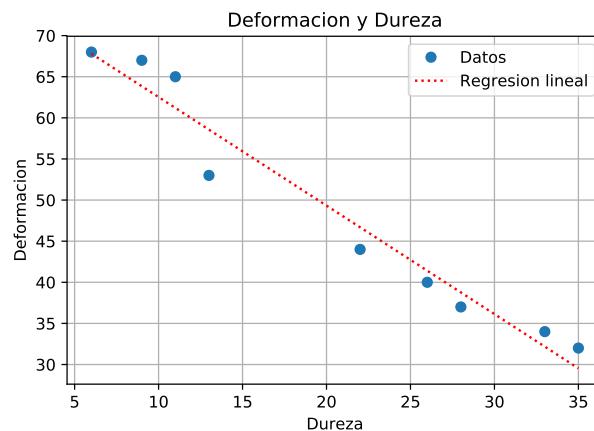


Figura 2.11. Regresión Lineal

```

Ordenada al origen 75.7199858757
pendiente [-1.31956215]
coeficiente de determinacion 0.955166014257
datos corregidos
[ 67.80261299  63.84392655  61.20480226  58.56567797  46.68961864
 41.41137006  38.77224576  32.17443503  29.53531073]

```

La ecuación que muestra la relación de la deformación y la dureza es

$$\text{Deformación} = -1.31956215 \text{Dureza} + 75.7199858757$$

como la pendiente es un valor negativo $[-1.31956215]$, se confirma que la relación es inversamente proporcional. ■

2.2.2 Polinómica

Cuando se sabe que la relación de la variable dependiente no es lineal, se aplica la regresión polinómica que ajusta un polinomio de grado mayor a 1 que ajusta mejor a los datos.

$$y = a + bx + cx^2 + dx^3 + \dots$$

Donde se deben determinar los coeficientes del polinomio de tal manera que se ajusten mejor a los datos, minimizando su distancia a los puntos.

■ **Ejemplo 2.10 — Regresión cuadrática.** Un proyectil es disparado hacia arriba desde el piso, se mide la altura del proyectil en el tiempo y se obtienen los siguientes datos.

Tiempo(s)	0	0.5	1.0	1.5	2.0	2.5
Altura(ft)	0	20.5	31.36	36.25	30.41	28.23

Encuentre la regresión cuadrática que describe la trayectoria del proyectil.

Solución

La altura es la variable dependiente y , y el tiempo es la variable independiente x .

Programa 2.11. Regresión cuadrática

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.array([0,0.5,1,1.5,2,2.5])
5 y = np.array([0,20.5,31.36,36.25,30.41,28.23])
6 xs=np.linspace(0,3.3,100)
7
8 p=np.polyfit(x,y,2)
9 print(p)
10 fig=plt.figure()
11 plt.scatter(x, y,label='Datos')
12 plt.plot(xs, np.polyval(p,xs), ':r', label='Interpolación ←
    cuadrática')
13 plt.xlabel('Tiempo')
14 plt.ylabel('Distancia')
15 plt.title('Tiro Parabólico')
16 plt.grid()
17 plt.show()
```

```
18 fig.savefig("regcuad.pdf", bbox_inches='tight')
```

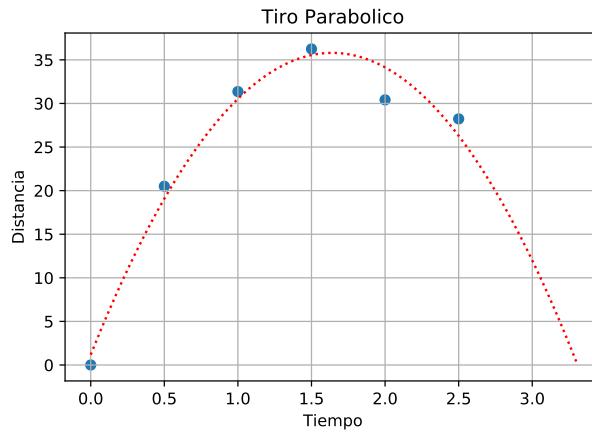


Figura 2.12. Regresión cuadrática

```
[ -12.87142857  42.22257143   1.17714286]
```

La ecuación que describe la trayectoria del proyectil es

$$\text{altura} = -12.87142857 \text{tiempo}^2 + 42.22257143 \text{tiempo} + 1.17714286$$

■ **Ejemplo 2.11 — Regresión cúbica.** Con los siguientes datos de Cp del propano vs temperatura, obtenga los coeficientes de la ecuación de Cp de la forma.

$$Cp = a + bT + cT^2 + dT^3$$

Temp(C)	50	100	150	200	273.16	298.15	300	400	...
Cp(kJ/kg)	34.06	41.3	48.79	56.07	68.74	73.6	73.93	94.01	

Temp(C)	500	600	700	800	900	1000	1100
Cp(kJ/kg)	112.59	142.67	154.77	163.35	174.6	182.67	128.7

Aplique la regresión cúbica para obtener los coeficientes a, b, c, d .

Solución

La temperatura es la variable independiente x , y el Cp es la variable dependiente y .

Programa 2.12. Regresión cúbica

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.array([50,100,150,200,273.16,298.15,300,400,500,600,\n
5 700,800,900,1000,1100,1200,1300,1400,1500])
6 y = np.array([34.06,41.3,48.79,56.07,68.74,73.6,73.93,94.01,\n
7 112.59,128.7,142.67,154.77,163.35,174.6,182.67,\n
8 189.74,195.85,201.21,205.89])
9
10 xs=np.linspace(50,1500,100)
11
12 p=np.polyfit(x,y,3)
13 print(p)
14 #fig=plt.figure()
15 plt.scatter(x, y,label='Datos')
16 plt.plot(xs, np.polyval(p,xs), ':r',label='Interpolacion cúbica')
17 plt.xlabel('Temperatura')
18 plt.ylabel('Cp')
19 plt.legend()
20 plt.title('Cp del propano')
21 plt.grid()
22 plt.show()
23 #fig.savefig("regcubi.pdf", bbox_inches='tight')

```

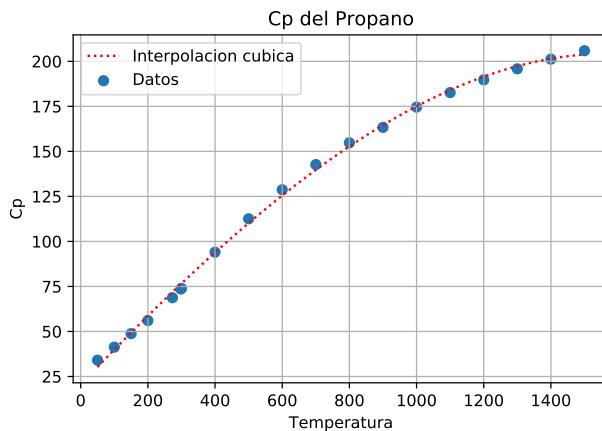


Figura 2.13. Regresión cúbica

```
[ -1.51288977e-08 -2.68480214e-05 1.96470716e-01 2.05235040e+01]
```

Por lo tanto, la ecuación del Cp del propano es

$$Cp = -1.51288977e-08T^3 - 2.68480214e-05T^2 + 1.96470716e-01T + 2.05235040e+01$$

2.3 No lineal

En una regresión no lineal, se ajusta una función que involucra tanto a la variable independiente x como un conjunto de parámetros β que se deben determinar para que la función f se ajuste a los datos.

$$y = f(x, \beta) + \epsilon$$

En algunos casos, los modelos a determinar se acercan a una expresión simple no lineal donde se aplican transformaciones algebraicas para lograr linealizar la ecuación y entonces aplicar una regresión lineal, al final se invierte la transformación para obtener los valores de los parámetros β .

2.3.1 Exponencial

El modelo exponencial se aplica a problemas de reacciones químicas. El modelo tiene la forma:

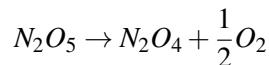
$$y = ae^{(bx)}$$

Donde a y b son los parámetros para determinar. Para linealizar la función anterior se aplican logaritmos de ambos lados de la ecuación:

$$\ln(y) = \ln(a) + bx$$

De esta forma se obtiene una ecuación de una línea recta y se aplica la regresión lineal para obtener los parámetros $\ln(a)$ y b , después para obtener el valor de a se calcula $\exp^{\ln(a)}$ y se sustituye en la ecuación exponencial. Observe también que para aplicar la regresión lineal se debe calcular el $\ln(y)$.

■ **Ejemplo 2.12 — Regresión exponencial.** El anhídrido nítrico se descompone homogéneamente de acuerdo con la siguiente reacción.



Los datos de la reacción de N_2O_5 en el tiempo a una temperatura de 313K se muestran en la siguiente tabla:

tiempo(s)	0	500	1000	1500	2000	2500	3000
$N_2O_5(\text{gr/l})$	0.1000	0.0892	0.0776	0.0705	0.0603	0.0542	0.0471

Asumiendo que la reacción es de primer orden $r = kC_A$, entonces la concentración varía de acuerdo con:

$$C_A = C_{A_0} e^{-kt}$$

Donde C_{A_0} es la concentración inicial. Encontrar los valores de C_{A_0} y k usando una regresión exponencial.

Solución

Se construye la función con la forma propuesta para obtener los parámetros.

Programa 2.13. Regresión exponencial

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 x = np.array([0,500,1000,1500,2000,2500,3000])
6 y = np.array([0.1000,0.0892,0.0776,0.0705,0.0603,0.0542,0.0471])
7
8 xs=np.linspace(0,3000,100)
9
10 def f(x,a,b):
11     return a*np.exp(-b*x)
12
13 param,_=curve_fit(f,x,y,p0=[0.1,-0.0001])
14 print(param)
15
16 fig=plt.figure()
17 plt.scatter(x, y,label='Datos')
18 plt.plot(xs, f(xs,*param), ':r',label='Regresión exponencial')
19 plt.xlabel('tiempo')
20 plt.ylabel('Concentración')
21 plt.legend()
22 plt.title('Reaccion de $N_2O_5$')
23 plt.grid()
24 plt.show()
25 fig.savefig("regexp.pdf", bbox_inches='tight')
```

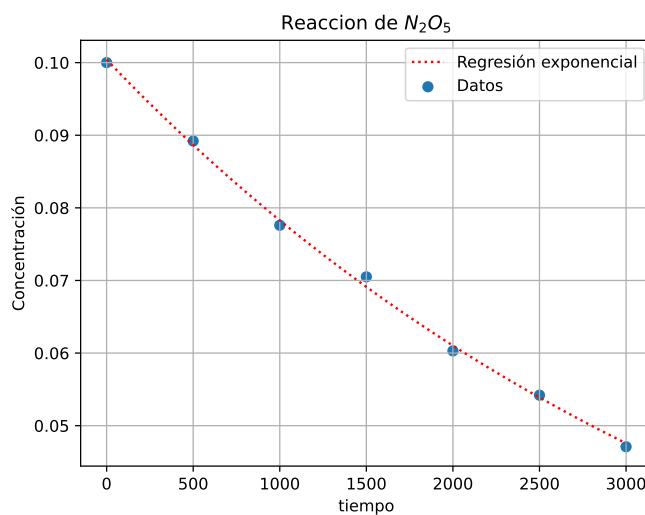


Figura 2.14. Regresión exponencial

```
[ 0.10040589  0.00024895]
```

Por lo tanto, la ecuación de la concentración de N_2O_5 es:

$$C_A = 0.10040589e^{-0.00024895t}$$

■

2.3.2 Potencial

El modelo potencial tiene la forma:

$$y = ax^b$$

Donde a y b son los parámetros para determinar. Para linealizar la función anterior se aplican logaritmos de ambos lados de la ecuación:

$$\ln(y) = \ln(a) + b\ln(x)$$

De esta forma se obtiene una ecuación de una línea recta y se aplica la regresión lineal para obtener los parámetros $\ln(a)$ y b , después para obtener el valor de a se calcula $\exp^{\ln(a)}$ y se sustituye en la ecuación potencial. Observe también que para aplicar la regresión lineal se debe calcular el $\ln(x)$ y $\ln(y)$.

■ **Ejemplo 2.13 — Regresión potencial.** La ecuación de Bernoulli para un flujo continuo que sale de un tanque se relaciona con el flujo Q y la altura h del tanque.

$$Q = kh^a$$

Los datos del flujo como función del nivel del líquido se muestran en la siguiente tabla:

Altura h	5	40	70	100
Flujo Q	0.2322	0.6450	0.820	0.9856

Estimar los valores de k y a de la ecuación.

Solución

Se construye la función con la forma propuesta para obtener los parámetros.

Programa 2.14. Regresión potencial

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 x = np.array([5, 40, 70, 100])
6 y = np.array([0.2322, 0.6450, 0.820, 0.9856])
7
```

```

8 xs=np.linspace(0,100,100)
9
10 def f(x,a,b):
11     return a*np.power(x,b)
12
13 param,_=curve_fit(f,x,y)
14 print(param)
15
16 fig=plt.figure()
17 plt.scatter(x, y, label='Datos')
18 plt.plot(xs, f(xs,*param), ':r', label='Regresión potencial')
19 plt.xlabel('altura')
20 plt.ylabel('Flujo')
21 plt.legend()
22 plt.title('Flujo del tanque')
23 plt.grid()
24 plt.show()
25 fig.savefig("regpot.pdf", bbox_inches='tight')

```

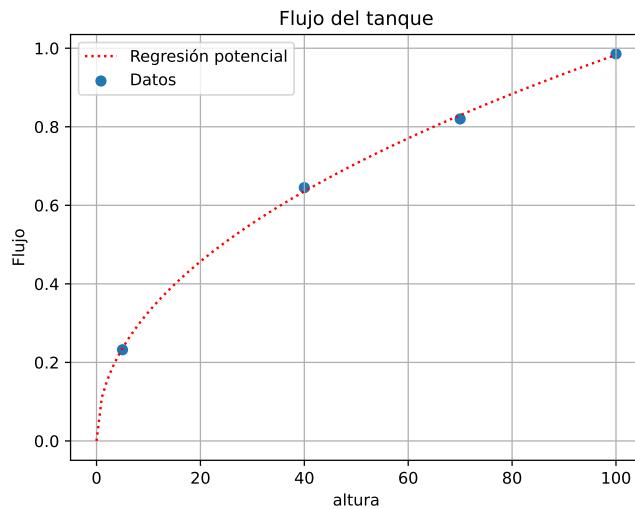


Figura 2.15. Regresión potencial

```
[ 0.10975514  0.47604171]
```

Por lo tanto, la ecuación es:

$$Q = 0.10975514h^{0.47604171}$$

■

2.3.3 Recíproco o hiperbólico

El modelo recíproco o hiperbólico tiene la forma:

$$y = \frac{1}{a+bx}$$

Donde a y b son los parámetros para determinar. Para linealizar la función anterior se aplican el recíproco de ambos lados de la ecuación:

$$\frac{1}{y} = a + bx$$

De esta forma se obtiene una ecuación de una línea recta y se aplica la regresión lineal para obtener los parámetros a y b , se sustituye en la ecuación recíproca. Observe también que para aplicar la regresión lineal se debe calcular $\frac{1}{y}$.

■ **Ejemplo 2.14 — Regresión recíproca o hiperbólica.** La ecuación de Antoine obtiene la presión de vapor de una sustancia a distintas temperaturas.

$$\log(P) = a + \frac{b}{c+T}$$

Los datos de la presión de vapor como función de la temperatura se muestran en la siguiente tabla:

T (C)	20	22	25	27	30	34	40	43	48	51
log(P) (mmHg)	1.175	1.230	1.312	1.365	1.443	1.544	1.690	1.761	1.875	1.942

Estimar los valores de a , b y c de la ecuación de Antoine.

Solución

Se construye la función con la forma propuesta para obtener los parámetros.

Programa 2.15. Regresión recíproca o hiperbólica

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 x = np.array([20,22,25,27,30,34,40,43,48,51])
6 y = np.array([1.175,1.230,1.312,1.365,1.443,
7                 1.544,1.690,1.761,1.875,1.942])
8
9 xs=np.linspace(20,51,100)
10
11 def f(x,a,b,c):
12     return a+b/(x+c)
13
14 param,_=curve_fit(f,x,y)
15 print(param)
16
17 fig=plt.figure()
18 plt.scatter(x, y,label='Datos')

```

```

19 plt.plot(xs, f(xs,*param), ':r', label='Regresión')
20 plt.xlabel('Temperatura')
21 plt.ylabel('Log(P)')
22 plt.legend()
23 plt.title('Presión de vapor de la sacarosa')
24 plt.grid()
25 plt.show()
26 fig.savefig("regAntoine.pdf", bbox_inches='tight')

```

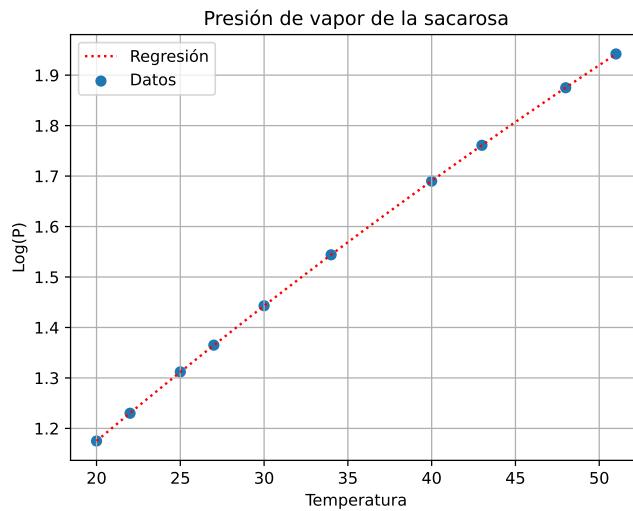


Figura 2.16. Regresión recíproca o hiperbólica

```
[ 7.97152383 -1657.04353334   223.80983991]
```

Por lo tanto, la ecuación de Antoine es:

$$\log(P) = 7.97152383 - \frac{1657.04353334}{223.80983991 + T}$$

■

2.4 Múltiple

En la regresión múltiple se tiene más de una variable independiente $x_1, x_2, x_3, \dots, x_n$ y sólo una variable dependiente y . La regresión múltiple nos permite analizar la relación que tiene la variable de estudio y con las variables regresoras $x_1, x_2, x_3, \dots, x_n$. Por ejemplo, la velocidad de reacción (variable de estudio) con la temperatura y la concentración (variables regresoras). Se puede seguir un modelo lineal

$$y = a_0 + a_1x_1 + a_2x_2 + \dots + a_nx_n$$

Donde los coeficientes a determinar son $a_0, a_1, a_2, \dots, a_n$. O se puede tener un modelo no lineal

$$y = f(X, \beta)$$

donde β es un conjunto de parámetros a determinar de acuerdo con el modelo propuesto.

■ **Ejemplo 2.15 — Regresión lineal múltiple.** En un estudio sobre el crecimiento de un parásito se contaron éstos en 10 localizaciones con diversas condiciones de temperatura y humedad, dando los siguientes resultados.

T	15	16	24	13	21	16	22	18	20	16
H	70	65	71	64	84	86	72	84	71	75
Parásitos	156	157	177	145	197	184	172	187	157	169

Usar el siguiente modelo para la cantidad de parásitos:

$$\text{Parásitos} = b_1 + b_2 \times \text{Temperatura} + b_3 \times \text{Humedad}$$

Estimar los valores de b_1, b_2 y b_3 de la ecuación anterior.

Solución

Se construye la función con la forma propuesta para obtener los parámetros.

Programa 2.16. Regresión lineal múltiple

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 x = np.array([[15,16,24,13,21,16,22,18,20,16], \
6               [70,65,71,64,84,86,72,84,71,75]])
7 z = np.array([156,157,177,145,197,184,172,187,157,169])
8
9 def f(x,a,b,c):
10     return a+b*x[0]+c*x[1]
11
12 param,_=curve_fit(f,x,z)
13 print(param)
14
15 X=x[:, :]
16 Y=x[:, :]
17
18 fig = plt.figure()
19 ax = fig.gca(projection='3d')
20 ax.view_init(15,60)
21 ax.scatter(X, Y, z)
22 ax.set_xlabel('Temperatura')
23 ax.set_ylabel('Humedad')
24 ax.set_zlabel('Crecimiento')
25 plt.title('Crecimiento de parásitos')
```

```

26
27 X = np.arange(13,24,0.5)
28 Y = np.arange(60, 85, 0.5)
29 X, Y = np.meshgrid(X, Y)
30 Z=f(np.array([X,Y]),*param)
31 surf = ax.plot_surface(X, Y, Z,cmap=plt.cm.coolwarm,
32                         linewidth=0, antialiased=False)
33
34
35 plt.show()
36 fig.savefig("regmultiplelin.pdf", bbox_inches='tight')

```

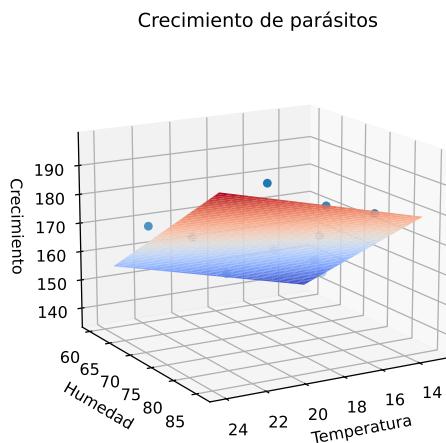


Figura 2.17. Regresión lineal múltiple

```
[ 13.97853561    1.65787362    1.69964895]
```

Por lo tanto, la ecuación es:

$$\text{Parásitos} = 13.97853561 + 1.65787362 \text{Temperatura} + 1.69964895 \text{Humedad}$$

■ **Ejemplo 2.16 — Regresión no lineal múltiple.** Los datos de una reacción irreversible no isotérmica se muestran en la siguiente tabla.

C_A (gmol/l)	0.05	0.55	0.65	0.75	0.9	1	1.05
T(K)	395	410	380	388	400	373	405
Vel. reacción(gmol/l s)	0.1561	2.8098	1.2224	1.9834	3.3847	1.3993	4.5913

Se propone el siguiente modelo para la velocidad de reacción:

$$r = k_0 e^{\frac{-\beta}{T}} C_A^n$$

Con los datos de la tabla, estimar k_0 , β y n

Solución

Se construye la función con el modelo propuesto para obtener los parámetros.

Programa 2.17. Regresión no lineal múltiple

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 x = np.array([[0.05, 0.55, 0.65, 0.75, 0.9, 1, 1.05], \
6               [395, 410, 380, 388, 400, 373, 405]])
7 z = np.array([0.1561, 2.8098, 1.2224, 1.9834, 3.3847, 1.3993, 4.5913])
8
9 xs=np.linspace(20,51,100)
10
11 def f(x,k,b,n):
12     return k*np.exp(-b/x[1])*np.power(x[0],n)
13
14 param,_=curve_fit(f,x,z)
15 print(param)
16
17 X=x[0,:]
18 Y=x[1,:]
19
20 fig = plt.figure()
21 ax = fig.gca(projection='3d')
22 ax.view_init(10,50)
23 ax.scatter(X, Y, z)
24 ax.set_xlabel('Concentración (gmol/l)')
25 ax.set_ylabel('Temperatura (K)')
26 ax.set_zlabel('Velocidad de reacción')
27 plt.title('Velocidad de reacción')
28
29 X = np.arange(0,1.05,0.01)
30 Y = np.arange(395, 405, 1)
31 X, Y = np.meshgrid(X, Y)
32 Z=f(np.array([X,Y]),*param)
33 surf = ax.plot_surface(X, Y, Z,cmap=plt.cm.coolwarm,
34                         linewidth=0, antialiased=False)
35
36
37 plt.show()
38 fig.savefig("regmultiple.pdf", bbox_inches='tight')
```

[1.52957201e+06 5.16886089e+03 9.87410537e-01]

Por lo tanto, la ecuación para la velocidad de reacción es:

$$r = 1.52957201e+06 e^{\frac{-5.16886089e+03}{T}} C_A^{9.87410537e-01}$$

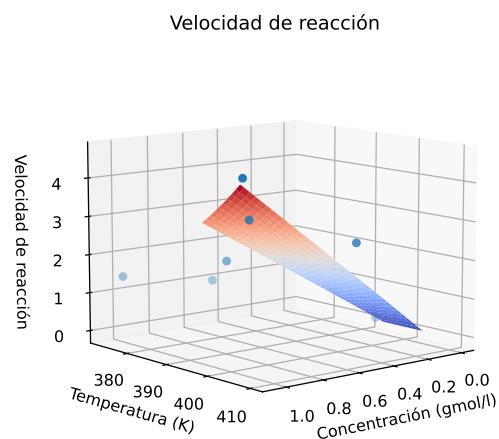
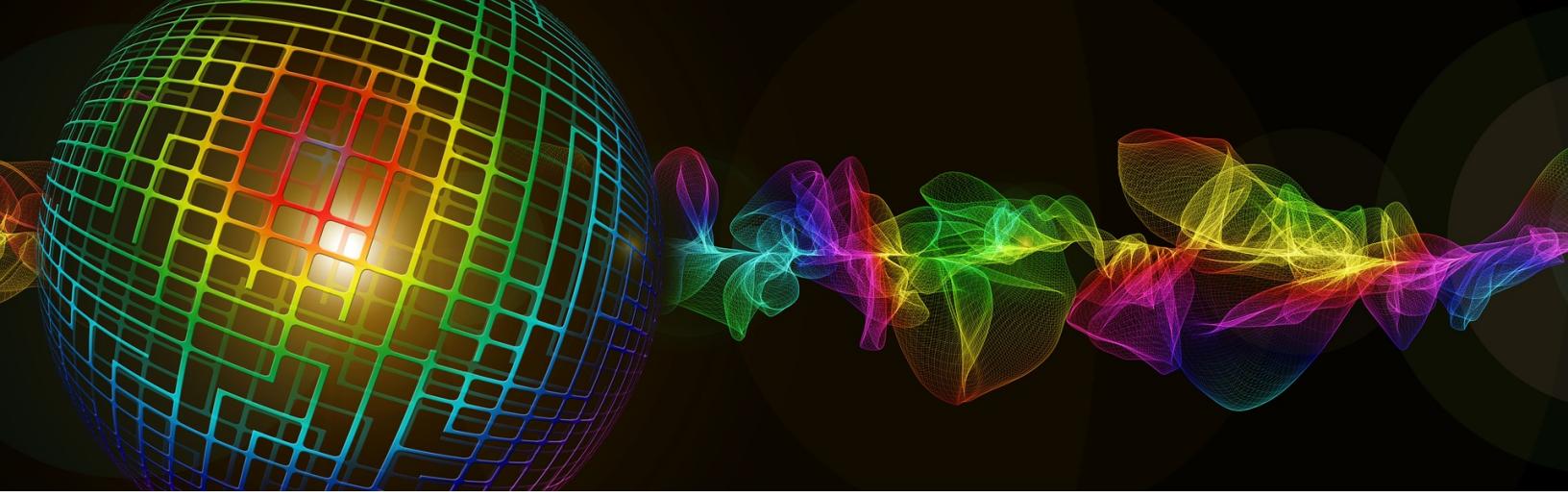


Figura 2.18. Regresión no lineal múltiple



3. Ecuaciones no lineales

La solución de ecuaciones no lineales significa encontrar un valor de x donde $f(x) = 0$, a este valor particular se llama **raíz**. El término raíz viene de la operación raíz cuadrada, cuando queremos saber el origen cuadrático (raíz cuadrada) de 4 ($\sqrt{4}$) sabemos que es 2, así la $\sqrt{9}$ es 3, es decir 3 es la raíz u origen cuadrático de 9. La operación $\sqrt{}$ es relativamente sencilla para aquellos valores donde la raíz es un número entero, pero para aquellos números donde la raíz no es entero o, en algunos casos, es un número irracional, la solución no es tan sencilla. Veamos el caso de $\sqrt{2}$, si lo planteamos algebráicamente, queremos encontrar la $\sqrt{2}$ el cual es un valor x , entonces:

$$\sqrt{2} = x$$

Si elevamos al cuadrado ambos lados de la ecuación:

$$2 = x^2$$

Expresado como una ecuación en x se obtiene:

$$f(x) = x^2 - 2 = 0$$

Entonces el problema se explica como: se desea conocer el valor de x , donde $f(x) = 0$. Entonces el valor de x es la raíz de la ecuación.

Observe que la ecuación $f(x)$ debe expresarse igualada a 0 para que los métodos que se aplican encuentren la raíz, es decir, el valor de x donde la función $f(x) = 0$.

Gráficamente $f(x) = 0$ quiere decir que la función vale 0 en un valor particular de x , ya sea que la función cruce el eje x o toque el eje x .

Programa 3.1. Método gráfico

```

1 import math
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5
6 def f(x):
7     return x**2-2
8
9 x=np.linspace(-2,2,100)
10 y=f(x)
11
12 fig = plt.figure()
13 ax = plt.gca()
14 ax.set_xlabel('x')
15 ax.set_ylabel('y')
16 plt.plot(x,y)
17 plt.title('$f(x)=x^2-2$')
18 plt.grid()
19
20 ax.spines['left'].set_position('zero')
21 ax.spines['right'].set_color('none')
22 ax.spines['bottom'].set_position('zero')
23 ax.spines['top'].set_color('none')
24
25 plt.show()
26 fig.savefig("grafica_raiz.pdf", bbox_inches='tight')

```

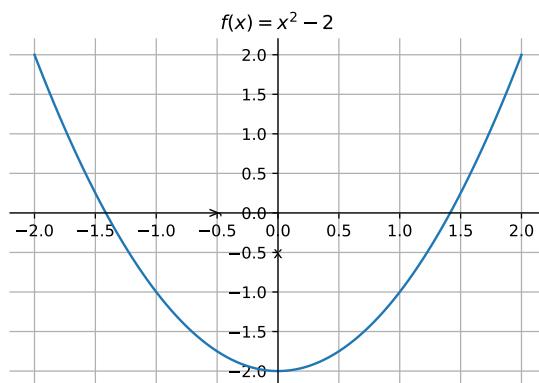


Figura 3.1. Gráfica para localizar la raíz de $f(x) = x^2 - 2$

Observando la gráfica, podemos decir “aproximadamente” que la raíz es 1.4, usted podría decir otra cifra, pero necesitamos otro mecanismo que no dependa del criterio del observador o de la escala con la que se grafica.

Otra desventaja del método gráfico es que sólo podemos “observar” las raíces reales, pero las raíces complejas no, tendríamos que graficar en un plano complejo para localizar las raíces.

Debemos ahora encontrar métodos que, bajo un algoritmo, sistemáticamente encuentren ese valor particular de x .

3.1 Búsqueda incremental

Un método para encontrar el intervalo donde se encuentra la raíz es el **método de búsqueda incremental** que determina un intervalo donde la función cruza el eje x . Al cruzar el eje x entonces la función cambia de signo, ahí es donde se localiza una raíz. La desventaja es que sólo encuentra las raíces reales que cruzan el eje x .

En el método de búsqueda incremental se inicia con un valor de x_0 y un incremento Δx , el valor inicial se incrementa hasta detectar que $f(x)$ cambia de signo.

■ **Ejemplo 3.1 — Búsqueda incremental.** Aplicar el método de búsqueda incremental para localizar el cambio de signo de la función $f(x) = x^2 - 2$. El valor de $x_0 = -2$ y $\Delta x = 0.2$

x	$f(x)$	
-2	2	
-1.8	1.24	
-1.6	0.56	
-1.4	-0.04	Cambio de signo, una raíz se encuentra $x \in [-1.6, -1.4]$
-1.2	-0.56	
-1	-1	
-0.8	-1.36	
-0.6	-1.64	
-0.4	-1.84	
-0.2	-1.96	
0	-2	
0.2	-1.96	
0.4	-1.84	
0.6	-1.64	
0.8	-1.36	
1	-1	
1.2	-0.56	
1.4	-0.04	
1.6	0.56	Cambio de signo, otra raíz se encuentra $x \in [1.4, 1.6]$
1.8	1.24	
2	2	

Cuadro 3.1. Método de búsqueda incremental

En el cuadro 3.1 se localizan dos cambios de signo, uno entre $[-1.6, -1.4]$ y el otro entre $[1.4, 1.6]$. Si el incremento $\Delta x = 10$, entonces no se detecta cambio de signo y no se reporta raíz de la función. Debemos entonces construir algoritmos que aseguren la localización de la raíz de $f(x)$.

Los métodos se clasifican en:

- **Métodos cerrados.** Requieren de dos puntos que **encierren** la raíz x_0 y x_1 . Los dos puntos se pueden obtener del método de búsqueda incremental, ya que la función debe cambiar de signo en el intervalo, para validar eso es suficiente con multiplicar $f(x_0)$ por $f(x_1)$; si el resultado es menor que 0, entonces hay un cambio de signo de la función y la raíz se encuentra en algún lugar de ese intervalo.

Estos métodos sólo localizan las raíces reales que cruzan el eje x .

- **Métodos abiertos.** Requieren de uno o más puntos iniciales. Localizan la raíz de $f(x)$ haciendo aproximaciones del comportamiento de $f(x)$ para estimar el valor de la raíz, si no es satisfactorio se hace la siguiente aproximación hasta localizar la raíz.

Estos métodos localizan las raíces que tocan o cruzan el eje x , también son capaces de localizar las raíces complejas bajo ciertas condiciones.

Dos importantes conceptos que se deben conocer para entender los métodos para resolver ecuaciones no lineales son:

Definición 3.1.1 — Criterio de convergencia. Es la condición de paro que se debe cumplir para que el algoritmo se detenga.

Definición 3.1.2 — Tolerancia. Es el valor máximo que puede tomar $f(x)$ para aceptar a x como la raíz.

Para resolver las ecuaciones no lineales, los métodos siguen un proceso iterativo como se muestra a continuación

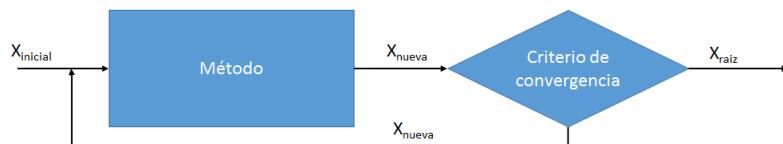


Figura 3.2. Proceso iterativo

Se inicia con un valor supuesto de $x_{inicial}$, que pueden ser uno o más valores, el proceso calcula el nuevo valor de $x_{calculada}$ como una aproximación a la raíz, se valida si cumple el criterio de convergencia, si lo cumple entonces es la raíz $x_{raíz}$, de otra forma, se vuelve a entrar el valor de x_{nueva} . El proceso se repite hasta que se cumpla el criterio de convergencia.

3.2 Métodos cerrados

Definición 3.2.1 — Método cerrados. Los métodos cerrados requieren de dos puntos x_0 y x_1 que encierran la raíz. Se debe cumplir que $f(x_0)f(x_1) < 0$. Sólo localizan las raíces reales que cruzan el eje x .

3.2.1 Bisección, Bolzano o partición binaria

El método de bisección, al ser un método cerrado, requiere de dos puntos x_0 y x_1 que encierran la raíz, el intervalo $[x_0, x_1]$ se parte a la mitad, generando 2 subintervalos $[x_0, x]$ y $[x, x_1]$, en uno de los dos se encuentra la raíz, se selecciona dicho subintervalo y se vuelve a repetir el procedimiento hasta que se cumpla el criterio de convergencia $|f(x)| < \text{tolerancia}$.

Mitad del
intervalo

$$x = \frac{x_i + x_{i+1}}{2} \quad (3.1)$$

A continuación, se muestra el algoritmo del método de la bisección.

Algoritmo 1 Método de bisección

```

1: procedimiento BISECCIÓN( $f, x_0, x_1, tol$ )           ▷  $f$  es la función a evaluar
2:   si  $f(x_0)f(x_1) > 0$  entonces                      ▷ valida el cambio de signo de  $f(x)$ 
3:     retornar No hay cambio de signo
4:   fin si
5:    $cumple \leftarrow \text{falso}$ 
6:   mientras no cumple hacer                         ▷ Criterio de convergencia
7:      $x \leftarrow \frac{x_0 + x_1}{2}$                           ▷ se calcula la mitad de intervalo
8:     si  $f(x_0)f(x) < 0$  entonces                  ▷ valida si la raíz se encuentra en el subintervalo  $[x_0, x]$ 
9:        $x_1 \leftarrow x$ 
10:      sino                                         ▷ sino entonces está en el segundo subintervalo  $[x, x_1]$ 
11:         $x_0 \leftarrow x$ 
12:      fin si
13:       $cumple \leftarrow |f(x)| < tol$ 
14:    fin mientras
15:    retornar  $x$                                      ▷ Raíz
16:  fin procedimiento
```

Revisemos las líneas importantes del algoritmo de bisección:

- **Línea 1.** El procedimiento requiere de 3 parámetros: la función a evaluar f , x_0 y x_1 donde la función cambie de signo, por último la tolerancia tol para validar el criterio de convergencia.
- **Línea 2.** Valida que exista el cambio de signo, si el valor es mayor a cero, quiere decir que $f(x_0)$ y $f(x_1)$ tienen el mismo signo, por lo tanto, no cruzan el eje x y no hay raíz. Si sabemos

que gráficamente sí existe una raíz, entonces se deben modificar los valores de x_0 y x_1 para que el valor de $f(x)$ cambie de signo y pueda pasar la validación.

- **Línea 6.** La estructura **mientras** se ejecuta mientras no se cumpla la condición del criterio de convergencia, el valor de *cumple* se inicializa a *falso* para que entre al ciclo.
- **Línea 7.** Se calcula x como la mitad del intervalo.
- **Línea 8.** Se valida en qué subintervalo se encuentra la raíz en $[x_0, x]$ o en $[x, x_1]$ y se hace el cambio de variable correspondiente.

■ **Ejemplo 3.2 — Método de bisección.** Aplicar el método de bisección a la función $f(x) = x^2 - 2$. El valor de $x_0 = -1.6$ y $x_1 = -1.3$. Los valores de x_0 y x_1 se obtuvieron del método de búsqueda incremental.

Programa 3.2. Método de bisección

```

1 from math import *
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Algoritmo de Biseccion
6 def biseccion(f, x0, x1, tol):
7     if f(x0) * f(x1) > 0:
8         raise ValueError ('La funcion no cambia de signo en este ←
9                         rango')
10
11    sigue = False
12    print('{:~10s} {:~10s} {:~10s} {:~10s} {:~10s} {:~10s}'.\
13          format('x0','x','x1','f(x0)','f(x)','f(x1)'))
14    while (not sigue):
15        x = (x0 + x1) / 2
16        print('{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}\n'.
17              format(x0,x,x1,f(x0),f(x),f(x1)))
18        if f(x0) * f(x) < 0:
19            x1 = x
20        else:
21            x0 = x
22        sigue = abs(f(x)) < tol
23    return x
24
25 # Funcion a evaluar
26 def f(x):
27     return x**2-2
28
29 def main():
30     # valores iniciales
31     x0=-1.6
32     x1=-1.4

```

```

32     tol=1e-4
33     # Llamada al algoritmo
34     raiz=biseccion(f,x0,x1,tol)
35     print('f({:e})={:e}'.format(raiz,f(raiz)))
36
37     x=np.linspace(x0,x1,100)
38     y=f(x)
39
40     fig = plt.figure()
41     ax = plt.gca()
42     plt.plot(x,y)
43     plt.scatter(raiz,f(raiz))
44     plt.text(raiz,f(raiz), ' Raíz '+str(raiz),color='red')
45     plt.grid()
46
47     plt.show()
48     #fig.savefig("biseccion.pdf", bbox_inches='tight')
49
50 if __name__ == "__main__": main()

```

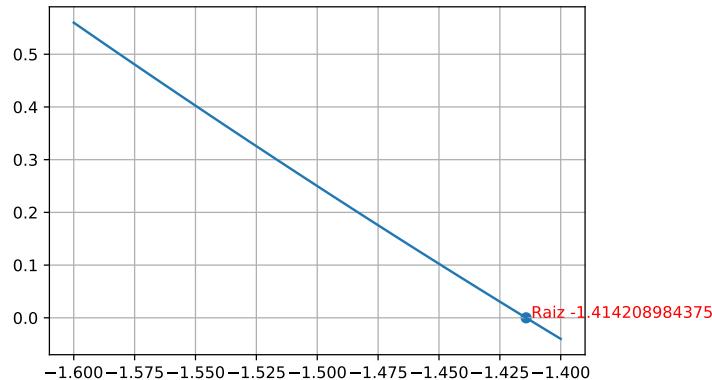


Figura 3.3. Método de bisección

x0	x	x1	f(x0)	f(x)	f(x1)
-1.60000	-1.50000	-1.40000	0.56000	0.25000	-0.04000
-1.50000	-1.45000	-1.40000	0.25000	0.10250	-0.04000
-1.45000	-1.42500	-1.40000	0.10250	0.03062	-0.04000
-1.42500	-1.41250	-1.40000	0.03062	-0.00484	-0.04000
-1.42500	-1.41875	-1.41250	0.03062	0.01285	-0.00484
-1.41875	-1.41562	-1.41250	0.01285	0.00399	-0.00484
-1.41562	-1.41406	-1.41250	0.00399	-0.00043	-0.00484
-1.41562	-1.41484	-1.41406	0.00399	0.00178	-0.00043
-1.41484	-1.41445	-1.41406	0.00178	0.00068	-0.00043
-1.41445	-1.41426	-1.41406	0.00068	0.00013	-0.00043
-1.41426	-1.41416	-1.41406	0.00013	-0.00015	-0.00043
-1.41426	-1.41421	-1.41416	0.00013	-0.00001	-0.00015
$f(-1.414209 \times 10^{-4}) = -1.294851 \times 10^{-5}$					

Por lo tanto, la raíz es -1.414208984375 en el intervalo [-1.6,-1.4].

Observe que existe otra raíz en el intervalo [1.4,1.6], sin embargo, sólo se localiza la raíz negativa; para localizar la raíz positiva, se requiere cambiar los valores iniciales como $x_0 = 1.4$ y $x_1 = 1.6$ para localizar la raíz positiva. Se deja al lector el cambio al programa para localizar la raíz positiva. ■

Como se observa en el ejemplo anterior, se requiere indicar los valores iniciales adecuados para localizar la raíz, si la función tiene otras raíces que cruzan el eje x , se deben cambiar los valores iniciales para encontrarlas. Los métodos cerrados sólo localizan una raíz a la vez, la que está encerrada en el intervalo.

3.2.2 Regla falsa o regula falsi o falsa posición

El método de la regla falsa, al ser un método cerrado, requiere de dos puntos x_0 y x_1 que encierran la raíz. Se hace una interpolación lineal entre $f(x_0)$ y $f(x_1)$, al ser de signos opuestos, entonces cruzan el eje x , por lo tanto, la interpolación lineal entre estos dos puntos también cruzará el eje x en un valor aproximado de la raíz de $f(x)$, al ser una raíz falsa, el método es llamado de la regla falsa o de la falsa posición.

La interpolación lineal entre $f(x_0)$ y $f(x_1)$, de acuerdo con la **ecuación 2.6**, es

$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

El punto donde la interpolación lineal cruza el eje x es la raíz de la interpolación, entonces

$$0 = f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

Despejamos entonces el valor de x , que nos dará la falsa posición de la raíz de $f(x)$.

$$\begin{aligned} -f(x_0)(x_1 - x_0) &= xf(x_1) - xf(x_0) - x_0f(x_1) + x_0f(x_0) \\ x_0f(x_0) - x_1f(x_0) &= xf(x_1) - xf(x_0) - x_0f(x_1) + x_0f(x_0) \\ x_0f(x_1) - x_1f(x_0) &= xf(x_1) - xf(x_0) \\ x &= \frac{x_0f(x_1) - x_1f(x_0)}{f(x_1) - f(x_0)} \end{aligned}$$

Ecuación iterativa de la regla falsa

$$x_{i+1} = \frac{x_i f(x_{i+1}) - x_{i+1} f(x_i)}{f(x_{i+1}) - f(x_i)} \quad (3.2)$$

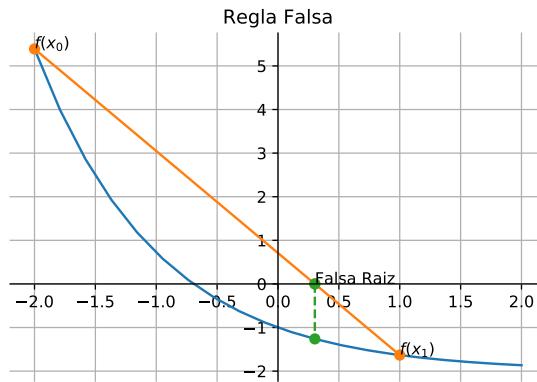


Figura 3.4. Método de la regla falsa

En la gráfica se observa que la intersección de la interpolación lineal con el eje x da una falsa posición de la raíz, esto es una aproximación a la raíz de $f(x)$. La interpolación lineal se aproxima al comportamiento de $f(x)$ en el intervalo $[x_0, x_1]$, por lo tanto, la raíz de la interpolación se aproxima a la raíz de $f(x)$.

El punto x crea dos subintervalos $[x_0, x]$ y $[x, x_1]$, se debe determinar en cuál se encuentra la raíz, el procedimiento es el mismo que en el método de bisección, se debe validar si $f(x_0)f(x) < 0$; entonces el primer subintervalo $[x_0, x]$ contiene la raíz y $x_1 = x$, de lo contrario, la raíz se encuentra en el segundo subintervalo $[x, x_1]$ y $x_0 = x$ y se vuelven a realizar los mismos pasos hasta que se cumpla el criterio de convergencia.

A continuación, se muestra el algoritmo del método de la regla falsa:

Algoritmo 2 Método de regla falsa

```

1: procedimiento REGLAFALSA( $f, x_0, x_1, tol$ )                                 $\triangleright f$  es la función a evaluar
2:   si  $f(x_0)f(x_1) > 0$  entonces                                          $\triangleright$  valida el cambio de signo de  $f(x)$ 
3:     retornar No hay cambio de signo
4:   fin si
5:    $cumple \leftarrow false$ 
6:   mientras no cumple hacer                                               $\triangleright$  Criterio de convergencia
7:      $x \leftarrow \frac{x_0f(x_1) - x_1f(x_0)}{f(x_1) - f(x_0)}$                           $\triangleright$  se calcula la intersección con el eje  $x$ 
8:     si  $f(x_0)f(x) < 0$  entonces                                          $\triangleright$  valida si la raíz se encuentra en el subintervalo  $[x_0, x]$ 
9:        $x_1 \leftarrow x$ 
10:      sino                                                  $\triangleright$  sino entonces está en el segundo subintervalo  $[x, x_1]$ 
11:         $x_0 \leftarrow x$ 
12:      fin si
13:       $cumple \leftarrow |f(x)| < tol$ 
14:    fin mientras
15:    retornar  $x$                                                                 $\triangleright$  Raíz
16:  fin procedimiento
  
```

Revisemos las líneas importantes del algoritmo de la regla falsa:

- **Línea 1.** El procedimiento requiere de 3 parámetros: la función a evaluar f , x_0 y x_1 donde la función cambie de signo; por último, la tolerancia tol para validar el criterio de convergencia.
- **Línea 2.** Valida que exista el cambio de signo, si el valor es mayor a cero, quiere decir que $f(x_0)$ y $f(x_1)$ tienen el mismo signo, por lo tanto, no cruzan el eje x y no hay raíz. Si sabemos que gráficamente sí existe una raíz, entonces se deben modificar los valores de x_0 y x_1 para que el valor de $f(x)$ cambie de signo y pueda pasar la validación.
- **Línea 6.** La estructura **mientras** se ejecuta mientras no se cumpla la condición del criterio de convergencia, el valor de *cumple* se inicializa a *falso* para que entre al ciclo.
- **Línea 7.** Se calcula x como la intersección de la interpolación lineal con el eje x .
- **Línea 8.** Se valida en qué subintervalo se encuentra la raíz en $[x_0, x]$ o en $[x, x_1]$ y se hace el cambio de variable correspondiente.

■ **Ejemplo 3.3 — Método de la regla falsa.** Se tiene un tanque esférico para almacenar agua y se desea calcular la altura del agua h para que el volumen sea 800. Usar la siguiente fórmula que calcula el volumen V en función de la altura h si $R = 10$.

$$V = \pi h^2 \left(\frac{3R - h}{3} \right)$$

Solución

Primero sustituimos el valor de R y V

$$800 = \pi h^2 \left(\frac{3(10) - h}{3} \right)$$

y transformamos la función a la forma $f(x) = 0$

$$f(h) = \pi h^2 \left(\frac{30 - h}{3} \right) - 800 = 0$$

Ahora debemos seleccionar los valores iniciales x_0 y x_1 , de tal manera que la función en $f(x_0)$ y $f(x_1)$ cambien de signo, para ello nos podemos apoyar en el método gráfico para observar el comportamiento de la función.

Observamos que la función tiene tres raíces, ya que es un polinomio de grado 3. Si analizamos un poco el contexto del problema, dice que se trata de un tanque esférico de $R = 10$, por lo tanto, la altura del agua debe estar en el intervalo $h \in [0, 20]$. El polinomio como tal tiene 3 raíces, pero el contexto del problema dice que sólo una raíz nos interesa. Definitivamente, la raíz que se observa cercana a 30 y a -5 no son viables, ya que la altura del agua no puede tener ese valor, entonces observemos más de cerca la gráfica.

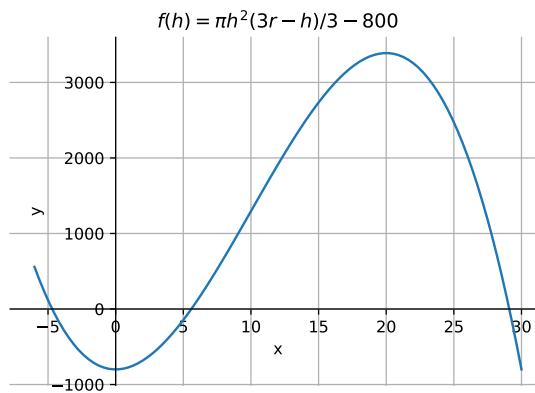


Figura 3.5. Comportamiento de la función del volumen

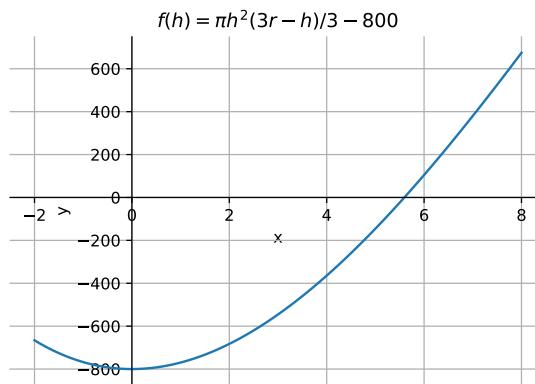


Figura 3.6. Comportamiento de la función del volumen (acercamiento)

Y nos damos cuenta de que tiene dos raíces más, una positiva y otra negativa, definitivamente la raíz negativa tampoco resuelve el problema, porque la altura del agua no puede ser negativa, entonces nos queda la raíz positiva. Para encerrar la raíz positiva que nos interesa, observamos que los valores pueden ser $x_0 = 4$ y $x_1 = 6$, estos valores también concuerdan con el contexto del problema porque la altura debe estar en $h \in [0, 20]$, entonces estos serán los valores iniciales.

Programa 3.3. Método de la regla falsa

```

1 import math
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Algoritmo de Bisección
6 def reglafalsa(f, x0, x1, tol=1e-8):
7     if f(x0) * f(x1) > 0:
8         raise ValueError ('La función no cambia de signo en este ←
9                         rango')

```

```

10     sigue = False
11     print('{:^10s} {:^10s} {:^10s} {:^10s} {:^10s} {:^10s}'.\
12         format('x0','x','x1','f(x0)','f(x)','f(x1)'))
13     while (not sigue):
14         x = ( x0*f(x1) - x1*f(x0) ) / ( f(x1) - f(x0) )
15         print('{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}'\
16             .format(x0,x,x1,f(x0),f(x),f(x1)))
17         if f(x0) * f(x) < 0:
18             x1 = x
19         else:
20             x0 = x
21         sigue = abs(f(x)) < tol
22     return x
23
24 # Funcion a evaluar
25 def f(x):
26     R=10
27     return math.pi*x**2*(3*R-x)/3-800
28
29 def main():
30     # valores iniciales
31     x0=4
32     x1=6
33     tol=1e-4
34     # Llamada al algoritmo
35     raiz=reglafalsa(f,x0,x1,tol)
36     print('f({:e})={:e}'.format(raiz,f(raiz)))
37
38     x=np.linspace(x0,x1,100)
39     y=f(x)
40
41     fig = plt.figure()
42     plt.plot(x,y)
43     plt.scatter(raiz,f(raiz))
44     plt.text(raiz,f(raiz), ' Raiz '+str(raiz),color='red')
45     plt.grid()
46
47     plt.show()
48     #fig.savefig("reglafalsa.pdf", bbox_inches='tight')
49
50 if __name__ == "__main__": main()

```

x0	x	x1	f(x0)	f(x)	f(x1)
4.00000	5.55332	6.00000	-364.36582	-10.49673	104.77868
5.55332	5.59399	6.00000	-10.49673	-0.22226	104.77868
5.59399	5.59485	6.00000	-0.22226	-0.00467	104.77868
5.59485	5.59487	6.00000	-0.00467	-0.00010	104.77868
$f(5.594871 \times 10^0) = -9.816057 \times 10^{-5}$					

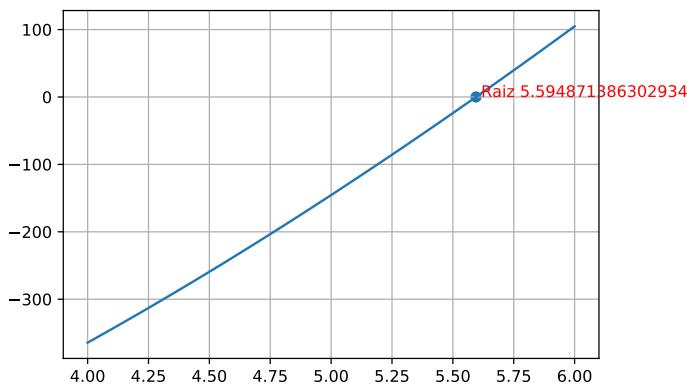


Figura 3.7. Método de la regla falsa

Por lo tanto, la raíz es $h = 5.594871$ y ésa es la altura del agua para que el tanque sea $V = 800$ ■

Se debe considerar siempre el contexto del problema para saber cuál de las múltiples raíces que tiene la función son las que nos interesan. También el contexto del problema nos ayuda para dar los valores iniciales que encierran la raíz que resuelve el problema.

3.3 Métodos abiertos

Definición 3.3.1 — Método abiertos. Los métodos abiertos requieren de uno o más valores iniciales para localizar las raíces de funciones no lineales. Localizan las raíces reales que cruzan o tocan el eje x , y en ciertas condiciones, también las raíces complejas.

3.3.1 Newton-Raphson

El método de Newton-Raphson se apoya en la serie de Taylor (**ecuación 1.7**, página 15) la cual hace una aproximación de $f(x)$

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots$$

Truncamos la serie de Taylor en el segundo término y como deseamos encontrar el valor de x donde $f(x) = 0$, entonces igualamos la serie truncada a 0.

$$0 = f(x_0) + f'(x_0)(x - x_0)$$

Esto nos daría una aproximación a la raíz de $f(x)$, para conocer el valor de x simplemente la podemos despejar de la ecuación.

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

En su forma iterativa

Ecuación iterativa de
Newton-Raphson

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (3.3)$$

x_{i+1} es una aproximación a la raíz, basta con validar el criterio de convergencia y si no se cumple se vuelve a iterar con un nuevo valor de x . Note que sólo se requiere de un valor inicial x_0 . El método de Newton-Raphson puede convertirse en un ciclo infinito en algunos casos, entonces se debe incluir otra condición de paro que es el número máximo de iteraciones. Otra consideración que se debe tener en cuenta es que la derivada $f'(x_0)$ en algún valor particular de x_0 sea cero, en ese caso la división $\frac{f(x_0)}{f'(x_0)}$ no se podría calcular y, por lo tanto, el valor calculado de x tampoco, en esos casos se puede incrementar x en un delta para que salga de ese punto y pueda continuar con la siguiente iteración.

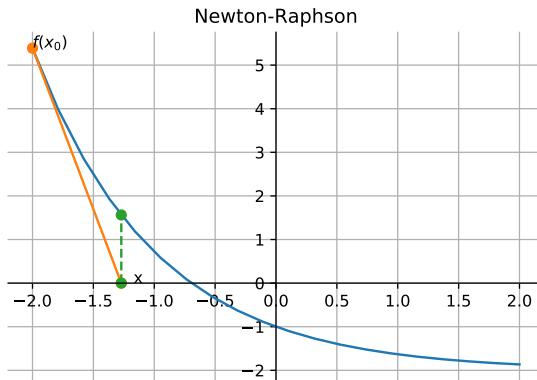


Figura 3.8. Método de Newton-Raphson

El método de Newton-Raphson suele converger rápido debido a que toma en cuenta la derivada y generalmente x apunta a un valor cercano de la raíz de $f(x)$. En otros casos, la derivada puede llevar el valor de x lejos de la raíz, incluso puede divergir. Como el algoritmo prevé un número máximo de iteraciones.

■ **Ejemplo 3.4 — Método de Newton-Raphson.** Fluye aire a una temperatura de 25°C y 1 atm a través de un tubo de 4 mm de diámetro a una velocidad promedio de 50 m/s . La rugosidad es de $\varepsilon = 0.0015\text{mm}$ y $Re = 13743$. Calcular el factor de fricción usando la ecuación de Colebrook.

$$\frac{1}{\sqrt{f}} = -2.0 \log \left(\frac{\varepsilon/D}{3.7} + \frac{2.51}{Re\sqrt{f}} \right)$$

Solución

La ecuación se expresa de la forma $f(x) = 0$

$$F = \frac{1}{\sqrt{f}} + 2.0 \log \left(\frac{\varepsilon/D}{3.7} + \frac{2.51}{Re\sqrt{f}} \right) = 0$$

Algoritmo 3 Método de Newton-Raphson

```

1: procedimiento NEWTONRAPHSON( $f, f', x_0, imax, tol$ )            $\triangleright f$  es la función a evaluar
2:    $k \leftarrow 0$ 
3:    $cumple \leftarrow \text{falso}$ 
4:   mientras no cumple &  $k < imax$  hacer                       $\triangleright$  Criterio de convergencia
5:     si  $f'(x_0) \neq 0$  entonces                                 $\triangleright$  valida si la derivada es distinta de 0
6:        $x \leftarrow x_0 - \frac{f(x_0)}{f'(x_0)}$                           $\triangleright$  se calcula la nueva x
7:     sino
8:        $x \leftarrow x_0 + tol$                                           $\triangleright$  se calcula la nueva x
9:     fin si
10:     $cumple \leftarrow |f(x)| < tol$ 
11:     $k \leftarrow k + 1$ 
12:  fin mientras
13:  retornar  $x$                                                $\triangleright$  Raíz
14: fin procedimiento

```

Y su derivada es:

$$F' = \frac{-1}{2\sqrt{f^3}} - \frac{2.51}{Re\sqrt{f^3} \frac{2.51}{Re\sqrt{f}} + \frac{\varepsilon/D}{3.7}}$$

Programa 3.4. Método de Newton-Raphson

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import math
5 import cmath
6
7 def newtonRaphson(f ,df ,x ,imax=100 ,tol=1e-8):
8     cumple=False
9     print('{:^10s} {:^10s} {:^10s}'.\
10         format('x','f(x)', 'df(x)'))
11    k=0
12    while (not cumple and k<imax):
13        if df(x)!=0:
14            x=x-f(x)/df(x)
15        else:
16            x=x+tol
17        print('{:10.5f} {:10.5f} {:10.5f}'.\
18            format(x,f(x),df(x)))
19        cumple=abs(f(x))<=tol
20        k+=1
21    if k<imax:
22        return x
23    else:
24        raise ValueError ('La función no converge')

```

```

25
26 # Funcion a evaluar
27 def f(x):
28     e=0.0015
29     D=4
30     Re=13743
31     return 1/np.sqrt(x)+2*np.log10(e/D/3.7+2.51/Re/np.sqrt(x))
32 # Derivada
33 def df(x):
34     e=0.0015
35     D=4
36     Re=13743
37     return -1/(2*x** (3/2)) - \
38             2.51/(Re*x** (3/2)* (2.51/(Re*np.sqrt(x)) + e/D/2.51))
39
40 def main():
41     # valores iniciales
42     x0=0.01
43     # Llamada al algoritmo
44     raiz=newtonRaphson(f,df,x0)
45     print('f({:e})={:e}'.format(raiz,f(raiz)))
46
47     x=np.linspace(0.0001,0.05,100)
48     y=f(x)
49
50     fig = plt.figure()
51     plt.plot(x,y)
52     plt.title('Factor de fricción de Colebrook')
53     plt.scatter(raiz,f(raiz))
54     plt.text(raiz,f(raiz), ' Raíz '+str(raiz),color='red')
55     plt.grid()
56
57     plt.show()
58     fig.savefig("newtonraphson.pdf", bbox_inches='tight')
59
60 if __name__ == "__main__": main()

```

x	f(x)	df(x)
0.01771	1.85024	-262.98659
0.02475	0.55879	-164.21399
0.02815	0.11094	-137.08561
0.02896	0.01583	-131.75483
0.02908	0.00204	-130.99404
0.02910	0.00026	-130.89614
0.02910	0.00003	-130.88375
0.02910	0.00000	-130.88219
0.02910	0.00000	-130.88199
0.02910	0.00000	-130.88197
0.02910	0.00000	-130.88196
f(2.909962e-02)=8.266604e-09		

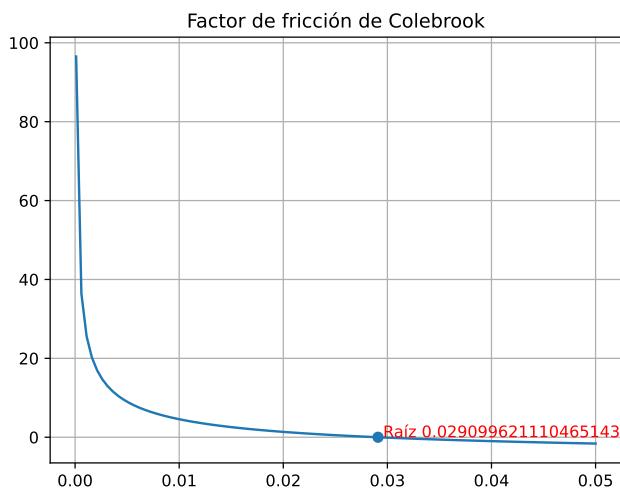


Figura 3.9. Método de Newton-Raphson

Por lo tanto, el factor de fricción es $f = 2.909948e - 02$

El siguiente programa es la implementación del método de Newton-Raphson usando la biblioteca *optimize* de *scipy*. El método Newton sólo recibe la función $f(x)$ y no requiere de la derivada $f'(x)$, necesariamente.

Programa 3.5. Método de Newton-Raphson de *optimize*

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 from scipy import optimize
4 import matplotlib.pyplot as plt
5 import math
6 import cmath
7
8 # Funcion a evaluar
9 def f(x):
10     e=0.0015
11     D=4
12     Re=13743
13     return 1/np.sqrt(x)+2*np.log10(e/D/3.7+2.51/Re/np.sqrt(x))
14
15 def main():
16     # valores iniciales
17     x0=0.01
18     # Llamada al algoritmo
19     raiz=optimize.newton(f,x0)
20     print('f({:e})={:e}'.format(raiz,f(raiz)))
21
22     x=np.linspace(0.0001,0.05,100)
23     y=f(x)
24

```

```

25     fig = plt.figure()
26     plt.plot(x,y)
27     plt.title('Factor de fricción de Colebrook')
28     plt.scatter(raiz,f(raiz))
29     plt.text(raiz,f(raiz), ' Raíz '+str(raiz),color='red')
30     plt.grid()
31
32     plt.show()
33     fig.savefig("newtonraphsonopt.pdf", bbox_inches='tight')
34
35 if __name__ == "__main__": main()

```

$f(2.909962e-02) = 1.776357e-15$

Se obtiene el mismo resultado $f = 2.909962e-02$

■

3.3.2 Newton-Raphson modificado

El criterio de convergencia para aceptar un valor de x como el valor de la raíz de $f(x)$ puede dar como resultado valores incorrectos de una raíz. Esto se hace muy evidente para funciones que tienen un comportamiento casi tangencial con el eje x , ya que el criterio de convergencia se puede cumplir en valores de x que están todavía lejos de la raíz.

Un ejemplo muy claro de este tipo de funciones son los polinomios con raíces múltiples, por ejemplo

$$f(x) = (x - 2)(x - 2)(x - 4)$$

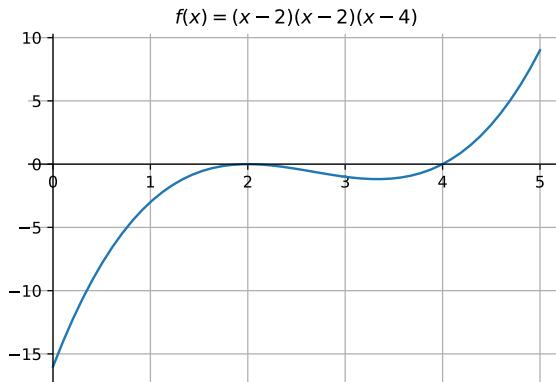
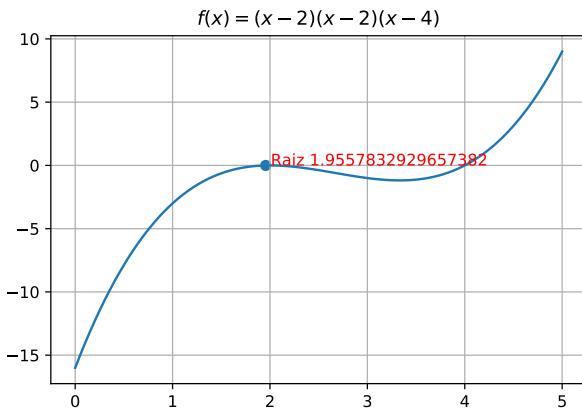


Figura 3.10. $f(x)=(x-2)(x-2)(x-4)$

La función expresada de esta manera se identifica como un polinomio de grado 3, por lo tanto, tiene 3 raíces, sus valores son 2 y 4, sólo que 2 es una raíz múltiple. Este tipo de polinomios tiene un comportamiento que se acerca mucho a cero antes y después de 2, por lo tanto, provoca que el criterio de convergencia se cumpla antes o después que el valor real de la raíz.

Figura 3.11. Newton-Raphson $f(x)=(x-2)(x-2)(x-4)$

Revisemos la ecuación iterativa de Newton-Raphson

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Para llegar a la convergencia $f(x)$ debe ser casi cero, de acuerdo con el criterio de convergencia $|f(x)| < \text{tolerancia}$, si $f(x)$ es casi cero entonces el término $\frac{f(x_i)}{f'(x_i)}$ es casi cero. En general, $f(x)$ llega a cero antes que su derivada $f'(x)$, por lo que se cumple el criterio y se detienen los cálculos.

Se pretende que el término $\frac{f(x_i)}{f'(x_i)}$ sea cero, no sólo el valor de $f(x)$, entonces apliquemos el algoritmo de Newton-Raphson a ese término:

$$g(x) = \frac{f(x_i)}{f'(x_i)}$$

La ecuación iterativa para $g(x)$ es

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

donde $g'(x)$ es la derivada de $g(x)$

$$g'(x) = \frac{f'(x)f'(x) - f(x)f''(x)}{(f'(x))^2}$$

sustituyendo $g(x)$ y $g'(x)$ en la ecuación iterativa de Newton finalmente tenemos

Ecuación iterativa de Newton modificado

$$x_{i+1} = x_i - \frac{f(x_i)f'(x_i)}{(f'(x_i))^2 - f(x_i)f''(x_i)} \quad (3.4)$$

Observamos que esta ecuación iterativa no sólo requiere de $f'(x)$, sino que también de $f''(x)$. A continuación, se presenta el algoritmo del método de Newton modificado.

Algoritmo 4 Método de Newton modificado

```

1: procedimiento NEWTONMODIFICADO( $f, f', f'', x_0, imax, tol$ )       $\triangleright f$  es la función a evaluar
2:    $k \leftarrow 0$ 
3:    $cumple \leftarrow \text{falso}$ 
4:   mientras no cumple &  $k < imax$  hacer                                 $\triangleright$  Criterio de convergencia
5:      $x \leftarrow x_0 - \frac{f(x_0)f'(x_0)}{(f'(x_0))^2 - f(x_0)f''(x_0)}$            $\triangleright$  se calcula la nueva x
6:      $cumple \leftarrow |f(x)| < tol$ 
7:      $k \leftarrow k + 1$ 
8:   fin mientras
9:   retornar  $x$                                                         $\triangleright$  Raíz
10:  fin procedimiento

```

■ **Ejemplo 3.5 — Método de Newton modificado.** Aplicar el método de Newton modificado a la función

$$f(x) = (x - 2)(x - 2)(x - 4)$$

Tomar como valor inicial $x = 1$

Solución

la derivada es:

$$f'(x) = (x - 4)(2x - 4) + (x - 2)^2$$

la segunda derivada es:

$$f''(x) = 2(3x - 8)$$

Programa 3.6. Método de Newton modificado

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import math
5 import cmath
6
7 def newtonRaphsonM(f ,df ,d2f ,x ,imax=100 ,tol=1e-8):
8     cumple=False
9     print('{:^10s} {:^10s} {:^10s} '.\
10         format('x','f(x)','df(x)','d2f(x)'))
11    k=0
12    while (not cumple and k<imax):
13        x=x-(f(x)*df(x))/(df(x)**2-f(x)*d2f(x))
14        print('{:10.5f} {:10.5f} {:10.5f}'.\

```

```
15         format(x,f(x),df(x),d2f(x)))
16     cumple=abs(f(x))<=tol
17     k+=1
18     if k<imax:
19         return x
20     else:
21         raise ValueError ('La función no converge')
22
23 # Funcion a evaluar
24 def f(x):
25     return (x-2)*(x-2)*(x-4)
26 # Derivada
27 def df(x):
28     return (x - 4)*(2*x - 4) + (x - 2)**2
29 # Segunda derivada
30 def d2f(x):
31     return 2*(3*x - 8)
32
33 def main():
34     # valores iniciales
35     x0=1
36     # Llamada al algoritmo
37     raiz=newtonRaphsonM(f,df,d2f,x0)
38     print('f({:e})={:e}'.format(raiz,f(raiz)))
39
40     x=np.linspace(1,5)
41     y=f(x)
42
43     fig = plt.figure()
44     plt.plot(x,y)
45     plt.title('$f(x)=(x-2)(x-2)(x-4)$')
46     plt.scatter(raiz,f(raiz))
47     plt.text(raiz,f(raiz), ' Raíz '+str(raiz),color='red')
48     plt.grid()
49
50     plt.show()
51     fig.savefig("newtonraphsonm.pdf", bbox_inches='tight')
52
53 if __name__ == "__main__": main()
```

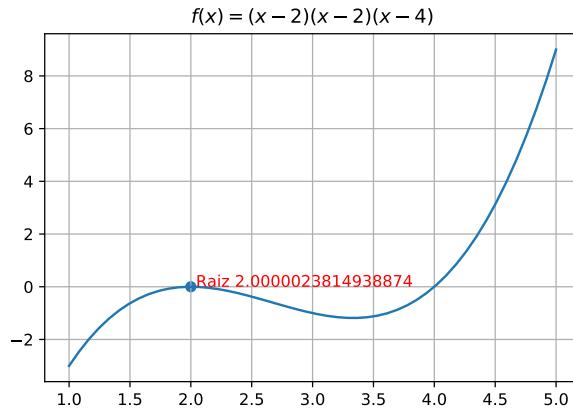


Figura 3.12. Método de Newton modificado

x	f(x)	df(x)	d2f(x)
2.10526	-0.02099	-0.38781	
2.00308	-0.00002	-0.01230	
2.00000	-0.00000	-0.00001	
$f(2.000002e+00) = -1.134301e-11$			

El valor de la raíz es un mejor resultado que el método de Newton-Raphson y con menos iteraciones.

El siguiente programa es la implementación del método de Haley usando la biblioteca *optimize* de *scipy*. El método Haley recibe la función $f(x)$ la derivada $f'(x)$ y la segunda derivada $f''(x)$

Programa 3.7. Método de Haley de *optimize*

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 from scipy import optimize
4 import matplotlib.pyplot as plt
5 import math
6 import cmath
7
8 # Funcion a evaluar
9 def f(x):
10     return (x-2)*(x-2)*(x-4)
11 # Derivada
12 def df(x):
13     return (x - 4)*(2*x - 4) + (x - 2)**2
14 # Segunda derivada
15 def d2f(x):
16     return 2*(3*x - 8)
17
18 def main():
19     # valores iniciales
20     x0=1
21     # Llamada al algoritmo

```

```

22     raiz=optimize.newton(f,x0,fprime=df,fprime2=d2f)
23     print('f({:e})={:e}'.format(raiz,f(raiz)))
24
25     x=np.linspace(1,5,100)
26     y=f(x)
27
28     fig = plt.figure()
29     plt.plot(x,y)
30     plt.title('$f=(x-2)(x-2)(x-4)$')
31     plt.scatter(raiz,f(raiz))
32     plt.text(raiz,f(raiz), ' Raiz '+str(raiz),color='red')
33     plt.grid()
34
35     plt.show()
36     fig.savefig("newtonraphsonoptm.pdf", bbox_inches='tight')
37
38 if __name__ == "__main__": main()

```

```
f(2.000000e+00)=-0.000000e+00
```

Se obtiene un resultado muy similar.

3.3.3 Secante

Uno de los inconvenientes de los métodos de Newton para ecuaciones no lineales es el uso de la derivada de $f(x)$, si hablamos de una función complicada donde sea muy difícil derivar, el método de Newton no es el recomendable. La alternativa es usar una aproximación a la derivada que sea más sencilla de obtener.

$$f'(x) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Sustituimos la aproximación de $f'(x)$ en la ecuación iterativa de Newton y tenemos

$$x_{i+2} = x_{i+1} - \frac{f(x_{i+1})}{\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}}$$

Ecuación iterativa de la secante

$$x_{i+2} = x_{i+1} - \frac{f(x_{i+1})(x_{i+1} - x_i)}{f(x_{i+1}) - f(x_i)} \quad (3.5)$$

Observe que se requieren de dos puntos x_0, x_1 para obtener la aproximación a la derivada, esto no quiere decir que los dos puntos deban encerrar a la raíz, son simplemente para poder hacer la aproximación a la derivada.

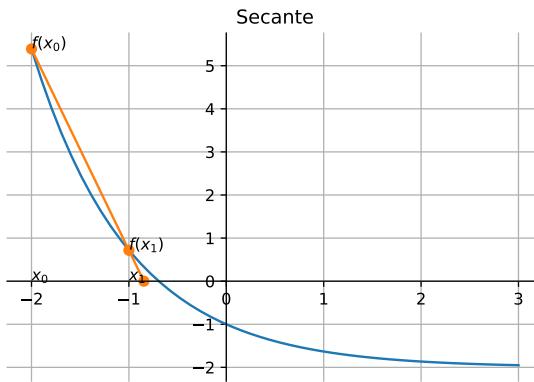


Figura 3.13. Método de la secante

El método de la secante se deriva del método de Newton-Raphson y es una alternativa al mismo cuando la función es difícil de derivar. Los dos puntos crean una línea secante a la función, es por eso su nombre.

Algoritmo 5 Método de la secante

```

1: procedimiento SECANTE( $f, x_0, x_1, imax, tol$ )            $\triangleright f$  es la función a evaluar
2:    $k \leftarrow 0$ 
3:    $cumple \leftarrow \text{falso}$ 
4:   mientras no cumple &  $k < imax$  hacer                    $\triangleright$  Criterio de convergencia
5:      $x \leftarrow x_1 - \frac{f(x_1)(x_1-x_0)}{f(x_1)-f(x_0)}$        $\triangleright$  se calcula la nueva x
6:      $cumple \leftarrow |f(x)| < tol$ 
7:      $x_0 \leftarrow x_1$ 
8:      $x_1 \leftarrow x$ 
9:      $k \leftarrow k + 1$ 
10:    fin mientras
11:    retornar  $x$                                           $\triangleright$  Raíz
12:  fin procedimiento
  
```

■ **Ejemplo 3.6 — Método de la secante.** La ecuación de estado de Redlich-Kwong es la siguiente

$$p = \frac{RT}{v-b} - \frac{a}{v(v+b)\sqrt{T}}$$

Expresado en la forma $f(x) = 0$

$$f(v) = \frac{RT}{v-b} - \frac{a}{v(v+b)\sqrt{T}} - p = 0$$

Donde R es la constante universal de los gases 0.158 kJ/kg, T es la temperatura absoluta (K), p es la presión absoluta (kPa) y v es el volumen por kilogramo del gas (m^3/kg). Los parámetros a y b se calculan como:

$$a = 0.427 \frac{R^2 T_c^{2.5}}{P_c}$$

$$b = 0.0866 \frac{T_c}{p_c}$$

Donde $p_c = 4600$ kPa y $T_c = 191$ K, calcular el volumen de metano a una temperatura de 270 K y una presión de 6,000 kPa.

Solución

Programa 3.8. Método de la secante

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4 import cmath
5
6 def secante(f ,x0 ,x1 ,imax=100 ,tol=1e-8):
7     cumple=False
8     print('{:^10s} {:^10s} {:^10s}'.\
9           format('x0','x1','x','f(x0)','f(x1)','f(x)'))
10    k=0
11    while (not cumple and k<imax):
12        x=x1-f(x1)*(x0-x1)/(f(x0)-f(x1))
13        print('{:10.5f} {:10.5f} {:10.5f}'.\
14              format(x0,x1,x,f(x0),f(x1),f(x)))
15        x0=x1
16        x1=x
17        cumple=abs(f(x))<tol
18        k+=1
19    if k<imax:
20        return x
21    else:
22        raise ValueError ('La función no converge')
23
24 # Funcion a evaluar
25 def f(v):
26     R=0.158
27     Pc=4600
28     Tc=191
29     a=0.427*R**2*Tc**(.5)/Pc
30     b=0.0866*Tc/Pc
31     p=6000
32     T=270
33     return R*T/(v-b)-a/(v*(v+b)*np.sqrt(T))-p
34
35 def main():
36     # valores iniciales
37     x0=0.006
38     x1=0.007
39     # Llamada al algoritmo
40     raiz=secante(f ,x0 ,x1 ,100 ,1e-4)
41     print('f({:e})={:e}'.format(raiz,f(raiz)))

```

```

42
43     x=np.linspace(0.005,0.02,100)
44     y=f(x)
45
46     fig = plt.figure()
47     plt.plot(x,y)
48     plt.title('Volumen de metano')
49     plt.scatter(raiz,f(raiz))
50     plt.text(raiz,f(raiz), ' Raíz '+str(raiz),color='red')
51     plt.grid()
52
53     plt.show()
54     fig.savefig("secante.pdf", bbox_inches='tight')
55
56 if __name__ == "__main__": main()

```

x0	x1	x	f(x0)	f(x1)	f(x)
0.00600	0.00700	0.00813	10508.85629	5572.87654	2664.4657
0.00700	0.00813	0.00916	5572.87654	2664.46574	1054.06532
0.00813	0.00916	0.00984	2664.46574	1054.06532	293.73962
0.00916	0.00984	0.01010	1054.06532	293.73962	43.03442
0.00984	0.01010	0.01015	293.73962	43.03442	2.04305
0.01010	0.01015	0.01015	43.03442	2.04305	0.01491
0.01015	0.01015	0.01015	2.04305	0.01491	0.00001
$f(1.014907e-02) = 5.203538e-06$					

El volumen de metano para esas condiciones es $v = 1.014907e-02$.

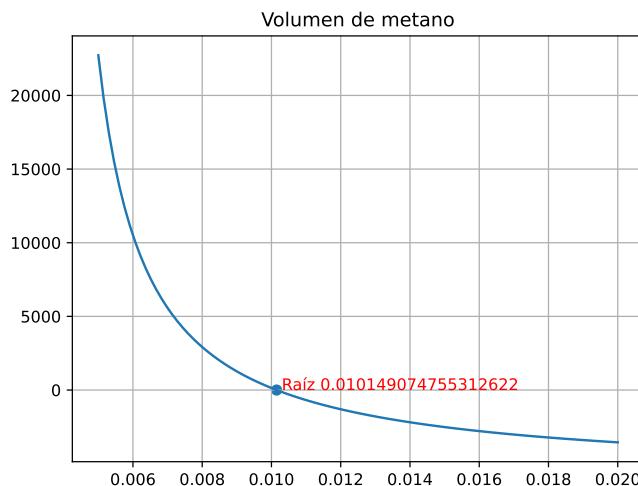


Figura 3.14. Método de la secante

3.3.4 Muller

Los métodos de Newton-Raphson y de la secante obtienen la aproximación a la raíz con una línea recta tangente a la función o secante a la función. El método de Muller obtiene la aproximación con una parábola, la aproximación a la raíz se obtiene calculando la raíz del polinomio cuadrático que es relativamente fácil de calcular.

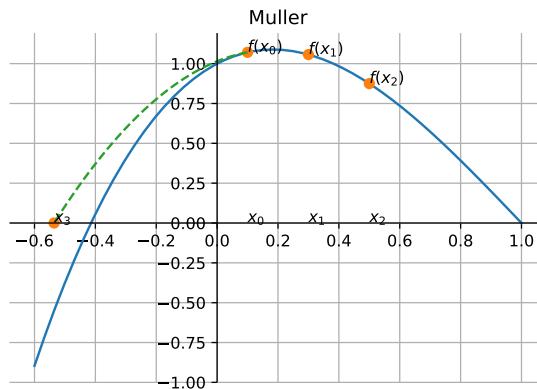


Figura 3.15. Método de Muller

El método de Muller requiere de tres puntos iniciales x_0, x_1, x_2 para hacer una interpolación cuadrática y obtener x_3 calculando la raíz del polinomio cuadrático, la cual es la aproximación a la raíz. Se prueba si x_3 cumple con el criterio de convergencia para detener los cálculos, de lo contrario, hace el cambio de variables para la siguiente iteración.

Para construir el polinomio de segundo grado se usa la interpolación de Newton, por lo tanto, es necesario calcular la primera y segunda diferencia. El polinomio de segundo grado tiene dos raíces, entonces se debe seleccionar una de las dos (la de mayor denominador), si la raíz seleccionada es compleja, entonces la raíz de $f(x)$ es compleja. El método de Muller encuentra raíces complejas o reales.

Se usa la interpolación de Newton para los tres puntos iniciales.

$$f(x) = f(x_0) + f[x_1, x_0](x - x_0) + f[x_2, x_1, x_0](x - x_0)(x - x_1)$$

Donde $f[x_1, x_0]$ y $f[x_2, x_1, x_0]$ son la primera y segunda diferencia respectivamente. La interpolación cuadrática se iguala a cero y se desarrolla para llegar a la forma del polinomio:

$$f(x) = c + bx + ax^2 = 0$$

Interpolación cuadrática igual a cero

$$0 = f(x_0) + f[x_1, x_0](x - x_0) + f[x_2, x_1, x_0](x - x_0)(x - x_1)$$

desarrollamos

$$0 = f(x_0) + xf[x_1, x_0] - x_0f[x_1, x_0] + \\ x^2f[x_2, x_1, x_0] - xx_1f[x_2, x_1, x_0] - xx_0f[x_2, x_1, x_0] + x_0x_1f[x_2, x_1, x_0]$$

agrupamos términos

$$0 = f(x_0) - x_0f[x_1, x_0] + x_0x_1f[x_2, x_1, x_0] + \\ x(f[x_1, x_0] - x_0f[x_2, x_1, x_0] - x_1f[x_2, x_1, x_0]) + \\ x^2f[x_2, x_1, x_0]$$

factorizamos $-f[x_2, x_1, x_0]$ de $-x_0f[x_2, x_1, x_0] - x_1f[x_2, x_1, x_0]$

$$0 = f(x_0) - x_0f[x_1, x_0] + x_0x_1f[x_2, x_1, x_0] + \\ x(f[x_1, x_0] - f[x_2, x_1, x_0](x_0 + x_1)) + \\ x^2f[x_2, x_1, x_0]$$

Identificamos los coeficientes del polinomio $c + bx + ax^2$

$$c = f(x_0) + x_0(x_1f[x_2, x_1, x_0] - f[x_1, x_0]) \\ b = f[x_1, x_0] - f[x_2, x_1, x_0](x_0 + x_1) \\ a = f[x_2, x_1, x_0]$$

Como el valor de $a = f[x_2, x_1, x_0]$, entonces lo sustituimos

$$a = f[x_2, x_1, x_0] \\ b = f[x_1, x_0] - a(x_0 + x_1) \\ c = f(x_0) + x_0(ax_1 - f[x_1, x_0])$$

El siguiente paso es calcular la raíz del polinomio, para ello utilizamos la fórmula general

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La fórmula general es susceptible a errores de redondeo si el radical $b^2 - 4ac$ es pequeño, si el radical se encuentra en el denominador, entonces el valor pequeño hace que el cálculo de x sea un valor mayor, se consideran más dígitos y el error de redondeo disminuye, hagamos los siguientes cambios

$$\left(\frac{-b - \sqrt{b^2 - 4ac}}{2a} \right) \left(\frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} \right) = \left(\frac{(-b)^2 - (b^2 - 4ac)}{2a(-b + \sqrt{b^2 - 4ac})} \right) \\ = \left(\frac{4ac}{2a(-b + \sqrt{b^2 - 4ac})} \right) = \left(\frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \right)$$

$$x_3 = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

Se debe seleccionar el denominador más grande que resulte de $(-b + \sqrt{b^2 - 4ac})$ o $(-b - \sqrt{b^2 - 4ac})$ para calcular x_3 como la aproximación a la raíz, se valida el criterio de convergencia, si no se cumple, entonces se repiten los cálculos haciendo el cambio de puntos iniciales por los nuevos. Para algunos casos x_3 puede ser un valor complejo por el término $\sqrt{b^2 - 4ac}$, de ser así, el método encuentra las raíces complejas de $f(x)$.

La aproximación de la raíz x_3 se logra con el polinomio cuadrático, en lugar de una línea recta como sucede en los métodos de Newton-Raphson y de la secante, la línea recta puede llevar a un punto lejano de la raíz que deseamos encontrar, incluso a un lugar donde la función no existe. La aproximación Muller, al tener un comportamiento cuadrático, su proximidad con la función es conveniente.

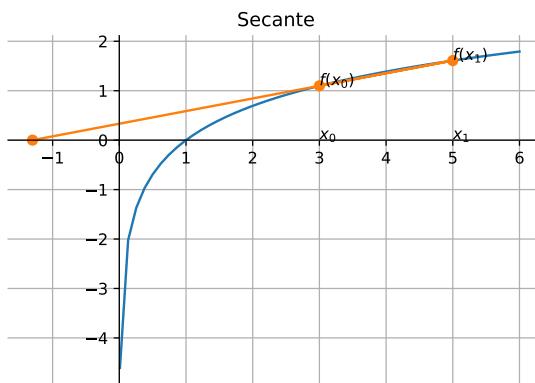


Figura 3.16. Método de la secante

En la figura 3.16 observamos, en este caso, que la aproximación de la raíz cae en un punto donde la función no existe y se detienen los cálculos sin encontrar la raíz.

Para aplicar un método en particular, se deben tomar en cuenta, tanto las características del método, como el comportamiento de la función. Los valores iniciales deben estar dentro del dominio de la función así como las nuevas x .

El método de la secante utiliza una línea recta para hacer la siguiente aproximación, si la función no tiene ese comportamiento, entonces la nueva x puede ser una mala aproximación.

Tenemos algunas opciones para corregir el problema, una es cambiar los valores iniciales de tal manera que la nueva x siga dentro del dominio de $f(x)$. La otra opción es cambiar de método.

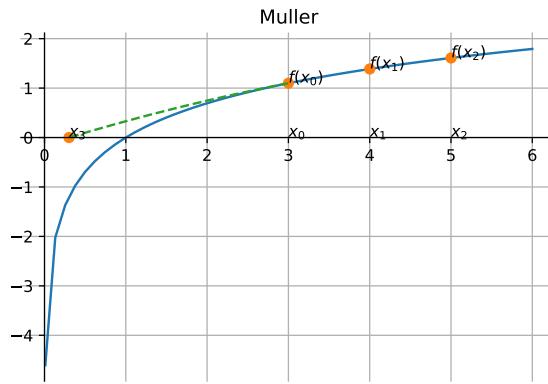


Figura 3.17. Método de Muller

En la figura 3.17 observamos que el método de Muller alcanza una estimación más cercana a la raíz, debido a que la aproximación la obtiene por medio de un polinomio de segundo grado.

Algoritmo 6 Método de Muller

```

1: procedimiento MULLER( $f, x_0, x_1, x_2, imax, tol$ )            $\triangleright f$  es la función a evaluar
2:    $k \leftarrow 0$ 
3:    $cumple \leftarrow \text{falso}$ 
4:   mientras no cumple &  $k < imax$  hacer                       $\triangleright$  Criterio de convergencia
5:      $d_1 \leftarrow (f(x_1) - f(x_0))/(x_1 - x_0)$ 
6:      $d_2 \leftarrow (f(x_2) - f(x_1))/(x_2 - x_1)$ 
7:      $d_3 \leftarrow (d_2 - d_1)/(x_2 - x_0)$ 
8:      $a \leftarrow d_3$ 
9:      $b \leftarrow d_1 - a(x_0 + x_1)$ 
10:     $c \leftarrow f(x_0) + x_0(ax_1 - d_1)$ 
11:     $den_1 \leftarrow -b + \sqrt{b^2 - 4ac},$ 
12:     $den_2 \leftarrow -b - \sqrt{b^2 - 4ac}$ 
13:    si  $|den_1| > |den_2|$  entonces
14:       $x_3 \leftarrow 2c/den_1$ 
15:    sino
16:       $x_3 \leftarrow 2c/den_2$ 
17:    fin si
18:     $cumple \leftarrow |f(x_3)| < tol$ 
19:     $x_0 \leftarrow x_1, x_1 \leftarrow x_2, x_2 \leftarrow x_3$ 
20:     $k \leftarrow k + 1$ 
21:  fin mientras
22:  retornar  $x_3$                                                $\triangleright$  Raiz
23: fin procedimiento
  
```

■ **Ejemplo 3.7 — Método de Muller.** Obtener la raíz de la función

$$f(x) = \cos(x) - x^2$$

Los puntos iniciales son $x_0 = 1, x_1 = 2, x_2 = 3$.

Solución

Programa 3.9. Método de Muller

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4 import cmath
5
6 def muller(f,x0,x1,x2,imax=100,tol=1e-8):
7     cumple=False
8     print('{:^10s} {:^10s} {:^10s} {:^10s} {:^10s} {:^10s} {:^10s} \n'
9           '{:^10s}'.\
10          format('x0','x1','x2','x3','f(x0)','f(x1)','f(x2)','f(x3)'))
11    k=0
12    while (not cumple and k<imax):
13        d1=(f(x1)-f(x0))/(x1-x0)
14        d2=(f(x2)-f(x1))/(x2-x1)
15        d3=(d2-d1)/(x2-x0)
16        a=d3
17        b=d1-a*(x0+x1)
18        c=f(x0)+x0*(a*x1-d1)
19        den1=-b+np.sqrt(b**2-4*a*c)
20        den2=-b-np.sqrt(b**2-4*a*c)
21        if abs(den1)>abs(den2):
22            x3=2*c/den1
23        else:
24            x3=2*c/den2
25
26        print(\n
27              '{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}\n'
28              'f} {:10.5f} {:10.5f}'.\
29             format(x0,x1,x2,x3,f(x0),f(x1),f(x2),f(x3)))
30        x0=x1
31        x1=x2
32        x2=x3
33        cumple=abs(f(x3))<tol
34        k+=1
35    if k<imax:
36        return x3
37    else:
38        raise ValueError ('La funcion no converge')
39
40 # Funcion a evaluar
41 def f(x):
42     return np.cos(x)-x**2
43

```

```

42 def main():
43     # valores iniciales
44     x0=1
45     x1=2
46     x2=3
47     # Llamada al algoritmo
48     raiz=muller(f,x0,x1,x2,100,1e-4)
49     print('f({:e})={:e}'.format(raiz,f(raiz)))
50
51     x=np.linspace(-3,3,100)
52     y=f(x)
53
54     fig = plt.figure()
55     plt.plot(x,y)
56     plt.title('$f(x)=\cos(x) - x^2$')
57     plt.scatter(raiz,f(raiz))
58     plt.text(raiz,f(raiz), ' Raiz '+str(raiz),color='red')
59     plt.grid()
60
61     plt.show()
62     #fig.savefig("muller.pdf", bbox_inches='tight')
63
64 if __name__ == "__main__": main()

```

x0	x1	x2	x3	f(x0)	f(x1)	↔
	f(x2)	f(x3)				
1.00000	2.00000	3.00000	0.84803	-0.45970	-4.41615	↔
	-9.98999	-0.05768				
2.00000	3.00000	0.84803	0.82748	-4.41615	-9.98999	↔
	-0.05768	-0.00800				
3.00000	0.84803	0.82748	0.82414	-9.98999	-0.05768	↔
	-0.00800	-0.00003				
$f(8.241432e-01)=-2.593213e-05$						

La raíz de $f(x) = \cos(x) - x^2$ es $8.241432e-01$. ■

3.3.5 Punto fijo

Los métodos anteriores han propuesto estrategias para encontrar el valor de x donde $f(x) = 0$. El método de punto fijo trata el problema de otra forma, analicemos un poco el ejemplo 3.7, donde

$$f(x) = \cos(x) - x^2 = 0$$

Para que la función sea cero, el valor de $\cos(x)$ debe ser igual al valor de x^2 para que al restar uno de otro sea cero. La interpretación gráfica es que las funciones $\cos(x)$ y x^2 se crucen en un valor de x . Ese valor es precisamente la raíz de $f(x) = \cos(x) - x^2$.

El método plantea que si transformamos $f(x) = 0$ a una expresión como $x = g(x)$, donde x se despeja de $f(x)$, entonces resulta una ecuación distinta a $f(x)$ que llamamos $g(x)$. Como $f(x)$ es una ecuación no lineal, entonces resulta una ecuación distinta que sigue dependiendo de x .

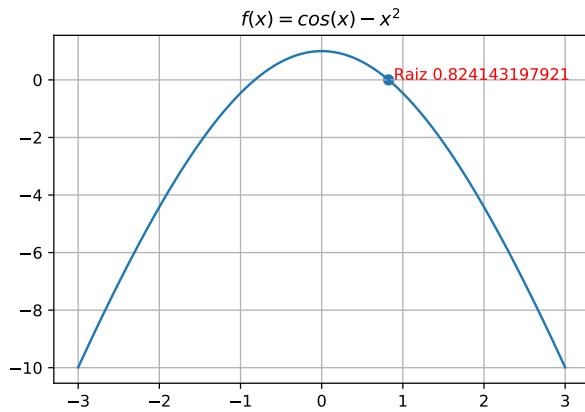
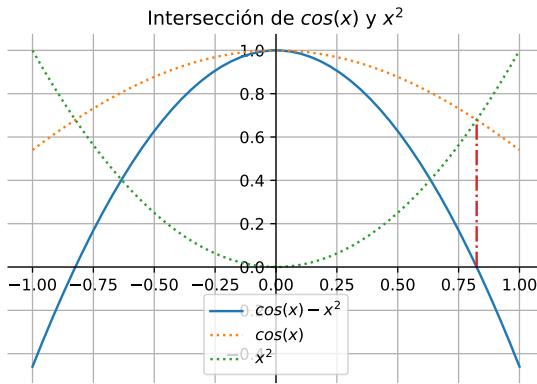


Figura 3.18. Método de Muller

Figura 3.19. Intersección de $\cos(x)$ y x^2

El método inicia con un valor de x , el cual se sustituye en $g(x)$ para obtener un resultado que se sustituye nuevamente como x , los cálculos se repiten hasta que se cumpla el criterio de convergencia $|x - g(x)| < \text{tolerancia}$, es decir, donde el valor de x y $g(x)$ se igualan o donde la función x interseca a la función $g(x)$.

■ **Ejemplo 3.8 — Método del punto fijo.** Calcular el volumen de 2 moles de CO₂ a una presión de 10 atm y 300 K, usando la ecuación de Van der Waals

$$\left(P + \frac{n^2 a}{V^2} \right) (V - nb) = nRT$$

Donde $a = 3.592$, $b = 0.04267$, $n = 2$, $R = 0.082$, $T = 300$ y $P = 10$.

La ecuación la debemos transformar a una expresión como $x = g(x)$, entonces debemos despejar V , en este caso despejamos V del término $(V - nb)$

Algoritmo 7 Método del punto fijo

```

1: procedimiento PUNTOFIJO( $g, x$ )                                 $\triangleright g$  es la función resultante de despejar  $x$ 
2:    $k \leftarrow 0$ 
3:    $cumple \leftarrow \text{falso}$ 
4:   mientras no cumple &  $k < imax$  hacer                       $\triangleright$  Criterio de convergencia
5:      $x \leftarrow g(x)$                                                $\triangleright$  se calcula la nueva  $x$ 
6:      $cumple \leftarrow |x - g(x)| < tol$ 
7:      $k \leftarrow k + 1$ 
8:   fin mientras
9:   retornar  $x$                                                   $\triangleright$  Raíz
10: fin procedimiento

```

$$V = \frac{nRT}{\left(P + \frac{n^2a}{V^2}\right)} + nb$$

Solución

Programa 3.10. Método del punto fijo

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import math
5 import cmath
6
7 def puntoFijo(g,x,imax=100,tol=1e-8):
8     cumple=False
9     print('{:~10s} {:~10s}'.\
10         format('x','g(x)'))
11    k=0
12    while (not cumple and k<imax):
13        print('{:10.5f} {:10.5f}'.\
14            format(x,g(x)))
15        x=g(x)
16        cumple=abs(x-g(x))<=tol
17        k+=1
18    if k<imax:
19        return x
20    else:
21        raise ValueError ('La función no converge')
22
23 # Funcion a evaluar
24 def g(v):
25     n=2
26     R=0.082
27     a=3.592
28     b=0.04267

```

```

29     T=300
30     P=10
31     return n*R*T/(P+n**2*a/v**2)+n*b
32
33 def main():
34     # valores iniciales
35     x0=2
36     # Llamada al algoritmo
37     raiz=puntoFijo(g,x0)
38     print('f({:e})={:e}'.format(raiz,g(raiz)-raiz))
39
40     x=np.linspace(0.01,6,100)
41     y=g(x)-x
42
43     fig = plt.figure()
44     plt.plot(x,y,label='$f(x)$')
45
46     plt.plot(x,x,':',label='$x$')
47     plt.plot(x,g(x),'--',label='$g(x)$')
48     plt.plot(np.array([raiz,raiz]),np.array([0,g(raiz)]),'-.')
49
50     plt.title('Volumen de $CO_2$')
51     plt.scatter(raiz,g(raiz)-raiz)
52     plt.text(raiz,g(raiz)-raiz,' Raiz '+str(raiz),color='red')
53     plt.legend()
54     plt.grid()
55     plt.show()
56     fig.savefig("puntofijo.pdf", bbox_inches='tight')
57
58 if __name__ == "__main__": main()

```

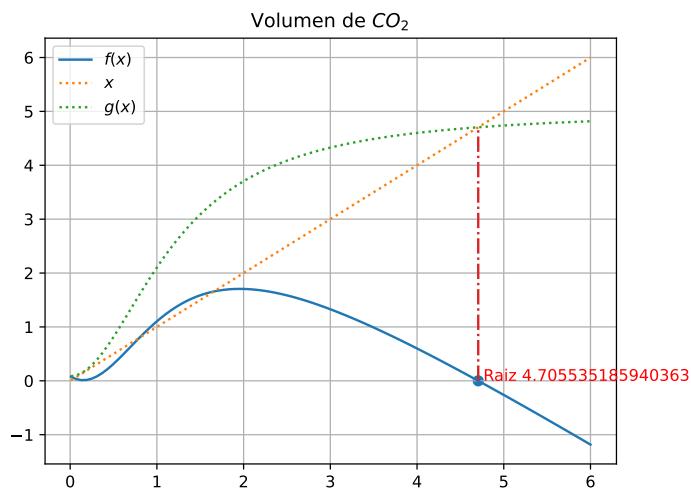


Figura 3.20. Método del punto fijo

x	$g(x)$
2.00000	3.70512
3.70512	4.53919
4.53919	4.68462
4.68462	4.70302
4.70302	4.70523
4.70523	4.70550
4.70550	4.70553
4.70553	4.70553
4.70553	4.70554
4.70554	4.70554
$f(4.705535e+00) = 6.514502e-09$	

El volumen de CO₂ es 4.705535. ■

El método del punto fijo puede divergir cuando las sustituciones en lugar de ser igual al valor sustituido se vuelven cada vez más diferentes. Para asegurar que $g(x)$ converge a la raíz se debe cumplir que $|g'(x)| < 1$.

3.3.6 Wegstein

El valor inicial x_0 en el método de punto fijo genera un punto en $g(x_0)$ que es x_1 , x_1 proyectado sobre el eje x genera el punto x_2 evaluando $g(x_2)$, hasta aquí tenemos el mismo resultado que el punto fijo, el siguiente paso es hacer una interpolación lineal entre los puntos $[x_0, g(x_0)]$ y $[x_2, g(x_2)]$, la intersección de la interpolación con la línea recta x genera el punto x_3 que es una mejor aproximación a la raíz.

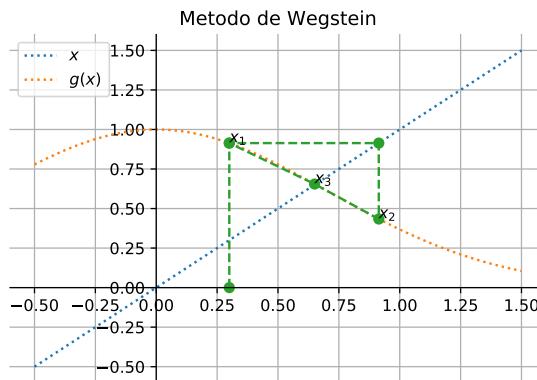


Figura 3.21. Método de Wegstein

$$x_1 = g(x_0)$$

$$x_2 = g(x_1)$$

Interpolación entre los puntos $[x_0, g(x_0)]$ y $[x_2, g(x_2)]$ se iguala a x_3 ya que está sobre la recta x , por lo tanto, la intersección de la interpolación con la gráfica de x es x_3

$$\begin{aligned}
 x_3 &= g(x_1) + \frac{g(x_2) - g(x_1)}{x_2 - x_1}(x_3 - x_1) \\
 (x_3 - g(x_1))(x_2 - x_1) &= (g(x_2) - g(x_1))(x_3 - x_1) \\
 x_2x_3 - x_1x_3 - x_2g(x_1) + x_1g(x_1) &= x_3g(x_2) - x_1g(x_2) - x_3g(x_1) + x_1g(x_1) \\
 x_1g(x_2) - x_2g(x_1) &= x_3g(x_2) - x_3g(x_1) - x_2x_3 + x_1x_3
 \end{aligned}$$

Se pasan todos los términos que multiplican a x_3 del lado derecho.

$$\begin{aligned}
 x_1g(x_2) - x_2g(x_1) &= x_3(g(x_2) - g(x_1) - x_2 + x_1) \\
 x_3 &= \frac{x_1g(x_2) - x_2g(x_1)}{x_1 - g(x_1) - x_2 + g(x_2)}
 \end{aligned}$$

Se puede observar que x_3 está más cerca que incluso x_1 y x_2 . En la siguiente iteración ahora se pasa la línea entre los puntos $[x_2, g(x_2)]$ y $[x_3, g(x_3)]$ y se vuelve a evaluar la siguiente estimación de la raíz con la intersección de la línea x . La ecuación iterativa para el método de Wegstein es:

Ecuación iterativa de Wegstein

$$x_{n+1} = \frac{x_{n-1}g(x_n) - x_ng(x_{n-1})}{x_{n-1} - g(x_{n-1}) - x_n + g(x_n)} \quad (3.6)$$

El método de Wegstein converge bajo las condiciones donde el método de punto fijo diverge, por lo que no se requiere validar que $|g'(x)| < 1$.

A continuación, el algoritmo:

Algoritmo 8 Método de Wegstein

1: procedimiento WEGSTEIN(g, x)	▷ g es la función resultante de despejar x
2: $k \leftarrow 0$	
3: $x_1 \leftarrow g(x_0)$	
4: $x_2 \leftarrow g(x_1)$	
5: $cumple \leftarrow \text{falso}$	
6: mientras no cumple & $k < imax$ hacer	▷ Criterio de convergencia
7: $x_3 \leftarrow (x_1 * g(x_2) - x_2 * g(x_1)) / (x_1 - g(x_1) - x_2 + g(x_2))$	▷ se calcula la nueva x
8: $cumple \leftarrow x_2 - g(x_2) < tol$	
9: $x_1 \leftarrow x_2$	
10: $x_2 \leftarrow x_3$	
11: $k \leftarrow k + 1$	
12: fin mientras	
13: retornar x	▷ Raíz
14: fin procedimiento	

■ **Ejemplo 3.9 — Método de Wegstein.** Calcular el volumen de 2 moles de CO₂ a una presión de 10 atm y 300 K, usando la ecuación de Van der Waals

$$\left(P + \frac{n^2 a}{V^2} \right) (V - nb) = nRT$$

Donde $a = 3.592$, $b = 0.04267$, $n = 2$, $R = 0.082$, $T = 300$ y $P = 10$.

La ecuación la debemos transformar a una expresión como $x = g(x)$, entonces debemos despejar V , en este caso, despejamos V del término ($V - nb$)

$$V = \frac{nRT}{\left(P + \frac{n^2 a}{V^2} \right)} + nb$$

Solución

Programa 3.11. Método de Wegstein

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import math
5 import cmath
6
7 def wegstein(g,x0,imax=100,tol=1e-8):
8     cumple=False
9     print('{:~10s} {:~10s} {:~10s} {:~10s} {:~10s}'.\
10         format('x1','x2','x3','g(x1)','g(x3)'))
11     x1=g(x0)
12     x2=g(x1)
13     k=0
14     while (not cumple and k<imax):
15         x3=(x1*g(x2)-x2*g(x1))/(x1-g(x1)-x2+g(x2))
16         print('{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}'.\
17             format(x1,x2,x3,g(x1),g(x2),g(x3)))
18         x1=x2
19         x2=x3
20         cumple=abs(x2-g(x2))<=tol
21
22         k+=1
23     if k<imax:
24         return x3
25     else:
26         raise ValueError ('La función no converge')
27
28 # Funcion a evaluar
29 def g(v):
30     n=2
31     R=0.082

```

```
32     a=3.592
33     b=0.04267
34     T=300
35     P=10
36     return n*R*T/(P+n**2*a/v**2)+n*b
37
38 def main():
39     # valores iniciales
40     x0=2
41     # Llamada al algoritmo
42     raiz=wegstein(g,x0)
43     print('f({:e})={:e}'.format(raiz,g(raiz)-raiz))
44
45     x=np.linspace(0.01,6,100)
46     y=g(x)-x
47
48     fig = plt.figure()
49     plt.plot(x,y,label='$f(x)$')
50
51     plt.plot(x,x,':',label='$x$')
52     plt.plot(x,g(x),':',label='$g(x)$')
53     plt.plot(np.array([raiz,raiz]),np.array([0,g(raiz)]),'-.')
54
55     plt.title('Volumen de $CO_2$')
56     plt.scatter(raiz,g(raiz)-raiz)
57     plt.text(raiz,g(raiz)-raiz,' Raíz '+str(raiz),color='red')
58     plt.legend()
59     plt.grid()
60     plt.show()
61     fig.savefig("wegstein.pdf", bbox_inches='tight')
62
63 if __name__ == "__main__": main()
```

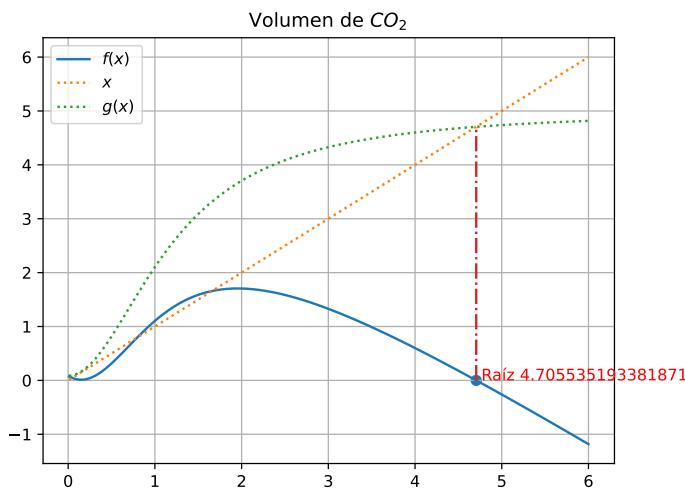


Figura 3.22. Método de Wegstein

x1	x2	x3	g(x1)	g(x3)	g(x3)
3.70512	4.53919	4.71533	4.53919	4.68462	4.70670
4.53919	4.71533	4.70547	4.68462	4.70670	4.70553
4.71533	4.70547	4.70554	4.70670	4.70553	4.70554
4.70547	4.70554	4.70554	4.70553	4.70554	4.70554
$f(4.705535 \times 10^0) = -3.654410 \times 10^{-11}$					

El volumen de CO_2 es 4.705535.

■

3.3.7 Raíces de polinomios

Un polinomio es una expresión matemática que tiene la forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

donde las a_i son los coeficientes de x^i del polinomio y x es una variable indeterminada.

El grado de un polinomio es el mayor exponente al que se encuentra elevada la variable x .

Las raíces o ceros de un polinomio son los valores de x que anulan el polinomio, esto es, $x = r$ es una raíz de $P(x)$ si $P(r) = 0$.

El teorema fundamental del álgebra enuncia que todo polinomio de grado mayor que cero tiene una raíz, esto implica que todo polinomio de grado n de una variable con grado mayor que cero tiene exactamente n raíces reales y/o complejas.

Los métodos vistos anteriormente, tanto cerrados como abiertos, encuentran una raíz a la vez, para encontrar las n raíces de un polinomio de grado n se deberá aplicar el método n veces. Si aplicamos métodos cerrados se deberán seleccionar los valores adecuados que encierran cada una de las n raíces,

pero si usamos los métodos abiertos no se asegura que cambiando los valores iniciales localicen la raíz deseada, para ello se deberá aplicar el método de agotamiento para asegurar que no se vuelve a localizar la misma raíz encontrada.

$$\begin{aligned} P(x) &= a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 && \text{Raíz r1} \\ Q(x) &= \frac{a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0}{(x - r1)} && \text{Raíz r2} \\ S(x) &= \frac{a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0}{(x - r1)(x - r2)} && \text{Raíz r3} \\ &\vdots \end{aligned}$$

■ **Ejemplo 3.10 — Raíces de un polinomio por agotamiento.** Se desea construir una caja rectangular a partir de una placa de acero de tamaño 30 x 50. Si se cortan piezas cuadradas en cada esquina de longitud x , determine el valor de x para obtener una caja de volumen 3000.

Solución

Si se cortan piezas cuadradas de longitud x en cada esquina de la placa, está claro que el largo de la caja será de $50 - 2x$, el ancho será de $30 - 2x$ y de alto x , por lo que el volumen de la caja se calcula como:

$$v = (50 - 2x)(30 - 2x)x$$

y la ecuación a resolver será:

$$p(x) = (50 - 2x)(30 - 2x)x - 3000 = 0$$

La ecuación es un polinomio de grado 3, por lo tanto, tiene 3 raíces, para obtenerlas por el método de la secante se requiere aplicar 3 veces agotando el polinomio con la raíz obtenida cada vez.

Programa 3.12. Raíces de un polinomio por el método de Newton-Raphson

```

1 from scipy.optimize import newton
2
3 #se define el polinomio
4 p=lambda x:(50-2*x)*(30-2*x)*x-3000
5 #se obtiene la primera raíz
6 r1=newton(p,0)
7 print(r1)
8
9 #se reduce el polinomio
10 q=lambda x:p(x)/(x-r1)
11 #se obtiene la segunda raíz
12 r2=newton(q,0)
13 print(r2)
14
15 #se reduce el polinomio
16 s=lambda x:q(x)/(x-r2)
```

```

17 #se obtiene la tercera raíz
18 r3=newton(s,0)
19 print(r3)

```

```

2.7525512860841075
9.999999999999973
27.247448713915887

```

Se observa que las dos primeras raíces resuelven el problema, es decir, hay dos alturas x de la caja para un volumen de 3000, una de 2.75 y otra de 9.99. La tercera raíz no es solución del problema, aunque cumpla matemáticamente, porque no es posible hacer cortes de 27.24 ya que las dimensiones de la placa no lo permiten.

La biblioteca *numpy* de Python cuenta con la función *roots* que obtiene las raíces de un polinomio, el argumento que recibe es una lista con los coeficientes del polinomio, para ello se debe desarrollar el polinomio anterior para identificar sus coeficientes:

$$p(x) = 4x^3 - 160x^2 + 1500x - 3000$$

Programa 3.13. Raíces de un polinomio con la función *roots*

```

1 import numpy as np
2
3 coef=[4, -160, 1500, -3000]
4 print(np.roots(coef))

```

```
[27.24744871 10.           2.75255129]
```

Observe que los valores son un poco distintos a los obtenidos con el método de Newton, esto se debe a que no se obtienen los valores exactos de cada raíz y ocasiona que la siguiente raíz tenga errores de redondeo.

■

3.3.8 Raíces complejas

Los métodos abiertos (Newton-Raphson, secante y Muller) obtienen raíces complejas, la condición es que los valores iniciales deben ser complejos, excepto el método de Muller. Una función que tiene una raíz compleja es $f(x) = x^2 + 2$, la raíz es $\sqrt{-2}$

- **Ejemplo 3.11 — Método de Newton-Raphson.** Encontrar la raíz compleja de

$$f(x) = x^2 + 2$$

Los valores iniciales son complejos.

Solución

Programa 3.14. Método de Newton-Raphson complejos

```

1 #importar las funciones de la biblioteca math
2 import numpy as np
3 import math
4 import cmath
5
6 def newtonRaphson(f ,df ,x ,imax=100 ,tol=1e-8):
7     cumple=False
8     print('{:^10s} {:^10s} {:^10s}'.\
9           format('x' , 'f(x)' , 'df(x)'))
10    k=0
11    while (not cumple and k<imax):
12        if df(x)!=0:
13            x=x-f(x)/df(x)
14        else:
15            x=x+tol
16        print('{:10.5f} {:10.5f} {:10.5f}'.\
17              format(x,f(x),df(x)))
18        cumple=abs(f(x))<=tol
19        k+=1
20    if k<imax:
21        return x
22    else:
23        raise ValueError ('La funcion no converge')
24
25 # Funcion a evaluar
26 def f(x):
27     return x**2+2
28 # Derivada
29 def df(x):
30     return 2*x
31
32 def main():
33     # valores iniciales
34     x0=1j
35     # Llamada al algoritmo
36     raiz=newtonRaphson(f ,df ,x0)
37     print('f({:e})={:e}'.format(raiz,f(raiz)))
38
39 if __name__ == "__main__": main()

```

x	f(x)	df(x)
0.00000+1.50000j	-0.25000+0.00000j	0.00000+3.00000j
0.00000+1.41667j	-0.00694+0.00000j	0.00000+2.83333j
0.00000+1.41422j	-0.00001+0.00000j	0.00000+2.82843j
0.00000+1.41421j	-0.00000+0.00000j	0.00000+2.82843j

```
f(0.000000e+00+1.414214e+00j)=-4.510614e-12+0.000000e+00j
```

La raíz es $0.000000e+00 + 1.414214e+00j$.

Programa 3.15. Método de la secante complejos

```

1 import numpy as np
2 import math
3 import cmath
4
5 def secante(f,x0,x1,imax=100,tol=1e-8):
6     cumple=False
7     print('{:^10s} {:^10s} {:^10s} {:^10s} {:^10s} {:^10s}'.\
8         format('x0','x1','x','f(x0)','f(x1)','f(x)'))
9     k=0
10    while (not cumple and k<imax):
11        x=x1-f(x1)*(x0-x1)/(f(x0)-f(x1))
12        print('{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}'.\
13            format(x0,x1,x,f(x0),f(x1),f(x)))
14        x0=x1
15        x1=x
16        cumple=abs(f(x))<tol
17        k+=1
18    if k<imax:
19        return x
20    else:
21        raise ValueError ('La funcion no converge')
22
23 # Funcion a evaluar
24 def f(x):
25     return x**2+2
26
27 def main():
28     # valores iniciales
29     x0=1j
30     x1=2j
31     # Llamada al algoritmo
32     raiz=secante(f,x0,x1,100,1e-4)
33     print('f({:e})={:e}'.format(raiz,f(raiz)))
34
35 if __name__ == "__main__": main()
```

x0	x1	x	f(x0)	f(x1)	f(x)
$0.00000+1.00000j$	$0.00000+2.00000j$	$0.00000+1.33333j$	$1.00000+0.00000j$	$-2.00000+0.00000j$	$0.22222+0.00000j$
$0.00000+2.00000j$	$0.00000+1.33333j$	$0.00000+1.40000j$	$-2.00000+0.00000j$	$0.22222+0.00000j$	$0.04000+0.00000j$
$0.00000+1.33333j$	$0.00000+1.40000j$	$0.00000+1.41463j$	$0.00000+1.40000j$	$0.00000+1.41463j$	$0.22222+0.00000j$

```

j 0.04000+0.00000j -0.00119+0.00000j
0.00000+1.40000j 0.00000+1.41463j 0.00000+1.41421j 0.04000+0.00000←
j -0.00119+0.00000j 0.00001+0.00000j
f(0.000000e+00+1.414211e+00j)=6.007287e-06+0.000000e+00j

```

La raíz es $0.000000e+00 + 1.414211e+00j$.

Programa 3.16. Método de Muller complejos

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4 import cmath
5
6 def muller(f,x0,x1,x2,imax=100,tol=1e-8):
7     cumpre=False
8     print('{:^10s} {:^10s} {:^10s} {:^10s} {:^10s} {:^10s} ←
9           {:^10s}'.\
10          format('x0','x1','x2','x3','f(x0)','f(x1)','f(x2)','f(x3←
11            )'))
12     k=0
13     while (not cumpre and k<imax):
14         d1=(f(x1)-f(x0))/(x1-x0)
15         d2=(f(x2)-f(x1))/(x2-x1)
16         d3=(d2-d1)/(x2-x0)
17         a=d3
18         b=d1-a*(x0+x1)
19         c=f(x0)+x0*(a*x1-d1)
20         den1=-b+np.sqrt(b**2-4*a*c)
21         den2=-b-np.sqrt(b**2-4*a*c)
22         if abs(den1)>abs(den2):
23             x3=2*c/den1
24         else:
25             x3=2*c/den2
26
27         print(\
28             '{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5←
29               f} {:10.5f} {:10.5f}'.\
30             format(x0,x1,x2,x3,f(x0),f(x1),f(x2),f(x3)))
31         x0=x1
32         x1=x2
33         x2=x3
34         cumpre=abs(f(x3))<tol
35         k+=1
36         if k<imax:
37             return x3
38         else:
39             raise ValueError ('La funcion no converge')
# Funcion a evaluar

```

```

39 def f(x):
40     return x**2+2
41
42 def main():
43     # valores iniciales
44     x0=0j #valor complejo para que los resultados puedan ser ←
45     # complejos
46     x1=1
47     x2=2
48     # Llamada al algoritmo
49     raiz=muller(f,x0,x1,x2,100,1e-4)
50     print('f({:e})={:e}'.format(raiz,f(raiz)))
51 if __name__ == "__main__": main()

```

x0	x1	x2	x3	f(x0)	f(x1) ←
f(x2)	f(x3)				
0.00000+0.00000j	1.00000	2.00000	-0.00000+1.41421j ←		
2.00000+0.00000j	3.00000	6.00000	0.00000+0.00000j		
f(-0.00000e+00+1.414214e+00j)=4.440892e-16+0.00000e+00j					

La raíz es $-0.000000e+00 + 1.414214e+00j$.

■

3.3.9 El fractal de Newton

El matemático francés Benoit Mandelbrot fue el responsable de desarrollar, en 1975, el concepto de fractal, que proviene del latín *fractus* (puede traducirse como “quebrado”). El término pronto fue aceptado por la comunidad científica y ya forma parte del diccionario de la Real Academia Española.

Estudiemos el comportamiento del método de Newton-Raphson, visto en la sección 3.3.1, revisemos el número de iteraciones necesarias para encontrar una raíz dependiendo del valor inicial dado.

Al ser un método abierto, dado un valor inicial no asegura encontrar una raíz en particular, de hecho, para algunos valores le toma más iteraciones llegar a una raíz que para otros.

Modifiquemos el programa de Newton-Raphson 3.4 para que devuelva la raíz y el número de iteraciones hechas.

Programa 3.17. Método de Newton-Raphson devuelve iteraciones

```

1 import numpy as np
2
3 def nr(f,df,x0,tol=1e-5):
4     k=0
5     while np.abs(f(x0))>tol and k<50:
6         if df(x0)!=0:
7             x0=x0-f(x0)/df(x0)

```

```

8         else:
9             x0=x0+tol
10            k=k+1
11        return x0,k

```

Probemos con algunos valores iniciales

```

f=lambda x:x**2-2
df=lambda x:2*x

r,it=nr(f,df,1) #valor inicial 1
print(r,it)

1.4142156862745099 3 #hizo 3 iteraciones para encontrar la raíz

r,it=nr(f,df,0.5) #valor inicial 0.5
print(r,it)

1.4142135625249321 5 #hizo 5 iteraciones para encontrar la raíz

r,it=nr(f,df,0.001) #valor inicial 0.001
print(r,it)

1.4142135626178514 14 #hizo 14 iteraciones para encontrar la raíz

```

Podemos pensar que existen valores de x_0 donde es más fácil encontrar la raíz y existen otros donde encontrar la raíz es un poco más tardado.

Seamos un poco curiosos y averigüemos qué comportamiento tiene el método de Newton-Raphson para otros valores iniciales. Si hacemos un proceso donde cambiemos el valor inicial de x_0 en el intervalo $[-2, 2]$ y obtenemos el número de iteraciones k que le toma llegar a la raíz en cada caso y grafiquemos esos resultados para observar el comportamiento.

Programa 3.18. Método de Newton-Raphson, valor inicial vs iteraciones para $f(x) = x^2 - 2$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #definición del método de Newton-Raphson
5 def nr(f,df,x0,tol=1e-5):
6     k=0
7     while np.abs(f(x0))>tol and k<50:
8         x0=x0-f(x0)/df(x0)
9         k=k+1
10    return x0,k #devuelve la raíz y las iteraciones
11
12 #define la función y su derivada
13 f=lambda x:x**2-2
14 df=lambda x:2*x
15

```

```

16 x0=-2 #valor inicial
17 x=np.empty(400)
18 y=np.empty(400)
19 for i in range(400):
20     x[i]=x0
21     _,y[i]=nr(f,df,x0)
22     x0=x0+0.01 #incrementa el valor inicial
23 plt.plot(x,y)
24 plt.show()

```

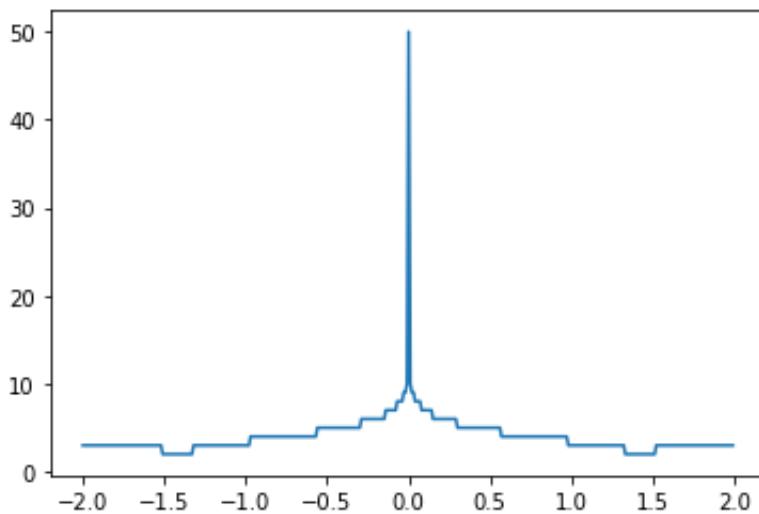


Figura 3.23. Valor inicial vs iteraciones para $f(x) = x^2 - 2$

Observamos dos mínimos cercanos a -1.4142 y 1.4142 , esto se debe a que existen dos raíces para esta función.

Estudiemos ahora la función $f(x) = x^3 - 1$

Programa 3.19. Método de Newton-Raphson, valor inicial vs iteraciones para $f(x) = x^3 - 1$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #definición del método de Newton-Raphson
5 def nr(f,df,x0,tol=1e-5):
6     k=0
7     while np.abs(f(x0))>tol and k<50:
8         x0=x0-f(x0)/df(x0)
9         k=k+1
10    return x0,k #devuelve la raíz y las iteraciones
11
12 #define la función y su derivada
13 f=lambda x:x**3-1
14 df=lambda x:3*x**2

```

```

15 x0=-2 #valor inicial
16 x=np.empty(400)
17 y=np.empty(400)
18 for i in range(400):
19     x[i]=x0
20     _,y[i]=nr(f,df,x0)
21     x0=x0+0.01 #incrementa el valor inicial
22 plt.plot(x,y)
23 plt.show()
24

```

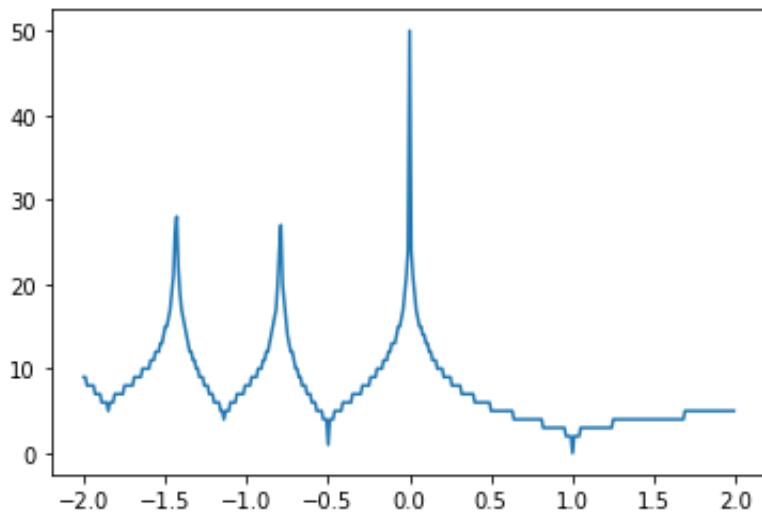


Figura 3.24. Valor inicial vs iteraciones para $f(x) = x^3 - 1$

Esta función al ser cúbica tiene 3 raíces, una es real y las otras complejas. El método de Newton-Raphson es capaz de encontrar raíces complejas, sólo que el valor inicial x_0 debe ser un valor complejo. Entonces debemos crear un valor complejo para los valores iniciales, y debemos variar tanto el valor real como el imaginario y hacer todas las combinaciones.

Ahora cambiemos la representación gráfica y en lugar de graficar el valor inicial vs iteraciones, grafiquemos en el eje x la parte real, y en el eje y la parte imaginaria, el número de iteraciones es el color del punto.

Programa 3.20. Método de Newton-Raphson, fractal para $f(x) = x^3 - 1$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #definición del método de Newton-Raphson
5 def nr(f,df,x0,tol=1e-5):
6     k=0
7     while np.abs(f(x0))>tol and k<50:
8         x0=x0-f(x0)/df(x0)

```

```

9      k=k+1
10     return x0,k #devuelve la raíz y las iteraciones
11
12 #define la función y su derivada
13 f=lambda x:x**8+15*x**4-16
14 df=lambda x:8*x**7+60*x**3
15
16 re = np.linspace(-2, 2, 500) #rango de valores reales
17 im = np.linspace(-2, 2, 500) #rango de valores imaginarios
18
19 its=[]
20 datos = np.empty((len(re), len(im)))
21 for i in range(len(re)):
22     for j in range(len(im)):
23         x0=complex(re[i], im[j]) #crear el número complejo
24         _,it = nr(f,df,x0) #corre el método y obtiene las ←
25             iteraciones
26         datos[i,j]=it #guarda el numero de iteraciones en el ←
27             arreglo
28
29 #grafica el arreglo
30 plt.imshow(datos.T, interpolation="spline36", cmap='turbo')
31 plt.axis('off')
32 plt.show()

```

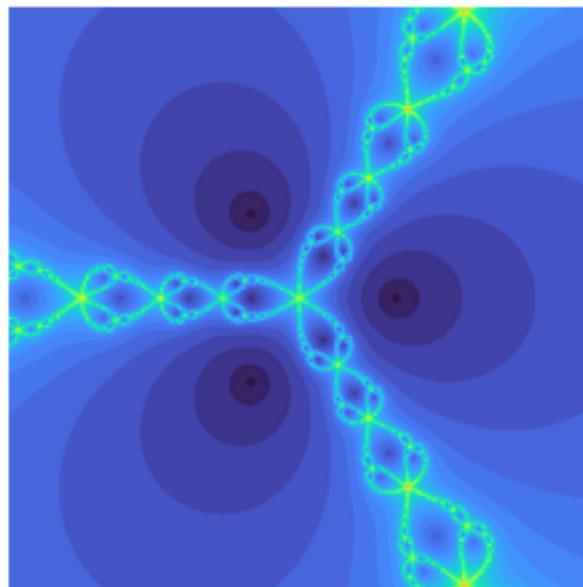


Figura 3.25. Fractal de Newton para $f(x) = x^3 - 1$

A este gráfico se le conoce como el fractal de Newton-Raphson. El gráfico presenta un comportamiento fractal, es decir, si hacemos un acercamiento en cualquier sección se observa un comportamiento similar.

Probemos ahora con la siguiente función $f(x) = x^6 + x^3 - 1$

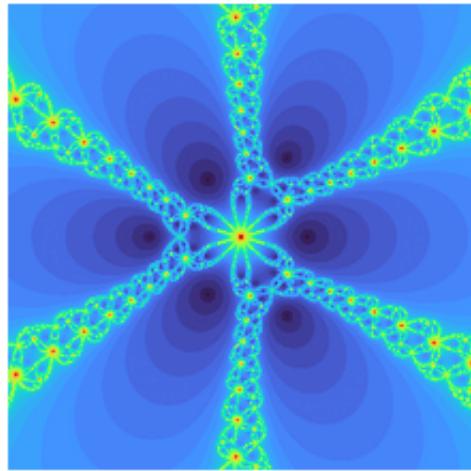


Figura 3.26. Fractal de Newton-Raphon para $f(x) = x^6 + x^3 - 1$

Hay fractales realmente impresionantes como el siguiente:

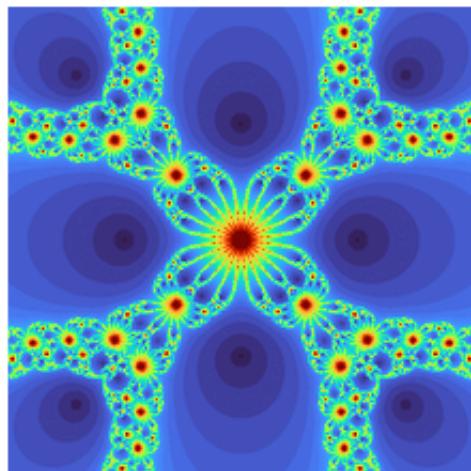
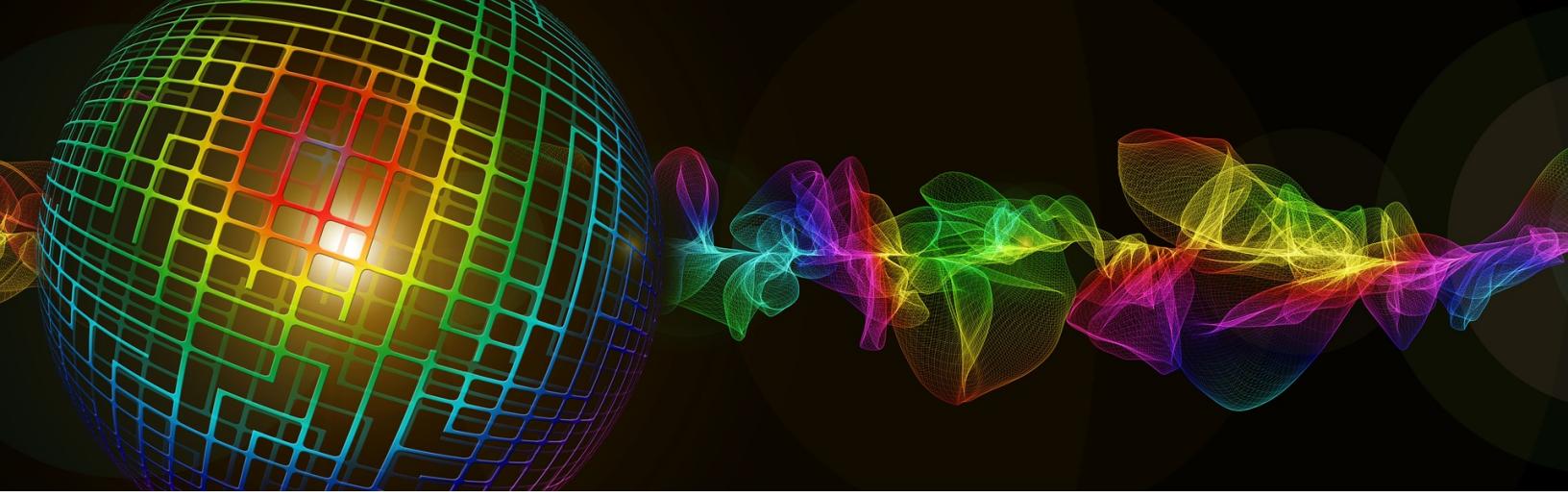


Figura 3.27. Fractal de Newton-Raphon para $f(x) = x^8 + 15x^4 - 16$



4. Sistemas de ecuaciones lineales

Una ecuación lineal tiene la forma:

$$a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n = b_1$$

Donde las a_i son los coeficientes, b_1 es una constante y x_i son las incógnitas para determinar. Las ecuaciones lineales son muy comunes en Ingeniería Química, por ejemplo, se generan a partir de la multiplicación de un flujo a_i por la concentración a determinar x_i .

Ejemplos de ecuaciones lineales:

$$\begin{aligned} 3x + 2y &= 5 \\ \pi x_1 + \sqrt{2}x_2 + 1.2x_3 &= -2 \end{aligned}$$

Ejemplos de ecuaciones no lineales:

$$\begin{aligned} 3x + 2\sin(y) &= 5 \\ x_1x_2 + x_3 &= 10 \end{aligned}$$

Un sistema de ecuaciones lineales son un conjunto de n ecuaciones con n incógnitas:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + \dots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + \dots + a_{2,n}x_n &= b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 + \dots + a_{3,n}x_n &= b_3 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + a_{n,3}x_3 + \dots + a_{n,n}x_n &= b_n \end{aligned}$$

En un sistema de ecuaciones lineales se indica la relación que tiene cada elemento dentro del sistema y cómo interactúan. Las b_i son las entradas al sistema, las $a_{i,i}$ son el comportamiento del sistema bajo esas entradas y las x_i son la respuesta que tiene el sistema.

Resolver el sistema de ecuaciones lineales es conocer la respuesta del sistema. Para manejar de una manera práctica el sistema, se representa en su forma matricial:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

A x b

Donde A es la matriz de coeficientes, x es el vector de las incógnitas y b es el vector de las constantes.

Resolver la ecuación $Ax=b$ nos lleva a tres casos:

- **Tiene solución única.** Un sistema consistente determinado es un sistema linealmente independiente donde:

$$Rango(A) = Rango(A|b) = \text{número de incógnitas}$$

Por ejemplo, el sistema

$$\begin{aligned} 5x + 2y &= 12 \\ 8x - 4y &= 7 \end{aligned}$$

Gráficamente la solución es el punto donde ambas ecuaciones se intersecan, es decir, es un punto que resuelve las dos ecuaciones simultáneamente.

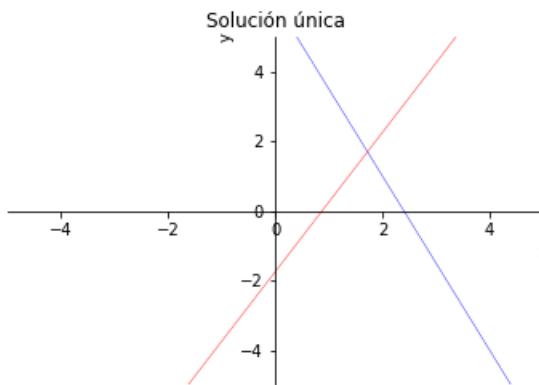


Figura 4.1. Solución única.

- **Tiene múltiples soluciones.** Un sistema consistente indeterminado es un sistema linealmente dependiente donde:

$$Rango(A) = Rango(A|b) < \text{número de incógnitas}$$

Por ejemplo, el sistema

$$5x + 2y = 12$$

$$10x + 4y = 24$$

Gráficamente ambas ecuaciones son la misma línea, es decir, la solución de una ecuación también es la solución de la segunda. Observe que la ecuación dos es dos veces la ecuación una.

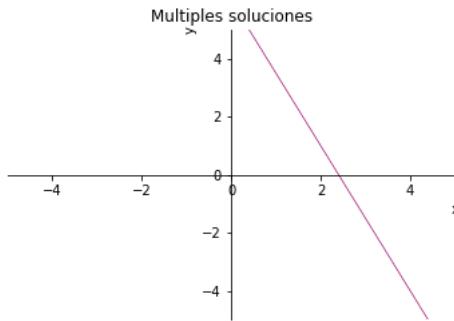


Figura 4.2. Solución múltiple

- **No tiene solución.** Un sistema inconsistente es un sistema linealmente dependiente, excepto por el vector de las constantes donde:

$$\text{Rango}(A) < \text{Rango}(A|b)$$

Por ejemplo, el sistema

$$5x + 2y = 12$$

$$10x + 4y = 10$$

Gráficamente ambas ecuaciones son paralelas, es decir, tienen la misma pendiente, pero distinta ordenada al origen, por lo que no se cruzan, por lo tanto, la solución de una no es la solución de la segunda.

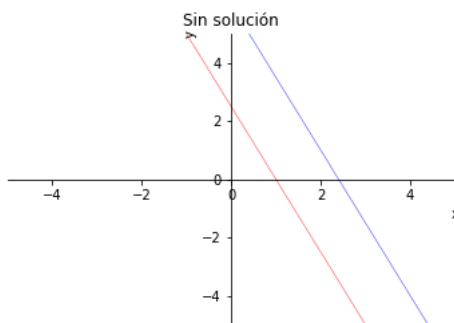


Figura 4.3. Sin solución

Los métodos para resolver el sistema $Ax = b$ se clasifican en:

- **Métodos directos.** Obtienen la solución después de aplicar un algoritmo con pasos finitos
 - Eliminación de Gauss
 - Eliminación de Gauss-Jordan
 - Inversa multiplicación
 - Regla de Cramer
- **Métodos iterativos.** Suponen un valor inicial y encuentran la solución cuando se cumple el criterio de convergencia
 - Jacobi
 - Gauss-Seidel

Los métodos se aplican para los sistemas consistentes tanto determinados (única solución) como indeterminados (múltiples soluciones).

4.1 Métodos directos

Los métodos directos transforman el sistema $Ax = b$ a un sistema equivalente, pero más fácil de resolver. Para lograr la transformación se aplican las operaciones elementales que no cambian el conjunto solución del sistema, pero crean ecuaciones equivalentes. Las operaciones elementales son las siguientes:

- **Multiplicación.** La operación de multiplicación por un escalar no nulo se aplica a una ecuación del sistema.

$$kE_i \rightarrow E_i$$

La operación de multiplicación se aplica para lograr 1 en la diagonal de A .

- **Intercambio.** La operación de intercambio de ecuaciones se aplica entre dos ecuaciones para cambiar su posición en el sistema.

$$E_i \leftrightarrow E_j$$

La operación de intercambio se aplica para el pivoteo parcial o para lograr la diagonalización del sistema.

- **Reducción.** La operación de reducción se aplica entre la ecuación pivote y la ecuación a reducir.

$$kE_i + E_j \rightarrow E_j$$

Donde $k = \frac{a_{j,i}}{a_{i,i}}$ es el factor de reducción, hacer cero un término de la ecuación, E_i es la ecuación pivote y E_j es la ecuación a reducir.

4.1.1 Eliminación de Gauss

El método de eliminación de Gauss toma la matriz de coeficientes A y la extiende con el vector b ($A|b$) y usando las operaciones elementales convenientemente, la convierte a un sistema $Ux = d$ donde U es la matriz triangular superior.

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array} \right) \xrightarrow{\text{operaciones elementales}} \left(\begin{array}{cccc|c} u_{1,1} & a_{1,2} & \cdots & a_{1,n} & d_1 \\ 0 & u_{2,2} & \cdots & u_{2,n} & d_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & u_{n,n} & d_n \end{array} \right) \quad A \mid b \quad U \mid d$$

El proceso de Reducción se logra tomando la ecuación E_i que es la ecuación pivote y se deben reducir las ecuaciones E_{i+1} hasta la ecuación E_n .

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \color{red}{a_{i,i}} & a_{i,n} & b_i \\ \hline 0 & 0 & \color{blue}{a_{j,i}} & a_{j,n} & b_j \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & a_{j,n} & a_{n,n} & b_n \end{array} \right) \xrightarrow{kE_i + E_j \rightarrow E_j} \left(\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & d_1 \\ 0 & u_{2,2} & \cdots & u_{2,n} & d_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & u_{i,i} & u_{i,n} & d_i \\ \hline 0 & 0 & u_{i,j} & u_{j,n} & d_j \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & u_{j,n} & u_{n,n} & d_n \end{array} \right)$$

El escalar no nulo k que se usa para la reducción es:

$$k = \frac{a_{j,i}}{a_{i,i}}$$

Donde $a_{j,i}$ es el elemento a reducir de la ecuación E_j y $a_{i,i}$ es el elemento pivote.

Algoritmo 9 Eliminación de Gauss

```

procedimiento GAUSS( $A, b, n$ )            $\triangleright$  A matriz de coeficientes, b vector de constantes
    para  $i \leftarrow 1$  hasta  $n - 1$  hacer            $\triangleright$  Reducción matriz U
        para  $j \leftarrow i + 1$  hasta  $n$  hacer
            si  $A_{j,i} \neq 0$  entonces
                 $f \leftarrow A_{j,i}/A_{i,i}$ 
                para  $k \leftarrow i + 1$  hasta  $n$  hacer
                     $A_{j,k} \leftarrow A_{j,k} - fA_{i,k}$ 
                fin para
            fin si
        fin para
    fin para
    para  $i \leftarrow n$  hasta  $n - 1$  hacer            $\triangleright$  Sustitución en reversa
         $x_i \leftarrow b_i$ 
        para  $j \leftarrow i$  hasta  $n - 1$  hacer
             $x_i \leftarrow x_i - x_{j+1}A_{i,j+1}$ 
        fin para
         $x_i \leftarrow x_i/A_{i,i}$ 
    fin para
fin procedimiento
```

La solución del sistema $Ux = d$ se obtiene despejando x_n de la última ecuación de U, $a_{n,n}x_n = d_n$

para obtener el valor de

$$x_n = \frac{d_n}{a_{n,n}}$$

luego con el resultado se resuelve la ecuación E_{n-1} para obtener x_{n-1} , este paso es la solución en reversa.

■ **Ejemplo 4.1 — Método de eliminación de Gauss.** Un cliente solicita una mezcla de 100 litros de producto con la siguiente especificación, en la cual la fracción volumétrica de cada componente debe ser:

Componente	% volumen
A	25
B	20
C	25
D	20
E	10

Determinar el volumen que se debe usar en cada tanque para cumplir con la especificación del cliente.

Fracción en volumen de cada componente en cada tanque					
Tanque	A	B	C	D	E
1	55.8	7.8	16.4	11.7	8.3
2	20.4	52.1	11.5	9.2	6.8
3	17.1	12.3	46.1	14.1	10.4
4	18.5	13.9	11.5	47.0	9.1
5	19.2	18.5	21.3	10.4	30.6

Solución

$$\begin{pmatrix} 55.8 & 20.4 & 17.1 & 18.5 & 19.2 & 25 \times 100 \\ 7.8 & 52.1 & 12.3 & 13.9 & 18.5 & 20 \times 100 \\ 16.4 & 11.5 & 46.1 & 11.5 & 21.3 & 25 \times 100 \\ 11.7 & 9.2 & 14.1 & 47.0 & 10.4 & 20 \times 100 \\ 8.3 & 6.8 & 10.4 & 9.1 & 30.6 & 10 \times 100 \end{pmatrix}$$

Programa 4.1. Método de eliminación de Gauss

```

1 import numpy as np
2
3 def gauss(a,b):
4     n,_=np.shape(a)
5     A=np.c_[a,b]
6     for i in range(n-1):
7         for j in range(i+1,n):

```

```

8         # Si el termino ya es cero continua con la siguiente
9         if (A[j,i]!=0 and A[i,i]!=0):
10            f=A[j,i]/A[i,i] # factor de reduccion
11            A[j,i+1:n+1]=A[j,i+1:n+1]-f*A[i,i+1:n+1]
12        # aplica la sustitucion inversa
13        x=np.zeros(n)
14        for i in range(n-1,-1,-1):
15            x[i]=(A[i,n]-np.dot(A[i,i+1:n],x[i+1:n]))/A[i,i]
16        return x
17
18 def main():
19     a=np.array([[55.8 , 20.4 , 17.1 , 18.5 , 19.2 ],\
20                [7.8   , 52.1 , 12.3 , 13.9 , 18.5 ],\
21                [16.4  , 11.5 , 46.1 , 11.5 , 21.3 ],\
22                [11.7  , 9.2  , 14.1 , 47.0 , 10.4 ],\
23                [8.3   , 6.8  , 10.4 , 9.1  , 30.6 ]])
24     b=np.array([2500,2000,2500,2000,1000])
25     n,c=np.shape(a)
26     r=np.linalg.matrix_rank(a)
27     ab=np.c_[a,b]
28     ra=np.linalg.matrix_rank(ab)
29     print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,n))
30     if (r==ra==n):
31         print('solución única')
32         x=gauss(a,b)
33         #x=np.linalg.solve(a, b)
34         print(x)
35     if (r==ra<n):
36         print('múltiples soluciones')
37     if (r<ra):
38         print('sin solución')
39 if __name__ == "__main__": main()

```

```
rango(A)=5 rango(Ab)=5 n=5
solucion unica
[ 17.65270678  19.67904691  35.15143662  22.69415361    4.82265608]
```

La solución es

Tanque	volumen
1	17.65
2	19.67
3	35.15
4	22.69
5	4.82

4.1.2 Gauss-Jordan

El método de Gauss-Jordan aplica el paso de la reducción hasta conseguir la matriz identidad, logrando entonces que la última columna sea el conjunto solución del sistema.

$$\left(\begin{array}{ccccc} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array} \right) \xrightarrow{\text{Operaciones elementales}} \left(\begin{array}{ccccc} 1 & 0 & \cdots & 0 & d_1 \\ 0 & 1 & \cdots & 0 & d_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & d_n \end{array} \right)$$

La matriz identidad

$$\left(\begin{array}{ccccc} 1 & 0 & \cdots & 0 & d_1 \\ 0 & 1 & \cdots & 0 & d_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & d_n \end{array} \right)$$

Es un sistema equivalente al sistema inicial, sólo que la columna $n + 1$ es la solución del sistema.

Algoritmo 10 Eliminación de Gauss-Jordan

```

procedimiento GAUSSJORDAN( $A, b, n$ )       $\triangleright A$  matriz de coeficientes,  $b$  vector de constantes
    para  $i \leftarrow 1$  hasta  $n$  hacer           $\triangleright$  Reducción matriz U
        para  $j \leftarrow 1$  hasta  $n$  hacer
            si  $A_{j,i} \neq 0$  &  $i \neq j$  entonces
                 $f \leftarrow A_{j,i}/A_{i,i}$ 
                para  $k \leftarrow i + 1$  hasta  $n$  hacer
                     $A_{j,k} \leftarrow A_{j,k} - fA_{i,k}$ 
                fin para
            fin si
        fin para
    fin para
fin procedimiento
```

■ **Ejemplo 4.2 — Método de eliminación de Gauss-Jordan.** En el siguiente esquema de destilación que se usa para separar los siguientes cuatro componentes Para-Xileno, Estireno, Tolueno y Benceno, calcule el flujo molar de A, B, C y D para cumplir con la composición requerida en cada flujo. [3]

Solución

Matriz extendida del sistema

$$\left(\begin{array}{ccccc} 0.07 & 0.18 & 0.15 & 0.24 & 0.10 \times 100 \\ 0.03 & 0.25 & 0.10 & 0.65 & 0.20 \times 100 \\ 0.55 & 0.41 & 0.55 & 0.09 & 0.40 \times 100 \\ 0.35 & 0.16 & 0.20 & 0.02 & 0.30 \times 100 \end{array} \right)$$

Programa 4.2. Método de eliminación de Gauss-Jordan

```

1 import numpy as np
2
3 def gaussJordan(a,b):
4     n,_=np.shape(a)
5     A=np.c_[a,b]
6     for i in range(n):
7         for j in range(n):
8             # Si el termino ya es cero continua con la siguiente
9             if (A[j,i]!=0 and A[i,i]!=0 and i!=j):
10                 # factor de reduccion
11                 f=A[j,i]/A[i,i]
12                 A[j,i+1:n+1]=A[j,i+1:n+1]-f*A[i,i+1:n+1]
13     # aplica la sustitucion inversa
14     x=np.zeros(n)
15     for i in range(n):
16         x[i]=A[i,n]/A[i,i]
17
18     return x
19
20 def main():
21     a=np.array([[55.8 , 20.4 , 17.1 , 18.5 , 19.2 ],\
22                [7.8   , 52.1 , 12.3 , 13.9 , 18.5 ],\ \
23                [16.4  , 11.5 , 46.1 , 11.5 , 21.3 ],\ \
24                [11.7  , 9.2   , 14.1 , 47.0 , 10.4 ],\ \
25                [8.3   , 6.8   , 10.4 , 9.1   , 30.6 ]])
26     b=np.array([2500,2000,2500,2000,1000])
27     n,c=np.shape(a)
28     r=np.linalg.matrix_rank(a)
29     ab=np.c_[a,b]
30     ra=np.linalg.matrix_rank(ab)
31
32     print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,n))
33
34     if (r==ra==n):
35         print('solución única')
36         x=gaussJordan(a,b)
37         #x=np.linalg.solve(a, b)
38         print(x)
39
40     if (r==ra<n):
41         print('múltiples soluciones')
42
43     if (r<ra):
44         print('sin solución')
45
46 if __name__ == "__main__": main()

```

```
rango(A)=4 rango(Ab)=4 n=4
solucion unica
```

[37.29281768 18.64640884 17.12707182 26.93370166]

La solución es

Flujo	Flujo molar
A	37.29
B	18.64
C	17.12
D	26.93

■

4.1.3 Sistemas homogéneos

Un sistema de ecuaciones lineales homogéneo se caracteriza porque el vector de constantes es cero $b = 0$

$$Ax = 0$$

En un sistema homogéneo siempre existe la solución trivial $x_1 = 0, x_2 = 0, x_3 = 0, \dots, x_n = 0$

En otros casos existen muchas soluciones, donde se debe seleccionar una de ellas para aplicar al problema. El resultado del paso de la reducción genera una matriz con menos ecuaciones que incógnitas, por lo tanto, el conjunto solución queda en términos de las mismas incógnitas, de manera que la definición de una de ellas da como resultado el resto de las variables, así hay múltiples soluciones.

Una aplicación muy común de un sistema de ecuaciones lineales homogénea es el balanceo de ecuaciones químicas.

■ **Ejemplo 4.3 — Método de reducción de matriz escalonada.** Balancear la siguiente ecuación química:



Solución

Balance por elemento

$$\begin{aligned} \text{Ag: } & x_1 = 2x_3 \\ \text{N: } & x_1 = x_4 \\ \text{O: } & 3x_1 + 4x_2 = 4x_3 + 3x_4 \\ \text{K: } & 2x_2 = x_4 \\ \text{Cr: } & x_2 = x_3 \end{aligned}$$

El sistema de ecuaciones homogénea es

$$\left| \begin{array}{cccc|c} x_1 & 0 & -2x_3 & 0 & 0 \\ x_1 & 0 & 0 & -x_4 & 0 \\ 3x_1 & 4x_2 & -4x_3 & -3x_4 & 0 \\ 0 & 2x_2 & 0 & -x_4 & 0 \\ 0 & x_2 & -x_3 & 0 & 0 \end{array} \right|$$

El sistema es homogéneo, aplicar la eliminación de Gauss-Jordan genera un sistema con menos renglones.

Programa 4.3. Método de reducción escalonada

```

1 import numpy as np
2 import sympy as sp
3
4 def main():
5     a=np.array([[1 , 0 , -2, 0 ], \
6                [ 1 , 0 , 0, -1 ], \
7                [ 3 , 4 , -4, -3 ], \
8                [ 0 , 2 , 0, -1 ], \
9                [ 0 , 1 , -1, 0 ]])
10    b=np.array([0,0,0,0,0])
11    n,c=np.shape(a)
12    r=np.linalg.matrix_rank(a)
13    ab=np.c_[a,b]
14    ra=np.linalg.matrix_rank(ab)
15    print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,c))
16
17    if (r==ra==c):
18        print('solución única')
19        x=np.linalg.solve(a, b)
20        print(x)
21    if (r==ra<c):
22        print('múltiples soluciones')
23        x = sp.Matrix(a).rref()
24        print(x)
25    if (r<ra):
26        print('sin solución')
27
28 if __name__ == "__main__": main()

```

```

rango(A)=3 rango(Ab)=3 n=4
múltiples soluciones
(Matrix([
[1.0, 0, 0, -1.0],
[ 0, 1.0, 0, -0.5],
[ 0, 0, 1.0, -0.5],
[ 0, 0, 0, 0],
[ 0, 0, 0, 0]]), [0, 1, 2])

```

El resultado es una matriz escalonada con tres renglones y cuatro incógnitas (x_1, x_2, x_3, x_4).

La primera ecuación

$$x_1 - x_4 = 0$$

$$x_1 = x_4$$

Segunda ecuación

$$x_2 - \frac{1}{2}x_4 = 0$$

$$x_2 = \frac{1}{2}x_4$$

Tercera ecuación

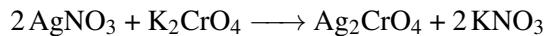
$$x_3 - \frac{1}{2}x_4 = 0$$

$$x_3 = \frac{1}{2}x_4$$

El conjunto solución es entonces:

$$(x_1, x_2, x_3, x_4) = \left(x_4, \frac{1}{2}x_4, \frac{1}{2}x_4, x_4 \right)$$

Cualquier valor dado de x_4 genera un conjunto solución, de acuerdo con lo especificado, que resuelve el sistema, por lo tanto, se tienen múltiples soluciones; para el caso que nos ocupa, los valores del conjunto solución deben ser enteros y deben ser el múltiplo menor, de manera que, si designamos el valor de $x_4 = 2$, el conjunto solución es $(2, 1, 1, 2)$ y la ecuación química queda balanceada como:



■

4.1.4 Sistemas rectangulares

Los sistemas rectangulares son sistemas con más ecuaciones que incógnitas (sobre determinado) o menos ecuaciones que incógnitas (sub determinado) $m \neq n$. Cualquiera de los casos resulta en un conjunto solución que se expresa en términos de una de las variables

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & \cdots & A_{1,m} \\ A_{1,1} & A_{1,2} & A_{1,3} & \cdots & A_{1,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & A_{n,3} & \cdots & A_{n,m} \end{pmatrix}$$

El paso de la reducción elimina las ecuaciones que son múltiples y el resultado se expresa como un vector que sigue ciertas proporciones de una de las variables.

■ **Ejemplo 4.4 — Método de sistemas rectangulares.** Balancear la siguiente ecuación química:



Solución

Balance por elemento

$$\begin{array}{ll}
 \text{Pb: } & x_1 = 3x_5 \quad & x_1 - 3x_5 = 0 \\
 \text{N: } & 6x_1 = x_6 \quad & 6x_1 - x_6 = 0 \\
 \text{Cr: } & x_2 = 2x_3 \quad & x_2 - 2x_3 = 0 \\
 \text{Mn: } & 2x_2 = x_4 \quad & 2x_2 - x_4 = 0 \\
 \text{O: } & 8x_2 = 3x_3 + 2x_4 + 4x_5 + x_6 \quad & 8x_2 - 3x_3 - 2x_4 - 4x_5 - x_6 = 0
 \end{array}$$

El sistema es rectangular porque tiene 5 ecuaciones y 6 incógnitas.

$$\left(\begin{array}{cccccc}
 1 & 0 & 0 & 0 & -3 & 0 \\
 6 & 0 & 0 & 0 & 0 & -1 \\
 0 & 1 & -2 & 0 & 0 & 0 \\
 0 & 2 & 0 & -1 & 0 & 0 \\
 0 & 8 & -3 & -2 & -4 & -1
 \end{array} \right)$$

Programa 4.4. Método de sistemas rectangulares

```

1 import numpy as np
2 import sympy as sp
3
4 def main():
5     a=sp.Matrix([[1, 0, 0, 0,-3, 0 ], \
6                 [ 6 , 0, 0, 0, 0,-1 ], \
7                 [ 0 , 1,-2, 0, 0, 0 ], \
8                 [ 0 , 2, 0,-1, 0, 0 ], \
9                 [ 0 , 8,-3,-2,-4,-1 ]])
10    b=sp.Matrix([0,0,0,0,0,0])
11    xvars=sp.symbols('x1,x2,x3,x4,x5,x6')
12    x=sp.Matrix(xvars)
13    sol=sp.solve(a*x-b,xvars)
14    print('Solucion:{}' .format(sol))
15
16
17 if __name__ == "__main__": main()

```

```
Solucion:{x2: 22*x6/45, x4: 44*x6/45, x3: 11*x6/45, x5: x6/18, x1:←
x6/6}
```

El resultado es una expresión con los valores del conjunto solución ($x_1, x_2, x_3, x_4, x_5, x_6$).

$$x_1 = \frac{1}{6}x_6$$

$$x_2 = \frac{22}{45}x_6$$

$$x_3 = \frac{11}{45}x_6$$

$$x_4 = \frac{44}{45}x_6$$

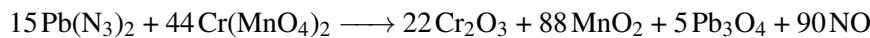
$$x_5 = \frac{1}{18}x_6$$

$$x_6 = x_6$$

El conjunto solución es entonces

$$(x_1, x_2, x_3, x_4, x_5, x_6) = \left(\frac{1}{6}x_6, \frac{22}{45}x_6, \frac{11}{45}x_6, \frac{44}{45}x_6, \frac{1}{18}x_6, x_6 \right)$$

Cualquier valor dado de x_6 genera un conjunto solución; si designamos el valor de $x_6 = 90$, el conjunto solución es $(15, 44, 22, 88, 5, 90)$ y la ecuación química queda balanceada como



■

4.1.5 Descomposición LU

La eliminación de Gauss aplica la reducción de la matriz extendida de coeficientes A y el vector de constantes b . Si en un segundo sistema se tiene una misma matriz A con un vector de constantes b distinto, se usarán los mismos factores de reducción k para resolver ambos sistemas.

El método de descomposición LU crea una matriz L con los factores k de reducción, la matriz U es la matriz triangular superior resultado de la reducción. Si se aplica en un siguiente paso los factores de reducción que se guardaron en L al vector de constantes b , se obtendrá el mismo resultado que en el método de eliminación de Gauss. La misma matriz L se aplica a otro vector de constantes b , sin la necesidad de volver a calcular los factores k , porque ya fueron almacenados en L .

Al multiplicar LU obtenemos la misma matriz A , de tal manera que

$$[Ax = b] = [\{LU\}x = b]$$

donde

$$U = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}, \quad L = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ k_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ k_{n,1} & k_{n,2} & \cdots & 1 \end{pmatrix}$$

Donde se cumplen las siguientes propiedades $A = LU$ y $Lb = d$.

De esta manera se obtiene la matriz triangular U y el vector d que son necesarios para el segundo paso de la sustitución hacia atrás. Lo conveniente del método de descomposición LU es que independiza los factores de reducción k y se pueden aplicar a distintos vectores de constantes b .

■ **Ejemplo 4.5 — Método de descomposición LU.** Una tubería de acero inoxidable, la cual tiene un diámetro interior de $d_1 = 20 \text{ mm}$ y un diámetro exterior de $d_2 = 25 \text{ mm}$ está cubierta con un material de aislamiento con un espesor de 35 mm. La temperatura de saturación del vapor que fluye por la tubería es de $T_s = 150 \text{ C}$ y el coeficiente de calor convectivo interior y exterior son $h_i = 1500 \text{ W/m}^2\text{K}$ y $h_o = 5 \text{ W/m}^2\text{K}$, respectivamente. La conductividad térmica de la tubería y el aislante son de $k_s = 45 \text{ W/mK}$ y $k_i = 0.065 \text{ W/mK}$, respectivamente. Si la temperatura ambiente es $T_a = 25 \text{ C}$, calcule la temperatura de la pared interna T_1 y externa T_2 de la tubería y la temperatura de la pared exterior del aislante T_3 .[1]

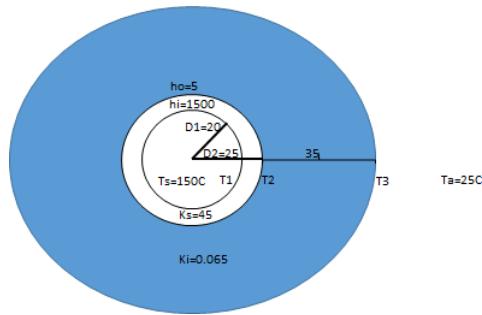


Figura 4.4. Tubería aislada

Solución

Calor transferido del vapor a la tubería

$$h\pi d_1(T_s - T_1) = \frac{T_1 - T_2}{\ln(\frac{d_2}{d_1})/(2\pi k_s)}$$

Calor transferido de la tubería al aislante

$$\frac{T_1 - T_2}{\ln(\frac{d_2}{d_1})/(2\pi k_s)} = \frac{T_2 - T_3}{\ln(\frac{d_3}{d_2})/(2\pi k_i)}$$

Calor transferido del aislante al medio ambiente

$$\frac{T_2 - T_3}{\ln(\frac{d_3}{d_2})/(2\pi k_i)} = h_0\pi d_3(T_3 - T_a)$$

Reordenando las ecuaciones:

$$\begin{aligned} \left(\frac{2k_s}{\ln(\frac{d_2}{d_1})} + h_i d_1 \right) T_1 - \left(\frac{2k_s}{\ln(\frac{d_2}{d_1})} \right) T_2 &= h_i d_1 T_s \\ \left(\frac{k_s}{\ln(\frac{d_2}{d_1})} \right) T_1 - \left(\frac{k_s}{\ln(\frac{d_2}{d_1})} + \frac{k_i}{\ln(\frac{d_3}{d_2})} \right) T_2 + \left(\frac{k_i}{\ln(\frac{d_3}{d_2})} \right) T_3 &= 0 \\ \left(\frac{2k_i}{\ln(\frac{d_3}{d_2})} \right) T_2 - \left(\frac{2k_i}{\ln(\frac{d_3}{d_2})} + h_o d_3 \right) T_3 &= -h_o d_3 T_a \end{aligned}$$

El sistema de ecuaciones lineales resultante es:

$$\begin{pmatrix} \left(\frac{2k_s}{\ln(\frac{d_2}{d_1})} + h_i d_1 \right) & - \left(\frac{2k_s}{\ln(\frac{d_2}{d_1})} \right) & 0 & h_i d_1 T_s \\ \left(\frac{k_s}{\ln(\frac{d_2}{d_1})} \right) & - \left(\frac{k_s}{\ln(\frac{d_2}{d_1})} + \frac{k_i}{\ln(\frac{d_3}{d_2})} \right) & \left(\frac{k_i}{\ln(\frac{d_3}{d_2})} \right) & 0 \\ 0 & \left(\frac{2k_i}{\ln(\frac{d_3}{d_2})} \right) & - \left(\frac{2k_i}{\ln(\frac{d_3}{d_2})} + h_o d_3 \right) & -h_o d_3 T_a \end{pmatrix}$$

La solución del sistema son las temperaturas que se buscan

Programa 4.5. Método de descomposición LU

```

1 import numpy as np
2 import scipy as sp
3
4 def main():
5     d1=20/1000 #diametro interno de la tuberia en m
6     d2=25/1000 #diametro interno de la tuberia en m
7     di=35/1000 #espesor del aislante en m
8     Ts=150+273 #temperatura del vapor en K
9     Ta=25+273 #temperatura ambiente en K
10    hi=1500 #coeficiente interior de calor convectivo
11    ho=5 #coeficiente exterior de calor convectivo
12    ks=45 #coeficiente de conductividad de calor de la ←
13        tuberia
14    ki=0.065 #coeficiente de conductividad de calor del aislante
15    d3=d2+2*di #diametro incluyendo el aislante
16
16    a=np.array([[2*ks/np.log(d2/d1)+hi*d1,\n
17                  -2*ks/np.log(d2/d1),\n
18                  0],\n
19                  [ks/np.log(d2/d1),\n
20                  -ks/np.log(d2/d1)-ki/np.log(d3/d2),\n
21                  ki/np.log(d3/d2)],\n
22                  [0,\n
23                  2*ki/np.log(d3/d2),\n
24                  -2*ki/np.log(d3/d2)-ho*d3]])
25    b=np.array([hi*d1*Ts, 0, -ho*d3*Ta])
26    n,c=np.shape(a)
27    r=np.linalg.matrix_rank(a)
```

```

28     ab=np.c_[a,b]
29     ra=np.linalg.matrix_rank(ab)
30
31     print('rango(A)={} rango(AB)={} n={}'.format(r,ra,c))
32
33     if (r==ra==c):
34         print('solución única')
35         lu, piv = sp.linalg.lu_factor(a)
36         x = sp.linalg.lu_solve((lu, piv), b)
37         print(lu)
38         print(x)
39
40     if (r==ra<c):
41         print('multiples soluciones')
42
43     if (r<ra):
44         print('sin solución')
45
46 if __name__ == "__main__": main()

```

```

rango(A)=3 rango(AB)=3 n=3
solucion única
[[ 4.33327811e+02 -4.03327811e+02  0.00000000e+00]
 [ 4.65384174e-01 -1.40102143e+01  4.86890997e-02]
 [ 0.00000000e+00 -6.95051461e-03 -5.72039785e-01]]
[ 422.66425778  422.63928488  319.20477185]

```

El conjunto solución es

$$\begin{aligned} T_1 &= 422.66K \\ T_2 &= 422.63K \\ T_3 &= 319.20K \end{aligned}$$

Observe, en el ejemplo anterior, que si cambiamos los datos de entrada del sistema como son la temperatura del vapor o la temperatura del ambiente, los únicos valores que cambian es el vector de constantes b , de tal manera que A es la misma matriz y, por lo tanto, los mismos factores de reducción, es decir, la matriz L es la misma, por lo que sólo se debe hacer el paso de aplicar la matriz L al nuevo vector b para obtener la nueva solución. L se convierte en un factor de reducción que se aplica al vector b que corresponda.

4.1.6 Inversa multiplicación

En un sistema de ecuaciones lineales, representado por la ecuación $Ax = b$ la incógnita a resolver es el vector x , si despejamos este vector de la ecuación, tendremos la solución del sistema.

$$Ax = b$$

$$A^{-1}Ax = A^{-1}b$$

$$x = A^{-1}b$$

Donde A^{-1} es la inversa de A , entonces la solución se obtiene usando la inversa A^{-1} y multiplicarla por el vector de constantes b

■ **Ejemplo 4.6 — Método de inversa-multiplicación.** Un arreglo de cuatro reactores en los cuales se lleva a cabo la reacción $A \xrightarrow{k_i} B$ Donde k_i es la constante de reacción en cada reactor. [1]

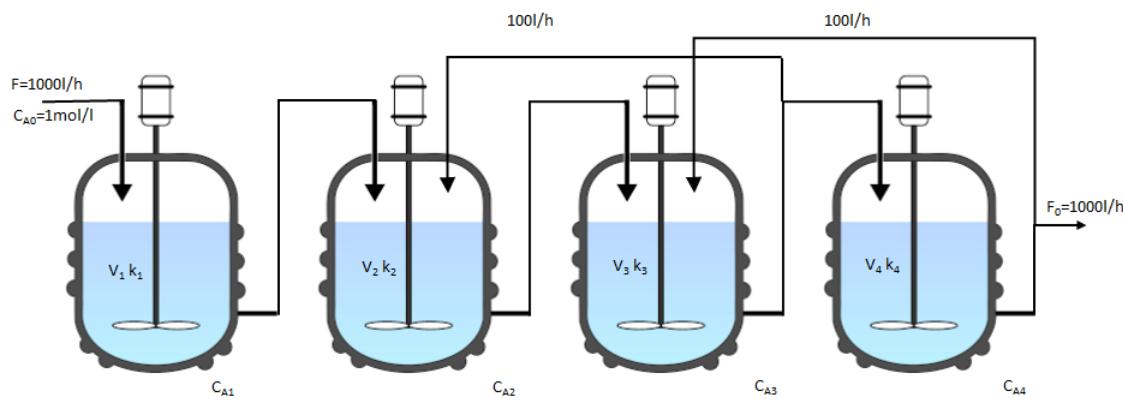


Figura 4.5. Arreglo de reactores agitados

Reactor	Volumen	Constante k
1	1000	0.2
2	1200	0.1
3	300	0.3
4	600	0.5

La reacción en cada tanque es de primer orden y la velocidad de reacción es $r = VkC_A$. Determinar la concentración de salida de cada tanque $C_{A1}, C_{A2}, C_{A3}, C_{A4}$

Solución

$$\begin{aligned} \text{Balance de masa del reactor 1: } & 1000C_{A0} = 1000C_{A1} + V_1 k_1 C_{A1} \\ \text{Balance de masa del reactor 2: } & 1000C_{A1} + 1000C_{A3} = 1100C_{A2} + V_2 k_2 C_{A2} \\ \text{Balance de masa del reactor 3: } & 1100C_{A2} + 100C_{A4} = 1200C_{A3} + V_3 k_3 C_{A3} \\ \text{Balance de masa del reactor 4: } & 1100C_{A3} = 1100C_{A4} + V_4 k_4 C_{A4} \end{aligned}$$

El sistema de ecuaciones lineales resultante es:

$$A = \begin{pmatrix} 1000 + V_1 k_1 & 0 & 0 & 0 \\ 1000 & -(1100 + V_2 k_2) & 100 & 0 \\ 0 & 1100 & -(1200 + V_3 k_3) & 100 \\ 0 & 0 & 1100 & -(1100 + V_4 k_4) \end{pmatrix}, \quad x = \begin{pmatrix} C_{A1} \\ C_{A2} \\ C_{A3} \\ C_{A4} \end{pmatrix},$$

$$b = \begin{pmatrix} 1000 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

La solución del sistema son las concentraciones en cada reactor.

Programa 4.6. Método de inversa-multiplicación

```

1 import numpy as np
2 import scipy as sp
3
4 def main():
5     v=np.array([1000,1200,300,600])
6     k=np.array([0.2,0.1,0.3,0.5])
7
8     a=np.array([[1000+v[0]*k[0],0,0,0],\
9                 [1000,-(1100+v[1]*k[1]),100,0],\
10                [0,1100,-(1200+v[2]*k[2]),100],\
11                [0,0,1100,-(1100+v[3]*k[3])]])
12    b=np.array([1000,0,0,0])
13
14    n,c=np.shape(a)
15    r=np.linalg.matrix_rank(a)
16    ab=np.c_[a,b]
17    ra=np.linalg.matrix_rank(ab)
18
19    print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,c))
20
21    if (r==ra==c):
22        print('solución única')
23        A_inv=np.linalg.inv(a)
24        x=np.dot(A_inv,b)
25        print(A_inv)
26        print(x)
27
28    if (r==ra<c):
29        print('múltiples soluciones')
30
31    if (r<ra):
32        print('sin solución')
33
34 if __name__ == "__main__": main()

```

```
rango(A)=4 rango(AB)=4 n=4
solucion unica
[[ 8.33333333e-04    0.00000000e+00    0.00000000e+00    0.00000000e+00]
 [ 7.37986828e-04   -8.85584193e-04   -7.31024688e-05   -5.22160491e-06]
 [ 6.70105964e-04   -8.04127157e-04   -8.91850119e-04   -6.37035799e-05]
 [ 5.26511829e-04   -6.31814194e-04   -7.00739379e-04   -7.64338527e-04]]
[ 0.83333333    0.73798683    0.67010596    0.52651183]
```

El conjunto solución es

$$\begin{aligned} C_{A_1} &= 0.8333 \\ C_{A_2} &= 0.7379 \\ C_{A_3} &= 0.6701 \\ C_{A_4} &= 0.5265 \end{aligned}$$

■

4.1.7 Sistemas de ecuaciones lineales múltiples

En la ecuación $Ax = b$, A representa los coeficientes de cómo funciona el sistema, el comportamiento del sistema; b es el vector de las entradas y x es la respuesta que tiene el sistema ante las entradas.

Si en un sistema se cambian las entradas, el vector b , el sistema responderá con un vector x distinto. Estos sistemas múltiples son comunes en Ingeniería Química ya que se suelen cambiar las entradas para el mismo sistema y se desea conocer el resultado correspondiente.

El vector b entonces se convierte en la matriz B , donde cada columna representa una entrada distinta. El conjunto solución será entonces una matriz X , donde cada columna es la respuesta a la correspondiente entrada de B .

Se aplican los métodos directos a estos sistemas para obtener la matriz solución X .

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix}, X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,m} \end{pmatrix},$$

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix}$$

■ **Ejemplo 4.7 — Sistema de ecuaciones lineales múltiples.** Se tiene una placa metálica de forma triangular que la rodean tres temperaturas distintas como se muestra en la **figura 4.6a**, suponga

que se cambian las temperaturas como se indica en la **figura 4.6b**. Se desea calcular la temperatura interna de las 4 secciones que la componen.

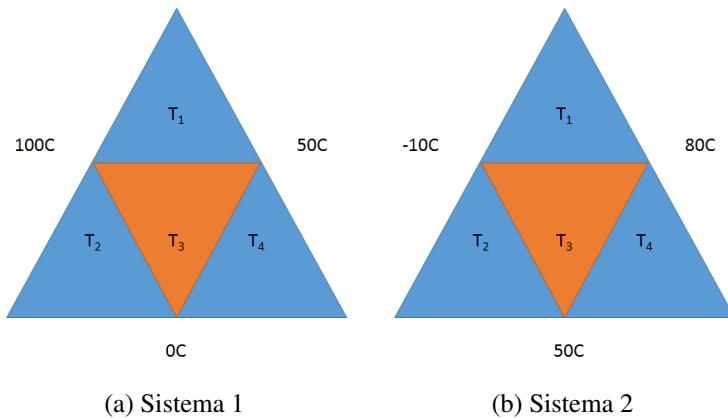


Figura 4.6. Placa triangular

La temperatura de cada sección es el promedio de las tres temperaturas que la rodean.

Solución

Para el sistema 1

$$\text{Sección 1: } T_1 = (100 + 50 + T_3)/3$$

$$\text{Sección 2: } T_2 = (100 + 0 + T_3)/3$$

$$\text{Sección 3: } T_3 = (T_1 + T_2 + T_4)/3$$

$$\text{Sección 4: } T_4 = (50 + 0 + T_3)/3$$

El sistema de ecuaciones lineales resultante es:

$$\begin{pmatrix} 3T_1 & 0 & -T_3 & 0 & 150 \\ 0 & 3T_2 & -T_3 & 0 & 100 \\ -T_1 & -T_2 & 3T_3 & -T_4 & 0 \\ 0 & 0 & -T_3 & 3T_4 & 50 \end{pmatrix}$$

Para el sistema 2

$$\text{Sección 1: } T_1 = (-10 + 80 + T_3)/3$$

$$\text{Sección 2: } T_2 = (-10 + 50 + T_3)/3$$

$$\text{Sección 3: } T_3 = (T_1 + T_2 + T_4)/3$$

$$\text{Sección 4: } T_4 = (50 + 80 + T_3)/3$$

El sistema de ecuaciones lineales resultante es:

$$\begin{pmatrix} 3T_1 & 0 & -T_3 & 0 & 70 \\ 0 & 3T_2 & -T_3 & 0 & 40 \\ -T_1 & -T_2 & 3T_3 & -T_4 & 0 \\ 0 & 0 & -T_3 & 3T_4 & 130 \end{pmatrix}$$

Se observa que es la misma matriz de coeficientes A y el vector b del sistema 1 es distinto al vector b del sistema 2. Por lo tanto, ambos sistemas se pueden representar de la siguiente manera.

$$A = \begin{pmatrix} 3 & 0 & -1 & 0 \\ 0 & 3 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ 0 & 0 & -1 & 3 \end{pmatrix}, X = \begin{pmatrix} T_{1,1} & T_{1,2} \\ T_{2,1} & T_{2,2} \\ T_{3,1} & T_{3,2} \\ T_{4,1} & T_{4,2} \end{pmatrix},$$

$$B = \begin{pmatrix} 150 & 70 \\ 100 & 40 \\ 0 & 0 \\ 50 & 130 \end{pmatrix}$$

Se aplica el método de la inversa multiplicación, para ambos sistemas se tiene la misma matriz inversa A^{-1} y la multiplicación se hace con la matriz B .

Programa 4.7. Método de inversa-multiplicación múltiple

```

1 import numpy as np
2 import scipy as sp
3
4 def main():
5     a=np.array([[3 , 0 , -1 , 0 ],\
6                 [0 , 3 , -1 , 0 ],\
7                 [-1 , -1 , 3 , -1 ],\
8                 [0 , 0 , -1 , 3 ]])
9     b=np.array([[150 , 70],\
10                [100 , 40],\
11                [0 , 0],\
12                [50 , 130]])
13     A_inv=np.linalg.inv(a)
14     x=np.dot(A_inv,b)
15     print(A_inv)
16     print(x)
17
18 if __name__ == "__main__": main()

```

```

[[ 0.38888889  0.05555556  0.16666667  0.05555556]
 [ 0.05555556  0.38888889  0.16666667  0.05555556]
 [ 0.16666667  0.16666667  0.5         0.16666667]
 [ 0.05555556  0.05555556  0.16666667  0.38888889]]
[[ 66.66666667  36.66666667]
 [ 50.          26.66666667]
 [ 50.          40.          ]
 [ 33.33333333  56.66666667]]
```

El conjunto solución es

	Sistema 1	Sistema 2
T_1 =	66.66	T_1 = 36.66
T_2 =	50.00	T_2 = 26.66
T_3 =	50.00	T_3 = 40.00
T_4 =	33.33	T_4 = 56.66

■

4.1.8 Regla de Cramer

La regla de Cramer se aplica a sistemas compatibles determinados, es decir, se aplica sólo a sistemas que tienen solución única. Para validar que tienen solución única, se requiere obtener el determinante de la matriz de coeficientes ΔA , si éste es distinto de cero, entonces tienen solución única, de lo contrario no tiene solución o tiene múltiples soluciones.

El siguiente paso es sustituir la primera columna de A por el vector b y obtener el determinante de esta matriz modificada ΔA_1 , el valor de la primera incógnita se calcula como:

$$x_1 = \frac{\Delta A_1}{\Delta A}$$

Para la segunda incógnita se sustituye la segunda columna de A por el vector b , se obtiene el determinante ΔA_2 , el valor de la segunda incógnita ahora se obtiene como:

$$x_2 = \frac{\Delta A_2}{\Delta A}$$

■ **Ejemplo 4.8 — Regla de Cramer.** Obtener el conjunto solución del siguiente sistema usando la regla de Cramer.

$$\begin{aligned} 3x_1 + 2x_2 - x_3 &= 1 \\ 2x_1 - 2x_2 + 4x_3 &= -2 \\ -x_1 + \frac{1}{2}x_2 - x_3 &= 0 \end{aligned}$$

Solución

Programa 4.8. Regla de Cramer

```

1 import numpy as np
2 import scipy as sp
3
4 def cramer(A,b):
5     dA=np.linalg.det(A)
6     _,n=np.shape(A)
7     if (dA==0):
8         raise ValueError ('El sistema no tiene solución por la ←
                           regla de Cramer')

```

```

9     x=np.zeros(n)
10    for i in range(n):
11        di=np.copy(A)
12        di[:,i]=b
13        x[i]=np.linalg.det(di)/dA
14
15    return x
16
17 def main():
18     A=np.array([[3,2,-1],[2,-2,4],[-1,1/2,-1]])
19     b=np.array([1,-2,0])
20     x=cramer(A,b)
21     print(x)
22
23 if __name__ == "__main__": main()

```

```
[ 1. -2. -2.]
```

El conjunto solución es

$$\begin{aligned}x_1 &= 1 \\x_2 &= -2 \\x_3 &= -2\end{aligned}$$

■

4.2 Métodos iterativos

Los métodos iterativos se caracterizan por encontrar el conjunto solución por medio de aproximaciones que parten de una suposición inicial, la condición de paro se da cuando se cumple una condición que satisface para aceptar la aproximación como correcta.

4.2.1 Jacobi

En el método de Jacobi se descompone la matriz de coeficientes A en dos matrices, una se compone de los elementos de la diagonal $D_{i,i} = A_{i,i}$. La matriz D no debe tener elementos cero, si fuera así, se deben intercambiar los renglones para evitar el caso. La otra matriz es el resto de los elementos, es decir, los elementos que no son la diagonal $R_{i,j} = A_{i,j}$, de tal manera que $A = D + R$.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix} \quad D = \begin{pmatrix} A_{1,1} & 0 & \cdots & 0 \\ 0 & A_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{n,n} \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & 0 & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & 0 \end{pmatrix}$$

Por lo tanto, la ecuación que describe un sistema de ecuaciones lineales $Ax = b$ ahora se expresa como $[D + R]x = b$.

$$\begin{aligned} Dx + Rx &= b \\ Dx &= b - Rx \\ D^{-1}Dx &= D^{-1}(b - Rx) \\ x &= D^{-1}(b - Rx) \end{aligned}$$

Ecuación iterativa de Jacobi

$$x^{(k+1)} = D^{-1} (b - Rx^{(k)}) \quad (4.1)$$

La ecuación iterativa de Jacobi se inicia con valores supuestos $x^{(k)}$ y la ecuación iterativa obtiene los siguientes valores $x^{(k+1)}$. El proceso se repite hasta que la diferencia absoluta entre $x^{(k+1)}$ y $x^{(k)}$ sea mínima.

Un proceso alternativo es despejar x_i de la i-ésima ecuación, luego se supone el vector inicial $x^{(0)}$ y se hacen sustituciones simultáneas, de tal manera que para todas las ecuaciones se tiene la ecuación iterativa:

Ecuación iterativa de Jacobi

$$x_i^{k+1} = \left(\frac{1}{a_{i,i}} \right) \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n \left(a_{i,j} x_j^{(k)} \right) \right) \quad (4.2)$$

■ **Ejemplo 4.9 — Método de Jacobi.** En una columna de 5 platos, se requiere absorber benceno contenido en una corriente de gas V, con aceite L que circula a contracorriente del gas. Considérese que el benceno transferido no altera sustancialmente el número de moles de V y L fluyendo a

contracorriente, que la relación de equilibrio está dada por la ley de Henry ($y = mx$) y que la columna opera a régimen permanente. Calcule la composición en cada plato.[3]

Datos: $V = 100$ moles/min
 $L = 500$ moles/min
 $y_0 = 0.09$ fracción molar de benceno en V
 $x_0 = 0.0$ fracción de benceno en L (el aceite entra por el domo sin benceno)
 $m = 0.12$

Solución

Entradas = Salidas + Acumulación

Los balances de materia para el benceno en cada plato son:

Plato 5

$$\begin{aligned} Lx_0 + Vy_4 &= Lx_5 + Vy_5 \\ L(x_0 - x_5) + V(y_4 - y_5) &= 0 \\ 500(0 - x_5) + 100(0.12x_4 - 0.12x_5) &= 0 \\ -512x_5 + 12x_4 &= 0 \end{aligned}$$

Plato 4

$$\begin{aligned} Lx_5 + Vy_3 &= Lx_4 + Vy_4 \\ L(x_5 - x_4) + V(y_3 - y_4) &= 0 \\ 500(x_5 - x_4) + 100(0.12x_3 - 0.12x_4) &= 0 \\ 500x_5 - 512x_4 + 12x_3 &= 0 \end{aligned}$$

Plato 3

$$\begin{aligned} Lx_4 + Vy_2 &= Lx_3 + Vy_3 \\ L(x_4 - x_3) + V(y_2 - y_3) &= 0 \\ 500(x_4 - x_3) + 100(0.12x_2 - 0.12x_3) &= 0 \\ 500x_4 - 512x_3 + 12x_2 &= 0 \end{aligned}$$

Plato 2

$$\begin{aligned} Lx_3 + Vy_1 &= Lx_2 + Vy_2 \\ L(x_3 - x_2) + V(y_1 - y_2) &= 0 \\ 500(x_3 - x_2) + 100(0.12x_1 - 0.12x_2) &= 0 \\ 500x_3 - 512x_2 + 12x_1 &= 0 \end{aligned}$$

Plato 1

$$\begin{aligned} Lx_2 + Vy_0 &= Lx_1 + Vy_1 \\ L(x_2 - x_1) + V(y_0 - y_1) &= 0 \\ 500(x_2 - x_1) + 100(0.09 - 0.12x_1) &= 0 \\ -500x_2 + 512x_1 &= 9 \end{aligned}$$

Sistema

$$\begin{pmatrix} -512x_5 & 12x_4 & 0 & 0 & 0 & 0 \\ 500x_5 & -512x_4 & 12x_3 & 0 & 0 & 0 \\ 0 & 500x_4 & -512x_3 & 12x_2 & 0 & 0 \\ 0 & 0 & 500x_3 & -512x_2 & 12x_1 & 0 \\ 0 & 0 & 0 & -500x_2 & 512x_1 & 9 \end{pmatrix}$$

Programa 4.9. Método de Jacobi

```

1 import numpy as np
2 import scipy as sp
3
4 def jacobi(A,b,x,imax=100,tol=1e-8):
5     D=np.diag(A)
6     R=A-np.diagflat(D)
7     cumple = False
8     k=0
9     while (not cumple and k<imax):
10         xk1=(b-np.dot(R,x))/D
11         norma=np.linalg.norm(x-xk1)
12         print('iteracion:{}->{} norma {}'.format(k,x,norma))
13         cumple=norma<tol
14         x=xk1.copy()
15         k+=1
16
17     if k<imax:
18         return x
19     else:
20         raise ValueError ('El sistema no converge')
21
22 def main():
23     A = np.array([[-512,12,0,0,0], \
24                  [500,-512,12,0,0], \
25                  [0,500,-512,12,0], \
26                  [0,0,500,-512,12], \
27                  [0,0,0,-500,512]])
28     b = np.array([0,0,0,0,9])
29     x = np.array([0,0,0,0,0])
30     x=jacobi(A,b,x)
31     print('Solucion: ',x)
32
33 if __name__ == "__main__": main()
```

```

iteracion:0->[0 0 0 0 0] norma 0.017578125
iteracion:1->[-0.          -0.          -0.          -0.          -0.] ←
    0.01757812] norma 0.0004119873046875
iteracion:2->[-0.          -0.          -0.          -0.          -0.] ←
    0.01757812] norma 0.00040244720698264793
iteracion:3->[ -0.00000000e+00   -0.00000000e+00   9.65595245e-06 ←
    4.11987305e-04
```

```

1.79804564e-02] norma 1.8860639955397488e-05
iteracion:4->[ -0.00000000e+00 2.26311386e-07 9.65595245e-06 ←
    4.30846587e-04
1.79804564e-02] norma 1.842919899944634e-05
iteracion:5->[ 5.30417310e-09 2.26311386e-07 1.03189741e-05 ←
    4.30846587e-04
1.79988736e-02] norma 1.0793356680990857e-06
iteracion:6->[ 5.30417310e-09 2.47030812e-07 1.03189741e-05 ←
    4.31925724e-04
1.79988736e-02] norma 1.0548275284568573e-06
iteracion:7->[ 5.78978465e-09 2.47030812e-07 1.03645002e-05 ←
    4.31925724e-04
1.79999275e-02] norma 6.917571784739033e-08
iteracion:8->[ 5.78978465e-09 2.48572059e-07 1.03645002e-05 ←
    4.31994882e-04
1.79999275e-02] norma 6.760995881613473e-08
iteracion:9->[ 5.82590764e-09 2.48572059e-07 1.03676262e-05 ←
    4.31994882e-04
1.79999950e-02] norma 4.6369458180272845e-09
Solucion: [ 5.82590764e-09 2.48680602e-07 1.03676262e-05 ←
    4.31999518e-04
    1.79999950e-02]

```

El conjunto solución es

$$\begin{aligned}
 Plato_1 &= 1.79999950e-02 \\
 Plato_2 &= 4.31999518e-04 \\
 Plato_3 &= 1.03676262e-05 \\
 Plato_4 &= 2.48680602e-07 \\
 Plato_5 &= 5.82590764e-09
 \end{aligned}$$

■

4.2.2 Gauss-Seidel

El método de Gauss-Seidel o el método de Liebmann es un método iterativo que separa la matriz de coeficientes A en dos matrices, una matriz triangular inferior L que contiene los elementos de A que están en la diagonal inferior (la diagonal no debe contener ceros) y la matriz U que contiene los elementos de A que están sobre la diagonal.

$$\begin{aligned}
 A &= \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix} & L &= \begin{pmatrix} A_{1,1} & 0 & \cdots & 0 \\ A_{2,1} & A_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix} \\
 U &= \begin{pmatrix} 0 & A_{1,2} & \cdots & A_{1,n} \\ 0 & 0 & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}
 \end{aligned}$$

De tal manera que $A = L + U$. La ecuación que describe un sistema de ecuaciones lineales $Ax = b$ ahora se expresa como $[L + U]x = b$.

$$\begin{aligned} Lx + Ux &= b \\ Lx &= b - Ux \\ L^{-1}Lx &= L^{-1}(b - Ux) \\ x &= L^{-1}(b - Ux) \end{aligned}$$

Ecuación iterativa de Gauss-Seidel

$$x^{(k+1)} = L^{-1} (b - Ux^{(k)}) \quad (4.3)$$

La ecuación iterativa de Gauss-Seidel se inicia con valores supuestos $x^{(k)}$ y la ecuación iterativa obtiene los siguientes valores $x^{(k+1)}$. El proceso se repite hasta que la diferencia absoluta entre $x^{(k+1)}$ y $x^{(k)}$ sea mínima.

Un proceso alternativo es usar las sustituciones sucesivas, se supone un vector inicial $x^{(0)}$ y se sustituyen los valores de la iteración anterior $x^{(k-1)}$ y para los valores que ya se calcularon de $x^{(k)}$

Ecuación iterativa de Gauss-Seidel

$$x_i^{k+1} = \left(\frac{1}{a_{i,i}} \right) \left(b_i - \sum_{j=1}^{i-1} (a_{i,j}x_j^{(k+1)}) - \sum_{j=i+1}^n (a_{i,j}x_j^{(k)}) \right) \quad (4.4)$$

■ **Ejemplo 4.10 — Método de Gauss-Seidel.** Resolver el **ejemplo 4.9** con los mismos datos usando el método de Gauss-Seidel

Sistema

$$\begin{pmatrix} -512x_5 & 12x_4 & 0 & 0 & 0 & 0 \\ 500x_5 & -512x_4 & 12x_3 & 0 & 0 & 0 \\ 0 & 500x_4 & -512x_3 & 12x_2 & 0 & 0 \\ 0 & 0 & 500x_3 & -512x_2 & 12x_1 & 0 \\ 0 & 0 & 0 & -500x_2 & 512x_1 & 9 \end{pmatrix}$$

Solución

Programa 4.10. Método de Gauss-Seidel

```
1 import numpy as np
```

```

2 import scipy as sp
3
4 def gaussSeidel(A,b,x,imax=100,tol=1e-8):
5     L=np.tril(A)
6     U=A-L
7     Linv=np.linalg.inv(L)
8     cumple = False
9     k=0
10    while (not cumple and k<imax):
11        xk1=np.dot(Linv,b-np.dot(U,x))
12        norma=np.linalg.norm(x-xk1)
13        print('iteracion:{}->{} norma {}'.format(k,x,norma))
14        cumple=norma<tol
15        x=xk1.copy()
16        k+=1
17
18    if k<imax:
19        return x
20    else:
21        raise ValueError ('El sistema no converge')
22
23 def main():
24     A = np.array([[-512,12,0,0,0],\
25                  [500,-512,12,0,0],\
26                  [0,500,-512,12,0],\
27                  [0,0,500,-512,12],\
28                  [0,0,0,-500,512]])
29     b = np.array([0,0,0,0,9])
30     x = np.array([0,0,0,0,0])
31     x=gaussSeidel(A,b,x)
32     print('Solucion: ',x)
33
34 if __name__ == "__main__": main()

```

```

iteracion:0->[0 0 0 0 0] norma 0.017578125
iteracion:1->[ 0.          0.          0.          0.          0.] ←
0.01757812] norma 0.000575850723898146
iteracion:2->[ 0.          0.          0.          0.          0.] ←
0.01798046] norma 2.8073220164515078e-05
iteracion:3->[  0.00000000e+00   0.00000000e+00   9.65595245e-06 ←
4.30846587e-04
1.79988736e-02] norma 1.663111234926709e-06
iteracion:4->[  0.00000000e+00   2.26311386e-07   1.03189741e-05 ←
4.31925724e-04
1.79999275e-02] norma 1.0896921936418888e-07
iteracion:5->[  5.30417310e-09   2.47030812e-07   1.03645002e-05 ←
4.31994882e-04
1.79999950e-02] norma 7.373382460830872e-09
Solucion: [ 5.78978465e-09   2.48572059e-07   1.03676262e-05 ←
4.31999518e-04

```

```
1.79999995e-02]
```

El conjunto solución es

$$\begin{aligned} Plato_1 &= 1.79999995e-02 \\ Plato_2 &= 4.31999518e-04 \\ Plato_3 &= 1.03676262e-05 \\ Plato_4 &= 2.48572059e-07 \\ Plato_5 &= 5.78978465e-09 \end{aligned}$$

A continuación, el código del método de Gauss-Seidel usando la ecuación iterativa 4.3

Programa 4.11. Método de Gauss-Seidel sustituciones sucesivas

```

1 import numpy as np
2
3 def gaussSeidel(A,b,x,imax=100,tol=1e-8):
4     cumple = False
5     n=A.shape[0]
6     k=0
7
8     while (not cumple and k<imax):
9         xk1 = np.zeros(n)
10        for i in range(n):
11            s1 = np.dot(A[i,:i], xk1[:i])
12            s2 = np.dot(A[i,i+1:], x[i+1:])
13            xk1[i] = (b[i]-s1-s2)/A[i, i]
14
15        norma=np.linalg.norm(x-xk1)
16        print('iteracion:{}->{} norma {}'.format(k,xk1,norma))
17        cumple=norma<tol
18        x=xk1
19        k+=1
20
21    if k<imax:
22        return x
23    else:
24        raise ValueError ('El sistema no converge')
25
26 def main():
27     A = np.array([[-512,12,0,0,0], \
28                  [500,-512,12,0,0], \
29                  [0,500,-512,12,0], \
30                  [0,0,500,-512,12], \
31                  [0,0,0,-500,512.0]])
32     b = np.array([0,0,0,0,9.0])
33     x = np.array([0.0,0.0,0.0,0.0,0.0])
34     x=gaussSeidel(A,b,x)
35     print('Solucion: ',x)
36

```

```
37 if __name__ == "__main__": main()
```

■

4.2.3 Método SOR

En los métodos de Jacobi y Gauss-Seidel se requiere que la matriz de coeficientes sea diagonalmente dominante, es decir, que los elementos de la diagonal $|a_{i,i}|$ deben ser mayores o iguales a la suma de los elementos del mismo renglón y misma columna, esta condición puede cumplirse solamente para algunos elementos, esto no quiere decir que el sistema no vaya a converger.

$$|a_{i,i}| \geq \sum |a_{i,j}|$$

$$|a_{i,i}| \geq \sum |a_{j,i}|$$

Si no se cumplen las condiciones anteriores, entonces el sistema no converge. Una alternativa para resolver un sistema que no es diagonalmente dominante es el método SOR, donde se puede mejorar usando la modificación sugerida por Southwell por la introducción de un factor (llamado factor de relajación), lo que da como resultado el método de sobrerrelajación sucesiva (SOR).

Ecuación iterativa de SOR

$$x_i^{k+1} = \left(\frac{1}{a_{i,i}} \right) \left(b_i - \sum_{j=1}^{i-1} (a_{i,j}x_j^{(k+1)}) - \sum_{j=i+1}^n (a_{i,j}x_j^{(k)}) \right) \omega + (1 - \omega)x^{(k)} \quad (4.5)$$

Donde ω es un factor con los siguientes rangos de valores:

- **Sub-relajación** $0 < \omega < 1$ se logra la convergencia en sistemas divergentes.
- **Sobre-relajación** $1 < \omega < 2$ acelera la convergencia de sistemas convergentes.

■ **Ejemplo 4.11 — Método de SOR.** Resolver el [ejemplo 4.9](#) con los mismos datos con el método SOR

Sistema

$$\begin{pmatrix} -512x_5 & 12x_4 & 0 & 0 & 0 & 0 \\ 500x_5 & -512x_4 & 12x_3 & 0 & 0 & 0 \\ 0 & 500x_4 & -512x_3 & 12x_2 & 0 & 0 \\ 0 & 0 & 500x_3 & -512x_2 & 12x_1 & 0 \\ 0 & 0 & 0 & -500x_2 & 512x_1 & 9 \end{pmatrix}$$

Solución

Programa 4.12. Método de SOR

```
1 import numpy as np
```

```

2
3 def sor(A,b,x,w=1,imax=100,tol=1e-8):
4     cumple = False
5     n=A.shape[0]
6     k=0
7
8     while (not cumple and k<imax):
9         xk1 = np.zeros(n)
10        for i in range(n):
11            s1 = np.dot(A[i,:i], xk1[:i])
12            s2 = np.dot(A[i,i+1:], x[i+1:])
13            xk1[i] = (b[i]-s1-s2)/A[i, i]*w+(1-w)*x[i]
14
15        norma=np.linalg.norm(x-xk1)
16        print('iteracion:{}->{} norma {}'.format(k,xk1,norma))
17        cumple=norma<tol
18        x=xk1
19        k+=1
20
21    if k<imax:
22        return x
23    else:
24        raise ValueError ('El sistema no converge')
25
26 def main():
27     A = np.array([[-512,12,0,0,0], \
28                  [500,-512,12,0,0], \
29                  [0,500,-512,12,0], \
30                  [0,0,500,-512,12], \
31                  [0,0,0,-500,512.0]])
32     b = np.array([0,0,0,0,9.0])
33     x = np.array([0.0,0.0,0.0,0.0,0.0])
34     w=1.1
35     x=sor(A,b,x,w)
36     print('Solucion: ',x)
37
38 if __name__ == "__main__": main()

```

```

iteracion:0->[-0.          -0.          -0.          -0.          ←
  0.01933594] norma 0.0193359375
iteracion:1->[ 0.          0.          0.          0.0004985   ←
  0.01793785] norma 0.0014843060791567123
iteracion:2->[ -0.00000000e+00 -0.00000000e+00  1.28520727e-05 ←
  4.26415586e-04
  1.80002164e-02] norma 9.618722250697878e-05
iteracion:3->[  0.00000000e+00   3.31342500e-07   1.00642539e-05 ←
  4.32237732e-04
  1.80002337e-02] norma 6.46369882444121e-06
iteracion:4->[  8.54242382e-09   2.35511227e-07   1.03901942e-05 ←
  4.32006094e-04

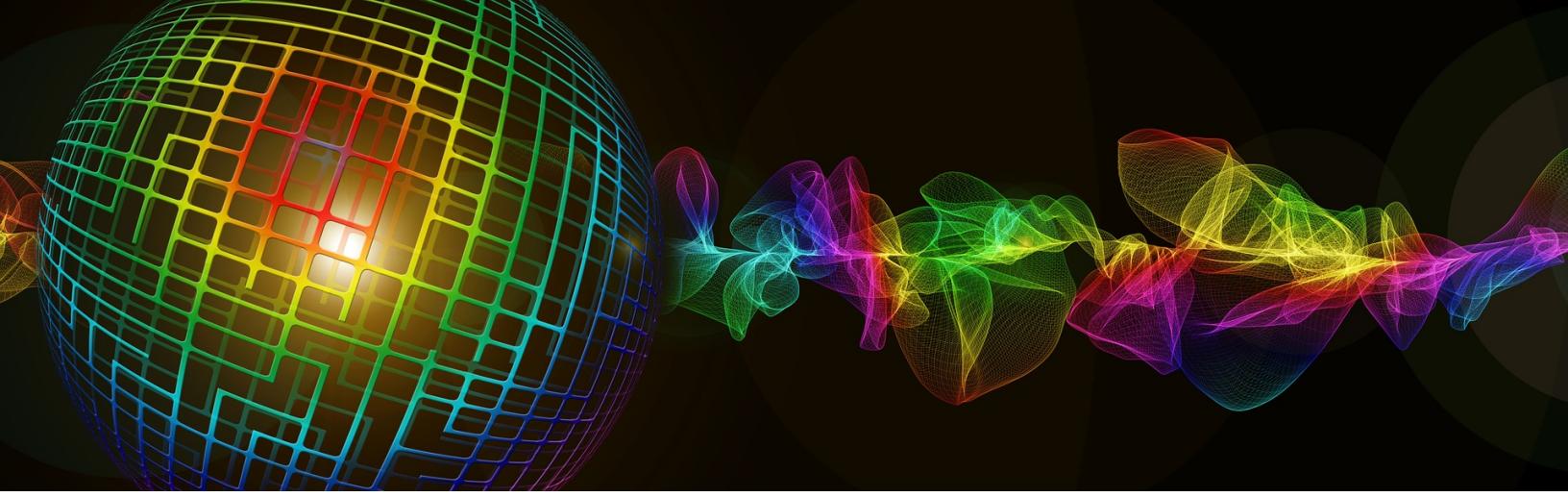
```

```
1.79999832e-02] norma 4.815903680254931e-07
iteracion:5->[ 5.21753144e-09    2.49925842e-07    1.03671127e-05 ←
    4.31998004e-04
1.79999995e-02] norma 3.2937161890614256e-08
iteracion:6->[ 5.92164747e-09    2.48645685e-07    1.03678371e-05 ←
    4.32000013e-04
1.80000001e-02] norma 2.6397189268971016e-09
Solucion: [ 5.92164747e-09    2.48645685e-07    1.03678371e-05 ←
    4.32000013e-04
    1.80000001e-02]
```

El conjunto solución es

$$\begin{aligned}Plato_1 &= 1.80000001e-02 \\Plato_2 &= 4.32000013e-04 \\Plato_3 &= 1.03678371e-05 \\Plato_4 &= 2.48645685e-07 \\Plato_5 &= 5.92164747e-09\end{aligned}$$

■



5. Sistemas de ecuaciones no lineales

Una ecuación no lineal univariable se expresa como $f(x)$, cuando hablamos de funciones no lineales multivariadas las podemos expresar como $f(x_1, x_2, x_3, \dots, x_n)$. Para resolver este tipo de funciones que tienen n variables, se requiere de un sistema de n ecuaciones que se puede expresar como:

$$F(x_1, x_2, x_3, \dots, x_n) = \begin{pmatrix} f_1(x_1, x_2, x_3, \dots, x_n) \\ f_2(x_1, x_2, x_3, \dots, x_n) \\ \vdots \\ f_n(x_1, x_2, x_3, \dots, x_n) \end{pmatrix}$$

Donde $F(x_1, x_2, x_3, \dots, x_n)$ es una función vectorial no lineal, el conjunto solución que resuelve el sistema satisface a todas las ecuaciones $f_i(x_1, x_2, x_3, \dots, x_n)$ simultáneamente.

5.1 Punto fijo multivariable

Se despeja x_i de la i -ésima ecuación y se genera un sistema iterativo

Ecuación iterativa
de punto fijo multivariable

$$\begin{pmatrix} x_1^{(k+1)} = g_1(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}) \\ x_2^{(k+1)} = g_2(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}) \\ \vdots \\ x_n^{(k+1)} = g_n(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}) \end{pmatrix} \quad (5.1)$$

El vector $x^{(k)}$ es un vector inicial que se sustituye en el sistema de la Ecuación 5.1 y se obtiene un nuevo vector $x^{(k+1)}$, el proceso se repite hasta que se cumple el criterio de convergencia.

■ **Ejemplo 5.1 — Método de punto fijo multivariable.** Resuelva el siguiente sistema de ecuaciones no lineales, utilizando el método de punto fijo multivariable:

$$\begin{aligned}3x_1 - \cos(x_2 x_3) - \frac{1}{2} &= 0 \\x_1^2 - 81(x_2 + 0.2)^2 + \sin(x_3) + 1.06 &= 0 \\e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3} &= 0\end{aligned}$$

Solución

Primero se despeja x_i de la i-ésima ecuación de la siguiente forma

$$\begin{aligned}x_1 &= \frac{1}{3}\cos(x_2 x_3) + \frac{1}{6} \\x_2 &= \frac{1}{9}\sqrt{x_1^2 + \sin(x_3) + 1.06} - 0.1 \\x_3 &= -\frac{1}{20}e^{-x_1 x_2} + \frac{10\pi - 3}{60}\end{aligned}$$

Programa 5.1. Método del punto fijo multivariable

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def pfm(g, x, imax=100, tol=1e-8):
5     cumple=False
6     k=0
7
8     while (not cumple and k<imax):
9         x0 = x.copy()
10        x=g(x0)
11        norma=np.linalg.norm(x-x0)
12        print('iteracion:{}->{} norma {}'.format(k,x,norma))
13        cumple=norma<tol
14        k+=1
15
16    if k<imax:
17        return x
18    else:
19        raise ValueError ('El sistema no converge')
20
21 def g(x):
22     return np.array([1/3*np.cos(x[1]*x[2])+1/6,\n                     1/9*np.sqrt(x[0]**2+np.sin(x[2])+1.06)-0.1,\n                     -1/20*np.exp(-x[0]*x[1])-(10*np.pi-3)/60])
23
24 def main():
25

```

```

27     x=np.array([0.1,0.1,-0.1])
28     x=pfm(g,x)
29     print('Solucion: ',x)
30
31 if __name__ == "__main__": main()

```

```

iteracion:0->[ 0.49998333  0.00944115 -0.52310127] norma ←
  0.5892387078421285
iteracion:1->[ 4.99995935e-01   2.55677468e-05  -5.23363311e-01] ←
  norma 0.009419236028559366
iteracion:2->[ 5.00000000e-01   1.23367204e-05  -5.23598136e-01] ←
  norma 0.00023523308131895355
iteracion:3->[ 5.00000000e-01   3.41679063e-08  -5.23598467e-01] ←
  norma 1.23069981712114e-05
iteracion:4->[ 5.00000000e-01   1.64870404e-08  -5.23598775e-01] ←
  norma 3.0807065103297765e-07
iteracion:5->[ 5.00000000e-01   4.56639865e-11  -5.23598775e-01] ←
  norma 1.644731715170715e-08
iteracion:6->[ 5.00000000e-01   2.20342355e-11  -5.23598776e-01] ←
  norma 4.1171297702499117e-10
Solucion: [ 5.00000000e-01   2.20342355e-11  -5.23598776e-01]

```

El conjunto solución es:

$$\begin{aligned} x_1 &= 5.00000000e-01 \\ x_2 &= 2.20342355e-11 \\ x_3 &= -5.23598776e-01 \end{aligned}$$

■

5.2 Gauss-Seidel multivariable

En el método de Gauss-Seidel multivariable se sigue el mismo procedimiento que en el método de punto fijo multivariable, sólo que las sustituciones son sucesivas en lugar de simultáneas. El algoritmo es el mismo.

Se despeja x_i de la i -ésima ecuación y se genera un sistema iterativo

Ecuación iterativa de Gauss-Seidel multivariable

$$\left(\begin{array}{l} x_1^{(k+1)} = g_1(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}) \\ x_2^{(k+1)} = g_2(x_1^{(k+1)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}) \\ x_3^{(k+1)} = g_2(x_1^{(k+1)}, x_2^{(k+1)}, x_3^{(k)}, \dots, x_n^{(k)}) \\ \vdots \\ x_n^{(k+1)} = g_n(x_1^{(k+1)}, x_2^{(k+1)}, x_3^{(k+1)}, \dots, x_n^{(k)}) \end{array} \right) \quad (5.2)$$

El vector $x^{(k)}$ es un vector inicial que se sustituye en el sistema de la ecuación 5.2 para obtener $x_1^{(k+1)}$, en el cálculo de $x_2^{(k+1)}$ se utiliza el valor recién obtenido de $x_1^{(k+1)}$ haciendo la sustitución sucesiva. El proceso se repite hasta cumplir con el criterio de convergencia.

■ **Ejemplo 5.2 — Método de Gauss-Seidel multivariable.** Resuelva el siguiente sistema de ecuaciones no lineales, utilizando el método de Gauss-Seidel multivariable

$$\begin{aligned} 5x_1^2 - x_2^2 &= 0 \\ x_2 - \frac{\sin(x_1) + \cos(x_2)}{4} &= 0 \end{aligned}$$

Solución

Primero se despeja x_i de la i-ésima ecuación de la siguiente forma

$$\begin{aligned} x_1 &= \sqrt{\frac{x_2^2}{5}} \\ x_2 &= \frac{\sin(x_1) + \cos(x_2)}{4} \end{aligned}$$

Programa 5.2. Método de Gauss-Seidel multivariable

```

1 import numpy as np
2
3 def gsm(g,x,imax=200,tol=1e-8):
4     cumple=False
5     k=0
6
7     while (not cumple and k<imax):
8         x0 = x.copy()
9         x=g(x0)
10        norma=np.linalg.norm(x-x0)
11        print('iteracion:{}->{} norma {}'.format(k,x,norma))
12        cumple=norma<tol
13        k+=1
14
15    if k<imax:
16        return x
17    else:
18        raise ValueError ('El sistema no converge')
19
20 def g(x):
21     xk1=x.copy()
22     xk1[0]=np.sqrt(x[1]**2/5)
23     xk1[1]=0.25*(np.sin(x[0])+np.cos(x[1]))
24     return xk1
25
26 def main():

```

```

27     x=np.array([0.5,0.5])
28     x=gsm(g,x)
29     print('Solucion: ',x)
30
31 if __name__ == "__main__": main()

```

```

iteracion:0->[ 0.2236068   0.33925203] norma 0.3197391337901425
iteracion:1->[ 0.15171812  0.29118798] norma 0.08647621141501442
iteracion:2->[ 0.13022322  0.27726006] norma 0.025612837861277812
iteracion:3->[ 0.12399447  0.27291613] norma 0.007593886940914559
iteracion:4->[ 0.1220518   0.27166649] norma 0.002309878578272204
iteracion:5->[ 0.12149295  0.27126851] norma 0.0006860784327828056
iteracion:6->[ 0.12131497  0.27115651] norma ←
    0.00021028814381171275
iteracion:7->[ 0.12126488  0.27111985] norma 6.207440350969236e-05
iteracion:8->[ 0.12124848  0.27110987] norma 1.9193347199193587e←
    -05
iteracion:9->[ 0.12124402  0.27110647] norma 5.609811430065538e-06
iteracion:10->[ 0.1212425   0.27110559] norma 1.7570508539963682e←
    -06
iteracion:11->[ 0.12124211  0.27110527] norma 5.06130703268258e-07
iteracion:12->[ 0.12124196  0.2711052 ] norma 1.6161985774738058e←
    -07
iteracion:13->[ 0.12124193  0.27110517] norma 4.559102004416909e←
    -08
iteracion:14->[ 0.12124192  0.27110516] norma 1.4984950396376303e←
    -08
iteracion:15->[ 0.12124191  0.27110516] norma 4.10560915978155e-09
Solucion: [ 0.12124191  0.27110516]

```

El conjunto solución es

$$\begin{aligned} x_1 &= 0.12124191 \\ x_2 &= 0.27110516 \end{aligned}$$

■

5.3 Newton-Raphson multivariable

El algoritmo del método de Newton-Raphson que se usó para una ecuación no lineal $f(x) = 0$ se aplica también a un sistema de ecuaciones no lineales.

$$F(x_1, x_2, x_3, \dots, x_n) = \begin{pmatrix} f_1(x_1, x_2, x_3, \dots, x_n) \\ f_2(x_1, x_2, x_3, \dots, x_n) \\ \vdots \\ f_n(x_1, x_2, x_3, \dots, x_n) \end{pmatrix}$$

Extendemos la serie de Taylor multivariable al sistema de ecuaciones:

$$\begin{aligned} f_1(x_1^*, x_2^*, x_3^*, \dots, x_n^*) &= f_1(x_1, x_2, x_3, \dots, x_n) + \frac{\partial f_1}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_1}{\partial x_2}(x_2^* - x_2) + \dots + \frac{\partial f_1}{\partial x_n}(x_n^* - x_n) \\ f_2(x_1, x_2, x_3, \dots, x_n) &= f_2(x_1, x_2, x_3, \dots, x_n) + \frac{\partial f_2}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_2}{\partial x_2}(x_2^* - x_2) + \dots + \frac{\partial f_2}{\partial x_n}(x_n^* - x_n) \\ &\vdots \\ f_n(x_1, x_2, x_3, \dots, x_n) &= f_n(x_n, x_2, x_3, \dots, x_n) + \frac{\partial f_n}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_n}{\partial x_2}(x_2^* - x_2) + \dots + \frac{\partial f_n}{\partial x_n}(x_n^* - x_n) \end{aligned}$$

las x_i^* son los nuevos valores de x_i , las derivadas parciales $\frac{\partial f_i}{\partial x_1}$ se evalúan en los valores iniciales x_1, x_2, \dots, x_n . Igualando cada ecuación a cero y expresándola en forma matricial tenemos:

$$0 = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \frac{\partial f_n}{\partial x_3} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \begin{pmatrix} x_1^* - x_1 \\ x_2^* - x_2 \\ x_3^* - x_3 \\ \vdots \\ x_n^* - x_n \end{pmatrix}$$

Donde el vector de las funciones se representa por $F(x)$, la matriz de derivadas parciales de f_i es la matriz Jacobiana $J(x)$ y se requiere determinar los valores de x_i^* , por lo tanto, la ecuación se puede expresar como:

$$F(x^{(k)}) + J(x^{(k)})(x^{(k+1)} - x^{(k)}) = 0$$

Si expresamos el término $(x^{(k+1)} - x^{(k)})$ como Δx , la cual es una incógnita para determinar, la expresión es:

$$F(x^{(k)}) + J(x^{(k)})\Delta x = 0$$

El vector $F(x^{(k)})$, después de evaluarse, es un vector de números; la matriz $J(x^{(k)})$, después de evaluarse, es una matriz de números y Δx es una incógnita, expresemos la ecuación de esta forma:

$$J(x^{(k)})\Delta x = -F(x^{(k)})$$

Y esto tiene la forma de un sistema de ecuaciones lineales que ya sabemos resolver, la solución del sistema es el valor de Δx , entonces para obtener los valores de x_i^* basta hacer un despeje sencillo.

$$\begin{aligned} x_i^{(k+1)} - x_i^{(k)} &= \Delta x_i \\ x_i^{(k+1)} &= \Delta x_i + x_i^{(k)} \end{aligned}$$

Hay que validar si esta aproximación al conjunto solución cumple con el criterio de convergencia, si no es así, se sustituyen como nuevos valores y se vuelve a evaluar $-F(x^{(k)})$ y $J(x^{(k)})$ para obtener un nuevo valor de Δx , los cálculos se repiten hasta cumplir con el criterio de convergencia.

■ **Ejemplo 5.3 — Método de Newton-Raphson multivariable.** Resuelva el siguiente sistema de ecuaciones no lineales, utilizando el método de Newton-Raphson multivariable

$$\begin{aligned}f_1(x_1, x_2) &= 2x_1 - x_2 - e^{-x_1} = 0 \\f_2(x_1, x_2) &= -x_1 + 2x_2 - e^{-x_2} = 0\end{aligned}$$

Solución

Obtenemos el Jacobiano del sistema:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} = 2 + e^{-x_1} & \frac{\partial f_1}{\partial x_2} = -1 \\ \frac{\partial f_2}{\partial x_1} = -1 & \frac{\partial f_2}{\partial x_2} = 2 + e^{-x_2} \end{pmatrix}$$

Programa 5.3. Método de Newton-Raphson multivariable

```

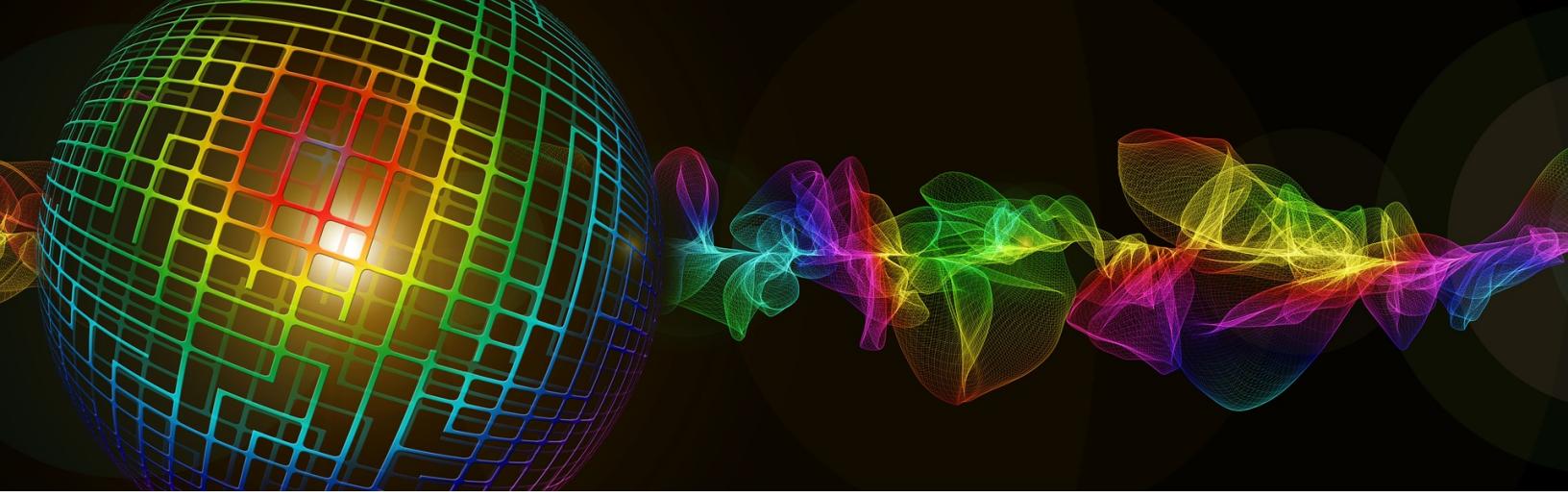
1 import numpy as np
2
3 # Metodo de Newton-Raphson Multivariable
4 def newtonRaphsonMV(F, J, x, imax=100, tol=1e-8):
5     cumple=False
6     k=0
7     while (not cumple and k<imax):
8         deltax=np.linalg.solve(J(x), -F(x))
9         x = x + deltax
10        print('iteracion:{}-> {}'.format(k,x))
11        cumple=np.linalg.norm(F(x))<=tol
12        k+=1
13    if k<imax:
14        return x
15    else:
16        raise ValueError ('La funcion no converge')
17
18 # Vector de Funciones
19 def f(x):
20     return np.array([2*x[0] - x[1] - np.exp(-x[0]), \
21                     -x[0] + 2*x[1] - np.exp(-x[1])])
22 # Matriz Jacobiana
23 def j(x):
24     return np.array([[2 + np.exp(-x[0]), -1], \
25                     [-1, 2 + np.exp(-x[1])]])
26
27 def main():
28     # valores iniciales
29     x0=np.array([-5., -5.])
30     # Llamada al algoritmo
31     raiz=newtonRaphsonMV(f, j, x0)
32     print('f({})={}'.format(raiz,f(raiz)))
33
34 if __name__ == "__main__": main()
```

```
iteracion:0-> [-3.9732286 -3.9732286]
iteracion:1-> [-2.91832728 -2.91832728]
iteracion:2-> [-1.82000345 -1.82000345]
iteracion:3-> [-0.70566754 -0.70566754]
iteracion:4-> [ 0.19703885  0.19703885]
iteracion:5-> [ 0.53974367  0.53974367]
iteracion:6-> [ 0.56700632  0.56700632]
iteracion:7-> [ 0.56714329  0.56714329]
f([ 0.56714329  0.56714329])=[ -5.32062894e-09   -5.32062894e-09]
```

El conjunto solución es

$$\begin{aligned}x_1 &= 0.56714329 \\x_2 &= 0.56714329\end{aligned}$$

■



6. Diferenciación numérica

6.1 Diferenciación

6.1.1 Repaso de la serie de Taylor

Si expandimos f alrededor de x_i y f es $n+1$ veces diferenciable en un intervalo abierto que contenga x_i , el teorema de la serie de Taylor y el término del residuo dice que si x_{i+1} es otro punto en este intervalo entonces

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \dots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n$$

Donde $R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$

ξ es un número en el intervalo abierto entre x_i y x_{i+1}

$h = x_{i+1} - x_i$

$f(x_{i+1}) = f(x_i + h)$

6.1.2 Teorema del valor medio

La aparición de ξ , un punto entre x_i y x_{i+1} , sugiere una conexión entre este resultado y el teorema del valor medio, el cual establece que un arco entre dos puntos, existe al menos un punto en el cual la tangente del arco es paralelo a la secante entre ambos puntos.

Si la función f es continua en el intervalo $[x_i, x_{i+1}]$ y diferenciable en (x_i, x_{i+1}) , entonces existe un punto ξ tal que

$$f'(\xi) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

La serie de Taylor de orden cero es

$$f(x_{i+1}) \approx f(x_i) + R_0$$

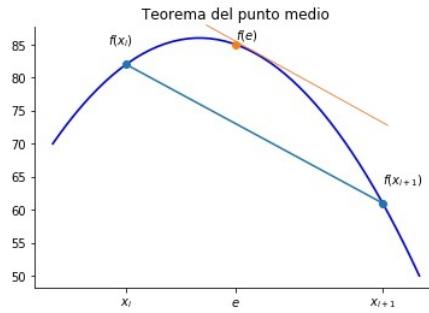


Figura 6.1. Teorema del valor medio

Donde $R_0 = f'(\xi)h$ y $h = x_{i+1} - x_i$ entonces

$$f(x_{i+1}) = f(x_i) + f'(\xi)(x_{i+1} - x_i)$$

Donde ξ está entre x_i y x_{i+1} . Éste es el teorema del punto medio, el cual se usa para probar el teorema de Taylor.

6.1.3 Diferencias hacia adelante

Si truncamos la serie de Taylor de primer orden tenemos

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + R_1$$

donde

$$R_1 = \frac{f''(\xi)}{2}h^2$$

$$\frac{R_1}{h} = \frac{f''(\xi)}{2}h$$

Rearreglando

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{R_1}{h}$$

Diferencia hacia adelante

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \quad (6.1)$$

6.1.4 Diferencias hacia atrás

La serie de Taylor se puede expresar hacia atrás para calcular el valor previo basado en el valor actual.

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \frac{f'''(x_i)}{3!}h^3 + \dots$$

Truncando la serie de primer orden

$$f(x_{i-1}) \approx f(x_i) - f'(x_i)h + R_1$$

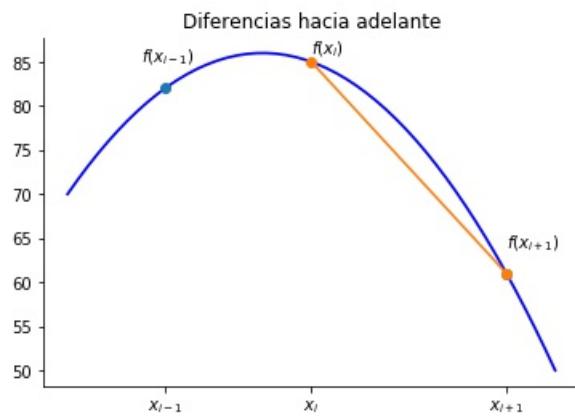


Figura 6.2. Diferencia hacia adelante

Rearreglando

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + O(h)$$

Diferencia hacia atrás

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + O(h) \quad (6.2)$$

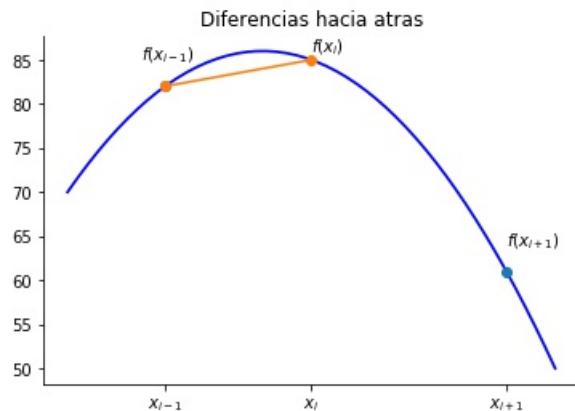


Figura 6.3. Diferencia hacia atrás

6.1.5 Diferencias centradas

La tercera forma para aproximar la primera derivada es restando la diferencia hacia adelante y la diferencia hacia atrás.

$$\begin{aligned} f(x_{i+1}) &= f(x_i) + f'(\xi)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \dots \\ f(x_{i-1}) &= f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \frac{f'''(x_i)}{3!}h^3 + \dots \\ f(x_{i+1}) - f(x_{i-1}) &= 2f'(x_i)h + \frac{2f'''(x_i)}{3!}h^3 + \dots \end{aligned} \quad (6.3)$$

Rearreglando

Diferencia hacia atrás

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + O(h^2) \quad (6.4)$$

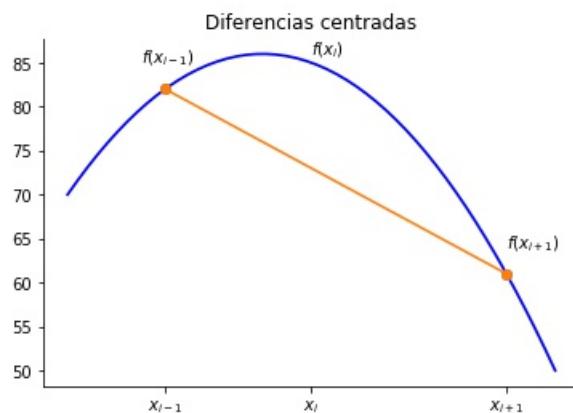


Figura 6.4. Diferencias centradas

Las diferencias centradas son más precisas porque tienen un error de $O(h^2)$

6.1.6 Diferencias de 2º orden

Para aproximar la derivada de segundo orden, escribiremos la expansión de la serie de Taylor hacia adelante para $f(x_{i+2})$ en términos de $f(x_i)$

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2h) + \frac{f''(x_i)}{2!}(2h)^2 + \frac{f'''(x_i)}{3!}(2h)^3 + \dots$$

Truncamos la serie de segundo orden

$$f(x_{i+2}) \approx f(x_i) + 2f'(x_i)h + 2f''(x_i)h^2 + R_2$$

Truncando la diferencia hacia adelante de segundo orden y multiplicando por 2

$$2f(x_{i+1}) \approx 2f(x_i) + 2f'(x_i)h + f''(x_i)h^2 + 2R_2$$

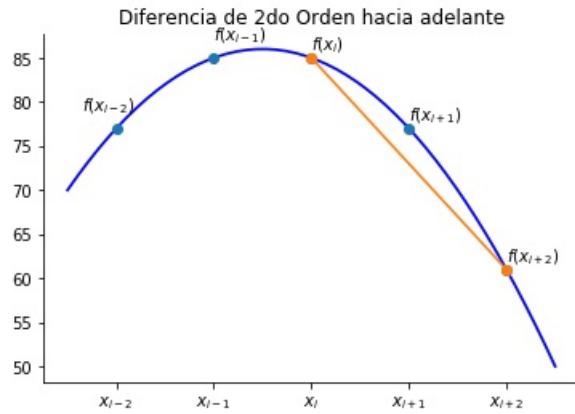


Figura 6.5. Diferencias de 2º orden hacia adelante

Restando de la ecuación anterior tenemos

$$f(x_{i+2}) - 2f(x_{i+1}) \approx -f(x_i) + f''(x_i)h^2 - R_2$$

Rearreglando

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + \frac{R_2}{h^2}$$

El residuo se puede escribir como

$$R_2 = \frac{f'''(\xi)}{3!}h^3$$

$$\frac{R_2}{h^2} = \frac{f'''(\xi)}{3!}h = O(h)$$

Por lo tanto

Diferencia de 2º orden hacia adelante

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h) \quad (6.5)$$

Seguimos el mismo procedimiento para la derivada de 2º orden para la diferencia hacia atrás

Diferencia de 2º orden hacia atrás

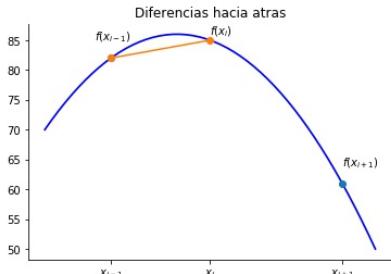
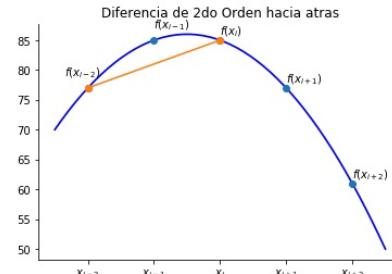
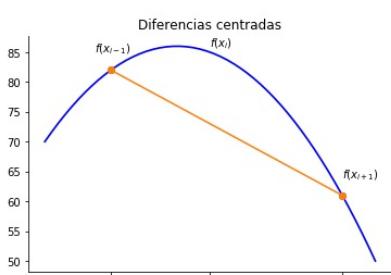
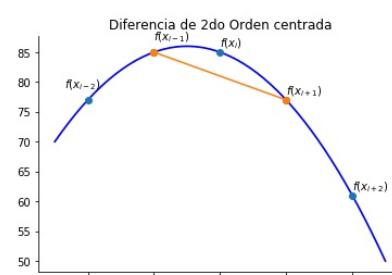
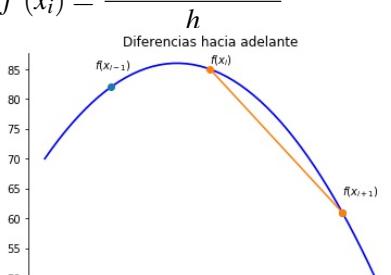
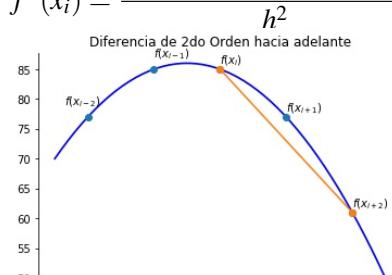
$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2} + O(h) \quad (6.6)$$

y centrada

Diferencia de 2º orden centrada

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + O(h) \quad (6.7)$$

Cuadro 6.1. Diferenciación

Tipo	Primer orden	Segundo orden
Hacia atrás	$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$	$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2}$
		
Centrada	$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$	$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2}$
		
Hacia adelante	$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$	$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2}$
		

■ **Ejemplo 6.1 — Gradiente de presión en una cama fluidizada.** A continuación, se muestran las medidas de presión de una cama fluidizada.

Posición z (m)	0	0.5	1.0	1.5	2.0
Presión P (kPa)	1.82	1.48	1.19	0.61	0.15

Estimar los valores del gradiente de presión $\frac{dP}{dz}$ en las diferentes posiciones

Solución

Programa 6.1. Diferenciación numérica

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def differ(x,y, tipo=0, orden=1):
5     # tipo = (-1:Diferencia hacia atras, 0:Centrada, 1:Diferencia ←
6         # hacia adelante )
7     # orden = (1:Primer orden, 2:Segundo orden)
8
9     n=x.size
10    if tipo== -1: #Diferencias hacia atras
11        if orden==1:
12            return [(y[i]-y[i-1])/(x[i]-x[i-1]) \
13                      for i in range(1,n)]
14        elif orden==2:
15            return [(y[i]-2*y[i-1]+y[i-2])/((x[i]-x[i-1])**2) \
16                      for i in range(2,n)]
17        else:
18            raise ValueError ('Parametro <Orden> incorrecto')
19    elif tipo== 0: #Diferencias centradas
20        if orden==1:
21            return [(y[i+1]-y[i-1])/((x[i]-x[i-1])/2) \
22                      for i in range(1,n-1)]
23        elif orden==2:
24            return [(y[i+1]-2*y[i]+y[i-1])/((x[i]-x[i-1])**2) \
25                      for i in range(1,n-1)]
26        else:
27            raise ValueError ('Parametro <Orden> incorrecto')
28    elif tipo== 1: #Diferencias hacia adelante
29        if orden==1:
30            return [(y[i+1]-y[i])/(x[i+1]-x[i]) \
31                      for i in range(0,n-1)]
32        elif orden==2:
33            return [(y[i+2]-2*y[i+1]+y[i])/((x[i]-x[i-1])**2) \
34                      for i in range(0,n-2)]
35        else:
36            raise ValueError ('Parametro <Orden> incorrecto')
37    else:
38        raise ValueError ('Parametro <Tipo> incorrecto')

```

```

39 def main():
40     # valores de posicion
41     x=np.array([0,0.5,1.0,1.5,2.0])
42     # valores de presion
43     y=np.array([1.82,1.48,1.19,0.61,0.15])
44     # Llamada al algoritmo
45     d=difer(x,y, tipo=0, orden=1)
46     print('gradiente {}; posicion {}'.format(d,x[1:4]))
47
48     fig = plt.figure()
49     plt.plot(x[1:4],d,'o')
50     plt.title('Gradiente de la Presion')
51     plt.xlabel('Posicion z')
52     plt.ylabel('dP/dz')
53     plt.grid()
54
55     plt.show()
56     fig.savefig("difer.pdf", bbox_inches='tight')
57
58 if __name__ == "__main__": main()

```

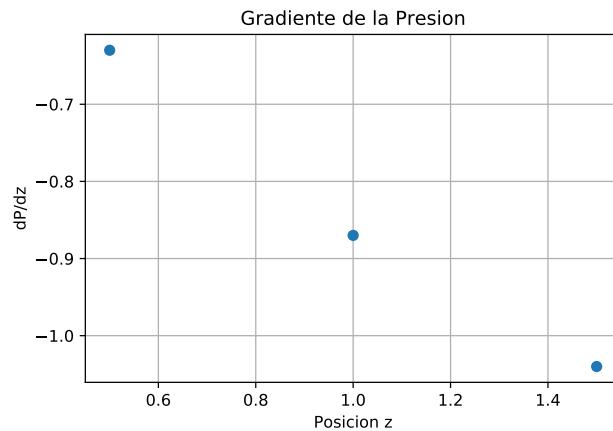
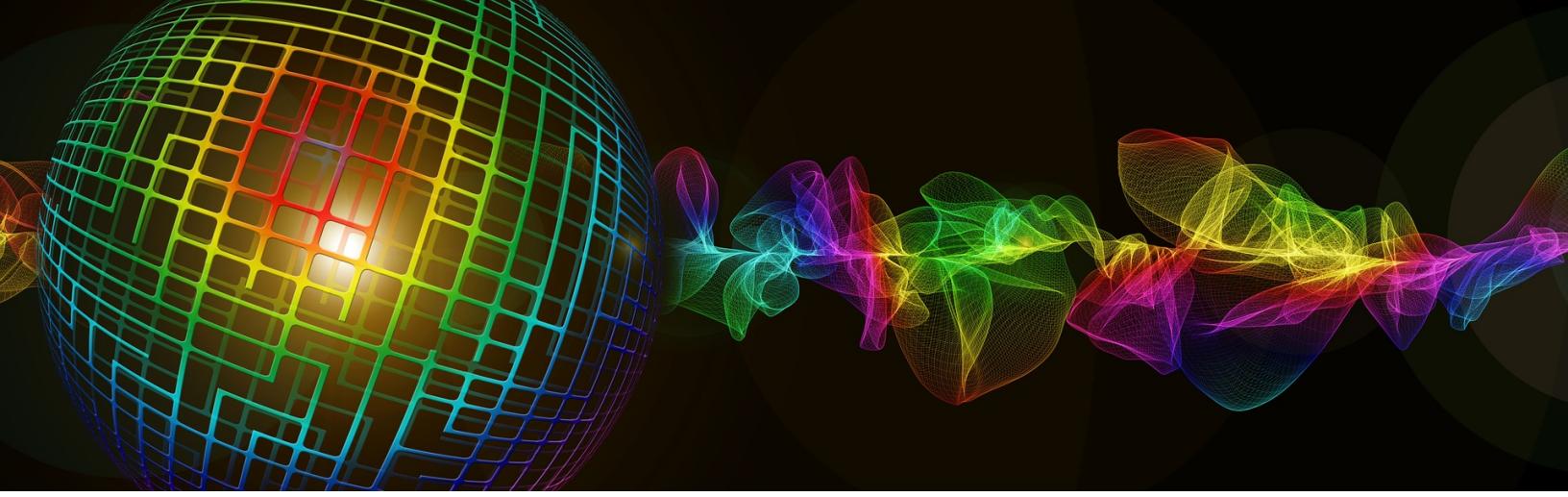


Figura 6.6. Diferencias centradas

```

gradiente [-0.63000000000000012, -0.87, -1.04]; posicion [ 0.5 1.5]

```



7. Integrales

El objetivo de los métodos de integración es calcular el área que completen los límites de integración a y b que son los lados izquierdo y derecho, el eje x el lado inferior y la función $f(x)$ el lado superior. El lado superior tiene la complejidad porque tiene un comportamiento distinto del lineal, en la mayoría de los casos; por lo tanto, no es posible obtener la integral simplemente con el área del polígono que se forma.

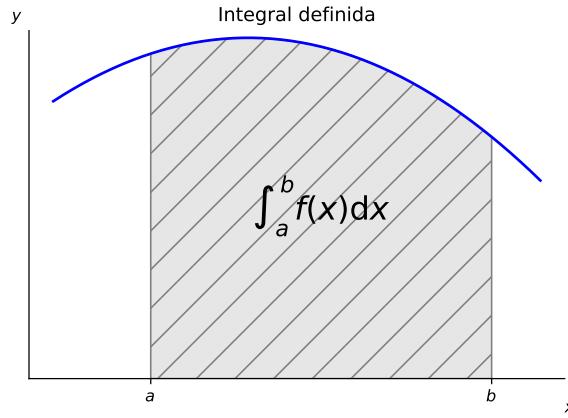


Figura 7.1. Integral definida

7.1 Fórmulas de Newton-Cotes

Las fórmulas de Newton-Cotes simplifican la integral definida de $f(x)$ en la integral de un polinomio $p(x)$ de grado n el cual es más fácil de resolver, lo cual genera un resultado aproximado.

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)dx$$

El uso de polinomios de diferente grado genera una familia de fórmulas llamadas de Newton-Cotes.

7.1.1 Regla de los rectángulos

La regla de los rectángulos es la aproximación de la integral definida de $f(x)$ con la integral de un polinomio de grado 0.

$$\int_a^b f(x)dx \approx \int_a^b p_0(x)dx$$

La integral del rectángulo se obtiene multiplicando la base $(b - a)$ por la altura $f(a)$.

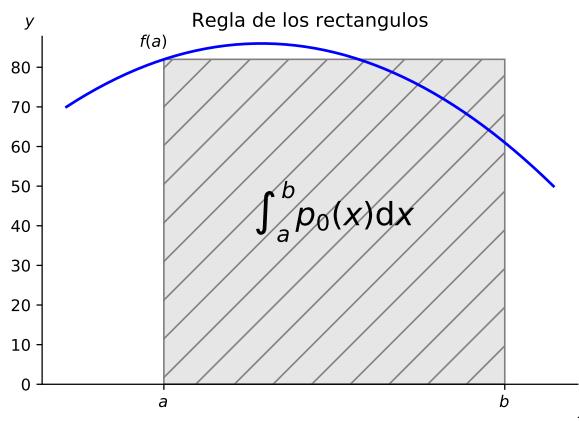


Figura 7.2. Regla de los rectángulos

$$\int_a^b p_0(x)dx = (b - a)f(a)$$

En algunos casos, el área del rectángulo será menor que el valor de la integral y, en otros casos, será menor. Para mejorar el resultado dividimos el intervalo de integración $[a, b]$ en n subintervalos, de tal manera que la integral cubre mejor el área bajo la curva. El valor de la integral será la suma del área de cada rectángulo.

$$x_0 = a; \text{ límite inferior}$$

$$x_n = b; \text{ límite superior}$$

$$h = \frac{b - a}{n}; n \text{ es el número de subintervalos}$$

$$x_0 = a, x_1 = x_0 + h, x_2 = x_1 + h, \dots, x_n = x_{n-1} + h = b$$

$$I = hf(x_0) + hf(x_1) + hf(x_2) + \dots + hf(x_{n-1})$$

$$I = h \sum_{i=0}^{n-1} f(x_i)$$

Rectángulos

$$I = h \sum_{i=0}^{n-1} f(x_i) \quad (7.1)$$

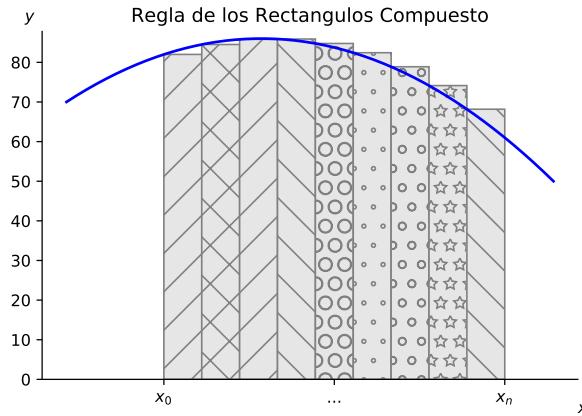


Figura 7.3. Regla de los rectángulos compuestos

■ **Ejemplo 7.1 — Método de los rectángulos.** Evalúe la integral de $f(x)$ de 0 a 1, con un rectángulo y con 10 rectángulos.

$$\int_0^1 e^{x^2} dx$$

Solución

Programa 7.1. Método de los rectángulos

```

1 import numpy as np
2
3 import matplotlib.pyplot as plt
4 from matplotlib.patches import Polygon
5
6 def regla_rectangulos(f,a,b,n):
7     h=(b-a)/n
8     xs=np.linspace(a,b,n+1)
9     ys=f(xs)
10    r=h*sum(ys[:n])
11    return r
12
13 #funcion a integrar
14 def f(x):
15     return np.exp(x**2)

```

```

16
17 def grafica_rectangulos(f,a,b,n):
18     x = np.linspace(a, b)
19     y = f(x)
20     fig, ax = plt.subplots()
21     ax.plot(x, y, 'b', linewidth=1.7)
22     ax.set_ylim(bottom=0)
23
24     ix = np.linspace(a, b,n+1)
25     iy = f(ix)
26     patterns=( '/','x','/','\\','0','.','o','*','\\','/','-','x','+←'
27     ')
28     for i in range(n):
29         verts = [(ix[i], 0), (ix[i],iy[i]),(ix[i+1],iy[i]), (ix[i←
29         +1], 0)]
30         poly = Polygon(verts, facecolor='0.9', edgecolor='0.5',←
31             hatch=patterns[i])
32         ax.add_patch(poly)
33
34     plt.title('Regla de los rectangulos $e^{x^2}$')
35     fig.savefig("integral_ex2_10.pdf", bbox_inches='tight')
36     return plt
37
38
39 def main():
40     a=0 # limite inferior
41     b=1 # limite superior
42     n=10 # numero de rectangulos
43     area=regla_rectangulos(f,a,b,n)
44     print('integral = ',area)
45     g=grafica_rectangulos(f,a,b,n)
46     g.show()
47
48 if __name__ == "__main__": main()

```

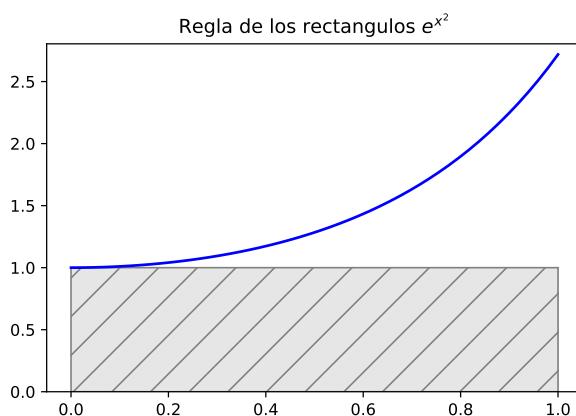


Figura 7.4. Regla de los rectángulos con 1 rectángulo

```
integral = 1.0
```

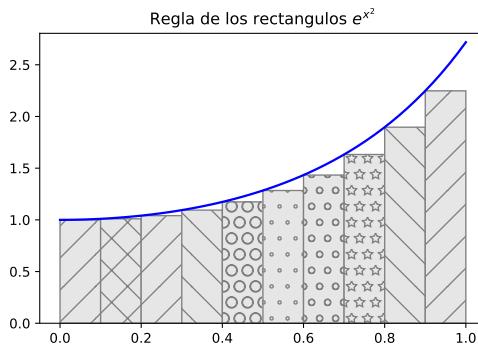


Figura 7.5. Regla de los rectángulos con 10 rectángulos

```
integral = 1.38126060132
```

■

7.1.2 Regla de los trapecios

La regla de los trapecios es la aproximación de la integral definida de $f(x)$ con la integral de un polinomio de grado 1.

$$\int_a^b f(x)dx \approx \int_a^b p_1(x)dx$$

La integral del trapecio se obtiene como:

$$\text{área} = (b-a) \left(\frac{f(a)+f(b)}{2} \right)$$

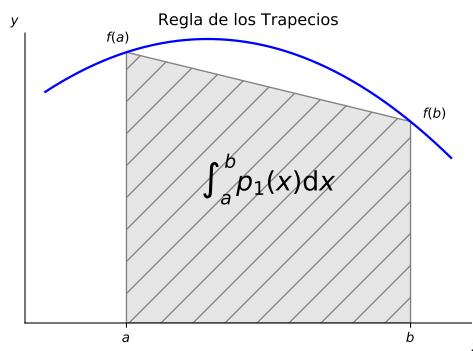


Figura 7.6. Regla de los trapecios

Regla de los trapecios

$$\int_a^b p_1(x)dx = (b-a) \left(\frac{f(a)+f(b)}{2} \right) \quad (7.2)$$

En algunos casos el área del trapecio será menor que el valor de la integral y en otros casos será mayor. Para mejorar el resultado dividimos el intervalo de integración $[a, b]$ en n subintervalos, de tal manera que la integral cubre mejor el área bajo la curva. El valor de la integral será la suma del área de cada trapecio.

$$x_0 = a; \text{límite inferior}$$

$$x_n = b; \text{límite superior}$$

$$h = \frac{b-a}{n}; n \text{ es el número de subintervalos}$$

$$x_0 = a, x_1 = x_0 + h, x_2 = x_1 + h, \dots, x_n = x_{n-1} + h = b$$

$$I = h \left(\frac{f(x_0) + f(x_1)}{2} \right) + h \left(\frac{f(x_1) + f(x_2)}{2} \right) + h \left(\frac{f(x_2) + f(x_3)}{2} \right) + \dots + h \left(\frac{f(x_{n-1}) + f(x_n)}{2} \right)$$

$$I = \frac{h}{2} \left(f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right)$$

Regla de los trapecios compuesta

$$I = \frac{h}{2} \left(f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right) \quad (7.3)$$

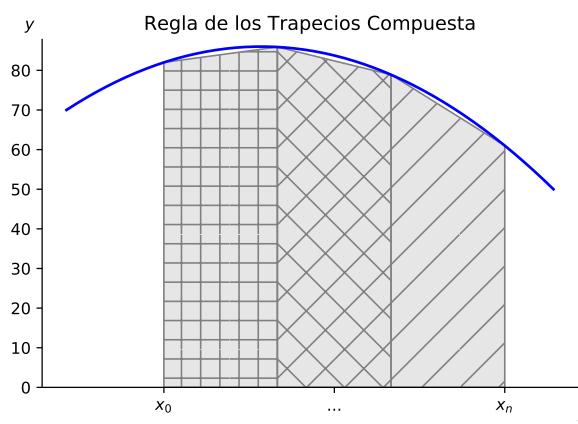


Figura 7.7. Regla de los trapecios compuesta

■ **Ejemplo 7.2 — Método de los trapecios.** Evalúe la integral de $f(x)$ de 0 a 1, con un trapecio y con 10 trapecios.

$$\int_0^1 e^{x^2} dx$$

Solución

Programa 7.2. Método de los trapecios

```

1 import numpy as np
2
3 import matplotlib.pyplot as plt
4 from matplotlib.patches import Polygon
5
6 def regla_trapezios(f,a,b,n):
7     h=(b-a)/n
8     xs=np.linspace(a,b,n+1)
9     ys=f(xs)
10    r=h*(ys[0]+2*sum(ys[1:n])+ys[n])/2
11    return r
12
13 #funcion a integrar
14 def f(x):
15     return np.exp(x**2)
16
17 def grafica_trapezios(f,a,b,n):
18     x = np.linspace(a, b)
19     y = f(x)
20     fig, ax = plt.subplots()
21     ax.plot(x, y, 'b', linewidth=1.7)
22     ax.set_ylim(bottom=0)
23
24     ix = np.linspace(a, b,n+1)
25     iy = f(ix)
26     patterns=( '|', '/','\\','0','.','o','*', '\\','/','-','x','+' )
27     for i in range(n):
28         verts = [(ix[i], 0), (ix[i],iy[i]),(ix[i+1],iy[i+1]), (ix[i+1], 0)]
29         poly = Polygon(verts, facecolor='0.9', edgecolor='0.5',←
30                         hatch=patterns[i])
31         ax.add_patch(poly)
32
33     plt.title('Regla de los Trapecios $e^{x^2}$')
34     fig.savefig("integral_T_ex2_1.pdf", bbox_inches='tight')
35     return plt
36
37 def main():
38     a=0 #limite inferior
39     b=1 #limite superior
40     n=1 #numero de rectangulos

```

```

40     area=regla_trapezios(f,a,b,n)
41     print('integral = ',area)
42     g=grafica_trapezios(f,a,b,n)
43     g.show()
44
45 if __name__ == "__main__": main()

```

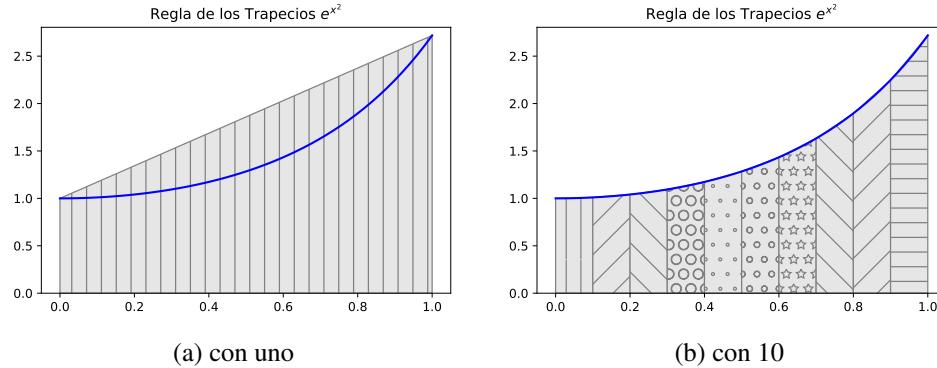


Figura 7.8. Regla de los trapecios

Regla de los trapecios con 1 trapecio

```
integral = 1.85914091423
```

Regla de los trapecios con 10 trapecios

```
integral = 1.46717469274
```

■

7.1.3 Regla de Simpson $\frac{1}{3}$

La regla de Simpson 1/3 hace la aproximación de la integral de $f(x)$ con un polinomio de segundo grado. El intervalo de integración se divide en 2 para generar un punto intermedio y obtener el polinomio entre los puntos $(x_0, f(x_0))$, $(x_1, f(x_1))$ y $(x_2, f(x_2))$.

$$\int_a^b f(x)dx \approx \int_a^b p_2(x)dx$$

Usando la interpolación de Lagrange de un polinomio de segundo grado desde x_0 hasta x_2 tenemos

$$\text{integral} = \int_{x_0}^{x_2} \left(\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \right) dx$$

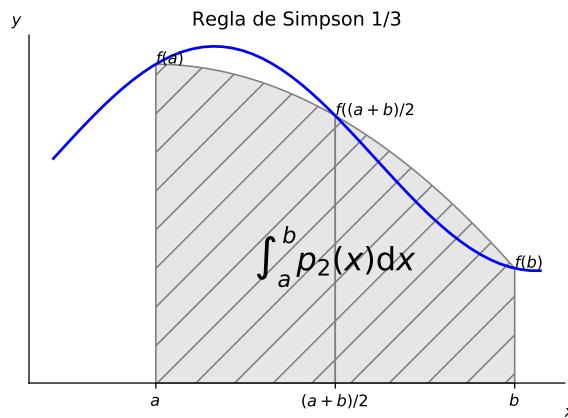


Figura 7.9. Regla de Simpson 1/3

Después de integrar y evaluar en los límites de integración

Regla de Simpson 1/3

$$\text{integral} = \frac{h}{3} (f(x_0) + 4f(x_1) + f(x_2)) \quad (7.4)$$

Donde $h = \frac{b-a}{2}$, $x_0 = a$, $x_1 = x_0 + h$ y $x_2 = b$

La versión compuesta de la regla de Simpson 1/3 se obtiene dividiendo el intervalo de integración en n subintervalos, cada par de subintervalos se aplica la regla de Simpson 1/3, la suma de cada integral es la integral del intervalo.

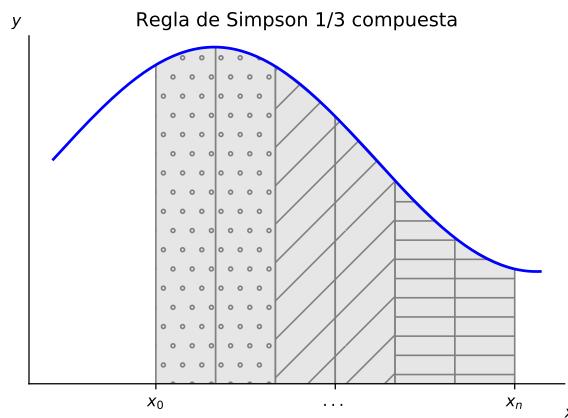


Figura 7.10. Regla de Simpson 1/3 compuesta

$$\begin{aligned}
 \text{área} = & \frac{h}{3} (f(x_0) + 4f(x_1) + f(x_2)) \\
 & + \frac{h}{3} (f(x_2) + 4f(x_3) + f(x_4)) \\
 & + \frac{h}{3} (f(x_4) + 4f(x_5) + f(x_6)) \\
 & \vdots \\
 & + \frac{h}{3} (f(x_{n-2}) + 4f(x_{n-1}) + f(x_n))
 \end{aligned}$$

Donde $h = \frac{b-a}{n}$, $x_0 = a$, $x_1 = x_0 + h$, $x_2 = b$

n es el número de subintervalos, el cual debe ser múltiplo de 2, porque cada 2 subintervalos se aplican la regla de Simpson 1/3. La ecuación se expresa como

Regla de Simpson 1/3 compuesta

$$\text{integral} = \frac{h}{3} \left(f(x_0) + 4 \sum_{i=1,3,5,\dots}^{n-1} f(x_i) + 2 \sum_{i=2,4,6,\dots}^{n-2} f(x_i) + f(x_n) \right) \quad (7.5)$$

■ **Ejemplo 7.3 — Método de Simpson $\frac{1}{3}$.** La cantidad de masa transportada por una tubería en un periodo de tiempo se calcula con la ecuación

$$M = \int_{t_1}^{t_2} Q(t)c(t)dx$$

Donde $Q(t) = 9 + 4\cos^2(0.4t)$ y $c(t) = 5e^{-0.5t} + 2e^{0.15t}$

$$M = \int_{t_1}^{t_2} (9 + 4\cos^2(0.4t)) (5e^{-0.5t} + 2e^{0.15t}) dx$$

Calcular la masa transportada entre $t_1 = 2$ y $t_2 = 8$

Solución

Programa 7.3. Método de Simpson $\frac{1}{3}$

```

1 import numpy as np
2 from scipy.interpolate import lagrange
3 from scipy.integrate import simps

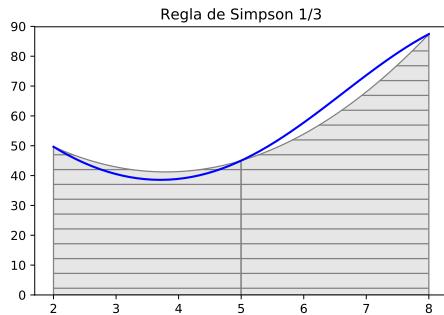
```

```
4 import matplotlib.pyplot as plt
5 from matplotlib.patches import Polygon
6
7 def regla_simpson13(f,a,b,n):
8     h=(b-a)/n
9     xs=np.linspace(a,b,n+1)
10    ys=f(xs)
11    r=h*(ys[0]+4*sum(ys[1:n:2])+2*sum(ys[2:n-1:2])+ys[n])/3
12    return r
13
14 #funcion a integrar
15 def f(x):
16     return (9+4*np.cos(0.4*x)**2)*(5*np.exp(-0.5*x)+2*np.exp(0.15*x))
17
18 def grafica_simpson(f,a,b,n):
19     x = np.linspace(a, b)
20     y = f(x)
21     fig, ax = plt.subplots()
22     ax.plot(x, y, 'b', linewidth=1.7)
23     ax.set_ylim(bottom=0)
24
25     h=(b-a)/n
26     x0,x1,x2=a,a+h,a+2*h
27     i=0
28     patterns=(-,/,\\,0,.,o,*,\\/,+-,x,+)
29     for i in range(0,n,2):
30         xx=np.array([x0,x1,x2])
31         yy=np.array([f(x0),f(x1),f(x2)])
32         pol=lagrange(xx,yy)
33
34         ix = np.linspace(x0,x1)
35         iy = pol(ix)
36         verts = [(x0, 0), *zip(ix,iy),(x1, 0)]
37         poly = Polygon(verts,facecolor='0.9', edgecolor='0.5',
38                         hatch=patterns[i])
39         ax.add_patch(poly)
40
41         ix = np.linspace(x1,x2)
42         iy = pol(ix)
43         verts = [(x1, 0), *zip(ix,iy),(x2, 0)]
44         poly = Polygon(verts,facecolor='0.9', edgecolor='0.5',
45                         hatch=patterns[i])
46         ax.add_patch(poly)
47
48     plt.title('Regla de Simpson 1/3')
49     #fig.savefig("int_simpson13C.pdf", bbox_inches='tight')
50     return plt
51
52 def main():
```

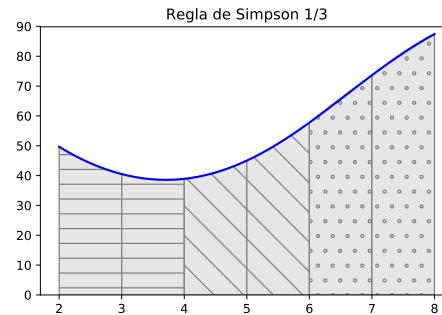
```

53     a=2 # limite inferior
54     b=8 # limite superior
55     n=6 # numero de subintervalos DEBE SER MULTIPLO DE 2
56     area=regla_simpson13(f,a,b,n)
57     print('integral = ',area)
58
59     g=grafica_simpson(f,a,b,n)
60     g.show()
61
62 if __name__ == "__main__": main()

```



(a) con 2 subintervalos



(b) con 6 subintervalos

Figura 7.11. Regla de Simpson $\frac{1}{3}$

Con 2 subintervalos

```
integral = 317.15529472
```

Con 6 subintervalos

```
integral = 322.332911492
```

■

7.1.4 Regla de Simpson $\frac{3}{8}$

La regla de Simpson 3/8 hace la aproximación de la integral de $f(x)$ con un polinomio de tercer grado. El intervalo de integración se divide en 3 subintervalos para generar 4 puntos y obtener el polinomio cúbico entre los puntos $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$ y $(x_3, f(x_3))$.

$$\int_a^b f(x)dx \approx \int_a^b p_3(x)dx$$

Usando la interpolación de Lagrange de un polinomio de tercer grado desde x_0 hasta x_3 tenemos

$$\text{integral} = \int_{x_0}^{x_3} \left(\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} f(x_0) + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} f(x_1) + \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} f(x_3) \right) dx$$

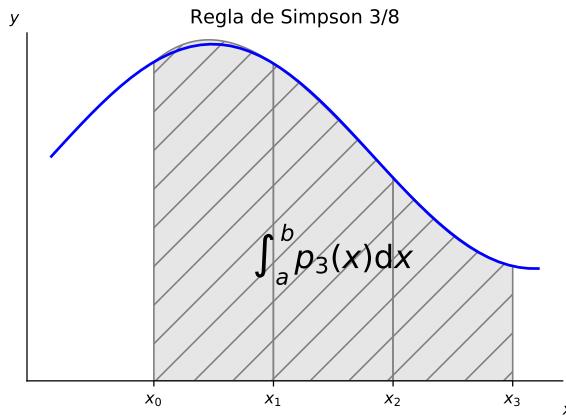


Figura 7.12. Regla de Simpson $\frac{3}{8}$

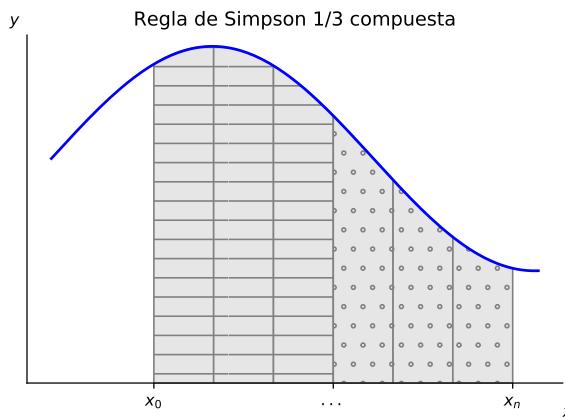
Después de integrar y evaluar en los límites de integración

Regla de Simpson 3/8

$$\text{integral} = \frac{3h}{8} (f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)) \quad (7.6)$$

Donde $h = \frac{b-a}{3}$, $x_0 = a$, $x_1 = x_0 + h$, $x_2 = x_1 + h$ y $x_3 = x_2 + h = b$

La versión compuesta de la regla de Simpson $\frac{3}{8}$ se obtiene dividiendo el intervalo de integración en n subintervalos, cada tres subintervalos se aplica la regla de Simpson $\frac{3}{8}$, la suma de cada integral es la integral del intervalo.

Figura 7.13. Regla de Simpson $\frac{3}{8}$ compuesta

$$\begin{aligned}
 \text{área} = & \frac{3h}{8} (f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)) + \\
 & + \frac{3h}{8} (f(x_3) + 3f(x_4) + 3f(x_5) + f(x_6)) \\
 & + \frac{3h}{8} (f(x_6) + 3f(x_7) + 3f(x_8) + f(x_9)) \\
 & \vdots \\
 & + \frac{3h}{8} (f(x_{n-3}) + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n))
 \end{aligned}$$

Donde $h = \frac{b-a}{n}$, n es el número de subintervalos, el cual debe ser múltiplo de 3 porque cada 3 subintervalos se aplica la regla de Simpson $\frac{3}{8}$. La ecuación se expresa como

Regla de Simpson 3/8 compuesta

$$\text{integral} = \frac{3h}{8} \left(f(x_0) + 3 \sum_{i=1,4,7,\dots}^{n-2} f(x_i) + 3 \sum_{i=2,5,8,\dots}^{n-1} f(x_i) + 2 \sum_{i=3,6,9,\dots}^{n-3} f(x_i) + f(x_n) \right) \quad (7.7)$$

■ **Ejemplo 7.4 — Método de Simpson $\frac{3}{8}$.** La cantidad de masa transportada por una tubería en un periodo de tiempo se calcula con la ecuación

$$M = \int_{t_1}^{t_2} Q(t)c(t)dx$$

Donde $Q(t) = 9 + 4\cos^2(0.4t)$ y $c(t) = 5e^{-0.5t} + 2e^{0.15t}$

$$M = \int_{t_1}^{t_2} (9 + 4\cos^2(0.4t)) (5e^{-0.5t} + 2e^{0.15t}) dx$$

Calcular la masa transportada entre $t_1 = 2$ y $t_2 = 8$

Solución

Programa 7.4. Método de Simpson $\frac{3}{8}$

```

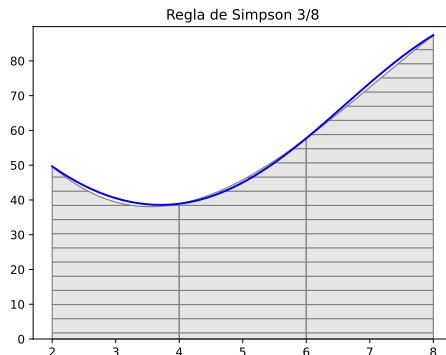
1 import numpy as np
2 from scipy.interpolate import lagrange
3 import matplotlib.pyplot as plt
4 from matplotlib.patches import Polygon
5
6 def regla_simpson38(f,a,b,n):
7     h=(b-a)/n
8     xs=np.linspace(a,b,n+1)
9     ys=f(xs)
10    r=3*h*(ys[0]+3*sum(ys[1:n-1:3])+3*sum(ys[2:n:3])+2*sum(ys[3:n-2:3])+ys[n])/8
11    return r
12
13 #funcion a integrar
14 def f(x):
15     return (9+4*np.cos(0.4*x)**2)*(5*np.exp(-0.5*x)+2*np.exp(0.15*x))
16
17 def grafica_trapezios(f,a,b,n):
18     x = np.linspace(a, b)
19     y = f(x)
20     fig, ax = plt.subplots()
21     ax.plot(x, y, 'b', linewidth=1.7)
22     ax.set_ylim(bottom=0)
23
24     h=(b-a)/n
25     x0,x1,x2,x3=a,a+h,a+2*h,a+3*h
26     i=0
27     patterns=(-,/,\\,0,.,o,*,\\/,/,-,x,+)
28     for i in range(0,n,3):
29         xx=np.array([x0,x1,x2,x3])
30         yy=np.array([f(x0),f(x1),f(x2),f(x3)])
31         pol=lagrange(xx,yy)
32
33         ix = np.linspace(x0,x1)
34         iy = pol(ix)
35         verts = [(x0, 0), *zip(ix,iy),(x1, 0)]
36         poly = Polygon(verts,facecolor='0.9', edgecolor='0.5',
37                         hatch=patterns[i])
38         ax.add_patch(poly)

```

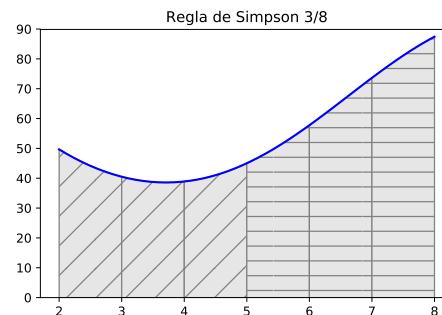
```

39     ix = np.linspace(x1,x2)
40     iy = pol(ix)
41     verts = [(x1, 0), *zip(ix,iy),(x2, 0)]
42     poly = Polygon(verts,facecolor='0.9', edgecolor='0.5', ←
43                     hatch=patterns[i])
44     ax.add_patch(poly)
45
46     ix = np.linspace(x2,x3)
47     iy = pol(ix)
48     verts = [(x2, 0), *zip(ix,iy),(x3, 0)]
49     poly = Polygon(verts,facecolor='0.9', edgecolor='0.5', ←
50                     hatch=patterns[i])
51     ax.add_patch(poly)
52
53     x0,x1,x2,x3=x3,x3+h,x3+2*h,x3+3*h
54
55     plt.title('Regla de Simpson 3/8')
56     fig.savefig("int_simpson38.pdf", bbox_inches='tight')
57     return plt
58
59 def main():
60     a=2 # limite inferior
61     b=8 # limite superior
62     n=3 # numero de subintervalos DEBE SER MULTIPLO DE 3
63     area=regla_simpson38(f,a,b,n)
64     print('integral = ',area)
65     g=grafica_trapezios(f,a,b,n)
66     g.show()
67
68 if __name__ == "__main__": main()

```



(a) con 3 subintervalos



(b) con 6 subintervalos

Figura 7.14. Regla de Simpson $\frac{3}{8}$

Con 3 subintervalos

```
integral = 320.293339145
```

Con 6 subintervalos

```
integral = 322.308419977
```

■

7.1.5 Integración de Romberg

Como se observa en los ejemplos 7.1 y 7.2, el resultado de la integral se mejora con las versiones compuestas, donde el valor de n (número de subintervalos) se incrementa. Ahora la pregunta salta inmediatamente, ¿cuál es el valor óptimo de n para que la integral sea correcta?, ¿si n tiende a infinito, el valor de la integral tiende al valor correcto?

El valor de h disminuye cuando n crece. Si observamos gráficamente los resultados de la integral con el valor de n podemos observar su comportamiento.

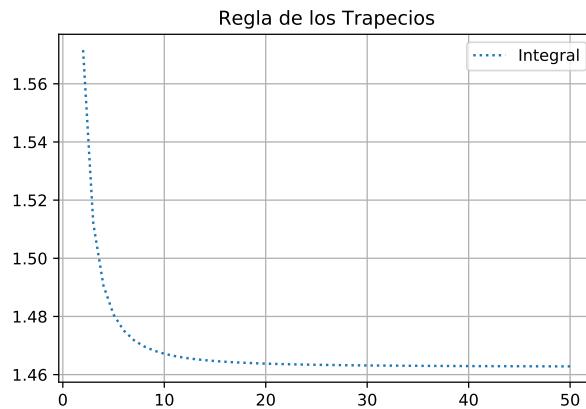


Figura 7.15. Regla de los trapecios vs número de intervalos

Se observa que al incrementar n el valor de la integral tiende a un valor constante de la integral.

La integración de Romberg se apoya en la extrapolación de Richardson, por lo que no requiere que n aumente demasiado, ya que el valor de h puede generar errores de redondeo.

El cálculo de la integral es una aproximación con un error asociado.

$$\text{Integral} = I(h) + O(h^n)$$

Donde *Integral* es el valor exacto, $I(h)$ es el valor calculado con un tamaño de intervalo h y $O(h^n)$ es el error resultado de la aproximación que significa el orden del error, la regla de los trapecios tiene un error del orden de h^2 , esto quiere decir que si el tamaño del intervalo se incrementa al doble, entonces el tamaño del intervalo es $h/2$ y el error es de $1/4$ de veces menor que el anterior. La regla de Simpson tiene un error del orden de h^4 .

La extrapolación de Richardson asume que el valor de $O(h^n)$ puede escribirse como Ch^n donde C es una constante de proporcionalidad, suponiendo que el error es constante.

$$\text{Integral} = I(h) + Ch^n \quad (7.8)$$

Si calculamos la integral con el doble de intervalos h , entonces tenemos

$$\text{Integral} = I(2h) + C(2h)^n$$

Y podemos igualar ambas expresiones de la *Integral*.

$$I(h) + Ch^n = I(2h) + C(2h)^n$$

Despejamos C

$$I(h) - I(2h) = C(2h)^n - Ch^n$$

$$I(h) - I(2h) = Ch^n(2^n - 1)$$

$$C = \frac{I(h) - I(2h)}{(2^n - 1)h^n}$$

Sustituimos C en la ecuación 7.8, tenemos

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(2h)}{(2^n - 1)h^n} \right) h^n$$

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(2h)}{(2^n - 1)} \right)$$

Para un error del orden h^2 , como la regla de los trapecios, la extrapolación de Richardson es

Extrapolación de Richardson trapecios

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(2h)}{3} \right) \quad (7.9)$$

La **ecuación 7.9** es válida si el orden del error es de h^2 , si aumentamos el número de intervalos al cuádruple, entonces $h/4$ y el orden del error es de h^4 y, por lo tanto:

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(4h)}{(2^4 - 1)} \right)$$

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(4h)}{15} \right)$$

El orden de error de la Regla de Simpson es de $O(h^4)$, por lo tanto, la extrapolación de Richardson para la regla de Simpson es también

Extrapolación de Richardson Simpson

$$\text{Integral} = I(h) + \left(\frac{I(h) - I(4h)}{15} \right) \quad (7.10)$$

En general, la extrapolación de Richardson para un orden de error n la ecuación es:

Extrapolación de Richardson de orden n

$$I_{i,j} = \frac{4^{j-1} I_{i,j-1} - I_{i-1,j-1}}{4^{j-1} - 1} \quad (7.11)$$

Para $i = 1, 2, \dots, n$ y $j = 2, \dots, i$ para un error del orden de $O(h_i^{2j})$. La ecuación 7.11 genera una tabla como la que sigue:

$$\begin{array}{ccccccccc} & & I_{1,1} & & & & & & \\ & I_{2,1} & & I_{2,2} & & & & & \\ & I_{3,1} & & I_{3,2} & & I_{3,3} & & & \\ & \vdots & & & & & & & \\ & I_{n,1} & & I_{n,2} & & I_{n,3} & \cdots & I_{n,n} & \end{array}$$

Donde la primera columna se obtiene de aplicar la regla de los trapecios con $n = 1$ para obtener $I_{1,1}$, $n = 2$ para obtener $I_{2,1}$, $n = 4$ para obtener $I_{3,1}$, hasta el último nivel $n = 2^{nivel-1}$ para obtener $I_{nivel,1}$.

Para obtener $I_{2,2}$ se requiere de $I_{1,1}$ e $I_{2,1}$; para obtener $I_{3,2}$ se requiere de $I_{2,1}$ e $I_{3,1}$; para obtener $I_{3,3}$ se requiere de $I_{2,2}$ e $I_{3,2}$; así en general para obtener $I_{i,j}$ se requiere de $I_{i,j-1}$ e $I_{i-1,j-1}$.

Los niveles continúan hasta que no haya un cambio significativo entre los valores de la integral $|I_{i,i} - I_{i,i-1}| < \varepsilon$.

■ **Ejemplo 7.5 — Método de Romberg.** Evalúe la integral de $f(x)$ de 0 a 1, usando el método de Romberg.

$$\int_0^1 e^{x^2} dx$$

Solución

Programa 7.5. Método de Romberg

```
1 import numpy as np
2 from scipy import integrate
3
4 def regla_trapezios(f,a,b,n):
5     h=(b-a)/n
6     xs=np.linspace(a,b,n+1)
7     ys=f(xs)
8     r=h*(ys[0]+2*sum(ys[1:n])+ys[n])/2
9     return r
10
11 def romberg(f,a,b,epsilon,show=True):
12     tabla=np.zeros((20,20),dtype=np.float)
13     i=0
14     sigue=True
15     while sigue:
16         n=2**i
17         tabla[i,0]=regla_trapezios(f,a,b,n)
18         for j in range(1,i+1):
19             tabla[i,j]=\
20                 (np.power(4,j)*tabla[i,j-1]-tabla[i-1,j-1])/(np.power←
21                 (4,j)-1)
22             if i>0:
23                 sigue=abs(tabla[i,i]-tabla[i-1,i-1])>epsilon
24             i+=1
25
26     salida=tabla[:i,:i]
27     if show:
28         print(salida)
29     return salida[i-1,i-1]
30
31 #funcion a integrar
32 def f(x):
33     return np.exp(x**2)
34
35
36 def main():
37     a=0 #limite inferior
38     b=1 #limite superior
39     error=1e-6 #error permitido
40
41     #llamada a la funcion romberg
42     integral=romberg(f,a,b,error)
43     print(integral)
44
45     #llamada a la funcion romberg de scipy
46     integral=integrate.romberg(f,a,b,show=True)
47     print(integral)
48
49 if __name__ == "__main__": main()
```

```

[[ 1.85914091  0.          0.          0.          0.          0.  ↵
    ]
 [ 1.57158317  1.47573058  0.          0.          0.          0.  ↵
    ]
 [ 1.49067886  1.46371076  1.46290944  0.          0.          0.  ↵
    ]
 [ 1.46971228  1.46272341  1.46265759  1.46265359  0.          0.  ↵
    ]
 [ 1.46442031  1.46265632  1.46265185  1.46265176  1.46265175  0.  ↵
    ]
 [ 1.4630941   1.46265203  1.46265175  1.46265175  1.46265175  ↵
    1.46265175]]
1.46265174591
Romberg integration of <function vectorize1.<locals>.vfunc at 0<→
x00000000C3D69D8> from [0, 1]

Steps StepSize Results
 1 1.000000  1.859141
 2 0.500000  1.571583  1.475731
 4 0.250000  1.490679  1.463711  1.462909
 8 0.125000  1.469712  1.462723  1.462658  1.462654
16 0.062500  1.464420  1.462656  1.462652  1.462652  1.462652
32 0.031250  1.463094  1.462652  1.462652  1.462652  1.462652<→
    1.462652

The final result is 1.46265174591 after 33 function evaluations.
1.46265174591

```

Observamos que el resultado del método de Romberg construido con el algoritmo es igual al resultado obtenido con el método de Romberg de scipy.

■

7.1.6 Fórmulas cerradas de Newton-Cotes de grado superior

La integral se puede seguir aproximando con las fórmulas de Newton-Cotes de un grado superior. En el **Cuadro 7.1** se resumen las primeras 10.

7.1.7 Fórmulas abiertas de Newton-Cotes

Las fórmulas de Newton-Cotes abiertas se construyen de la siguiente manera:

Integramos la función $f(x)$ entre a y b con la regla de los trapecios, pero sin utilizar los valores de $f(a)$ y $f(b)$. Tomamos entonces dos puntos internos x_1 y x_2 , de tal manera que:

$$h = \frac{b-a}{n+2} = \frac{b-a}{3}$$

Entonces $x_0 = a$, $x_1 = x_0 + h$, $x_2 = x_1 + h$ y $x_3 = x_2 + h = b$, los extremos no son evaluables, por lo que la integral la obtenemos con los nuevos límites x_1 y x_2 .

Cuadro 7.1. Fórmulas cerradas de Newton-Cotes

Grado	Puntos	Fórmula	Nombre
0	2	$\int_{x_0}^{x_1} f(x)dx = (x_1 - x_0)f(x_0)$	Regla de los rectángulos
1	2	$\int_{x_0}^{x_1} f(x)dx = (x_1 - x_0)\frac{f(x_1) - f(x_0)}{2}$	Regla de los trapezios
2	3	$\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2))$	Regla de Simpson 1/3
3	4	$\int_{x_0}^{x_3} f(x)dx = \frac{3h}{8}(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$	Regla de Simpson 3/8
4	5	$\int_{x_0}^{x_4} f(x)dx = \frac{2h}{45}(7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4))$	Regla de Bode
5	6	$\int_{x_0}^{x_5} f(x)dx = \frac{5h}{288}(19f(x_0) + 75f(x_1) + 50f(x_2) + 50f(x_3) + 75f(x_4) + 19f(x_5))$	
6	6	$\int_{x_0}^{x_6} f(x)dx = \frac{h}{140}(41f(x_0) + 216f(x_1) + 27f(x_2) + 272f(x_3) + 27f(x_4) + 216f(x_5) + 41f(x_6))$	
7	7	$\int_{x_0}^{x_7} f(x)dx = \frac{7h}{17280}(751f(x_0) + 3577f(x_1) + 1323f(x_2) + 2989f(x_3) + 2989f(x_4) + 1323f(x_5) + 3577f(x_6) + 751f(x_7))$	
8	8	$\int_{x_0}^{x_8} f(x)dx = \frac{4h}{14175}(989f(x_0) + 5888f(x_1) - 928f(x_2) + 10496f(x_3) - 4540f(x_4) + 10496f(x_5) - 928f(x_6) + 5888f(x_7) + 989f(x_8))$	
9	9	$\int_{x_0}^{x_9} f(x)dx = \frac{9h}{89600}(2857f(x_0) + 15741f(x_1) + 1080f(x_2) + 19344f(x_3) + 5778f(x_4) + 5778f(x_5) + 19344f(x_6) + 1080f(x_7) + 15741f(x_8) + 2857f(x_9))$	
10	10	$\int_{x_0}^{x_{10}} f(x)dx = \frac{5h}{299376}(16067f(x_0) + 106300f(x_1) - 48525f(x_2) + 272400f(x_3) - 260550f(x_4) + 427368f(x_5) - 260550f(x_6) + 272400f(x_7) - 48525f(x_8) + 106300f(x_9) + 16067f(x_{10}))$	

$$\int_a^b f(x)dx \approx \int_{x_1}^{x_2} \left(f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) \right) dx = \frac{3h}{2} (f(x_1) - f(x_2))$$

Se aplica el mismo procedimiento para un polinomio de segundo grado o Simpson 1/3.

$$\int_a^b f(x)dx \approx \int_{x_1}^{x_3} P_2(x)dx = \frac{4h}{3} (2f(x_1) - f(x_2) + 2f(x_3))$$

$$h = \frac{b-a}{n+2} = \frac{b-a}{4}$$

Donde $x_0 = a$, $x_1 = x_0 + h$, $x_2 = x_1 + h$, $x_3 = x_2 + h$ y $x_4 = x_3 + h = b$, los extremos no son evaluables, por lo que la integral la obtenemos con los nuevos límites x_1 y x_3 . El cuadro 7.2 resume seis de ellas.

7.2 Cuadraturas

Las fórmulas cerradas de Newton-Cotes usan los límites de integración para fijar los puntos de la función y entre esos límites trazan los polinomios de grado n para hacer la aproximación. Por ejemplo, en el método de los trapecios, se toman los puntos $x_0, f(x_0)$ y $x_1, f(x_1)$ para trazar el trapecio. Si cambiamos los límites de integración a un lugar conveniente, de tal manera que el área del trapecio sea más precisa, lograremos un mejor resultado.[2]

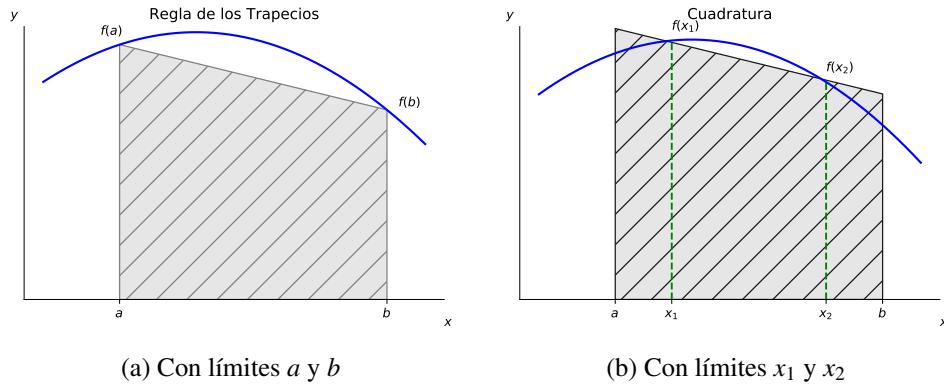


Figura 7.16. Comparación trapecios vs cuadratura

Para encontrar los puntos x_1 y x_2 , de tal manera que mejoren la integral, se aplica la técnica de los coeficientes indeterminados.

Suponga que la regla de los trapecios se puede expresar como:

$$I \approx w_1 f(x_1) + w_2 f(x_2)$$

Donde w_1, x_1, w_2, x_2 están indeterminados, por lo que se requiere de 4 ecuaciones para obtener los valores.

Cuadro 7.2. Fórmulas abiertas de Newton-Cotes

Grado	Puntos	Fórmula
0	3	$\int_{x_1}^{x_2} f(x)dx = 2hf(x_1)$
1	3	$\int_{x_1}^{x_3} f(x)dx = \frac{3h}{2}(f(x_1) + f(x_2))$
2	3	$\int_{x_1}^{x_4} f(x)dx = \frac{4h}{3}(2f(x_1) - 4f(x_2) + 2f(x_3))$
3	4	$\int_{x_1}^{x_5} f(x)dx = \frac{5h}{24}(11f(x_1) + f(x_2) + f(x_3) + 11f(x_4))$
4	5	$\int_{x_1}^{x_6} f(x)dx = \frac{6h}{20}(11f(x_1) - 14f(x_2) + 26f(x_3) - 14f(x_4) + 11f(x_5))$
5	6	$\int_{x_1}^{x_7} f(x)dx = \frac{7h}{1440}(611f(x_1) - 453f(x_2) + 562f(x_3) + 562f(x_4) - 453f(x_5) + 611f(x_6))$
6	6	$\int_{x_1}^{x_8} f(x)dx = \frac{8h}{945}(460f(x_1) - 954f(x_2) + 2196f(x_3) - 2459f(x_4) + 2196f(x_5) - 954f(x_6) + 460f(x_7))$

7.2.1 Gauss-Legendre

Como indicamos anteriormente, la cuadratura de Gauss-Legendre busca encontrar los valores de w_1, x_1, w_2 y x_2 para poder calcular la integral, por lo que se requiere de 4 ecuaciones y resolver el sistema. Las cuatro ecuaciones las obtenemos de las integrales que conocemos $1, x, x^2$ y x^3 , para simplificar los cálculos integraremos entre los límites -1 y 1. Posteriormente, haremos la transformación necesaria para que los límites sean los valores genéricos a y b .

Determinar los coeficientes de

$$\int_{-1}^1 f(x)dx = w_1 f(x_1) + w_2 f(x_2) \quad (7.12)$$

Se plantean las siguientes 4 ecuaciones

$$\int_{-1}^1 1dx = x \Big|_{-1}^1 = 1 - (-1) = 2 \Rightarrow w_1(1) + w_2(1) = 2 \quad (7.13)$$

$$\int_{-1}^1 xdx = \frac{x^2}{2} \Big|_{-1}^1 = \frac{1^2}{2} - \frac{(-1)^2}{2} = 0 \Rightarrow w_1(x_1) + w_2(x_1) = 0 \quad (7.14)$$

$$\int_{-1}^1 x^2dx = \frac{x^3}{3} \Big|_{-1}^1 = \frac{1^3}{3} - \frac{(-1)^3}{3} = \frac{2}{3} \Rightarrow w_1(x_1^2) + w_2(x_1^2) = \frac{2}{3} \quad (7.15)$$

$$\int_{-1}^1 x^3dx = \frac{x^4}{4} \Big|_{-1}^1 = \frac{1^4}{4} - \frac{(-1)^4}{4} = 0 \Rightarrow w_1(x_1^3) + w_2(x_1^3) = 0 \quad (7.16)$$

Resolver las 4 ecuaciones da como resultado

$$\begin{aligned} w_1 &= 1 & x_1 &= -\frac{1}{\sqrt{3}} \\ w_2 &= 1 & x_2 &= \frac{1}{\sqrt{3}} \end{aligned}$$

Sustituyendo estos valores en la ecuación 7.12 obtenemos

$$\int_{-1}^1 f(x)dx = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \quad (7.17)$$

La ecuación es válida para los límites de integración -1 a 1, debemos transformar estos límites para a y b . Para hacer la transformación, se crea una nueva variable z que está relacionada con x de forma lineal, donde a_0 y a_1 son coeficientes para determinar, de tal manera que relacionen los puntos a y b

$$x = a_0 + a_1 z \quad (7.18)$$

En la relación lineal se debe cumplir que si $z = -1$ entonces la relación debe dar a y, si sustituimos $z = 1$, debe ser b .

$$a = a_0 + a_1(-1) \quad (7.19)$$

$$b = a_0 + a_1(1) \quad (7.20)$$

De la ecuación 7.19 despejamos a_0 y la sustituimos en la ecuación 7.20

$$a_0 = a + a_1 \quad (7.21)$$

$$b = (a + a_1) + a_1 \quad (7.22)$$

$$a_1 = \frac{b - a}{2} \quad (7.23)$$

Sustituir la ecuación a_1 en la ecuación 7.21

$$a_0 = a + \frac{b - a}{2} \quad (7.24)$$

$$a_0 = \frac{a + b}{2} \quad (7.25)$$

Sustituir las ecuaciones 7.25 y 7.23 en la ecuación 7.18

$$x = \frac{b - a}{2}z + \frac{a + b}{2} \quad (7.26)$$

Esta ecuación es la transformación de x a los límites a y b mediante la relación lineal, se requiere la derivada de x para sustituir en la integral dx

$$dx = \frac{b - a}{2}dz \quad (7.27)$$

Finalmente sustituimos x y dx en la ecuación de la integral

$$\int_a^b f(x)dx = \frac{b - a}{2} \int_{-1}^1 f\left(\frac{b - a}{2}z + \frac{a + b}{2}\right) dz \quad (7.28)$$

Sustituyendo los valores de la integral que obtuvimos con los límites de -1 a 1 (ecuación 7.17) obtenemos

Cuadratura de Gauss-Legendre para 2 puntos

$$\int_a^b f(x)dx = \frac{b - a}{2} \left[f\left(\frac{b - a}{2} \left(-\frac{1}{\sqrt{3}}\right) + \frac{a + b}{2}\right) + f\left(\frac{b - a}{2} \left(\frac{1}{\sqrt{3}}\right) + \frac{a + b}{2}\right) \right] \quad (7.29)$$

Esta ecuación es muy parecida a la ecuación de la regla de los trapecios (ecuación 7.2), lo que cambia son los argumentos de la función porque ya no se toman los límites a y b sino

$$\left(\frac{b-a}{2} \left(-\frac{1}{\sqrt{3}}\right) + \frac{a+b}{2}\right) \text{ y } \left(\frac{b-a}{2} \left(\frac{1}{\sqrt{3}}\right) + \frac{a+b}{2}\right).$$

■ **Ejemplo 7.6 — Cuadratura de Gauss-Legendre de 2 puntos.** Estimar la cantidad de calor (cal) requerido para calentar 1 gmol de propano de 300° C a 600°C a 1 atm. La capacidad calorífica del propano está dada por:

$$C_p = 2.41 + 0.057195T - 4.3 \times 10^{-6}T^2$$

Solución

Programa 7.6. Cuadratura de Gauss-Legendre de 2 puntos

```

1 import numpy as np
2 from scipy.interpolate import lagrange
3 import matplotlib.pyplot as plt
4 from matplotlib.patches import Polygon
5
6 #cuadratura de 2 puntos
7 def cuadratura2(f,a,b):
8     x1=(b-a)/2*(-1/np.sqrt(3))+(a+b)/2
9     x2=(b-a)/2*(1/np.sqrt(3))+(a+b)/2
10    r=(b-a)/2*(f(x1)+f(x2))
11    return r
12
13 #funcion a integrar
14 def f(x):
15     return 2.41+0.057195*x-4.3e-6*x**2
16
17 def grafica_cuadratura2(f,a,b):
18     x = np.linspace(a, b)
19     y = f(x)
20     fig, ax = plt.subplots()
21     ax.plot(x, y, 'b', linewidth=1.7)
22     ax.set_ylimits(bottom=0)
23
24     x1=(b-a)/2*(-1/np.sqrt(3))+(a+b)/2
25     x2=(b-a)/2*(1/np.sqrt(3))+(a+b)/2
26
27     linea=lagrange(np.array([x1,x2]),np.array([f(x1),f(x2)]))
28
29     ax.plot(np.array([x1,x1]),np.array([0,f(x1)]), '--g')
30     ax.plot(np.array([x2,x2]),np.array([0,f(x2)]), '--g')
31
32     ax.set_ylimits(bottom=0)
33     plt.title('Cuadratura')
34
35     patterns=( '/ ', 'x', '/ ', '\\\\', '0', '.', 'o', '*', '\\\\', '/ ', '-' , 'x', '+--'
36             ')
37
38     verts = [(a, 0), (a,linea(a)),(b,linea(b)), (b, 0)]
39     poly = Polygon(verts, facecolor='0.9', edgecolor='0.1', hatch=patterns[0])
40     ax.add_patch(poly)

```

```

41     plt.title('Cuadratura de Gauss-Legendre')
42     fig.savefig("int_cuadratura2.pdf", bbox_inches='tight')
43     return plt
44
45 def main():
46     a=300+273.15 #limite inferior
47     b=600+273.15 #limite superior
48     area=cuadratura2(f,a,b)
49     print('integral = ',area)
50     g=grafica_cuadratura2(f,a,b)
51     g.show()
52
53 if __name__ == "__main__": main()

```

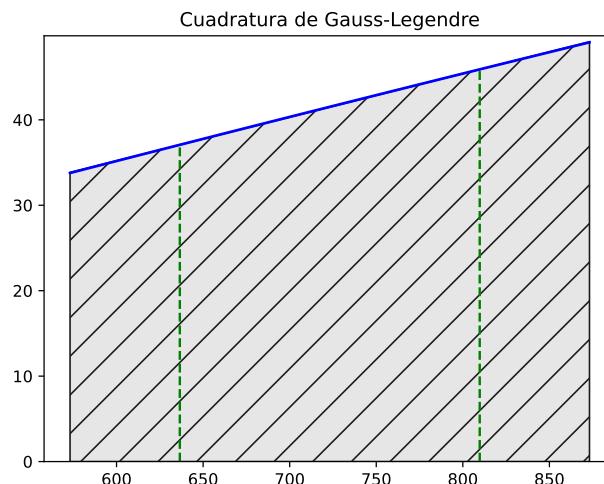


Figura 7.17. Cuadratura con 2 puntos

```
integral = 12446.894035
```

Calor = 12446.894035 cal

■

7.2.2 Cuadraturas con más puntos

La cuadratura para n puntos se puede expresar como

$$\int_{-1}^1 f(x)dx = w_1f(x_1) + w_2f(x_2) + w_3f(x_3) + \cdots + w_nf(x_n) \quad (7.30)$$

Haciendo las transformaciones de la misma manera que se hizo con la cuadratura de Gauss-Legendre de 2 puntos, ahora para n puntos tenemos

Cuadratura de Gauss-Legendre para n puntos

$$\int_a^b f(x)dx = \frac{b-a}{2} \sum_{i=1}^n \left(w_i f \left(\frac{b-a}{2} z_i + \frac{a+b}{2} \right) \right) \quad (7.31)$$

Donde n es el número de puntos iniciando con 2, w_i son los pesos y z_i son los factores que afectan a cada punto. A continuación, se muestran los valores de w_i y z_i para los primeros 6 puntos.

Puntos	w_i	z_i
2	$w_1 = 1$	$z_1 = -0.577350269$
	$w_2 = 1$	$z_2 = 0.577350269$
3	$w_1 = 0.555555556$	$z_1 = -0.774596669$
	$w_2 = 0.88888889$	$z_2 = 0$
	$w_3 = 0.555555556$	$z_3 = 0.774596669$
4	$w_1 = 0.3478548$	$z_1 = -0.861136312$
	$w_2 = 0.6521452$	$z_2 = -0.339981044$
	$w_3 = 0.6521452$	$z_3 = 0.339981044$
	$w_4 = 0.3478548$	$z_4 = 0.861136312$
5	$w_1 = 0.2369269$	$z_1 = -0.906179846$
	$w_2 = 0.4786287$	$z_2 = -0.538469310$
	$w_3 = 0.5688889$	$z_3 = 0$
	$w_4 = 0.4786287$	$z_4 = 0.538469310$
	$w_5 = 0.2369269$	$z_5 = 0.906179846$
6	$w_1 = 0.1713245$	$z_1 = -0.932469514$
	$w_2 = 0.3607616$	$z_2 = -0.661209386$
	$w_3 = 0.4679139$	$z_3 = -0.238619186$
	$w_4 = 0.4679139$	$z_4 = 0.238619186$
	$w_5 = 0.3607616$	$z_5 = 0.661209386$
	$w_6 = 0.1713245$	$z_6 = 0.932469514$

Cuadro 7.3. Cuadratura de Gauss-Legendre

■ **Ejemplo 7.7 — Cuadratura de Gauss-Legendre con n puntos.** Estimar la cantidad de calor (cal) requerido para calentar 1 gmol de propano de 300°C a 600°C a 1 atm. La capacidad calorífica del propano está dada por:

$$C_p = 2.41 + 0.057195T - 4.3 \times 10^{-6}T^2$$

Solución

Programa 7.7. Cuadratura de Gauss-Legendre con n puntos

```
1 import numpy as np
2 from scipy.integrate import quadrature,fixed_quad
3
4
5 def cuadraturaN(f,a,b,n):
6     w=np.array([[1.,1.],
7                 [0.55555556,0.88888889,0.55555556],
8                 [0.3478548,0.6521452,0.6521452,0.3478548],
9                 [0.2369269,0.4786287,0.5688889,0.4786287,\n
10                  0.2369269],
11                 [0.1713245,0.3607616,0.4679139,0.4679139,\n
12                  0.3607616,0.1713245]])\n\n
13
14     z=np.array([[-0.577350269,0.577350269],\n
15                 [-0.774596669,0,0.774596669],\n
16                 [-0.861136312,-0.339981044,0.339981044,\n
17                  0.861136312],\n
18                 [-0.906179846,-0.538469310,0,0.538469310,\n
19                  0.906179846],\n
20                 [-0.932469514,-0.661209386,-0.238619186,\n
21                  0.238619186,0.661209386,0.932469514]])\n\n
22
23     s=0
24     for i in range(n):
25         s+=w[n-2][i]*f((b-a)/2*z[n-2][i]+(a+b)/2)
26     r=(b-a)/2*s
27     return r
28
29 #funcion a integrar
30 def f(x):
31     return 2.41+0.057195*x-4.3e-6*x**2
32
33 def main():
34     a=300+273.15 #limite inferior
35     b=600+273.15 #limite superior
36     n=3           #numero de puntos
37
38     #llamada a la funcion cuadratura
39     area=cuadraturaN(f,a,b,n)
40     print('integral = ',area)
41
42
43     #llamada a la funcion quadrature de scipy
44     area=quadrature(f,a,b)
45     print('quadrature = ',area)
46
47     #llamada a la funcion fixed_quad de scipy (Gauss)
48     area=fixed_quad(f,a,b,(),n)
49     print('fixed_quad = ',area)
```

```

51 if __name__ == "__main__": main()

integral = 12446.894097186478
quadrature = (12446.894034974999, 0.0)
fixed_cuad = (12446.894034974999, None)

```

Calor = 12446.894097186478 cal ■

7.3 Integrales múltiples

Le damos a la integral simple la interpretación geométrica del área bajo la curva, la cual es la suma de cada cambio de la variable independiente x y su efecto sobre la variable dependiente. La función $f(x)$ expresa cómo es ese cambio.

La integral múltiple involucra más de una variable independiente y se debe acumular cada cambio de cada una de las variables independientes con respecto a la variable dependiente. La función $f(x_1, x_2, \dots, x_n)$ expresa cómo es ese cambio.

7.3.1 Integrales dobles y triples

La interpretación geométrica de la doble integral es el volumen, la función $f(x, y)$ expresa cómo cambia la variable dependiente por cada cambio de las variables independientes x e y . Los límites de integración deben acotar el cambio de ambas variables independientes, límites de x y límites de y .

$$\int_{y_1}^{y_2} \int_{x_1}^{x_2} f(x, y) dx dy \quad (7.32)$$

Se calcula la integral interna con respecto a la primera variable independiente indicada en la diferencial dx entre los límites x_1 y x_2 , la variable y se considera como constante, el resultado es una ecuación en términos de y , la cual se integra en los límites y_1 e y_2 para obtener el resultado final.

$$\int_{y_1}^{y_2} \left[\int_{x_1}^{x_2} f(x, y) dx \right] dy \quad (7.33)$$

■ **Ejemplo 7.8 — Integral doble.** Calcular la siguiente integral doble

$$\int_{x_1=-5}^{x_2=5} \left[\int_{y_1=-5}^{y_2=5} \sin(\sqrt{x^2 + y^2}) dy \right] dx$$

Solución

Programa 7.8. Integral doble

```

1 import numpy as np
2 from scipy.integrate import dblquad
3 import matplotlib.pyplot as plt
4 from matplotlib import cm

```

```

5
6 #funcion a integrar NOTA: el argumento y debe ser el primero, x el←
7     segundo
8 def f(y,x):
9
10    return np.sin(np.sqrt(x**2 + y**2))
11
12 def grafica(f,x1,x2,y1,y2):
13    x = np.linspace(x1, x2)
14    y = np.linspace(y1, y2)
15
16    X, Y = np.meshgrid(x, y)
17    Z = f(X, Y)
18    fig = plt.figure()
19    ax = fig.gca(projection='3d')
20    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
21                           linewidth=0, antialiased=False)
22    ax.set_xlabel('x')
23    ax.set_ylabel('y')
24    ax.set_zlabel('z')
25    ax.view_init(60, 35)
26    fig.colorbar(surf, shrink=0.5, aspect=5)
27    plt.show()
28    fig.savefig("int_cuadratura2d.pdf", bbox_inches='tight')
29
30 def main():
31    x1=-5
32    x2=5
33    y1=-5
34    y2=5
35
36    #llamada a la funcion dblquad de scipy
37    area,_=dblquad(f,           #funcion a integrar
38                    x1,x2,      #limites de x
39                    lambda y:y1,lambda y:y2 #limites de y
40                    )
41
42    print('quadrature = ',area)
43    grafica(f,x1,x2,y1,y2)
44 if __name__ == "__main__": main()

```

```
quadrature = -26.768989141374988
```

■ **Ejemplo 7.9 — Integral triple.** Calcular la siguiente integral triple

$$\int_{z_1=0}^{z_2=1} \left[\int_{y_1=0}^{y_2=1} \left[\int_{x_1=0}^{x_2=2} \left(x^2 e^{-x^2} e^{-0.5yz/x} \right) dx \right] dy \right] dz$$

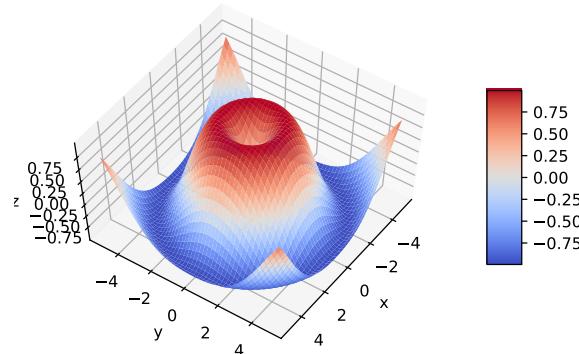


Figura 7.18. Cuadratura doble

Solución

Programa 7.9. Integral triple

```

1 import numpy as np
2 from scipy.integrate import tplquad
3
4 #funcion a integrar NOTA: el argumento "y" debe ser el primero, "x" el segundo
5 def f(x,y,z):
6     return x**2*np.exp(-x**2)*np.exp(-0.5*y*z/x)
7
8 def main():
9     x1,x2=0,2
10    y1,y2=lambda x:0,lambda x:1
11    z1,z2=lambda x,y:0,lambda x,y:1
12
13    #llamada a la funcion dblquad de scipy
14    area,_=tplquad(f,      #funcion a integrar
15                    x1,x2,    #limites de x
16                    y1,y2,    #limites de y
17                    z1,z2    #limites de z
18                    )
19
20    print('quadrature = ',area)
21
22 if __name__ == "__main__": main()

```

```
quadrature = 0.2705334762387924
```

7.4 Integral con intervalos desigualmente espaciados

Hasta ahora hemos calculado la integral definida de una función continua, lo cual es muy conveniente porque tenemos la libertad de suponer un tamaño de intervalo y evaluar la función en los puntos intermedios, pero ¿qué podemos hacer cuando no tenemos la función sino puntos discretos? Ahora estamos condicionados a los puntos conocidos que no necesariamente se encuantran equidistantes, es decir, están desigualmente espaciados y se requiere obtener la integral.

x	x_0	x_1	x_2
y	y_0	y_1	y_2

■ **Ejemplo 7.10 — Integral con intervalos desigualmente espaciados.** Calcular la integral con los siguientes datos

x	1	2	5	6	7	9
y	10	4	11	12	15	12

Solución

Programa 7.10. Integral con intervalos desigualmente espaciados

```

1 import numpy as np
2 from scipy.integrate import trapz,simps
3
4 def main():
5     x=np.array([1,2,5,6,7,9])
6     y=np.array([10,4,11,12,15,12])
7
8     #llamada a la funcion trapz y simps (el argumento "y" es ←
9     #      primero , luego "x")
10    t=trapz(y,x)
11    s=simps(y,x)
12
13    print('trapecios = ',t)
14    print('Simpson = ',s)
15
if __name__ == "__main__": main()

```

```

trapecios =  81.5
Simpson =  78.375

```

7.5 Integrales impropias

Las integrales impropias son aquellas integrales definidas donde uno o ambos límites son infinitos. Dado que no se puede evaluar la integral en un valor infinito, entonces se puede usar la cuadratura Gaussiana para evaluarlas.

- Si la función es continua en el intervalo $[a, \infty)$, entonces

$$\int_a^{\infty} f(x)dx = \lim_{b \rightarrow \infty} \int_a^b f(x)dx$$

- Si la función es continua en el intervalo $(-\infty, b]$, entonces

$$\int_{-\infty}^b f(x)dx = \lim_{a \rightarrow -\infty} \int_a^b f(x)dx$$

- Si la función es continua en el intervalo $(-\infty, \infty)$, entonces

$$\int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^c f(x)dx + \int_c^{\infty} f(x)dx$$

En cada caso, si el límite es finito, se dice que la integral impropia converge y que el límite es un valor de la integral impropia, si el límite no existe, entonces la integral diverge.

- **Ejemplo 7.11 — Integrales impropias.** Calcular la siguiente integral

$$\int_0^{\infty} \frac{e^{-x}}{\sqrt{x}} dx$$

Solución

Programa 7.11. Integral impropia

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import quad
4
5 #funcion a integrar
6 def f(x):
7     return np.exp(-x)/np.sqrt(x)
8
9 def main():
10     a=0 #limite inferior
11     b=np.Infinity #limite superior
12
13     #llamada a la funcion quad de scipy
14     area=quad(f,a,b)
15     print('quad = ',area)
16
17     #grafica de f(x)
18     fig=plt.figure()
19     x = np.linspace(a,4,100)
20     y = f(x)
21     plt.plot(x,y)
22     plt.fill_between(x,y)
23
24     plt.show()
```

```

25     fig.savefig("int_improperas.pdf", bbox_inches='tight')
26
27 if __name__ == "__main__": main()

```

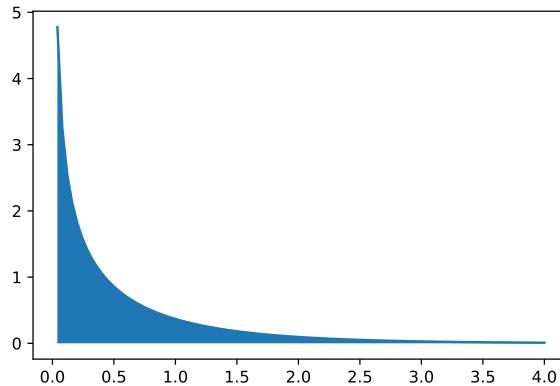


Figura 7.19. Integral impropia

```
quad = (1.7724538509051107, 9.587064475624629e-11)
```

■ **Ejemplo 7.12 — Integrales impropias 2.** Calcular la siguiente integral

$$\int_{-\infty}^{\infty} \frac{1}{(1+x^2)} dx$$

Solución

Programa 7.12. Integral impropia π

```

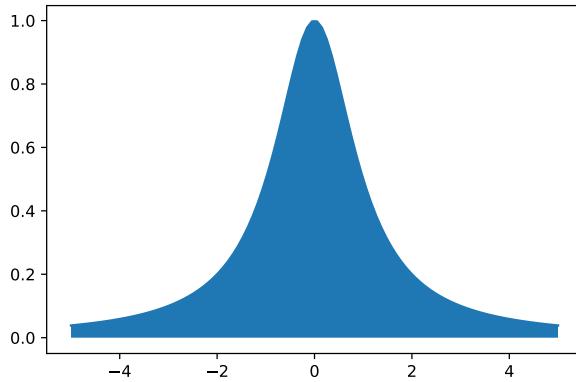
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import quad
4
5
6
7 #funcion a integrar
8 def f(x):
9     return 1/(1+x**2)
10
11 def main():
12     a=-np.Infinity #limite inferior
13     b=np.Infinity #limite superior
14
15     #llamada a la funcion quad de scipy
16     area=quad(f,a,b)

```

```

17     print('quad = ',area)
18
19     #grafica de f(x)
20     fig=plt.figure()
21     x = np.linspace(-5,5,100)
22     y = f(x)
23     plt.plot(x,y)
24     plt.fill_between(x,y)
25
26     plt.show()
27     fig.savefig("int_improperas2.pdf", bbox_inches='tight')
28
29 if __name__ == "__main__": main()

```

Figura 7.20. Integral impropia π

```
quad = (3.141592653589793, 5.155583041197975e-10)
```

■

7.6 Integrales de funciones con asíntotas verticales

Otro tipo de integrales impropias son aquellas donde la función tiene una asíntota y los límites de integración incluyen esos puntos.

- **Ejemplo 7.13 — Integrales impropias 3.** Calcular la siguiente integral

$$\int_0^1 \frac{1}{\sqrt{x}} dx$$

Solución

Programa 7.13. Integral impropia 3

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import quad
4
5 #funcion a integrar
6 def f(x):
7     return 1/np.sqrt(x)
8
9 def main():
10    a=0 #limite inferior
11    b=1 #limite superior
12
13    #llamada a la funcion quad de scipy
14    area=quad(f,a,b)
15    print('quad = ',area)
16
17    #grafica de f(x)
18    fig=plt.figure()
19    x = np.linspace(0.1,1,100)
20    y = f(x)
21    plt.plot(x,y)
22    plt.fill_between(x,y)
23
24    plt.show()
25    fig.savefig("int_improperas3.pdf", bbox_inches='tight')
26
27 if __name__ == "__main__": main()

```

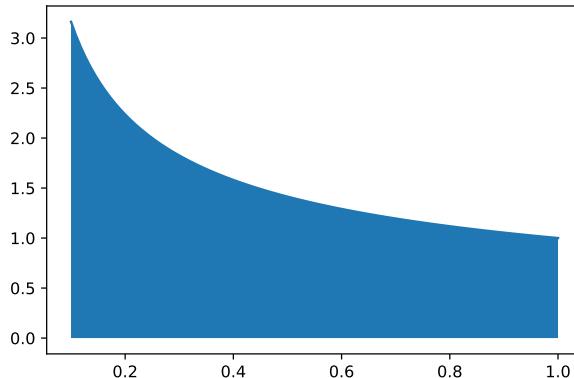
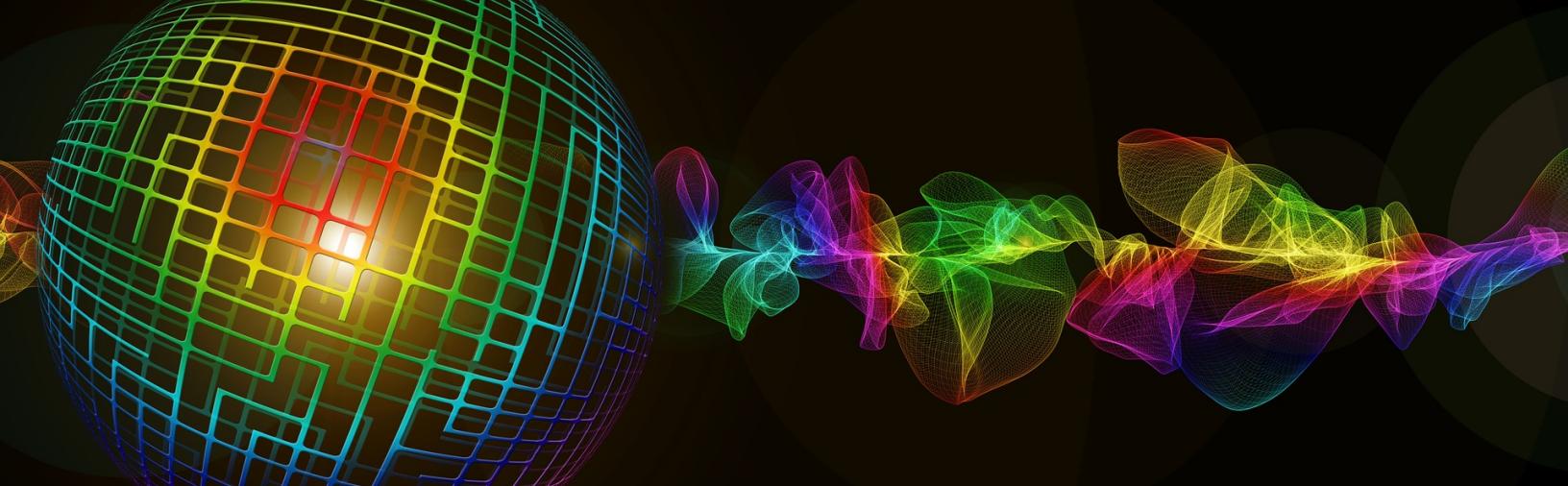


Figura 7.21. Integral impropia

```
quad = (2.0000000000000004, 5.10702591327572e-15)
```



8. Ecuaciones diferenciales ordinarias

Una ecuación diferencial ordinaria (EDO) expresa la razón de cambio de la variable dependiente y con respecto al cambio de una variable independiente x

$$\frac{dy}{dx} = f(x, y)$$

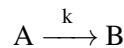
En una ecuación diferencial ordinaria, la función depende solamente de una variable y el orden se refiere a la derivada más alta. El grado de la ecuación es la potencia de la derivada de mayor orden que se tiene en la ecuación.

$$\frac{dy}{dx} = 2xy$$

La variable dependiente puede ser la población de una bacteria en el tiempo, la temperatura de una sustancia en el tiempo, la concentración de un producto en un reactor; como vemos en muchos casos, la variable independiente es el tiempo, porque se quiere medir el cambio de algo (productos, concentración, población, altura, etc.) con respecto al tiempo. Dado que cada instante es distinto al anterior.

Las ecuaciones diferenciales se resuelven eliminando las derivadas que contienen y obteniendo las funciones que cumplen con la relación expresada en la ecuación diferencial.

En la cinética de las reacciones químicas, el problema habitual es investigar las reacciones químicas y la velocidad con la que un compuesto A (reactivo) se transforma en otro compuesto B (producto).



En los casos más sencillos esta velocidad se puede expresar de la forma

$$\frac{dC_A}{dt} = -kC_A$$

Donde se expresa el cambio de la concentración de A , la cual involucra una constante de proporcionalidad, al ser negativo, entonces la concentración de A disminuye.

Matemáticamente, la ecuación diferencial tiene una infinidad de soluciones, dado que una ecuación al derivarse resuelve la ecuación diferencial, indistintamente de la constante independiente. Esto se entiende porque la razón de cambio de la concentración es la misma, sin importar la concentración inicial de A . En la gráfica 8.1a se muestran las posibles soluciones de la ecuación diferencial.

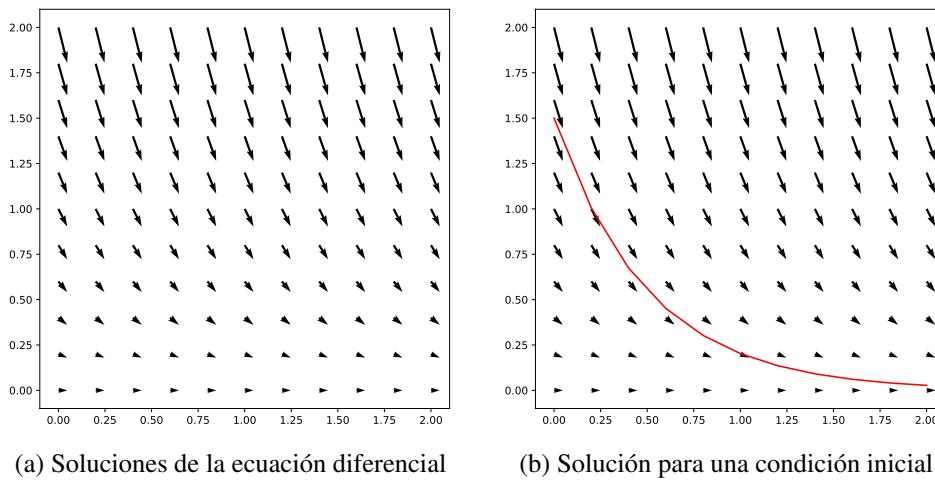


Figura 8.1. Solución de la ecuación diferencial

Si requerimos una solución en particular, se debe especificar la concentración inicial de A , condición inicial de la ecuación diferencial.

$$\frac{dC_A}{dt} = -kC_A$$

$$C_A(0) = C_{A_0}$$

Y entonces se define una solución para esa condición como se muestra en la figura 8.1b.

Los métodos aplicados a la resolución de ecuaciones diferenciales en Ingeniería Química son muchos y dependen principalmente del tipo de ecuación diferencial generado en cada problema. De modo general, podemos distinguir dos grandes grupos:

- **Métodos analíticos.** Obtienen una solución exacta. Los métodos analíticos se ajustan a un solo tipo de ecuación diferencial, por lo que no hay un método general y en algunos casos de ecuaciones diferenciales no existen métodos analíticos, implican un conocimiento de cada método para aplicarlos.
- **Métodos numéricos.** Obtienen una solución aproximada. Son métodos más genéricos y sólo dependen de las condiciones iniciales, no es una solución continua y se requiere hacer cálculos repetitivos.

8.1 Problemas de valor inicial

Como hemos visto, las ecuaciones diferenciales expresan la razón de cambio de una variable dependiente con respecto a cada cambio de la variable independiente, la solución de la ecuación diferencial debe cumplir una condición inicial dada, ya que, de lo contrario, sólo se tiene la solución general. Por ejemplo, para la ecuación diferencial

$$\frac{dC_A}{dt} = -2C_A$$

tiene como solución general

$$y = Ce^{-2x}$$

Donde C es una constante arbitraria que puede ser cualquier valor real y cada uno es una solución de la ecuación diferencial. Como se requiere un valor inicial para obtener la solución particular, entonces el problema se convierte en uno de valor inicial, se requieren de tres puntos para resolver un problema de valor inicial.

- Una ecuación diferencial de primer orden
- Un punto inicial conocido $y(x_0) = y_0$
- El valor final al que se quiere llegar.

8.1.1 Método de Euler

Para resolver una ecuación diferencial con un método numérico, sólo se cuenta con la ecuación diferencial $f(x, y)$ que gráficamente es la pendiente de una recta tangente a la función en el punto inicial x_0 y sabemos el valor de y_0 para identificar la solución única. Con estos elementos, lo que podemos construir es una línea recta con la pendiente que nos da la ecuación diferencial y evaluar cuánto vale en el punto final, esto nos dará una aproximación al valor buscado.

De tal manera que el punto calculado en $y(x_0 + h)$ se obtiene con:

$$\begin{aligned} y(x) &= y_0 + f(x_0, y_0)(x - x_0) \\ y(x_0 + h) &= y_0 + f(x_0, y_0)(x_0 + h - x_0) \\ y(x_0 + h) &= y_0 + hf(x_0, y_0) \end{aligned} \tag{8.1}$$

El cual nos da una aproximación al valor de $y(x_0 + h)$ dado que no conocemos la ecuación de y .

La diferencia entre la aproximación de Euler con el valor correcto se da porque se toma la derivada en el punto inicial x_0 para hacer la aproximación; como se observa en la figura 8.2, el valor de la pendiente cambia en el intervalo x_0 y $x_0 + h$, si tomamos cada cambio de la derivada se puede hacer una mejor aproximación.

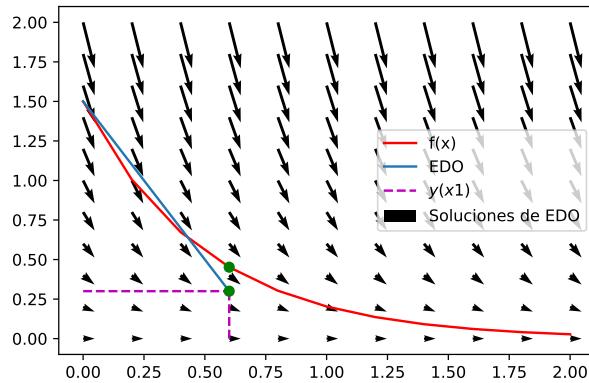


Figura 8.2. Solución con el método de Euler

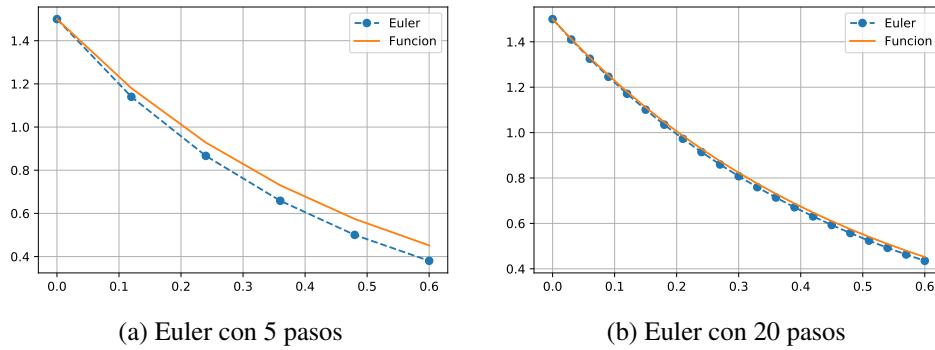


Figura 8.3. Método de Euler con pasos

Se observa en la figura 8.3a el método de Euler con 5 pasos y en la figura 8.3b con 20 pasos. Definimos h como el tamaño de paso, y n como el número de pasos.

$$h = \frac{x_n - x_0}{n}$$

Cada paso será de tamaño h , de tal manera que $x_1 = x_0 + h, x_2 = x_1 + h, \dots, x_n = x_{n-1} + h$ y los valores de cada y_{i+1} se obtienen con la siguiente ecuación:

Euler con pasos

$$y_{i+1} = y_i + h f(x_i, y_i) \quad (8.2)$$

■ **Ejemplo 8.1 — Método de Euler.** Calcular la concentración final de A si la constante de velocidad de reacción $k = 2$ y la concentración inicial $C_{A_0} = 1.5$ en el tiempo 0, calcular en el tiempo $t_n = 0.6$ $A \xrightarrow{k=2} B$

$$\frac{dC_A}{dt} = -2C_A$$

Solución

Programa 8.1. Euler con pasos

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Metodo de Euler
5 def euler(f,x0,y0,x1,n):
6     h=(x1-x0)/n          #tamano de paso
7     xi=np.zeros(n+1)      #vector de x variable independiente
8     yi=np.zeros(n+1)      #vector de y variable dependiente
9     xi[0]=x0              #tiempo iniicial
10    yi[0]=y0              #concentracion inicial
11
12    for i in range(n):
13        yi[i+1]=yi[i]+h*f(xi[i],yi[i])
14        xi[i+1]=xi[i]+h
15    return xi,yi      #Vector de valores calculados
16
17 def f(x,y):
18     return -2*y
19
20 def f2(x):
21     return np.exp(-2*x)*1.5
22
23 def main():
24     x0=0      #valor inicial de tiempo
25     y0=1.5   #valor inicial de la concentracion
26     x1=0.6   #valor final del tiempo
27     n=20     #numero de pasos
28     #llamada a la funcion euler
29     x,y=euler(f,x0,y0,x1,n)
30     print('x = ',x)
31     print('y = ',y)
32
33     #Grafica
34     fig=plt.figure()
35     plt.plot(x,y,'--',label='Euler')
36     plt.plot(np.linspace(x0,x1,50),f2(np.linspace(x0,x1,50)),label='Funcion')
37     plt.grid()
38     plt.legend()
39     plt.title('Metodo de Euler con '+str(n)+' pasos')
40     plt.xlabel('tiempo')
41     plt.ylabel('Concentracion')
42     plt.show()
43     #fig.savefig("edo_euler5.pdf", bbox_inches='tight')
44
45 if __name__ == "__main__": main()
```

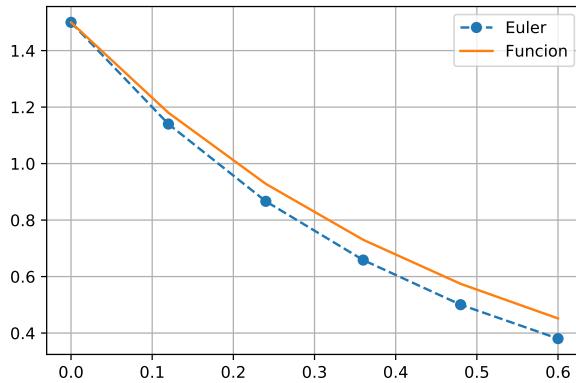


Figura 8.4. Método de Euler

```
x = [ 0.      0.12   0.24   0.36   0.48   0.6 ]
y = [ 1.5      1.14     0.8664    0.658464    0.50043264 ←
      0.38032881 ]
```

8.1.2 Método de Euler modificado

Como observamos en el método de Euler de un paso, el cálculo de y_{i+1} se obtiene con la ecuación de la línea recta con pendiente $f(x_i, y_i)$ (figura 8.5a), la cual es una aproximación con un cierto error. Si calculamos la pendiente en el punto y_{i+1} evaluando la ecuación diferencial como $f(x_{i+1}, y_{i+1})$ (figura 8.5b) y promediamos ambas pendientes, obtendremos un mejor resultado (figura 8.5c).

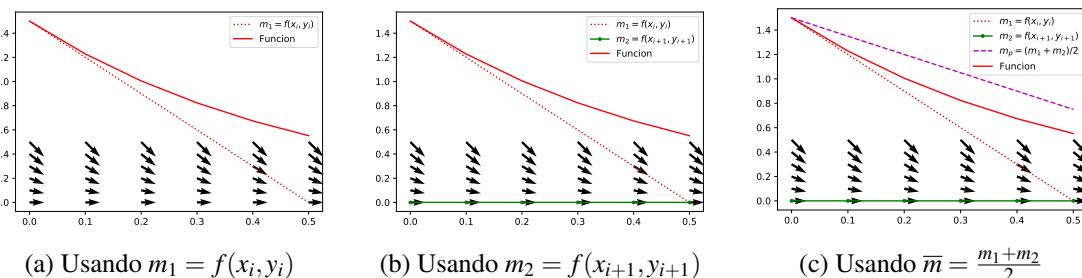


Figura 8.5. Método de Euler modificado

$$\begin{aligned}
 m_1 &= f(x_i, y_i) \\
 m_2 &= f(x_{i+1}, y_{i+1}) \\
 \bar{m} &= \frac{m_1 + m_2}{2} \\
 \bar{m} &= \left(\frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1})}{2} \right) \\
 y_{i+1}^* &= y_i + h f(x_i, y_i) \\
 y_{i+1} &= y_i + h \left(\frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^*)}{2} \right)
 \end{aligned} \tag{8.3}$$

Euler modificado

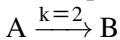
$$y_{i+1} = y_i + h \left(\frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^*)}{2} \right) \tag{8.4}$$

Primero se hace un cálculo inicial de y_{i+1} con el método de Euler, luego se hace un ajuste de y_{i+1} , recalculando con el promedio de las pendientes, para diferenciar del primer cálculo de y_{i+1} con el segundo, el primero se identifica como y_{i+1}^* para estar claros.

El método de Euler mejorado puede ser más eficiente si se hace con pasos, donde h es el tamaño de paso.

$$\begin{aligned}
 h &= \frac{x_n - x_0}{n} \\
 y_{i+1}^* &= y_i + h f(x_i, y_i) \\
 y_{i+1} &= y_i + h \left(\frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^*)}{2} \right)
 \end{aligned} \tag{8.5}$$

■ **Ejemplo 8.2 — Método de Euler modificado.** Calcular la concentración final de A si la constante de velocidad de reacción $k = 2$ y la concentración inicial $C_{A_0} = 1.5$ en el tiempo 0, calcular en el tiempo $t_n = 0.6$



$$\frac{dC_A}{dt} = -2C_A$$

Solución

Programa 8.2. Euler modificado con pasos

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3

```

```
4 #Metodo de Euler
5 def euler(f,x0,y0,x1,n):
6     h=(x1-x0)/n          #tamano de paso
7     xi=np.zeros(n+1)    #vector de x variable independiente
8     yi=np.zeros(n+1)    #vector de y variable dependiente
9     xi[0]=x0            #tiempo iniicial
10    yi[0]=y0            #concentracion inicial
11
12    for i in range(n):
13        xi[i+1]=xi[i]+h
14        yi[i+1]=yi[i]+h*f(xi[i],yi[i])
15        yi[i+1]=yi[i]+h*(f(xi[i],yi[i])+f(xi[i+1],yi[i+1]))/2
16
17    return xi,yi      #Vector de valores calculados
18
19 def f(x,y):
20     return -2*y
21
22 def f2(x):
23     return np.exp(-2*x)*1.5
24
25 def main():
26     x0=0    #valor inicial de tiempo
27     y0=1.5 #valor inicial de la concentracion
28     x1=0.6 #valor final del tiempo
29     n=5    #numero de pasos
30     #llamada a la funcion euler
31     x,y=euler(f,x0,y0,x1,n)
32     print('x = ',x)
33     print('y = ',y)
34
35 #Grafica
36 fig=plt.figure()
37 plt.plot(x,y,'o--',label='Euler modificado')
38 plt.plot(x,f2(x),label='Funcion')
39 plt.grid()
40 plt.legend()
41 plt.title('Euler modificado de 5 pasos')
42 plt.xlabel('tiempo')
43 plt.ylabel('Concentracion')
44 plt.show()
45 fig.savefig("edo_euler_mod5.pdf", bbox_inches='tight')
46
47 if __name__ == "__main__": main()
```

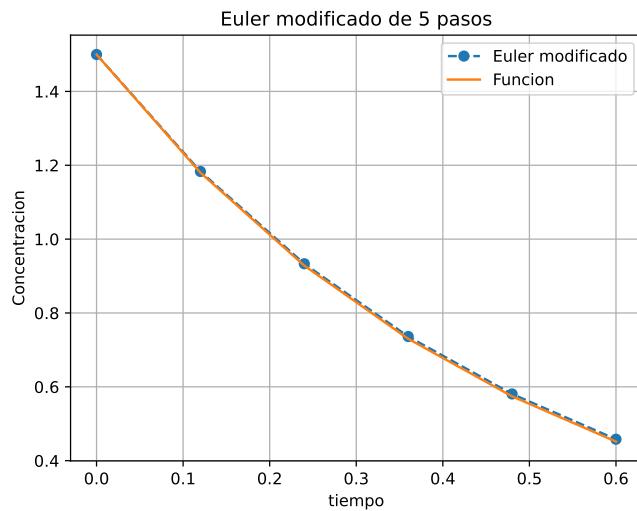


Figura 8.6. Método de Euler modificado

```

x = [ 0.      0.12    0.24   0.36    0.48    0.6 ]
y = [ 1.5          1.1832        0.93330816   0.73619348   0.58070941 ←
      0.45806359]

```

■ **Ejemplo 8.3 — Método de Euler modificado aplicado a mezclas.** Se requiere calcular la cantidad de una sustancia, $C(t)$, que hay en un tanque en cada instante de tiempo t . Sabiendo que la derivada de C respecto a t expresa la razón de cambio de la sustancia presente en el tanque, se cumple la relación

$$\frac{dC}{dt} = \text{Velocidad de entrada} - \text{Velocidad de salida}$$

Dada la velocidad a la que el fluido que contiene la sustancia entra en el tanque y la concentración de la sustancia, se cumple la relación

Velocidad de entrada = Velocidad de flujo entrante x concentración

Suponiendo que la concentración de la sustancia es uniforme, para calcular la concentración se divide $C(t)$ por el volumen de la mezcla que hay en el instante t . Así,

Velocidad de salida = Velocidad de flujo saliente x concentración.

Por ejemplo, un tanque de 1000 litros de una solución tiene 50 kg de sal. La solución se mantiene bien agitada y entra un flujo a una velocidad de 20 litros/min con una concentración de 0.08 kg/litro, por otro lado, sale un flujo de 20 litros/min, ¿cuál es la cantidad de sal pasados 30 min?

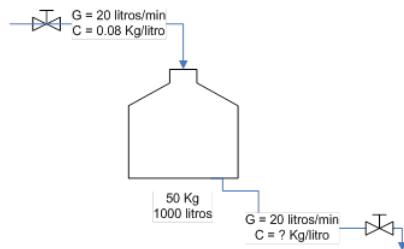


Figura 8.7. Mezcla en un tanque

Condición inicial $C_0(0) = 50$ se busca $y_n(30) = ?$

Solución

$$\frac{dC}{dt} = \text{Velocidad de entrada} \left(\frac{20 \text{ litros}}{\text{min}} \times \frac{0.08 \text{ kg}}{\text{litro}} \right) - \text{Velocidad de salida} \left(\frac{20 \text{ litros}}{\text{min}} \times \frac{C \text{ kg}}{1000 \text{ litros}} \right)$$

$$\frac{dC}{dt} = 1.6 - 0.002C$$

Programa 8.3. Euler modificado mezclas

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Metodo de Euler
5 def euler_mod(f,x0,y0,x1,n):
6     h=(x1-x0)/n          #tamano de paso
7     xi=np.zeros(n+1)      #vector de x variable independiente
8     yi=np.zeros(n+1)      #vector de y variable dependiente
9     xi[0]=x0              #tiempo iniicial
10    yi[0]=y0              #concentracion inicial
11
12    for i in range(n):
13        xi[i+1]=xi[i]+h
14        yi[i+1]=yi[i]+h*f(xi[i],yi[i])
15        yi[i+1]=yi[i]+h*(f(xi[i],yi[i])+f(xi[i+1],yi[i+1]))/2
16
17    return xi,yi      #Vector de valores calculados
18
19 def f(x,y):
20     return 1.6-0.02*y
21
22 def main():
23     x0=0      #valor inicial de tiempo
24     y0=50    #valor inicial de la concentracion
25     x1=30    #valor final del tiempo

```

```

26     n=30      #numero de pasos
27     #llamada a la funcion euler
28     x,y=euler_mod(f,x0,y0,x1,n)
29     print('x = ',x)
30     print('y = ',y)
31
32     #Grafica
33     fig=plt.figure()
34     plt.plot(x,y,'o--',label='Euler modificado')
35     plt.grid()
36     plt.legend()
37     plt.show()
38     fig.savefig("edo_euler_mod_mezclas.pdf", bbox_inches='tight')
39
40 if __name__ == "__main__": main()

```

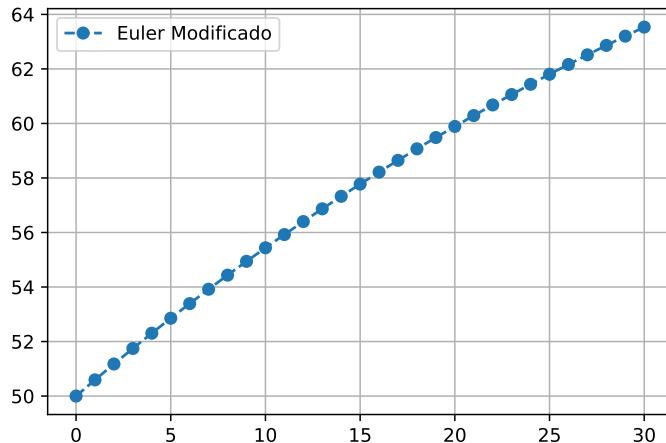


Figura 8.8. Método de Euler modificado aplicado a mezclas

```

x = [ 0.    1.    2.  ...,  28.   29.   30.]
y = [ 50.          50.594        51.1762388  ...,  62.86307865  ↵
      63.20238969
      63.53498237]

```

La cantidad de sal a los 30 min es de 63.53 kg

■

8.1.3 Runge-Kutta de 4º orden

En un problema de ecuaciones diferenciales, el objetivo es conocer el valor de la variable dependiente en un punto distinto a la condición inicial. La aproximación que se puede hacer con los elementos con que contamos es sólo la ecuación diferencial que mide el cambio de la variable dependiente, la ecuación es la del método de Euler.

$$y_{i+1} = y_i + hf(x_i, y_i)$$

Si desarrollamos la serie de Taylor (ecuación 1.7, página 15) bajo este concepto tenemos

$$y_{i+1} = y_i + hy'_i + \frac{h}{2!}y''_i + \frac{h}{3!}y'''_i + \dots$$

Utilizando la expansión de primer orden obtenemos el método de Euler, $y_{i+1} = y_i + hf(x_i, y_i)$ en donde $f(x_i, y_i)$ es la propia función f de la ecuación diferencial, es decir, la pendiente en (x_i, y_i) . En este caso, el error es del orden de h^2 .

En el método de Runge-Kutta, f es una media ponderada de valores de $f(x, y)$ en el intervalo $[x_i, x_{i+1}]$, y se dice que es de orden m si alcanza una aproximación equiparable a la del desarrollo de Taylor de ese orden.

El método de Runge-Kutta de segundo orden

$$y_{i+1} = y_i + h(ak_i + bk_2)$$

El primer término de la media ponderada es siempre la pendiente al principio del intervalo

$$k_1 = f(x_i, y_i)$$

y el segundo

$$k_2 = f(x_i + \lambda h, y_i + \mu h k_1)$$

Donde $0 < \lambda < 1$ Los pasos a,b y los números λ, μ se fijan imponiendo que el algoritmo sea compatible con un desarrollo de Taylor de orden 2, y se ha impuesto la forma $\lambda h k_1$ del segundo incremento para facilitar dicha comparación. Desarrollando k_2 ,

$$f(x_i + \lambda h, y_i + \mu h k_1) = k_1 + \frac{\partial f(x_i, y_i)}{\partial x} \lambda h + \frac{\partial f(x_i, y_i)}{\partial y} \mu h k_1 + O(h^2)$$

resulta

$$y_{i+1} = y_i + h(a+b)k_1 + h^2 b \left(\frac{\partial f(x_i, y_i)}{\partial x} \lambda h + \frac{\partial f(x_i, y_i)}{\partial y} \mu h k_1 \right) + O(h^2)$$

Comparamos ahora esta última expresión con el desarrollo de Taylor de $y(x_{i+1}) = y(x_i + h)$

$$y_{i+1} = y_i + hy'_i + \frac{h}{2!}y''_i + \dots$$

Sustituyendo $y(x_i)$ por su valor aproximado y_i y notando que

$$y_{i+1} \approx y_i + hf(x_i, y_i) + \frac{h^2}{2} \left(\frac{\partial f(x_i, y_i)}{\partial x} \lambda h + \frac{\partial f(x_i, y_i)}{\partial y} \mu f(x_i, y_i) \right)$$

Comparando se llega al sistema $a+b=1, b\lambda=1/2, b\mu=1/2$ que es indeterminado. Dejando libre b resulta $a=1-b, \lambda=\mu=b/2$. Si $b=1$, lo que proporciona un método particular de Runge-Kutta, conduce a $a=0, \lambda=\mu=1/2$

$$y_{i+1} = y_i + h f \left(x_i + \frac{h}{2}, y_i + \frac{h}{2} k_1 \right)$$

Un Runge-Kutta de tercer orden está dado por

$$y_{i+1} = y_i + h(ak_1 + bk_2 + ck_3)$$

Donde

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \lambda h, y_i + \mu \lambda k_1) \\ k_3 &= f(x_i + \lambda_2 h, y_i + \mu_2 k_2 + (\lambda_2 - \mu_2) h k_1) \end{aligned}$$

Los tres pasos a, b, c y los tres coeficientes incrementales $\mu, \lambda, \mu_2, \lambda_2$ se calculan desarrollando k_2 y k_3 en serie de Taylor de dos variables hasta orden h^2 , e identificando los factores que multiplican a h y h^2 con los correspondientes del desarrollo de Taylor de una variable de $y(x_i + h)$. El sistema de ecuaciones así obtenido es indeterminado, y sus distintas soluciones corresponden a diferentes esquemas Runge-Kutta.

El método de Runge-Kutta de cuarto orden conjuga bien la precisión con el esfuerzo de computación.

Runge-Kutta de 4° orden

$$y_{i+1} = y_i + h \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \right) \quad (8.6)$$

Donde

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} k_1\right) \\ k_3 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} k_2\right) \\ k_4 &= f(x_i + h, y_i + h k_3) \end{aligned}$$

■ **Ejemplo 8.4 — Método de Runge-Kutta de 4° orden.** Un tanque esférico de radio R está inicialmente lleno de agua. En el fondo del tanque hay un agujero de radio r, por el cual escapa el agua bajo la influencia de la gravedad. La ecuación diferencial que expresa la profundidad del agua como función del tiempo es:

$$\frac{dy}{dt} + \frac{r^2 \sqrt{2g}}{2R\sqrt{y} - \sqrt{y^3}} = 0$$

donde $g = 32.2 \text{ ft/s}^2$, $R = 12 \text{ ft}$, $r = 1/8 \text{ ft}$. La condición inicial es que en $t = 0$, $y = 22$. Encontrar la altura del agua al minuto 1000.

Solución

$$\frac{dy}{dt} = -\frac{r^2 \sqrt{2g}}{2R\sqrt{y} - \sqrt{y^3}}$$

Programa 8.4. Método de Runge-Kutta de 4º orden

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Metodo de Runge-Kutta
5 def rungekutta4o(f ,x0 ,y0 ,x1 ,n):
6     h=(x1-x0)/n          #tamano de paso
7     xi=np.zeros(n+1)      #vector de x variable independiente
8     yi=np.zeros(n+1)      #vector de y variable dependiente
9     xi[0]=x0              #tiempo iniicial
10    yi[0]=y0              #concentracion inicial
11
12    for i in range(n):
13        k1=f(xi[i],yi[i])
14        k2=f(xi[i]+h/2,yi[i]+k1*h/2)
15        k3=f(xi[i]+h/2,yi[i]+k2*h/2)
16        k4=f(xi[i]+h,yi[i]+k3*h)
17        yi[i+1]=yi[i]+h*(k1/6+k2/3+k3/3+k4/6)
18        xi[i+1]=xi[i]+h
19
20    return xi,yi      #Vector de valores calculados
21
22 #ecuacion diferencial
23 def f(x,y):
24     g=32.2
25     R=12
26     r=1/8
27     return -(r**2*np.sqrt(2*g))/(2*R*np.sqrt(y)-np.sqrt(y**3))
28
29
30 def main():
31     x0=0      #valor inicial de tiempo
32     y0=22    #valor inicial de la altura
33     x1=1000  #valor final del tiempo
34     n=10     #numero de pasos
35     #llamada a la funcion rungekutta4o
36     x,y=rungekutta4o(f,x0,y0,x1,n)
37     print('x = ',x)
38     print('y = ',y)
39     #Grafica
40     fig=plt.figure()

```

```

41     plt.plot(x,y,'o--',label='Runge-Kutta 4to orden')
42     plt.title('Altura del agua')
43     plt.xlabel('tiempo')
44     plt.ylabel('Altura')
45     plt.legend()
46     plt.show()
47 #fig.savefig("edo_rungekutta.pdf", bbox_inches='tight')
48
49 if __name__ == "__main__": main()

```

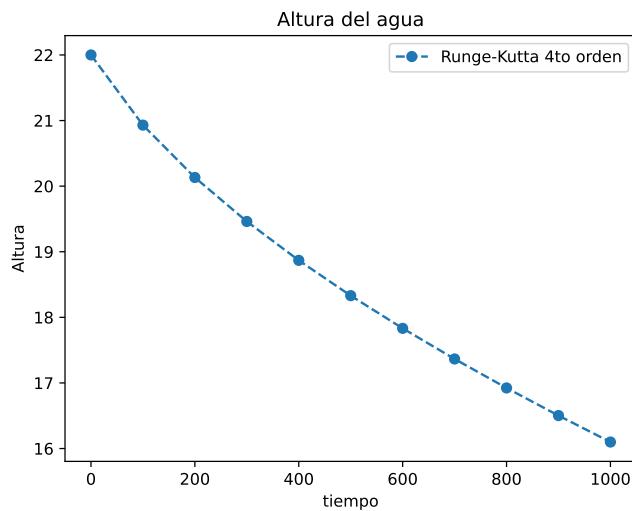


Figura 8.9. Método de Runge-Kutta 4° orden

```

x = [ 0.    100.   200. ... , 800.   900.  1000.]
y = [ 22.    20.9302613   20.13213069 ... , 16.92348771 ←
      16.50255067   16.09914568]

```

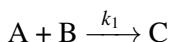
La altura del agua es de 16.099 m.

■

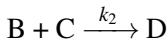
8.1.4 Sistemas de ecuaciones diferenciales de primer orden

En un sistema de ecuaciones diferenciales se expresan los cambios de una o más variables dependientes con respecto a la misma variable independiente. El sistema también expresa cómo interactúa el cambio de una variable dependiente con la otra u otras variables dependientes. La solución debería ser el estado final de cada una de ellas al final de la variable independiente.

Explicado de otra manera, desde el punto de vista de la Ingeniería Química: suponga una reacción química en un reactor:



Donde la concentración (C_A, C_B variables dependientes) de A y de B cambian de manera estequiométrica, ambas en el tiempo (variable independiente t). Ahora supongamos una segunda reacción parásita (no deseada), consecuencia de la primera:



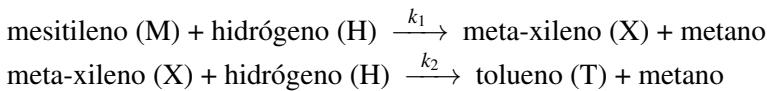
Entonces tenemos un sistema de cuatro ecuaciones diferenciales para medir el cambio de la concentración de cada sustancia.

$$\begin{aligned}\frac{dC_A}{dt} &= -k_1[C_A][C_B] \\ \frac{dC_B}{dt} &= -k_1[C_A][C_B] - k_2[C_B][C_C] \\ \frac{dC_C}{dt} &= k_1[C_A][C_B] - k_2[C_B][C_C] \\ \frac{dC_D}{dt} &= k_2[C_B][C_C]\end{aligned}\tag{8.7}$$

Se entiende que por cada cambio de concentración de A habrá un cambio de concentración de B y C por la primera reacción, y de B, C y D por la segunda reacción. Para resolver el sistema se tendrá que calcular la concentración de cada componente en cada instante hasta llegar al tiempo final.

En el sistema anterior sólo existen derivadas de primer orden, por eso se llama sistema de primer orden. El orden de un sistema de ecuaciones diferenciales es el orden de la derivada de mayor orden que aparece en el sistema. En la solución se deberá observar cómo cambia la concentración de cada componente en el tiempo.

■ **Ejemplo 8.5 — Sistema de ecuaciones diferenciales de primer orden.** El meta-xileno se produce en un reactor de flujo de pistón a 1500 R y 35 atm a partir de mesitileno. Ocurren dos reacciones en este reactor [1]



La segunda reacción no es deseable porque convierte el meta-xileno a Tolueno. El siguiente sistema se obtiene del balance

$$\begin{aligned}\frac{dC_H}{dt} &= -k_1C_H^{0.5}C_M - k_2C_H^{0.5}C_X \\ \frac{dC_M}{dt} &= -k_1C_H^{0.5}C_M \\ \frac{dC_X}{dt} &= k_1C_H^{0.5}C_M - k_2C_H^{0.5}C_X\end{aligned}$$

Donde k_1 es la constante de la reacción 1, k_2 es la constante de la reacción 2, C_H, C_M y C_X son las concentraciones de hidrógeno, mesitileno y meta-xileno en un t específico en el reactor. Las

concentraciones de hidrógeno y mesitileno en la entrada del reactor son 0.021 y 0.0105 lbmol/ ft^3 y $k_1 = 55.2 \text{ ft}^3/\text{lbmol 0.5/h}$ $k_2 = 30.2 \text{ ft}^3/\text{lbmol 0.5/h}$.

Graficar la concentración de hidrógeno, mesitileno, meta-xileno como una función de t de 0 a 0.5 h.

Determine el t óptimo en el reactor para obtener la máxima cantidad de producto.

Solución

Programa 8.5. Sistema de ecuaciones diferenciales de primer orden

```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 #Sistema de ecuaciones diferenciales
6 def f(C,t):
7     k1=55.2
8     k2=30.2
9     CH, CM, CX = C
10    #C[0] concentracion de Hidrogeno
11    #C[1] concentracion de Mesitileno
12    #C[2] concentracion de Meta-Xileno
13    dch = -k1 * CH ** 0.5 * CM - k2 * CH ** 0.5 * CX # Hidrogeno
14    dcm = -k1 * CH ** 0.5 * CM # Mesitileno
15    dcx = k1 * CH ** 0.5 * CM - k2 * CH ** 0.5 * CX # Meta - ←
16        Xileno
17    return [dch,dcm,dcx]
18
19 def main():
20     x0=0 #valor inicial de tiempo
21     #y0=[Hidrogeno Mesitileno Meta-Xylen]
22     y0=[0.021,0.0105,0.] #condiciones iniciales de
23     x1=0.5 #valor final del tiempo
24     tiempo=np.linspace(x0,x1)
25     #llamada a la funcion odeint
26     sol=odeint(f,y0,tiempo)
27
28     #obtiene la concentracion maxima de meta-xileno X
29     cbmax=np.max(sol[:,2])
30     #obtiene el indice del valor maximo
31     idx=np.where(sol[:,2]==cbmax)
32     #obtiene el valor del tiempo del indice
33     idx=tiempo[idx]
34     print('Concentración máxima {}, tiempo {}'.format(cbmax,idx))
35     #Grafica
36     fig=plt.figure()
37     plt.plot(tiempo,sol[:,0],label='$Hidrógeno (H)$')
38     plt.plot(tiempo,sol[:,1],label='$Mesitileno (M)$')
39     plt.plot(tiempo,sol[:,2],label='$Meta-xileno (X)$')
```

```

40     plt.legend()
41     plt.grid()
42     plt.show()
43     fig.savefig("edo_sistema.pdf", bbox_inches='tight')
44
45 if __name__ == "__main__": main()

```

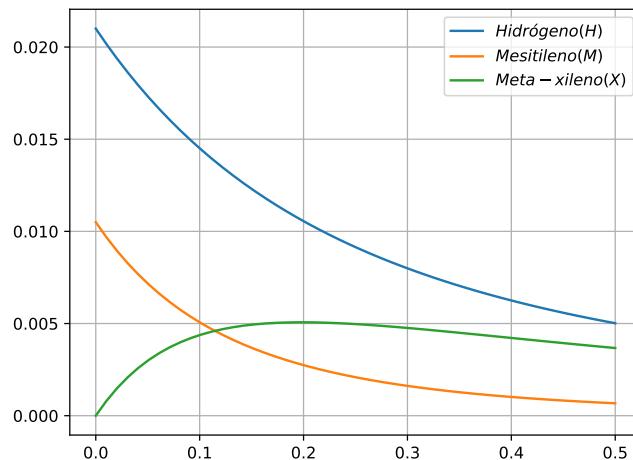


Figura 8.10. Sistema de ecuaciones diferenciales de primer orden

```
Concentracion maxima 0.005067242062323989 , tiempo [ 0.19387755]
```

■

8.2 Problemas de valor en la frontera

Un problema de valor en la frontera se refiere a una ecuación diferencial ordinaria de segundo orden

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right)$$

Donde x y y son variables independientes y $\frac{dy}{dx}$ es la derivada de y en x . La expresión señala la modificación del cambio de la variable dependiente con respecto a la variable independiente.

$$\frac{d\left(\frac{dy}{dx}\right)}{dx} = \frac{d^2y}{dx^2}$$

La expresión $\frac{d^2y}{dx^2}$ se obtiene de la multiplicación:

$$\frac{d}{dx} \left(\frac{dy}{dx} \right) = \frac{d(dy)}{(dx)(dx)} = \frac{d^2y}{dx^2}$$

Donde se entiende que se expresa la razón de cambio de dy en dx de la razón de cambio de y en x , o como se conoce: la derivada de segundo orden.

Un ejemplo de Ingeniería; imagine una barra de metal que está a una temperatura T_1 en un extremo y T_2 en el otro extremo, como se muestra en la figura 8.11. Suponiendo que sólo existe el cambio de temperatura a lo largo y no a lo ancho de la barra (sólo en la dimensión x), entonces la temperatura cambia a lo largo de la barra, ésa es la ecuación diferencial $\frac{dT}{dx}$ y la ecuación de segundo orden $\frac{d^2T}{dx^2}$ expresa cómo varía el cambio de la temperatura a lo largo de la barra.



Figura 8.11. Barra metálica

Se llama problema de valores en la frontera porque se conocen los valores de $y(x_a) = y_a$ y $y(x_b) = y_b$, es decir, se conoce el valor de la función $y(x)$ en dos puntos x_a y x_b .

En el problema de la barra metálica, se conoce el valor de la temperatura (variable dependiente) en el extremo x_1 y se conoce la temperatura en el extremo x_2 .

$$T(x_1) = T_1$$

$$T(x_2) = T_2$$

La ecuación diferencial que expresa el cambio de temperatura de la barra es

$$\frac{d^2T}{dx^2} = \alpha(T - T_a) \quad (8.8)$$

Donde α es el coeficiente de transmisión de calor que expresa la disipación de calor en el ambiente, T_a es la temperatura alrededor de la barra.

Suponga que la longitud de la barra es $L = 1\text{ m}$ y las temperaturas son $T(x_1 = 0) = 0$ y $T(x_2 = 1) = 1$ y que el coeficiente de transmisión de calor es $\alpha = 8\text{ m}^{-2}$. También asumimos que la temperatura alrededor de la barra es un gradiente de la forma

$$T_a = \frac{d}{dx} \left(\frac{T}{4} \right)$$

Entonces la ecuación diferencial es

$$\frac{d^2T}{dx^2} + 8 \left(\frac{d}{dx} \left(\frac{T}{4} \right) - T \right) = 0$$

$$\frac{d^2T}{dx^2} + 2 \frac{dT}{dx} - 8T = 0$$

8.2.1 Método de disparo

El método de disparo convierte el problema de valor en la frontera a un problema de valor inicial equivalente y puede resolverse con uno de los métodos vistos (Euler, Euler modificado o Runge-Kutta).

Se convierte la ecuación a una de primer orden haciendo un cambio de variable

$$\frac{dT}{dx} = v$$

$$\frac{d^2T}{dx^2} = \frac{dv}{dx}$$

Y para resolver el problema de valor inicial necesitamos el valor de $v(0)$ el cual se debe suponer y obtener el resultado 1, el cual se compara con la condición $T(x_2) = T_2$, si no es correcta, entonces se hace una segunda suposición de $v(0)$ y se obtiene el resultado 2, si no es correcto, es decir, no es $T(x_2) = T_2$, entonces se hace una interpolación entre los dos resultados, donde x es el valor que se desea para que la interpolación sea T_2 .

Como quien hace dos disparos con un cañón y observa dónde cae la bala, luego hace un ajuste para poder dar en el blanco en el tercer intento.

■ **Ejemplo 8.6 — Problema de valor en la frontera por el método del disparo.** Usando el ejemplo de la barra metálica mostrado en la figura 8.11 y la ecuación 8.8, y los siguientes datos:

$$L = 10m \text{ Longitud de la barra}$$

$$\alpha = 0.01m^{-2} \text{ Factor de dispersión de calor}$$

$$T_a = 25^\circ\text{C} \text{ Temperatura ambiente}$$

$$T(0) = 50^\circ\text{C} \text{ Temperatura de la barra en el extremo izquierdo}$$

$$T(L) = 200^\circ\text{C} \text{ Temperatura de la barra en el extremo derecho}$$

Solución

Convertir la ecuación de segundo orden en un sistema de ecuaciones de primer orden para aplicar los métodos vistos.

$$\begin{aligned}\frac{dT}{dx} &= v \\ \frac{d^2T}{dx^2} &= \frac{dv}{dx} = \alpha(T - T_a)\end{aligned}$$

Esto es un sistema de ecuaciones ordinarias, con los valores iniciales $T(x_1 = 0) = 50$ para la segunda ecuación y $v(x_1 = 0) = ?$ un valor que debemos suponer para obtener un primer resultado para $T(x_2 = L)$, si no es el valor esperado, entonces se hace otra suposición.

El primer valor supuesto será $v(x_1 = 0) = 10$

Programa 8.6. Problema de valores en la frontera: tiro 1

```
1 import numpy as np
2 from scipy.integrate import odeint
3 import matplotlib.pyplot as plt
4
5 def f(U, y):
6     u1, u2 = U
7     alfa = 0.01
8     Ta = 25
9     du1dy = u2
10    du2dy = alfa*(u1-Ta)
11    return [du1dy, du2dy]
12
13 def main():
14     L = 10      # Longitud de la barra
15     Tx1 = 50    # temperatura en el extremo x1=0
16     Tx2 = 200   # temperatura en el extremo x2=L
17     dT = 10     # Valor supuesto de la derivada <<<<<<<< Nuevo ←
18             valor
19
20     longitud = np.linspace(0, L) #Vector de la longitud de la ←
21             barra
22     # Solucion del sistema
23     U = odeint(f, [Tx1, dT], longitud)
24     print('Temperatura en el extremo x2=L : {}'.format(U[-1,0]))
25     #Grafica
26     fig=plt.figure()
27     plt.plot(longitud, U[:,0],label='Primer tiro')
28     plt.plot([L],[Tx2], 'ro',label='Valor esperado')
29     plt.legend()
30     plt.grid()
31     plt.title('Solución tiro 1')
32     plt.xlabel('Longitud')
33     plt.ylabel('Temperatura')
34     fig.savefig("edo_bvp10.pdf", bbox_inches='tight')
35
36 if __name__ == "__main__": main()
```

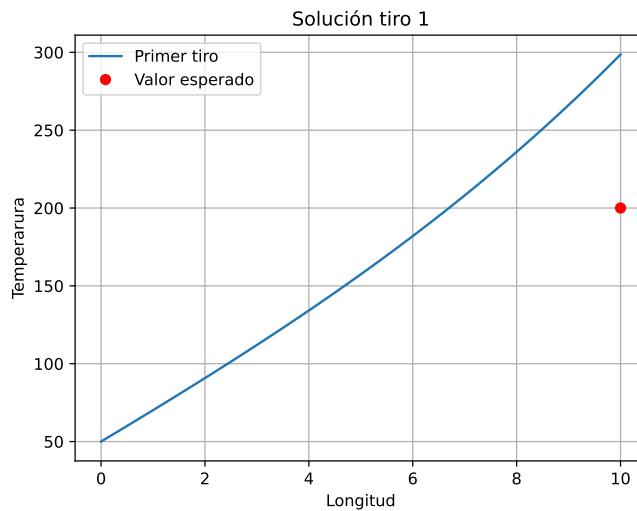


Figura 8.12. Solución del primer tiro

```
Temperatura en el extremo x2=L : 181.09713828684858
```

El resultado 181.09 es menor al esperado de 200, suponemos un valor más alto $v(x_1 = 0) = 20$

Programa 8.7. Problema de valores en la frontera: tiro 2

```

1 import numpy as np
2 from scipy.integrate import odeint
3 import matplotlib.pyplot as plt
4
5 def f(U, y):
6     u1, u2 = U
7     alfa = 0.01
8     Ta = 25
9     du1dy = u2
10    du2dy = alfa*(u1-Ta)
11    return [du1dy, du2dy]
12
13 def main():
14     L = 10      # Longitud de la barra
15     Tx1 = 50    # temperatura en el extremo x1=0
16     Tx2 = 200   # temperatura en el extremo x2=L
17     dT = 20     # Valor supuesto de la derivada <<<<<<< Nuevo ←
18             valor
19
20     longitud = np.linspace(0, L) #Vector de la longitud de la ←
21             barra
22     # Solucion del sistema
23     U = odeint(f, [Tx1, dT], longitud)
24     print('Temperatura en el extremo x2=L : {}'.format(U[-1,0]))
25     #Grafica

```

```

24     fig=plt.figure()
25     plt.plot(longitud, U[:,0],label='Primer tiro')
26     plt.plot([L],[Tx2], 'ro',label='Valor esperado')
27     plt.legend()
28     plt.grid()
29     plt.title('Solución tiro 1')
30     plt.xlabel('Longitud')
31     plt.ylabel('Temperatura')
32     fig.savefig("edo_bvp10.pdf", bbox_inches='tight')
33
34 if __name__ == "__main__": main()

```

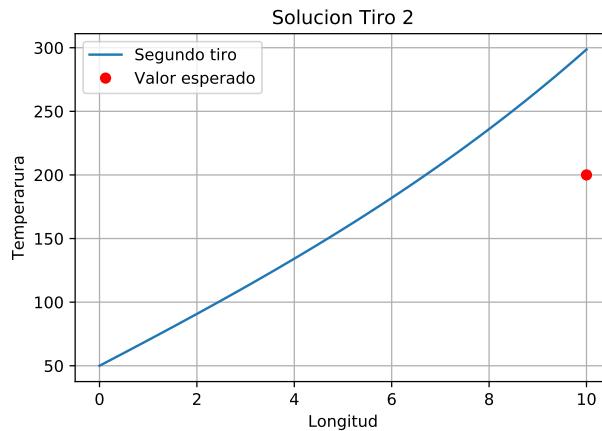


Figura 8.13. Solución del segundo tiro

```
Temperatura en el extremo x2=L : 298.6172583809675
```

El resultado 298.61 es mayor al esperado 200. Hacemos el ajuste del tiro con la interpolación lineal de Lagrange.

v	10	20
T	181.09	298.61

$$v = \frac{(200 - 298.61)}{(181.09 - 298.61)} 10 + \frac{(200 - 181.09)}{(298.61 - 181.09)} 20 = 11.609$$

Programa 8.8. Problema de valores en la frontera: tiro 3

```

1 import numpy as np
2 from scipy.integrate import odeint
3 import matplotlib.pyplot as plt
4
5 def f(U, y):

```

```

6     u1, u2 = U
7     alfa = 0.01
8     Ta = 25
9     du1dy = u2
10    du2dy = alfa*(u1-Ta)
11    return [du1dy, du2dy]
12
13 def main():
14     L = 10      # Longitud de la barra
15     Tx1 = 50    # temperatura en el extremo x1=0
16     Tx2 = 200   # temperatura en el extremo x2=L
17     dT = 11.609 # Valor supuesto de la derivada <<<<<<< ←
18         Nuevo valor
19
20     longitud = np.linspace(0, L) #Vector de la longitud de la ←
21         barra
22     # Solucion del sistema
23     U = odeint(f, [Tx1, dT], longitud)
24     print('Temperatura en el extremo x2=L : {}'.format(U[-1,0]))
25     #Grafica
26     fig=plt.figure()
27     plt.plot(longitud, U[:,0],label='Tercer tiro')
28     plt.plot([L],[Tx2], 'ro',label='Valor esperado')
29     plt.legend()
30     plt.grid()
31     plt.title('Solución tiro 3')
32     plt.xlabel('Longitud')
33     plt.ylabel('Temperatura')
34     fig.savefig("edo_bvp11.pdf", bbox_inches='tight')
35
36 if __name__ == "__main__": main()

```

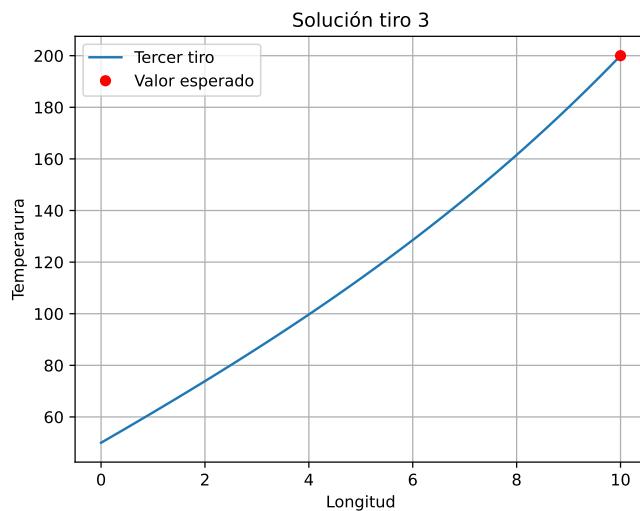


Figura 8.14. Solución del tercer tiro

```
Temperatura en el extremo x2=L : 200.006125528332
```

El resultado es 200.00 °C en el extremo derecho, que coincide con el valor esperado, por lo tanto, las temperaturas del interior de la barra son las esperadas.

■

8.2.2 Método de diferencias finitas

En el capítulo 6 revisamos cómo se puede expresar una derivada numérica por medio de sus diferencias finitas.

El método de las diferencias finitas consiste en aproximar las derivadas con diferencias donde el incremento de x no tiende a cero, sino a un valor finito como sigue:

$$\frac{dy(x_i)}{dx} = \frac{y(x_{i+1}) - y(x_{i-1})}{2h} \quad (8.9)$$

y

$$\frac{d^2y(x_i)}{dx^2} = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2} \quad (8.10)$$

Luego se sustituyen en la ecuación diferencial usando estas expresiones para distintos puntos x_i para obtener un sistema de ecuaciones cuya solución son los puntos intermedios $y(x_i)$.

La sustitución de las derivadas por sus diferencias finitas genera la discretización de una ecuación continua en incógnitas, que se tendrán que resolver para conocer su valor.

■ **Ejemplo 8.7 — Problema de valor en la frontera método de diferencias finitas.** Usando el ejemplo de la barra metálica mostrado en la figura 8.11 y la ecuación 8.8, y los siguientes datos:

$$L = 10m \text{ Longitud de la barra}$$

$$\alpha = 0.02m^{-2} \text{ Factor de dispersión de calor}$$

$$T_a = 25^\circ\text{C} \text{ Temperatura ambiente}$$

$$T(0) = 50^\circ\text{C} \text{ Temperatura de la barra en el extremo izquierdo}$$

$$T(L) = 200^\circ\text{C} \text{ Temperatura de la barra en el extremo derecho}$$

Solución

Usando la aproximación a la segunda derivada de la ecuación 8.10

$$\frac{T(x_{i+1}) - 2T(x_i) + T(x_{i-1})}{h^2} + \alpha(T_a - T(x_i)) = 0$$

Para i=1 y h=1 tenemos

$$\begin{aligned} T(x_2) - 2T(x_1) + T(x_0) + \alpha T_a - \alpha T(x_1) &= 0 \\ -2.02T(x_1) + T(x_2) &= -50.5 \end{aligned}$$

Para i=2

$$\begin{aligned} T(x_3) - 2T(x_2) + T(x_1) + \alpha T_a - \alpha T(x_2) &= 0 \\ T(x_1) - 2.02T(x_2) + T(x_3) &= -0.5 \end{aligned}$$

Para i=3

$$\begin{aligned} T(x_4) - 2T(x_3) + T(x_2) + \alpha T_a - \alpha T(x_3) &= 0 \\ T(x_2) - 2.02T(x_3) + T(x_4) &= -0.5 \end{aligned}$$

Para i=4

$$\begin{aligned} T(x_5) - 2T(x_4) + T(x_3) + \alpha T_a - \alpha T(x_4) &= 0 \\ T(x_3) - 2.02T(x_4) + T(x_5) &= -0.5 \end{aligned}$$

Para i=5

$$\begin{aligned} T(x_6) - 2T(x_5) + T(x_4) + \alpha T_a - \alpha T(x_5) &= 0 \\ T(x_4) - 2.02T(x_5) + T(x_6) &= -0.5 \end{aligned}$$

Para i=6

$$\begin{aligned} T(x_7) - 2T(x_6) + T(x_5) + \alpha T_a - \alpha T(x_6) &= 0 \\ T(x_5) - 2.02T(x_6) + T(x_7) &= -0.5 \end{aligned}$$

Para i=7

$$\begin{aligned} T(x_8) - 2T(x_7) + T(x_6) + \alpha T_a - \alpha T(x_7) &= 0 \\ T(x_6) - 2.02T(x_7) + T(x_8) &= -0.5 \end{aligned}$$

Para i=8

$$\begin{aligned} T(x_9) - 2T(x_8) + T(x_7) + \alpha T_a - \alpha T(x_8) &= 0 \\ T(x_7) - 2.02T(x_8) + T(x_9) &= -0.5 \end{aligned}$$

Para i=9

$$\begin{aligned} T(x_{10}) - 2T(x_9) + T(x_8) + \alpha T_a - \alpha T(x_9) &= 0 \\ T(x_8) - 2.02T(x_9) &= -200.5 \end{aligned}$$

Obtenemos un sistema de ecuaciones que se tendrá que resolver usando alguno de los métodos vistos en el capítulo 4. El sistema resultante es diagonalmente dominante, por lo tanto, los métodos que se pueden usar son Jacobi, Gauss-Seider o SOR.

El vector de las constantes expresa las entradas al sistema, en este caso, las temperaturas en la frontera, la temperatura ambiente y el factor de dispersión de calor.

Si expresamos el sistema de ecuaciones de forma matricial, tenemos:

$$\left(\begin{array}{ccccccccc} -2.02 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2.02 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2.02 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2.02 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2.02 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2.02 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2.02 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2.02 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2.02 \end{array} \right) \begin{pmatrix} T(x_1) \\ T(x_2) \\ T(x_3) \\ T(x_4) \\ T(x_5) \\ T(x_6) \\ T(x_7) \\ T(x_8) \\ T(x_9) \end{pmatrix} = \begin{pmatrix} -50.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -200.5 \end{pmatrix}$$

Programa 8.9. Problema de valores en la frontera: diferencias finitas

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     a=np.array([[ -2.02,  1,  0,  0,  0,  0,  0,  0,  0 ], \
6                 [1,-2.02,  1,  0,  0,  0,  0,  0,  0 ], \
7                 [0,  1,-2.02,  1,  0,  0,  0,  0,  0 ], \
8                 [0,  0,  1,-2.02,  1,  0,  0,  0,  0 ], \
9                 [0,  0,  0,  1,-2.02,  1,  0,  0,  0 ], \
10                [0,  0,  0,  0,  1,-2.02,  1,  0,  0 ], \
11                [0,  0,  0,  0,  0,  1,-2.02,  1,  0 ], \
12                [0,  0,  0,  0,  0,  0,  1,-2.02,  1 ], \
13                [0,  0,  0,  0,  0,  0,  0,  1,-2.02 ]])
14 b=np.array([[-50.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-0.5,-200.5]])
15 n,c=np.shape(a)
16 r=np.linalg.matrix_rank(a)
17 ab=np.c_[a,b]
18 ra=np.linalg.matrix_rank(ab)
19
20 print('rango(A)={} rango(AB)={} n={}'.format(r,ra,n))
21
22 if (r==ra==n):

```

```
23     print('solucion unica')
24     x=np.linalg.solve(a, b) #Resuelve el sistema
25     print(x)
26     L=10      #Longitud de la barra
27     n=10      #Pasos
28     T0=50    #Condicion en la frontera lado izquierdo
29     TL=200   #Condicion en la frontera lado derecho
30     x_plot=np.linspace(0,L,n+1) #Vector de longitud de la ←
31           barra
32     y_plot=np.append([T0],x)      #Aregar el lado ←
33           izquierdo
34     y_plot=np.append(y_plot,[TL])#Aregar el lado derecho
35
36     fig=plt.figure()
37     plt.plot(x_plot, y_plot)
38     plt.grid()
39     plt.xlabel("x")
40     plt.ylabel("Temperatura")
41     plt.show()
42     fig.savefig("edo_bvp_diffin.pdf", bbox_inches='tight'←
43           )
44 if (r==ra<n):
45     print('multiples soluciones')
46
47 if __name__ == "__main__": main()
```

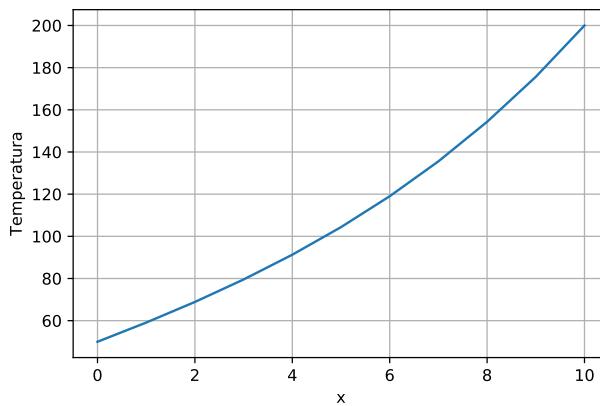
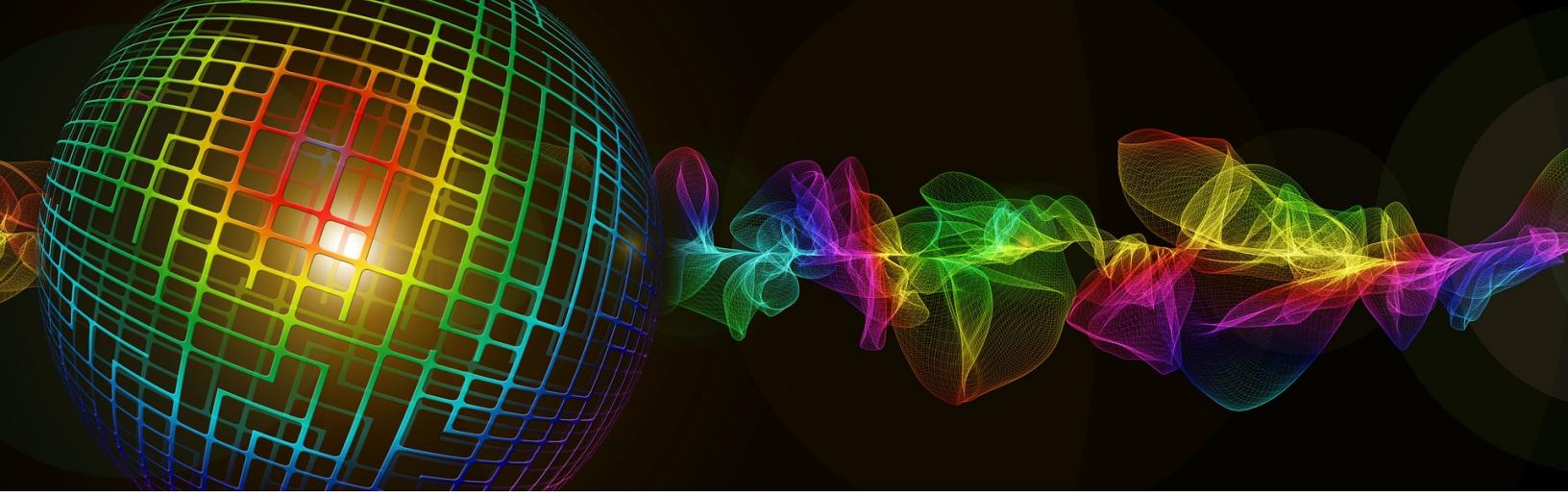


Figura 8.15. Solución del sistema de ecuaciones

```
rango(A)=9 rango(AB)=9 n=9
solucion unica
[ 59.09774126    68.87743734    79.53468217    91.28262064 ←
 104.35621152
 119.01692664   135.55798029   154.31019354   175.64861066]
```

■



9. Ecuaciones diferenciales parciales

Una ecuación diferencial parcial (EDP) es una ecuación que involucra una función multivariable desconocida y sus derivadas con dos o más variables independientes. Este tipo de ecuaciones describen fenómenos físicos como el calor, el sonido, la dinámica de fluidos, etc.

El orden de una EDP se refiere a la derivada más alta de dicha ecuación. El grado está determinado por la derivada parcial más alta. Por ejemplo

- **Primer orden**

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0$$

- **Segundo orden**

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0$$

- **Tercer orden**

$$\frac{\partial^3 u}{\partial x^3} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0$$

La mayoría de los problemas de Ingeniería están expresados por este tipo de ecuaciones y fundamentalmente por ecuaciones diferenciales de segundo orden. Una EDP de segundo orden genérica tiene la forma:

$$A(x,y) \frac{\partial^2 u}{\partial x^2}(x,y) + 2B(x,y) \frac{\partial^2 u}{\partial x \partial y}(x,y) + C(x,y) \frac{\partial^2 u}{\partial y^2}(x,y) + \\ a(x,y) \frac{\partial u}{\partial x}(x,y) + b(x,y) \frac{\partial u}{\partial y}(x,y) + c(x,y)u(x,y) = f(x,y)$$

En una cierta región $\mathcal{U} \subset R^2$ se dice que la EDP es de tipo:

- **Elíptica** si $\Delta = B^2 - 4AC < 0$ en \mathcal{U} .

Por ejemplo, la ecuación de Laplace (estado estable con dos dimensiones espaciales)

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial x^2} = 0$$

Ecuación de Poisson

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial x^2} = f(x, y)$$

- **Parabólica** si $\Delta = B^2 - 4AC = 0$ en \mathcal{U} .

Por ejemplo, la ecuación de calor en una dimensión (variable de tiempo con una dimensión espacial)

$$\frac{\partial u}{\partial y} = \frac{\partial^2 u}{\partial x^2}$$

En dimensión m

$$\frac{\partial u}{\partial y} = \nabla^2 u$$

Donde ∇^2 =Laplaciano para $u(x_1, \dots, x_m)$

- **Hiperbólica** si $\Delta = B^2 - 4AC > 0$ en \mathcal{U} .

Por ejemplo, la ecuación de ondas (variable de tiempo con una dimensión espacial)

$$\frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial x^2}$$

Cada clasificación está asociada a problemas específicos y requiere de técnicas de solución especiales.

Denotemos a R como la región del plano sobre la que se define $u(x, y)$, S es la frontera de R considerada una red o malla de puntos $\{(x_i, y_j) : i = 0, 1, \dots, N+1, j = 0, 1, \dots, M+1\}$, para la región R , llamaremos $u_{ij} = u(x_i, y_j)$ será el valor exacto de la solución de la EDP (supuesta su existencia y unicidad); u_{ij} , al valor numérico proporcionado por un método de resolución aproximada de la EDP.

Si la malla es rectangular y parcialmente uniforme, denotemos por h = paso en la dirección x , k = paso en la dirección y , de modo que: $x_i = x_0 + ih$, $y_j = y_0 + jk$

Para $R = [a, b] \times [c, d]$ una red de puntos para R es la figura 9.1.

Las condiciones de frontera se agrupan en tres tipos principalmente

1. **Condiciones Dirichlet**, cuando se fija la solución a lo largo de la frontera

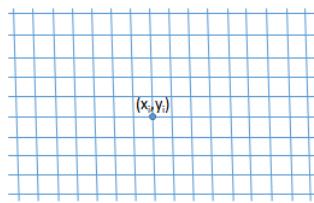


Figura 9.1. Red de puntos

2. **Condiciones Neumann**, cuando se fijan las derivadas a lo largo de la frontera
3. **Condiciones Robin**, cuando la velocidad de cambio de la variable dependiente en la frontera es una función de la misma variable dependiente.

Las condiciones de **Cauchy** son las condiciones iniciales o valores iniciales.

Las EDP pueden ser resueltas planteando distintos esquemas numéricicos donde las derivadas parciales son reemplazadas por su aproximación en diferencias finitas divididas.

9.1 Ecuaciones diferenciales parciales elípticas

Ejemplos de EDP elípticas son la ecuación de Poisson y Laplace.

Este tipo de ecuaciones permite resolver problemas de equilibrio, que son problemas donde se busca la solución de una ecuación diferencial en un dominio cerrado, sujeta a condiciones de frontera prescritas. Los ejemplos más comunes de tales problemas incluyen a distribuciones estacionarias de temperatura, flujo de fluidos incompresibles no viscosos, distribución de tensiones en sólidos en equilibrio, el campo eléctrico en una región que contenga una densidad de carga dada y, en general, problemas donde el objetivo sea determinar un potencial.

La EDP que modela este fenómeno es:

$$\frac{\partial^2 u}{\partial x^2}(x, y) + \frac{\partial^2 u}{\partial y^2}(x, y) = f(x, y)$$

Donde $f(x, y)$ es una función definida sobre la región R con borde S .

La ecuación de Poisson es del tipo elíptica, ya que $A = 1, B = 0, C = 1$, por lo tanto,

$$\Delta = B^2 - 4AC < 0$$

Esta ecuación es equivalente a:

$$\Delta u(x, y) = f(x, y)$$

Donde Δ es el operador de Laplace o Laplaciano

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

La ecuación de Laplace es un caso particular de la ecuación de Poisson donde $f(x,y) = 0$ que significa que la región R es plana

$$\Delta u(x,y) = 0$$

Si el valor de $u(x,y)$ en la región R es determinada por el valor de la función en el borde S , se les denomina condiciones de borde de Dirichlet que viene dado por:

$$u(x,y) = g(x,y), \forall (x,y) \in S$$

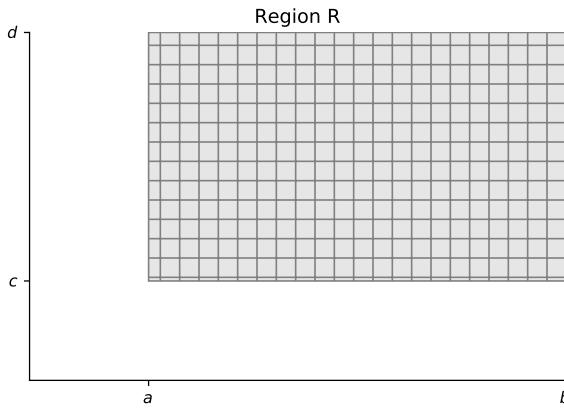


Figura 9.2. Región R

En la figura, la región R está dada por:

$$R = \{(x,y) \in R^2 : a \leq x \leq b, c \leq y \leq d\} \subset R^2, a > 0$$

9.1.1 Métodos numéricos para las EDP elípticas

Para resolver numéricamente este tipo de ecuaciones se utiliza una adaptación del método de diferencias finitas. El método consiste en generar una rejilla equiespaciada en la región R para obtener aproximaciones de la función en los puntos de la rejilla $u(x,y)$ iterativamente.

El número de puntos de la rejilla para x, y son m, n respectivamente, por lo tanto, el espaciado entre los puntos es:

$$h = \frac{b-a}{n} \text{ equiespaciado para } x$$

$$k = \frac{d-c}{m} \text{ equiespaciado para } y$$

Para expresar numéricamente $y''(x_i)$ se utiliza la ecuación de las diferencias centradas de 2º orden (ecuación 6.7 de la página 169)

$$y''(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2}$$

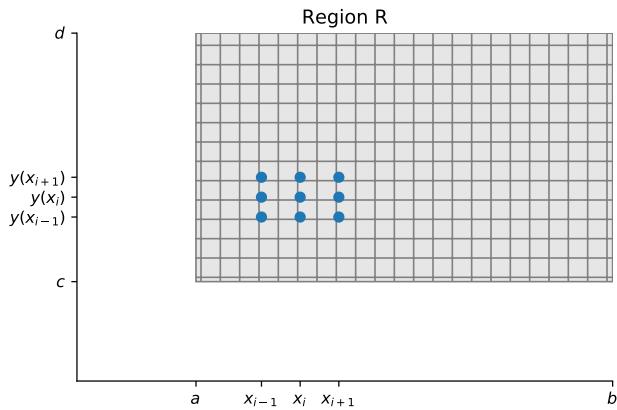


Figura 9.3. Región R

Podemos usar las expansiones de Taylor mostradas anteriormente para resolver la ecuación de Poisson expresando:

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) \approx \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{k^2}$$

Ahora podemos expresar la ecuación en el punto (x_i, y_j) aproximadamente como:

$$\frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} + \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{k^2} \approx f(x_i, y_j)$$

Agrupando términos y expresando $u_{i,j} \approx u(x_i, y_j)$ se tiene:

$$2 \left[\left(\frac{h}{k} \right)^2 + 1 \right] u_{i,j} - u_{i+1,j} - u_{i-1,j} - \left(\frac{h}{k} \right)^2 (u_{i,j+1} + u_{i,j-1}) = -h^2 f(x_i, y_j)$$

Para definir las condiciones de frontera, se debe especificar que el valor de $u(x, y)$ en la frontera está definido por la función $g(x, y)$.

$$u_{0,j} = g(x_0, y_j) \text{ y } u_{n,j} = g(x_n, y_j) \quad \forall j = 1, \dots, m$$

$$u_{i,0} = g(x_i, y_0) \text{ y } u_{i,m} = g(x_i, y_m) \quad \forall i = 1, \dots, n-1$$

Las ecuaciones anteriores definen un sistema de ecuaciones que puede ser representado matricialmente.

La solución numérica de las ecuaciones elípticas, por ejemplo, la ecuación de Laplace, se utiliza en diversas áreas de Ingeniería donde se trata con problemas que involucran la determinación de un potencial.

Para el caso de flujo de calor en régimen estacionario en una placa delgada, la expresión es

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Representamos la placa como una malla de puntos discretos (nodos), donde expresamos la ecuación diferencial como diferencias finitas, lo cual transforma a la EDP en una ecuación algebraica.

Utilizando diferencias finitas centradas de segundo orden, entonces podemos escribir:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2}$$

y

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2}$$

Reemplazando estas expresiones en la EDP, queda

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2} = 0$$

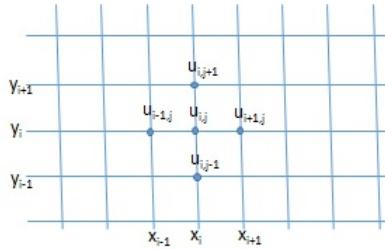


Figura 9.4. Discretización de la ecuación de Laplace

Las condiciones de frontera deben ser especificadas para que exista una solución única.

Es importante comentar que, si solamente se especifican condiciones de Neumann, existirán infinitas soluciones. Por lo tanto, para obtener una única solución deberá especificarse al menos una condición de tipo Dirichlet en algún punto de la frontera.

Además de los valores de la función potencial, suele interesar el valor de variables secundarias. En general, estas variables están asociadas al valor del flujo en cada punto del dominio y en las fronteras donde se ha especificado una condición forzada. En un caso podrá representar el flujo de energía, en otro será la velocidad, etc. Estas variables secundarias o derivadas están vinculadas con la derivada del potencial a través de una constante (o función) que multiplica al valor de la derivada en el punto. Por ejemplo, para un problema de transmisión de calor será el flujo de calor, para el escurrimiento de un fluido en un medio poroso será el campo de velocidades.

Flujo en la dirección x

$$Q_x = -\alpha \frac{\partial u}{\partial x} = -\alpha (-u_{i-1,j} + u_{i+1,j})$$

Flujo en la dirección y

$$Q_y = -\alpha \frac{\partial u}{\partial y} = -\alpha (-u_{i,j-1} + u_{i,j+1})$$

■ **Ejemplo 9.1 — Condiciones de frontera de tipo Dirichlet.** Supongamos que tenemos un problema de Dirichlet, es decir, conocemos los valores en la frontera de $u(x,y)$ en la frontera de la región R con las condiciones de frontera

- $u(x_1, y_j) = u_{1,j}$ para $2 \leq j \leq m-1$ a la izquierda
- $u(x_i, y_1) = u_{i,1}$ para $2 \leq i \leq n-1$ abajo
- $u(x_n, y_j) = u_{n,j}$ para $2 \leq j \leq m-1$ a la derecha
- $u(x_i, y_m) = u_{i,m}$ para $2 \leq i \leq n-1$ arriba

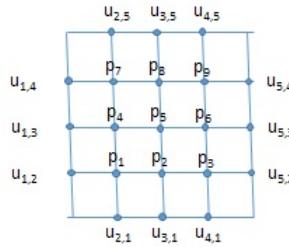


Figura 9.5. Condiciones de Dirichlet

Determinar la solución aproximada de la ecuación de Laplace en la región R , donde $u(x,y)$ denota la temperatura en un punto (x,y) , los valores de frontera son:

- $u(x,0) = 20$ para $0 \leq x \leq 4$ abajo
- $u(x,4) = 180$ para $0 \leq x \leq 4$ arriba
- $u(0,y) = 80$ para $0 \leq y \leq 4$ izquierda
- $u(4,y) = 0$ para $0 \leq y \leq 4$ derecha

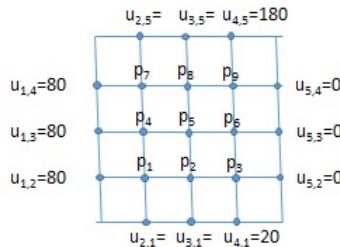


Figura 9.6. Sustituyendo las condiciones de Dirichlet

Solución

Sustituyendo en cada punto se construye el sistema

$$\begin{pmatrix} -4p_1 & p_2 & 0 & p_4 & 0 & 0 & 0 & 0 & 0 & -100 \\ p_1 & -4p_2 & p_3 & 0 & p_5 & 0 & 0 & 0 & 0 & -20 \\ 0 & p_2 & -4p_3 & 0 & 0 & p_6 & 0 & 0 & 0 & -20 \\ p_1 & 0 & 0 & -4p_4 & p_5 & 0 & p_7 & 0 & 0 & -80 \\ 0 & p_2 & 0 & p_4 & -4p_5 & p_6 & 0 & p_8 & 0 & 0 \\ 0 & 0 & p_3 & 0 & p_5 & -4p_6 & 0 & 0 & p_9 & 0 \\ 0 & 0 & 0 & p_4 & 0 & 0 & -4p_7 & p_8 & 0 & -260 \\ 0 & 0 & 0 & 0 & p_5 & 0 & p_7 & -4p_8 & p_9 & -180 \\ 0 & 0 & 0 & 0 & 0 & p_6 & 0 & p_8 & -4p_9 & -180 \end{pmatrix}$$

Resolviendo el sistema de ecuaciones lineales

Programa 9.1. Solución de Laplace con condiciones Dirichlet

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     a=np.array([[-4, 1, 0, 1, 0, 0, 0, 0, 0], \
6                [1, -4, 1, 0, 1, 0, 0, 0, 0], \
7                [0, 1, -4, 0, 0, 1, 0, 0, 0], \
8                [1, 0, 0, -4, 1, 0, 1, 0, 0], \
9                [0, 1, 0, 1, -4, 1, 0, 1, 0], \
10               [0, 0, 1, 0, 1, -4, 0, 0, 1], \
11               [0, 0, 0, 1, 0, 0, -4, 1, 0], \
12               [0, 0, 0, 0, 1, 0, 1, -4, 1], \
13               [0, 0, 0, 0, 0, 1, 0, 1, -4]])
14     b=np.array([-100,-20,-20,-80,0,0,-260,-180,-180])
15     n,c=np.shape(a)
16     r=np.linalg.matrix_rank(a)
17     ab=np.c_[a,b]
18     ra=np.linalg.matrix_rank(ab)
19
20     print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,n))
21
22     if (r==ra==n):
23         print('solucion unica')
24         x=np.linalg.solve(a, b)
25         z=x.reshape(3,3)
26         print(z)
27         X,Y=np.meshgrid(np.arange(1,4), np.arange(1,4))
28
29         fig, ax = plt.subplots()
30         CS = ax.contour(X, Y, z,10)
31         ax.clabel(CS, inline=1, fontsize=10)
32         fig.colorbar(CS, shrink=0.8, extend='both')
33         ax.set_title('Laplace con condiciones Dirichlet')
34         fig.savefig("edp_laplace_dirichlet.pdf", bbox_inches='tight')
35
36     if (r==ra<n):

```

```

37     print('multiples soluciones')
38
39     if (r<ra):
40         print('sin solucion')
41
42 if __name__ == "__main__": main()

```

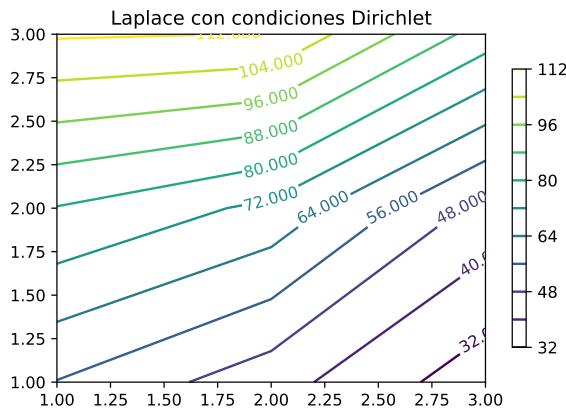


Figura 9.7. Ecuación de Laplace con condiciones Dirichlet

```

rango(A)=9 rango(Ab)=9 n=9
solucion unica
[[ 55.71428571    43.21428571    27.14285714]
 [ 79.64285714    70.          45.35714286]
 [ 112.85714286   111.78571429   84.28571429]]

```

En las condiciones de borde de **Neumann** se especifican valores de la derivada direccional de $u(x,y)$, en la dirección perpendicular al contorno R , supongamos que

$$\frac{\partial u(x,y)}{\partial n} = 0$$

En el contexto de los problemas de temperatura, significa contorno aislado, no hay flujo de calor a través de él, para $R = \{(x,y), 0 \leq x \leq a, 0 \leq y \leq b\}$ la condición de contorno de la derivada para $x = a$ es:

$$\frac{\partial u(x_n, y_j)}{\partial x} = u_x(x_n, y_j) = 0$$

La ecuación de diferencias de Laplace para (x_n, y_j)

$$u_{n+j} + u_{n-j} - 4u_{nj} + u_{nj+1} + u_{nj-1} = 0$$

Donde el valor u_{n+j} es desconocido, pues está fuera del dominio R . Sin embargo, podemos usar la fórmula de la derivación numérica

$$\frac{u_{n+j} - u_{n-j}}{2h} \approx u_x(x_n, y_j)$$

$$u_{n+j} \approx u_{n-j}$$

Y obtenemos la fórmula que relaciona u_{nj} con sus valores adyacentes

$$2u_{n-j} - 4u_{nj} + u_{nj+1} + u_{nj-1} = 0$$

Las condiciones de Neumann para los puntos de los demás lados se obtienen de manera similar. Los cuatro casos son

$$2u_{i,2} + u_{i-1,1} + u_{i+1,1} - 4u_{i,1} = 0 \text{ abajo } \square$$

$$2u_{i,m-1} + u_{i-1,m} + u_{i+1,m} - 4u_{i,m} = 0 \text{ arriba } \square$$

$$2u_{2,j} + u_{1,j-1} + u_{1,j+1} - 4u_{1,j} = 0 \text{ izquierdo } \square$$

$$2u_{n-1,j} + u_{n,j-1} + u_{n,j+1} - 4u_{n,j} = 0 \text{ derecha } \square$$

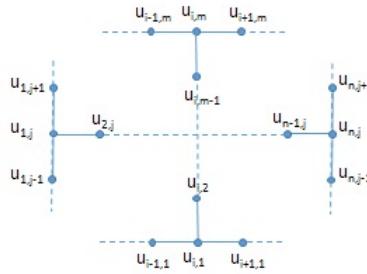


Figura 9.8. Bordes de condiciones Neumann

■ **Ejemplo 9.2 — Condiciones de borde de tipo Neumann y Dirichlet.** Determinar la solución aproximada de la ecuación de Laplace en el rectángulo donde $u(x, y)$ denota la temperatura en un punto (x, y) , los valores de frontera son:

- $u(x, 4) = 180$ para $0 \leq x \leq 4$ arriba (Dirichlet)
- $u(x, 0) = 0$ para $0 \leq x \leq 4$ abajo (Neumann)
- $u(0, y) = 80$ para $0 \leq y \leq 4$ izquierda (Dirichlet)
- $u(4, y) = 0$ para $0 \leq y \leq 4$ derecha (Dirichlet)

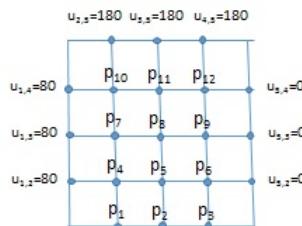


Figura 9.9. Condiciones de Neumann y Dirichlet

Solución

Sustituyendo en cada punto, se construye el sistema

$$\left(\begin{array}{cccccccccccc|c} -4p_1 & p_2 & 0 & 2p_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -80 \\ p_1 & -4p_2 & p_3 & 0 & 2p_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_2 & -4p_3 & 0 & 0 & 2p_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_1 & 0 & 0 & -4p_4 & p_5 & 0 & p_7 & 0 & 0 & 0 & 0 & 0 & -80 \\ 0 & p_2 & 0 & p_4 & -4p_5 & p_6 & 0 & p_8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & p_3 & 0 & p_5 & -4p_6 & 0 & 0 & p_9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_4 & 0 & 0 & -4p_7 & p_8 & 0 & p_{10} & 0 & 0 & -80 \\ 0 & 0 & 0 & 0 & p_5 & 0 & p_7 & -4p_8 & p_9 & 0 & p_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p_6 & 0 & p_8 & -4p_9 & 0 & 0 & p_{12} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_7 & 0 & 0 & -4p_{10} & p_{11} & 0 & -260 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_8 & 0 & p_{10} & -4p_{11} & p_{12} & 0 & -180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_9 & 0 & p_{11} & -4p_{12} & 0 & -180 \end{array} \right)$$

Resolviendo el sistema de ecuaciones lineales

Programa 9.2. Solución de Laplace con condiciones Neumann-Dirichlet

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     a=np.array([[-4, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0], \
6                 [1, -4, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0], \
7                 [0, 1, -4, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0], \
8                 [1, 0, 0, -4, 1, 0, 1, 0, 0, 0, 0, 0, 0], \
9                 [0, 1, 0, 1, -4, 1, 0, 1, 0, 0, 0, 0, 0], \
10                [0, 0, 1, 0, 1, -4, 0, 0, 1, 0, 0, 0, 0], \
11                [0, 0, 0, 1, 0, 0, -4, 1, 0, 1, 0, 0, 0], \
12                [0, 0, 0, 0, 1, 0, 1, -4, 1, 0, 1, 0, 0], \
13                [0, 0, 0, 0, 0, 1, 0, 1, -4, 0, 0, 1, 0], \
14                [0, 0, 0, 0, 0, 0, 1, 0, 0, -4, 1, 0, 0], \
15                [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, -4, 1, 0], \
16                [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, -4]])
17    b=np.array([-80,0,0,-80,0,0,-80,0,0,-260,-180,-180])
18    n,c=np.shape(a)
19    r=np.linalg.matrix_rank(a)
20    ab=np.c_[a,b]
21    ra=np.linalg.matrix_rank(ab)
22
23    print('rango(A)={} rango(Ab)={} n={}'.format(r,ra,n))
24
25    if (r==ra==n):
26        print('solucion unica')
27        x=np.linalg.solve(a, b)
28        z=x.reshape(4,3)
29        print(z)

```

```

30     X,Y=np.meshgrid(np.arange(1,4), np.arange(1,5))
31
32     fig, ax = plt.subplots()
33     CS = ax.contour(X, Y, z,10)
34     ax.clabel(CS, inline=1, fontsize=10)
35     fig.colorbar(CS, shrink=0.8, extend='both')
36     ax.set_title('Laplace con condiciones Newmann - Dirichlet')
37
38     fig.savefig("edp_laplace_newmann_dirichlet.pdf", bbox_inches='tight')
39
40     if (r==ra<n):
41         print('multiples soluciones')
42
43     if (r<ra):
44         print('sin solucion')
45
46 if __name__ == "__main__": main()

```

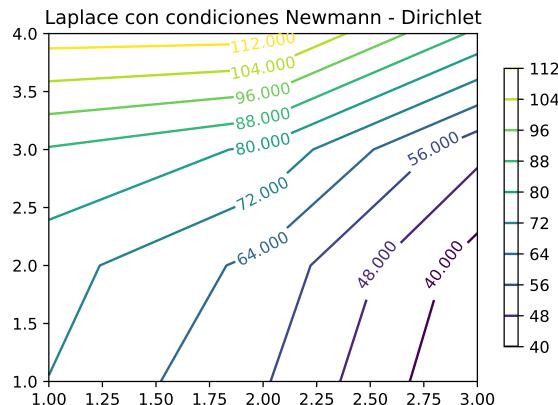


Figura 9.10. Ecuación de Laplace con condiciones Neumann-Dirichlet

```

rango(A)=12 rango(AB)=12 n=12 solucion unica
[[ 71.82182413   56.85429572   32.23419526]
 [ 75.2165004    61.68058174   36.04124267]
 [ 87.36359573   78.61028818   50.25019366]
 [ 115.62759433  115.14678158   86.34924381]]

```

9.2 Ecuaciones diferenciales parciales parabólicas

Este tipo de ecuaciones diferenciales permite resolver los denominados problemas de propagación que son problemas transitorios, donde la solución de la ecuación diferencial parcial es requerida sobre un dominio abierto, sujeta a condiciones iniciales y de frontera prescritas.

Un ejemplo de EDP parabólica es la ecuación de calor o difusión, usada para modelar el flujo de calor a lo largo de una varilla de longitud L (figura 9.11), en la cual la temperatura es uniforme en cada sección transversal, esta ecuación también es usada para estudiar la difusión del gas y, en general, problemas donde la solución cambia con el tiempo.

La ecuación de calor o difusión es



Figura 9.11. Flujo de calor a lo largo de una barra

La EDP que modela este fenómeno es:

$$\frac{\partial T}{\partial t}(x, t) - \alpha^2 \frac{\partial^2 T}{\partial x^2}(x, t) = 0 \quad (9.1)$$

$$0 < x < L, t > 0$$

Sujeto a las condiciones

$$u(x, 0) = f(x) \quad (9.2)$$

$$u(0, t) = T_1 \quad (9.3)$$

$$u(L, t) = T_2 \quad (9.4)$$

Donde α es la conductividad de material, T es la temperatura, que depende de x y de t , x es la variable independiente espacial, t es la variable independiente temporal, $f(x)$ es la distribución de calor en el momento inicial y T_1 y T_2 son las temperaturas constantes al inicio y al final de la barra.

En este caso, hay que considerar que la solución presenta cambios en el espacio y en el tiempo. La malla usada para la resolución por diferencias finitas de las EDP con dos variables independientes puede ser representada en la figura 9.12. Observe que la malla está abierta en la dimensión temporal t .

9.2.1 Método de diferencias progresivas (*forward differences*)

Se obtiene una expresión para las derivadas parciales usando las series de Taylor

$$u(x_i, t_{j+1}) = u(x_i, t_{j+k}) \approx u(x_i, t_j) + k u(x_i, t_j)$$

Por lo tanto, tenemos una expresión para:

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{k} \quad (9.5)$$

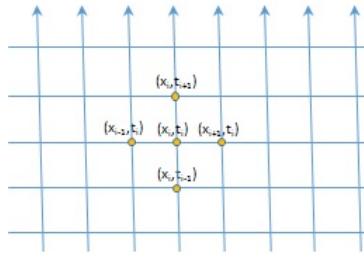


Figura 9.12. Malla para el flujo de calor en la variable dimensional x y el tiempo t

Así, también puede expresarse la segunda derivada parcial como:

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_j) \approx \frac{u(x_{i+1}, t_j) - 2u(x_i, t_j) + u(x_{i-1}, t_j)}{h^2} \quad (9.6)$$

Usando las ecuaciones anteriores podemos expresar la ecuación 9.1 como:

$$\frac{u_{i,j+1} - u_{i,j}}{k} - \alpha^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} = 0$$

Resolviendo la ecuación para $u_{i,j+1}$ se obtiene un sistema de ecuaciones:

$$u_{i,j+1} = (1 - 2\lambda)u_{i,j} + \lambda(u_{i+1,j} + u_{i-1,j}) \quad (9.7)$$

con $\lambda = \frac{\alpha^2 k}{h^2}$ para $i = 1, \dots, m-1$ y $j = 1, 2, \dots$

De las condiciones iniciales en 9.2 se tiene que

$$u_{i,0} = f(x_i), \forall i = 1, 2, \dots, m-1$$

$$u_{0,j} = u_{1,j} = 0 \quad \forall j = 0, 2, \dots$$

La solución se va obteniendo para cada paso de tiempo $j = 1, 2, \dots$ para $t = 0$ la solución $u(0)$ se conoce y es:

$$u^{(0)} = \begin{bmatrix} u_{1,0} \\ \vdots \\ u_{m-1,0} \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_{m-1}) \end{bmatrix}$$

La solución para el siguiente paso de tiempo está definida matricialmente por la ecuación 9.7

$$u^{(0)} = Au^{(j-1)}$$

Donde

$$A = \begin{bmatrix} 1 - 2\lambda & \lambda & 0 & \cdots & \cdots & 0 \\ \lambda & 1 - 2\lambda & \lambda & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & \lambda & 1 - 2\lambda & \lambda \\ 0 & \cdots & \cdots & 0 & \lambda & 1 - 2\lambda \end{bmatrix}$$

9.2.2 Método de diferencias regresivas (backward differences)

Esta modificación surge para evitar que la efectividad del método dependa de la condición $\lambda < 0.5$. La modificación consiste en expresar la ecuación ahora para resolver $u_{i,j-1}$. El cambio se realiza para la siguiente expresión:

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{u(x_i, t_j) - u(x_i, t_{j-1})}{k}$$

Usando esta expresión, ahora se tiene que

$$u_{i,j-1} = (1 + 2\lambda)u_{i,j} - \lambda(u_{i+1,j} + u_{i-1,j})$$

El sistema de ecuaciones a resolver será entonces

$$Au^{(j)} = u^{(j-1)}$$

Donde

$$A = \begin{bmatrix} 1+2\lambda & -\lambda & 0 & \cdots & \cdots & 0 \\ -\lambda & 1+2\lambda & -\lambda & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & -\lambda & 1+2\lambda & -\lambda \\ 0 & \cdots & \cdots & 0 & -\lambda & 1+2\lambda \end{bmatrix}$$

En este caso también resolvemos $u^{(j)}$ en función de $u^{(j-1)}$. Sin embargo, la matriz A es positiva definida y estrictamente diagonal, lo que asegura estabilidad en la solución.

9.2.3 Método de Crank-Nicholson

Para buscar un método cuyo error de truncamiento local sea de orden $O(k^2) + O(h^2)$ hay que usar una aproximación para $u_t(x, t)$ cuyo error sea de orden $O(k^2)$. En el método de Crank-Nicholson esto se consigue approximando la solución de la ecuación de calor en puntos que están entre las líneas de la cuadrícula. Específicamente, usamos la fórmula de diferencias centradas para aproximar:

$$u_t\left(x, t + \frac{k}{2}\right) = \frac{u(x, t+k) - u(x, t)}{k} + O(k^2)$$

Por otro lado, aproximamos $u_{xx}\left(x, t + \frac{k}{2}\right)$ promediando las aproximaciones a $u_{xx}(x, t)$ y $u_{xx}(x, t+k)$

$$u_{xx}\left(x, t + \frac{k}{2}\right) = \frac{u(x+h, t+k) - 2u(x, t+k) + u(x-h, t+k)}{h^2} + \frac{u(x+h, t) - 2u(x, t) + u(x-h, t)}{h^2} + O(k^2)$$

Trabajando de forma similar a como lo hicimos para obtener el método de diferencias progresivas, sustituimos estas aproximaciones en la ecuación diferencial y despreciando los términos $O(k^2)$ y $O(h^2)$ obtenemos:

$$-\lambda w_{i-1,j+1} + (1 + 2\lambda)w_{i,j+1} - \lambda w_{i+1,j+1} = \lambda w_{i-1,j} + (1 - 2\lambda)w_{i,j} + \lambda w_{i+1,j}$$

Para $i = 2, \dots, n-1$. Dado que los valores en la frontera son $w_{1,j} = w_{n,j} = 0$ para todo j , ahora se tiene una relación entre los vectores w_j y w_{j+1} que, como en los casos anteriores es de tipo lineal: $Aw_{j+1} = Bw_j$, donde A es la matriz tridiagonal, simétrica y de diagonal dominante

$$A = \begin{bmatrix} 1+2\lambda & -\lambda & 0 & \cdots & \cdots & 0 \\ -\lambda & 1+2\lambda & -\lambda & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & -\lambda & 1+2\lambda & -\lambda \\ 0 & \cdots & \cdots & 0 & -\lambda & 1+2\lambda \end{bmatrix}$$

y B es

$$B = \begin{bmatrix} 1-2\lambda & \lambda & 0 & \cdots & \cdots & 0 \\ \lambda & 1-2\lambda & \lambda & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & \lambda & 1-2\lambda & \lambda \\ 0 & \cdots & \cdots & 0 & \lambda & 1-2\lambda \end{bmatrix}$$

9.2.4 Método explícito

La ecuación de conducción del calor que estamos tratando requiere de dos aproximaciones. Para la derivada segunda respecto de la variable espacial x , podemos hacerla con una diferencia dividida centrada con una aproximación de segundo orden (ecuación 9.6).

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_j) \approx \frac{u(x_{i+1}, t_j) - 2u(x_i, t_j) + u(x_{i-1}, t_j)}{(\Delta x)^2}$$

y una diferencia dividida finita hacia adelante para aproximar a la derivada en el tiempo (ecuación 9.5).

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{\Delta t}$$

Sustituyendo en la ecuación 9.1 tenemos

$$\frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{\Delta t} - \alpha^2 \frac{u(x_{i+1}, t_j) - 2u(x_i, t_j) + u(x_{i-1}, t_j)}{(\Delta x)^2} = 0$$

Reacomodando tenemos

$$u(x_i, t_{j+1}) = u(x_i, t_j) + \left(\frac{\alpha^2 \Delta t}{(\Delta x)^2} \right) (u(x_{i+1}, t_j) - 2u(x_i, t_j) + u(x_{i-1}, t_j))$$

si hacemos

$$\lambda = \left(\frac{\alpha^2 \Delta t}{(\Delta x)^2} \right)$$

tenemos

$$u(x_i, t_{j+1}) = u(x_i, t_j) + \lambda (u(x_{i+1}, t_j) - 2u(x_i, t_j) + u(x_{i-1}, t_j)) \quad (9.8)$$

Esta ecuación, que puede ser escrita para todos los nodos interiores de la barra, proporciona un modo explícito para calcular los valores en cada nodo para un tiempo siguiente, con base en los valores actuales del nodo y sus vecinos.

Si las condiciones de contorno son del tipo Dirichlet, donde el valor de la función incógnita es conocido, la ecuación anterior no debe ser aplicada en los puntos de la frontera, puesto que allí no hay incógnitas.

Las condiciones de contorno o de frontera del tipo de Neumann (o condición natural) pueden ser incorporadas sin inconvenientes a las ecuaciones parabólicas, de la misma manera que con las elípticas. En el caso particular de la ecuación de conducción de calor unidimensional, deberán agregarse dos ecuaciones para caracterizar el balance de calor en los nodos extremos, para el nodo inicial tenemos:

$$u(x_0, t_{j+1}) = u(x_0, t_j) + \lambda (u(x_1, t_j) - 2u(x_0, t_j) + u(x_{-1}, t_j))$$

Donde el punto x_{-1} es exterior al dominio de análisis. Este punto puede escribirse en función de los interiores utilizando las condiciones de contorno que correspondan. En este caso:

$$q_x = -\alpha \rho C \frac{\partial T}{\partial x}$$

Utilizando una diferencia dividida finita centrada para aproximar a la derivada de T respecto de la variable espacial x

$$\frac{\partial T}{\partial x} = \frac{u(x_{i-1}, t_j) - u(x_{i+1}, t_j)}{2\Delta x}$$

Sustituyendo

$$q_x = -\alpha \rho C \left(\frac{u(x_{-1}, t_j) - u(x_1, t_j)}{2\Delta x} \right)$$

despejando $u(x_{-1}, t_j)$ obtenemos:

$$u(x_{-1}, t_j) = u(x_1, t_j) - \frac{2\Delta x q_x}{\alpha \rho C}$$

Por lo tanto, para calcular el primer punto

$$u(x_0, t_{j+1}) = u(x_0, t_j) + \lambda \left(u(x_1, t_j) - 2u(x_0, t_j) + u(x_{-1}, t_j) - \frac{2\Delta x q_x}{\alpha \rho C} \right)$$

Simplificando

$$u(x_0, t_{j+1}) = u(x_0, t_j) + 2\lambda \left(u(x_1, t_j) - u(x_0, t_j) - \frac{\Delta x q_x}{\alpha \rho C} \right)$$

■ **Ejemplo 9.3 — Conducción de calor unidimensional en el tiempo.** Calcular la distribución de temperatura de una barra larga y delgada que tiene una longitud de 10 cm.

El coeficiente de difusividad térmica es: $\alpha^2 = 0.835 \text{ cm}^2/\text{s}$.

Como condición de frontera tenemos que en los extremos de la barra la temperatura es constante todo el tiempo $T(0, t) = 100^\circ\text{C}$ y $T(10, t) = 50^\circ\text{C}$.

Como condición inicial tenemos que en el interior de la barra la temperatura para el tiempo $t = 0$ es:

$T(x, 0) = 0^\circ C$ para $0 < x < 10$. Si tomamos $\Delta x = 2\text{cm}$ y $\Delta t = 0.1\text{s}$ tendremos que:

$$\lambda = \left(\frac{\alpha^2 \Delta t}{(\Delta x)^2} \right) = \left(\frac{0.835 \times 0.1}{2^2} \right) = 0.020875$$

Usaremos la ecuación 9.8 para obtener la temperatura de la barra en cada Δx para cada incremento del tiempo Δt

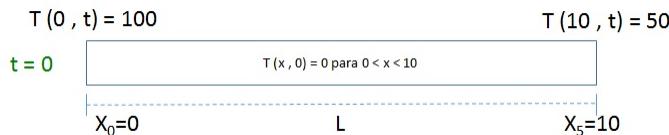


Figura 9.13. Condiciones iniciales de la barra

Solución

Cuadro 9.1. Cálculos por cada Δt y Δx

Tiempo t	Distancia x	Fórmula	Resultado
$t = 0.1$	$x = 2$	$u(x = 2, t = 0.1) = 0 + 0.020875(0 - 2(0) + 100)$	2.0875
	$x = 4$	$u(x = 4, t = 0.1) = 0 + 0.020875(0 - 2(0) + 0)$	0
	$x = 6$	$u(x = 6, t = 0.1) = 0 + 0.020875(0 - 2(0) + 0)$	0
	$x = 8$	$u(x = 8, t = 0.1) = 0 + 0.020875(50 - 2(0) + 0)$	1.0438
$t = 0.2$	$x = 2$	$u(x = 2, t = 0.2) = 2.0875 + 0.020875(0 - 2(2.0875) + 100)$	4.0878
	$x = 4$	$u(x = 4, t = 0.2) = 0 + 0.020875(0 - 2(0) + 2.0875)$	0.043577
	$x = 6$	$u(x = 6, t = 0.2) = 0 + 0.020875(1.0438 - 2(0) + 0)$	0.021788
	$x = 8$	$u(x = 8, t = 0.2) = 1.0438 + 0.020875(50 - 2(1.0438) + 0)$	2.0439

Programa 9.3. Temperatura de la barra en el tiempo, método explícito

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     L=10          #Longitud de la barra
6     dx=2          #delta x
7     dt=0.1        #delta t
8     alfa=0.835   #coeficiente de difusividad termica
9     lamda=alfa*dt/dx**2 #valor de lamda
10    Ti,Td=100,50      #valores de la frontera (izquierda,derecha)
11
12    n=200          #vector tiempo
13    m=L//dx        #vector dimension
14    u=np.zeros((m+1,n)) #matriz de resultados
15    u[0,:]=Ti       #se asignan los valores de la frontera
16    u[m,:]=Td
17

```

```

18     #calcula la temperatura de cada posicion en cada tiempo
19     for ti in range(n-1):
20         u[1:m,ti+1]=[u[xi,ti]+lamda*(u[xi-1,ti]-2*u[xi,ti]+u[xi+1,←
21             ti])\
22             for xi in np.arange(1,m)]
23
24     np.set_printoptions(precision=2)
25     print (u)
26     fig = plt.figure()
27     plt.plot(np.linspace(0,L,m+1),u[:,::10])
28     plt.title('Temperatura de la barra en el tiempo')
29     plt.xlabel('Distancia')
30     plt.ylabel('Temperatura')
31     fig.savefig("edp_barra.pdf", bbox_inches='tight')
32 if __name__ == "__main__": main()

```

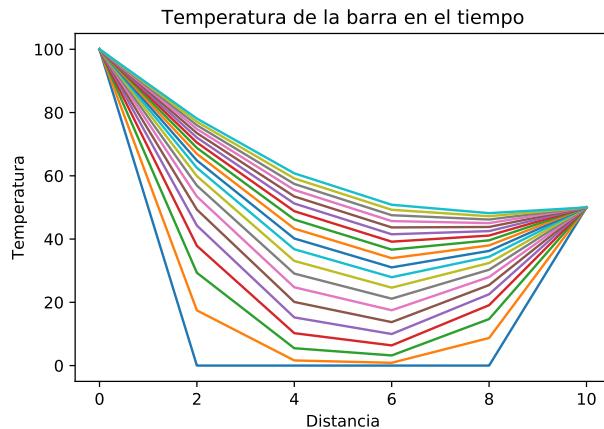


Figura 9.14. Temperatura de la barra en el tiempo

```

[[1.00e+02 1.00e+02 1.00e+02 ... , 1.00e+02 1.00e+02 1.00e+02]
 [0.00e+00 2.09e+00 4.09e+00 ... , 7.87e+01 7.88e+01 7.89e+01]
 [0.00e+00 0.00e+00 4.36e-02 ... , 6.18e+01 6.20e+01 6.21e+01]
 [0.00e+00 0.00e+00 2.18e-02 ... , 5.19e+01 5.20e+01 5.22e+01]
 [0.00e+00 1.04e+00 2.04e+00 ... , 4.88e+01 4.89e+01 4.90e+01]
 [5.00e+01 5.00e+01 5.00e+01 ... , 5.00e+01 5.00e+01 5.00e+01]]

```

9.2.5 Método implícito

La forma explícita aproxima la derivada espacial en el nivel de tiempo j , de modo que nos queda una ecuación con una sola incógnita $u_{i,j+1}$, que se despeja en forma explícita. En la forma implícita, la derivada espacial es aproximada en un nivel de tiempo posterior $j + 1$, de modo que quedan más de una incógnita en una misma ecuación, impidiendo la solución en forma sencilla como ocurre en el método explícito.

Entonces, al quedar una ecuación con varias incógnitas, el sistema completo de ecuaciones que se genera debe resolverse simultáneamente. Esto es posible porque junto con las condiciones de frontera, las formas implícitas dan como resultado un conjunto de ecuaciones lineales algebraicas con el mismo número de incógnitas. Entonces el método se reduce a la solución de un conjunto de ecuaciones simultáneas para cada instante de tiempo.

La ecuación de conducción del calor que estamos tratando requiere de dos aproximaciones. Para la segunda derivada con respecto de la variable espacial x , podemos obtenerla con una diferencia dividida centrada con una aproximación de segundo orden

$$\frac{\partial^2 T}{\partial x^2} = \frac{u(x_{i+1}, t_{j+1}) - 2u(x_i, t_j + 1) + u(x_{i-1}, t_j + 1)}{(\Delta x)^2}$$

Y la diferencia dividida finita hacia delante para aproximar a la derivada en el tiempo

$$\frac{\partial T}{\partial t} = \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{\Delta t}$$

Sustituyendo en la ecuación

$$k \frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$$

Se obtiene

$$k \frac{u(x_{i+1}, t_{j+1}) - 2u(x_i, t_{j+1}) + u(x_{i-1}, t_{j+1})}{(\Delta x)^2} = \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{\Delta t}$$

Re arreglando

$$-\lambda u(x_{i-1}, t_{j+1}) + (1 + 2\lambda)u(x_i, t_{j+1}) - \lambda u(x_{i+1}, t_{j+1}) = u(x_i, t_j)$$

Donde los términos en rojo son incógnitas que se deben calcular y

$$\lambda = \frac{k\Delta t}{(\Delta x)^2}$$

Esta ecuación se aplica a todos los nodos, excepto en el primero y el último, generando un sistema de ecuaciones lineales donde las **incógnitas** se deben determinar. El sistema de ecuaciones que se genera es tridiagonal.

■ **Ejemplo 9.4 — Conducción de calor unidimensional en el tiempo.** Resolver el ejemplo 9.3 con el método implícito simple. Hay que recordar que la condición de frontera en los extremos son temperaturas conocidas que se mantienen constantes en todo intervalo de tiempo. Por lo tanto, la ecuación debe aplicarse, en este caso, sólo a los puntos interiores del dominio.

$$-\lambda u(x_{i-1}, t_{j+1}) + (1 + 2\lambda)u(x_i, t_{j+1}) - \lambda u(x_{i+1}, t_{j+1}) = u(x_i, t_j)$$

Solución

Observamos que la matriz de coeficientes se construye

$$\begin{pmatrix} (1+2\lambda) & -\lambda & 0 & 0 \\ -\lambda & (1+2\lambda) & -\lambda & 0 \\ 0 & -\lambda & (1+2\lambda) & -\lambda \\ 0 & 0 & -\lambda & (1+2\lambda) \end{pmatrix}$$

El sistema es

$$\begin{pmatrix} 1.04175 & -0.020875 & 0 & 0 \\ -0.020875 & 1.04175 & -0.020875 & 0 \\ 0 & -0.020875 & 1.04175 & -0.020875 \\ 0 & 0 & -0.020875 & 1.04175 \end{pmatrix} \begin{pmatrix} u(x_1, t_1) \\ u(x_2, t_1) \\ u(x_3, t_1) \\ u(x_4, t_1) \end{pmatrix} = \begin{pmatrix} 2.0875 \\ 0 \\ 0 \\ 1.04375 \end{pmatrix}$$

Resolviendo el sistema

$$\begin{pmatrix} u(x_1, t_1) \\ u(x_2, t_1) \\ u(x_3, t_1) \\ u(x_4, t_1) \end{pmatrix} = \begin{pmatrix} 2.0047 \\ 0.0406 \\ 0.0209 \\ 1.0023 \end{pmatrix}$$

que son las temperaturas para $u(x_i, t_1 = 0.1)$

Programa 9.4. Temperatura de la barra en el tiempo, método implícito

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     L=10          #Longitud de la barra
6     dx=2          #delta x
7     dt=0.1        #delta t
8     alfa=0.835   #coeficiente de difusividad termica
9     lamda=alfa*dt/dx**2 #valor de lamda
10    T0 = 0         #temperatura inicial de la barra
11    Ti,Td=100,50   #valores de la frontera (izquierda,derecha)
12
13    n=200          #vector tiempo
14    m=L//dx-1      #vector dimension
15    u=np.zeros((m+2,n)) #matriz de resultados
16    u[0,:]=Ti
17    u[m+1,:]=Td
18    u[1:m,0]=T0
19    #calcula la temperatura de cada posicion en cada tiempo
20    a=np.zeros((m,m)) #matriz del sistema
21    b=np.zeros(m)
22
23    #matriz de coeficientes es la misma
24    a[0,0:2]=np.array([1+2*lamda,-lamda])
25    for i in range(m-2):
26        a[i+1,i:i+3]=np.array([-lamda,1+2*lamda,-lamda])
27        a[i+2,i+1:i+3]=np.array([-lamda,1+2*lamda])
```

```

28     #print(a)
29
30     # ciclo del tiempo
31     for j in range(n-1):
32         b[0]=u[1,j]+lamda*u[0,j+1]
33         #ciclo de la dimension espacial
34         for i in range(m-2):
35             b[i+1]=u[i+2,j]
36
37         b[m-1]=u[m,j]+lamda*u[m+1,j+1]
38         #calculo de las temperaturas (solucion del sistema)
39         sol=np.linalg.solve(a, b)
40         #asigna resultados
41         u[1:m+1,j+1]=sol
42
43     np.set_printoptions(precision=4)
44     print(u)
45
46     fig = plt.figure()
47     plt.plot(np.linspace(0,L,m+2),u[:,::10])
48     plt.title('Temperatura de la barra en el tiempo')
49     plt.xlabel('Distancia')
50     plt.ylabel('Temperatura')
51     fig.savefig("edp_barra2.pdf", bbox_inches='tight')
52
53 if __name__ == "__main__": main()

```

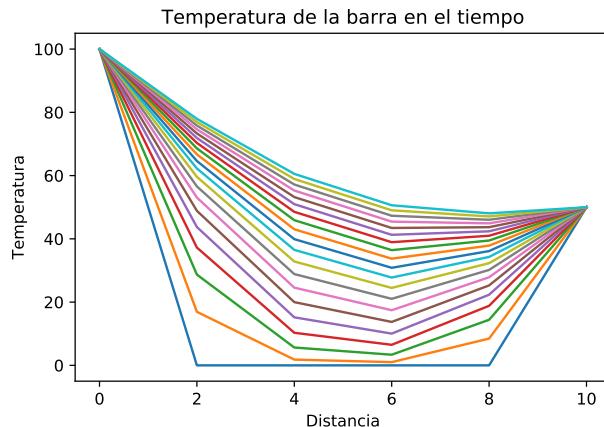


Figura 9.15. Temperatura de la barra en el tiempo, método implícito.

```

[[1.0000e+02 1.0000e+02 1.0000e+02 ... , 1.0000e+02 1.0000e+02 ←
 1.0000e+02]
 [0.0000e+00 2.0047e+00 3.9305e+00 ... , 7.8599e+01 7.8690e+01 ←
 7.8780e+01]
 [0.0000e+00 4.0589e-02 1.1896e-01 ... , 6.1602e+01 6.1748e+01 ←
 6.1893e+01]

```

```
[0.0000e+00 2.0899e-02 6.1827e-02 ... , 5.1662e+01 5.1806e+01 ←
 5.1950e+01]
[0.0000e+00 1.0023e+00 1.9653e+00 ... , 4.8695e+01 4.8784e+01 ←
 4.8872e+01]
[5.0000e+01 5.0000e+01 5.0000e+01 ... , 5.0000e+01 5.0000e+01 ←
 5.0000e+01]]
```

■

9.2.6 Método implícito de Crank-Nicolson

Desarrollando las aproximaciones por diferencias en el punto medio del incremento en el tiempo, la primera derivada temporal puede ser aproximada en $t_{l+1/2}$ por:

$$\frac{\partial T}{\partial t} = \frac{1}{2} \left(\frac{u_{i,j+1} - u_{j+1/2}}{\Delta t/2} + \frac{u_{i,j+1} - u_j}{\Delta t/2} \right) = \frac{u_{i,j+1} - u_{i,j}}{\Delta t}$$

La segunda derivada en el espacio puede ser determinada en el punto medio al promediar las aproximaciones por diferencias al inicio t_j y al final t_{j+1} del intervalo del incremento del tiempo.

$$\frac{\partial^2 T}{\partial x^2} = \frac{1}{2} \left(\frac{u_{i+1,j} - 2u_{i,j} - u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{(\Delta x)^2} \right)$$

Sustituyendo en la ecuación de conducción de calor queda

$$\frac{k}{2} \left(\frac{u_{i+1,j} - 2u_{i,j} - u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{(\Delta x)^2} \right) = \frac{u_{i,j+1} - u_{i,j}}{\Delta t}$$

Reordenando:

$$\frac{\Delta t k}{2(\Delta x)^2} ((u_{i+1,j} - 2u_{i,j} - u_{i-1,j}) + (u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1})) = u_{i,j+1} - u_{i,j}$$

Si hacemos

$$\lambda = \frac{\Delta t k}{(\Delta x)^2}$$

El valor λ es importante porque determina la estabilidad del método, un valor de $\lambda \leq 0.5$ es recomendable para que sea estable, de lo contrario, se ajusta el valor de Δt para cumplir la condición.

$$\frac{\lambda}{2} ((u_{i+1,j} - 2u_{i,j} - u_{i-1,j}) + (u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1})) = u_{i,j+1} - u_{i,j}$$

$$\frac{\lambda}{2} (u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}) - u_{i,j+1} = -u_{i,j} - \frac{\lambda}{2} (u_{i+1,j} - 2u_{i,j} - u_{i-1,j})$$

$$\lambda (u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}) - 2u_{i,j+1} = -2u_{i,j} - \lambda (u_{i+1,j} - 2u_{i,j} - u_{i-1,j})$$

$$\lambda u_{i+1,j+1} - (2\lambda + 2)u_{i,j+1} + \lambda u_{i-1,j+1} = -\lambda u_{i+1,j} - (2\lambda - 2)u_{i,j} - \lambda u_{i-1,j} \quad (9.9)$$

■ **Ejemplo 9.5 — Solución implícita de Crank-Nicolson.** Resolver el ejemplo 9.3 con el método implícito de Crank-Nicolson. Hay que recordar que la condición de borde en los extremos son temperaturas conocidas que se mantienen constantes en todo intervalo de tiempo. Por lo tanto, la ecuación debe aplicarse, en este caso, sólo a los puntos interiores del dominio, aplicando la ecuación (9.9) de Crank-Nicolson.

$$-\lambda u_{i-1,j+1} + (2\lambda + 2)u_{i,j+1} - \lambda u_{i+1,j+1} = \lambda u_{i-1,j} + (2\lambda - 2)u_{i,j} + \lambda u_{i+1,j}$$

$$\lambda = \frac{\Delta t k}{(\Delta x)^2} = \frac{0.1 k}{(2)^2} = 0.020875$$

Solución

Cuadro 9.2. Cálculos por cada Δt y Δx

Tiempo t	Distancia x	Fórmula
$t = 0$	$x = 2$	$\begin{aligned} -\lambda u_{0,1} + 2(1 + \lambda)u_{1,1} - \lambda u_{2,1} &= \lambda u_{0,0} + 2(\lambda - 1)u_{1,0} + \lambda u_{2,0} \\ -0.020875(100) + 2.04175u_{1,1} - 0.020875u_{2,1} &= 0.020875(100) + 1.95825(0) + 0.020875(0) \\ 2.04175u_{1,1} - 0.020875u_{2,1} &= 4.175 \end{aligned}$
	$x = 4$	$\begin{aligned} -\lambda u_{1,1} + 2(\lambda + 1)u_{2,1} - \lambda u_{3,1} &= \lambda u_{1,0} + 2(\lambda + 1)u_{2,0} + \lambda u_{3,0} \\ -0.020875u_{1,1} + 2(1.020875)u_{2,1} - 0.020875u_{3,1} &= 0.020875(0) + 2(-0.979125)(0) + 0.020875(0) \\ -0.020875u_{1,1} + 2.04175u_{2,1} - 0.020875u_{3,1} &= 0 \end{aligned}$
	$x = 6$	$\begin{aligned} -\lambda u_{2,1} + 2(\lambda + 1)u_{3,1} - \lambda u_{4,1} &= \lambda u_{2,0} + 2(\lambda + 1)u_{3,0} + \lambda u_{4,0} \\ -0.020875u_{2,1} + 2(1.020875)u_{3,1} - 0.020875u_{4,1} &= 0.020875(0) + 2(-0.979125)(0) + 0.020875(0) \\ -0.020875u_{2,1} + 2.04175u_{3,1} - 0.020875u_{4,1} &= 0 \end{aligned}$
	$x = 8$	$\begin{aligned} -\lambda u_{3,1} + 2(\lambda + 1)u_{4,1} - \lambda u_{5,1} &= \lambda u_{3,0} + 2(\lambda + 1)u_{4,0} + \lambda u_{5,0} \\ -0.020875u_{3,1} + 2(1.020875)u_{4,1} + 0.020875(50) &= 0.020875(0) + 2(-0.979125)(0) + 0.020875(0) \\ -0.020875u_{3,1} + 2.04175u_{4,1} &= 2.0875 \end{aligned}$

El sistema es

$$\begin{pmatrix} 2.04175 & -0.020875 & 0 & 0 \\ -0.020875 & 2.04175 & -0.020875 & 0 \\ 0 & -0.020875 & 2.04175 & -0.020875 \\ 0 & 0 & 0.020875 & 2.04175 \end{pmatrix} \begin{pmatrix} u(x_1, t_1) \\ u(x_2, t_1) \\ u(x_3, t_1) \\ u(x_4, t_1) \end{pmatrix} = \begin{pmatrix} 4.175 \\ 0 \\ 0 \\ 2.0875 \end{pmatrix}$$

Resolviendo el sistema

$$\begin{pmatrix} u(x_1, t_1) \\ u(x_2, t_1) \\ u(x_3, t_1) \\ u(x_4, t_1) \end{pmatrix} = \begin{pmatrix} 2.0450 \\ 0.0210 \\ 0.0107 \\ 1.0225 \end{pmatrix}$$

que son las temperaturas para $u(x_i, t_1 = 0.1)$

Programa 9.5. Temperatura de la barra en el tiempo, método implícito de Crank-Nicolson

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def main():
5     L = 10
6     dx = 2
7     dt = 0.1
8     alfa = 0.835
9     lamda = alfa * dt / dx ** 2
10    T0 = 0
11    Tx0 = 100
12    Tx10 = 50
13
14    n = 40
15    m = L // dx - 1
16    u = np.zeros((m + 2, n))
17    u[0,] = Tx0
18    u[m + 1,] = Tx10
19    u[1:m, 0] = T0
20
21    b = np.zeros(m)
22    # matriz tri diagonal
23    # matriz de coeficientes es la misma
24    A = np.diag([-lamda] * (m - 1), -1) + np.diag([2 * (lamda + 1) * m] * m) + np.diag([-lamda] * (m - 1), 1)
25
26    # ciclo del tiempo
27    for j in range(1, n):
28        for i in range(m):
29            b[i] = lamda * u[i, j - 1] + 2 * (lamda + 1) * u[i + 1, j - 1] + lamda * u[i + 2, j - 1]
30
31    b[0] = b[0] + lamda * u[0, j]
32    b[-1] = b[-1] + lamda * u[-1, j]
33
34    # solución del sistema
35    sol = np.linalg.solve(A, b)
36
37    # asigna los resultados
38    u[1:-1, j] = sol
39
40    # muestra los primeros 5 resultados
41    print(u[:, :5].round(4))
42
43    fig = plt.figure()
44    plt.plot(np.linspace(0, L, m+2), u[:, :5])
45    plt.title('Temperatura de la barra en el tiempo')
46    plt.xlabel('Distancia')
47    plt.ylabel('Temperatura')
48    plt.show()
```

```

49     fig.savefig("edp_barra3.pdf", bbox_inches='tight')
50
51 if __name__ == "__main__": main()

```

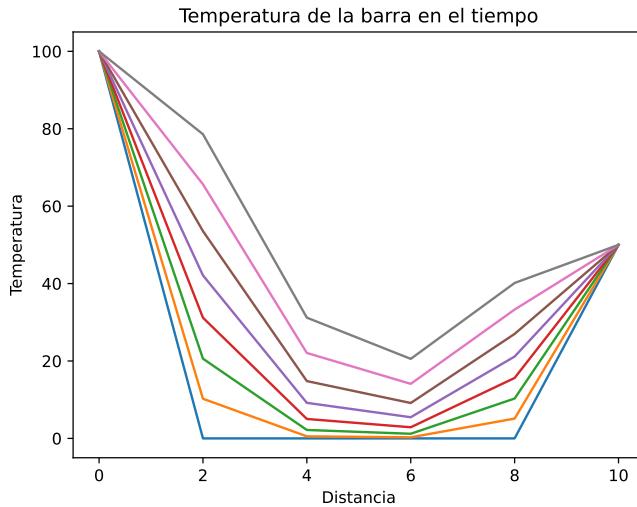


Figura 9.16. Temperatura de la barra en el tiempo, método implícito de Crank-Nicolson.

```

[[1.0000e+02 1.0000e+02 1.0000e+02 1.0000e+02 1.0000e+02]
[0.0000e+00 2.0450e+00 4.0909e+00 6.1385e+00 8.1888e+00]
[0.0000e+00 2.1000e-02 8.4300e-02 1.9030e-01 3.3960e-01]
[0.0000e+00 1.0700e-02 4.3100e-02 9.8200e-02 1.7690e-01]
[0.0000e+00 1.0225e+00 2.0455e+00 3.0693e+00 4.0945e+00]
[5.0000e+01 5.0000e+01 5.0000e+01 5.0000e+01 5.0000e+01]]

```

9.3 Ecuaciones diferenciales parciales hiperbólicas

Un ejemplo de EDP hiperbólica es la ecuación de onda, pero con la distinción de que aparece una segunda derivada respecto del tiempo. En consecuencia, la solución consiste en distintos estados característicos con los cuales oscila el sistema. Es el caso de problemas de vibraciones, ondas de un fluido, transmisión de señales acústicas y eléctricas.

La siguiente figura muestra un pulso que viaja a través de una cuerda con sus extremos fijos

La EDP que modela este fenómeno es

$$\frac{\partial u}{\partial t}(x, t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x, t) = 0 \quad (9.10)$$

$$0 < x < L, t > 0$$

Sujeto a las condiciones

$$u(0, t) = u(L, t) = 0 \quad (9.11)$$

$$u(x, 0) = f(x) \quad (9.12)$$

$$\frac{\partial u}{\partial t}(x, 0) = g(x) \quad (9.13)$$

$$(9.14)$$

Donde α es una constante equivalente a la velocidad de propagación de la onda.

Es necesario seleccionar un entero $m > 0$ y el tamaño de avance $k > 0$. Definiendo entonces los pasos como:

$$x_i = ih$$

$$t_j = jk$$

Al igual que en el caso anterior, se obtiene una expresión para las derivadas parciales usando las series de Taylor.

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) \approx \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{k^2}$$

Usando estas expresiones e ignorando el término del error asociado

$$\frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{k^2} - \alpha^2 \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} = 0$$

Haciendo un pequeño cambio de notación se tiene

$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} - \alpha^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} = 0$$

Si se hace $\lambda = \alpha \frac{k}{h}$ se puede escribir la ecuación anterior de la forma

$$u_{i,j+1} - 2u_{i,j} + u_{i,j-1} - \lambda^2 u_{i+1,j} + 2\lambda^2 u_{i,j} - \lambda^2 u_{i-1,j} = 0$$

Resolviendo la ecuación para $u_{i,j+1}$ se obtiene un sistema de ecuaciones

$$u_{i,j+1} = 2(1 - \lambda^2)u_{i,j} + \lambda^2(u_{i+1,j} + u_{i-1,j}) - u_{i,j-1}$$

Ecuación aplicable para los $i = 1, 2, \dots, m-1$; $j = 1, 2, \dots$ Y las condiciones de frontera dan:

$$u_{0,j} = u_{m,j} = 0$$

$$j = 1, 2, 3, \dots$$

Y la condición inicial implica que

$$u_{i,0} = f(x_i)$$

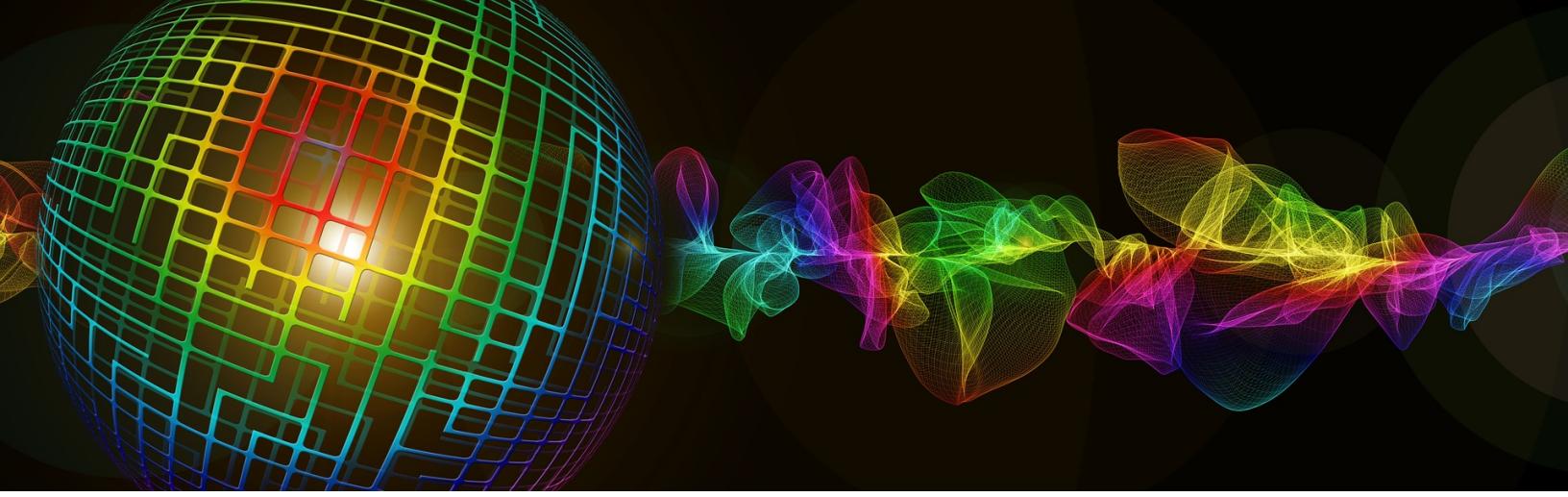
$$i = 1, 2, 3, \dots, m-1$$

El sistema de ecuaciones a resolver será:

$$u^{(j+1)} = Au^{(j)} - u^{(j-1)}$$

Donde la matriz A tiene la siguiente forma:

$$A = \begin{bmatrix} 2(1-\lambda^2) & \lambda^2 & 0 & \cdots & \cdots & 0 \\ \lambda^2 & 2(1-\lambda^2) & \lambda^2 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & \lambda^2 & 2(1-\lambda^2) & \lambda^2 \\ 0 & \cdots & \cdots & 0 & \lambda^2 & 2(1-\lambda^2) \end{bmatrix}$$



Bibliografía

- [1] Adidharma, H. and Temyanko, V. (2007). *Mathcad for chemical engineers*. Trafford Publishing.
- [2] Chapra, S. (2015). *Métodos numéricos para ingenieros*. McGraw-Hill, México, D.F.
- [3] Nieves Hurtado, A. and Domínguez Sánchez, F. (2002). *Métodos numéricos: Aplicados a la ingeniería*. Grupo Editorial Patria.

Métodos numéricos usando Python. Con aplicaciones a la Ingeniería Química es una obra editada por la Facultad de Química.

Se utilizaron en la composición las familias de las fuentes Ninbus Roman No9 L, URW Gotic L, F256 y DejaVuSans.

Publicación autorizada por el Comité Editorial de la Facultad de Química.

Marzo de 2022

Los métodos numéricos son herramientas muy poderosas que permiten resolver problemas que involucran ecuaciones complicadas, en la mayoría de los métodos se hacen aproximaciones hasta llegar a la solución, para ello se requiere ejecutar una serie de tareas de manera repetitiva, es aquí donde los lenguajes de programación son de mucha utilidad ya que cuentan con estructuras de decisión y repetición. Se debe seleccionar un lenguaje sencillo de aprender y, si es posible, que tenga una colección de funciones pre construidas que reduzca el esfuerzo de programación. Python representa una opción que cuenta con una extensa biblioteca de funciones para ciencias e ingeniería, por lo tanto, es ideal para programar los métodos numéricos, y es la combinación perfecta para ser utilizados por los ingenieros.

El presente libro hace una combinación de los métodos numéricos, el lenguaje Python y su aplicación a la Ingeniería Química, permitiendo al estudiante resolver problemas de una manera sencilla y con poco esfuerzo.

