Interpolación numérica y sus aplicaciones en la estadística

Universidad de Sevilla

Facultad de Matemáticas

Departamento de Análisis numérico

Trabajo Fin de Grado

Grado en Estadística

Curso 2023-2024

Dirigido por :
María Anguiano Moreno

Fdo : *Luis Carlos Ocaña Hoeber*
Sevilla, 23 de mayo de 2024

Índice general

Agradecimientos II
Resumen III
Abstract III
Índice de Figuras 1

1. Introducción **2**

2. Interpolación de Lagrange **3**
2.1. El error de interpolación 7
2.2. Aproximación de datos y método de Neville 12
2.3. Fórmula de interpolación de Newton 14
2.4. Minimización del error y polinomio de Chebychev 19

3. Interpolación mediante funciones spline **23**
3.1. Interpolación de spline cúbicas 24
3.2. Convergencia en la interpolación por funciones spline 30

4. Normal tabulada **33**

5. Regresión spline **39**

A. Apéndice: Funciones python **48**
A.1. Clase “Interpolacion Lagrange” 48
A.2. Función “Cota error” 49
A.3. Función “Área entre dos funciones” 50
A.4. Función “Interpolación Neville” 50
A.5. Función “Diferencias Newton” 50
A.6. Función “Puntos Chebychev” 51
A.7. Función “erf” 51
A.8. Función “norm_acum” 51

Bibliografía **52**

Agradecimientos

Escrito colocado al comienzo de una obra en el que se hacen comentarios sobre la obra o su autor, o se introduce en su lectura; a menudo está realizado por una persona distinta del autor.

También se podrían incluir aquí los agradecimientos.

Resumen

El presente trabajo se centra en analizar y comparar diversas técnicas de interpolación numérica desde una perspectiva estadística y de ciencia de datos. Se revisan conceptos fundamentales como la interpolación de Lagrange, métodos de spline y regresión spline, destacando su aplicación en la estimación precisa de datos entre observaciones discretas. La implementación práctica de estos métodos en un entorno computacional facilita la reproducibilidad y verificación de los resultados, contribuyendo al entendimiento y uso efectivo de la interpolación numérica en contextos estadísticos.

Abstract

This present work focuses on analyzing and comparing various numerical interpolation techniques from a statistical and data science perspective. Fundamental concepts such as Lagrange interpolation, spline methods, and regression spline are reviewed, highlighting their application in accurately estimating data between discrete observations. The practical implementation of these methods in a computational environment enhances reproducibility and result verification, contributing to the understanding and effective use of numerical interpolation in statistical contexts.

Índice de figuras

2.1. Polinomio de interpolación $P_3(x) = 2x - 3$ 5
2.2. Polinomio de interpolación con clase 'Lagrange' 6
2.3. Intersección de funciones y polinomio de Lagrange 7
2.4. Error de interpolación 10
2.5. Fenómeno de Runge 11
2.6. Polinomios de Chebychev 20
2.7. Evitando el fenómeno de Runge 22

3.1. Comparación de splines 24
3.2. Resultado spline cúbico 30
3.3. Logo Cicar 32
3.4. Logo Cicar con splines cúbicos 32

4.1. Normal estándar 33

5.1. Rendimiento del Modelo Spline en función del Número de Nodos 44
5.2. Spline cúbico con intervalo de confianza 45
5.3. Regresión lineal con Intervalo de Confianza 46

Capítulo 1

Introducción

En el campo de la estadística y la ciencia de datos, la interpolación numérica desempeña un papel fundamental al permitir la estimación y predicción de valores entre observaciones discretas. La capacidad de interpolar datos de manera precisa y eficiente es esencial para diversas aplicaciones estadísticas, como la imputación de valores faltantes, la estimación de funciones de densidad de probabilidad y la generación de muestras sintéticas.

El presente Trabajo de Fin de Grado se centra en explorar y analizar diferentes técnicas de interpolación numérica desde una perspectiva estadística y de ciencia de datos. El objetivo principal es analizar y comparar distintos métodos de interpolación, evaluando su precisión y desempeño en la aproximación de datos.

Una parte crucial de este estudio es la implementación práctica de los métodos de interpolación en un entorno computacional. El código desarrollado para este propósito se encuentra disponible en el repositorio público de GitHub, bajo el nombre de usuario luiocahoe. Este repositorio no solo proporciona acceso al código fuente utilizado en el estudio, sino que también permite la reproducibilidad y verificación de los resultados obtenidos.

El trabajo está estructurado de la siguiente manera: en primer lugar, se revisarán los fundamentos teóricos de la interpolación de Lagrange, incluyendo aspectos como el error de interpolación, el método de Neville y la fórmula de interpolación de Newton. También se abordará la minimización del error utilizando el polinomio de Chebychev. Posteriormente, se estudiará la interpolación mediante funciones spline, enfocándose en las spline cúbicas y su convergencia en contextos estadísticos. Además, se analizará la importancia y aplicaciones de las tablas de la distribución normal estándar en estadística. Finalmente, se introducirá la regresión spline como una técnica avanzada para ajustar modelos de regresión a datos estadísticos.

Cada sección combinará una revisión teórica con aplicaciones prácticas, incluyendo implementaciones computacionales y discusiones sobre los resultados obtenidos.

Capítulo 2

Interpolación de Lagrange

El contenido teórico de este cápitulo ha sido obtenido de [2] y [17]

La interpolación de Lagrange es un método utilizado en matemáticas y análisis numérico para construir un polinomio que pasa a través de un conjunto dado de puntos de datos. Fue desarrollada por el matemático francés Joseph-Louis Lagrange en el siglo XVIII. Este método es particularmente útil cuando se desea aproximar una función desconocida basándose en un conjunto limitado de datos discretos.

Notación 2.1 Denotaremos por

$$\mathbb{R}[x] = \left\{P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0, a_i \in \mathbb{R}, n \in \mathbb{N} \cup \{0\}\right\}$$

al conjunto de polinomios reales en la variable x, por

$$\mathcal{P}_n = \{a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{R}[x]\}, n \in \mathbb{N} \cup \{0\}$$

al conjunto de polinomios reales de grado menor o igual que n y por ∂P al *grado* del polinomio P.

Este problema consiste en lo siguiente:

Teorema 2.1 Dada la función $f : [a, b] \longrightarrow \mathbb{R}$ y $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ con $x_i \neq x_j$ si $i \neq j$ (donde no se supone que estos puntos sean equidistantes ni tampoco que estén enunciados en su orden natural) existe un único polinomio $P_n \in \mathcal{P}_n$ que verifica:

$$P(x_i) = f(xi)$$

para $i = 0, 1, \ldots, n$. Además, este polinomio viene dado por:

$$P_n(x) = \sum_{i=0}^{n} f(x_i)L_i(x) \tag{2.1}$$

donde, para cada $i \in \{0, 1, \ldots, n\}$,

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{2.2}$$

Definición 2.1 El polinomio P_n dado en (2.1) se denomina *polinomio de interpolación de Lagrange* relativo a la función f (o, simplemente, *polinomio de interpolación* de f) en los puntos $\{x_0, x_1, \ldots, x_n\}$ y los polinomios $L_i(x)$ dados en (2.2) son los *polinomios básicos de interpolación de Lagrange*.

Estos *polinomios básicos de interpolación de Lagrange* tienen la propiedad de que $L_i(x_j) = \delta_{ij}$, siendo δ_{ij} la delta de Kronecker, que está definida por

$$\delta_{ij} \equiv \begin{cases} 0 & \text{si } i \neq j \\ 1 & \text{si } i = j \end{cases} \tag{2.3}$$

Demostración 2.1

1. **Existencia:** teniendo en cuenta que para cada $i \in \{0, 1, \ldots, n\}$

$$L_i(x) = \frac{x - x_0}{x_i - x_0} \frac{x - x_1}{x_i - x_1} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n} \tag{2.4}$$

es inmediato comprobar con la propiedad (2.3)

$$\begin{cases} L_i \in \mathcal{P}_n,\ i = 0, 1, \ldots, n & \Rightarrow \quad P_n \in \mathcal{P}_n \\ L_i(x_j) = \delta_{ij} & \Rightarrow \quad P_n(x_j) = \sum_{i=0}^{n} f(x_j) L_i(x_j) = \sum_{i=0}^{n} f(x_j)\delta_{ij} = f(x_j) \end{cases}$$

2. **Unicidad:** supongamos que existen dos polinomios $P_n, Q_n \in P_n$ tales que

$$P_n(x_i) = f(x_i) = Q_n(x_i)$$

para $i = 0, 1, \ldots, n$. De esta forma, el polinomio $D_n = P_n - Q_n$ verifica $D_n \in \mathcal{P}_n$ y, para cada $i \in \{0, 1, \ldots, n\}$,

$$D_n(x_i) = P_n(x_i) - Q_n(x_i) = 0.$$

Es decir, D_n es un polinomio de grado menor o igual que n con $n + 1$ raíces; consecuentemente por el teorema Fundamental del Álgebra, $D_n \equiv 0$ de donde se concluye que $P_n \equiv Q_n$.

□

Observación 2.1 Al interpolar una función f en $n+1$ puntos puede que nuestro polinomio de interpolación de Lagrange no sea de grado n, sino inferior. Veamos un ejemplo:

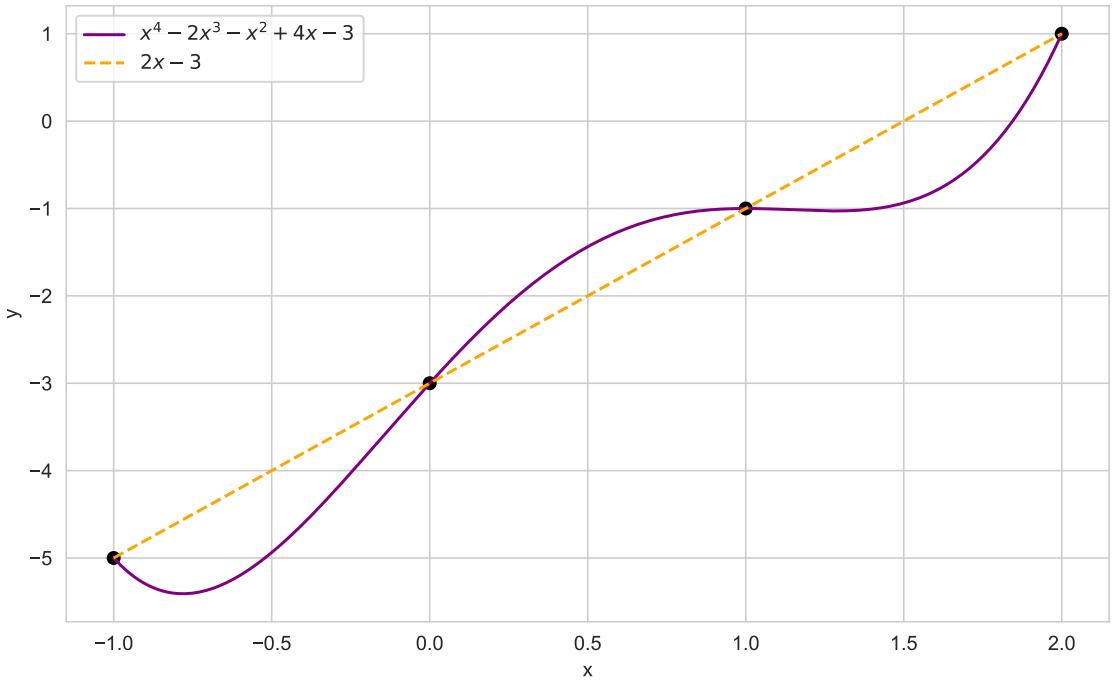

Figura 2.1: Polinomio de interpolación $P_3(x) = 2x - 3$

Observación 2.2 Calcular el polinomio de interpolación a partir de la fórmula (2.1) no es particularmente difícil, pero puede ser tedioso, especialmente a medida que aumenta el número de puntos de interpolación. Además, si una vez calculado este polinomio en $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ queremos añadir un nuevo punto x_{n+1} distinto de los anteriores, debemos repetir todo el proceso, dado que cambian todos los polinomios básicos.

Este método es fácil de implementar y entender, y proporciona una aproximación polinómica suave que pasa exactamente por los puntos de datos. Sin embargo, puede ser sensible a la distribución de los puntos de datos y al número de puntos utilizados. A medida que se incrementa el número de puntos de datos utilizados en la interpolación, el grado del polinomio resultante también puede aumentar, lo que se pueder traducir en una mayor oscilación entre los puntos de datos. Como consecuencia, una interpolación de alto grado puede ofrecer una mala predicción.

Tomando de inspiración [18] creamos una clase en Python para que nos calcule un polinomio de Lagrange y obtener un gráfico.

El código de la función se encuentra en el Apéndice A, bajo el nombre “Clase Interpolacion Lagrange”.

Veamos su funcionamiento:

```
xi = np.array([0, 0.2, 0.3, 0.4, 0.6])
fi = np.array([1, 1.6, 1.7, 2.0, 3])
lagrange = InterpolacionLagrange(xi, fi)
```

Para que calcule el polinomio debemos usar el método `calcular_polinomio`:

```
lagrange.calcular_polinomio()
```

Y para obtener el polinomio `imprimir_polinomio`:

```
lagrange.imprimir_polinomio()
```

```
## -83.3333333333335*x**4 + 116.666666666667*x**3 - 49.1666666666667*x**2
## + 8.83333333333334*x + 1.0
```

Es decir:

$$P_4(x) \approx -83.333x^4 + 116.667x^3 - 49.167x^2 + 8.833x + 1$$

Es importante destacar que los cálculos realizados por el ordenador pueden verse afectados por errores de redondeo debido a la representación finita de números en su sistema de punto flotante.

Para que nos muestre el gráfico donde aparece tanto el polinomio de lagrange como los puntos usados debemos usar el método `graficar`:

```
lagrange.graficar()
```

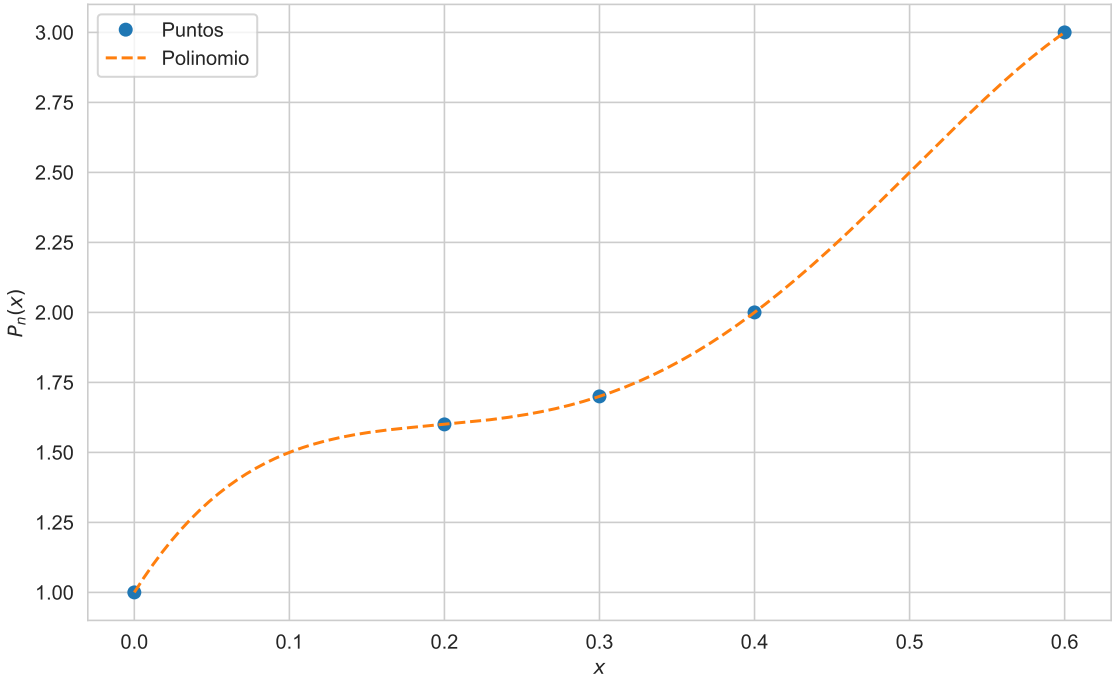

Figura 2.2: Polinomio de interpolación con clase 'Lagrange'

2.1. El error de interpolación

El objetivo de la interpolación es obtener, o mejor dicho, predecir los valores de la función f con nuestro polinomio de interpolación. Al hacer esto, asumiremos un determinado *error*, el cuál de forma intuitiva es

$$E_n(x) = f(x) - P_n(x), \quad x \in [a, b]$$

Como hemos visto anteriormente, los polinomios de Lagrange están diseñados para pasar exactamente por los puntos de datos dados, lo que implica que el error de interpolación se reduce a cero en estos puntos específicos. Sin embargo, fuera de estos puntos, el comportamiento de la función f puede ser bastante complejo y difícil de predecir. Sin hipótesis adicionales sobre esta función no podremos calcular este error, además que puede no existir una única función por la que pasen los puntos conocidos. Veamos un ejemplo:

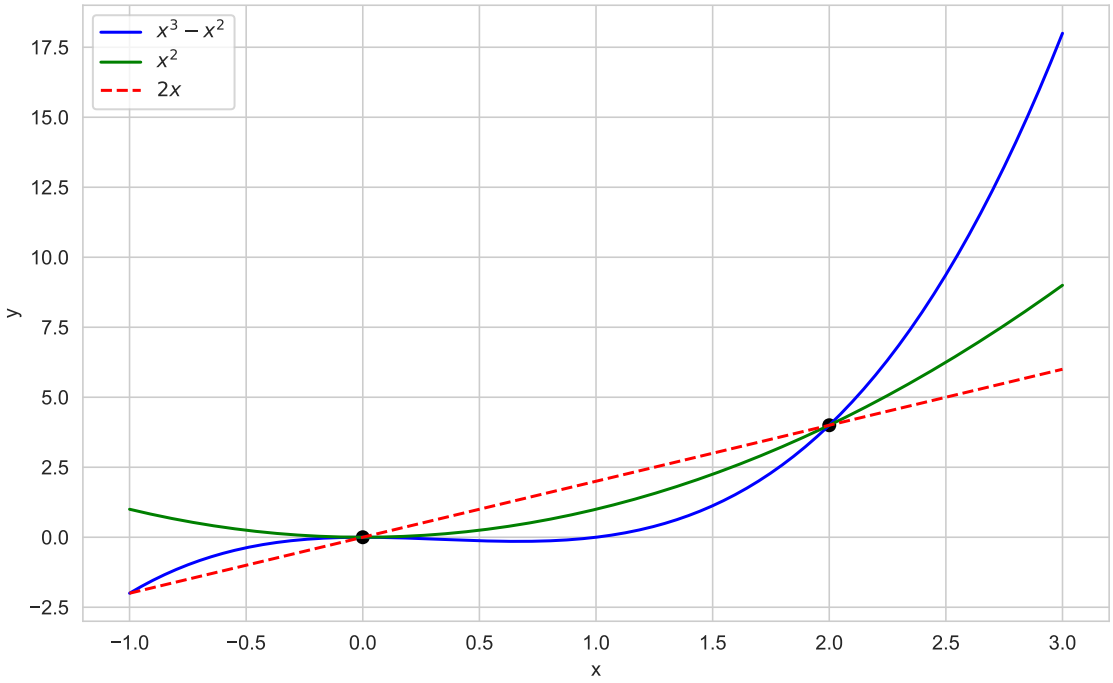

Figura 2.3: Intersección de funciones y polinomio de Lagrange

No obstante, cuando la función f es suficientemente regular, podemos precisar el error que se comete en cada punto de interpolación en términos de las derivadas de f. Concretamente:

Notación 2.2 Dados $n+1$ puntos distintos $\{x_0, x_1, \ldots, x_n\}$ denotaremos

$$\Pi_n(x) = \prod_{i=0}^{n}(x - x_i) = (x - x_0)(x - x_1)\cdots(x - x_n) \tag{2.5}$$

Teorema 2.2 Sea $f \in \mathcal{C}^{n+1}([a, b])$, $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ con $x_i \neq x_j$ si $i \neq j$ y $P_n \in \mathcal{P}_n$ el polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_n\}$. Para cada $x \in [a, b]$ existe $\xi_n \in I_x$, siendo I_x el mínimo intervalo cerrado que contiene a los puntos $\{x_0, x_1, \ldots, x_n, x\}$, tal que

$$E_n(x) = f(x) - P_n(x) = \frac{f^{n+1)}(\xi_x)}{(n+1)!}\Pi_n(x). \tag{2.6}$$

Demostración 2.2 Dado $x \in [a,b]$ dividimos la demostración en 2 partes:

1. Si $x = x_i$ para algún $i \in \{0,1,\dots,n\}$, tenemos que $f(x_i) = P_n(x_i)$ y $\Pi_n(x_i) = \prod_{i=0}^{n}(x-x_i) = (x-x_0)\cdots(x-x_n) = 0$. Por lo que se verifica (2.6).

2. Si $x \neq x_i \ \forall i \in \{0,1,\dots,n\}$. Probar (2.6) es equivalente a probar:

$$f^{n+1)}(\xi_x)\Pi_n(x) - (f(x) - P_n(x))(n+1)! = 0 \qquad (1)$$

Debemos buscar una función $F : [a,b] \to \mathbb{R}$, que sea $n+1$ veces derivable tal que

$$F^{n+1)}(y) = f^{n+1)}(y)\Pi_n(x) - (f(x) - P_n(x))(n+1)! \qquad (2)$$

y que se verifique que $F^{n+1)} = 0$. Así tendríamos probado (1), y por tanto (2.6). Consideramos F dada por:

$$F(y) = (f(y) - P_n(y))\Pi_n(x) - (f(x) - P_n(x))\Pi_n(y) \qquad (3)$$

con $y \in [a,b]$. Elegimos esta función porque va a verificar (2) y cumple que $F^{n+1)}(\xi_x) = 0$. Veamoslo:

- **Veamos que F dada por (3) verifica (2):** Como $F \in \mathcal{C}^{n+1}([a,b])$, $P_n \in \mathcal{C}^{n+1}([a,b])$ y $\Pi_n(x) \in \mathcal{C}^{n+1}([a,b])$, podemos derivar F $n+1$ veces y obtenemos:

$$F^{n+1)}(y) = (f^{n+1)}(y) - P_n^{n+1)})\Pi_n(x) - (f(x) - P_n(x))\Pi_n^{n+1)}(y)$$

Como el grado$(P_n) \leq n$, entonces $P_n^{n+1)}(y) = 0$. Por otro lado, como $\Pi_n(y) = \prod_{i=0}^{n}(y-x_i) = (y-x_0)(y-x_1)\cdots(y-x_n)$ es un polinomio de grado $n+1$, si lo derivamos $n+1$ veces obtenemos $(n+1)n(n-1)(n-2)\dots 1 = (n+1)!$. Entonces $\Pi_n^{n+1)} = (n+1)!$. Por tanto:

$$F^{n+1)}(y) = f^{n+1)}(y)\Pi_n(x) - (f(x) - P_n(x))(n+1)!$$

quedando asi probado (2).

- **Veamos que $F^{n+1)}(\xi_x) = 0$:** Probar esto es equivalente a probar que ξ_x es raíz de $F^{n+1)}$, que a su vez es equivalente a probar que $F^{n)}$ tiene al menos dos raíces gracias al teorema de Rolle. Reiteramos el argumento hasta llegar a que F tiene al menos $n+2$ raíces. Se tiene entonces:

$$F(x_i) = \underbrace{(f(x_i) - P_n(x_i))}_{(*)}\Pi_n(x) - (f(x) - P_n(x))\underbrace{\Pi_n(x_i)}_{(**)} = 0$$

ya que:

$$(*) \quad f(x_i) - P_n(x_i) = 0, \quad \text{ya que } f(x_i) = P_n(x_i)$$

$$(**) \quad \Pi_n(x_i) = (x_i - x_0)(x_i - x_1)\cdots\underbrace{(x_i - x_i)}_{\to 0}\cdots(x_i - x_n) = 0$$

y

$$F(x) = (f(x) - P_n(x))\Pi_n(x) - (f(x) - P_n(x))\Pi_n(x) = 0$$

Por ende, obtenemos que F tiene al menos $n+2$ raíces, que son $\{x_0, x_1, \dots, x_n, x\}$.

□

Observación 2.3

1. La función $E_n(x)$ indicada en (2.6) no se puede usar para calcular el error, ya que normalmente no conocemos ξ_x como una función de x (excepto cuando $f^{n+1} \equiv$ constante, es decir, $f \in \mathcal{P}_{n+1}$). Sin embargo, podemos establecer un límite para el error, dado por:

$$|f(x) - P_n(x)| \leq \frac{|\Pi_n(x)|}{(n+1)!} \|f^{n+1}\|_{L^\infty(a,b)},\ x \in [a,b] \tag{2.7}$$

donde

$$||g||_{L^\infty(a,b)} = \max_{a \leq x \leq b} |g(x)| \tag{2.8}$$

representa la *norma máxima* de una función continua $g : [a,b] \to \mathbb{R}$.

2. La estimación del error proporcionada es *óptima* en el sentido de que existe una función para la cual se cumple la igualdad. De hecho, basta con considerar la función

$$f(x) = \Pi_n(x) = \prod_{i=0}^{n} (x - x_i)$$

que satisface

$$P_n(x) = 0 \quad \text{y} \quad f^{n+1)}(x) = (n+1)!$$

estableciendo así la igualdad $|\Pi_n(x)| = |\Pi_n(x)|$ en (2.7).

Ejemplo 2.1 Si consideramos la función

$$f(x) = \text{sen } x, \quad x \in \left[0, \frac{\pi}{2}\right]$$

se verifica que

$$P_1(x) = f(x_0)L_0(x) + f(x_1)L_1(x) = \underbrace{f(0)}_{0} \frac{x - \frac{\pi}{2}}{0 - \frac{\pi}{2}} + \underbrace{f\left(\frac{\pi}{2}\right)}_{1} \frac{x - 0}{\frac{\pi}{2} - 0} = \frac{2}{\pi} x$$

es el polinomio de interpolación de f en los puntos $\left\{x_0 = 0, x_1 = \frac{\pi}{2}\right\}$. Como, en este caso,

$$|f''(x)| = |\text{sen } x| \leq 1, \quad x \in \left[0, \frac{\pi}{2}\right]$$

se presenta en $\tilde{x} = \frac{\pi}{4}$ y toma el valor $|\Pi_1(\tilde{x})| = \frac{\pi^2}{16}$ entonces

$$|E_1(x)| = |f(x) - P_1(x)| \leq \frac{\frac{\pi^2}{16}}{(1+1)!} \cdot 1 = \frac{\pi^2}{32} \backsimeq 0.308425, \quad x \in \left[0, \frac{\pi}{2}\right].$$

Creamos una función que nos calcule la cota de error. El código de la función se encuentra en el Apéndice A, bajo el nombre "Función Cota error".

```
from sympy import sin, pi
def f(x):
    return sin(x)
import numpy as np
error_acotado=acotar_error(f, 2, 0, np.pi/2)
print("El error de interpolación es inferior a:", error_acotado)
```

```
## El error de interpolación es inferior a: 0.308425137528880
```

Podemos observar que obtenemos el mismo resultado que obtuvimos de manera manual. Veamos gráficamente ambas funciones:

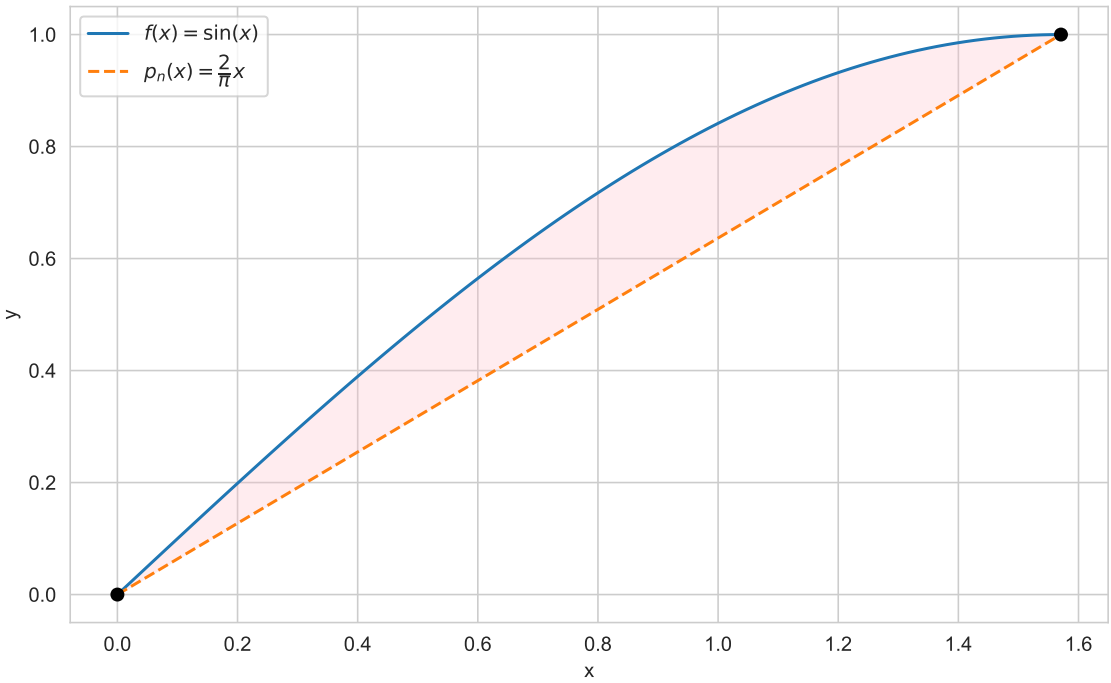

Figura 2.4: Error de interpolación

El área sombreada de rojo es el error de interpolación. Como hemos visto anteriormente, este es inferior a 0.308425137528880. Calculemos el área para comprobar que es inferior a ese valor. Para ello creamos una función, el código de esta se encuentra en el Apéndice A, bajo el nombre "Área entre 2 funciones".

```
from sympy import sin, pi
def f(x):
    return sin(x)
def pn(x):
    return (2 / pi) * x
area = calcular_area_entre_funciones(f, pn, 0, pi/2)
print("El área entre las funciones es:", area)
```

```
## El área entre las funciones es: 0.2146018366025517
```

Efectivamente se cumple la hipótesis.

Supongamos que $f : [a, b] \to \mathbb{R}$ y $P_n \in \mathcal{P}_n$ es el polinomio de interpolación de f en una selección de puntos diferentes $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$. Es natural preguntarse si se cumple la siguiente afirmación:

$$\lim_{n \to +\infty} P_n(x) = f(x)$$

para cada $x \in [a, b]$. Sin embargo, la respuesta suele ser negativa. Un ejemplo clásico que ilustra esta situación es la función $f(x) = \frac{1}{1+25x^2}$. Si consideramos una secuencia de puntos equidistantes para $i = 0, 1, \ldots, n$, entonces el polinomio de interpolación de Lagrange $P_n(x)$ de grado n asociado a estos puntos no convergerá uniformemente a $f(x)$ a medida que n tiende a infinito.

Usando y adaptando a Python el código de [**?**], creamos un gráfico para poder visualizarlo:

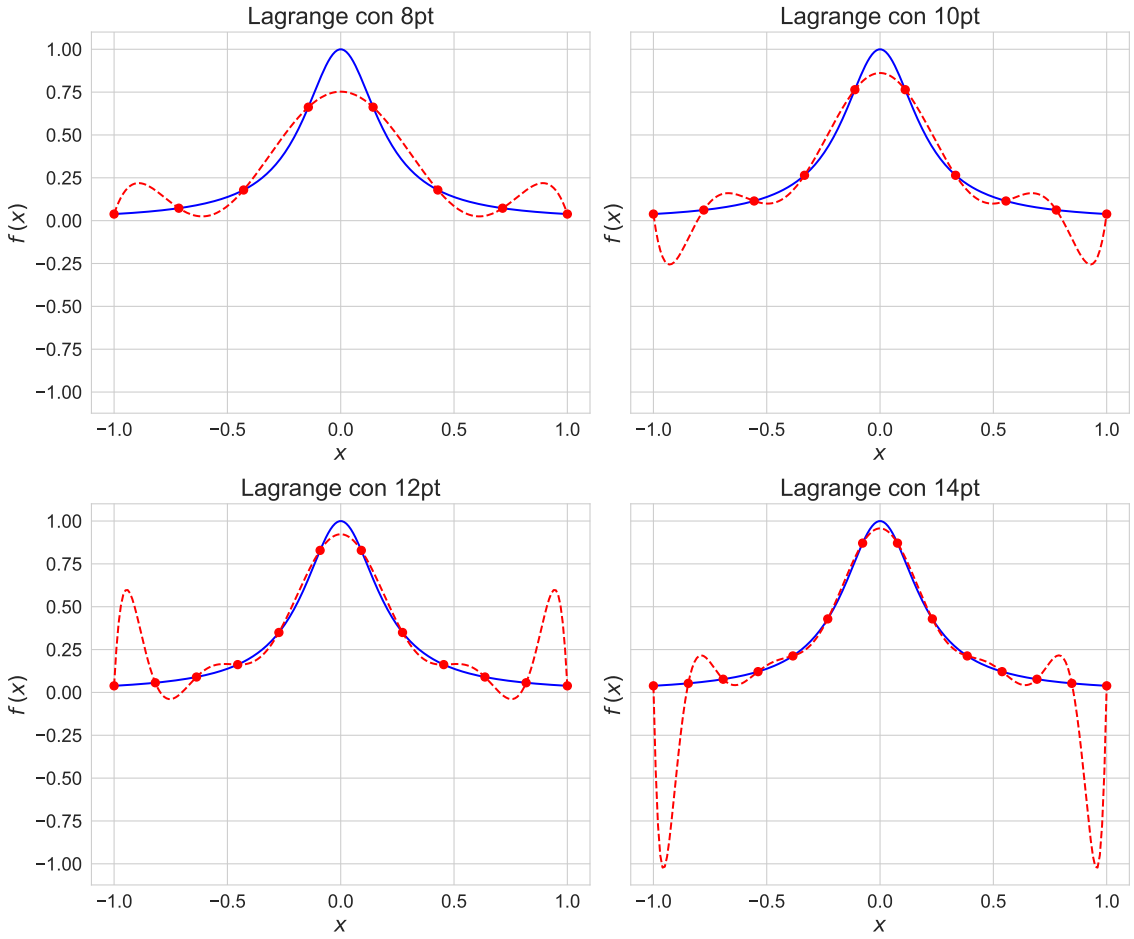

Figura 2.5: Fenómeno de Runge

A este fenómeno se le conoce como *el fenómeno de Runge* [1], un problema de oscilación en los bordes de un intervalo que ocurre cuando se usa la interpolación polinomial con polinomios de alto grado sobre un conjunto de interpolaciones equiespaciadas puntos. Esto lo podemos solucionar con *el polinomio de Chebychev*, del cual hablaremos más adelante.

2.2. Aproximación de datos y método de Neville

El Método de Neville destaca por su utilidad cuando se busca interpolar una función en un punto específico dentro del rango de datos proporcionados. El procedimiento consiste en calcular una secuencia de valores intermedios utilizando los datos conocidos y, posteriormente, combinarlos de manera apropiada para obtener la aproximación requerida.

Definición 2.2 Consideremos f como una función definida en el conjunto $x_0, x_1, \ldots, x_n$ y asumamos que $m_1, m_2, \ldots, m_k$ enteros distintos, con $0 \leq m_i \leq n$ para cada i. El polinomio de Lagrange que coincide con $f(x)$ en los puntos $x_{m_1}, x_{m_2}, \ldots, x_{m_k}$ se representa como $P_{m_1,m_1,\ldots,m_k}(x)$

Teorema 2.3 Sea f definida en $\{x_0, x_1, \ldots, x_k\}$ y sean x_j y x_i dos números distintos en este conjunto. Entonces

$$P(x) = \frac{(x - x_j)P_{0,1,\ldots,j-1,j+1,\ldots,k}(x) - (x - x_i)P_{0,1,\ldots,i-1,i+1,\ldots,k}(x)}{(x_i - x_j)}$$

es el k-ésimo polinomio de Lagrange que interpola f en los puntos $\{x_0, x_1, \ldots, x_k\}$

Notación 2.3 Para la facilidad de la notación, sea $Q \equiv P_{0,1,\ldots,i-1,i+1,\ldots,k}$ y $\hat{Q} \equiv P_{0,1,\ldots,j-1,j+1,\ldots,k}$.

Demostración 2.3 Dado que $Q(x)$ y $\hat{Q}(x)$ son polinomios de grado $k-1$ o menor, $P(x)$ tiene un grado máximo de k.
En primer lugar, notemos que $\hat{Q}(x_i) = f(x_i)$ implica que

$$P(x_i) = \frac{(x_i - x_j)\hat{Q}(x_i) - (x_i - x_i)Q(x_i)}{(x_i - x_j)} = \frac{(x_i - x_j)}{(x_i - x_j)}\hat{Q}(x_i) = f(x_i)$$

De manera similar, dado que $Q(x_j) = f(x_j)$, concluimos que $P(x_j) = f(x_j)$.
Además, si $0 \leq r \leq k$ y $r \neq i \neq j$, entonces $Q(x_r) = \hat{Q}(x_r) = f(x_r)$. Por lo tanto,

$$P(x_r) = \frac{(x_r - x_j)\hat{Q}(x_r) - (x_r - x_i)Q(x_r)}{(x_i - x_j)} = \frac{(x_i - x_j)}{(x_i - x_j)}f(x_r) = f(x_r)$$

Sin embargo, por definición, $P_{0,1,\ldots,k}(x)$ es el único polinomio de grado máximo k que coincide con f en $\{x_0, x_1, \ldots, x_k\}$. Por lo tanto, $P \equiv P_{0,1,\ldots,k}$. □

El algoritmo de interpolación iterada de Neville es un método numérico utilizado para interpolar un valor $f(x)$ correspondiente a un valor x específico, dado un conjunto de puntos (x_i, y_i). La descripción del algoritmo paso a paso es:

- ENTRADA: números $\{x, x_0, x_1, \ldots, x_n\}$; valores $f(x_0), f(x_1), \ldots, f(x_n)$ como primera columna $Q_{0,0}, Q_{1,0}, \ldots, Q_{n,0}$ de la tabla Q.
- SALIDA: la tabla Q con $P(x) = Q_{n,n}$

1. Para $i = 1, 2, \ldots, n$
   Para $j = 1, 2, \ldots, i$
   Hacer

$$Q_{i,j} = \frac{(x - x_{i-j})Q_{i,j-1} - (x - x_i)Q_{i-1,j-1}}{(x_i - x_{i-j})} \tag{2.9}$$

2. SALIDA (Q);
   PARE

El código de la función se encuentra en el Apéndice A, bajo el nombre "Interpolación Neville".

```
x=np.array([1,2,3])
y=x**4
objetivo = 2.9
tabla = interpolacion_neville(x, y, objetivo)
```

```
## [1, 0, 0]
## [16, 29.5, 0]
## [81, 74.5, 72.25]
```

Recordemos que en Python, al igual que en muchos otros lenguajes de programación, los índices de las listas, tuplas, tablas y otros tipos de estructuras de datos comienzan desde 0 en lugar de empezar desde 1. Es decir, que la posición (1,1) de la matriz es (0,0) para el algoritmo que hemos creado, y que nuestros datos a su vez también comienzan en 0, $\{x_0, x_1, \ldots, x_n\}$. Por tanto aplicando la fórmula (2.9):

$$Q_{2,1}(2.9) = \frac{(2.9-2)\cdot 81 - (2.9-3)\cdot 16}{(3-2)} = 74.5$$

Añadamos un nuevo punto x_3:

```
x=np.array([1,2,3,4])
y=x**4
objetivo = 2.9
tabla = interpolacion_neville(x, y, objetivo)
```

```
## [1, 0, 0, 0]
## [16, 29.5, 0, 0]
## [81, 74.5, 72.25, 0]
## [256, 63.499999999999986, 69.55, 70.54]
```

Observación 2.4 Como podemos observar, cuando se agrega un nuevo punto en la interpolación mediante el método de Neville, los cálculos ya realizados pueden ser aprovechados nuevamente para mejorar la precisión de la interpolación. Esto se debe a que el método de Neville emplea un enfoque iterativo para desarrollar el polinomio interpolante, donde cada paso utiliza la información obtenida en las etapas previas. En lugar de reiniciar todo el proceso de interpolación para cada nuevo punto, el método de Neville puede utilizar la información ya calculada para agilizar la interpolación.

2.3. Fórmula de interpolación de Newton

Como habíamos comentado anteriormente, la fórmula de clásica de el polinomio de interpolación de Lagrange que vimos en el teorema (2.1) requiere recalcular todos los polinomios básicos al introducir un nuevo punto, lo cual puede resultar poco eficiente. Es por esto que vimos otro método, el método Neville. El problema de este, es que no nos ofrece una expresión explicita de nuestro polinomio, solo nos devuelve el valor de la función interpolada en el valor objetivo que deseamos.

En esta sección, ahondaremos en un enfoque más eficaz para construir el polinomio de interpolación, lo que nos permitirá actualizar el polinomio existente al incluir nuevos puntos de interpolación. Esta *fórmula de interpolación de Newton* nos brindará la posibilidad de expresar el polinomio de interpolación en términos de "diferencias", ya sean divididas o finitas, entre los valores de la función en los puntos de interpolación.

Definición 2.3 Supongamos que $f : [a, b] \to \mathbb{R}$ y $\{x_0, x_1, x_2, \dots\} \subset [a, b]$ donde $x_i \neq x_j$ para $i \neq j$. Para cada $i \in \mathbb{N} \cup \{0\}$, definimos las siguientes expresiones:

$$\begin{cases} f[x_i] = f(x_i) \\ f[x_1, x_{i+1}, \dots, x_{i+m}] = \dfrac{f[x_i, x_{i+1}, \dots, x_{i+m-1}] - f[x_{i+1}, x_{i+2}, \dots, x_{i+m}]}{x_i - x_{i+m}} \end{cases}$$

Donde $f[x_i, x_{i+1}, \dots, x_{i+m}]$ se refiere a la *diferencia dividida* de orden $m \in \mathbb{N} \cup \{0\}$ de f en el punto x_i. Similarmente, definimos la secuencia:

$$\begin{cases} \Delta^0 f(x_i) = f(x_i) \\ \Delta^m f(x_i) = \Delta^{m-1} f(x_{i+1}) - \Delta^{m-1} f(x_i) \end{cases}$$

Aquí, $\Delta^m f(x_i)$ representa la *diferencia finita* de orden $m \in \mathbb{N} \cup \{0\}$ de f en el punto x_i.

Observación 2.5 Los operadores Δ^m son *lineales*, es decir,

$$\Delta^m(\alpha f + \beta g)(x_i) = \alpha \Delta^m f(x_i) + \beta \Delta^m g(x_i),\ \alpha, \beta \in \mathbb{R},\ m \in \mathbb{N} \cup \{0\}$$

Teorema 2.4 Si $f : [a, b] \to \mathbb{R}$ y $\{x_0, x_1, \dots, x_n\} \subset [a, b]$ es una red de puntos de paso $h > 0$, es decir, $x_i = x_0 + ih$, $i = 0, 1, \dots, n$, entonces

$$f[x_i, x_{i+1}, \dots, x_{i+m}] = \frac{\Delta^m f(x_i)}{m! h^m}$$

Demostración 2.4 Lo mostramos por inducción sobre el orden de las diferencias:

1. Para las diferencias de orden $m = 1$, como $x_{i+1} = x_i + h$ entonces

$$f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = \frac{\Delta f(x_i)}{h}$$

2. Supongamos cierto el resultado para las diferencias de orden $m - 1$ y lo probamos para las de orden m. Por definición se tiene que

$$f[x_i, x_{i+1}, \dots, x_{i+m}] = \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+m}] - f[x_i, x_{i+1}, \dots, x_{i+m-1}]}{x_{i+m} - x_i}$$

Como $x_{i+m} - x_i = mh$, aplicando las hipótesis de inducción, podemos escribir

$$\begin{aligned} f[x_i, x_{i+1}, \ldots, x_{i+m}] &= \frac{1}{mh}\left(\frac{\Delta^{m-1} f(x_{i+1})}{(m-1)! h^{m-1}} - \frac{\Delta^{m-1} f(x_i)}{(m-1)! h^{m-1}}\right) \\ &= \frac{\Delta^{m-1} f(x_{i+1}) - \Delta^{m-1} f(x_i)}{m! h^m} = \frac{\Delta^m f(x_i)}{m! h^m} \end{aligned}$$

□

Teorema 2.5 Sea $f : [a, b] \to \mathbb{R}$ y $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ con $x_i \neq x_j$ si $i \neq j$. El polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_n\}$ viene dado por

$$\begin{aligned} P_n(x) &= f(x_0) + \sum_{i=1}^{n} \Pi_{i-1}(x) f[x_0, x_1, \ldots, x_i] \\ &= f(x_0) + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2] \\ &\quad + \cdots + (x - x_0)(x - x_1) \cdots (x - x_{n-1}) f[x_0, x_1, \ldots, x_n] \end{aligned}$$

Además, si $x \notin \{x_0, x_1, \ldots, x_n\}$, entonces

$$E_n(x) = f(x) - P_n(x) = \Pi_n(x) f[x_0, x_1, \ldots, x_n, x] \tag{2.10}$$

Demostración 2.5 Procedemos por inducción sobre el grado del polinomio:

1. Para $n = 0$, $P_0(x) = f(x_0)$ es el polinomio de interpolación de f en x_0 y, para todo punto $x \neq x_0$, se verifica que

   $$f[x_0, x] = \frac{f(x) - f(x_0)}{x - x_0},$$

   por lo que

   $$f(x) = f(x_0) + (x - x_0) f[x_0, x] = P_0 + \Pi_0(x) f[x_0, x]$$

2. Suponemos cierto el resultado $n - 1$, es decir, que

   $$\begin{aligned} P_{n-1}(x) =& f(x_0) + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2] \\ &+ \cdots + (x - x_0)(x - x_1) \cdots (x - x_{n-2}) f[x_0, x_1, \ldots, x_{n-1}] \end{aligned}$$

   es el polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_{n-1}\}$ y

   $$f(x) - P_{n-1}(x) = \Pi_{n-1}(x) f[x_0, x_1, \ldots, x_{n-1}, x] \tag{2.11}$$

   para $x \notin \{x_0, x_1, \ldots, x_{n-1}\}$. Consideremos el polinomio

   $$\begin{aligned} Q(x) =& f(x_0) + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2] \\ &+ \cdots + (x - x_0)(x - x_1) \cdots (x - x_{n-2}) f[x_0, x_1, \ldots, x_{n-1}] \\ &+ (x - x_0)(x - x_1) \cdots (x - x_{n-1}) f[x_0, x_1, \ldots, x_n] \end{aligned}$$

   que, por la hipótesis de inducción, podemos expresarlo como

   $$Q(x) = P_{n-1}(x) + \Pi_{n-1}(x) f[x_0, x_1, \ldots, x_n]$$

Obviamente $Q \in \mathcal{P}_n$,

$$Q(x_i) = P_{n-1}(x_i) + \Pi_{n-1}(x_i) f[x_0, x_1, \ldots, x_n] = P_{n-1}(x_i) = f(x_i)$$

para $i = 0, 1, \ldots, n-1$ y

$$Q(x_n) = P_{n-1}(x_n) + \Pi_{n-1}(x_n) f[x_0, x_1, \ldots, x_n] = f(x_n)$$

aplicando (2.11) en el punto $x = x_n$. Por tanto, por unicidad del polinomio de interpolación, Q es el polinomio de interpolación P_n de f en los puntos $\{x_0, x_1, \ldots, x_n\}$. Por otra parte, para todo punto $x \notin \{x_0, x_1, \ldots, x_n\}$ se verifica que

$$\begin{aligned} f[x_0, x_1, \ldots, x_n, x] &= f[x, x_0, \ldots, x_{n-1}, x_n] \\ &= \frac{f[x, x_0, \ldots, x_{n-2}, x_{n-1}] - f[x_0, x_1, \ldots, x_{n-1}, x_n]}{x - x_n} \\ &= \frac{f[x_0, x_1, \ldots, x_{n-1}, x] - f[x_0, x_1, \ldots, x_{n-1}, x_n]}{x - x_n} \end{aligned}$$

de donde

$$f[x_0, x_1, \ldots, x_{n-1}, x] = f[x_0, x_1, \ldots, x_{n-1}, x_n] + (x - x_n) f[x_0, x_1, \ldots, x_n, x]$$

Sustituyendo este valor en (2.11) se obtiene que

$$f(x) - P_{n-1}(x) = \Pi_{n-1}(x)(f[x_0, \ldots, x_{n-1}, x] + (x - x_n) f[x_0, \ldots, x_n, x])$$

para $x \notin \{x_0, x_1, \ldots, x_n\}$, es decir,

$$\begin{aligned} f(x) &= P_{n-1}(x) + \Pi_{n-1}(x) f[x_0, x_1, \ldots, x_{n-1}, x_n] \\ &\quad + \Pi_{n-1}(x)(x - x_n) f[x_0, x_1, \ldots, x_n, x] \\ &= P_n(x) + \Pi_n(x) f[x_0, x_1, \ldots, x_n, x] \end{aligned}$$

de donde sigue (2.10).

□

Observación 2.6

1. De (2.6) y (2.10) se deduce que, cuando la función $f \in \mathcal{C}^{n+1}([a, b])$, para cada punto $x \in [a, b]$ $\{x_0, x_1, \ldots, x_n\}$ existe $\xi_x \in I_x$ verificando

$$f[x_0, x_1, \ldots, x_n, x] = \frac{f^{n+1)}(\xi_x)}{(n+1)!} \tag{2.12}$$

   Nótese que, aunque esta última expresión pudiera parecer que se determina el error de forma exacta, esto no es del todo cierto, pues para calcular $f[x_0, x_1, \ldots, x_n, x]$ se necesita el valor de $f(x)$ que, en general, es desconocido (obviamente, si conocemos el valor que toma la función f en el punto x entonces el error que se comete $f(x) - P_n(x)$ se determina explícitamente).

2. En la siguiente tabla (2.1) se muestran el algoritmo para construir el polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_n\}$ a partir de las diferencias divididas.

Tabla 2.1: Algoritmo de Newton diferencias divididas

$f(x_0)$	$f[x_0, x_1]$	$\cdots$	$f[x_0, x_1, \ldots, x_{n-1}]$	$f[x_0, x_1, \ldots, x_n]$
$f(x_1)$	$f[x_1, x_2]$	$\cdots$	$f[x_1, x_2, \ldots, x_n]$	
$f(x_2)$	$f[x_2, x_3]$	$\cdots$		
$\vdots$	$\vdots$	$\ddots$		
$f(x_{n-1})$	$f[x_{n-1}, x_n]$			
$f(x_n)$				

La propiedad más significativa de la fórmula de interpolación de Newton radica en su capacidad para obtener el polinomio de interpolación f en ciertos puntos a partir de polinomios de interpolación en subconjuntos de ellos. En particular,

Corolario 2.1 Sea $f : [a, b] \to \mathbb{R}$ y $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ con $x_i \neq x_j$ si $i \neq j$. Si P_n es el polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_n\}$ y x_{n+1} es otro punto de $[a, b]$ tal que $x_{n+1} \notin \{x_0, x_1, \ldots, x_n\}$, entonces

$$P_{n+1}(x) = P_n(x) + \Pi_n(x) f[x_0, x_1, \ldots, x_n, x_{n+1}]$$

es el polinomio de interpolación de f en los puntos $\{x_0, x_1, \ldots, x_n, x_{n+1}\}$

Creamos una función en Python para que nos calcule las diferencias divididas de Newton. Sigue el siguiente formato aproximadamente:

- ENTRADA: los números $\{x_0, x_1, \ldots, x_n\}$; valores $f(x_0), f(x_1), \ldots, f(x_n)$ conforme $F_{0,0}, F_{1,0}, \ldots, F_{n,0}$.
- SALIDA: los números $F_{0,0}, F_{1,1}, \ldots, F_{n,n}$ donde

$$P_n(x) = F_{0,0} + \sum_{i=1}^{n} F_{i,i} \prod_{j=0}^{i-1} (x - x_j). \quad (F_{i,i} \text{ es } f[x_0, x_1, \ldots, x_i].)$$

1. Para $i = 1, 2, \ldots, n$
   Para $j = 1, 2, \ldots, i$
   Hacer

$$F_{i,j} = \frac{F_{i,j-1} - F_{i-1,j-1}}{x_i - x_{i-j}}. \quad (F_{i,j} = f[x_{i-j}, \ldots, x_i]) \tag{2.13}$$

2. SALIDA tabla;
   PARE

El código de la función se encuentra en el Apéndice A, bajo el nombre "Diferencias Newton"

```
x = [1, 1.3, 1.6, 1.9]
y = [0.7651977, 0.6200860, 0.4554022, 0.2818186]
coeficientes = diferencias_newton(x, y)
```

```
## +-----+------+----------+-----------+------------+-----------+
## |   i |   xi |       fi |      F[1] |       F[2] |      F[3] |
## +=====+======+==========+===========+============+===========+
## |   0 |  1   | 0.765198 | -0.483706 | -0.108734  | 0.0658784 |
## +-----+------+----------+-----------+------------+-----------+
## |   1 |  1.3 | 0.620086 | -0.548946 | -0.0494433 | 0         |
## +-----+------+----------+-----------+------------+-----------+
## |   2 |  1.6 | 0.455402 | -0.578612 |  0         | 0         |
## +-----+------+----------+-----------+------------+-----------+
## |   3 |  1.9 | 0.281819 |  0        |  0         | 0         |
## +-----+------+----------+-----------+------------+-----------+
```

$$\begin{aligned} P_3(x) \approx 0.765198 &- 0.483706(x-1) \\ &- 0.108734(x-1)(x-1.3) \\ &+ 0.0658784(x-1)(x-1.3)(x-1.6) \end{aligned}$$

Añadamos un nuevo punto x_4:

```
x = [1, 1.3, 1.6, 1.9, 2.2]
y = [0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623]
coeficientes = diferencias_newton(x, y)
```

```
## +-----+------+----------+-----------+------------+-----------+-----------+
## |   i |   xi |       fi |      F[1] |       F[2] |      F[3] |      F[4] |
## +=====+======+==========+===========+============+===========+===========+
## |   0 |  1   | 0.765198 | -0.483706 | -0.108734  | 0.0658784 | 0.0018251 |
## +-----+------+----------+-----------+------------+-----------+-----------+
## |   1 |  1.3 | 0.620086 | -0.548946 | -0.0494433 | 0.0680685 | 0         |
## +-----+------+----------+-----------+------------+-----------+-----------+
## |   2 |  1.6 | 0.455402 | -0.578612 |  0.0118183 | 0         | 0         |
## +-----+------+----------+-----------+------------+-----------+-----------+
## |   3 |  1.9 | 0.281819 | -0.571521 |  0         | 0         | 0         |
## +-----+------+----------+-----------+------------+-----------+-----------+
## |   4 |  2.2 | 0.110362 |  0        |  0         | 0         | 0         |
## +-----+------+----------+-----------+------------+-----------+-----------+
```

$$P_4(x) \approx P_3(x) + 0.0018251(x-1)(x-1.3)(x-1.6)(x-1.9)$$

2.4. Minimización del error y polinomio de Chebychev

Según lo observado, si $f \in \mathcal{C}^{n+1}([a,b])$ y P_n es el polinomio de interpolación de f en $n+1$ puntos distintos $\{x_0, x_1, \ldots, x_n\} \subset [a,b]$, para cada $x \in [a,b]$ existe $\xi_x \in [a,b]$ tal que

$$E_n(x) = f(x) - P_n(x) = \frac{f^{(n+1)}}{(n+1)!}\Pi_n(x)$$

De esta forma, llamando

$$M_{n+1} = \left\| f^{(n+1)} \right\|_{L^\infty(a,b)}$$

se tiene que

$$\|f - P_n\|_{L^\infty(a,b)} \leq \frac{M_{n+1}}{(n+1)!} \|\Pi_n\|_{L^\infty(a,b)}$$

donde la norma del máximo está definida en (2.8). El problema que nos planteamos ahora es el de encontrar, de entre todos los polinomios de interpolación de Lagrange de grado menor o igual que n, el que minimice esta acotación óptima (véase la observación 2.3) del error. Los únicos parámetros con los que se puede jugar son las abcisas de interpolación, puesto que la función y el grado del polinomio están fijados. Por tanto, debemos elegir los puntos de interpolación $\{x_0, x_1, \ldots, x_n\}$ que haga mínimo el valor de $\|\Pi_n\|_{L^\infty(a,b)}$. El primero en resolver este problema fue el matemático ruso Chebychev y su solución conduce a una clase de polinomios que también sirven para tratar otro tipo de problemas.

Definición 2.4 A la sucesión de polinomios $\{T_n\}_{n\geq 0}$ definida por recurrecia como sigue:

$$\begin{cases} T_0 = 1, \quad T_1(x) = x \\ T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x), \quad \forall n \geq 1 \end{cases} \tag{2.14}$$

se le denomina *sucesión de polinomios de Chebychev*.

Ejemplo 2.2 Los primeros polinomios de Chebychev de la sucesión definida en (2.14) que siguen a $T_0(x) = 1$ y $T_1(x) = x$ son:

$$T_2(x) = 2x^2 - 1, \quad T_3(x) = 4x^3 - 3x, \quad T_4(x) = 8x^4 - 8x^2 + 1, \quad \ldots$$

En Python se pueden calcular fácilmente esto polinomios de la siguiente manera:

```
import numpy as np
cheb = np.polynomial.chebyshev.Chebyshev((0,0,0,1))
coef = np.polynomial.chebyshev.cheb2poly(cheb.coef)
pol_chev_3 = np.poly1d(coef[::-1])
print(pol_chev_3)
```

```
##    3
## 4 x - 3 x
```

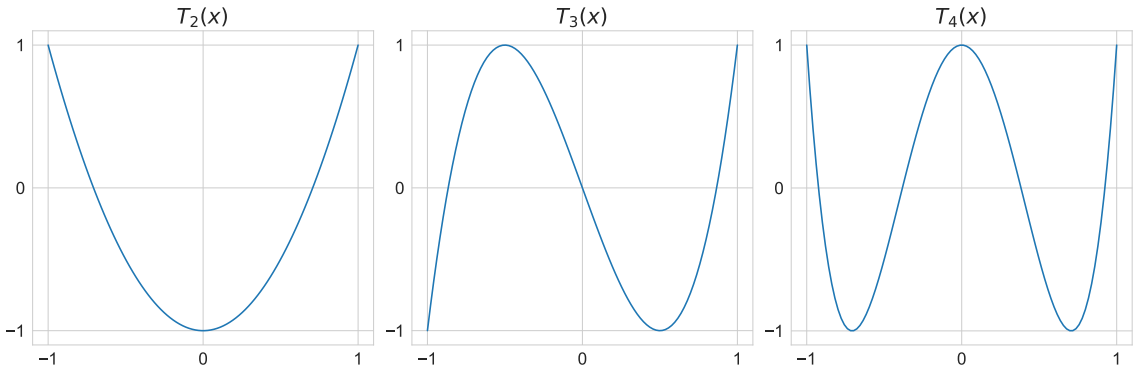

Figura 2.6: Polinomios de Chebychev

Proposición 2.1 Se tiene que

- T_n es un polinomio de grado n con coeficiente principal (o director) 2^{n-1}. Recordemos que este término se refiere al coeficiente que acompaña al término de mayor grado en un polinomio.
- Para cada $n \in \mathbb{N} \cup \{0\}$ se verifica que
$$T_n(x) = \cos(n \arccos x), \quad x \in [-1, 1]$$
- T_n tiene n raíces reales y distintas en (-1,1) que son:
$$x_k = \cos \frac{(2k+1)\pi}{2n}, \quad k = 0, \ldots, n-1$$
- T_n tiene $n-1$ puntos críticos en (-1,1), a saber
$$z_k = \cos \frac{k\pi}{n}, \quad k = 1, \ldots, n-1$$
y los extremos del intervalo corresponden a $k = 0$ y $k = n$. Además, para cada $k = 0, \ldots, n$, $T_n(z_k) = (-1)^k$ y $\|T_n\|_{\infty,[-1,1]} = 1$ para todo $n \geq 0$.

Definición 2.5 Un *polinomio mónico* es un polinomio cuyo coeficiente principal, es decir, el coeficiente del término de mayor grado, es igual a 1. En otras palabras, un polinomio mónico tiene un coeficiente líder de 1.

Notación 2.4 Vamos a denotar por
$$\mathcal{P}_n^m = \{P(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \in \mathcal{P}_n\}, \ n \in \mathbb{N} \cup \{0\}$$
al conjunto de *polinomios mónicos de grado n.*

Teorema 2.6 El polinomio mónico
$$T_n^m(x) = \frac{1}{2^{n-1}} T_n(x)$$
minimiza la norma del máximo en el intervalo [-1,1] entre los polinomios mónicos de grado n, es decir,
$$\|P\|_{L^\infty(-1,1)} \geq \|T_n^m\|_{L^\infty(-1,1)}$$
para todo $P \in \mathcal{P}_n^m$.

Demostración 2.6 Para empezar, según lo afirmado en la proposición 2.1, podemos establecer que el polinomio T_n^m es un polinomio mónico que, dentro del intervalo $[-1, 1]$, oscila entre los valores extremos de $\pm\dfrac{1}{2^{n-1}}$ de manera alternada en $n+1$ puntos distintos, definidos por

$$z_k = \cos\frac{k\pi}{n}, \quad k = 0, 1, \ldots, n-1$$

Por consiguiente, la norma máxima de T_n^m en este intervalo es $\dfrac{1}{2^{n-1}}$.
Ahora, para establecer una contradicción, supongamos que existe un polinomio

$$\|P\|_{L^\infty(-1,1)} \leq \|T_n^m\|_{L^\infty(-1,1)} = \frac{1}{2^{n-1}} \tag{2.15}$$

Consideremos entonces el polinomio

$$Q(x) = T_n^m(x) - P(x).$$

Dado que $T_n^m(x)$ y P son polinomios mónicos de grado n, el grado de Q es como máximo $n-1$. Por otra parte,

$$Q(z_k) = T_n^m(z_k) - P(z_k) = \frac{(-1)^k}{2^{n-1}} - P(z_k) = (-1)^k\left(\frac{1}{2^{n-1}} - (-1)^k P(z_k)\right)$$

Como podemos observar, obtenemos una expresión que indica que Q cambia alternadamente de signo en los $n+1$ puntos distintos $\{z_0, z_1, \ldots, z_n\}$. Así, según el teorema de Bolzano, esto implica que Q tiene al menos n raíces distintas. Dado que el grado de Q es como máximo $n-1$, concluimos que $Q \equiv 0$, es decir, $P \equiv T_n^m$, lo cual es una contradicción.
Por lo tanto, se puede concluir que la suposición inicial de que existe un polinomio P con norma máxima menor o igual a la de T_n^m es falsa, lo que implica que T_n^m tiene la mayor norma máxima en el intervalo $[-1, 1]$, como se quería demostrar. □

Creamos una función en Python que nos devuelvan los puntos de Chebychev en un determinado intervalo usando la expresión dada en (2.4). El código de la función se encuentra en el Apéndice A, bajo el nombre "Puntos Chebychev"

```
chebychev_intervalo = puntos_cheby(-2, 2, 10)
print(chebychev_intervalo)
```

```
## [ 1.97537668  1.78201305  1.41421356  0.907981    0.31286893 -0.31286893
##  -0.907981   -1.41421356 -1.78201305 -1.97537668]
```

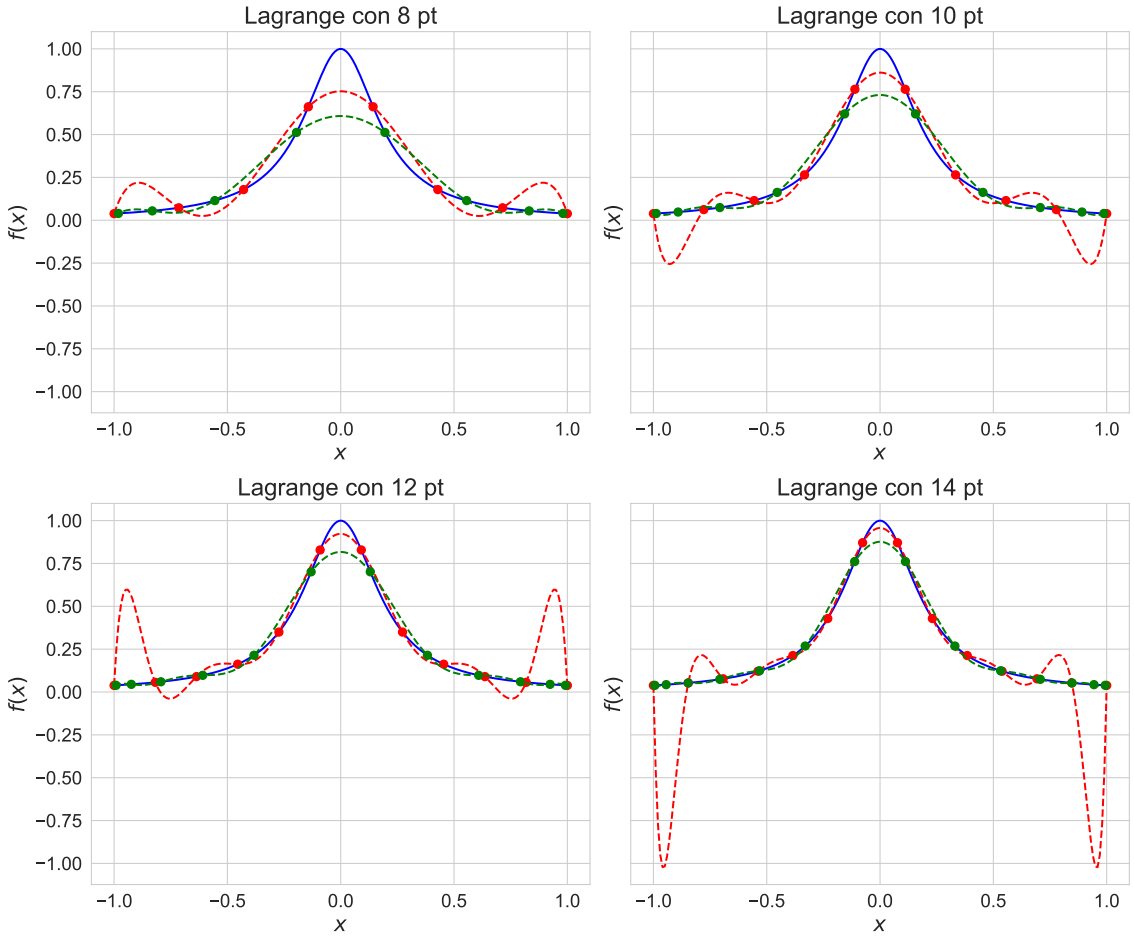

Figura 2.7: Evitando el fenómeno de Runge

Capítulo 3

Interpolación mediante funciones spline

La palabra "spline" deriva de un instrumento flexible utilizado en dibujo técnico, diseñado para trazar curvas suaves adaptándose a la forma deseada. Esta flexibilidad y capacidad de adaptación son precisamente las características que definen a las funciones spline en el ámbito matemático.

Desde una perspectiva matemática, una función spline definida en un intervalo $[a, b]$ está compuesta por polinomios definidos en subintervalos de $[a, b]$, cumpliendo ciertas condiciones de regularidad. Concretamente,

Definición 3.1 Consideremos $\Delta = \{a = x_0 < x_1 < \cdots < x_n = b\}$ como una *partición* del intervalo $[a, b]$. Una *función spline* de orden $k \in \mathbb{N}$ asociada a Δ, denotada por $S_\Delta : [a, b] \to \mathbb{R}$, cumple las siguientes condiciones:

- $S_\Delta \in \mathcal{C}^{k-1}([a, b])$
- En cada subintervalo $[x_i, x_{i+1}]$, para $i = 0, 1, \ldots, n-1$, coincide con un polinomio de grado $\leq k$.

Esencialmente, una función spline es una combinación de segmentos de polinomios suaves que se unen para formar una curva continua y diferenciable en el intervalo dado.

Observación 3.1

1. Cuando se toma el valor $k = 1$ hablaremos de *spline lineal*.
2. Cuando se toma el valor $k = 2$ hablaremos de *spline cuadrático*.
3. Cuando se toma el valor $k = 3$ nos referimos a *splines cúbico*, de las cuales hablaremos a continuación, ya que son las más usadas en la práctica.

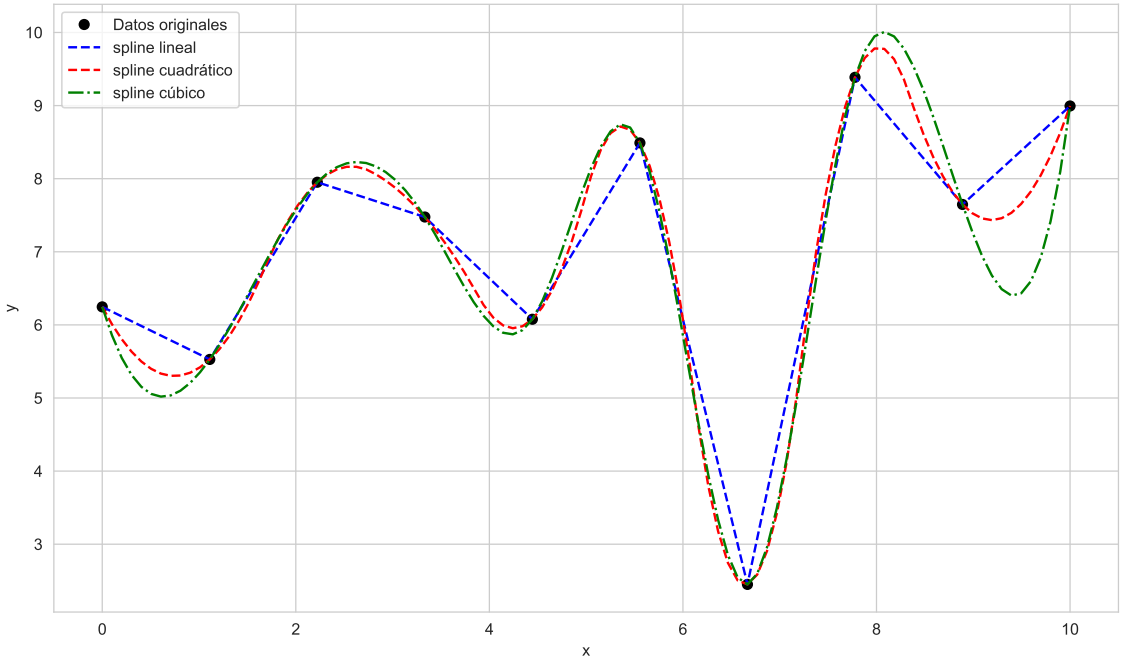

Figura 3.1: Comparación de splines

3.1. Interpolación de spline cúbicas

Notación 3.1 Para $y = (y_0, y_1, \ldots, y_n) \in \mathbb{R}^{n+1}$, designaremos como $S_\Delta(y, \cdot)$ a una función *spline* cúbica de interpolación que cumple

$$S_\Delta(y, x_i) = y_i$$

para $i = 0, 1, \ldots, n$.

Observación 3.2 Estas condiciones de interpolación no determinan de forma única una función spline. Por ejemplo, si tomamos $y = (y_0, y_1, y_2, y_3) \in \mathbb{R}^4$ entonces

$$S_\Delta(y, x) = \begin{cases} a_3x^3 + a_2x^2 + a_1x + a_0, & x \in [x_0, x_1] \\ b_3x^3 + b_2x^2 + b_1x + b_0, & x \in [x_1, x_2] \\ c_3x^3 + c_2x^2 + c_1x + c_0, & x \in [x_2, x_3] \end{cases}$$

con $S_\Delta \in \mathcal{C}^2([x_0, x_3])$ y $S_\Delta(y, x_i) = y_i$, $i = 0, 1, 2, 3$. Por tanto,

$$\begin{cases} a_3x_0^3 + a_2x_0^2 + a_1x_0 + a_0 = y_0 \\ a_3x_1^3 + a_2x_1^2 + a_1x_1 + a_0 = y_1 \\ b_3x_1^3 + b_2x_1^2 + b_1x_1 + b_0 = y_1 \\ b_3x_2^3 + b_2x_2^2 + b_1x_2 + b_0 = y_2 \\ c_3x_2^3 + c_2x_2^2 + c_1x_2 + c_0 = y_2 \\ c_3x_3^3 + c_2x_3^2 + c_1x_3 + c_0 = y_3 \\ 3a_3x_1^2 + 2a_2x_1 + a_1 = 3b_3x_1^2 + 2b_2x_1 + b_1 \\ 3b_3x_2^3 + 2b_2x_2 + b_1 = 3c_3x_2^3 + 2c_2x_2 + c_1 \\ 6a_3x_1 + 2a_2 = 6b_3x_1 + 2b_2 \\ 6b_3x_2^2 + 2b_2 = 6c_3x_2^2 + 2c_2 \end{cases}$$

Se puede demostrar que las 10 ecuaciones previas son mutuamente independientes; dado que tenemos 12 incógnitas, el sistema anterior presenta dos *grados de libertad*. En el caso general, en el cual $y = (y_0, y_1, \ldots, y_n) \in \mathbb{R}^{n+1}$, se origina un sistema de $4n-2$ ecuaciones ($2n$ relacionadas con los valores interpolados por S_Δ, $n-1$ que provienen de la continuidad de $S'\Delta$, y $n-1$ derivadas de la continuidad de $S''\Delta$) con $4n$ incógnitas.

Normalmente, para determinar unívocamente las funciones *spline* se impone una de las siguientes restricciones:

1. Tipo I: $S''_\Delta(y,a) = S''_\Delta(y,b) = 0$. Se suelen denominar condiciones *naturales*.
2. Tipo II: $S'_\Delta(y,a) = y'_0$, $S'_\Delta(y,b) = y'_n$ siendo $y'_0, y'_n \in \mathbb{R}$ valores prefijados.
3. Tipo III: $S^{k)}_\Delta(y,a) = S^{k)}_\Delta(y,b)$, $k = 0,1,2$.

Vamos a describir la función $S_\Delta(y,\cdot)$ a través de lo que llamaremos sus *momentos*, representados por

$$M_j = S''_\Delta(y, x_j)$$

para $j = 0, 1, \ldots, n$. En efecto, demostraremos que, una vez conocidos los momentos de una función *spline* cúbica, esta se determina de manera única a partir de ellos. Luego, demostraremos que los momentos están también completamente determinados por los datos del problema. De este modo, habremos establecido la existencia y unicidad de la función *spline* cúbica interpoladora, bajo condiciones de cualquiera de los tres tipos mencionados anteriormente. Además, el proceso será constructivo y nos proporcionará un algoritmo para calcular las funciones *spline* cúbicas.

Para cada $j \in 0, 1, \ldots, n-1$ vamos a definir

$$h_{j+1} = x_{j+1} - x_j$$

Dado que $S_\Delta(y,\cdot)$ coincide en cada intervalo $[x_j, x_{j+1}]$ con un polinomio de grado ≤ 3, la función $S''\Delta(y,\cdot)$ coincide en cada intervalo $[x_j, xj+1]$ con una función lineal que puede ser expresada en términos de los momentos $S_\Delta(y,\cdot)$ en los extremos del intervalo.

$$S''_\Delta(y,x) = M_j \frac{x_{j+1} - x}{h_{j+1}} + M_{j+1} \frac{x - x_j}{h_{j+1}},\ x \in [x_j, x_{j+1}] \tag{3.1}$$

Integrando esta igualdad, para cada $j = 0, 1, \ldots, n-1$, se obtiene

$$S'_\Delta(y,x) = -M_j \frac{(x_{j+1} - x)^2}{2h_{j+1}} + M_{j+1} \frac{(x - x_j)^2}{2h_{j+1}} + A_j = S'_\Delta(y,x),\ x \in [x_j, x_{j+1}] \tag{3.2}$$

Integrando nuevamente obtenemos

$$S_\Delta(y,x) = M_j \frac{(x_{j+1} - x)^3}{6h_{j+1}} + M_{j+1} \frac{(x - x_j)^3}{6h_{j+1}} + A_j(x - x_j) + B_j = S_\Delta(y,x),\ x \in [x_j, x_{j+1}]$$

Determinemos las constantes A_j y B_j para $j = 0, 1, \ldots, n-1$. Como

$$S_\Delta(y, x_j) = y_j$$

entonces

$$\begin{cases} y_j = M_j \dfrac{(x_{j+1}-x_j)^3}{6h_{j+1}} + B_j = M_j \dfrac{h_{j+1}^2}{6} + B_j \\ y_{j+1} = M_{j+1} \dfrac{(x_{j+1}-x_j)^3}{6h_{j+1}} + A_j(x_{j+1}-x_j) + B_j = M_{j+1}\dfrac{h_{j+1}^2}{6} + A_j h_{j+1} + B_j \end{cases}$$

de donde

$$B_j = y_j - M_j \frac{h_{j+1}^2}{6}$$

y

$$\begin{aligned} A_j =& \frac{1}{h_{j+1}} \left(y_{j+1} - B_j - M_{j+1}\frac{h_{j+1}^2}{6} \right) \\ =& \frac{1}{h_{j+1}} \left(y_{j+1} - y_j + M_j \frac{h_{j+1}^2}{6} - M_{j+1}\frac{h_{j+1}^2}{6} \right) \\ =& \frac{y_{j+1}-y_j}{h_{j+1}} - \frac{h_{j+1}}{6}(M_{j+1}-M_j) \end{aligned}$$

Por tanto, para cada $j \in \{0, 1, \ldots, n-1\}$, podemos escribirlo como

$$S_\Delta(y, x) = \alpha_j + \beta_j(x - x_j) + \gamma_j(x - x_j)^2 + \delta_j(x - x_j)^3, \ x \in [x_j, x_{j+1}]$$

donde, utilizando (3.1) y (3.2), estos coeficientes vienen dados por

$$\begin{cases} \alpha_j = S_\Delta(y, x_j) = y_j \\ \beta_j = S'_\Delta(y, x_j) = -M_j \dfrac{h_{j+1}}{2} + A_j = \dfrac{y_{j+1}-y_j}{h_{j+1}} - \dfrac{2M_j + M_{j+1}}{6} h_{j+1} \\ \gamma_j = \dfrac{S''_\Delta(y, x_j)}{2!} = \dfrac{M_j}{2} \\ \delta_j = \dfrac{S''_\Delta(y, x_j^+)}{3!} = \dfrac{M_{j+1}-M_j}{6h_{j+1}} \end{cases}$$

Es decir, para cada $j \in \{0, 1, \ldots, n-1\}$, se tiene que

$$\begin{aligned} S_\Delta(y, x) = y_j &+ \left(\frac{y_{j+1}-y_j}{h_{j+1}} - \frac{2M_j + M_{j+1}}{6} \right)(x - x_j) \\ &+ \frac{M_j}{2}(x - x_j)^2 + \frac{M_{j+1}-M_j}{6h_{j+1}}(x - x_j)^3 \end{aligned}$$

para todo $x \in [x_j, x_{j+1}]$.

La expresión anterior proporciona de manera exclusiva la función $S_\Delta(y, \cdot)$ siempre y cuando se tengan conocimiento de sus momentos. A continuación, examinaremos el procedimiento para calcular dichos momentos.

Dado que la función $S_\Delta(y, \cdot)$ pertenece a $\mathcal{C}^2([a, b])$, entonces, en particular, cumple con la condición:

$$S'_\Delta(y, x_j^-) = S'_\Delta(y, x_j^+) \tag{3.3}$$

para $j = 1, 2, \ldots, n-1$. Utilizando la relación (3.2) con el valor previamente calculado de A_j para cada índice $j \in \{1, 2, \ldots, n-1\}$, se comprueba que:

$$S'_\Delta(y, x_j^-) = S'_\Delta(y, x_j^+) \tag{3.4}$$

para $j = 1, 2, \ldots, n-1$. Al emplear la relación (3.2) con el valor de A_j ya determinado, se obtiene para cada índice $j \in \{1, 2, \ldots, n-1\}$ las siguientes igualdades:

$$\begin{aligned} S'_\Delta(y, x_j^-) &= M_j \frac{h_j}{2} + \frac{y_j - y_{j-1}}{h_j} - \frac{h_j}{6}(M_j - M_{j-1}) \\ &= \frac{y_j - y_{j-1}}{h_j} + \frac{h_j}{3} M_j + \frac{h_j}{6} M_{j-1} \end{aligned}$$

y

$$\begin{aligned} S'_\Delta(y, x_j^+) &= -M_j \frac{h_{j+1}}{2} + \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{h_{j+1}}{6}(M_{j+1} - M_j) \\ &= \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{h_{j+1}}{3} M_j - \frac{h_{j+1}}{6} M_{j+1} \end{aligned}$$

De este modo, para $j = 1, 2, \ldots, n-1$, la relación (3.4) establece que:

$$\frac{h_j}{6} M_{j-1} + \frac{h_j + h_{j+1}}{3} M_j + \frac{h_{j+1}}{6} M_{j+1} = \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j} \tag{3.5}$$

Introduciendo la notación:

$$\lambda_j = \frac{h_{j+1}}{h_j - h_{j+1}}, \quad \mu_j = 1 - \lambda = \frac{h_j}{h_j + h_{j+1}}$$

y

$$d_j = \frac{6}{h_j + h_{j+1}} \left(\frac{y_{j+1} - y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j} \right),$$

para $j = 1, 2, \ldots, n-1$, y multiplicando por $\dfrac{6}{h_j + h_{j+1}}$, la igualdad (3.1) se convierte en:

$$\mu_j M_{j-1} + 2M_j + \lambda_j M_{j+1} = d_j \tag{3.6}$$

para $j = 1, 2, \ldots, n-1$. Para examinar lo que sucede en los extremos (correspondientes a $j = 0$ y $j = n$) y así obtener otras dos ecuaciones, debemos considerar los diferentes tipos de condiciones:

a) Tipo I: $S''_\Delta(y,a) = S''_\Delta(y,b) = 0$. En este caso:

$$M_0 = S''_\Delta(y,a) = 0 = S''_\Delta(y,b) = M_n \Rightarrow M_0 = M_n = 0$$

Considerando

$$\lambda_0 = d_0 = \mu_n = d_n = 0$$

se verifica, obviamente,

$$\begin{cases} 2M_0 + \lambda_0 M_1 = d_0 \\ \mu_n M_{n-1} + 2M_n = d_n \end{cases} \tag{3.7}$$

b) Tipo II: $S'_\Delta(y,a) = y'_0,\ S_\Delta(y,b) = y'_n$. En este caso,

$$\begin{aligned} y'_0 = S'_\Delta(y,a) &= -M_0\frac{h_1}{2} + \frac{y_1 - y_0}{h_1} - \frac{h_1}{6}(M_1 - M_0) \\ &= -M_0\frac{h_1}{3} + \frac{y_1 - y_0}{h_1} - M_1\frac{h_1}{6} \end{aligned}$$

e

$$\begin{aligned} y'_n = S'_\Delta(y,b) &= M_n\frac{h_n}{2} + \frac{y_n - y_{n-1}}{h_n} - \frac{h_n}{6}(M_n - M_{n-1}) \\ &= M_n\frac{h_n}{3} + \frac{y_n - y_{n-1}}{h_n} + M_{n-1}\frac{h_n}{6} \end{aligned}$$

es dedir,

$$\begin{aligned} \frac{h_1}{3}M_0 + \frac{h_1}{6}M_1 &= \frac{y_1 - y_0}{h_1} - y'_0 \\ \frac{h_n}{6}M_{n-1} + \frac{h_n}{3}M_n &= y'_n - \frac{y_n - y_{n-1}}{h_n} \end{aligned}$$

Multiplicando la primera expresión por $\dfrac{6}{h_1}$ y la segunda por $\dfrac{6}{h_n}$ y llamando

$$\lambda_0 = \mu_n = 1, \quad d_0 = \frac{6}{h_1}\left(\frac{y_1 - y_0}{h_1} - y'_0\right) \quad \text{y} \quad d_n = \frac{6}{h_n}\left(y'_n - \frac{y_n - y_{n-1}}{h_n}\right)$$

volvemos a obtener la relación expresada en (3.7).

c) Tipo III: $S^{k)}_\Delta(y,a) = S^{k)}_\Delta(y,b)$, se obtiene

$$M_0 = M_n$$

y, de la relación $S'_\Delta(y,a) = S'_\Delta(y,b)$, se obtiene

$$-M_0\frac{h_1}{2} + \frac{y_1 - y_0}{h_1} - \frac{h_1}{6}(M_1 - M_0) = M_n\frac{h_n}{2} + \frac{y_n - y_{n-1}}{h_n} - \frac{h_n}{6}(M_n - M_{n-1})$$

Como $y_0 = y_n$ y $M_0 = M_n$ entonces

$$-M_n\frac{h_1}{2} + \frac{y_1 - y_n}{h_1} - \frac{h_1}{6}(M_1 - M_n) = M_n\frac{h_n}{2} + \frac{y_n - y_{n-1}}{h_n} - \frac{h_n}{6}(M_n - M_{n-1})$$

es decir,

$$\frac{h_n}{6}M_{n-1} + \frac{h_1 + h_n}{3}M_n + \frac{h_1}{6}M_1 = \frac{y_1 - y_n}{h_1} - \frac{y_n - y_{n-1}}{h_n}$$

Si multiplicamos la igualdad anterior $\frac{6}{h_1 + h_n}$ y denotamos

$$\lambda_n = \frac{h_1}{h_1 + h_n}, \quad \mu_n = 1 - \lambda_n = \frac{h_n}{h_1 + h_n}$$

y

$$d_n = \frac{6}{h_1 + h_n}\left(\frac{y_1 - y_n}{h_1} - \frac{y_n - y_{n-1}}{h_n}\right)$$

obtenemos

$$\mu_n M_{n-1} + 2M_n + \lambda_n M_1 = d_n$$

Después de todo este contexto teórico ya estamos en condiciones de aplicar este conocimiento en Python utilizando la función `CubicSpline` de la biblioteca `scipy.interpolate`.

La función `CubicSpline` nos permite crear una interpolación cúbica natural a partir de un conjunto de puntos dados. Esta función puede utilizarse para calcular valores interpolados dentro del rango de los puntos originales y también para obtener los coeficientes de los polinomios cúbicos que aproximan la función en segmentos.

Veamos un ejemplo de uso:

```
import numpy as np
from scipy.interpolate import CubicSpline
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([12,11,13,39,58,77])
cs = CubicSpline(x,y,bc_type="natural")
# Para ver los coeficientes
cs.c
```

```
## array([[-1.05263158e+00,  8.26315789e+00, -1.10000000e+01,
##          4.73684211e+00, -9.47368421e-01],
##        [ 0.00000000e+00, -3.15789474e+00,  2.16315789e+01,
##         -1.13684211e+01,  2.84210526e+00],
##        [ 5.26315789e-02, -3.10526316e+00,  1.53684211e+01,
##          2.56315789e+01,  1.71052632e+01],
##        [ 1.20000000e+01,  1.10000000e+01,  1.30000000e+01,
##          3.90000000e+01,  5.80000000e+01]])
```

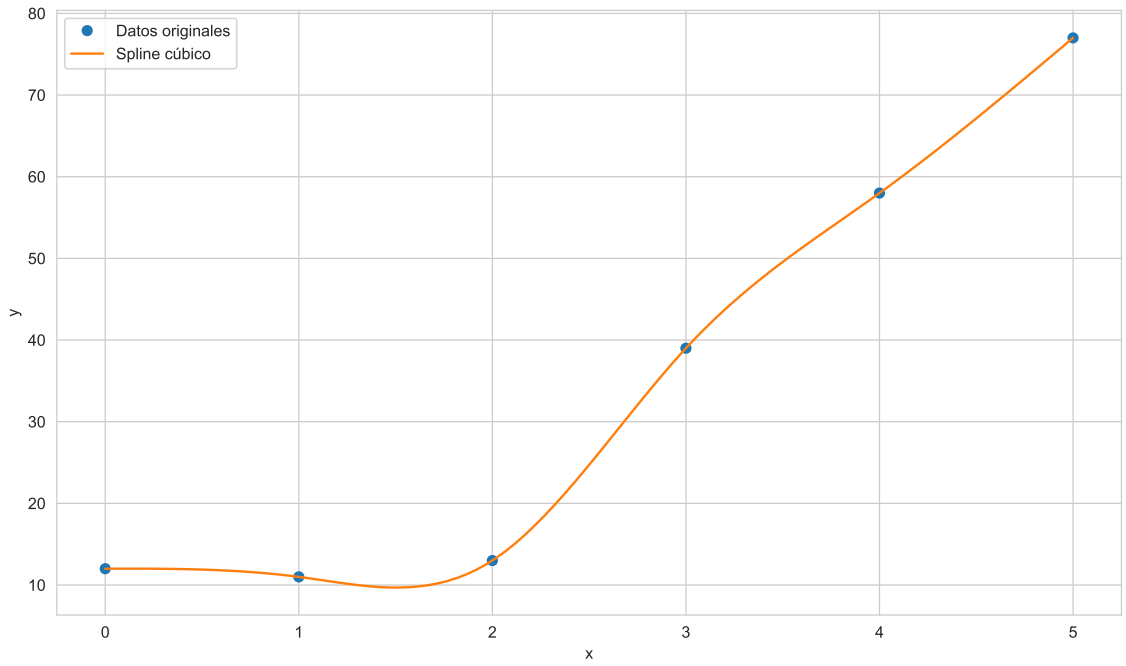

Figura 3.2: Resultado spline cúbico

Quedando nuestro spline como:

$$S_\Delta(y,x) = \begin{cases} 12.0 + 0.0526(x-0) + 0.0(x-0)^2 - 1.0526(x-0)^3, & \text{si } 0 \leq x \leq 1 \\ 11.0 - 3.1053(x-1) - 3.1579(x-1)^2 + 8.2632(x-1)^3, & \text{si } 1 < x \leq 2 \\ 25.6316 + 0.0526(x-2) - 3.1053(x-2)^2 + 15.3684(x-2)^3, & \text{si } 2 < x \leq 3 \\ 39.0 + 25.6315(x-3) - 11.3684(x-3)^2 + 47.3684(x-3)^3, & \text{si } 3 < x \leq 4 \\ 58.0 + 17.1053(x-4) + 2.8421(x-4)^2 - 0.9474(x-4)^3, & \text{si } 4 < x \leq 5 \end{cases}$$

3.2. Convergencia en la interpolación por funciones spline

Se ha observado que en la interpolación de Lagrange, los polinomios de interpolación no convergen hacia la función a medida que las particiones se hacen arbitrariamente finas (consultar referencia). Por el contrario, los splines convergen uniformemente hacia la función interpolada bajo hipótesis bastante generales.

Definición 3.2 Sea $\Delta = a = x_0 < x_1 < \cdots < x_n = b$ una partición del intervalo $[a,b]$. Definimos el *diámetro* de Δ como la cantidad

$$\varrho(\Delta) = \max_{0 \leq j \leq n-1} |x_{j+1} - x_j|.$$

Observación 3.3 Si los puntos de la partición Δ están equiespaciados, entonces $\varrho(\Delta) = h$.

Teorema 3.1 Sea $\Delta = a = x_0 < x_1 < \cdots < x_n = b$ una partición de $[a,b]$ con diámetro $\varrho(\Delta)$ que cumple la condición

$$\frac{\varrho(\Delta)}{|x_{j+1} - x_j|} \leq \mathrm{K}$$

para $j = 0, 1, \ldots, n-1$. Supongamos que $f \in \mathcal{C}^4([a,b])$ y satisface

$$|f^{(iv)}(x)| \leq \mathrm{L}, \quad \text{para todo } x \in [a,b],$$

y sea S_Δ la función spline cúbica que interpola a f en los puntos $x_0, x_1, \ldots, x_n$ con las condiciones

$$S'\Delta(y,a) = f'(a) \quad \text{y} \quad S'\Delta(y,b) = f'(b).$$

Entonces, existen constantes $0 \leq c_i \leq 2$ (independientes de Δ para $i = 0,1,2,3$) tales que para todo $x \in [a,b]$, se tiene

$$\left| f^{(i)}(x) - S_\Delta^{(i)}(x) \right| \leq c_i \mathrm{LK}(\varrho(\Delta))^{4-i}$$

para $i = 0,1,2,3$.

Observación 3.4

1. El parámetro K representa la medida de *uniformidad* de la partición.
2. En particular, si los puntos de la partición están equiespaciados, se puede elegir K=1, lo que resulta en
$$\left| f^{(i)}(x) - S_\Delta^{(i)}(x) \right| \leq c_i \mathrm{L} h^{4-i}$$
para todo $x \in [a,b]$ e $i = 0,1,2,3$.
3. Si $\Delta_k = \{a = x_0^{(k)} < x_1^{(k)} < \cdots < x_n^{(k)} = b\}$, $k \in \mathbb{N}$, son particiones del intervalo $[a,b]$ que cumplen
$$\lim_{k \to +\infty} \varrho(\Delta_k) = 0 \quad \text{y} \quad \sup_k \left\{ \max_j \frac{\varrho(\Delta_k)}{|x_{j+1}^{(k)} - x_j^{(k)}|} \right\} \leq \mathrm{K},$$
entonces las correspondientes funciones spline S_{Δ_k} que interpolan a f en los puntos de Δ_k y satisfacen
$$S'_{\Delta_k}(a) = f'(a) \quad \text{y} \quad S'_{\Delta_k}(b) = f'(b)$$
convergen uniformemente a f en $[a,b]$. Además, sus tres primeras derivadas convergen uniformemente a las derivadas respectivas de f.

A modo de ejemplo, intentaremos replicar el logo de la empresa Cicar con seis funciones spline de tipo I. El diseño de este logo fue obra de César Manrique, un destacado artista, arquitecto y defensor del medio ambiente de Lanzarote, que fue precursor de la idea de un turismo responsable que valorara y protegiera el entorno natural, promoviendo la sostenibilidad y la preservación del patrimonio cultural de las Islas Canarias.

Figura 3.3: Logo Cicar

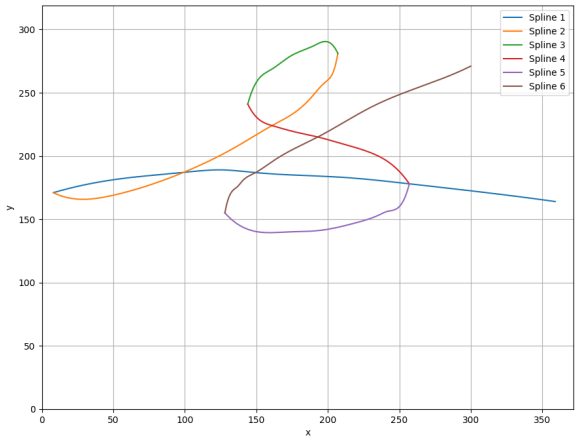

Figura 3.4: Logo Cicar con splines cúbicos

Tabla 3.1: Conjuntos de datos para gráfico Cicar

x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4	x_5	y_5	x_6	y_6
359	164	8	171	144	241	144	241	257	178	128	155
296	173	24	166	148	254	149	232	253	165	131	168
257	178	50	169	160	268	161	224	241	156	137	176
212	183	99	187	174	279	177	219	237	154	140	180
148	187	137	208	190	287	193	215	233	152	148	186
126	189	161	224	202	289	209	210	219	147	163	197
99	187	181	238	207	281	229	203	193	141	193	215
55	182	189	247			245	193	172	140	231	239
29	177	196	256			257	178	142	143	275	259
8	171	200	260					128	155	300	271

Capítulo 4

Normal tabulada

Se dice que una variable aleatoria sigue una distribución normal estándar si su función de densidad viene dada por

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{\frac{-z^2}{2}}.$$

A esta variable aleatoria se le denota usualmente por $Z \sim N(0,1)$. Su función de distribución viene dada por

$$F(z_1) = \int_{-\infty}^{z_1} f(t)\,dt = \int_{-\infty}^{z_1} \frac{1}{\sqrt{2\pi}} e^{\frac{-t^2}{2}}\,dt.$$

Gráficamente:

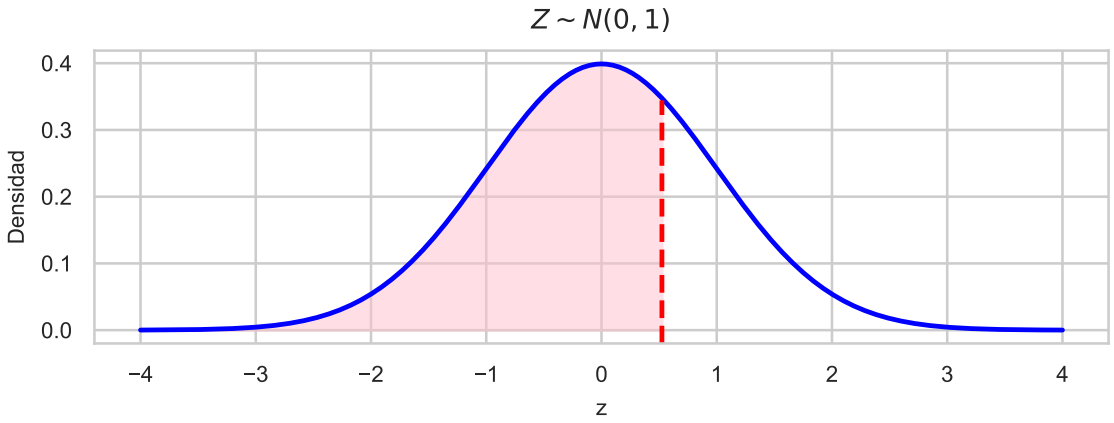

Figura 4.1: Normal estándar

Como podemos observar, esta integral no tiene una forma analítica simple, de hecho no existe forma de encontrar su primitiva. Es por esto que esta distribución se encuentra tabulada. En este apartado veremos una de las diversas formas para tabular esta distribución.

Como era de esperar, usaremos métodos numéricos para resolver esta cuestión, más especificamente, la regla de *Simpson 1/3*.

Teorema 4.1 Sea $f \in \mathcal{C}^4([a,b])$, $h = \dfrac{b-a}{2}$ y $x_i = a + ih$ con $i = 0, 1, 2$. La *fórmula de Simpson 1/3* está dada por:

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) \tag{4.1}$$

Demostración 4.1 Sea $P_2(x)$ el polinomio de interpolación de f en los puntos $\{x_0 = a, x_1 = \dfrac{a+b}{2}, x_2 = b\}$, dado por:

$$\begin{aligned} P_2(x) &= f(x_0) + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] \\ &= f(x_0) + (x - x_0)\frac{f(x_1) - f(x_0)}{h} + (x - x_0)(x - x_1)\frac{f(x_0) - 2f(x_1) + f(x_2)}{2h^2}. \end{aligned}$$

Integrando entre a y b, obtenemos

$$\int_a^b P_2(x)\,dx = \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right),$$

y teniendo en mente que $f(x) \approx P_2(x)$, deducimos

$$\int_a^b f(x)\,dx \approx \int_a^b P_2(x)\,dx = \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right),$$

obteniendo la fórmula (4.1). □

Teorema 4.2 Consideremos $f \in \mathcal{C}^4([a,b]), h = \dfrac{b-a}{2m}$ con $m \in \mathbb{N}$ y $x_i = a + ih$, $i = 0, 1, \ldots, 2m$. La *regla de Simpson compuesta* con m subintervalos se escribe por

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}\left(f(a) + 4\sum_{i=1}^{m} f(x_{2i-1}) + 2\sum_{i=1}^{m-1} f(x_{2i}) + f(b)\right), \tag{4.2}$$

donde el error cometido en esta aproximación viene dado por

$$R_{(a,b)}(f) = -(b-a)\frac{h^4}{180}f^{iv)}(\theta),$$

para $\theta \in [a,b]$.

Demostración 4.2 Como tenemos m intervalos de la forma

$$[x_{2(i-1)}, x_{2i}]$$

para $i = 1, 2, \ldots, m$, si aplicamos en cada intervalo $[x_{2(i-1)}, x_{2i}]$ la fórmula de Simpson y denotamos por

$$S = \frac{h}{3}\left(f(a) + 4\sum_{i=1}^{m} f(x_{2i-1}) + 2\sum_{i=1}^{m-1} f(x_{2i}) + f(b)\right),$$

obtenemos

$$\begin{aligned}\int_a^b f(x)\,dx &= \sum_{i=1}^{m}\int_{x_{2(i-1)}}^{x_{2i}} f(x)\,dx \\ &= \frac{h}{3}\sum_{i=1}^{m}\left(f(x_{2i-2})+4f(x_{2i-1})+f(x_{2i})\right) - \frac{h^5}{90}\sum_{i=1}^{m} f^{iv)}(\theta_i) \\ &= S - \frac{b-a}{m}\frac{h^4}{180}\sum_{i=1}^{m} f^{iv)}(\theta_i) \\ &= S - (b-a)\frac{h^4}{180}f^{iv)}(\theta),\end{aligned}$$

donde hemos aplicado el teorema de los Valores Intermedios a la función $f^{iv)}$. □

Por tanto, podemos aproximar el valor de nuestra función de distribución. Por ejemplo, supongamos que queremos saber $\mathbb{P}(1,2 \leq Z \leq 2,5)$. Podemos obtener este valor con la fórmula (4.2). Usaremos $m = 5$, es decir, que dividimos el intervalo en 10 subintervalos con puntos equiespaciados $\{1.2 = x_0, x_1, \ldots, x_9, x_{10} = 2.5\}$

$$\begin{aligned}\int_{1.2}^{2.5} f(x)\,dx \approx \frac{0.13}{3}(&f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+4f(x_5)+2f(x_6)+ \\ &4f(x_7)+2f(x_8)+4f(x_9)+f(x_{10})) \approx 0.108892\end{aligned}$$

Mientras con la tabla de la normal, podríamos usar

$$\int_{1.2}^{2.5} f(x)\,dx = F(x)\Big]_{1.2}^{2.5} = F(2.5) - F(1.2) = 0,99379 - 0,88493 = 0.10886$$

La fórmula utilizada por Python para calcular la función de densidad de probabilidad de una distribución normal es parte de la biblioteca SciPy. La implementación específica de esta fórmula se encuentra en el archivo `ndtr.h`, ubicado en el repositorio de SciPy en GitHub [19]. La función `norm.cdf` utiliza el cálculo de la función de distribución acumulativa normal, que a su vez utiliza la función de error (`erf`) para evaluar la integral.

Veamos que relación guardan ambas funciones:

Teorema 4.3 Sea $F(x)$ la función de distribución de la normal estándar y sea $\operatorname{erf}(x)$ la función de error dada por $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$. Se tiene la siguiente relación entre $F(x)$ y erf:

$$F(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right).$$

Demostración 4.3 Partimos de $F(x)$ y hacemos un cambio de variable:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2}\, dt = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(\frac{t}{\sqrt{2}})^2}\, dt = \left\{ \begin{array}{l} z = \dfrac{t}{\sqrt{2}} \Rightarrow dz = \dfrac{1}{\sqrt{2}}\, dt \\ \Rightarrow \sqrt{2}\, dz = dt \end{array} \right\} =$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\frac{x}{\sqrt{2}}} e^{-z^2} \sqrt{2}\, dz = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\frac{x}{\sqrt{2}}} e^{-z^2}\, dz = \frac{1}{\sqrt{\pi}} \left(\int_{-\infty}^{0} e^{-z^2}\, dz + \int_{0}^{\frac{x}{\sqrt{2}}} e^{-z^2}\, dz \right) =$$

$$= \frac{1}{\sqrt{\pi}} \left(\int_{-\infty}^{0} e^{-z^2}\, dz + \frac{\sqrt{\pi}}{2} \underbrace{\frac{2}{\sqrt{\pi}} \int_{0}^{\frac{x}{\sqrt{2}}} e^{-z^2}\, dz}_{\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)} \right) = \frac{1}{\sqrt{\pi}} \left(\underbrace{\int_{-\infty}^{0} e^{-z^2}\, dz}_{(**)} + \frac{\sqrt{\pi}}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \right).$$

Ahora solo nos queda calcular (**), para ello usaremos la propiedad:

$$\int_{-\infty}^{0} e^{-z^2}\, dz = \frac{1}{2} \int_{-\infty}^{+\infty} e^{-z^2}\, dz.$$

Observamos que usando el *Teorema de Fubini* obtenemos la siguiente relación:

$$\iint_{\mathbb{R}^2} e^{-x^2-y^2}\, dx\, dy = \int_{-\infty}^{+\infty} \left(\int_{-\infty}^{+\infty} e^{-x^2-y^2}\, dy \right) dx = \left(\int_{-\infty}^{+\infty} e^{-x^2}\, dx \right) \left(\int_{-\infty}^{+\infty} e^{-y^2}\, dy \right) =$$
$$= \left(\int_{-\infty}^{+\infty} e^{-z^2}\, dz \right)^2$$

Y, por otro lado, usando coordenadas polares, con $r \geq 0$ y $\theta \in [0, 2\pi)$:

$$\begin{cases} x = r\, \cos\theta, \\ y = r\, \operatorname{sen}\theta, \end{cases}$$

obtenemos

$$\iint_{\mathbb{R}^2} e^{-x^2-y^2}\, dx\, dy = \int_{0}^{+\infty} \int_{0}^{2\pi} r \cdot e^{-r^2}\, d\theta\, dr = \int_{0}^{2\pi} 1\, d\theta \int_{0}^{+\infty} r \cdot e^{-r^2}\, dr =$$
$$= 2\pi \left(-\frac{1}{2} \right) \int_{0}^{+\infty} -2r \cdot e^{-r^2}\, dr = 2\pi \left(-\frac{1}{2} \right) \left[e^{-r^2} \right]_{r=0}^{r=+\infty} = \pi$$

Teniendo en cuenta esto deducimos que:

$$\iint_{\mathbb{R}^2} e^{-x^2-y^2}\, dx\, dy = \pi = \left(\int_{-\infty}^{+\infty} e^{-z^2}\, dz \right)^2 \Rightarrow \int_{-\infty}^{+\infty} e^{-z^2}\, dz = \sqrt{\pi}$$

Y por tanto:

$$\int_{-\infty}^{0} e^{-z^2}\, dz = \frac{1}{2} \int_{-\infty}^{+\infty} e^{-z^2}\, dz = \frac{1}{2} \sqrt{\pi}$$

Y concluimos que:

$$F(x) = \frac{1}{\sqrt{\pi}} \left(\int_{-\infty}^{0} e^{-z^2}\, dz + \frac{\sqrt{\pi}}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}} \right) \right) = \frac{1}{\sqrt{\pi}} \left(\frac{\sqrt{\pi}}{2} + \frac{\sqrt{\pi}}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}} \right) \right)$$
$$= \frac{1}{2} \left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}} \right) \right)$$

Que era lo que queríamos demostrar. □

La función de error (`erf`) está definida en el código base de Python como:

$$\text{erf}(x) = x \cdot \frac{P_4(x^2)}{Q_5(x^2)}$$

donde P_4 y Q_5 son polinomios cuyos coeficientes están definidos en el script, y de los cuales no tenemos más información de la manera exacta en que se calculan se calculan estos coeficientes no está explícitamente documentada en el código fuente. Es por esto que decidimos hacer nuestra propia función.

Como ya comentamos anteriormente, no podemos calcular la primitiva de la función. Pero sí podemos aproximarla usando el desarrollo de Taylor en 0 (también llamado el desarrollo de McLaurin). Para ello, consideramos $g(x) = e^{-x^2}$, donde el desarrollo de Taylor en 0 está dado por:

$$g(x) = g(0) + g'(0) + \frac{g''(0)}{2} + \frac{g'''(0)}{3!} + \dots$$

Teniendo en cuenta que:

$$\begin{aligned} &g(0) = 1 \\ &g'(x) = -2xe^{-x^2} \Rightarrow g'(0) = 0 \\ &g''(x) = -2e^{-x^2} + 4x^2e^{-x^2} \Rightarrow g''(0) = -2 \\ &\vdots \end{aligned}$$

obtenemos:

$$\begin{aligned} &g(x) = 1 - \frac{2}{2}x^2 + \dots \Rightarrow e^{-x^2} = 1 - x^2 + \dots \\ &\Rightarrow \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt = \frac{2}{\sqrt{\pi}} \int_0^x (1 - t^2 + \dots)\, dt \\ &= \frac{2}{\sqrt{\pi}} \left[t - \frac{t^3}{3} + \dots \right]_{t=0}^{t=x} = \frac{2}{\sqrt{\pi}} \left(x - \frac{x^3}{3} + \dots \right) \end{aligned}$$

Esta expresión es válida cerca del cero, es decir $\forall x \in (0, 1)$. Para hacer cálculos iterativos (como usa Python) se usan fórmulas alternativas a la anterior, por ejemplo, usando [21] podemos usar la fórmula alternativa:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{x}{(2n+1)} \prod_{k=1}^{n} \frac{-x^2}{k}$$

Teniendo en cuenta todo el contexto anterior, ya podemos crear nuestra propia función. Se encuentra en el Apendice A, bajo el nombre Función “erf” y Función “norm_acum”:

```
norm_acum(2)
```

```
## 0.9772498680518207
```

Veamos la función de Python:

```
import scipy
scipy.stats.norm.cdf(2)
```

```
## 0.9772498680518208
```

Muy parecidas. Sigamos con el ejemplo anterior:

$$\mathbb{P}(1.2 \leq Z \leq 2.5) = \mathbb{P}(Z \leq 2.5) - \mathbb{P}(Z \leq 1.2)$$

```
norm_acum(2.5)-norm_acum(1.2)
```

```
## 0.10886000489593228
```

Exactamente igual que el resultado que obtuvimos con la tabal de la normal. Veamos en cuanto a rendimiento:

```
import numpy as np
from scipy.stats import norm
import timeit

def time_norm_cdf():
    x = np.random.randn(10000)
    result = norm.cdf(x)

def time_norm_acum():
    x = np.random.randn(10000)
    result = norm_acum(x)

time_scipy = timeit.timeit(time_norm_cdf, number=100)
time_custom = timeit.timeit(time_norm_acum, number=100)
```

```
## Tiempo promedio para norm.cdf (SciPy): 0.060967 segundos
```

```
## Tiempo promedio para norm_acum: 2.322185 segundos
```

Mucho más eficiente la base.

Capítulo 5

Regresión spline

Un modelo de regresión se representa como

$$Y = f(X_1, X_2, \ldots, X_n) + \varepsilon$$

donde Y es la *variable de respuesta*, $X_1, X_2, \ldots, X_n$ son las *variables predictoras*, y ε representa el error o la discrepancia entre nuestro modelo y los valores reales. El objetivo principal de cualquier modelo de regresión es minimizar este error (ε) para todos los valores de y, evitando la introducción de variables aleatorias innecesarias. En este capítulo, nos centramos exclusivamente en la relación entre una única variable predictora y la variable de respuesta para mantener la simplicidad, utilizando X en lugar de X_1 con el fin de evitar confusiones. Así, escribimos nuestros modelos de regresión de la forma

$$Y = f(X) + \varepsilon$$

donde $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)^T$ es un vector de errores aleatorios no correlados que tienen media cero y varianza común σ^2 y f es una función de regresión desconocida.

Si tras analizar nuestros datos determinamos que la relación no es lineal, sino posiblemente un polinomio de orden superior, podemos expandir nuestra base agregando términos como x^2 o x^3 al modelo, lo cual implica la incorporación de nuevas columnas en la matriz de diseño X y de parámetros asociados (β_2 y β_3) en el vector de coeficientes b. Sin embargo, incluso esta ampliación sigue siendo limitante y computacionalmente demandante. Si los datos presentan valores absolutos o puntos no diferenciables, ninguno de estos enfoques logrará ajustar adecuadamente el modelo. Ya sabemos que muchas funciones de valor absoluto pueden ser aproximadas de manera efectiva utilizando funciones a trozos. Por ejemplo, $f(x) = |x|$, que se define como:

$$f(x) = \begin{cases} x & : x > 0 \\ -x & : x < 0 \end{cases}$$

Aunque existen métodos para transformar los datos y ajustar modelos lineales simples o de orden superior, como dividir el conjunto de datos en subconjuntos o asumir simetría, lo ideal sería contar con un modelo de regresión único que no requiera modificaciones en los datos originales. Por ejemplo, la función $|x|$ puede ser aproximada usando dos líneas unidas en el punto $x = 0$. Afortunadamente, existe un método llamado *regresión spline*,

que utiliza funciones polinómicas continuas por partes. Matemáticamente, un *spline de grado n* es una función continua compuesta por múltiples polinomios de grado n que se unen suavemente a través de una serie de puntos. Los puntos de unión de estos polinomios se conocen como "nodos", ya que conectan las funciones en una curva continua. Este método permite ajustar modelos más flexibles a conjuntos de datos complejos.

En lugar de ajustar un polinomio de alto grado sobre todo el rango de X, la regresión polinómica por partes implica ajustar polinomios de bajo grado por separado sobre diferentes regiones de X. Lo más usual es usar un polinomio cúbico por partes:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \varepsilon_i,$$

donde los coeficientes β_0, β_1, β_2 y β_3 difieren en diferentes partes del rango de X. Los puntos donde los coeficientes cambian se llaman nodos (knots). De forma matricial:

$$Y = X\beta^T + \varepsilon$$

donde:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}, \quad \text{y} \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

Por ejemplo, un polinomio cúbico por partes con un solo nodo en un punto c toma la forma:

$$\begin{cases} \beta_{01} + \beta_{11} x_i + \beta_{21} x_i^2 + \beta_{31} x_i^3 + \varepsilon_i & \text{si } x_i < c \\ \beta_{02} + \beta_{12} x_i + \beta_{22} x_i^2 + \beta_{32} x_i^3 + \varepsilon_i & \text{si } x_i \geq c. \end{cases}$$

En otras palabras, ajustamos dos funciones polinómicas diferentes a los datos, una en el subconjunto de observaciones con $x_i < c$, y otra en el subconjunto de observaciones con $x_i \geq c$. La primera función polinómica tiene coeficientes β_{01}, β_{11}, β_{21} y β_{31}, y la segunda tiene coeficientes β_{02}, β_{12}, β_{22} y β_{32}. Cada una de estas funciones polinómicas puede ajustarse utilizando mínimos cuadrados aplicados a funciones simples del predictor original.

Las regresiones por splines que acabamos de ver en la sección anterior pueden parecer algo complejas. Un spline cúbico con nodos en ξ_k para $k = 1, \ldots, K$ también puede ser representado por:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \cdots + \beta_{K+3} b_{K+3}(x_i) + \varepsilon_i$$

La manera más directa de representar un spline cúbico mediante esta formulación es iniciar con una base para un polinomio cúbico, es decir, X, X^2 y X^3, y luego agregar *funciones de base de potencia truncada*, denotadas por b_i. Es decir:

$$\begin{aligned} b_1(X) &= X^1 \\ b_2(X) &= X^2 \\ b_3(X) &= X^3 \\ b_{(3+k)}(X) &= (X - \xi_k)_+^3, \ k = 1, \ldots, K \end{aligned}$$

donde:

$$h(X, \xi_k) = (X - \xi_k)_+^3 = \begin{cases} (X - \xi_k)^3 & \text{si } X > \xi_k \\ 0 & \text{en caso contrario} \end{cases}$$

En otras palabras, para ajustar un spline cúbico a un conjunto de datos con K nodos, realizamos una regresión por mínimos cuadrados con una intersección y $3 + K$ predictores, en la forma X, X^2, X^3, $h(X, \xi_1), h(X, \xi_2), \ldots, h(X, \xi_k)$, donde $\xi_1, \ldots, \xi_k$ son los nodos. Esto equivale a estimar un total de $K + 4$ coeficientes de regresión, es por esto que ajustar un spline cúbico con K nodos utiliza $K + 4$ grados de libertad.

Para un spline cúbico con K nodos, se busca ajustar una curva suave y flexible mediante múltiples segmentos de polinomios cúbicos conectados en los puntos de los nodos. La suavidad de la curva se logra imponiendo la continuidad en las derivadas hasta el segundo orden en cada nodo, como comentamos en el capítulo (3.1).

Una estrategia común consiste en ubicar más nodos en regiones donde se espera que la función varíe rápidamente, y menos nodos en zonas de mayor estabilidad. Aunque esta estrategia puede ser efectiva, en la práctica es común colocar los nodos de manera uniforme o basada en percentiles. Más adelante, exploraremos cómo llevar a cabo esta colocación de nodos en Python.

Para evaluar la eficacia y la validez de un modelo de regresión, es esencial utilizar medidas de evaluación. A continuación, exploraremos las medidas más comúnmente utilizadas.

El coeficiente de determinación, o R^2, es una medida del ajuste del modelo. Puede variar en un rango de 0 a 1, y se define como:

$$R^2 = \frac{SS_R}{SS_T} = 1 - \frac{SS_{Res}}{SS_T}$$

La suma de cuadrados residual (SS_{Res}) se define como la suma de los cuadrados de las diferencias entre los valores observados y los valores predichos por el modelo. Matemáticamente, si tenemos nn observaciones con respuestas $y_1, y_2, \ldots, y_n$ y las predicciones del modelo para estas observaciones son $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ entonces la suma de cuadrados residual (SS_{Res}) se calcula como:

$$SS_{Res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

La suma de cuadrados de regresión (SS_R) se define como la suma de los cuadrados de las diferencias entre los valores predichos por el modelo y la media de los valores observados. Entonces la suma de cuadrados de regresión (SS_R) se calcula como:

$$SS_R = \sum_{i=1}^{n} (\hat{y}_i - \overline{y})^2$$

La suma total de cuadrados corregida (SS_T) se define como la suma de los cuadrados de las diferencias entre cada valor observado y la media de todos los valores observados. Entonces la suma total de cuadrados corregida (SS_T) se calcula como:

$$SS_T = \sum_{i=1}^{n} (y_i - \overline{y})^2$$

Se puede demostrar que la suma total de cuadrados corregida (SS_T) es igual a la suma de cuadrados residual (SS_{Res}) más la suma de cuadrados de regresión (SS_R). Matemáticamente, esta relación se expresa como:

$$SS_T = SS_{Res} + SS_R$$

Otra medida muy usada es el AIC (Criterio de Información de Akaike), un criterio utilizado para seleccionar entre modelos estadísticos. Su objetivo es encontrar un equilibrio entre la bondad de ajuste del modelo y su complejidad, evitando el sobreajuste.

$$AIC = 2k - \log L$$

donde:

- k es el número de parámetros en el modelo.
- L es la función de verosimilitud del modelo evaluada en los datos.

Este criterio favorece modelos que tienen un buen ajuste (reflejado en una alta verosimilitud) pero penaliza aquellos modelos que son más complejos (tienen más parámetros).

Después de revisar el contexto teórico sobre la regresión spline cúbica, estamos ahora listos para aplicar este método utilizando Python. Vamos a utilizar el conocido conjunto de datos de Boston para realizar una regresión spline cúbica. Este conjunto de datos contiene información sobre propiedades inmobiliarias en Boston y nos permitirá explorar cómo la regresión spline cúbica se ajusta a estos datos. Usaremos las variables:

- lstat: Porcentaje de población de bajo estatus socioeconómico.
- medv: Valor mediano de las viviendas ocupadas por sus propietarios en miles de dólares.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from patsy import dmatrix
import statsmodels.api as sm
from statsmodels.datasets import get_rdataset
# Cargar el conjunto de datos Boston Housing
boston_data = get_rdataset('Boston', package='MASS').data
# Preparar los datos
datos = pd.DataFrame(boston_data)
x = datos['lstat']
y = datos['medv']
# Definir el rango de nodos (puntos de corte) que deseas probar
max_nodes = 10  # Número máximo de nodos a probar
nodes_range = np.arange(3, max_nodes + 1)  # De 3 a 10 nodos
r_squared_list = []
# Ajustar modelos de regresión spline para diferentes números de nodos
for num_nodes in nodes_range:
    # Calcular los percentiles para los nodos
    cuts = np.percentile(x, np.linspace(0, 100, num_nodes))
    # Diseñar la matriz de diseño para el modelo spline
    x_design = dmatrix(
      f"bs(x, knots=cuts, degree=3, include_intercept=True)",
      {"x": x}, return_type='dataframe')
    # Ajustar el modelo de regresión spline
    model = sm.OLS(y, x_design).fit()
    # Calcular el coeficiente de determinación (R^2)
    r_squared = model.rsquared
    # Almacenar los resultados en la lista
    r_squared_list.append(r_squared)
```

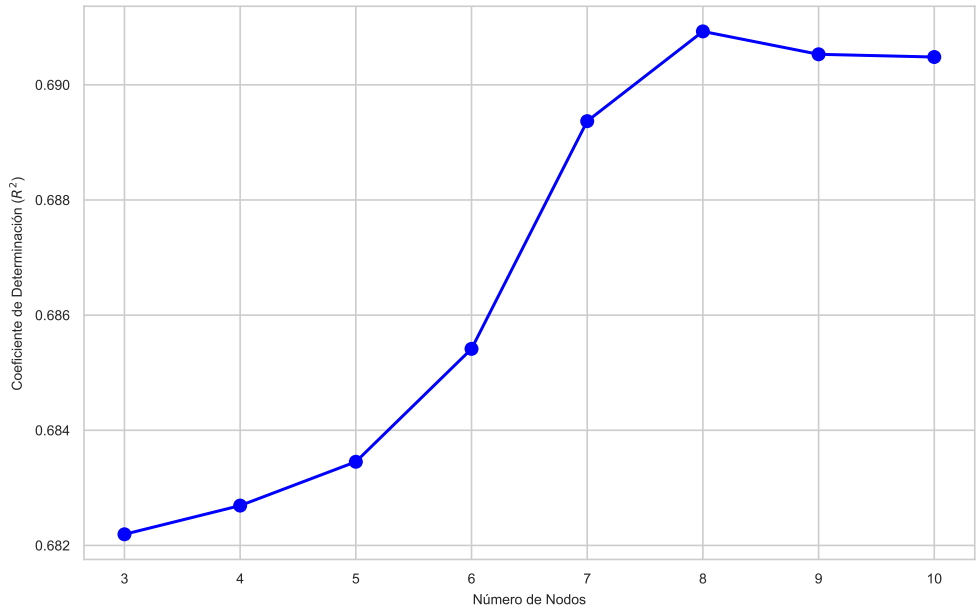

Figura 5.1: Rendimiento del Modelo Spline en función del Número de Nodos

Como podemos ver en el gráfico, la diferencia en el coeficiente de determinación R^2 entre el modelo con 3 nodos (0.682) y el modelo con 8 nodos (0.691) es mínima, apenas 0.009 puntos. Esta pequeña diferencia en el rendimiento no justifica la mayor complejidad del modelo con 8 nodos. Es por esto que optamos por el modelo más simple. Ajustemos de nuevo el modelo:

```
# Calcular los percentiles (1er cuartil, mediana y 3er cuartil)
cuts = np.percentile(x, [25, 50, 75])
# Ajustar el modelo de regresión spline cúbico
x_design = sm.add_constant(
  dmatrix("bs(x, knots=cuts, degree=3, include_intercept=True)",
  {"x": x}, return_type='dataframe'))
model_spline = sm.OLS(datos['medv'], x_design).fit()
# Obtener valores de x para la predicción (de 0 a 45)
sort_x = np.linspace(0, 45, 100)
sort_x_design = sm.add_constant(
  dmatrix("bs(x, knots=cuts, degree=3, include_intercept=True)",
  {"x": sort_x}, return_type='dataframe'))
# Predicción y intervalo de confianza
pred_spline = model_spline.get_prediction(sort_x_design)
pred_mean_spline = pred_spline.predicted_mean
pred_ci_spline = pred_spline.conf_int(alpha=0.05)  # IC 95%
```

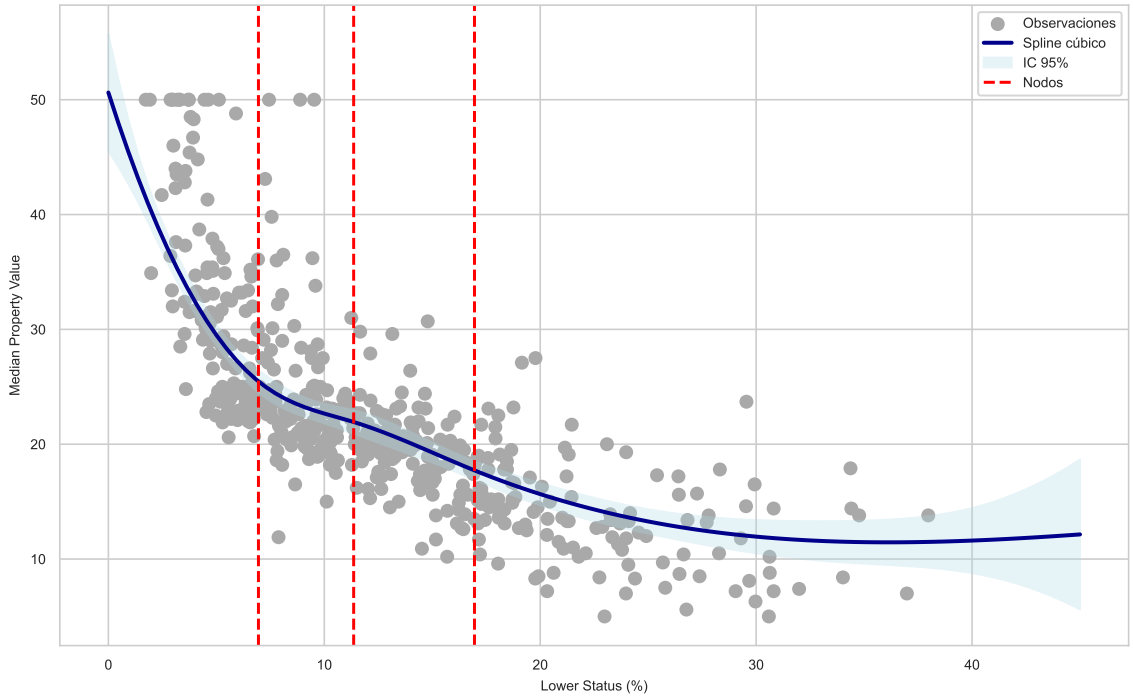

Figura 5.2: Spline cúbico con intervalo de confianza

Veamos un "cuadro resumen" de nuestro modelo:

```
summary_spline = model_spline.summary()
summary_text = summary_spline.tables[0]
print(summary_text)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   medv   R-squared:                       0.683
Model:                            OLS   Adj. R-squared:                  0.680
Method:                 Least Squares   F-statistic:                     179.5
Date:                Mon, 06 May 2024   Prob (F-statistic):          3.68e-121
Time:                        17:05:20   Log-Likelihood:                -1549.3
No. Observations:                 506   AIC:                             3113.
Df Residuals:                     499   BIC:                             3142.
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
```

El método utilizado para ajustar el modelo es el de mínimos cuadrados ordinarios (OLS), que es común en análisis de regresión para estimar los parámetros del modelo.

El coeficiente de determinación (R^2) es aproximadamente 0.683. Esto significa que alrededor del 68.3 % de la variabilidad en la variable de respuesta (medv) puede ser explicada por el modelo spline cúbico ajustado.

El estadístico F es 179.5, con un p-valor (Prob F-statistic) prácticamente nulo, lo que indica que al menos uno de los predictores en el modelo tiene efecto significativo sobre la variable de respuesta.

Ahora vamos a desarrollar un modelo lineal simple para compararlo con los modelos spline cúbico y de regresión lineal que hemos explorado anteriormente. El objetivo es evaluar cómo se comporta este modelo más sencillo en comparación con las aproximaciones más complejas que hemos utilizado hasta ahora.

```
model_lm = sm.OLS(datos['medv'], sm.add_constant(x)).fit()
sort_x_lm = np.linspace(0, 45, 100)
sort_x_with_intercept = sm.add_constant(sort_x_lm)
pred_lm = model_lm.get_prediction(sort_x_with_intercept)
pred_mean_lm = pred_lm.predicted_mean
pred_ci_lm = pred_lm.conf_int(alpha=0.05)
```

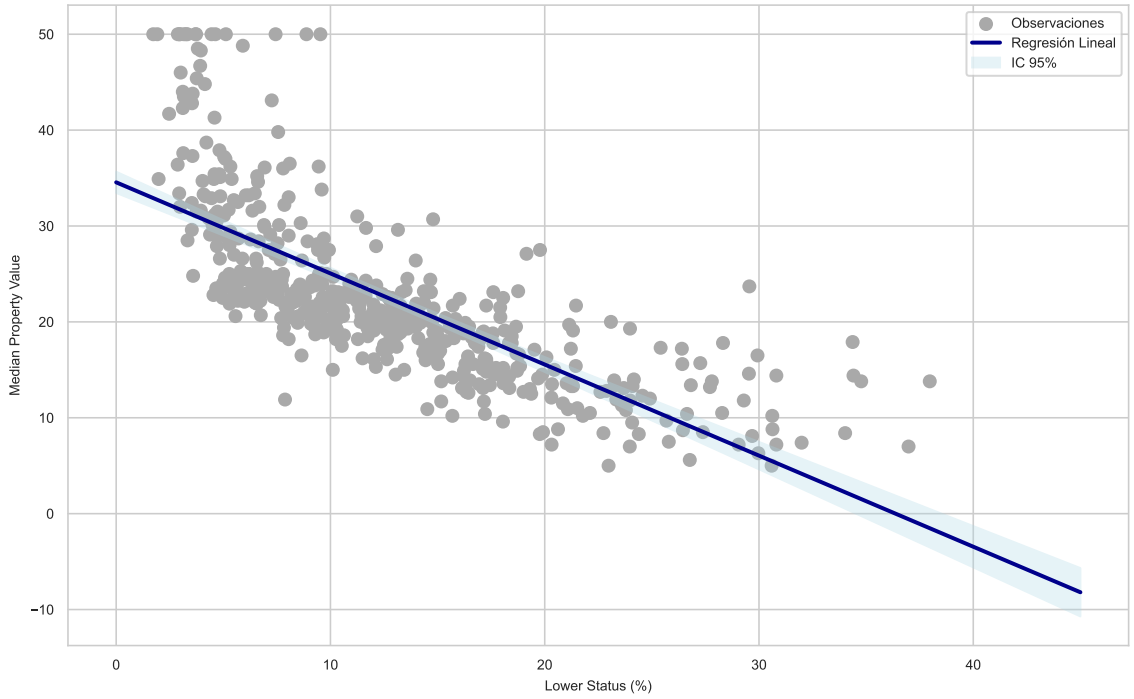

Figura 5.3: Regresión lineal con Intervalo de Confianza

```
summary_lm = model_lm.summary()
summary_lm_text = summary_lm.tables[0]
print(summary_lm_text)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   medv   R-squared:                       0.544
Model:                            OLS   Adj. R-squared:                  0.543
Method:                 Least Squares   F-statistic:                     601.6
Date:                Mon, 06 May 2024   Prob (F-statistic):           5.08e-88
Time:                        17:05:22   Log-Likelihood:                -1641.5
No. Observations:                 506   AIC:                             3287.
Df Residuals:                     504   BIC:                             3295.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
```

Gráficamente, podemos ver que nuestro modelo spline cúbico se ajusta mejor a los datos. Además, varias métricas confirman nuestra teoría, ya que el R^2 es superior (0.683 vs. 0.544) y tienen un AIC menor (3113 vs. 3287). En resumen, el modelo spline cúbico proporciona una forma flexible de modelar la relación entre lstat y medv, permitiendo capturar patrones no lineales y ajustarse mejor a la estructura subyacente de los datos.

Apéndice A

Apéndice: Funciones python

A.1. Clase “Interpolacion Lagrange”

```
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import seaborn as sns

class InterpolacionLagrange:
    def __init__(self, xi, fi):
        self.xi = xi
        self.fi = fi
        self.divisorL = np.zeros(len(xi), dtype=float)
        self.polinomio = None
        self.polinomio_simplificado = None

    def calcular_polinomio(self):
        n = len(self.xi)
        x = sym.Symbol("x")
        polinomio = 0

        for i in range(0, n, 1):
            numerador = 1
            denominador = 1
            for j in range(0, n, 1):
                if (j != i):
                    numerador *= (x - self.xi[j])
                    denominador *= (self.xi[i] - self.xi[j])
            terminoLi = numerador / denominador
            polinomio += terminoLi * self.fi[i]
            self.divisorL[i] = denominador

        self.polinomio = polinomio
        self.polinomio_simplificado = polinomio.expand()

    def graficar(self):
        if self.polinomio is None:
            self.calcular_polinomio()

        x = sym.Symbol("x")
        px = sym.lambdify(x, self.polinomio_simplificado)
```

```
        muestras = 200
        a = np.min(self.xi)
        b = np.max(self.xi)
        pxi = np.linspace(a, b, muestras)
        pfi = px(pxi)
        plt.plot(self.xi, self.fi, "o", label="Puntos")
        plt.plot(pxi, pfi, label="Polinomio", linestyle="--")
        plt.legend()
        plt.xlabel("$x$")
        plt.ylabel("$P_n (x)$")
        plt.grid(True)
        plt.show()

    def mostrar_polinomio(self):
        if self.polinomio is None:
            self.calcular_polinomio()

        print("Valores de fi: ", self.fi)
        print("Divisores en L(i): ", self.divisorL)
        print()
        print("Polinomio de Lagrange, expresiones:")
        print(self.polinomio)
        print()
        print("Polinomio de Lagrange: ")
        print(self.polinomio_simplificado)

    def imprimir_polinomio(self):
        ancho_max=70
        polinomio=self.polinomio_simplificado
        polinomio_string=polinomio.__str__()
        componentes = polinomio_string.split()
        lineas = []
        linea_actual = componentes[0]
        for componente in componentes[1:]:
            if len(linea_actual) + 1 + len(componente) <= ancho_max:
                linea_actual += " " + componente
            else:
                lineas.append(linea_actual)
                linea_actual = componente
        lineas.append(linea_actual)
        return print("\n".join(lineas))
```

A.2. Función “Cota error”

```
import numpy as np
from scipy.optimize import fminbound
from sympy import symbols, diff, factorial, lambdify

def acotar_error(f, n, a, b):
    puntos = np.linspace(a, b, n)
    x = symbols("x")

    producto = 1
    for punto in puntos:
        producto *= (x - punto)
```

```
    producto_func = lambdify(x, producto, modules="numpy")

    resultado_maximo_productorio=fminbound(lambda x: -abs(producto_func(x)), a, b)
    valor_absoluto_productorio=abs(producto_func(resultado_maximo_productorio))

    derivada_n = diff(f(x), x,n)
    derivada_n_func = lambdify(x, derivada_n)
    resultado_maximo_f=fminbound(lambda x: -abs(derivada_n_func(x)), a, b)
    valor_absoluto_f=abs(derivada_n_func(resultado_maximo_f))

    factorial_n = factorial(n)

    cota_error= (valor_absoluto_productorio/factorial_n)*valor_absoluto_f

    return cota_error
```

A.3. Función "Área entre dos funciones"

```
from scipy.integrate import quad

def calcular_area_entre_funciones(funcion1, funcion2, limite_inferior, limite_superior):
    def diferencia(x):
        return abs(funcion1(x) - funcion2(x))

    area, _ = quad(diferencia, limite_inferior, limite_superior)

    return area
```

A.4. Función "Interpolación Neville"

```
def interpolacion_neville(x, y, objetivo):
    n = len(x)
    Q = []
    for _ in range(n):
      fila = [0] * n
      Q.append(fila)
    for i in range(n):
        Q[i][0] = y[i]
    for i in range(1, n):
        for j in range(1, i + 1):
            Q[i][j] = ((objetivo - x[i - j]) * Q[i][j - 1] -
            (objetivo - x[i]) * Q[i - 1][j - 1]) / (x[i] - x[i - j])
    for fila in Q:
      print(fila)
    return Q
```

A.5. Función "Diferencias Newton"

```
import numpy as np
from tabulate import tabulate

def diferencias_newton(x, y):
    n = len(y)
```

```
    coef = np.zeros([n, n+3])
    coef[:,0] = np.arange(n)  # Índices i
    coef[:,1] = x             # Valores xi
    coef[:,2] = y             # Valores fi

    # Calcular las diferencias divididas
    for j in range(1,n):
        for i in range(n-j):
            coef[i][j+2] = (coef[i+1][j+1] - coef[i][j+1]) / (x[i+j]-x[i])

    # Imprimir la tabla utilizando tabulate
    coef=coef[:,:-1] #eliminamos la última columna, siempre es 0
    headers = ["i",'xi', 'fi'] + ['F[{}]'.format(i+1) for i in range(n-1)]
    tabla = np.vstack([headers] + coef.tolist())
    print(tabulate(tabla, headers='firstrow', tablefmt='grid'))
    return coef
```

A.6. Función "Puntos Chebychev"

```
def puntos_cheby(a, b, n):
    k = np.arange(1, n + 1)
    x = (a + b) / 2 + (b - a) / 2 * np.cos((2*k - 1) * np.pi / (2*n))
    return x
```

A.7. Función "erf"

```
import numpy as np

def erf(x, m=50):
    suma_erf = 0.0
    for n in range(m):
        primero = x / (2*n + 1)
        producto = 1.0
        for k in range(1, n+1):
            producto *= -(x**2) / k
        suma_erf += primero * producto

    erf_valor = (2 / np.sqrt(np.pi)) * suma_erf
    return erf_valor
```

A.8. Función "norm_acum"

```
import numpy as np

def norm_acum(x):
    n = 50
    return 0.5 * (1 + erf(x / np.sqrt(2)))
```

Bibliografía

[1] AcademiaLab «Fenómeno de runge».
https://academia-lab.com/enciclopedia/fenomeno-de-runge/.

[2] Burden, Richard L.; Faires, Douglas J. y Burden, Annette M. *Numerical analysis*. Cengage Learning, 10.ª edición. ISBN 978-1-305-25366-7. OCLC: ocn898154569.

[3] Campos Pinto, Martin «Approximation numérique des fonctions».

[4] Cody, By W. J. «Rational Chebyshev approximations for the error function».
https://www.semanticscholar.org/paper/Rational-Chebyshev-approximations-for-the-error-Cody/e02dcb90df0b0b2c55ec5185894ae60fa70bffd6.

[5] Cohen, Albert y Després, Bruno «Introduction à l'analyse numérique».

[6] D. Cook, John «Runge phenomenon for interpolation at evenly spaced nodes».
https://www.johndcook.com/blog/2017/11/18/runge-phenomena/.

[7] Doubova, Anna y Guillén González, Francisco *Un curso de cálculo numérico: Interpolación, Aproximación, Integración y Resolución de Ecuaciones Diferenciales*. Universidad de Sevilla. ISBN 978-84-472-0941-5.

[8] Encyclopedia of Mathematics «Spline interpolation».
https://encyclopediaofmath.org/wiki/Spline_interpolation.

[9] Fuentes, Justo «Exploracxión del Data Frame Boston del paquete MASS».
https://rpubs.com/justorfc/DatasetBoston.

[10] García Ramirez, José Antonio «Interpolacion lagrange, newton y hermite».
https://rpubs.com/Fou/356838.

[11] Griggs, Whitney «Penalized Spline Regression and its Applications».

[12] Habermann, Christian y Kindermann, Fabian «Multidimensional Spline Interpolation: Theory and Applications». **30(2)**, pp. 153–169. ISSN 1572-9974. doi: 10.1007/s10614-007-9092-4.
https://doi.org/10.1007/s10614-007-9092-4.

[13] Jackson, Samuel «Chapter 9 Splines | Machine Learning».
https://bookdown.org/ssjackson300/Machine-Learning-Lecture-Notes/splines.html.

[14] James, Gareth; Witten, Daniela; Hastie, Trevor; Tibshirani, Robert y Taylor, Jonathan E. *An introduction to statistical learning: with applications in Python*. Springer texts in statistics. Springer. ISBN 978-3-031-38747-0 978-3-031-38746-3.

[15] Luque Calvo, Pedro L. *Escribir un Trabajo Fin de Estudios con R Markdown*, 2017.
http://destio.us.es/calvo.

[16] Madhukar, Bhoomika «Hands-On Guide To Spline Regression».
https://analyticsindiamag.com/hands-on-guide-to-spline-regression/.

[17] Rey Cabezas, José María y Infante del Río, Juan Antonio *Métodos numéricos: Teoría, problemas y prácticas con MATLAB*. Pirámide, 4.ª edición. ISBN 978-84-368-4580-8.

[18] Rosario, Edison del «Interpolación polinómica de Lagrange con Python – Métodos numéricos». http://blog.espol.edu.ec/analisisnumerico/interpolacion-de-lagrange/.

[19] SciPy «Repositorio Github». https://github.com/scipy/scipy/blob/main/scipy/special/special/cephes/ndtr.h.

[20] Weisstein, Eric W. «Monic Polynomial». https://mathworld.wolfram.com/MonicPolynomial.html.

[21] Wikipedia «Función error». https://es.wikipedia.org/w/index.php?title=Funci%C3%B3n_error&oldid=159001549. Page Version ID: 159001549.

[22] Wolfram MathWorld «Lagrange Interpolating Polynomial». https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html.