

Proyecto Análisis Numérico

Rodrigo Castillo Camargo

Análisis Numérico

Universidad del Rosario

Parte 1 - Teoría de la aproximación

¿En qué consiste la teoría de la aproximación?

- Datos $m + 1$ puntos $(x_0, y_0), \dots, (x_m, y_m)$ encontrar una función que se aproxime lo suficiente.

Para ésto, existe el concepto de interpolación mediante el polinomio de Lagrange, sin embargo...

1. En la interpolación dados m puntos, necesitaremos un polinomio de grado $m - 1$
2. En la aproximación, si se tienen m puntos, el polinomio puede ser de grado n tal que $0 < n < m$

Esto porque la interpolación tiene mucha varianza, ya que retorna un polinomio que se acompla exactamente a los puntos que se le dan.

Idea general

$$Error = ||y - p(x)||$$

en donde se pueden definir dos tipos de normas:

1. $|| * ||$
 2. $|| * ||_1$
- La idea general de la aproximación es encontrar un polinomio que minimice el error, definiendo el error con las normas especificadas.

Método de mínimos cuadrados lineales: El método de mínimos cuadrados lineales es un algoritmo utilizado en teoría de la aproximación para encontrar la mejor aproximación lineal a un conjunto de datos. En términos generales, la idea es encontrar una línea recta que pase lo más cerca posible de todos los puntos de datos.

El método de mínimos cuadrados lineales se basa en el principio de que la mejor aproximación lineal es aquella que minimiza la suma de los cuadrados de las diferencias entre los valores reales y los valores predichos por la línea recta. Es decir, se busca la línea que minimiza la función:

Implementación:

```

def linear_regression(x, y):
    n = len(x)
    x_mean = sum(x) / n
    y_mean = sum(y) / n

    xy_sum = 0
    x_squared_sum = 0

    for i in range(n):
        xy_sum += x[i] * y[i]
        x_squared_sum += x[i] ** 2

    # Calcular los coeficientes de la línea recta
    b = (xy_sum - n * x_mean * y_mean) / (x_squared_sum - n * x_mean ** 2)
    a = y_mean - b * x_mean

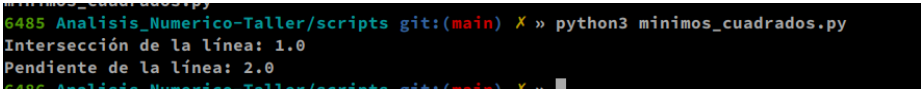
    return a, b

# Datos
x = [1, 2, 3, 4, 5]
y = [3, 5, 7, 9, 11]

# Calcular los coeficientes de la línea recta
a, b = linear_regression(x, y)

# Imprimir los coeficientes
print("Intersección de la línea:", a)
print("Pendiente de la línea:", b)

```



```

6485 Analisis_Numerico-Taller/scripts git:(main) X » python3 minimos_cuadrados.py
Intersección de la línea: 1.0
Pendiente de la línea: 2.0
6486 Analisis_Numerico-Taller/scripts git:(main) X »

```

Figure 1: resultado

Aproximación de grado n

- En una aproximación de grado n, se busca ajustar un polinomio de grado n a los datos observados. En lugar de una línea recta como en la regresión lineal (mínimos cuadrados lineales), se utiliza una función polinómica de grado n para modelar la relación entre las variables

Implementación

```

import numpy as np
def polynomial_regression(x, y, n):
    # Construir la matriz de diseño

```

```

X = []
for i in range(len(x)):
    row = [x[i]**j for j in range(n+1)]
    X.append(row)

# Convertir a arrays
X = np.array(X)
y = np.array(y)

# Calcular los coeficientes utilizando el método de mínimos cuadrados
X_transpose = np.transpose(X)
X_transpose_X = np.dot(X_transpose, X)
X_transpose_y = np.dot(X_transpose, y)
coeffs = np.linalg.solve(X_transpose_X, X_transpose_y)

return tuple(coeffs)

x = [0, 1, 2, 3, 4, 5]
y = [1, 3, 5, 4, 6, 8]

coeffs = polynomial_regression(x, y, 3)

print("Coeficientes del polinomio: ", coeffs)

```

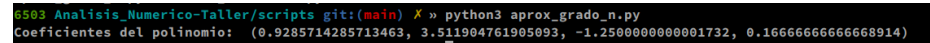


Figure 2: solución

Aproximaciones exponenciales

- En el análisis numérico y la teoría de aproximación, el algoritmo de aproximaciones exponenciales se refiere a un algoritmo utilizado para aproximar una función mediante una combinación lineal de funciones exponenciales.
- La idea básica detrás del método de aproximaciones exponenciales es que cualquier función suave puede aproximarse mediante una combinación lineal de funciones exponenciales de la forma:

$$f(x) \approx \sum_i c_i e^{\alpha_i x}$$

Implementación

```

import numpy as np

def exponential_approximation(x, y, n, alpha, c):

```

```

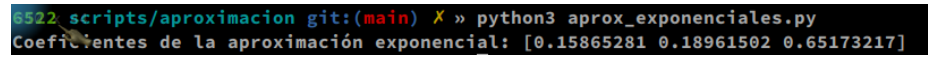
A = np.zeros((len(x), n+1))
for i in range(len(x)):
    A[i,0] = 1
    for j in range(n):
        A[i,j+1] = np.exp(alpha[j]*x[i])
c = np.linalg.lstsq(A, y, rcond=None)[0]
return c

# Datos de ejemplo
x = np.array([0, 1, 2])
y = np.array([1, 2, 5])
n = 2
alpha = np.array([-1, 1])
c = np.zeros(n+1)

# Calcular los coeficientes de la aproximación exponencial
c = exponential_approximation(x, y, n, alpha, c)

# Imprimir los coeficientes
print("Coeficientes de la aproximación exponencial:", c)

```



```

6522 scripts/aproximación git:(main) X » python3 aprox_exponenciales.py
Coeficientes de la aproximación exponencial: [0.15865281 0.18961502 0.65173217]

```

Figure 3: resultado

Series de Fourier

- El método de aproximación por series de Fourier es un método utilizado en la teoría de la aproximación para aproximar funciones periódicas mediante la suma de una serie de funciones sinusoidales y cosinusoidales.
- En particular, la serie de Fourier de una función $f(x)$ periódica con período $2L$ se define como:

$$f(x) \approx \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{n\pi x}{L}\right) + b_n \sin\left(\frac{n\pi x}{L}\right) \right]$$

Implementación

x Implementé una serie de Taylor con el grado del polinomio $n=5$

```

import math
import matplotlib.pyplot as plt

# Función a aproximar
def f(x):

```

```

    return math.sin(x) + math.sin(3*x) + math.sin(5*x)

# Coeficientes de Fourier
def a(n):
    if n == 0:
        return 2*math.pi
    else:
        return 0

def b(n):
    return 2*(-1)**n / (n*math.pi)

# Aproximación mediante la serie de Fourier
def fourier_series(x, n):
    s = a(0)/2
    for i in range(1, n+1):
        s += a(i) * math.cos(i*x) + b(i) * math.sin(i*x)
    return s

# Intervalo de evaluación
x_vals = [i*math.pi/100 for i in range(-200, 201)]

# Evaluar la función original
y_vals = [f(x) for x in x_vals]

# Evaluar la aproximación por la serie de Fourier
n = 5
y_fourier = [fourier_series(x, n) for x in x_vals]

# Graficar los resultados
plt.plot(x_vals, y_vals, label="Función original")
plt.plot(x_vals, y_fourier, label="Aproximación de Fourier")
plt.legend()
plt.show()

```

Polinomios ortogonales trigonométricos El método de aproximación mediante polinomios ortogonales trigonométricos es un método de la teoría de la aproximación que se utiliza para aproximar funciones periódicas. Se basa en la idea de que cualquier función periódica puede ser aproximada mediante una serie trigonométrica de la forma:

$$f(x) \approx a_0 + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)]$$

Donde los coeficientes a_n y b_n se calculan de la siguiente manera:

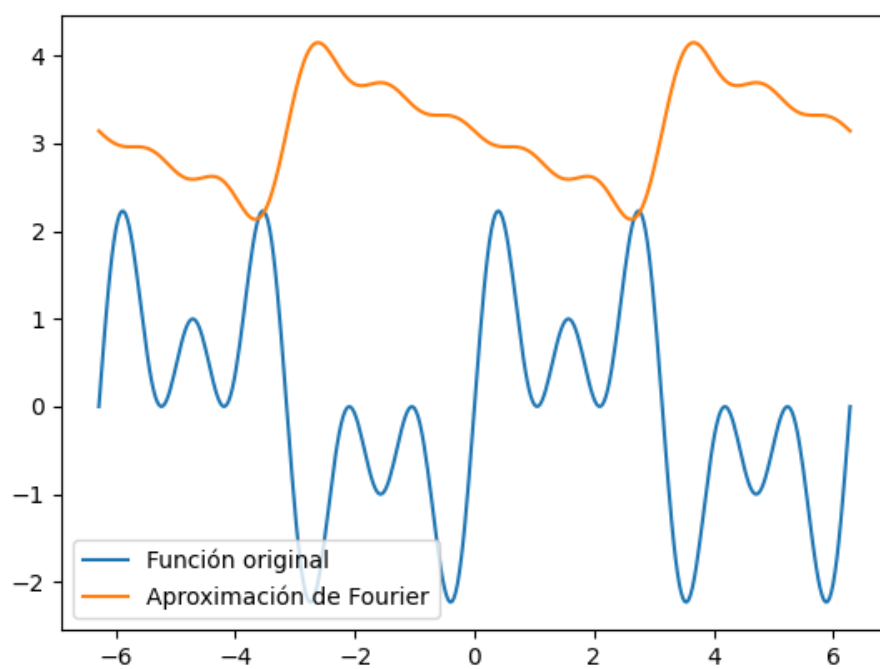


Figure 4: Grafica Series de Fourier

1. $a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx) dx$
2. $b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) dx$

Implementación:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import jn # Función Bessel de primera especie

def trigonometric_approximation(x, y, degree):
    n = len(x)
    a = np.zeros(degree+1)
    b = np.zeros(degree+1)
    for k in range(degree+1):
        a[k] = (2/n)*np.sum(y*np.cos(k*np.array(x)))
        b[k] = (2/n)*np.sum(y*np.sin(k*np.array(x)))
    p = np.zeros(n)
    for k in range(degree+1):
        p += a[k]*np.cos(k*np.array(x)) + b[k]*np.sin(k*np.array(x))
    c = np.zeros(degree+1)
    for k in range(degree+1):
        c[k] = np.sqrt(a[k]**2 + b[k]**2)
    return p, c

# Definir la función seno en el intervalo [0, pi]
x = np.linspace(0, np.pi, 100)
y = np.sin(x)

# Aproximación mediante polinomios ortogonales trigonométricos de grado 5
p, c = trigonometric_approximation(x, y, degree=5)

# Graficar la función original y la aproximación
plt.plot(x, y, label='Seno original')
plt.plot(x, p, label='Aproximación')
plt.legend()
plt.show()
```

Aproximación trigonométrica discreta: -El método de aproximación trigonométrica discreta es una técnica utilizada en teoría de aproximación para aproximar una función mediante una combinación lineal de funciones trigonométricas discretas. Se realiza utilizando la siguiente formula:

$$f(x) \approx a_0 + \sum_{k=1}^n [a_k \cos(kx) + b_k \sin(kx)]$$

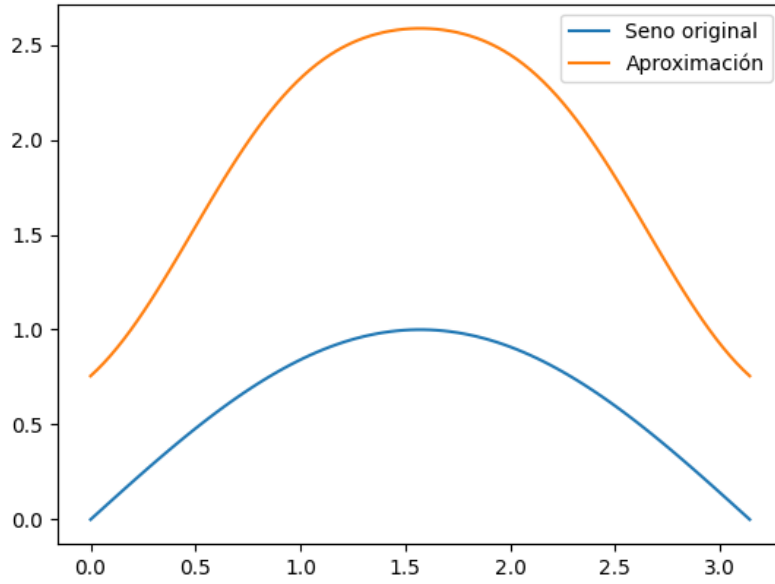


Figure 5: polinomios ortogonales

donde a_0 , a_k y b_k son coeficientes que se determinan a partir de la función que se desea aproximar y n es el número de términos utilizados en la aproximación.

Implementación

```
import numpy as np
import matplotlib.pyplot as plt

def DFT(f):
    N = len(f)
    c = np.zeros(N, dtype=complex)
    for k in range(N):
        for n in range(N):
            c[k] += f[n] * np.exp(-2j*np.pi*k*n/N)
    return c

def IDFT(c):
    N = len(c)
    f = np.zeros(N, dtype=complex)
    for n in range(N):
        for k in range(N):
            f[n] += c[k] * np.exp(2j*np.pi*k*n/N)
        f[n] /= N
    return f

def trig_approx(f, M):
    N = len(f)
    L = N/2
    x = np.linspace(-L, L, N+1)[: -1]
    c = DFT(f)
    cf = np.zeros(N, dtype=complex)
    cf[0] = c[0]/N
    for m in range(1, M+1):
        cf[m] = c[m]/N
        cf[-m] = c[-m]/N
    return lambda x: np.real(np.sum(cf[k] * np.exp(2j*np.pi*k*x/L) for k in range(-M, M+1)))

# Ejemplo de uso
f = lambda x: np.sin(x)
M = 2
N = 2*M+1
L = np.pi
x = np.linspace(-L, L, N)
f_vals = f(x)
f_approx = trig_approx(f_vals, M)
```

```
plt.plot(x, f_vals, label='Función original')
plt.plot(x, f_approx(x), label='Aproximación')
plt.legend()
plt.show()
```

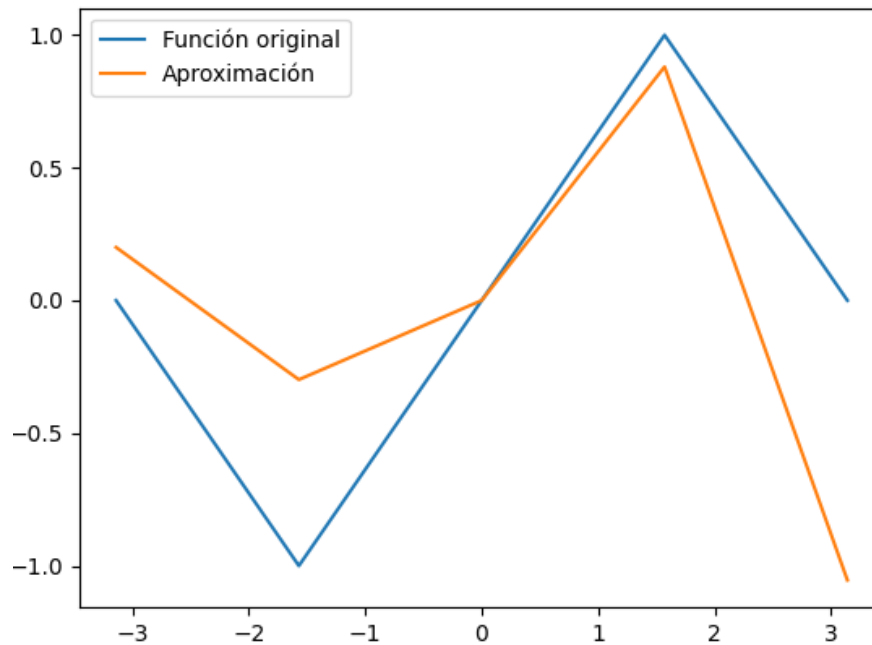


Figure 6: Aproximacion Trigonométrica discreta 2 términos

Transformada rápida de Fourier

- La Transformada Rápida de Fourier (FFT, por sus siglas en inglés) es un algoritmo que permite calcular la transformada de Fourier discreta (DFT) de una secuencia de datos de manera más rápida que utilizando el método directo de la DFT. Es un algoritmo de divide y vencerás

Implementación

```
import math
import matplotlib.pyplot as plt

def FFT(f):
    # Calcula la FFT de una señal f
    N = len(f)
    if N == 1:
        return f
```

```

else:
    Feven = FFT([f[i] for i in range(0, N, 2)])
    Fodd = FFT([f[i] for i in range(1, N, 2)])
    combined = [0] * N
    for m in range(N//2):
        combined[m] = Feven[m] + math.e**(-2j*math.pi*m/N)*Fodd[m]
        combined[m + N//2] = Feven[m] - math.e**(-2j*math.pi*m/N)*Fodd[m]
    return combined

# Crear un arreglo de puntos de prueba
t = [i/100 for i in range(20)]
f = [math.sin(2*math.pi*5*i) + math.sin(2*math.pi*10*i) for i in t]

# Calcular la transformada rápida de fourier de los puntos
f_fft = FFT(f)

# Calcular la frecuencia de cada muestra en los puntos original
freq = [i/(t[-1]-t[0]) for i in range(len(t))]

# Graficar los puntos original y los puntos reconstruidos de la FFT
f_ifft = [i.real for i in FFT(f_fft[::-1])] # Calcula la transformada inversa de Fourier
plt.plot(t, f, label='Señal original')
plt.plot(t, f_ifft, label='FFT reconstruida')
plt.legend()
plt.show()

```

Tabla de comparación de métodos:

Método	Ventajas	Desventajas	Bueno para	Malo para
Aproximación por espacios de funciones	Versatilidad en la elección de funciones base	La elección de las funciones puede ser complicada	Funciones complicadas	Datos con patrones no lineales
Aproximación por series de Fourier	Buena aproximación de funciones periódicas	Puede ser difícil de ajustar a funciones no periódicas	Funciones periódicas	Funciones no periódicas
Aproximación por series de Taylor	Aproximación exacta en un punto	Sólo es válido localmente, puede diverger lejos del punto	Funciones analíticas en un punto	Funciones no analíticas, singularidades

Método	Ventajas	Desventajas	Bueno para	Malo para
Aproximación mediante polinomios ortogonales trigonométricos	Convergencia uniforme rápida	La elección de los polinomios puede ser complicada	Funciones periódicas, datos con ruido	Funciones no periódicas
Aproximación trigonométrica discreta	Simple y eficiente	Puede tener dificultades para aproximar funciones complicadas	Funciones periódicas, datos con ruido	Funciones no periódicas
Transformada rápida de Fourier (FFT)	Eficiente para cálculo en grandes conjuntos de datos	Requiere una elección adecuada de la resolución	Análisis de señales, procesamiento de imágenes	Funciones no periódicas

Tabla de tiempos computacionales y memorias

Método	Aproximación	Big Oh	Memoria
Ajuste de mínimos cuadrados	Funciones arbitrarias	$O(n^3)$	$O(n^2)$
Polinomios de Taylor	Funciones analíticas	$O(n^2)$	$O(n)$
Serie de Fourier	Funciones periódicas	$O(n \log n)$	$O(n)$
Polinomios trigonométricos	Funciones periódicas	$O(n^2)$	$O(n)$
Transformada Rápida de Fourier	Funciones periódicas	$O(n \log n)$	$O(n)$

Parte 2 - Métodos iterativos.

¿En qué consisten los métodos iterativos?

Referencias:

1. Libro de la clase
2. Talleres del curso subidos a E-aulas

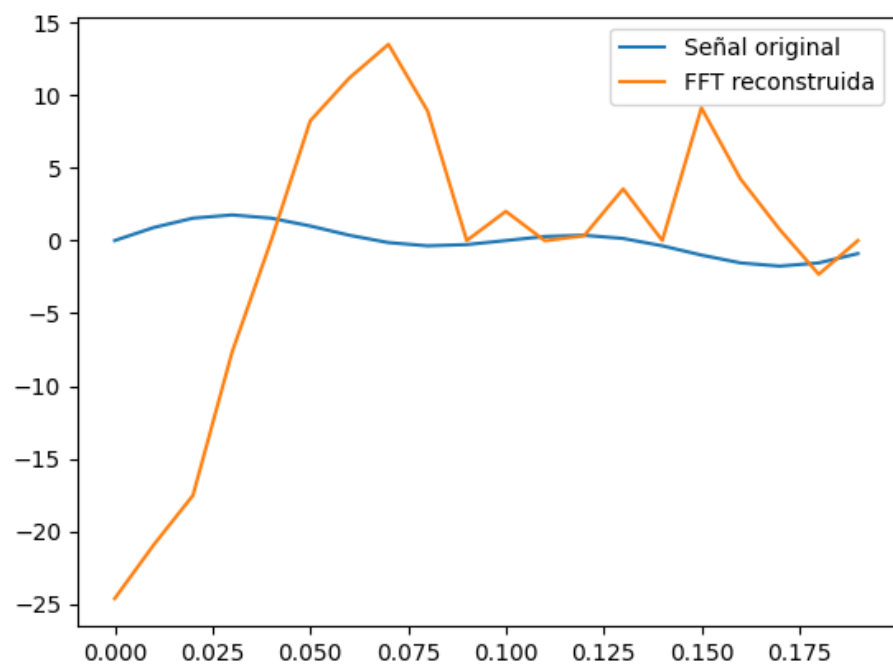


Figure 7: FFT2