T3_Interpolacion

May 30, 2023

1 Tarea 3 - Interpolación

1.0.1 Héctor Hibran Tapia Fernández - A01661114

Hecho en Google Colab:

[ ]:
```
!pip install numpy
!pip install scipy
!pip install matplotlib
```

Espacio para importar librerías:

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline as CS
```

1.1 Parte 1

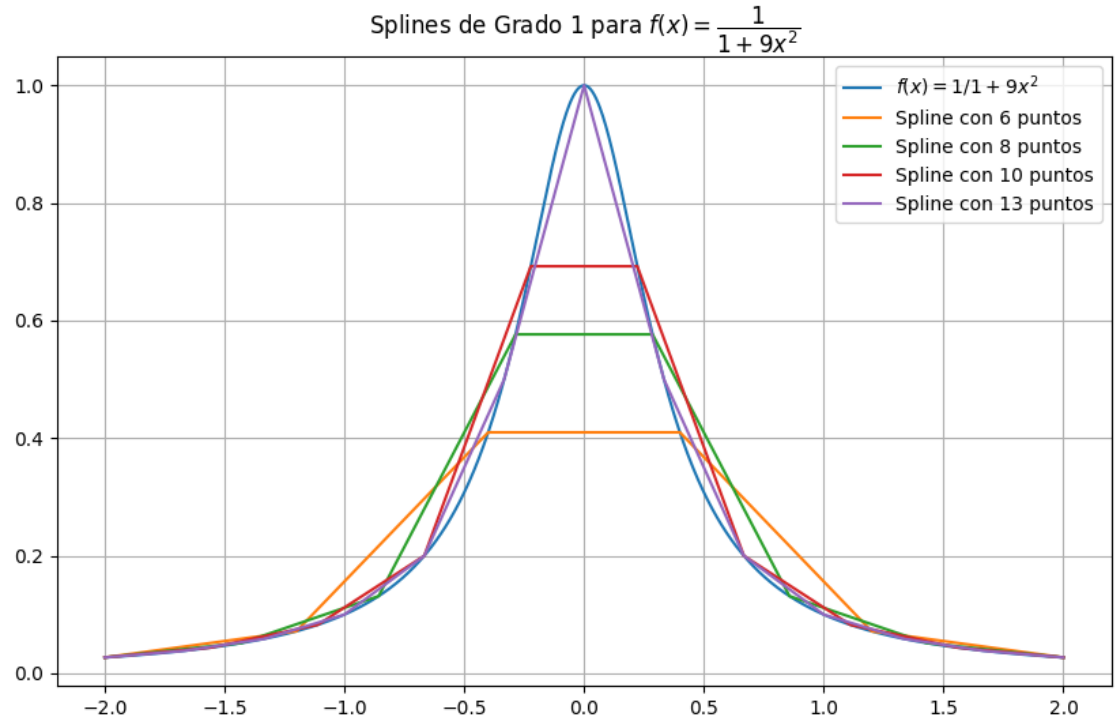
Considera la gráfica de la función $f : [-2, 2] \to R$ dada por $f(x) = \dfrac{1}{1 + 9x^2}$

---

1. Construye 6, 8, 10 y 13 puntos equiespaciados en su dominio (utiliza np.linspace(-2,2,n)) y dibuja gráficas de splines de grado 1 que interpolen tus puntos (esto lo puedes hacer gratis utilizando plt.plot). En este punto no te pido que des las funciones explícitas de los splines. Compara en una sola gráfica dichos splines con la gráfica original de la función.

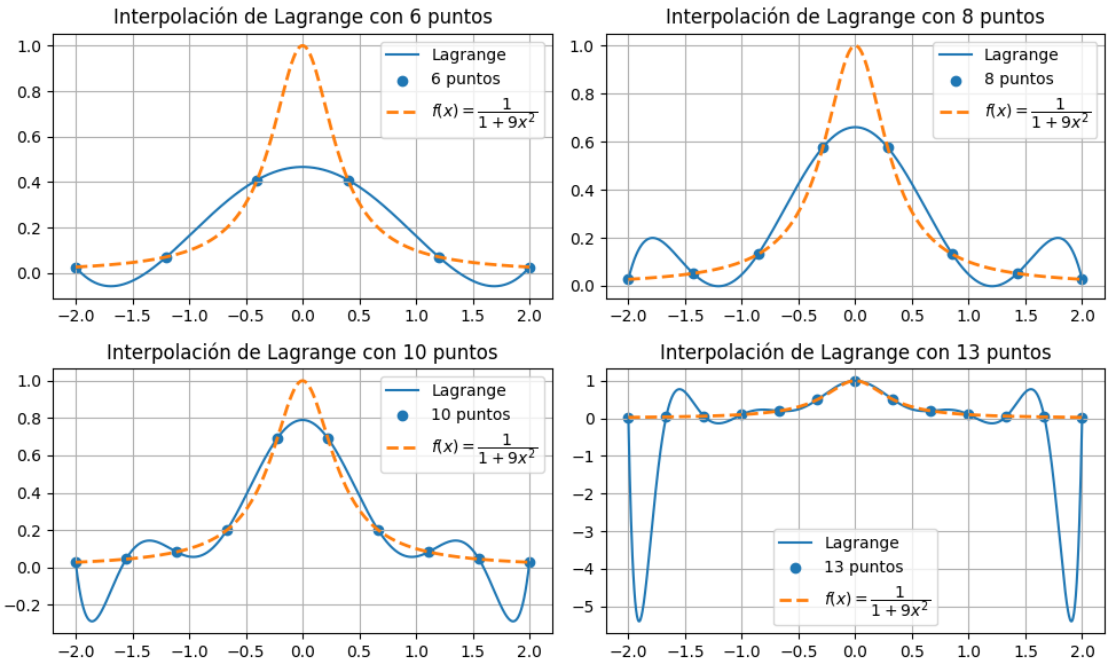
[3]:
```
# Iniciamos creando una función de python donde guardamos la función a plotear.
def funcion_a_plotear(x):
    return 1 / (1 + 9 * (x**2))

x = np.linspace(-2, 2, 1000) # Definimos el dominio, pero no de los splines␣
 ↪sino del canvas en el que python va a dibujar nuestras lineas.
y = funcion_a_plotear(x) # Aquí llamamos a la función de python donde guardamos␣
 ↪nuestra función a graficar.
```

```
# Definimos el figsize, y ploteamos la función, por ahora no hemos ploteado los␣
 ↪splines.
plt.figure(figsize = (10, 6))
plt.plot(x, y, label = '$f(x) = 1 / 1+9x^2$')


# GENERANDO SPLINES
# Generamos con un loop los splines de todos los puntos que pidió.
for n in [6, 8, 10, 13]: # Mientras más puntos tenga el spline, se parecerá más␣
 ↪a nuestra función.
    x_spline = np.linspace(-2, 2, n) # Aquí es donde aplicamos el dominio de␣
 ↪los splines. # n es el número de puntos que queremos en el spline.
    y_spline = funcion_a_plotear(x_spline) # Aquí volvemos a llamar a la␣
 ↪función de python f(x) y les aplica los valones que establecimos arriba.
    plt.plot(x_spline, y_spline, label = f'Spline con {n} puntos')

# Ploteamos y añadimos formato a la gráfica.
plt.title('Splines de Grado 1 para $f(x)=\dfrac{1}{1+9x^2}$')
plt.legend()
plt.grid(True)
plt.show()
```



2. Utiliza bases de Lagrange para calcular polinomios que pasen por todos los puntos del inciso anterior, debes calcular un polinomio para cada caso y graficarlos, nuevamente comparando con la gráfica de la función original.

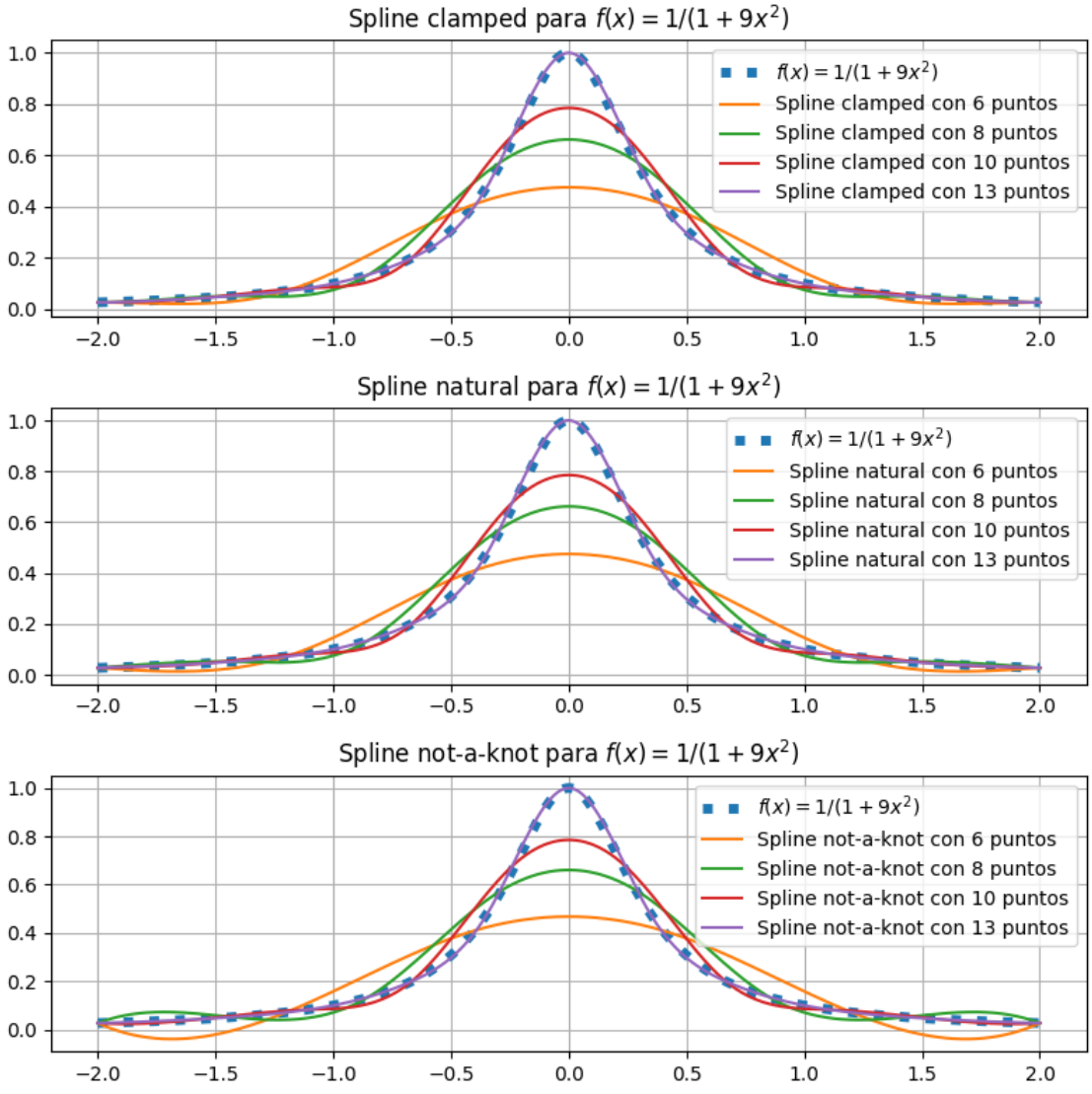
[4]:
```
def interpolacion_lagrange(puntos: np.ndarray, x: np.ndarray,␣
 ↪y_puntos_evaluados):

    l = [1] * len(puntos)

    for k in range(len(puntos)):
        for j in range(len(puntos)):
            if k != j:
                l[k] = l[k] * ((x - puntos[j]) / (puntos[k] - puntos[j]))

    resultado_final = y_puntos_evaluados[0] * l[0]

    for i in range(1, len(puntos)):
        resultado_final += y_puntos_evaluados[i] * l[i]

    return resultado_final


fig, axes = plt.subplots(2, 2, figsize=(10, 6))

for i, n in enumerate([6, 8, 10, 13]):
    x_spline = np.linspace(-2, 2, n)
    y_spline = funcion_a_plotear(x_spline)

    x_extra = np.linspace(-2, 2, 1000)
    lagrange = interpolacion_lagrange(x_spline, x_extra, y_spline)

    row = i // 2
    col = i % 2

    ax = axes[row, col]
    ax.plot(x_extra, lagrange, label = f"Lagrange")
    ax.scatter(x_spline, y_spline, label = f"{n} puntos")
    ax.plot(x, y, label = '$f(x)=\dfrac{1}{1+9x^2}$', linewidth = 2, linestyle␣
 ↪= '--')

    ax.set_title(f'Interpolación de Lagrange con {n} puntos')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```



3. Finalmente, utiliza la función CubicSpline para interpolar tus puntos con splines cúbicos. Haz una gráfica comparativa para cada tipo de spline (clamped, natural y not-a-knot).

[5]:
```
x = np.linspace(-2, 2, 1000)

fig, ax = plt.subplots(3, 1, figsize = (8, 8))

condiciones_limites = ['clamped', 'natural', 'not-a-knot']

for i, sp in enumerate(condiciones_limites):
    ax[i].plot(x, funcion_a_plotear(x), label = '$f(x) = 1 / (1+9x^2)$',␣
 ↪linewidth = 5, linestyle = ':')

    for n in [6, 8, 10, 13]:
        x_spline = np.linspace(-2, 2, n)
        y_spline = funcion_a_plotear(x_spline)

        if sp == 'clamped':
            cs = CS(x_spline, y_spline, bc_type = sp, extrapolate = True)
        elif sp == 'natural':
            cs = CS(x_spline, y_spline, bc_type = sp, extrapolate = True)
        elif sp == 'not-a-knot':
            cs = CS(x_spline, y_spline, bc_type = sp)

        y_spline_interp = cs(x)
```

```
        ax[i].plot(x, y_spline_interp, label = f'Spline {sp} con {n} puntos')

    ax[i].set_title(f'Spline {sp} para $f(x) = 1 / (1+9x^2)$')
    ax[i].legend()
    ax[i].grid(True)

plt.tight_layout()
plt.show()
```



4. Revisa la definición de cada tipo de spline y explica las diferencias entre ellos.

[6]:

```
# Clamped: Se dan los valores de la primera derivada en los extremos de la␣
 ↪función. Esto puede ser útil si se conoce la pendiente de la función en␣
 ↪estos puntos.

# Natural: La segunda derivada se establece en cero en los extremos del␣
 ↪conjunto de datos. Este termina teniendo el valor de una función de grado 1␣
 ↪en los bordes.

# Not-a-knot: La tercera derivada es continua en el segundo y penúltimo punto.␣
 ↪Es decir, el primer y el último segmento del spline forman un único␣
 ↪polinomio cúbico.
```

5. Basado en tu respuesta anterior ¿qué tipo de spline consideras más útil para interpolar puntos en la gráfica de f si ahora cambias el dominio a uno de la forma $[-r, r]$ con r grande.

[7]:
```
# Se puede notar en las orillas de cada spline sobre todo para el spline con 6␣
 ↪puntos, donde el spline not-a-knot se nota demasiado la variabilidad,␣
 ↪mientras que en el
# spline natural no tanto, pero el que tiene menos variabilidad es el spline␣
 ↪clamped, por lo tanto este sería el más útil.
```

1.2 Parte 2

Ahora considera la función $h : [-1, 1] \to R$ dada por $h(x) = \sqrt[4]{1 - x^4}$

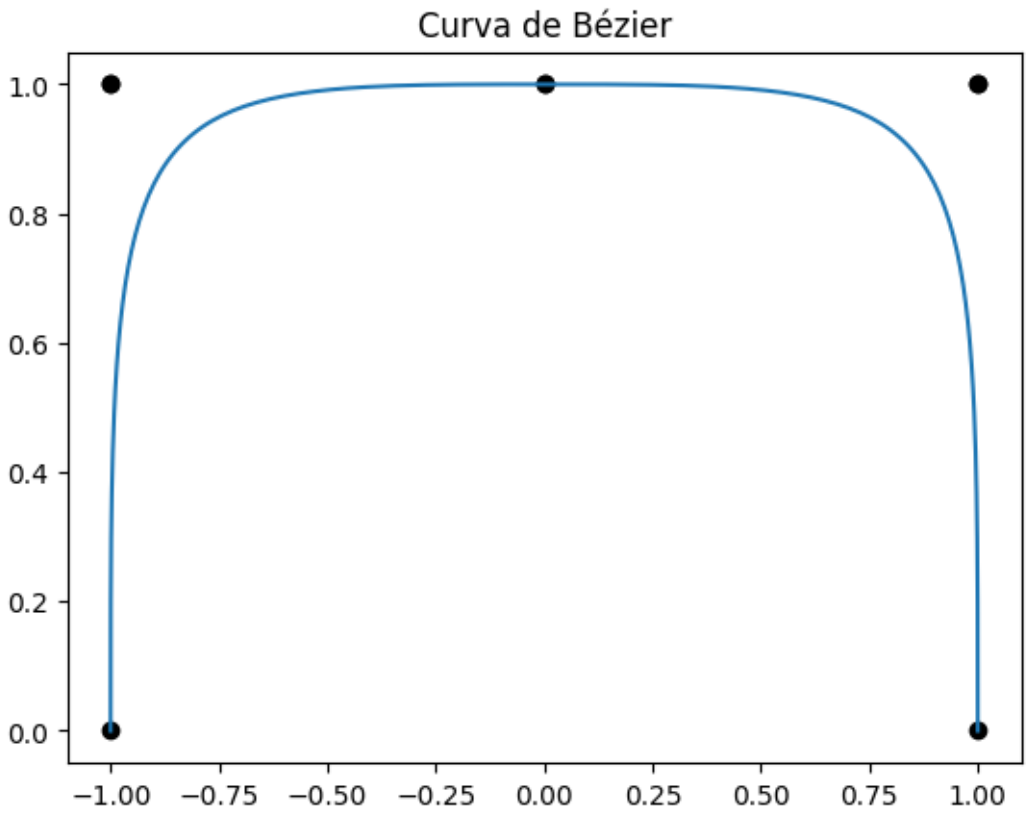
---

1. Considera los siguientes tres puntos en la gráfica de h : (-1,0), (0,1) y (1,0). Utiliza el algoritmo de De Casteljau para definir una curva racional que interpole los puntos anteriores. En clase te daré varias pistas para lograr una buena interpolación.

2. Grafica la curva de Bezier que obtuviste en el punto anterior. Como requisito, tu curva debe tener tangentes verticales en el punto inicial y final y tangente horizontal en el punto (0,1).

[8]:
```
h = 3
k = 3

puntos_control_izquierda = np.array([[-1, 0], [-1, k/3], [-h/3, 1], [0, 1]])
puntos_control_derecha = np.array([[0, 1], [h/3, 1], [1, k/3], [1, 0]])

t = np.linspace(0, 1, 100)

def calcular_puntos_curva_bezier(puntos_control, t):
    return (
        puntos_control[0]*(1-t)**3
```

```
        + puntos_control[1]*3*t*(1-t)**2
        + puntos_control[2]*3*(1-t)*t**2
        + puntos_control[3]*t**3
    )

coordenadas_bezier_x_izq =␣
 ↪calcular_puntos_curva_bezier(puntos_control_izquierda[:, 0], t)
coordenadas_bezier_y_izq =␣
 ↪calcular_puntos_curva_bezier(puntos_control_izquierda[:, 1], t)
coordenadas_bezier_x_der = calcular_puntos_curva_bezier(puntos_control_derecha[:
 ↪, 0], t)
coordenadas_bezier_y_der = calcular_puntos_curva_bezier(puntos_control_derecha[:
 ↪, 1], t)

coordenadas_bezier_x = np.concatenate((coordenadas_bezier_x_izq,␣
 ↪coordenadas_bezier_x_der))
coordenadas_bezier_y = np.concatenate((coordenadas_bezier_y_izq,␣
 ↪coordenadas_bezier_y_der))

plt.figure()
plt.plot(coordenadas_bezier_x, coordenadas_bezier_y)
puntos_control = np.concatenate((puntos_control_izquierda,␣
 ↪puntos_control_derecha))
plt.scatter(puntos_control[:, 0], puntos_control[:, 1], color='black')
plt.title("Curva de Bézier")
plt.show()
```


Curva de Bézier
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00