Universidad Nacional de Ingeniería
Escuela Profesional de Matemática
Ciclo 2021-3

[[**Análisis y Modelamiento Numérico I - CM4F1**]
[**Los profesores**] UNI, 24 de febrero de 2022

Práctica Calificada 4

1. (7 puntos) Dibuje la curva de Bézier B que tiene como puntos de control $P_0 = (1,0; 1,0)$, $P_1 = (2,0; 7,0)$, $P_2 = (8,0; 6,0)$ y $P_3 = (12,0; 2,0)$. Sobre la misma gráfica, trace el polígono de control. Halle el valor de B en $t = 0{,}25$. Implemente el algoritmo de Casteljau.

Solución: Siguiendo [3] pag 367. Sea la curva de Bézier de grado $n = 3$ con puntos de control

$$B(t) = \sum_{i=0}^{n} P_i b_{i,n}(t), \quad \text{donde } b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i},\ i \in \{0, \cdots, n\},\ t \in [0,1]$$
$$= (-7t^3 + 15t^2 + 3t + 1\,;\, 4t^3 - 21t^2 + 18t + 1).$$

Usando el algoritmo de Casteljau, obtenemos $B(t)$ de manera recursiva:

$$\beta_i^{(0)} := P_i,\ \ i \in \{0, 1, \cdots, n\}$$
$$\beta_i^{(j)} := \beta_i^{(j-1)}(1-t) + \beta_{i+1}^{(j-1)} t,\ \ i \in \{0, 1, \cdots, n-j\},\ j \in \{1, \cdots, n\}.$$

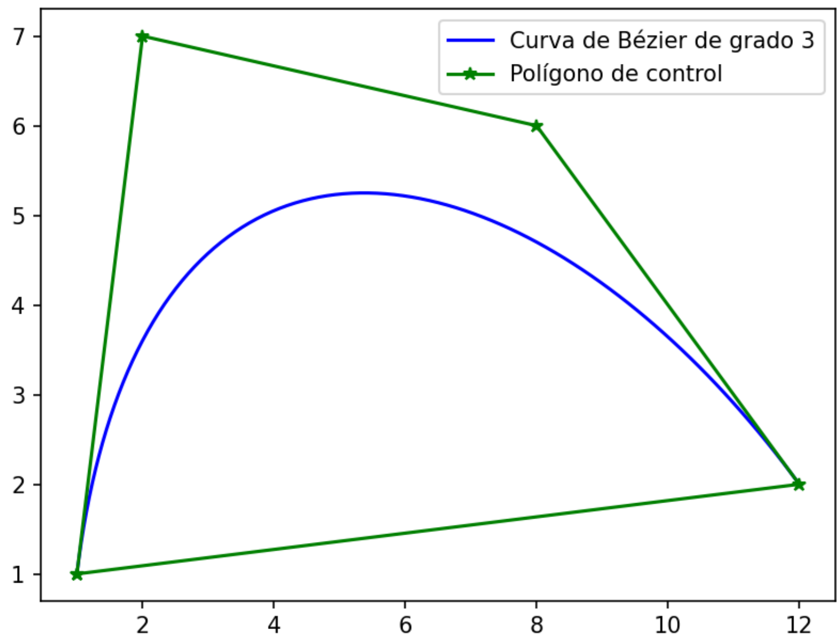
Finalmente, en el n-ésimo paso obtenemos el único coeficiente, que representa la función vectorial:

$$B(t) = \beta_0^{(n)}.$$

Los resultados obtenidos por el algoritmo, concuerda con la función de Bézier de grado 3:

```
Ecuación de la curva de Bézier:
B(t) = (-7*t**3 + 15*t**2 + 3*t + 1, 4*t**3 - 21*t**2 + 18*t + 1)

Polinomio de Bézier evaluado en t=0.25: (2.578125, 4.25)
```



```python
# Solucion 1
#Algoritmo de Casteljou

import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

def casteljau(t,coefs):
    beta = [c for c in coefs] # guarda todos los puntos
    t=sp.Symbol('t')
    n = len(beta)

    #Algoritmo de Casteljou
    for j in range(1, n):
        for k in range(n - j):
            beta[k] = beta[k] * (1 - t)+ (beta[k + 1] * t)

    beta[0][0] = beta[0][0].expand() # se guarda la primera ecuacion
    beta[0][1] = beta[0][1].expand() # se guarda la segunda ecucacion

    return beta[0][0],beta[0][1]

coefs=np.array([[1,1],[2,7],[8,6],[12,2]]) # puntos de control
puntos=np.linspace(0,1,100) #Puntos para la grafica
n = len(coefs)
t=sp.Symbol('t')

#Ecuacion de la curva
print("\nEcuacion de la curva de Bezier: ")
print("\nB(t) =",casteljau(t,coefs))

#Polinomio evaluado
px = sp.lambdify(t,casteljau(t,coefs))
p_x = px(0.25)
print("\nPolinomio de Bezier evaluado en t=0.25:", p_x)

#GRAFICA DE LA CURVA
p1,p2= casteljau(t,coefs)
px1 = sp.lambdify(t,p1)
px2 = sp.lambdify(t,p2)

plt.plot(px1(puntos),px2(puntos),
        color="blue",
        label="Curva de Bezier de grado " + str(n-1),
        )

#POLIGONO DE CONTROL
list_x= []
list_y = []

for i in range(0,n):
    list_x.append(coefs[i][0])
    list_y.append(coefs[i][1])
list_x.append(coefs[0][0])
list_y.append(coefs[0][1])

plt.plot(list_x,list_y, marker = '*', color="green", label="Poligono de control")

plt.legend()
plt.show()
```

2. (7 puntos) Las funciones $f(x) = 1/(1+x^2)$ y $g(x) = e^{-x^2}$ tienen una apariencia similar. ¿Se comportan de manera similar en el proceso de interpolación para nodos igualmente espaciados?.

(Sug. Utilice la forma de Lagrange del polinomio de interpolación, considerando $5, 10, 20, 40$ y 80 nodos sobre el intervalo $[-1, 1]$. Halle el error de aproximación con la norma del supremo)

Solución: Se espera que los polinomios de interpolación p_n de grado n convergen a la función f que interpola sobre un intervalo $[a, b]$. Es decir

$$\|f - p_n\|_\infty = \max_{a \leq x \leq} |f(x) - p_n(x)|$$

converge a 0 cuando $n \to \infty$. Pero, en general los polinomios de interpolación no convergen a la función que interpolan, salvo para cierta elección de puntos de interpolación (Véase Teorema 7 del capítulo 6 de [2]). A continuación consideramos mallados uniformes del intervalo $[-1, 1]$, es decir los puntos $x_i = -1 + \frac{2i}{n}$ para $i \in \{1, \cdots, n\}$ y $n \in \{5, 10, 20, 40, 80\}$.

```
Para 5 Nodos
Error de aproximacion en 5 nodos para f: 2.11e-02
Error de aproximacion en 5 nodos para g: 9.02e-03
Para 10 Nodos
Error de aproximacion en 10 nodos para f: 1.23e-03
Error de aproximacion en 10 nodos para g: 5.47e-05
Para 20 Nodos
Error de aproximacion en 20 nodos para f: 6.80e-06
Error de aproximacion en 20 nodos para g: 4.18e-11
Para 40 Nodos
Error de aproximacion en 40 nodos para f: 2.34e-07
Error de aproximacion en 40 nodos para g: 3.49e-07
Para 80 Nodos
Error de aproximacion en 80 nodos para f: 5.95e+04
Error de aproximacion en 80 nodos para g: 2.55e+05
```

Podemos apreciar convergencia hasta los 20 nodos, luego se pierde la convergencia debido a los errores de redondeo que involucra a polinomios de alto orden. Podemos observar que aunque f y g tienen perfiles similares, los polinomios de interpolación de g convergen a mayor velocidad.

```python
# Solucion 2
# Polinomio de interpolacion de Lagrange

import numpy as np
import sympy as sym
import math

# PROCEDIMIENTO
# Polinomio de Lagrange
def Lagrange(n,xi,fi):
    n = len(xi)
    x = sym.Symbol('x')
    polinomio = 0
    divisorL = np.zeros(n, dtype = float)
    for i in range(0,n,1):
        # Termino de Lagrange
        numerador = 1
        denominador = 1
        for j  in range(0,n,1):
            if (j!=i):
                numerador = numerador*(x-xi[j])
                denominador = denominador*(xi[i]-xi[j])
        terminoLi = numerador/denominador
        polinomio = polinomio + terminoLi*fi[i]
        divisorL[i] = denominador
    px = sym.lambdify(x,polinomio)
```

```
    return px

##########################################################################

x = np.array([5,10,20,40,80])# Asignamos a x, los valores a evaluar
for n in x:
    mm=2*n
    aux = np.linspace(-1,1,mm)

    print(f'Para {n} Nodos')
    #Funcion f(x)
    f = lambda x: 1/(1+x**2)
    xi = np.linspace(-1,1,n)
    fi = f(xi)
    p1 = Lagrange(n,xi,fi)
    #Error de aproximacion
    max_err=np.max(abs(f(aux) -p1(aux)))
    print(f'Error de aproximacion en {n} nodos para f: {max_err:.2e}')

    #Funcion g(x)
    g = lambda x: math.e**(-x**2)
    fi = g(xi)
    p2 = Lagrange(n,xi,fi)
    #Error de aproximacion
    max_err=np.max(abs(g(aux) -p2(aux)))
    print(f'Error de aproximacion en {n} nodos para g: {max_err:.2e}')
```

3. (6 puntos) Considere el siguiente conjunto de puntos x y los valores correspondientes de una función f en esos puntos:

$x:$	0	0,5	1	2
$f(x):$	1	1,8987	3,7183	11,3891

Implemente el método de diferencias divididas de Newton para encontrar el polinomio de interpolación de estos puntos y aproximar $f(1{,}5)$ y $f(8{,}4)$. Escriba la tabla de diferencias dividida. Halle el error relativo de la aproximación con respecto a $f(x) = x^2 + e^x$.

Solución: Aplicamos la fórmula de diferencias divididas a los nodos $x_0, x_1, \cdots, x_n$ (Ver Teorema 1 del capítulo 6 de [2]) recursivamente la k-ésima diferencia dividida relativa a x_1, x_{i+1}, $x_{i+2}, \cdots, x_{i+k}$:

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

para $i \in \{0, 1, \cdots, n-1\}$ y $k = 0, 1, \cdots, n$. Es decir, el proceso termina con

$$f[x_0, x_1, \cdots, x_n] = \frac{f[x_1, x_2, \cdots, x_n] - f[x_0, x_1, \cdots, x_{n-1}]}{x_n - x_0},$$

donde se ha inicializado el proceso con

$$f[x_i] = f(x_i), \;\; i \in \{0, 1, 2 \cdots, n\}.$$

A continuación, hallamos el polinomio de interpolación de diferencias divididas de Newton de orden $n = 3$, con los datos del problema

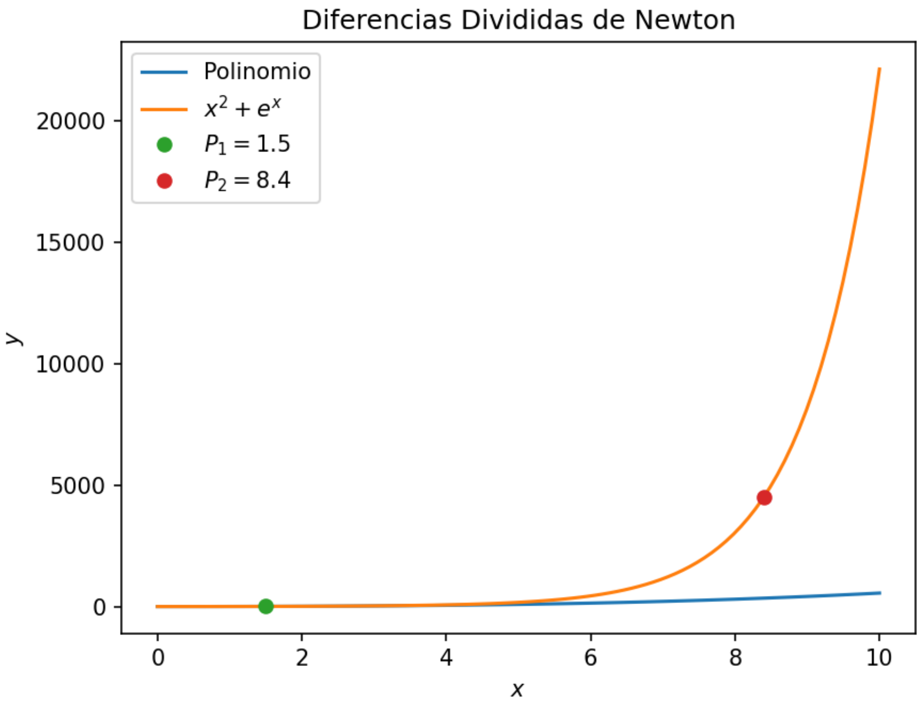
$$p_n(x) = f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, \cdots, x_k](x - x_0) \cdots (x - x_{k-1}).$$

Aplicando el algoritmo anterior, obtenemos los siguientes resultados

```
Tabla de diferencias divididas
[['i   ', 'xi  ', 'fi  ', 'F[1]', 'F[2]', 'F[3]']]
[[ 0.      0.      1.      1.7974  1.8418  0.423 ]
 [ 1.      0.5     1.8987  3.6392  2.6877  0.    ]
 [ 2.      1.      3.7183  7.6708  0.      0.    ]
 [ 3.      2.     11.3891  0.      0.      0.    ]]
Coeficientes de las diferencias dividida:
[1.     1.7974 1.8418 0.423 ]
Polinomio de interpolacin de Newton:
0.422966666666666*x*(x - 1.0)*(x - 0.5) + 1.8418*x*(x - 0.5) + 1.7974*x + 1.0

Evaluamos f(1.5) = 6.731689
Error relativo para 1.5: 6.59e-03
Evaluamos f(8.4) = 4517.626748
Error relativo para 8.4: 9.23e-01
Error global: 4.39e-05
```

Podemos apreciar que el error $9{,}23e{-}01$ de aproximación de p_n en el punto 8,4 es muy grande en comparación con el error global que se hace para una discretización de 100 puntos del intervalo $[0,2]$ de $4{,}39e-05$. Esto se debe a que este punto 8,4 está fuera del intervalo de interpolación $[0,2]$ donde el polinomio p_n se ajusta a la función $f(x) = x^2 + e^x$, como se aprecia en la siguiente figura.




```
# Solucion 3
# Metodo de diferencias divididas

import numpy as np
import sympy as sym
import matplotlib.pyplot as plt

#ingreso de datos
xi=np.array([0,0.5,1,2])
fi=np.array([1, 1.8987, 3.7183, 11.3891])

#procedimiento
```

```python
titulo = ['i   ','xi  ','fi  ']
n=len(xi)
ki=np.arange(0,n,1)
tabla=np.concatenate(([ki],[xi],[fi]),axis=0)
tabla=np.transpose(tabla)

dfinita=np.zeros(shape=(n,n-1), dtype=float)
tabla=np.concatenate((tabla, dfinita), axis=1)

[n,m]=np.shape(tabla)
diagonal=n-1
j=3

while(j<m):
    titulo.append('F['+str(j-2)+']')
    paso=j-2
    i=0
    while(i<diagonal):
        numerador=tabla[i+1,j-1]-tabla[i,j-1]
        denomiador=xi[i+paso]-xi[i]
        tabla[i,j]= numerador/denomiador
        i=i+1
    diagonal=diagonal-1
    j=j+1
dfinita=tabla[0,2:]
n=len(dfinita)
x=sym.Symbol('x')
polinomio=0
for j in range(0,n):
    factor=dfinita[j]
    termino=1
    for k in range(0,j):
        termino=termino*(x-xi[k])
    polinomio=polinomio+factor*termino
px=sym.lambdify(x,polinomio)

###################################
# ERROR GLOBAL
muestras=100
a=np.min(xi)
b=np.max(xi)
p_xi=np.linspace(a,b,muestras)
pfi=px(p_xi)
np.set_printoptions(precision=4)
def f(x):
    fx = x**2+(np.e)**x
    return fx
error = np.linalg.norm(f(xi)-px(xi),np.inf)

# RESULTADOS
print('Tabla de diferencias divididas')
print([titulo])
print(tabla)
print('Coeficientes de las diferencias dividida: ')
print(dfinita)
print('Polinomio de interpolacin de Newton: ')
print(polinomio)

print(f"\nEvaluamos f(1.5) = {f(1.5):4.6f}")
error2 = abs((f(1.5)-px(1.5))/f(1.5))
print(f'Error relativo para 1.5: {error2:.2e}')
print(f"Evaluamos f(8.4) = {f(8.4):4.6f}")
error3 = abs((f(8.4)-px(8.4))/f(8.4))
print(f'Error relativo para 8.4: {error3:.2e}')
print(f"Error global: {error:.2e}")
```

```
muestras=100
a=0
b=10
p_xi=np.linspace(a,b,muestras)
pfi=px(p_xi)

plt.plot(p_xi, pfi, label='Polinomio')
plt.plot(p_xi, f(p_xi), label='$x^2+e^x$')
plt.plot(1.5, f(1.5), 'o', label='$P_1 = 1.5$')
plt.plot(8.4, f(8.4), 'o', label='$P_2 = 8.4$')
plt.legend()
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.title('Diferencias Divididas de Newton')
plt.show()
```

Referencias

[1] R. L. Burden, J. D. Faires and A. M. Burden. Numerical Analysis. Boston, MA : Cengage Learning, Tenth edition (2016).

[2] D. Kincaid and W. Cheney. Numerical Analysis: Mathematics Of Scientific Computing. American Mathematical Society, Third Edition (2012).

[3] A. Quarteroni, R. Sacco and F. Saleri. Numerical Mathematics. Texts in Applied Mathematics 37, Second edition, Springer (2006).