Pontificia Universidad Javeriana

Departamento de Ingeniería de Sistemas

Análisis Numérico

28 de Febrero 2018

# Parcial 1

Juan David Rodríguez Arévalo

## Punto 1

a) **Algoritmo suma submatriz triangular superior**

```
23 def sumaMatriz(matrix):
24     upper = 0
25     oper = 0
26     for j in range(0, len(matrix)):
27         for i in range(0, len(matrix)):
28             if j<=i:
29                 upper+= (matrix[j][i])
30                 oper += 1
31             else:
32                 pass
33     return upper, oper
34
```

b) **Función para evaluar f(n)**

```
35 def evaluarFn():
36     s,o = sumaMatriz(matrix2)
37     print ('Suma para n:',len(matrix2),' = ',s,', f(n) = ',o)
38     s,o = sumaMatriz(matrix3)
39     print ('Suma para n:',len(matrix3),' = ',s,', f(n) = ',o)
40     s,o = sumaMatriz(matrix4)
41     print ('Suma para n:',len(matrix4),' = ',s,', f(n) = ',o)
42     s,o = sumaMatriz(matrix5)
43     print ('Suma para n:',len(matrix5),' = ',s,', f(n) = ',o)
44
45 evaluarFn()
46
```

**Pruebas obtenidas**

```
In [77]: runfile('C:/Users/Juanda/Documents/Analisis Numerico/Parcial/
Punto1.py', wdir='C:/Users/Juanda/Documents/Analisis Numerico/Parcial')
Suma para n: 2  =  7 , f(n) =  3
Suma para n: 3  =  26 , f(n) =  6
Suma para n: 4  =  70 , f(n) =  10
Suma para n: 5  =  155 , f(n) =  15
```

**c)** Para determinar f(n) se hizo pruebas con f(n) = #operaciones:

f(2) = 3

f(3) = 6

f(4) = 10

f(5) = 15

$$\boldsymbol{f(n) = n(n+1)/2}$$

```
Numero de operaciones segun f(n)= n*(n+1)/2
Numero de operaciones  3.0
Numero de operaciones  6.0
Numero de operaciones  10.0
Numero de operaciones  15.0
La formula coincide con las pruebas realizadas
```

Un indicio que se puede relacionar para encontrar la formula es que el área la cual se va a operar corresponde al área de un triángulo, formula que coincide con lo obtenido.

**Tiempos de ejecución**

**Algoritmo**

```
25 def sumaMatriz(matrix):
26     upper = 0
27     oper = 0
28     starttime = time.time()
29     for j in range(0, len(matrix)):
30         for i in range(0, len(matrix)):
31             if j<=i:
32                 upper+= (matrix[j][i])
33                 oper += 1
34             else:
35                 pass
36     endtime = time.time() - starttime
37     print ("El tiempo de ejecucion fue : %.4f Sg", endtime)
38     return upper, oper
```

**Ejecución**

```
In [86]: runfile('C:/Users/Juanda/Documents/Analisis Numerico/Parcial/
Punto1.py', wdir='C:/Users/Juanda/Documents/Analisis Numerico/Parcial')
El tiempo de ejecucion fue :  0.0 sg
Suma para n: 2  =  7 , f(n) =  3
El tiempo de ejecucion fue :  0.0 sg
Suma para n: 3  =  26 , f(n) =  6
El tiempo de ejecucion fue :  0.0 sg
Suma para n: 4  =  70 , f(n) =  10
El tiempo de ejecucion fue :  0.0 sg
Suma para n: 5  =  155 , f(n) =  15
```

Las operaciones son muy pocas ya que las pruebas se hicieron con valores de n pequeños por lo cual el tiempo es aproximadamente 0. Para la capacidad de la maquina es 0. Siendo O() el evaluó de complejidad (número de operaciones ya fue obtenido previamente obtenemos en el peor de los casos:

$$\boldsymbol{O(n^2)}$$

## Punto 2

**Algoritmo de Aitken, serie de Taylor para f(x) = e^x; x0 = 0 con x = 1 y determinar su convergencia**

```
from numpy import *
import math


def f(x): return math.e**x

def Aitken(x0,x1,x2):
    return x2-((x2-x1)**2)/(x2-2*x1+x0)
def serTaylor(x,k):
    return sum((x**i)/math.factorial(i) for i in range(k+1))

def detConvergencia():
    y=1
    x0=serTaylor(y,0)
    x1=serTaylor(y,1)
    x2=serTaylor(y,2)
    err=1
    i=3
    p0=Aitken(x0,x1,x2)
    print('Pn(x)','\t\t','Error')
    while(err>1.e-8):
        p1=p0
        print(p0,'\t\t',err)
        x0=serTaylor(y,i-2)
        x1=serTaylor(y,i-1)
        x2=serTaylor(y,i)
        i+=1
        p0=Aitken(x0,x1,x2)
        err=abs(p0-p1)/abs(p0)

detConvergencia()

```

**Salida para un error de 1e-9**

```
In [95]: runfile('C:/Users/Juanda/Documents/Analisis Numerico/Parcial/
Punto2.py', wdir='C:/Users/Juanda/Documents/Analisis Numerico/Parcial')
Pn(x)           Error
3.0             1
2.7499999999999996              0.09090909090909109
2.722222222222222               0.010204081632653026
2.7187499999999996              0.0012771392081737274
2.7183333333333333              0.00015328019619851172
2.718287037037037               1.70314229753378e-05
2.71828231292517                1.7379033239352849e-06
2.7182818700396827              1.6292846311355922e-07
2.718281831765628               1.4080237797666107e-08
```

La serie converge como se esperaba ( se comprobó mediante la herramienta Wolfram) al **número e, dado por 2.7182818…** , para este caso no se conocía si convergía inicialmente o no por lo cual el error se estableció inicialmente en 1e-5, una vez se estableció la convergencia se redujo el error para obtener mayor precisión en la respuesta.

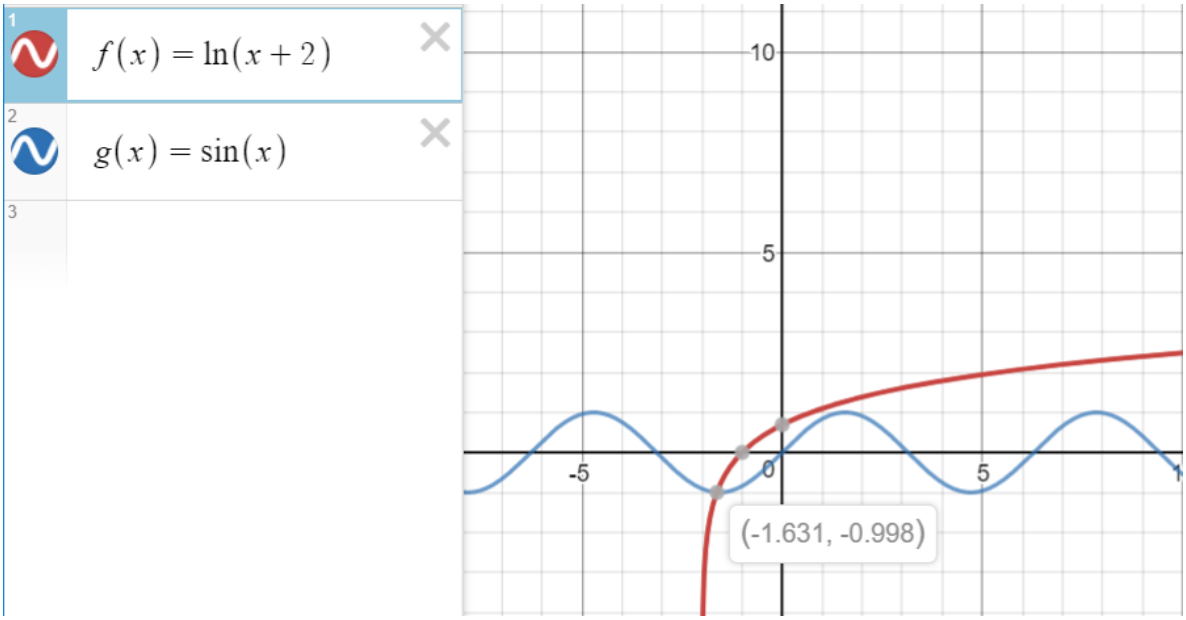
A continuación se presenta la obtención de la serie de Taylor dado un f(x)

A Maclaurin series is a Taylor series expansion of a function about 0,

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \ldots + \frac{f^{(n)}(0)}{n!}x^n + \ldots.$$

## Punto 3

**Justificación del intervalo tomado:**




La función f(x) = ln(x+2), tiene una asíntota vertical en -2 luego el dominio de la función está dado por (-2,∞) con esto conocemos la cota por la izquierda. Para establecer la cota por la derecha conocemos que la función g(x) = sin(x) tiene rango de [-1.1] luego, no es necesario evaluar valores de f(x) donde su rango sea mayor a 1, aproximadamente 1. Luego el intervalo está dado por (-2,1), para el método de la secante se redujo el rango aún más para hacerlo converger en menos iteraciones, el rango fue (-1.8,-1.0).

1. **Método de la secante**

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 28 07:42:16 2018

@author: Juanda
"""
import numpy as np

def f(x): return np.log(x+2) - np.sin(x)
def f1x(x): return (1/(x+2)) - np.cos(x)

def secante(x0,x1,tol):

  x = (f(x1)*x0-f(x0)*x1)/(f(x1)-f(x0))
  error = 1
  print("\t x","\t\t\t","Error")
  while (error > tol):
    x0 = x1
    x1 = x
    x = (f(x1)*x0-f(x0)*x1)/(f(x1)-f(x0))
    if (f(x) == 0):
        break
    error = abs(f(x)/f1x(x))
    print(x,"\t",error)

secante(-1.8,-1.0,1.e-7)
```

**Ejecución**

```
In [101]: runfile('C:/Users/Juanda/Documents/Analisis Numerico/Parcial/
Punto3a.py', wdir='C:/Users/Juanda/Documents/Analisis Numerico/Parcial')
         x                       Error
-1.840215350016111       0.13334201147774308
-1.573703771789245       0.06274762265997307
-1.6123100245020496      0.01968388196267236
-1.6331029050504058      0.0016551578952348689
-1.6313958129917459      4.778741787903624e-05
-1.6314434774712356      1.194976707349975e-07
-1.6314435969774892      8.604381612152636e-12
```

## 2. Método Newton Generalizado

```
Created on Wed Feb 28 07:06:01 2018

@author: Juanda
"""
import math

def newtonRaphson(f,df,a,b,tol=1.0e-5):
    'import error'
    from numpy import sign
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if sign(fa) == sign(fb): print('Root is not bracketed')
    x = 0.5*(a + b)
    for i in range(30):
        fx = f(x)
        if fx == 0.0: return x
        # Tighten the brackets on the root
        if sign(fa) != sign(fx): b = x
        else: a = x
        # Try a Newton-Raphson step
        dfx = df(x)
        # If division by zero, push x out of bounds
        try: dx = -fx/dfx
        except ZeroDivisionError: dx = b - a
        x = x + dx
        # If the result is outside the brackets, use bisection
        if (b - x)*(x - a) < 0.0:
            dx = 0.5*(b - a)
            x = a + dx
        # Check for convergence
        print('Iteration: ',i,'X: ',x)
        if abs(dx) < tol*max(abs(b),1.0): return x
    print('Too many iterations in Newton-Raphson')

def f(x): return math.log(x+2) - math.sin(x)
def df(x): return (1/(x+2) - math.cos(x))
x = newtonRaphson(f,df,-1.99999,0.0)
print ('Raiz: ',x)
```

**Ejecución**

```
In [104]: runfile('C:/Users/Juanda/Documents/Analisis Numerico/Parcial/
Punto3b.py', wdir='C:/Users/Juanda/Documents/Analisis Numerico/Parcial')
Iteration:  0 X:  -1.4999924999999998
Iteration:  1 X:  -1.6577564856798133
Iteration:  2 X:  -1.6324930040914034
Iteration:  3 X:  -1.6314452567692699
Iteration:  4 X:  -1.6314435969730363
Raiz:  -1.6314435969730363
```

Como se puede observar ambos métodos llevan al mismo valor aproximado. Con esto podemos concluir algunos puntos importantes a la hora de tener en cuenta el un método u otro.

1. El método de la secante es útil en comparación con el método de Newton Generalizado ya que este no depende del cálculo de la derivada, en algunos casos pueden ser difíciles de calcular.
2. Como contra para el método de la secante se necesita establecer dos valores iniciales de x. En algunos casos puede ser difícil identificarlos por lo cual Newton puede ser una gran alternativa.