ILI 285
Laboratorio #2

Alonso Sandoval Acevedo
asandova@alumnos.inf.utfsm.cl
201073011-5

Hernán Vargas Leighton
hvargas@alumnos.inf.utfsm.cl
201073009-3

28 de mayo de 2014

1. Descripción del experimento

Entre la gran variedad de métodos para obtener las raíces trabajaremos con dos muy utilizados como son el método de la bisección y el método de Newton. Analizaremos objetivamente los pros y contras de cada uno de ellos comparando algunos de sus aspectos. Por lo visto en clases esperamos que el método de bisección tenga una fácil implementación y, a pesar de que su tiempo de ejecución debería ser alto, nos entregue buenos resultados, por otra parte, el método de Newton debería ser un poco más difícil de implementar pero debería ser mucho más rápido.

2. Desarrollo

2.1. Encontrando los ceros

1. Uno de los requisitos para poder utilizar el método de la bisección es que la función evaluada en los extremos del rango $[a, b]$ cumpla con que $f(a) \cdot f(b) < 0$, es decir, un cambio de signo entre a y b. Entre las funciones a las que debemos encontrarles raíces notamos que algunas de ellas no cumplen con este requisito, por ellos fueron modificadas de la siguiente manera:

   $$f_1 = 1 - \frac{\sin x}{x} = 0 \to 1 = \frac{\sin x}{x} \to x = \sin x \to 0 = \sin x - x \to f_1 = \sin x - x$$
   $$f_2 = x^{10} \to x^{10} = (x^5)^2 \to f_2 = x^5$$
   $$f_4 = |x - 10| \to f_4 = x - 10$$

   Para f_2 sabemos que la única raíz de x^2 es 0, por ello, el problema puede simplificarse a obtener la raíz de x^5. Por otro lado, para f_4 nos servirá tanto la raíz de $10 - x$ como la de $x - 10$ pues lo dentro del valor absoluto es lineal. Para f_3 tenemos el problema de que las raíces son $\pm\infty$
   Nuestra implementación de bisección recibe como parámetros una función y un intervalo desde `a` hasta `b` y encuentra una raíz en dicho intervalo.

2. En el método de Newton se utilizó la librería "sympy"para el manejo algebraico, obtención de derivadas y límite para casos de indefinición (división por cero principalmente). Para las funciones de Bessel, se utilizó, tal como en el caso anterior, la librería "scipy", la cual retorna las soluciones de las funciones de Bessel y su respectiva derivada para cierto parámetro (x_0 en este caso) .
   Se implementó el método de Newton visto en clases, para cual, antes de evaluar cada función se hace un "test"para chequear la posible multiplicidad de raíces. Con las 5 primeras iteraciones, se analiza el error lineal ($\frac{e_{i+1}}{e_i}$), el cual, al ser relativamente constante (con respecto a una tolerancia definida), se determina la multiplicidad a través de este valor:

   $$S = \frac{m-1}{m} \to m = \frac{1}{1-S}$$

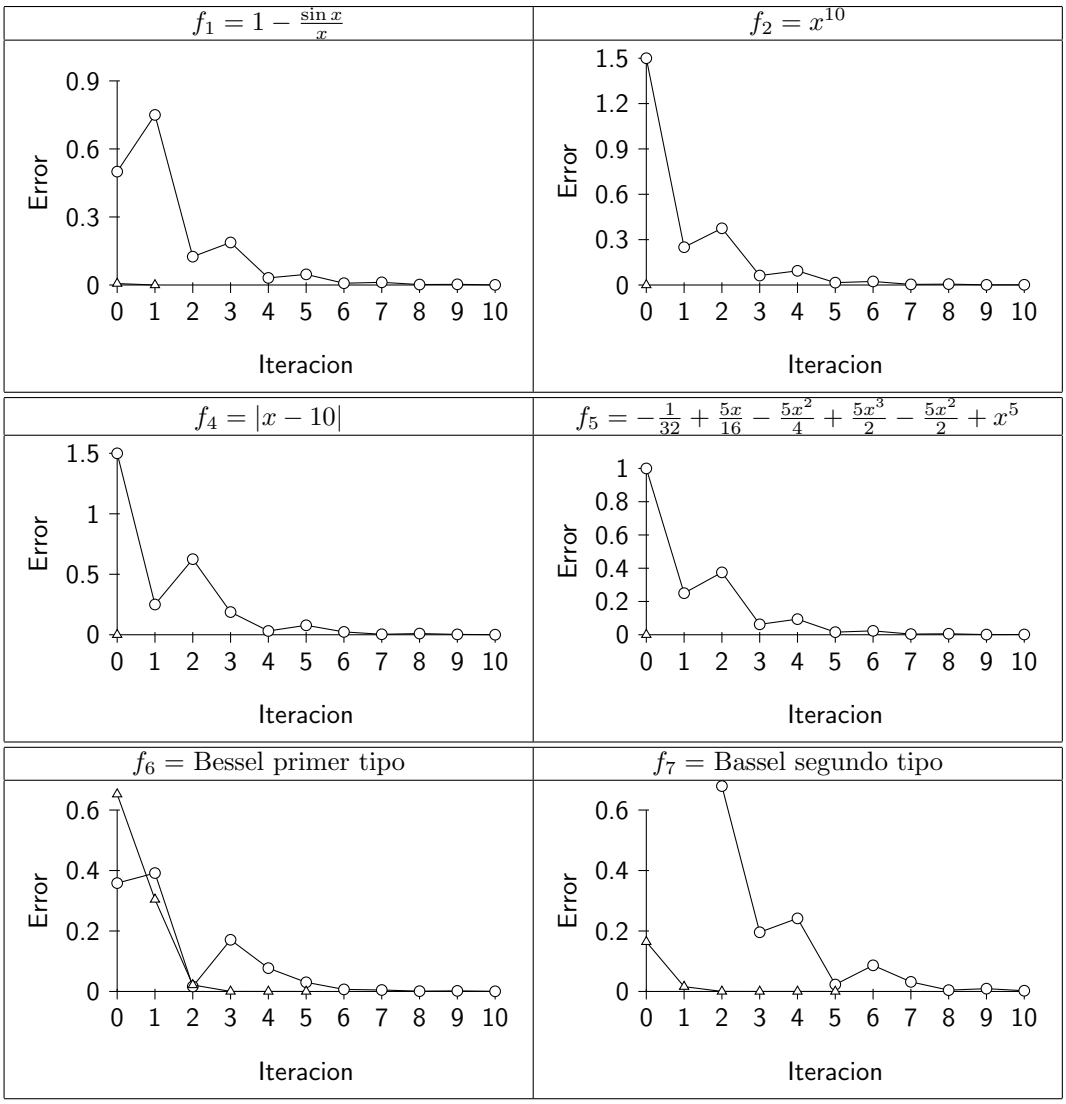
   Haciendo un redondeo superior obtenemos un valor aproximado de la multiplicidad, en las funciones de la tarea, calzan exactamente. Si ($\frac{e_{i+1}}{e_i}$) no es constante, se utiliza multiplicidad = 1.

Así, en los casos de existir raíz múltiple, se utiliza el método de newton mejorado.
Los errores en tiempo de ejecución se calculan entre el valor actual y el anterior ($|x_i - x_0|$), al momento de encontrar una raíz, se obtienen los errores con respecto a esta para el posterior análisis de rendimiento.
Por último, para el caso en que la derivada de la función evaluada en algún x_0 sea 0, nuestro algoritmo no puede continuar (a menos que se simplifique algebraicamente como: $\frac{x^3}{x^2}$), caso que ocurre para la función e^{-x^2}.

3. Tabla con los ceros encontrados por cada método.

Función	**Bisección**	**Newton**
f_1	$-0{,}0000000074505$	$0{,}0000000126992$
f_2	$-0{,}0000000055879$	0
f_3	Falló	Falló
f_4	9,9999999953433	10
f_5	0,4999999944120	0,5
f_6	3,1415926534682	3,14159265359
f_7	$-1{,}570796324871$	1,57079632679

4. Gráficos comparando errores entre los métodos, circulo para bisección, triangulo para Newton.
NOTA: Por simplicidad solo hasta la iteración 10.



Notamos que en la mayoría de los casos para el método de Newton se llega a una solución rápidamente y con errores cercanos a cero.

Por otro lado, la bisección tiene errores que oscilan y, aunque terminan llegando al mismo valor, toma cerca de 30 iteraciones (Dependiendo de la tolerancia definida).

5. Tabla de comparación de iteraciones, convergencia y tiempo.

Método de la Bisección		
$f_i(x)$	Tiempo	Iteraciones
$f_1(x)$	0,000083923339843s	27
$f_2(x)$	0,000115156173706s	29
$f_3(x)$	Falló	-
$f_4(x)$	0,000063896179199s	30
$f_5(x)$	0,000102043151855s	29
$f_6(x)$	0,00263714790344s	29
$f_7(x)$	0,00366306304932s	31

Método de Newton		
$f_i(x)$	Tiempo	Iteraciones
$f_1(x)$	0,898484945297s	2
$f_2(x)$	0,00857090950012s	1
$f_3(x)$	Falló	-
$f_4(x)$	0,02476811409s	1
$f_5(x)$	0,0450839996338s	1
$f_6(x)$	0,000953912734985s	6
$f_7(x)$	0,000863075256348s	6

Cabe destacar que la tasa de convergencia de el método de la bisección es lineal constante igual a 0,5, por otro lado, para el método de Newton tenemos una taza de convergencia de orden superior , es decir, converge cada vez más rápido.

3. Conclusiones

Al analizar las tablas y los gráficos, queda bastante claro que el número de iteraciones baja considerablemente con el método de newton, sin embargo los tiempos de ejecución no se condicen con esta ventaja en nuestro caso, como habíamos supuesto al principio. Las razones a esto último están asociadas al coste computacional que conlleva el manejo algebraico de las funciones, los cálculos de derivadas y límite, etc., funciones utilizadas de la librería sympy para el método de newton. Independiente de la implementación, el rendimiento del método de newton se ve afectado por las dificultades en el computo de derivadas, las cuales no siempre son sencillas de resolver.
Por otro lado, las funciones para el método de la bisección debieron ser modificadas en su mayoría, dado que varias de ellas eran estrictamente positivas o negativas, no así para el método de newton. Esto da una ventaja significativa al segundo método, dado que para hacer las modificaciones pertinentes, se requiere de cierto conocimiento en el comportamiento de las funciones, cosa que puede ser no trivial en algunos casos.

4. Referencias

1. Libro guía del curso, Numerical Analysis, 2nd Edition, Timothy Sauer, 2006 Pearson Education.
2. Función de Bessel, http://en.wikipedia.org/wiki/Bessel_function

5. Anexo

../Codigos/Laboratorio2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import math as m
from sympy import *
import sympy
import scipy.special
import time

# Función de Bessel
Jbessel = scipy.special.sph_jn
Ybessel = scipy.special.sph_yn
# Tolerancia de bisección y newton
TOL = 0.00000001
# Símbolo x
x = symbols('x')

#f es el diccionario de funciones:
f = {   1: lambda x:   m.sin(x) - x,
        2: lambda x:   x**3,
        3: lambda x:   1.0/m.exp(x**2),
        4: lambda x:   (x - 10),
        5: lambda x:   ((2*x-1)**5)/32,
        6: lambda x:   tuple(Jbessel(0,x)[0])[0],
        7: lambda x:   tuple(Ybessel(0,x)[0])[0] }

#g es el diccionario de funciones simbólicas.
g = {   1:  1 - (sin(x)/x),
    2:  x**10,
    3:  1/exp(x**2),
    4:  ((x-10)**2)**0.5,
    5:  ((2*x-1)**5)/32,
    6:  lambda x: Jbessel(0,x),
    7:  lambda x: Ybessel(0,x)}

#x0s es el diccionario de valores iniciales sugeridos.
x0s = { 1: 0.5, 2: 0.5, 3: 0, 4: 8, 5: 0, 6: 1, 7: 2 }

#rangos es el diccionario de rangos para la bisección.
rangos = {1: (-2, 3), 2: (-4, 1), 3: (-5, 3),
          4: (5, 12), 5: (-1, 4), 6: (2, 5), 7: (-4,10) }

#timeTest retorna el tiempo de ejecución de una función.
def timeTest(f, args):
    start = time.time()
    if hasattr(args, '__iter__'): out = f(*args)
    else: out = f(args)
    totalTime = time.time() - start
    return out, totalTime

#getSolutionError calcula el error de los datos con respecto a la solución.
def getSolutionError(value, data):
    if not hasattr(data, '__iter__'): return m.fabs(data - value)
```

```
    errors = []
    for elem in data:
        errors.append(m.fabs(elem - value))
    return errors

### PARTE 1: MÉTODO DE BISECCIÓN ###
#fcheck evalúa si dos puntos de una función tienen diferentes signos.
def fcheck(func,a,b):
    fa,fb = func(a), func(b)
    if (fa < 0  and fb <0) or (fa > 0 and fb >0): return False
    return True

def bisec(f, a, b):
    if not fcheck(f,a,b): raise ValueError('f(a)*f(b)>0')
    count = 0
    prev_value = []
    while True:
        count +=1
        c = (a+b)/2.0
        prev_value.append(c)
        r = f(c)
        if (b-a)/2 < TOL or r == 0: break
        if fcheck(f,a,c): b = c
        else: a=c
    return c, count, prev_value

#Imprimiendo resultados
verbose = True      #Cambiar a false para que no imprima los errores
for i in range(1,8):
    try:
        a,b = rangos[i]
        [value, iterations, prev_values], t = timeTest(bisec, (f[i],a,b))
        print 'F =',i,'-> Se obtuvo:',repr(value),\
              'en', iterations,'iteraciones.'
        print 'El algoritmo demoro',t,'segundos'
        if verbose:
            print 'Los errores son:\n\titer\tERROR'
            for i, elem in enumerate(getSolutionError(value, prev_values)):
                print '\t',i,'\t',elem
    except ValueError, e:
        print 'F =',i,'-> No pudo ser evaluado por biseccion ya que:',e
    finally:
        print '#'*70

### PARTE 2: MÉTODO DE NEWTON ###

#error Cálcula los errores entre iteraciones
def error(xi,x0,e,ALL):
    ei = ((xi-x0)**2)**0.5
    if e != 0: #No consideramos el error para la 1 iteración
        elin = ei/e
        equa = ei/(e**2)
    else:
        elin = ei
        equa = ei
```

```
    if ALL:
        return [ei, elin, equa]
    print "X_i: ", xi, " | e/ei: ",elin," | e/ei^2: ",equa
    return ei

#newton implementa el método de newton
def newton(f, x0, key):
    if key < 6:
        df = diff(f, x) #derivamos la función
    e = 0 #error
    saveErr=[] #data ei for Plot
    #Cálculo de la multiplicidad
    m = testMult(f, x0, key)
    if m==0:
        return
    print "f",key," |initial: ",x0,"|multiple root: ",m
    print "#Iteration#"
    while True:
        if key < 6:
            if limit(f, x, x0)==0:
                break
        fx0 = f.evalf(subs={x:x0}) if key<6 else f(x0)[0][0]
        dfx0 = df.evalf(subs={x:x0}) if key<6 else f(x0)[1][0]
        if dfx0 == 0:
            print "F'(Xi) = 0, Metodo no valido"
            return
        xi = x0 - m*(fx0/dfx0) #Método de Newton
        e = error(xi,x0,e, False) #error
        saveErr.append(xi)
        x0 = xi
        if e < TOL:
            break
    print "Root found: ", x0
    print "Errores: "
    i = 1
    for elem in saveErr:
        print '\t',i,'\t', ((elem-x0)**2)**0.5
        i = i + 1

#Chequea raices múltiples y convergencia
def testMult(f,x0, key):
    if key < 6:
        df = diff(f,x)
    e,err, check1, check2 = 0,[],[],[]
    for i in range(0,5):
        if key < 6:
            if limit(f,x,x0)==0:
                return 1
        fx0 = f.evalf(subs={x:x0}) if key<6 else f(x0)[0][0]
        dfx0 = df.evalf(subs={x:x0}) if key<6  else f(x0)[1][0]
        xi = x0 - (fx0/dfx0)
        err = error(xi,x0,e,True) #error
        e = err[0]
        check1.append(err[1]) #e/ei
        check2.append(err[0]) #e/ei^2
```

```
        x0 = xi
    #Si el error crece, no hay raices
    if check2[4]/check2[3] > 1:
        print "The function not have root"
        return 0
    #Si la tasa de error lineal es constante
    if abs(check1[4] - check1[3]) < 1:
        return round(1/(1-check1[4])) #se resuelve: s=m-1/m -> m=1/(1-s)
    else:
        return 1

for key in x0s:
    trash, t = timeTest(newton, (g[key], x0s[key], key))
    print "DEMORO:",t,"segundos.\n",'#'*70
```