Instituto Tecnológico de Costa Rica
Ingeniería en Computadores

CE-3102: Análisis Numéricos para Ingeniería
Semestre: I - 2021

Catálogo Grupal de Algoritmos

Integrantes:

- Brayan Alfaro González - Carné 2019380074
- Sebastián Alba Vives - Carné 2017097108
- Kevin Zeledón Salazar - Carné 2018076244
- Daniel Camacho González - Carné 2017114058

1 Integración Numérica

1.1 Regla del trapecio y cota de error

Código 1: Regla del trapecio y cota de error en Octave

```
% Ejemplo numerico
function trapecio
  pkg load symbolic;
  warning('off','all');
  clc;
  f='ln(x)';
  a=2;
  b=5;
  [aprox,error]=trapecioAux(f,a,b);

  display(["El valor aproximado por el metodo es ",num2str(aprox)," con un error de
     ",num2str(error)]);

 end

% Funcion auxiliar que aproxima el valor para la integral definida de f en el
   intervalo [a,b]
% Entradas :
%      f: Funcion a la cual se le quiere calcular la integral.
%      a: Limite inferior del intervalo sobre el que se quiere calcular la integral.
%      b: Limite superior del intervalo sobre el que se quiere calcular la integral.
% Salida :
%      aprox : Valor aproximado de la funcion en el intervalo indicado.
%      error: Error de la aproximacion

 function[aprox,error]=trapecioAux(f,a,b)
  %Definicion de la variable simbolica x y de la funcion de MatLab f
  x=sym('x');
  f=matlabFunction(sym(f));

  %Calculo del valor aproximado.
```

```
  aprox=((b-a)/2)*(f(a)+f(b));

  %Definicion de la funcion f_2 con la que se pretende calcular el maximo de f''(x)
     en el intervalo [a,b]
  f_2=function_handle(-1*abs(diff(diff(f,x),x)));

  %Calculo de la constante alpha para calcular el error de la aproximacion.
  alpha=-1*f_2(fminbnd(f_2,a,b));

  %Calculo del error.
  error=((b-a)^3/12)*alpha;
  return;

 end
```

1.2 Regla de Simpson y cota de error

Código 2: Regla de Simpson y cota de error en Python

```
import numpy as np
from sympy import *
import math
from numpy.core.umath import maximum, minimum
#Entradas:
#f: Funcion a la que se le quiere aproximar la integral
#   Se espera recibir un string de variable x
#a: inicio del intervalo
#b: final del intervalo
#Salidas:
#approx: Una aproximacion a la integral de f en el intervalo [a,b]
#error: aproximado del error de la integral, devuelve -1 en caso de que no se pueda
   calcular

def  simpson (f,a,b,x):
    #f = parse_expr(ff)
    #print(f.args)
    intervalo = True
    approx = 0;
    if a>b:
        intervalo = False #El intervalo que nos dieron era degenerado
        #Invertimos el orden de integracion
        c = a
        b = a
        a = c
    h = (b-a)/2
    approx = f.subs(x,a)+f.subs(x,b)+4*f.subs(x,(a+b)/2)
    approx *= h/3
    if(intervalo == False):
        #Cambiamos el signo de la integral en caso de que el intervalo fuera
           degenerado
        approx*=-1
    error  = h**5/90
    try:
```

```python
        derivadaCuarta = f.diff(x)
        derivadaCuarta = derivadaCuarta.diff(x)
        derivadaCuarta = derivadaCuarta.diff(x)
        derivadaCuarta = derivadaCuarta.diff(x)
        approxMaximo = abs(derivadaCuarta.subs(x,a))
        for i in range(1,10000):
            approxMaximo = max(approxMaximo,derivadaCuarta.subs(x,a+(b-a)/10000*i))
        error*= approxMaximo
    except:
        error = -1
    return approx,error;

x = Symbol('x')
print(simpson(x**2+1,0,np.pi,x))
```

1.3 Regla compuesta del trapecio y cota de error

Código 3: Regla compuesta del trapecio y cota de error en C++

```cpp
#include <ginac/ginac.h>
#include <iostream>
#include <math.h>

using namespace std;
using namespace GiNaC;

/** Metodo encargado de evaluar una funcion representada en string en un valor
    especifico utilizando la libreria GiNaC
* @param funct La funcion que se quiere evaluar en formato de string
* @param value El valor en el cual se va a evaluar la funcion
*/
double f(string funct, double value){
  symbol x;
  symtab table;
  table["x"] = x;
  parser reader(table);
  ex f = reader(funct);
  ex sol = evalf(f.subs(x==value));
  if(!is_a<numeric>(sol)){
    throw logic_error("Se va a intentar convertir un expresion a double que no es
        numerica");
  }
  return ex_to<numeric>(sol).to_double();
}

/** Metodo encargado de evaluar una funcion representada en su version simbolica en
    un valor especifico utilizando la libreria GiNaC
* @param funct La funcion que se quiere evaluar en formato simbolico
* @param value El valor en el cual se va a evaluar la funcion
*/
double f(ex funct, double value, symbol x){
  ex sol = evalf(funct.subs(x==value));
  if(!is_a<numeric>(sol)){
```

```
    throw logic_error("Se va a intentar convertir un expresion a double que no es
        numerica");
  }
  return ex_to<numeric>(sol).to_double();
}
/**
 * Metodo encargado de encontrar la solucion de una ecuacion utilizando Newton-
    Raphson
 * @param funct La funcion que se quiere evaluar en formato simbolico
 * @param xi Un valor inicial para arrancar el Algoritmo
 * @param iterMax La cantidad maxima de iteraciones a llevar a cabo
 * @param x La variable simbolica
 */
double newtonRaphson(ex funct, double xi, int iterMax, symbol x){
  ex df = funct.diff(x);
  cout << df << endl;
  cout << funct << endl;
  ex xk = xi;
  for (int i = 0; i < iterMax; i++){
    if(evalf(df.subs(x==xk)) <= 1e-8){
      break;
    }
    xk = xk - funct.subs(x==xk)/df.subs(x==xk);
  }
  // Retornando
  if(!is_a<numeric>(xk)){
    throw logic_error("Se va a intentar convertir una expresion a double que no es
        numerica");
  }
  double sol = ex_to<numeric>(xk).to_double();
  return sol;
}

/**
 * Metodo encargado de retornar la version simbolica de una funcion
 * @param funct La funcion que se quiere evaluar en formato de string
 * @returns La funcion simbolica representada por el string ingresado
 */
ex fSym(string funct, symbol x){
  symtab table;
  table["x"] = x;
  parser reader(table);
  ex f= reader(funct);
  return f;
}

/**
 * Metodo encargado de calcular el m ximo de una funcion dado un intervalo
 * @param funct La funcion que se quiere evaluar en formato de string
 * @param x La variable simbolica
 * @param a Inicio del intervalo
 * @param b Final del intervalo
 */
double maxFun(ex funct, symbol x, double a, double b){
```

```
  ex befConv; // Esta variable se va a usar para verificar si las expresiones que se
     van a convertir a double son numericas
  ex f= funct; // Funcion
  ex df = f.diff(x); // Funcion diferenciada

  // Primero se tiene que calcular cu ndo la derivada es 0
  double valorLimite = newtonRaphson(funct, (a+b)/2, 500, x);

  // Luego se valua donde la derivada da 0 y en los limites
  vector<double> posibles = {};

  if(!(valorLimite < a || valorLimite > b)){ // Si el valor limite se encuentra en el
      rango..
    befConv = evalf(f.subs(x==valorLimite));
    posibles.push_back(
      abs(ex_to<numeric>(befConv).to_double())
    );
  }

  befConv = evalf(f.subs(x==a));
  posibles.push_back(
    abs(ex_to<numeric>(befConv).to_double())
    );
  befConv = evalf(f.subs(x==b));
  posibles.push_back(
    abs(ex_to<numeric>(befConv).to_double())
    );
  double max = posibles[0];
  // Ahora se debe encontrar el mayor n mero dentro de 'posibles'
  for(int i = 0; i < posibles.size(); i++){
    if(posibles[i]>max){
      max = posibles[i];
    }
  }

  if(!is_a<numeric>(max)){
    throw logic_error("Se va a intentar convertir una expresion a double que no es
       numerica");
  }

  return ex_to<numeric>(max).to_double();
}

/**
 * M todo que calcula una aproximacion de la integral de una funcion
 * A partir el m todo del trapecio compuesto. Se asume que la funcion
 * Ingresada es dos veces derivable
 * @param funct Funcion a evaluar en formato de string
 * @param puntos Cantidad de puntos en los que se va a dividir el intervalo
 * @param a Inicio del intervalo de integracion
 * @param b Final del intervalo de integracion
 * @return Imprime la aproximacion de la integral junto con su error, ademas devuelve
     la aproximacion de la integral
*/
```

```
double trapecio_compuesto(string funct, int puntos, double a, double b){
  symbol x("x");

  vector<double> s = {};
  double espacio = (b-a)/(puntos-1);
  double i = a;
  while (i<=b){
    s.push_back(i);
    i = i+espacio;
  }

  ex diff2 = fSym(funct,x).diff(x,2);

  double h = s[2]-s[1];
  double sum = 0;
  i = 0;
  while (i<=puntos-2){
    sum = sum + f(funct,s[i]) + f(funct,s[i+1]);
    i=i+1;
  }
  double alph=maxFun(fSym(funct, x).diff(x,2), x, a, b);
  double power = ex_to<numeric>((GiNaC::pow(h,2))).to_double();
  double aprox = (h/2)*sum;
  double err = power*((b-a)/12)*alph;

  cout << "La aproximacion es: " << aprox << endl;
  cout << "Con error: " << err << endl;

  return aprox;
}


/**
 * Ejemplo Num rico
 */
int main (int argc, char const* argv[]){
  trapecio_compuesto("log(x)", 4,2,5);
  return 0;
}
```

1.4 Regla compuesta de Simpson y cota de error

Código 4: Regla compuesta de Simpson y cota de error en Python

```
import numpy as np
import sympy as S
from scipy import optimize

# Se usan las librerias numpy, sympy, scipy y scikit-optimize

# Funcion para calcular una integral mediante la regla compuesta de simpson
# Entradas:
#       funcion : la funcion a integrar en string
#       n_puntos : n mero de puntos en los cuales dividir el intervalo
#       intervalo : tupla que indica el intervalo de integraci n
# Salida:
#       integral: resultado de la integraci n
#       error: cota de error de la aproximaci n
def simpson_compuesto(funcion, n_puntos, intervalo):
    try:
        # Se establece el simbolo x
        x = S.symbols("x")
        # Se lee la funcion y se deriva
        funcion = S.sympify(funcion)
        f4 = S.diff(funcion, x, 4)
        # Se convierte a funcion de python
        funcion = S.lambdify(x, funcion)
        f4_abs = S.lambdify(x, -abs(f4))

        # Se establecen los valores iniciales
        a = intervalo[0]
        b = intervalo[1]
        x = np.linspace(a, b, num=n_puntos)
        h = x[1]-x[0]
        suma_par = 0
        suma_impar = 0
        n = n_puntos-1
        # Se calcula la integracion
        for i in range(1,int(n/2)):
            suma_par += funcion(x[2*i])
        for i in range(1,int(n/2)+1):
            suma_impar += funcion(x[2*i-1])

        integral = (h/3)*(funcion(x[0]) + 2*suma_par + 4*suma_impar + funcion(x[n]))

        # Se calcula el maximo de la cuarta derivada de f en el intervalo
        f4_abs_max = -optimize.minimize_scalar(f4_abs, bounds=(a, b), method='bounded').fun
        # Se calcula el error
        error = ((b-a)*h**4/180)*f4_abs_max
        return (integral, error)
    except:
        return (0,float('inf'))

# Ejemplo Num rico
funcion = "ln(x)"
```

```
puntos = 7
intervalo = (2,5)
resultado = simpson_compuesto(funcion, puntos, intervalo)
print("Funcion: "+funcion+" | Puntos: "+str(puntos)+" | Intervalo: "+str(intervalo))
print("Resultado: "+str(resultado[0])+" | Error: "+str(resultado[1]))
```

1.5 Cuadratura Gaussiana y cota de error

Código 5: Cuadratura Gaussiana y cota de error en Octave

```
%% Requiere la libreria miscellaneous: pkg install -forge miscellaneous


%% Ejemplo numerico
function [approx,err] = cuad_gaussiana (f,n,a,b)
  pkg load miscellaneous
  f = @(x) 4/(1+x^2);
  [aprox, error] = cuad_gaussiana_aux(f,20,0,1)
endfunction

%{
Entradas:
f: Funcion a la que se le quiere calcular la integral
n: Numero de orden
a: Principio del intervalo
b: Final del intervalo
Salidas:
approx: Aproximacion de la integral de f en el intervalo [a,b]
error: Aproximacion del error de la integral
%}

function [approx,error] = cuad_gaussiana_aux (f,n,a,b)
  polLegendre = legendrepoly(n);
  r = roots(polLegendre);
  r = sort(r);
  approx = 0;
  derivLegendre = polyder(polLegendre);
  for i = 1: n
    xi = r(i);
    wi = 2/((1-xi^2)*(polyval(derivLegendre,xi))^2);
    approx+= f((b-a)*xi/2+(b+a)/2)*wi;
  endfor
  approx*= (b-a)/2;
  error = abs(approx- quad(f,a,b));
endfunction
```