Catálogo Grupal de Algoritmos

Integrantes:

- Josué Araya García - 2017103205
- Jonathan Guzmán Araya - 2013041216
- Mariano Muñoz Masís - 2016121607
- Luis Daniel Prieto Sibaja - 2016072504

Índice

1. Tema 1: Ecuaciones no Lineales

1.1. Método 1: Bisección

Código 1: Lenguaje M.

```
%{
    Metodo de la Biseccion
    Parametros de Entrada
        @param f: funcion a la cual se le aplicara el algoritmo
        @param a: limite inferior del intervalo
        @param b: limite superior del intervalo
        @param MAXIT:  iteraciones maximas
        @param TOL:  tolerencia del algoritmo

    Parametros de Salida
        @return xAprox: valor aproximado de x
        @return error: porcentaje de error del resultado obtenido
%}

clc;
clear;

function [xAprox, err] = biseccion(f, a, b, MAXIT, TOL)

    if(f(a) * f(b) < 0)

        iter = 1;
```

```
        err = 1;
        iterl = []; % Lista que almacena el numero de iteraciones para despues graficar
        errl = []; % Lista que almacena el % de error de cada iteracion para despues graficar

        while(iter < MAXIT)
            xAprox = (a + b) / 2;
            fx = f(xAprox);

            if(f(a) * fx < 0)
                b = xAprox;
            elseif(f(b) * fx < 0)
                a = xAprox;
            endif

            iterl(iter) = iter;
            errl(iter) = err;
            err = (b - a) / (2)^(iter-1);

            if(err < TOL)
                grafica(iterl, errl);
                return;
            else
                iter = iter + 1;
            endif
      endwhile
      grafica(iterl, errl);
    else
        error("Condiciones en los parametros de entrada no garantizan el cero de la funcion.")
    endif
    return;
endfunction

%{
    Parametros de Entrada
        @param listaValoresX: valores del eje 'x'
        @param listaValoresY: valores del eje 'y'

    Parametros de Salida
        @return: Grafico de los datos ingresados
%}
function grafica(listaValoresX, listaValoresY)
    plot(listaValoresX, listaValoresY, 'bx');
    title("Metodo de la Biseccion");
    xlabel("Iteraciones");
    ylabel("% Error");
endfunction

%Valores iniciales
a = 0;
b = 2;
%Iteraciones maximas
MAXIT = 100;
%Tolerancia
TOL = 0.0001;
```

```
%Funcion
funct = @(x) e^x - x - 2;
%Llamado de la funcion
[xAprox, err] =  biseccion(funct, a, b, MAXIT, TOL);
printf("######################################### \n");
printf("Metodo de la Biseccion \n");
printf('xAprox = %f\n%%Error = %d \n', xAprox, err);
```

1.2. Método 2: Newton-Raphson

Código 2: Lenguaje Python.

```
##############################################################################
import math
import matplotlib.pyplot as plt
from scipy.misc import derivative
##############################################################################


def newton_raphson(func, x0, MAXIT, TOL):
    '''
    Metodo de Newton-Raphson
    :param func: es la funcion a analizar
    :param x0: valor inicial
    :param MAXIT: es la cantidad de iteraciones maximas a realizar
    :param TOL: es la tolerancia del algoritmo
    :return: xAprox: es la solucion, valor aproximado de x
    :return: error: pocentaje de error del resultado obtenido
    '''
    itera = 1
    err = 1
    iterl = []  # Lista que almacena el numero de iteraciones
    errl = []  # Lista que almacena el % de error de cada iteracion
    xAprox = x0

    while (itera < MAXIT):
        xk = xAprox
        fd = derivative(func, xk, dx=1e-6)
        xAprox = xk - (func(xk)) / (fd)
        err = (abs(xAprox - xk)) / (abs(xAprox))
        iterl.append(itera)
        errl.append(err)

        if(err < TOL):
            grafica(iterl, errl)
            return xAprox, err
        else:
            itera = itera + 1

    grafica(iterl, errl)
    return xAprox, err
```

```
def grafica(listaValoresX, listaValoresY):
    '''
    Grafica
    :param listaValoresX: valores que se graficaran en el eje 'x'
    :param listaValoresY: valores que se graficaran en el eje 'y'
    :return: Grafico con lo valores ingresados
    '''
    plt.plot(listaValoresX, listaValoresY, 'bx')
    plt.title("Metodo de Newton-Raphson")
    plt.xlabel("Iteraciones")
    plt.ylabel("% Error")
    plt.show()


if __name__ == '__main__':
    # Valor inicial
    x0 = 1
    # Tolerancia
    TOL = 0.0001
    # Maximo iteraciones
    MAXIT = 100
    # Funcion
    def func(x): return (math.e)**x - 1/x
    # Llamado de la funcion
    xAprox, err = newton_raphson(func, x0, MAXIT, TOL)
    print("############################################################")
    print("Metodo de Newton-Raphson \n")
    print('xAprox = {}\n%Error = {}'.format(xAprox, err))
```

1.3. Método 3: Secante

Código 3: Lenguaje C++.

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

using namespace std;
using namespace GiNaC;

/**
 * @param funcion: Funcion a evaluar en el metodo
 * @param x0: primer valor inicial
 * @param x1: segundo valor inicial
 * @param MAXIT: cantidad maxima de iteraciones
 * @param TOL: tolerancia del resultado
 * @return tuple<ex, ex>: valor aproximado, error del valor aproximado
 */
tuple<ex, ex> secante(string funcion, ex x0, ex x1, ex MAXIT, ex TOL)
{
    symbol x;
    symtab table;
    table["x"] = x;
    parser reader(table);
```

```
    ex f = reader(funcion);
    ex xk = x1;
    ex xkm1 = x0;
    ex xk1;
    int iter = 0;
    ex err = TOL + 1;

    while (iter < MAXIT)
    {
        xk1 = xk -
              ((((xk - xkm1)) / ((evalf(subs(f, x == xk))))) - evalf(subs(f, x == xkm1)))
        xkm1 = xk;
        xk = xk1;
        err = abs(evalf(subs(f, x == xk)));

        if (err < TOL)
        {
            break;
        }
        else
        {
            iter = iter + 1;
        }
    }
    xk;
    err = abs((evalf(subs(f, x == xk))));
    return make_tuple(xk, err);
}

int main(void)
{
    tuple<ex, ex> testS = secante("exp(-pow(x, 2)) - x", 0, 1, 100, 0.001);
    cout << "Aproximacion: " << get<0>(testS) << endl;
    cout << "Error: " << get<1>(testS) << endl;
    return 0;
}
```

1.4. Método 4: Falsa Posición

Código 4: Lenguaje C++.

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

using namespace std;
using namespace GiNaC;

/**
 * @param funcion: Funcion a evaluar en el metodo
 * @param x0: primer valor inicial
 * @param x1: segundo valor inicial
 * @param MAXIT: cantidad maxima de iteraciones
 * @param TOL: tolerancia del resultado
```

```
 * @return tuple<ex, ex>: valor aproximado, error del valor aproximado
 */
tuple<ex, ex> secante(string funcion, ex x0, ex x1, ex MAXIT, ex TOL)
{
    symbol x;
    symtab table;
    table["x"] = x;
    parser reader(table);
    ex f = reader(funcion);
    ex xk = x1;
    ex xkm1 = x0;
    ex xk1;
    int iter = 0;
    ex err = TOL + 1;

    while (iter < MAXIT)
    {
        xk1 = xk -
              ((((xk - xkm1)) / ((evalf(subs(f, x == xk))))) - evalf(subs(f, x == xkm1)))
        xkm1 = xk;
        xk = xk1;
        err = abs(evalf(subs(f, x == xk)));

        if (err < TOL)
        {
            break;
        }
        else
        {
            iter = iter + 1;
        }
    }
    xk;
    err = abs((evalf(subs(f, x == xk))));
    return make_tuple(xk, err);
}

int main(void)
{
    tuple<ex, ex> testS = secante("exp(-pow(x, 2)) - x", 0, 1, 100, 0.001);
    cout << "Aproximacion: " << get<0>(testS) << endl;
    cout << "Error: " << get<1>(testS) << endl;
    return 0;
}
```

1.5. Método 5: Punto Fijo

Código 5: Lenguaje Python.

```
#################################################################################
import matplotlib.pyplot as plt
import numpy as np
#################################################################################
```

```
def punto_fijo(funcion, valor_inicial, iteraciones_maximas):
    '''
    Metodo del punto fijo
    :param funcion: Funcion por aproximar - funcion lambda
    :param valor_inicial: Valor por el cual se empezara a aproximar - int, float, double
    :param iteraciones_maximas: Numero maximo de itreaciones - int
    :return: aproximacion: aproximacion de la solucion
    '''
    lista_error = []  # lista para graficar
    iteracion = 1
    b = funcion(valor_inicial)  # valor para obtener error
    error = abs(b - valor_inicial)
    while (iteracion <= iteraciones_maximas):  # condicion de parada
        valor_inicial = b  # reajuste de valores de error
        b = funcion(valor_inicial)
        error = abs(b - valor_inicial)
        lista_error.append(error)
        iteracion += 1

    aproximacion = b
    # Construccion de tabla
    plt.plot(lista_error, label='errores por interacion')
    plt.ylabel('Error')
    plt.xlabel('Iteracion')
    # Los ejes estan limitados por las iteraciones y el error maximo
    plt.axis([0, iteraciones_maximas, 0, lista_error[0]])
    plt.title('Punto Fijo')
    plt.legend()
    plt.show()
    print('Aproximacion: ' + str(aproximacion) + ', error: ' + str(error))
    return aproximacion, error


if __name__ == '__main__':
    # Valor inicial
    x0 = 0
    # Maximo iteraciones
    MAXIT = 100
    # Funcion
    def funcion(x): return np.exp(-x)
    # Llamado de la funcion
    print("############################################################")
    print("Metodo del Punto Fijo \n")
    punto_fijo(funcion, x0, MAXIT)
```

1.6. Método 6: Muller

Código 6: Lenguaje M.

```
%{
    Metodo de Muller
```

```
    Parametros de Entrada
        @param func: funcion a la cual se le aplicara el algoritmo
        @param x0: primer valor inicial
        @param x1: segundo valor inicial
        @param x2: segundo valor inicial
        @param MAXIT:  iteraciones maximas
        @param TOL:  tolerencia del algoritmo

    Parametros de Salida
        @return r: valor aproximado de x
        @return error: porcentaje de error del resultado obtenido
%}

clc;
clear;

function [r, err] = muller(func, x0, x1, x2, MAXIT, TOL)
    iter = 1;
    err = 1;
    iterl = []; % Lista que almacena el numero de iteraciones para despues graficar
    errl = []; % Lista que almacena el % de error de cada iteracion para despues graficar

    while(iter < MAXIT)

        a = ((x1-x2)*[func(x0)-func(x2)]-(x0-x2)*[func(x1)-func(x2)])/((x0-x1)*(x0-x2)*(x1-x2));
        b = (((x0-x2)^2)*[func(x1)-func(x2)]-((x1-x2)^2)*[func(x0)-func(x2)])/((x0-x1)*(x0-x2)*(x1-x2));
        c = func(x2);

        discriminante = b^2 - 4*a*c;

        if(discriminante < 0)
            error("Error, la solucion no es real.")
            return;
        endif

        r = x2 - (2*c) / (b + (sign(b))*(sqrt(discriminante)));
        err = (abs(r - x2)) / (abs(r));
        errl(iter) = err;
        iterl(iter) = iter;
        iter = iter + 1;

        if(err < TOL)
            grafica(iterl, errl);
            return;
        endif

        x0Dist = abs(r - x0);
        x1Dist = abs(r - x1);
        x2Dist = abs(r - x2);

        if (x0Dist > x2Dist && x0Dist > x1Dist)
            x0 = x2;
        elseif (x1Dist > x2Dist && x1Dist > x0Dist)
            x1 = x2;
```

```
        endif
        x2 =  r;
    endwhile

    grafica(iterl, errl);
    return;
endfunction

%{
    Parametros de Entrada
        @param listaValoresX: valores del eje 'x'
        @param listaValoresY: valores del eje 'y'

    Parametros de Salida
        @return: Grafico de los datos ingresados
%}
function grafica(listaValoresX, listaValoresY)
    plot(listaValoresX, listaValoresY, 'bx');
    title("Metodo de Muller");
    xlabel("Iteraciones");
    ylabel("% Error");
endfunction

%Valores iniciales
x0 = 2;
x1 = 2.2;
x2 = 1.8;
%Iteraciones maximas
MAXIT = 100;
%Tolerancia
TOL = 0.0000001;
%Funcion
func = @(x) sin(x) - x/2;
%Llamado de la funcion
[r, err] = muller(func, x0, x1, x2, MAXIT, TOL);
printf("########################################### \n");
printf("Metodo de Muller \n");
printf('r = %f\n %%Error = %i \n', r, err);
```

2. Optimización

2.1. Método 1: Descenso Coordinado

Código 7: Lenguaje M.

```
%{
    Metodo del Descenso Coordinado
    Parametros de Entrada
      @param func: funcion a la cual se le aplicara el algoritmo
      @param vars: variables que oomponen la funcion
      @param xk: valores iniciales
      @param MAXIT:  iteraciones maximas
```

```
    Parametros de Salida
      @return xAprox: valor aproximado de xk
      @return error: porcentaje de error del resultado obtenido
%}

clc;
clear;

pkg load symbolic;
syms x y;
warning("off","all");

function [xAprox, err] = coordinado(func, vars, xk, MAXIT)
    n = length(vars);
    iter = 0;
    iterl = [];
    err = [];
    while(iter < MAXIT)
        xk_aux = xk;
        v = 1;
        while(v != n + 1)
            ec_k = func;
            j = 1;
            while(j != n + 1)
                if(j != v)
                    vars(j);
                    xk(j);
                    ec_k = subs(ec_k, vars(j), xk(j));
                endif
                j = j + 1;
            endwhile
            fv = matlabFunction(ec_k);
            min = fminsearch(fv, 0);
            xk(v) = min;
            v = v + 1;
        endwhile
        cond = xk - xk_aux;
        norma = norm(cond, 2);
        errl(iter+1) = norma;
        iterl(iter+1) = iter;
        iter = iter + 1;
    endwhile
    xAprox = xk;
    err = norma;
    grafica(iterl, errl);
    return;
endfunction

%{
    Parametros de Entrada
        @param listaValoresX: valores del eje 'x'
        @param listaValoresY: valores del eje 'y'
```

```
    Parametros de Salida
        @return: Grafico de los datos ingresados
%}
function grafica(listaValoresX, listaValoresY)
    plot(listaValoresX, listaValoresY, 'bx');
    title("Metodo del Descenso Coordinado");
    xlabel("Iteraciones");
    ylabel("% Error");
endfunction

%Valores iniciales
xk = [1, 1];
%Variables
vars = [x, y]
%Iteraciones maximas
MAXIT = 9;
%Tolerancia
TOL = 0.000001;
%Funcion
funct = '(x - 2)**2 + (y + 3)**2 + x * y';
%Llamado de la funcion
[xAprox, err] = coordinado(funct, vars, xk, MAXIT, TOL);
printf("########################################## \n");
printf("Metodo del Descenso Coordinado \n");
printf('xAprox X = %f\nxAprox Y = %f\n%%Error = %d \n', xAprox, err);
```

2.2. Método 2: Gradiente Conjugado No Lineal

Código 8: Lenguaje Python.

```
###############################################################################
import math
import matplotlib.pyplot as plt
from scipy.misc import derivative
from sympy import sympify, Symbol, diff
from numpy import linalg, array
###############################################################################


def gradiente(func, variables, xk, MAXIT):
    '''
    Metodo del Gradiente Conjugado No Lineal
    :param func: string con la funcion a evaluar
    :param variables: lista con las variables de la ecuacion
    :param xk: vector con los valores iniciales
    :param MAXIT: es la cantidad de iteraciones maximas a realizar
    :return: xAprox: es la solucion, valor aproximado de x
    :return: error: pocentaje de error del resultado obtenido
    '''
    funcion = sympify(func)  # Obtenemos la funcion del string
    itera = 0
    iterl = []  # Lista que almacena el numero de iteraciones
    errl = []  # Lista que almacena el % de error de cada iteracion
```

```
    if(len(variables) != len(xk)):  # Comprueba la cantidad de variables en xk
        return "Variables y xk deben ser del mismo tamano"

    listaSimb = []
    n = len(variables)
    for i in range(0, n):
        # Se crean los Symbol de las variables de la funcion
        listaSimb += [Symbol(variables[i])]

    gradiente = []
    for i in range(0, n):  # Se calcula el gradiente de la funcion
        gradiente += [diff(funcion, variables[i])]

    # Se calculan los valores iniciales de gk y dk
    gk = evaluarGradiente(gradiente, variables, xk)
    dk = [i * -1 for i in gk]

    while(itera < MAXIT):
        # Se calcula el alpha
        ak = calcularAlphaK(funcion, variables, xk, dk, gk)
        # Se calcula el nuevo valor del vector: x1 = x0 + a * d0
        alphakdk = [i * ak for i in dk]
        vecx = [x1 + x2 for(x1, x2) in zip(xk, alphakdk)]
        # Se calcula el nuevo valor del vector gk
        gkx = evaluarGradiente(gradiente, variables, vecx)
        # Se calcula el vector para encontrar el error
        vecFinal = evaluarGradiente(gradiente, variables, vecx)
        # Se calcula la norma para el error
        norma = linalg.norm(array(vecFinal, dtype='float'), 2)
        bk = calcularBetaK(gkx, gk)  # Se calcula el valor de beta
        # Se calcula el nuevo valor del vector dk
        betakdk = [i * bk for i in dk]
        mgk = [i * -1 for i in gkx]
        dk = [x1 + x2 for (x1, x2) in zip(mgk, betakdk)]
        xk = vecx.copy()
        gk = gkx.copy()
        iterl.append(itera)
        errl.append(norma)
        itera += 1
    grafica(iterl, errl)
    return vecx, norma


def evaluarGradiente(gradiente, variables, xk):
    '''
    Evaluar Gradiente
    :param gradiente: gradiente a evaluar
    :param variables: lista con las variables de la ecuacion
    :param xk: vector con los valores iniciales
    :return: gradResult: resultado de evaluar el vector en el gradiente
    '''
    n = len(variables)
    gradResult = []
```

```
    # Se recorre cada una de las derivadas parciales en el gradiente
    for i in range(0, n):
        funcion = gradiente[i]  # Se obtiene la derivada parcial
        # Se sustituyen cada una de las variables por el valor en el vector
        for j in range(0, n):
            funcion = funcion.subs(variables[j], xk[j])
        gradResult += [funcion.doit()]
    return gradResult


def calcularAlphaK(func, variables, xk, dk, gk):
    '''
    Calcular alpha k
    :param func: funcion en la que se evaluar
    :param variables: lista con las variables de la ecuacion
    :param xk: vector con los valores iniciales
    :param dk:
    :param gk: gradiente a evaluar
    :return: gradResult: resultado de evaluar el vector en el gradiente
    '''
    a = 1
    while 1:
        adk = [i * a for i in dk]  # Se calcula la multiplicacion de ak * dk
        # Se calcula la operacion xk + a * dk
        vecadk = [x1 + x2 for (x1, x2) in zip(xk, adk)]
        # Se evalua la funcion f(xk + a * dk)
        refvecadk = evaluarFuncion(func, variables, vecadk)
        # Se evalua la funcion f(xk)
        refvec = evaluarFuncion(func, variables, xk)
        # Se calcula la parte izquierda de la desigualdad
        izquierdaDesigualdad = refvecadk - refvec
        # Se calcula la operacion gk * dk
        multiplicargkdk = [x1 * x2 for(x1, x2) in zip(gk, dk)]
        # Se suman todos los elementos de la multiplicacon anterior
        sumagkdk = sum(multiplicargkdk)
        # Se calcula la multiplicacion de 0.5 * ak * gk * dk (parte derecha)
        derechaDesigualdad = 0.5 * a * sumagkdk
        if(izquierdaDesigualdad < derechaDesigualdad):  # Se verifica la desigualdad
            break
        a /= 2
    return a


def evaluarFuncion(func, variables, xk):
    '''
    Evaluar en la funcion
    :param func: string con la funcion a evaluar
    :param variables: lista con las variables de la ecuacion
    :param xk: vector con los valores iniciales
    :return: func: resultado de evaluar en la funcion
    '''
    n = len(variables)
    # Se sustituyen cada una de las variables por el valor en el vector
    for i in range(0, n):
```

```
        func = func.subs(variables[i], xk[i])
    return func


def calcularBetaK(gk, prevGK):
    '''
    Calcular beta k
    :param gk: vector gk
    :param prevGK: vector gk de la iteracion anterior
    :return: b: valor del Bk canculado
    '''
    # Se calcula la norma 2 del vector actual
    normagk = linalg.norm(array(gk, dtype='float'), 2)
    # Se calcula la norma 2 del vector anterior
    normaprevGK = linalg.norm(array(prevGK, dtype='float'), 2)
    b = (pow(normagk, 2)) / (pow(normaprevGK, 2))
    return b


def grafica(listaValoresX, listaValoresY):
    '''
    Grafica
    :param listaValoresX: valores que se graficaran en el eje 'x'
    :param listaValoresY: valores que se graficaran en el eje 'y'
    :return: Grafico con lo valores ingresados
    '''
    plt.plot(listaValoresX, listaValoresY, 'bx')
    plt.title("Metodo del Gradiente Conjugado No Lineal")
    plt.xlabel("Iteraciones")
    plt.ylabel("% Error")
    plt.show()


if __name__ == '__main__':
    # Valores iniciales
    xk = [0, 3]
    # Variables de la ecuacion
    variables = ['x', 'y']
    # Maximo iteraciones
    MAXIT = 14
    # Funcion
    func = '(x-2)**4 + (x-2*y)**2'
    # Llamado de la funcion
    xAprox, err = gradiente(func, variables, xk, MAXIT)
    print("#########################################################")
    print("Metodo del Gradiente Conjugado No Lineal \n")
    print('xAprox = {}\n%Error = {}'.format(xAprox, err))
```

3. Sistemas de Ecuaciones

3.1. Método 1: Eliminación Gaussiana

Código 9: Lenguaje M.

```
%{
    Metodo de Eliminacion Gaussiana
    Parametros de Entrada
        @param matrizD: matriz de coeficientes
        @param matrizI: vector de terminos independientes

    Parametros de Salida
        @return vectorResultado: solucion del sistema
%}

clc;
clear;
warning('off', 'all');

function X = gaussiana(matrizD, matrizI)
  [n, m] = size(matrizD);
  if (n ~= m)
    disp("La matrizD debe ser cuadrada");
  endif

  n = length(matrizD);
  X = [matrizD, matrizI];
  % Por cada argumento de la matriz
  for(i = 1 : n)
    pivot = X(i, i);
    pivotRow = X(i, :);
    % Multiplica los vectores
    M = zeros(1, n - i);
    m = length(M);
    % Obtiene cada fila multiplicada
    for(k = 1 : m)
      M(k) = X(i + k, i) / pivot;
    endfor
    % Modifica cada fila
    for(k = 1 : m)
      X(i + k, :) = X(i + k, :) - pivotRow*M(k);
    endfor
  endfor
  X = sustitucionAtras(X(1 : n, 1 : n), X(:, n + 1));
endfunction

%{
    Metodo de Sustitucion Atras
    -Resuelve un sistema del tipo Ax = b
    Parametros de Entrada
        @param matrizA: matriz triangular superior NxN
        @param matrizB: matriz Nx1

    Parametros de Salida
        @return X: solucion de la matriz
%}

function X = sustitucionAtras(matrizA, matrizB)
```

```
  n = length(matrizB);
  X = zeros(n, 1);
  X(n) = matrizB(n)/matrizA(n, n);

  for(k = n-1 : -1 : 1)
    div = matrizA(k, k);
    if (div != 0)
      X(k) = (matrizB(k) - matrizA(k, k + 1 : n)*X(k + 1 : n))/matrizA(k, k);
    else
      disp("Error: se ha producido una division por cero");
    endif
  endfor
endfunction

%Matriz de coeficientes
A = [2 -6 12 16 ; 1 -2 6 6; -1 3 -3 -7; 0 4 3 -6];
%Matriz de terminos independientes
B = [70 26 -30 -26]';
%Llamado de la funcion
X = gaussiana(A, B);
printf("########################################## \n");
printf("Metodo de la Eliminacion Gaussiana \n");
printf('X = %f\n', X);
```

3.2. Método 2: Factorización LU

Código 10: Lenguaje Python.

```
################################################################################
import numpy as np
import scipy.linalg as la
################################################################################


def fact_lu(matrizD, matrizI):
    '''
    Metodo de la Factorizacion LU
    :param matrizD: matriz de coeficientes
    :param matrizI: matriz de terminos independientes
    :return: X: solucion del sistema
    '''
    if(np.linalg.det(matrizD) == 0):
        print("La matriz no es singular")
        return
    else:
        pass
    n = len(matrizD)
    L = np.eye(n)
    U = matrizD

    for i in range(1, n):
        pivot = U[i - 1][i - 1]
        pivotRow = U[i - 1]
```

```
        M = np.zeros((1, n - i))
        m = M.size + 1

        for k in range(1, m):
            try:
                M[i - 1][k - 1] = (U[i + k - 1][i - 1]) / pivot
            except:
                M = (U[i + k - 1][i - 1]) / pivot

        for k in range(1, m):
            try:
                U[i + k - 1] = U[i + k - 1] - \
                    (np.multiply(pivotRow, M[i - 1][k - 1]))
                L[i + k - 1][i - 1] = M[i - 1][k - 1]
            except:
                U[i + k - 1] = U[i + k - 1] - (np.multiply(pivotRow, M))
                L[i + k - 1][i - 1] = M

    Y = ((np.linalg.inv(L)).dot(np.transpose(matrizI)))
    X = (np.linalg.inv(U)).dot(Y)
    return X


if __name__ == '__main__':
    # Matriz de coeficientes
    A = [[4, -2, 1], [20, -7, 12], [-8, 13, 17]]
    # Vector de terminos independientes
    B = [11, 70, 17]
    # Llamado de la funcion
    X = fact_lu(A, B)
    print("############################################################")
    print("Metodo de la Factorizacion LU\n")
    print('X = {}\n'.format(X))
```

3.3. Método 3: Factorización Cholesky

Código 11: Lenguaje C++.

```
#include <iostream>
#include <armadillo>
#include <cmath>

using namespace std;
using namespace arma;

/**
 * @param A: Una matriz A de cualquier tamano, simetrica y positiva definida
 * @return mat: Una matriz L que es la factorizacion de la matriz A
 */
mat cholesky(mat A)
{
    mat L(A.n_rows, A.n_cols, fill::zeros);
    for (int i = 0; i < A.n_rows; i++)
```

```
    {
        for (int j = 0; j < i + 1; j++)
        {
            double suma = 0;
            if (j == i)
            {
                for (int k = 0; k < j; k++)
                {
                    suma += L(j, k) * L(j, k);
                }
                L(j, j) = sqrt(A(j, j) - suma);
            }
            else
            {
                for (int k = 0; k < j; k++)
                {
                    suma += L(i, k) * L(j, k);
                }
                L(i, j) = (A(i, j) - suma) / L(j, j);
            }
        }
    }
    return L;
}

/**
 * @param L: Una matriz L que es la factorizacion de Cholesky de otra matriz
 * @param y: Un vector d que es el vector de terminos independientes
 * @return colvec: Un vector y que es la solucion de este sistema de ecuaciones
 */
colvec sust_atras(mat L, colvec y)
{
    colvec x(L.n_rows, fill::zeros);
    for (int i = L.n_rows - 1; i > -1; i--)
    {
        int suma = 0;
        for (int j = i; j < L.n_rows; j++)
        {
            suma += L(i, j) * x(j);
        }
        x(i) = (y(i) - suma) / L(i, i);
    }
    return x;
}

/**
 * @param L: Una matriz L que es la transpuesta de la factorizacion de Cholesky de otra
 * @param b: Un vector y que es el vector de terminos independientes
 * @return colvec: Un vector x que es la solucion de este sistema de ecuaciones
 */
colvec sust_adelante(mat L, colvec b)
{
    colvec y(L.n_rows, fill::zeros);
    for (int i = 0; i < L.n_rows; i++)
```

```
    {
        double suma = 0;
        for (int j = 0; j < i; j++)
        {
            suma += L(i, j) * y(j);
        }
        y(i) = (b(i) - suma) / L(i, i);
    }
    return y;
}

/**
 * @param A: Una matriz A de cualquier tamano
 * @param b: Un vector d que es el vector de terminos independientes
 */
void fact_Cholesky(mat A, colvec b)
{
    //Revisa si es simetrica positiva definida con una funcion propia de Armadillo
    if (!A.is_sympd())
    {
        A = A * trans(A);
        b = b * trans(A);
    }
    //Llama a las demas funciones y las guarda en variables
    cout << "Matriz: \n"
         << A << endl;
    cout << "Vector: \n"
         << b << endl;
    mat L = cholesky(A);
    cout << "Matriz factorizada: \n"
         << L << endl;
    colvec y = sust_adelante(L, b);
    cout << "Vector independiente: \n"
         << y << endl;
    colvec x = sust_atras(trans(L), y);
    cout << "Solucion del sistema: \n"
         << x << endl;
}

int main()
{
    //Matriz A que es simetrica positiva definida
    mat A = "25 15 -5 -10; 15 10 1 -7; -5 1 21 4; -10 -7 4 18";
    //Vector de terminos independientes
    colvec d = "-25 -19 -21 -5";
    //Realiza la factorizacion de Cholesky
    fact_Cholesky(A, d);
    return 0;
}
```

3.4. Método 4: Método de Thomas

Código 12: Lenguaje Python.

```
#########################################################################################################
import numpy as np
#########################################################################################################


def thomas(matrizC, vectorTI):
    '''
    Metodo de Thomas
    :param matrizC: matriz de coeficientes
    :param vectorTI: matriz de terminos independientes
    :return: X: solucion del sistema
    '''
    A = matrizC
    for i in range(len(A)):
        for j in range(len(A[0])):
            if (i == j and (matrizC[i][j] == 0)):
                print("La matriz no es tridiagonal 1")
                return
            elif (j == (i + 1) and matrizC[i][j] == 0):
                print("La matriz no es tridiagonal 2")
                return
            elif (j == (i - 1) and matrizC[i][j] == 0):
                print("La matriz no es tridiagonal 3")
                return
            elif ((j > i + 1) and (matrizC[i][j] != 0)):
                print("La matriz no es tridiagonal 4")
                return
            elif (((j < i - 1) and (matrizC[i][j] != 0))):
                print("La matriz no es tridiagonal 5")
                return
    xn = []
    ci = 0
    di = 0
    qi = 0
    bi = 0
    pi = 0
    n = len(matrizC)
    if (len(matrizC) == len(vectorTI)):
        for i in range(0, n):
            if (i == 0):
                ci = matrizC[i][i + 1]
                bi = matrizC[i][i]
                di = vectorTI[i]
                pi = ci / bi
                qi = di / bi
                xn.append(qi)
            elif (i <= n - 2):
                ai = matrizC[i + 1][i]
                bi = matrizC[i][i]
                di = vectorTI[i]
                ci = matrizC[i][i + 1]
                pi = ci / (bi - pi * ai)
                qi = (di - qi * ai) / (bi - pi * ai)
```

```
                xn.append(qi - pi * xn[i - 1])
            else:
                ai = matrizC[i][i - 1]
                bi = matrizC[i][i]
                di = vectorTI[i]
                qi = (di - qi * ai) / (bi - pi * ai)
                xn.append(qi * xn[i - 1])
        return xn
    else:
        print("Error: el vector y la matriz deben ser del mismo tamano")


def creaTridiagonal(N, a, b, c):
    '''
    Funcion para crear la matriz tridiagonal
    :param N: tamano de la matriz
    :param a: valor debajo de la diagonal principal
    :param b: valor de la diagonal principal
    :param c: valor sobre la diagonal principal
    :return: matriz: matriz tridiagonal
    '''
    matriz = np.zeros((N, N))
    np.fill_diagonal(matriz, b)
    n = N
    for i in range(0, n - 1):
        matriz[i][i + 1] = c
        matriz[i + 1][i] = a
    return matriz


def creaD(N, ext, inte):
    '''
    Funcion para crear el vector d
    :param N: tamano del vector
    :param ext: valor en los extremos del vector
    :param inte: valor en el interior del vector
    :return: d: vector d
    '''
    n = N
    d = []
    for i in range(0, n):
        if ((i == 0) or (i == n - 2)):
            d.append(ext)
        else:
            d.append(inte)
    return d


if __name__ == '__main__':
    # Creacion de la matriz tridiagonal
    matrizC = creaTridiagonal(7, 1, 5, 1)
    # Creacion del vector D
    vectorTI = creaD(7, -12, -14)
    # Llamado del metodo
```

```
    print("#########################################################")
    print("Metodo de Thomas\n")
    X = thomas(matrizC, vectorTI)
    print('X = {}\n'.format(X))
```

3.5. Método 5: Método de Jacobi

Código 13: Lenguaje C++.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

/**
 * @param A: matriz de coeficientes
 * @param b: vector de terminos independientes
 * @param xInicial: vector de valores iniciales
 * @param MAXIT: cantidad de iteraciones maximas
 * @param TOL: tolerancia de la respuesta
 * @return tuple<vec, double>: vector solucion, error de la solucion
 */
tuple<vec, double> jacobi(mat A, vec b, vec xInicial, int MAXIT, double TOL)
{

    mat D(size(A), fill::zeros);
    mat U(size(A), fill::zeros);
    mat L(size(A), fill::zeros);

    for (int i = 0; i < A.n_rows; i++)
    {
        for (int j = 0; j < A.n_cols; j++)
        {
            if (j < i)
            {
                L(i, j) = A(i, j);
            }
            else if (j > i)
            {
                U(i, j) = A(i, j);
            }
            else if (i == j)
            {
                D(i, j) = A(i, j);
            }
            else
            {
                cout << "Error" << endl;
            }
        }
    }
```

```
    vec xk = xInicial;
    vec xk1;
    int iter = 0;
    double err = TOL + 1;

    while (iter < MAXIT)
    {
        xk1 = ((-D.i()) * (L + U) * (xk)) + ((D.i()) * (b));
        xk = xk1;
        err = norm(A * xk - b);

        if (err < TOL)
        {
            break;
        }
        else
        {
            iter = iter + 1;
        }
    }
    return make_tuple(xk, err);
}

/**
 * Ejemplo numerico
 */
int main()
{
    tuple<vec, double> testJ = jacobi("5 1 1; 1 5 1; 1 1 5", "7 7 7", "0 0 0", 100, 0.00
    cout << "Aproximacion: \n"
         << get<0>(testJ) << endl;
    cout << "Error: " << get<1>(testJ) << endl;
    return 0;
}
```

3.6. Método 6: Método de Gauss-Seidel

Código 14: Lenguaje M.

```
%{
    Metodo de Gauss-Seidel
    Parametros de Entrada
        @param matrizD: matriz de coeficientes
        @param matrizI: vector de terminos independientes
        @param x: valor inicial
        @param MAXIT: iteraciones maximas
        @param TOL: tolerancia de la respuesta

    Parametros de Salida
        @return xAprox: valor aproximado de X
        @return error: porcentaje de error del resultado obtenido
%}
```

```
clc;
clear;
warning('off', 'all');

function [xAprox, err] = gaussSeidel(matrizD, matrizI, x, MAXIT, TOL)

  if(estrDiag(matrizD) == 0)
    disp("La matriz no es estrictamente diagonal dominante");
    return;
  else
    L = tril(matrizD, -1);
    D = diag(diag(matrizD));
    U = triu(matrizD, 1);
    b = matrizI';
    iter = 0;
    xAprox = x';
    xAnt = xAprox;
    err = TOL + 1;
    M = L + D;
    inversa = inv(M);
    iterl = [];
    errl = [];

    while(iter < MAXIT)
        xAprox = (-inversa*U*xAnt)+(inversa*b);
        iterl(iter+1) = iter;
        errl(iter+1) = err;
        err = norm(xAprox - xAnt);
        xAnt = xAprox;
        if(err < TOL)
            grafica(iterl, errl);
            return;
        else
            iter = iter + 1;
        endif
    endwhile
    grafica(iterl, errl);
    return;
  endif
endfunction

%{
    Parametros de Entrada
        @param A: matriz a determinar si es tridiagonal dominante o no.

    Parametros de Salida
        @return ft: retorna un valor de 1 o 0 si la matriz es dominante o no.
%}
function ft = estrDiag(A)
  n = length(A);
  m = length(A(1));
  d = 0;

  for(i = 1 : n)
```

```
    suma = 0;
    for(j = 1 : m)
      if(i == j)
        d = A(i, j);
      else
        suma = suma + (A(i, j)).^2;
      endif
    endfor

    if abs(d) < sqrt(suma)
      ft = 0;
      return;
    endif
  endfor
  ft = 1;
  return;
endfunction

%{
    Parametros de Entrada
        @param listaValoresX: valores del eje 'x'
        @param listaValoresY: valores del eje 'y'

    Parametros de Salida
        @return: Grafico de los datos ingresados
%}
function grafica(listaValoresX, listaValoresY)
    plot(listaValoresX, listaValoresY, 'x-');
    title("Metodo de Gauss-Seidel");
    xlabel("Iteraciones");
    ylabel("% Error");
endfunction

%Valor inicial
x = [0 0 0];
%Iteraciones maximas
MAXIT = 10;
%Tolerancia
TOL = 0.000001;
%Matriz de coeficientes
A = [5 1 1; 1 5 1; 1 1 5];
%Vector de terminos independientes
B = [7 7 7];
%Llamado de la funcion
[xAprox, err] =  gaussSeidel(A, B, x, MAXIT, TOL);
printf("############################################ \n");
printf("Metodo de Gauss-Seidel \n");
printf('xAprox = %f\nxAprox = %f\nxAprox = %f\n%%Error = %d \n', xAprox, err);
```

3.7. Método 7: Método de Relajacion

Código 15: Lenguaje Python.

```
###############################################################################
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
###############################################################################


def relajacion(A, b, maxI, tol, w):
    '''
    Metodo de Relajacion
    :param A: matriz de cofactores
    :param b: matriz de respuestas
    :param maxI: maxima cantidad de iteraciones
    :param tol: tolerancia para calcular error
    :param w: constante por usar en el metodo
    :return:
    '''
    # Se debe verificar que la matriz sea definida positiva
    if (np.all(np.linalg.eigvals(A) > 0)):
        pass
    else:
        print("La matriz no es definida positiva")
        return
    # Se debe verificar que la matriz sea tridiagonal
    for i in range(len(A)):
        for j in range(len(A[0])):
            i0 = j - 1
            i1 = j + 1
            if (i == j):
                if i0 < 0 or i0 >= len(A[0]):
                    if A[i][j] == 0 or A[i1][j] == 0:
                        print(1)
                        print("La matriz no es tridiagonal")
                        return
                elif i1 < 0 or i1 >= len(A[0]):
                    if A[i][j] == 0 or A[i0][j] == 0:
                        print(2)
                        print("La matriz no es tridiagonal")
                        return
                else:
                    if A[i][j] == 0 or A[i0][j] == 0 or A[i1][j] == 0:
                        print(3)
                        print("La matriz no es tridiagonal")
                        return
            else:
                if abs(i - j) > 1:
                    if A[i][j] != 0:
                        print(i)
                        print(j)
                        print("La matriz no es tridiagonal")
                        return

    # Calculo de D, L y U
    (P, L, U) = la.lu(A)
```

```
    D = np.diag(np.diag(U))
    # k es la iteracion actual
    k = 0
    # Un requisito es que la matriz D+wL sea invertible
    x = D + w * L
    try:
        inverse = np.linalg.inv(x)
    except np.linalg.LinAlgError:
        print("D+wL no es invertible")
        return
    else:
        # lista de errores para graficar
        lista_error = []
        x = []
        for i in A:
            x.append(0)
        x = np.transpose(x)
        # iteraciones, esperando que se cumplan las iteraciones
        while k < maxI:
            x0 = np.transpose(x)
            M = w ** -1 * (w * L + D)
            N = w ** -1 * ((1 - w) * D - w * U)
            x = np.matmul(np.matmul(np.linalg.inv(M), N), x0) + \
                np.matmul(np.linalg.inv(M), b)
            errorM = b - np.matmul(A, np.transpose(x))
            errorM = np.transpose(errorM)
            error = 0
            # Revisar si el error se cumple, sino se sigue iterando
            for i in errorM:
                error = error + i ** 2
            lista_error.append(error ** 0.5)
            if (error ** 0.5 < tol):
                break
            k = k + 1

        # Construccion de tabla
        plt.plot(lista_error, label='errores por interacion')
        plt.ylabel('Error')
        plt.xlabel('Iteracion')
        # Los ejes estan limitados por las iteraciones y el error maximo
        plt.axis([0, maxI, 0, max(lista_error)])
        plt.title('Relajacion')
        plt.legend()
        plt.show()
        print('x: ' + str(x) + ', error: ' + str(error))
        return x, error


if __name__ == '__main__':
    # Constante w
    w = 1.24
    # Maximo iteraciones
    MAXIT = 5
    # Tolerancia
```

```
    TOL = 0.0001
    # Matriz de cofactores
    A = [[4, 3, 0], [3, 4, -1], [0, -1, 4]]
    # Vector de respuestas
    b = [7, 7, 7]
    # Llamado de la funcion
    print("############################################################")
    print("Metodo del Punto Fijo \n")
    relajacion(A, b, MAXIT, TOL, w)
```

3.8. Método 8: Método de la Pseudoinversa

Código 16: Lenguaje C++.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

/**
 * @param A: matriz de coeficientes
 * @param b: vector de terminos independientes
 * @param MAXIT: cantidad de iteraciones maximas
 * @param TOL: tolerancia de la respuesta
 * @return tuple<vec, double>: vector solucion, error de la solucion
 */

tuple<vec, double> pseudoinversas(Mat<double> A, vec b, int MAXIT, double TOL)
{

    int iter = 0;
    double err = TOL + 1;

    // Se definie el valor de alpha para el metodo iterativo de Schlutz
    double alpha = eig_sym(A * trans(A)).max();
    //Se genera la el vector inicial
    Mat<double> xk = (1 / alpha) * trans(A);
    //matriz identidad
    // vec I = eye(A.n_cols);

    Mat<double> I(A.n_rows, A.n_rows, fill::eye);
    //variable para la interacion
    mat xk1;

    while (iter < MAXIT)
    {
        xk1 = xk * (2 * I - A * xk);
        // cout << xk1 << endl;
        xk = xk1;
        err = norm(A * xk * A - A, 2);

        if (err < TOL)
```

```
        {
            break;
        }
        else
        {
            iter = iter + 1;
        }
    }
    mat A_pseudo = xk;
    vec x = A_pseudo * b;

    return make_tuple(x, err);
}

/**
 * Ejemplo numerico
 */
int main()
{
    Mat<double> A = {{1, 2, -1}, {-3, 1, 5}};
    Col<double> b = {1, 4};
    tuple<vec, double> testP = pseudoinversas(A, b, 100, 0.00001);
    cout << "Aproximacion: \n"
         << get<0>(testP) << endl;
    cout << "Error: " << get<1>(testP) << endl;
    return 0;
}
```

4. Polinomio de Interpolación

4.1. Método 1: Método de Lagrange

Código 17: Lenguaje C++.

```
#include <iostream>
#include <ginac/ginac.h>
#include <armadillo>
#include <cmath>

using namespace std;
using namespace GiNaC;
using namespace arma;

/**
 * @param xv: Un vector de cualquier tamano que corresponde al eje X del polinomio
 * @param k: La cantidad de puntos que se han evaluado en el polinomio
 * @return q: La funcion de Lagrange que se le suma al polinomio de interpolacion
 * */
ex Lk(vec xv, int k)
{
    //Se carga la variable simbolica
    symbol x("x");
```

```
    symtab table;
    table["x"] = x;
    parser reader(table);
    //Se definen las variables
    int m = xv.size();
    ex q = 1;
    for (int i = 0; i < m; i++)
    {
        if (i != k)
        {
            //Se realiza la multiplicatoria de la funcion de Lagrange
            q *= (x - xv(i)) / (xv(k) - xv(i));
        }
        else
        {
            continue;
        }
    }
    return q;
}

/**
 * @param xv: Un vector de cualquier tamano que corresponde al eje X del polinomio
 * @param yv: Un vector de cualquier tamano que corresponde al eje Y del polinomio
 * @return p: El polinomio de interpolacion del conjunto de puntos
 */
ex lagrange(vec xv, vec yv)
{
    //Se carga la variable simbolica
    symbol x("x");
    symtab table;
    table["x"] = x;
    parser reader(table);
    //Se definen las variables
    int m = xv.size();
    int n = m - 1;
    ex p = 0;
    for (int i = 0; i < n; i++)
    {
        //Se realiza la sumatoria del polinomio de interpolacion
        p += yv(i + 1) * Lk(xv, i + 1);
    }
    return simplify_indexed(expand(p));
}

int main()
{
    //Vectores x y y tomados como ejemplo
    vec xv = "-2 0 1";
    vec yv = "0 1 -1";
    //Se carga la funcion y se ejecuta el ejemplo
    ex pol_int = lagrange(xv, yv);
    cout << "Polinomio de interpolacion : " << pol_int << endl;
    return 0;
```

```
}
```

4.2. Método 2: Método de Diferencias Divididas de Newton

Código 18: Lenguaje M.

```
%{
    Metodo de Diferencias Divididas de Newton
    Parametros de Entrada
        @param listaPO: vector con los pares ordenados xk, yk

    Parametros de Salida
        @return polinomio: polinomio de interpolacion
%}

clc;
clear;
pkg load symbolic;
warning("off","all");

function polinomio = dd_newton(listaPO)
    [n, m] = size(listaPO);

    if(m ~= 2)
        disp("Error, la cantidad de puntos ingresada no es correcta");
        return;
    else
        x = sym('x');
        rk = [];
        for(i = 1 : n)
            rk = [rk listaPO(i, 2)];
        endfor

        polinomio = listaPO(1, 2);
        multi1 = 1;
        m = n - 1;
        for(i = 2 : n)
            multi1 = multi1 * (x - listaPO(i - 1, 1));
            rk1 = [];
            for(j = 1: m)
                numerador = rk(j) - rk(j + 1);
                denominador = listaPO(j, 1) - listaPO(j + i - 1, 1);
                rk1 = [rk1 (numerador / denominador)];
            endfor
            m = m - 1;
            polinomio =  polinomio + rk1(1) * multi1;
            rk = rk1;
        endfor
        polinomio = expand(polinomio);
        return;
    endif
endfunction
```

```
%Vector de pares ordenados
listaPO = [-2 0; 0 1; 1 -1];
% listaPO = [1 2/3; 3 1; 5 -1; 6 0];
%Llamado de la funcion
polinomio =  dd_newton(listaPO);
printf("########################################## \n");
printf("Metodo Diferencias Divididas de Newton \n");
printf('Polinomio de Lagrange\n');
disp(polinomio);
```

4.3. Método 3: Trazador Cúbico Natural

Código 19: Lenguaje Python.

```
##############################################################################
import math
import numpy as np
import matplotlib.pyplot as plt
import sympy
import numpy as np
from math import *
from sympy import *
import sympy as sym
from sympy import symbols
from Jacobi import jacobi
##############################################################################


def traz_cubico(xk, yk):
    '''
    Metodo del Trazador Cubico
    :param xk: valores en x de los pares ordenados
    :param yk: valores en y de los pares ordenados
    :return: Sx: trazadores cubicos
    '''
    points = []
    n = len(xk)
    for i in range(0, n):
        points.append([xk[i], yk[i]])

    points = np.array([np.array(p) for p in points])
    delta_hk = points[1:, 0] - points[:-1, 0]
    delta_yk = points[1:, 1] - points[:-1, 1]
    A, u = [], []
    k = delta_hk.shape[0]
    for i in range(1, k):
        # Primer caso Ms[1] = 0
        if i == 1:
            A.append([2 * (delta_hk[i - 1] + delta_hk[i]),
                      delta_hk[i]] + [0] * (k - 3))
        # Segundo caso Ms[n+1] = 0
        elif i == k - 1:
            A.append([0] * (k - 3) + [delta_hk[i - 1],
```

```
                    2 * (delta_hk[i - 1] + delta_hk[i])])
        else:
            A.append(
                [0] * (i - 2) + [delta_hk[i - 1], 2 * (delta_hk[i - 1] + delta_hk[i]), d
        # Creando el vector u
        u.append(6 * (delta_yk[i] / delta_hk[i] -
                 delta_yk[i - 1] / delta_hk[i - 1]))

    A = np.array([np.array(a) for a in A])
    u = np.array(u)
    # Resolviendo el sistema mediante Jacobi
    x0 = np.zeros(u.shape)
    Ms = jacobi(A, u, u * 0, 0.0000001)
    # Append Ms[1] = 0 and Ms[n+1] = 0
    Ms = np.append(0, np.append(Ms, 0))
    a, b, c, d = [[], [], [], []]
    xk = points[:, 0]
    yk = points[:, 1]
    # Calculando los coeficientes
    for i in range(k):
        a.append((Ms[i + 1] - Ms[i]) / (6 * delta_hk[i]))
        b.append(Ms[i] / 2)
        c.append((yk[i + 1] - yk[i]) / delta_hk[i] -
                 (2 * delta_hk[i] * Ms[i] + delta_hk[i] * Ms[i + 1]) / 6)
        d.append(yk[i])
    a = np.array(a)
    b = np.array(b)
    c = np.array(c)
    d = np.array(d)

    # Polinomio trazador
    x = sympy.Symbol('x')
    Sx_tabla = []
    for i in range(0, len(xk) - 1, 1):
        with evaluate(False):
            pxtramo = a[i] * (x - xk[i]) ** 3 + b[i] * (x - xk[i]) ** 2 + c[i] * (x - xk
            Sx_tabla.append(pxtramo)

    print('Trazadores cubicos por tramos \n')
    for tramo in range(1, len(xk), 1):
        print('Sx = [' + str(xk[tramo - 1]) + ',' + str(xk[tramo]) + ']')
        print(str(Sx_tabla[tramo - 1]))

    return a, b, c, d, Sx_tabla


if __name__ == '__main__':
    # Valores xk
    xk = [1, 1.05, 1.07, 1.1]
    # Valores yk
    yk = [2.718282, 3.286299, 3.527609, 3.905416]
    # Funcion
    def func(x): return 3*x*(math.pow(math.e, x)) - 2*(math.pow(math.e, x))
    # Llamado de la funcion
```

```
    print("#########################################################")
    print("Metodo del Trazador Cubico \n")
    traz_cubico(xk, yk)
```

4.4. Método 4: Cota Error Polinomio de Interpolación

Código 20: Lenguaje M.

```
%{
    Cota de Error del Metodo de la Interpolacion
    Parametros de Entrada
        @param func: funcion a la cual se le aplicara el algoritmo
        @param puntos: numero de puntos

    Parametros de Salida
        @return xAprox: valor aproximado de x
        @return error: porcentaje de error del resultado obtenido
%}

% |f(x) - p(x)| <= |f^(n+1)(Ex)*1/(n+1!)*(x-x0)(x-x1)...(x-xn)|
%%fmax = f^(n+1)(Ex) ==> donde la funcion derivada a la n+1 es maxima
%%a = 1/(n+1!)       ==> alpha
%%p = (x-x0)(x-x1)...(x-xn)
% Ex pertence a [a,b]

clc
clear;
%
%Se carga el paquete simbolico%



function [error] = cota_poly_inter(func, puntos)
pkg load symbolic
syms x
%devuelve la cantidad de puntos
n = length(puntos);
puntos = sort(puntos);

a = puntos(1);
b = puntos(n);

m = 1/factorial(n+1);
p = '';

%Haciendo el polinomio p
for i = 1:length(puntos)
  var = int2str(puntos(i))
  class(var)
  if i == 1
     p = [p, '(x-',var,')'];
  else
      p = [p, '*(x-',var,')'];
```

```
  end
end



%Se pasa la funcion a simbolico
h = sym(p);
%Se pasa la funcion simbolica a ecuacion
h1=matlabFunction(h);
%se deriva la funcion
hd = diff(h,x)==0;
%puntos criticos
puntos_criticos_h = double(cell2mat(solve(hd,x)));
%Extremos del intervalo
puntos_a_evaluar_h=[a b puntos_criticos_h'];
valores_evaluados_h= [h1(puntos_a_evaluar_h)];
[pmax,p_max]=max(valores_evaluados_h);




%Encontrar el maximo de la funcion en un intervalo



%%%Para Fmax
%Se pasa la funcion a simbolico
f = sym(func);
%Se pasa la funcion simbolica a ecuacion
f1=matlabFunction(f);
%se deriva la funcion
fd = diff(f,x,n+1)==0;
%puntos criticos
puntos_criticos = double(cell2mat(solve(fd,x)));
%Extremos del intervalo
puntos_a_evaluar=[a b puntos_criticos'];
valores_evaluados= [f1(puntos_a_evaluar)];
[fmax,x_max]=max(valores_evaluados);


error = abs(fmax*m*p);

endfunction

func = 'x^2+4';
puntos = [2, 3, 5];

[error] = cota_poly_inter(func, puntos)
```

4.5. Método 5: Cota Error Trazador Cúbico Natural

Código 21: Lenguaje Python.

```
##############################################################################
import numpy as np
import sympy as sym
##############################################################################


def cota_tras_cubico(func, S):
    '''
    Metodo de Cota de Error del Trazador Cubico
    :param func: funcion con al menos 4ta derivada
    :param S: Lista de puntos
    :return: cota_error: Cota de error
    '''
    h = 0
    x = sym.symbols('x')
    f = sym.sympify(func)      # La funcion se castea en x
    f4 = sym.diff(f, x, 4)       # La 4ta derivada
    norm = 0
    for i in range(len(S)):            # Norma infinira
        if (f4.subs(x, S[i]) > norm):
            norm = f4.subs(x, S[i])
        else:
            pass

    for i in range(len(S) - 1):  # Valor h
        if (S[i+1] - S[i] > h):
            h = S[i+1] - S[i]
        else:
            pass
    cota_error = 5*h**4 * norm / 384     # Valor cota
    return float(cota_error)


if __name__ == '__main__':
    S = [1, 2, 3, 4, 5, 6]
    func = 'x**4  - (1 / x)'
    cota_error = cota_tras_cubico(func, S)
    print("##########################################################")
    print("Metodo de Cota de Error del Trazador Cubico\n")
    print('Cota de error = {}\n'.format(cota_error))
```

5. Integración Númerica

5.1. Regla del Trapecio y Cota de Error

Código 22: Lenguaje M.

```
%{
    Metodo de la Regla del Trapezio
    Parametros de Entrada
        @param func: funcion a la cual se le calculara el area
        @param a: valor inferior
```

```
        @param b: valor superior

    Parametros de Salida
        @return xAprox: valor aproximado del area
        @return err: porcentaje de error del resultado obtenido
%}

clc;
clear;
pkg load symbolic;
warning("off","all");

function [xAprox, err] = trapecio(func, a, b)
    syms f(x);
    f(x) = func;
    xAprox = ((b - a)/(2))*((func(a)) + (func(b)));
    fd = diff(f, 2);
    fd = function_handle(fd);
    lista = [];
    lista(1) = abs(fd(a));
    lista(2) = abs(fd(b));
    vmax = max(lista);
    err = (((b - a)**3)/12)*vmax;
    return;
endfunction


%Intervalo inferior
a = 2;
%Intervalo superior
b = 5;
%Funcion
func = @(x) log(x);
%Llamado de la funcion
[xAprox, err] = trapecio(func, a, b);
% trapecio(func, a, b);
printf("########################################## \n");
printf("Metodo de la Regla del Trapecio \n");
printf('xAprox = %f\n%%Error = %i \n', xAprox, err);
```

5.2. Regla de Simpson y Cota de Error

Código 23: Lenguaje Python.

```
################################################################################
from numpy import double
from sympy import symbols, sympify, diff
################################################################################


def simpson(funcion, a, b):
    '''
    Metodo de la Regla de Simpson
```

```
    :param funcion: funcion a la cual se le calculara el area
    :param a: intervalo inferior
    :param b: intervalo superior
    :return: xAprox: aproximacion del area
    :return: error: error de la solucion
    '''
    x = symbols('x')
    f = sympify(funcion)
    m = (a + b)/2
    xAprox = double(((b - a)/6)*(f.subs(x, a) + 4*f.subs(x, m) + f.subs(x, b)))
    fdddd = diff(f, x, x, x, x)
    lista = []
    lista.append(abs(double(fdddd.subs(x, a))))
    lista.append(abs(double(fdddd.subs(x, b))))
    vmax = max(lista)
    error = double(((((b - a)/2)**5)/(90))*(vmax))
    return xAprox, error


if __name__ == '__main__':
    # Intervalo inferior
    a = 2
    # Intervalo superior
    b = 5
    # Funcion
    funcion = "ln(x)"
    # Llamado de la funcion
    print("############################################################")
    print("Metodo de la Regla de Simpson \n")
    xAprox, error = simpson(funcion, a, b)
    print('xAprox = {}\n%Error = {}'.format(xAprox, error))
```

5.3. Regla Compuesta del Trapecio y Cota de Error

Código 24: Lenguaje C++.

```
#include <iostream>
#include <ginac/ginac.h>
#include <armadillo>
#include <cmath>

using namespace std;
using namespace GiNaC;
using namespace arma;

/**
 * @param func: La funcion a evaluar en la regla del trapecio
 * @param a: El primer elemento del intervalo a evaluar
 * @param b: El ultimo elemento del intervalo a evaluar
 * @return p: El polinomio de interpolacion del conjunto de puntos
 */
double trapecio(string func, double a, double b) {
    //Se carga la variable simbolica
```

```
    symbol x("x");
    symtab table;
    table["x"] = x;
    parser reader(table);
    ex f = reader(func);
    //Se evalua la variable simbolica en los intervalos
    ex f0 = evalf(subs(f, x == a));
    ex f1 = evalf(subs(f, x == b));
    //Se define la variable h
    ex h = (b - a)/2;
    //Se calcula la integral
    double I = ex_to<numeric>(h * (f0 + f1)).to_double();
    return I;
}

/**
 * @param func: La funcion a evaluar en la regla del trapecio compuesta
 * @param a: El primer elemento del intervalo a evaluar
 * @param b: El ultimo elemento del intervalo a evaluar
 * @param N: La cantidad de puntos que se le van a pasar al intervalo
 * @return p: El polinomio de interpolacion del conjunto de puntos
 */
tuple<double, double> trapecio_compuesto(string func, double a, double b, int N) {
    //Se carga la variable simbolica
    symbol x("x");
    symtab table;
    table["x"] = x;
    parser reader(table);
    ex f = reader(func);
    //Se define la variable h
    double h = (b - a)/(N - 1);
    //Se define el vector de puntos sobre el que se va a operar
    vec xv(N, fill::zeros);
    for (int i = 0; i < N; i++) {
        xv(i) = a;
        a += h;
    }
    //Se realiza el calculo de la integral mediante la regla del trapecio
    double I = 0;
    for (int i = 0; i < N - 1; i++) {
        I += trapecio(func, xv(i), xv(i + 1));
    }
    //Se obtiene la segunda derivada de la funcion
    ex fd = f.diff(x, 2);
    vec xd(2, fill::zeros);
    //Se obtiene el alfa maximo a utilizar en la cota de error mediante el maximo de la
    xd(0) = ex_to<numeric>(evalf(subs(fd, x == a))).to_double();
    xd(1) = ex_to<numeric>(evalf(subs(fd, x == b))).to_double();
    double alphaMax = *max_element(xd.begin(), xd.end());
    //Se obtiene la cota de error
    double error = (((b - a) * (h * h))/12) * alphaMax;
    return make_tuple(I, error);
}
```

```cpp
int main() {
    //Funcion a integrar
    string f = "log(x)";
    //Regla del trapecio y cota de error
    tuple<double, double> testTrapecio = trapecio_compuesto(f, 2, 5, 1000);
    cout << "Regla del Trapecio: " << get<0>(testTrapecio)  << endl;
    cout << "Cota de Error: " << get<1>(testTrapecio)  << endl;
    return 0;
}
```

5.4. Regla Compuesta de Simpson y Cota de Error

Código 25: Lenguaje Python.

```python
##############################################################################
import sympy as sym
##############################################################################

def simpson_compuesto(funcion,numero_puntos,intervalo):
    '''
    Metodo de Simpson Compuesto para calculo de integrales
    :param funcion: funcion con al menos 4ta deriva
    :param numero_puntos: Cantidad de puntos a utilizar
    :param intervalo: lista con valor inicial a final que define los limites de la integ
    :return: integral, error: Resultado de integral y error
    '''
    x = sym.symbols('x')
    f = sym.sympify(funcion)
    suma_pares=0
    suma_impares=0
    x0= intervalo[0]
    h= (intervalo[1]-intervalo[0])/(numero_puntos-1)
    lista_x=[]
    lista_x.append(x0)
    k=1
    while k<=numero_puntos:
        temp = x0+k*h
        lista_x.append(temp)
        k=k+1
    for i in range(len(lista_x)-1):
        if i==0:
            pass
        else:
            if i%2==0:
                suma_pares=suma_pares+f.subs(x,i)
            else:
                suma_impares=suma_impares+f.subs(x,i)
    integral= h/3 * (f.subs(x,lista_x[0])+2*suma_pares+4*suma_impares+f.subs(x,lista_x[-
    integral = float(integral)
    f4 = sym.diff(f,x,4)
    error = (intervalo[1]-intervalo[0])/180 * abs(f4.subs(x,intervalo[0]))
    print("La integral da como resultado "+str(integral)+" con un error de "+str(error))
    return integral,error
```

```
if __name__ == '__main__':
    funcion = 'ln(x)'
    # Llamado de la funcion
    print("############################################################")
    print("Metodo de Simpson Compuesto y Cota de Error \n")
    simpson_compuesto(funcion,7,[2,5])
```

5.5. Cuadratura Gaussiana y Cota de Error

Código 26: Lenguaje M.

```
%{
    Metodo de Diferencias Divididas de Newton
    Parametros de Entrada
        @param funcion: funcion a calcular la integral
        mediante el metodo de cradraturas gaussianas
        @param a: intervalo inferior de la integral
        @param b: intervalo superior de la integral
        @param n: orden del polinomio

    Parametros de Salida
        @return xAprox: aproximacion de la integral
        @return err: error del calculo
%}

clc;
clear;
warning("off","all");
pkg load miscellaneous;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Se utiliza el paquete miscellaneous, este se puede
# instalar utilizando el comando:
# pkg install -forge miscellaneous
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xAprox, err] = cuad_gaussiana(funcion, a, b, n)
    xAprox = 0;
    i = 1;
    legendrePol = legendrepoly(n);
    cerosPol = roots(legendrePol);
    derPol = polyder(legendrePol);
    while(i <= n)
        xi = cerosPol(i);
        Pi = polyval(derPol, xi);
        wi = 2/((1-(xi^2))*(Pi)^2);
        xAprox = xAprox + wi*funcion(xi);
        i = i + 1;
    endwhile
    q = integral(funcion, a, b);
    xAprox;
    err = abs(xAprox - q);
```

```
endfunction

%Intervalos de la integral
a = -1;
b = 1;
%Grado del polinomio
n = 3;
%Funcion
funct = @(x) exp(x).*cos(x);
%Llamado de la funcion
[xAprox, err] =  cuad_gaussiana(funct, a, b, n);
printf("########################################## \n");
printf("Metodo de las Cuadraturas Gaussianas \n");
printf('xAprox = %f\n%%Error = %d \n', xAprox, err);
```

6. Diferenciación Númerica

7. Valores y Vectores Propios