Catálogo Grupal de Algoritmos

Integrantes:

- Jessica Espinoza Quesada - Carnet 2018135811
- Jose David Sánchez Schnitzler - Carnet 2018142388
- Tomás Felipe Segura Monge - Carnet 2018099729

1. Tema 1: Ecuaciones No lineales

1.1. Método 1: Bisección

Código 1: Biseccion.

```
def biseccion(f, a, b, tol, iterMax):

    """
    Esta funcion aproxima la solucion de la ecuacion f(x)=0,
    utilizando el motodo de la biseccion.

    Sintaxis:  biseccion(f, a, b, tol, iterMax)

    Parametros Iniciales:
                f = es una cadena de caracteres (string) que representa a la funcion f
                a, b = son los extremos del intervalo [a, b]
                tol = un numero positivo que representa a la tolerancia para el criterio |f
                iterMax = cantidad de iteraciones maximas

    Parametros de Salida:
                [x, k, error], donde
                x = aproximacion del cero de la funcion f
                k = numero de iteraciones realizados
                error = |f(x)|
    """

    from sympy import sympify
    import matplotlib.pyplot as plt

    func = sympify(f)
    fa = func.subs({'x': a})
    fb = func.subs({'x': b})

    if fa * fb > 0:
        x = []
        k = []
        error = []
```

```
        display('El teorema de Bolzano no se cumple, es decir, f(a) * f(b) > 0')
    else:
        error = tol + 1
        k = 0
        it = []
        er = []
        while error > tol and k < iterMax:
            k = k + 1
            x = (a + b) / 2
            fa = func.subs({'x': a})
            fx = func.subs({'x': x})
            error = abs(fx)
            it.append(k)
            er.append(error)
            if fa * fx < 0: # Se cumple la condicion en el intervalo 1.
                b = x
            else: # Se cumple la condicion en el intervalo 2.
                a = x
    plt.plot(it, er)
    plt.show()
    return [x, k, error]

help(biseccion)

biseccion('exp(x)-2*x-10', 2, 4, 10**-5, 1000)
```

1.2. Método 2: Newton-Raphson

Código 2: Newton Raphson

```
pkg load symbolic

function [xk, k, error] = newton_raphson(f, xk0, tol, iterMax)

    %Esta funcion aproxima la solucion de la ecuacion f(x) = 0, utilizando el metodo de Newton-Raphson
    %
    %Sintaxis: newton_raphson(f, xk0, tol, iterMax)
    %
    %Parametros Iniciales:
    %           f = es un string que representa a la funcion f
    %           xk0 = es el valor inicial de la iteracion, este sera el primer valor evaluado en la
        funcion
    %           tol = un numero positivo que representa a la tolerancia para el criterio |f(x_k)| < tol
    %           iterMax = cantidad de iteraciones maximas
    %
    %Parametros de Salida:
    %           x_k = aproximacion del cero de la funcion f
    %           k = numero de iteraciones realizados
    %           error = |f(x)|

    func = matlabFunction(sym(f));
    dfunc = matlabFunction(diff(sym(func)));

    k = 0;
    error = tol + 1;
    e = [];

    if dfunc(xk0) < tol
        xk = []
        k = []
        error = []
        display('La derivada se indefine.')
    else
        while and(error > tol, k < iterMax)
            k = k + 1;
            if dfunc(xk0) < tol
                break;
            else
                xk = xk0 - (func(xk0) / dfunc(xk0));
                error = abs(func(xk));
                e = [e error];
                xk0 = xk;
            end
        end
        plot(1 : k, e)
    end
end

help newton_raphson
% Comando que corre la funcion Newton Raphson, editar al gusto.
[xk, k, error] = newton_raphson('-cos(x) + x**2', 5, 10**-8, 50)
```

1.3. Método 3: Secante

Código 3: Metodo de la secante.

```
#include <iostream>
#include <armadillo>
#include "matplotlibcpp.h"
#include <cmath>

using namespace std;
using namespace arma;
namespace plt = matplotlibcpp;

/**
 * @brief En esta funcion se define la f(x) que se desea analizar con el
 * metodo de la secante.
 *
 * @param x  valor de x
 * @return double resultado de evaluar f(x)
 */
double myFunction(double x)
{
  return pow(exp(1), (-1 * pow(x, 2.0))) - x;
}

/**
 * @brief Funcion para aproximar el valor de f(x) por el metodo de la secante
 *
 * @param x_k0 extremo menor del intervalo para evaluar
 * @param x_k1 extremos mayor del intervalor para evaluar
 * @param tol numero positivo que representa la tolerancia para |f(x)| < tol
 * @param iterMax numero maximo de iteraciones para cerrar
 *
 * @return vector con el error y la aproximacion final
 */
std::vector<double>  secante(double x_k0, double x_k1, double tol, int iterMax)
{
  //Definimos las variables para el error y el x_k siguiente.
  double error_k = 1;
  double x_knext = 1;
  //Definimos los vectores para almacenar los datos.
  std::vector<double> results = {};
  std::vector<double> errors = {};
  std::vector<double> iteractions = {};

  int k = 0;
  //Iteramos hasta alcanzar las dos condiciones
  // 1. que el error_k sea menor a la tolerancia
  // 2. cumplir con las iteraciones maximas
  while(k < iterMax || error_k >= tol){

    //Definimos el x_k+1 como se define por el metodo de la secante.
    x_knext=x_k1-((x_k1-x_k0)/(myFunction(x_k1)-myFunction(x_k0)))*myFunction(x_k1);

    //Definimos el error de x_k+1 como error_k a partir de
```

```
    //la formula de error relativo.
    error_k = (abs(x_knext - x_k1)) / (abs(x_knext));

    //Insertamos los errores y los resultados en sus respectivos vectores.
    results.insert(results.begin() + k, x_knext);
    errors.insert(errors.begin() + k, error_k);
    iteractions.insert(iteractions.begin() + k, k + 1);

    //Redefinimos las varaibles para la siguiente iteracion
    x_k0 = x_k1;
    x_k1 = x_knext;

    k++;
  }

  //Tras completar el ciclo, contruimos el grafico
  plt::figure();
  plt::plot(iteractions, errors, "b");
  plt::title("Aproximacion por el metodo de la secante");
  plt::xlabel("Numero de iteracion");
  plt::ylabel("Aproximacion");
  plt::show();

  std::vector<double> finalResult = {x_knext, error_k};
  return finalResult;
}

int main()
{
  secante(0, 1, pow(10, -3), 7);
  return 0;
}
```

1.4. Método 4: Falsa Posición

Código 4: Metodo de la falsa posicion.

```cpp
#include <iostream>
#include "matplotlibcpp.h"
#include <cmath>

using namespace std;
namespace plt = matplotlibcpp;

/**
 * @brief En esta funcion se define la f(x) que se desea analizar con el
 * metodo de la falsa posicion.
 *
 * @param x  valor de x
 * @return double resultado de evaluar f(x)
 */
double myFunction(double x)
{
  return cos(x) - x;
}

/**
 * @brief Funcion para aproximar el valor de f(x) por el metodo de la falsa posicion
 *
 * @param x0 extremo menor del intervalo para evaluar
 * @param x1 extremos mayor del intervalor para evaluar
 * @param TOL numero positivo que representa la TOLerancia para |f(x)| < TOL
 * @param MAXITER numero maximo de iteraciones para cerrar
 *
 * @return vector con el error y la aproximacion final
 */
std::vector<double> falsa_posicion(double x0, double x1, double TOL, int MAXITER)
{
  //Definimos las variables para el error y el x_k siguiente.
  double error = 1;
  double x2 = 1;
  //Definimos los vectores para almacenar los datos.
  std::vector<double> results = {};
  std::vector<double> errors = {};
  std::vector<double> iterations = {};

  int k = 0;
  //Iteramos hasta alcanzar las dos condiciones
  // 1. que el error sea menor a la TOLerancia
  // 2. cumplir con las iteraciones maximas
  while(k < MAXITER || error < TOL){

    //Definimos el x_k+1 como se define por el metodo de la falsa posicion.
    x2 = x1 - ((x1 - x0) / (myFunction(x1) - myFunction(x0))) * myFunction(x1);

    //Definimos el error de x_k+1 como error a partir de la formula de error relativo.
    error = (abs(x2 - x1) / (x2));
```

```
    //Insertamos los errores y los resultados en sus respectivos vectores.
    results.insert(results.begin() + k, x2);
    errors.insert(errors.begin() + k, error);
    iterations.insert(iterations.begin() + k, k + 1);

    //Redefinimos las varaibles para la siguiente iteracion

    if(myFunction(x2) * myFunction(x1) < 0){
     x0 = x2;
      }

    if(myFunction(x0) * myFunction(x2) < 0){
      x1 = x2;
    }
    cout << x2 << endl;

    k++;
  }

  //Tras completar el ciclo, contruimos el grafico
  plt::figure();
  plt::plot(iterations, errors, "b");
  plt::title("Aproximacion por el metodo de la secante");
  plt::xlabel("Numero de iteracion");
  plt::ylabel("Aproximacion");
  plt::show();

  std::vector<double> finalResult = {x2, error};
  return finalResult;
}

int main()
{
  falsa_posicion(1/2, M_PI/4, pow(10, -3), 10);
  return 0;
}
```

1.5. Método 5: Punto Fijo

Código 5: Punto Fijo.

```
pkg load symbolic

function [xk, k, error] = punto_fijo(phi, xk0, a, b, tol, iterMax)

    %Esta funcion aproxima la solucion de la ecuacion f(x) = 0, utilizando el metodo del punto fijo, pero
        es necesario indicar el phi(x) de la funcion, f(x) = phi(x) - x
    %
    %Sintaxis: punto_fijo(phi, xk0, a, b, tol, iterMax)
    %
    %Parametros Iniciales:
    %           f = es un string que representa a la funcion f
    %           xk0 = es el valor inicial de la iteracion, estos seran los primeros valores evaluados en
        la funcion
    %           tol = un numero positivo que representa a la tolerancia para el criterio |f(x_k) - x_k|
       < tol
    %           iterMax = cantidad de iteraciones maximas
    %
    %Parametros de Salida:
    %           x_k = aproximacion del cero de la funcion f
    %           k = numero de iteraciones realizados
    %           error = |f(x) - x_k|

    func = matlabFunction(sym(phi));
    dfunc = matlabFunction(diff(sym(func)));

    syms x
    sol = solve(dfunc, x);
    pcm = func(sol);

    if and(a > func(a), func(a) > b)
        xk = [];
        k = [];
        error = [];
        display("El valor a, se sale del intervalo dado")
    elseif and(a > func(b), func(b) > b)
        xk = [];
        k = [];
        error = [];
        display("El valor b, se sale del intervalo dado")
    elseif and(-1 >= dfunc(a), dfunc(a) >= 1)
        xk = [];
        k = [];
        error = [];
        display("El valor a, se sale del intervalo ]-1, 1[")
    elseif and(-1 >= dfunc(b), dfunc(b) >= 1)
        xk = [];
        k = [];
        error = [];
        display("El valor b, se sale del intervalo ]-1, 1[")
    else
        ultpc = pcm(end, end);
```

```
i = 1;
validar = 1;
while pcm(i, end) != ultpc
    cpc = pcm(i, end);
    if and(a > cpc, cpc > b)
        display("Hay un punto critico que se sale del intervalo dado")
        validar = 0;
        break;
    else
        i += 1;
    end
end

if and(a > ultpc, ultpc > b)
    display("Hay un punto critico que se sale del intervalo dado")
    validar = 0;
else
    if validar == 1
        d2func = matlabFunction(diff(sym(dfunc)));
        sol2 = solve(d2func, x);
        pcm2 = dfunc(sol2);

        ultpc2 = pcm2(end, end);
        i = 1;
        validar = 1;
        while pcm2(i, end) != ultpc2
            cpc2 = pcm2(i, end);
            if and(-1 >= cpc2, cpc2 >= 1)
                display("Hay un punto critico que se sale del intervalo ]-1, 1[")
                validar = 0;
                break;
            else
                i += 1;
            end
        end

        if and(-1 >= ultpc, ultpc >= 1)
            display("Hay un punto critico que se sale del intervalo ]-1, 1[")
            validar = 0;
        else
            if validar == 1
                k = 0;
                error = tol + 1;
                e = [];
                while and(error > tol, k < iterMax)
                    k = k + 1;
                    xk = func(xk0);
                    error = abs(func(xk) - xk);
                    e = [e error];
                    xk0 = xk;
                end
                plot(1 : k, e)
            else
                xk = [];
```

```
                        k = [];
                        error = [];
                    end
                end
            else
                xk = [];
                k = [];
                error = [];
            end
        end
    end
end

help punto_fijo

[xk, k, error] = punto_fijo('cos(x) + 10', 1, -1, 1, 10^-8, 50)
```

1.6. Método 6: Muller

Código 6: Muller.

```
def muller(f, xk0, xk1, xk2, tol, iterMax):

    """
    Esta funcion aproxima la solucion de la ecuacion f(x) = 0, utilizando el metodo de muller, pero e

    Sintaxis:  muller(f, xk0, xk1, xk2, tol, iterMax)

    Parametros Iniciales:
                f = es un string que representa a la funcion f
                xk0, xk1, xk2 = son los valores iniciales de la iteracion, estos seran los primeros v
                tol = un numero positivo que representa a la tolerancia para el criterio |f(x_k)| < t
                iterMax = cantidad de iteraciones maximas

    Parametros de Salida:
                [x_k, k, error], donde
                x_k = aproximacion del cero de la funcion f
                k = numero de iteraciones realizados
                error = |f(x)|
    """

    from sympy import sympify
    import matplotlib.pyplot as plt
    import numpy
    import math

    func = sympify(f)

    k = 0
    error = tol + 1
    er = []
    it = []
    while error > tol and k < iterMax:
        k = k + 1
        fxk0 = func.subs({'x': xk0}).evalf()
        fxk1 = func.subs({'x': xk1}).evalf()
        fxk2 = func.subs({'x': xk2}).evalf()
        a=(((xk1-xk2)*(fxk0-fxk2))-((xk0-xk2)*(fxk1-fxk2)))/((xk0-xk1)*(xk0-xk2)*(xk1-xk2))
        b=((((xk0-xk2)**2)*(fxk1-fxk2))-(((xk1-xk2)**2)*(fxk0-fxk2)))/((xk0-xk1)*(xk0-xk2)*(xk1-xk2))
        c = fxk2
        xk = xk2 - ((2 * c) / (b + (numpy.sign(b) * math.sqrt((b ** 2) - (4 * a * c)))));
        fxk = func.subs({'x' : xk}).evalf()
        error = abs(fxk)
        er.append(error)
        it.append(k)

        if abs(xk - xk0) < abs(xk - xk1):
            if abs(xk - xk1) < abs(xk - xk2):
                xk2 = xk
            else:
                xk1 = xk2
                xk2 = xk
        else:
            if abs(xk - xk1) < abs(xk - xk2):
                if abs(xk - xk0) < abs(xk - xk2):
                    xk2 = xk
                else:
```

```
                    xk0 = xk2
                    xk2 = xk
            else:
                xk0 = xk2
                xk2 = xk

    plt.plot(it, er)
    plt.show()
    return [xk, k, error]

help(muller)

muller('sin(x) - cos(x)', 1, 2, 3, 10**-8, 50)
```

2. Tema 2: Optimización

2.1. Método 1: Descenso Coordinado

Código 7: coordinado

```
def coordinado(f, xk, vars, tol, iterMax):

    """
    Esta funcion aproxima la solucion de la funcion f(x), donde
    x = [x1, x2, x3, ..., xn], utilizando el metodo del descenso coordinado.

    Sintaxis: coordinado(f, xk, vars, tol, iterMax)

    Parametros Iniciales:
                f = es un string que representa a la funcion f de varias variables.
                xk = es una lista con los valores iniciales de la funcion.
                vars = es una lista con strings de todas las variables que se
                encuentran en la funcion.
                tol = un numero positivo que representa a la tolerancia para el
                criterio ||f(xk)|| < tol.
                iterMax = cantidad de iteraciones maximas.

    Parametros de Salida:
                xk = aproximacion de la convergencia de la funcion f.
                k = numero de iteraciones realizados.
                error = ||f(xk)||
    """

    from sympy import sympify
    from sympy import Symbol
    from sympy import diff
    from sympy import solve
    import math
    import matplotlib.pyplot as plt

    func = sympify(f)
    symVars = []
    g = []

    for i in vars:
        symVars.append(str(Symbol(i)))

    for i in symVars:
        g.append(func.diff(Symbol(i)))

    k = 0
    error = tol + 1
    e = []
    it = []

    while error > tol and k < iterMax:
        for i in range(len(symVars)):
            tempxk = {}
```

```
            for j in range(len(symVars)):
                if j == i:
                    tempxk.update({vars[j]: symVars[j]})
                else:
                    tempxk.update({vars[j]: xk[j]})
            tempFunc = func.subs(tempxk)
            dFunc = diff(tempFunc)
            sol = solve(dFunc, symVars[i])
            if len(sol) > 1:
                mini = 0
                for j in range(len(sol)):
                    currentSol = sol[j].evalf()
                    if currentSol > 0 and mini == 0:
                        mini = currentSol
                    elif currentSol < mini and mini != 0:
                        mini = currentSol
                    else:
                        continue
                xk[i] = mini
            else:
                xk[i] = sol[0].evalf()

        evas = {}
        for i in range(len(vars)):
            evas.update({vars[i]: xk[i]})

        gk = []
        for i in g:
            gk.append(i.subs(evas))

        add = 0
        for i in gk:
            add += math.pow(i, 2)

        error = math.sqrt(add)

        e.append(error)
        k = k + 1
        it.append(k)

    plt.plot(it, e)
    plt.show()
    return [xk, k, error]

help(coordinado)

coordinado('x**3+y**3+z**3-2*x*y-2*x*z-2*y*z',[1,1,1],['x','y','z'],10**-8,10)
```

2.2. Método 2: Gradiente Conjugado no Lineal

Código 8: gradiente

```
pkg load symbolic;
syms x y;

function [xk, k, error] = gradiente(func, x0, y0, tol, iterMax)

  %Esta funcion aproxima la solucion de la ecuacion f(x) = 0, utilizando el metodo del gradiente
      conjugado no lineal.
  %
  %Sintaxis: gradiente(f, x0, y0, tol, iterMax).
  %
  %Parametros iniciales:
  %
  %            f = representa la funcion f.
  %            x0 = valor inicial de x.
  %            y0 = valor inicial de y.
  %            tol = un numero positivo que representa la tolerancia para el criterio (error < tol).
  %            iterMax = cantidad de iteraciones maximas.
  %
  %Parametros de salida:
  %            xk = solucion aporximada de la funcion.
  %            k = numero de iteraciones realizados.
  %            error = |f(x)|
  syms x y;

  g = gradient(func);
  xk = [x0; y0];
  gk = double (subs(g,[x y],xk));#gradiente evaluado en valores iniciales
  gk_plus1 = 0;
  dk = -gk;#valor inicial de d
  bk = 0;
  a = 1; #valor inicial de alpha
  sigma = 0.5; #constante para calcular alpha
  listError = [];

  for k = 0:iterMax
    for j = 1: iterMax #se calcula alpha
      izq = double(subs(func,[x y],(xk+a*dk)') - subs(func,[x y],xk'));
      der = double (sigma*a*(subs(g,[x y],xk'))'*dk);
      if (izq<der)
        break;
      else
        a = a/5;
      endif
    endfor

    gk = double (subs(g,[x y],xk));
    xk = xk + a*dk; #actualiza el xk
    gk_plus1 = double (subs(g,[x y],xk));
    error = norm(gk_plus1);
    listError = [listError;error];
```

```
    if (error<tol)
      break;
    else
      bk = (norm(gk_plus1)^2) / (norm(gk)^2);
      dk = -gk_plus1 + bk*dk;
    endif
  endfor
  plot(1:k+1,listError)

endfunction
help gradiente
[xk, k, error] = gradiente(((x-2)^4+(x-2*y)^2), 2, 4, 10**-5, 10)
```

3. Tema 3: Sistemas de Ecuaciones

3.1. Métodos auxiliares

3.1.1. Determinante de matriz

Código 9: determinante de matriz nxn

```
from copy import deepcopy
import numpy

def determinanteNxN(A):

    """
    Esta funcion calcula el determinante de una matriz dada por el usuario

    Sintaxis: determinanteNxN(A)

    Parametros Iniciales:
                A = es una matriz de dimension nxn

    Parametros de Salida:
                determinanteMatriz = determinante de la matriz
    """

    n = len(A)
    if n == 1:
        return A[0][0]

    elif n == 2:
        return (A[0][0] * A[1][1]) - (A[0][1] * A[1][0])

    else:
        determinanteMatriz = 0

        for i in range(n):
            B = deepcopy(A)
            B = numpy.delete(B, 0, 0)
            B = numpy.delete(B, i, 1)

            if (i + 1) % 2 == 1:
                determinanteMatriz += A[0][i] * determinanteNxN(B)

            else:
                determinanteMatriz -= A[0][i] * determinanteNxN(B)

        #print (determinanteMatriz)
        return determinanteMatriz

help(determinanteNxN)
```

3.1.2. Sustitución hacia atrás

Código 10: sustitucion hacia atras

```
from copy import deepcopy

def sust_atras(A, b):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio de susticion hacia atras.

    Sintaxis: sust_atras(A, b)

    Parametros Iniciales:
                A = es una matriz de dimension nxn triangular inferior.
                b = es un vector con los valores de las igualdades de las ecuaciones.

    Parametros de Salida:
                x = es un vector con las soluciones del sistema de ecuaciones.
    """

    n = len(b)
    i = n - 1
    j = n - 1

    x = deepcopy(b)

    x[i] = b[i] / A[i][j]

    while i > -1:
        tempAdd = 0

        while j > i:
            tempAdd += A[i][j] * x[j]
            j -= 1

        if b[i] - tempAdd > 10**-3 or b[i] - tempAdd < -10**-3:
            x[i] = (b[i] - tempAdd) / A[i][j]
            j = n - 1
            i -= 1

        else:
            x[i] = 0
            j = n - 1
            i -= 1

    #print (x)
    return x

help(sust_atras)
```

3.1.3. Sustitución hacia adelante

Código 11: sustitucion hacia adelante

```
from copy import deepcopy

def sust_adelante(A, b):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio de susticion hacia adelante.

    Sintaxis: sust_adelante(A, b)

    Parametros Iniciales:
                A = es una matriz de dimension nxn triangular superior.
                b = es un vector con los valores de las igualdades de las ecuaciones.

    Parametros de Salida:
                x = es un vector con las soluciones del sistema de ecuaciones.
    """

    n = len(b)
    i = 0
    j = 0

    y = deepcopy(b)

    y[i] = b[i] / A[i][j]

    while i < n:
        tempAdd = 0

        while j < i:
            tempAdd += A[i][j] * y[j]
            j += 1

        if b[i] - tempAdd > 10**-3 or b[i] - tempAdd < -10**-3:
            y[i] = (b[i] - tempAdd) / A[i][j]
            j = 0
            i += 1

        else:
            y[i] = 0
            j = 0
            i += 1

    #print (y)
    return y

help(sust_adelante)
```

3.2. Método 1: Eliminación Gaussiana

Código 12: eliminacion gaussiana

```
from sust_atras import *
from sympy import Symbol
from sympy.solvers import solve

def gaussiana(A, b):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio de eliminacion gaussiana.

    Sintaxis: gaussiana(A, b)

    Parametros Iniciales:
                A = es una matriz de dimension nxn.
                b = es un vector con los valores de las igualdades de las ecuaciones.

    Parametros de Salida:
                x = es un vector con las soluciones del sistema de ecuaciones.
    """

    X = Symbol('X')

    l = len(b)
    i = 0
    j = 0
    n = 0

    while i < l:

        if i == 0:
            div = A[0][0]

            while j < l:
                A[i][j] = A[i][j] / div
                j += 1

            b[i] = b[i] / div
            i += 1
            j = 0

        else:

            if j == i:
                div = A[i][j]

                while j < l:
                    A[i][j] = A[i][j] / div
                    j += 1

                b[i] = b[i] / div
                i += 1
```

```
                j = 0

            elif A[i][j] != 0:
                factor = solve(A[i][j] + A[j][j] * X, X)
                factor = factor[0]
                n = j

                while n < l:
                    A[i][n] = A[i][n] + A[j][n] * factor
                    n += 1

                b[i] = b[i] + b[j] * factor
                j += 1

            else:
                j += 1

    print (sust_atras(A, b))
    return sust_atras(A, b)

help(gaussiana)
```

3.3. Método 2: Factorización LU

Código 13: Metodo de la factorizaciÃşn LU.

```
#include "determinant.h"

/**
 * @brief Funcion auxiliar
 * de sustitucion hacia atras
 *
 * @param A matrix
 * @param b vector
 * @return mat resultante
 */
mat sust_atras(mat A, mat b)
{
  int m = A.n_rows;
  mat x;
  x.zeros(m, 1);
  for (int i = m - 1; i >= 0; i--)
  {
    double aux = 0;
    for (int j = i + 1; j <= m - 1; j++)
    {
      aux += A(i, j) * x(j);
    }
    x(i) = (1 / A(i, i)) * (b(i) - aux);
  }
  return x;
}

/**
 * @brief Funcion auxiliar
 * para que implementa sustitucion hacia adelante
 *
 * @param A matriz
 * @param b vector
 * @return mat resultante
 */
mat sust_adelante(mat A, mat b)
{
  int m = A.n_rows;
  mat y = zeros<mat>(m, 1);

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

```
}

/**
 * @brief Implementacion computacional
 * del metodo Factorizacion LU
 *
 * @param A matriz a analizar
 * @param vect vector de resultados
 */
void fact_lu(mat A, mat vect)
{
  // Verificamos que sea invertible
  if ((determinantOfMatrix(A, A.n_cols) != 0) && A.is_square())
  {
    int size = A.n_rows;
    // Creamos y populamos la diagonal con 1's de la matriz L
    mat L;
    L.eye(A.n_rows, A.n_rows);
    // Creamos la matriz L
    mat U = A;

    // Eliminacion gaussiana
    for (int j = 0; j <= (size - 2); j++)
    {
      for (int i = j + 1; i <= (size - 1); i++)
      {
        // Operaciones entre filas
        L(i, j) = U(i, j) / U(j, j);
        U(i, span(j, size - 1)) -= L(i, j) * U(j, span(j, size - 1));
      }
    }
    mat y = zeros<mat>(size, 1);
    mat x = zeros<mat>(size, 1);
    y = sust_adelante(L, vect);
    x = sust_atras(U, y);
    cout << "L = \n"
         << L << "\nU = \n"
         << U << "\nVector y = \n"
         << y << "\nVector x = \n"
         << x << endl;
  }
  else
  {
    cout << "La matriz no es invertible o no es cuadrada." << endl;
  }
}


int main()
{
  mat A = {{1, 4, 0, 0},
           {3, 4, 1, 0},
           {0, 2, 3, 1},
           {0, 0, 1, 3}};
```

```
  mat vect = {1, 1, 1, 1};
  fact_lu(A, vect);
}
```

Código 14: Metodo auxiliar de la factorizacion LU para obtener el determinante de una matriz.

```
// C++ program to find Determinant of a matrix
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;


/**
 * @brief Obtiene el  cofactor
 *
 * @param A
 * @param tmp
 * @param p
 * @param q
 * @param n
 * @return mat
 */
mat getCofactor(mat A, mat tmp, int p,
                int q, int n)
{
    int i = 0, j = 0;
     for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            // Se copia en la matriz temporal solo aquellos
            // elementos que no estan en cierta columna fila
            if (row != p && col != q)
            {
                tmp(i,j++) = A(row,col);
                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
    return tmp;
}

/**
 * @brief Funcion general para obtener el determinante
 * de una matriz
 *
 * @param A
 * @param n
 * @return int
 */
int determinantOfMatrix(mat A, int n)
{
    int D = 0; // Resultado
```

```
    // Si la matriz solamente contiene un elemento
    if (n == 1)
        return A(0,0);

    // Matriz para guardar los cofactores
    mat tmp(A.n_cols, A.n_cols);
    int sign = 1;

    for (int f = 0; f < n; f++)
    {
        // Obtiene el cofactor de mat[0][f]
        tmp = getCofactor(A, tmp, 0, f, n);
        D += sign * A(0,f) * determinantOfMatrix(tmp, n - 1);
        // Se alterna el signo
        sign = -sign;
    }

    return D;
}
```

3.4. Método 3: Factorización de Cholesky

Código 15: factorizacion de Cholesky

```
from sust_adelante import *
from sust_atras import *
from determinanteNxN import *

from copy import deepcopy
import numpy
import math

def fact_cholesky(A, b):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio de factorizacion de Cholesky.

    Sintaxis: fact_cholesky(A, b)

    Parametros Iniciales:
                A = es una matriz de dimension nxn.
                b = es un vector con los valores de las igualdades de las ecuaciones.

    Parametros de Salida:
                x = es un vector con las soluciones del sistema de ecuaciones.
    """

    A = numpy.array(A)
    AResp = deepcopy(A)
    AT = numpy.transpose(A)
    check = False
    n = len(A)
    i = 0
    j = 0
    k = 0

    while i < n:
        while j < n:

            if A[i][j] != AT[i][j]:
                check = True
            else:
                j += 1
        i += 1
        j = 0

    i = 0
    j = 0

    while i < n - 1:
        while j < n - 1:
            A = numpy.delete(A, i + 1, 0)
            A = numpy.delete(A, i + 1, 1)
            j += 1
```

```
        mdet = determinanteNxN(A)

        if mdet <= 0:
            check = True

        else:
            i += 1
            A = deepcopy(AResp)
            k += 1
            j = k

    mdet = determinanteNxN(A)

    if mdet <= 0:
        check = True

    if check:
        print('La Matriz no es simÃľtrica definida positiva')

    else:
        i = 0
        j = 0
        k = 0
        l = len(b)
        L = numpy.eye(l, l)

        while j < l:
            i = j
            k = j
            tempAdd = A[i][j]
            k -= 1

            while k > -1:
                tempAdd -= math.pow(L[i][k], 2)
                k -= 1

            L[i][j] = math.sqrt(tempAdd)
            i += 1
            k = j - 1

            while i < l:
                tempAdd = A[i][j]
                while k > -1:
                    tempAdd -= (L[i][k] * L[j][k])
                    k -= 1

                L[i][j] = tempAdd / L[j][j]
                i += 1
                k = j - 1

            j += 1

        y = sust_adelante(L, b)
```

```
        print (sust_atras(numpy.transpose(L), y))
        return sust_atras(numpy.transpose(L), y)

help(fact_cholesky)
```

3.5. Método 4: Método de Thomas.

Código 16: Metodo de Thomas.

```
#include <iostream>
#include <armadillo>
using namespace arma;
using namespace std;

/**
 * @brief Retorna el vector de la diagonal adyacente superior
 *
 * @param A matriz
 * @return mat vector resultante
 */
mat sub_Diagonal(mat A)
{
    int size = A.n_rows;
    mat result;
    result.zeros(1, size - 1); // Define ceros la matriz de resultado
    int cont = 0;
    for (int i = 1; i < size; i++)
    { // Popula la matriz
        result(i - 1) = A(i, cont);
        cont++;
    }
    cout << "La diagonal inferior es: " << result << endl;
    return result;
}

/**
 * @brief Retorna la diagonal de la matriz
 *
 * @param A matriz por analizar
 * @return mat vector resultante
 */
mat diagonal(mat A)
{
    int size = A.n_rows;
    mat result;
    result.zeros(1, size);
    for (int i = 0; i < size; i++)
    { // popula el vector resultante
        result(i) = A(i, i);
    }
    cout << "La diagonal es: " << result << endl;
    return result;
}

/**
 * @brief Retorna la diagonal superior
 *
 * @param A matriz por analizar
 * @return mat vector resultante
 */
```

```
mat superior_Diagonal(const mat A)
{
    int size = A.n_rows;
    mat result;
    result.zeros(1, size); // Popula con ceros el vector resultantes
    int cont = 1;
    for (int i = 0; i < size - 1; i++)
    { //Popula el vector resultante
        result(i) = A(i, cont);
        cont++;
    }
    cout << "La diagonal superior es: " << result << endl;
    return result;
}

/**
 * @brief Determina si la matriz es tridiagonal
 *
 * @param matrix matriz por analizar
 * @param vect vector de resultados
 * @return true si la matriz es tridiagonal
 * @return false si la matriz no es tridiagonal
 */
bool isTriDiagonal(mat matrix, mat vect)
{
    if (matrix.is_square())
    // Verifica que sea cuadrada
    {
        int size = matrix.n_rows;
        for (int x = 0; x < size; x++)
        {
            for (int y = 0; y < size; y++)
            {
                int cell = matrix(x, y);

                if ((x == y) || (x - 1 == y) || (x + 1 == y))
                // Verifica que las posiciones de las diagonales sean diferentes de cero
                {
                    if (cell == 0)
                    {
                        cout << "La matriz no es tridiagonal" << endl;
                        return false;
                    }
                }
                else
                {
                    if (cell != 0)
                    // Verifica que las posiciones que no pertenecen a las diagonales sean
                    {
                        cout << "La matriz no es tridiagonal" << endl;
                        return false;
                    }
                }
            }
```

```
        }
        if (matrix.n_cols == vect.size())
        {
            cout << "La matriz es tridiagonal" << endl;
            return true;
        }
    }
}

/**
 * @brief Funcion auxiliar para la sustitucion hacia delante
 *
 * @param vect
 * @param A
 * @param vect_len
 * @return mat
 */
mat sustitucion_adelante(mat vect, mat A, int vect_len)
{
    for (int i = 1; i < vect_len; i++)
    {
        vect(i) = vect(i) - A(i - 1) * vect(i - 1);
    }
    return vect;
}

/**
 * @brief Funcion auxiliar para la sustitucion hacia atras
 *
 * @param vect
 * @param X
 * @param C
 * @param B
 * @param vect_len
 * @return mat
 */
mat sustitucion_atras(mat vect, mat X, mat C, mat B, int vect_len)
{
    X(vect_len - 1) = vect(vect_len - 1) / B(vect_len - 1);
    for (int i = vect_len - 2; i >= 0; --i)
    {
        X(i) = (vect(i) - C(i) * X(i + 1)) / B(i);
    }
    return X;
}

/**
 * @brief Implementacion computacional del metodo de
 * Thomas para soluciones de sistemas de ecuaciones
 *
 * @param matrix
 * @param vect
 */
void thomas(mat matrix, mat vect)
```

```
{
    // determinamos que sea tridiagonal
    isTriDiagonal(matrix, vect);
    int vect_len = vect.size();

    // Determinamos las tres diagonales de la matriz
    mat A = sub_Diagonal(matrix);
    mat B = diagonal(matrix);
    mat C = superior_Diagonal(matrix);

    mat X; // creamos la matriz resultante
    X.zeros(1, vect_len);

    // Realizamos la descomposicion de las matrices
    for (int i = 1; i < vect_len; i++)
    {
        A(i - 1) = A(i - 1) / B(i - 1);
        B(i) = B(i) - A(i - 1) * C(i - 1);
    }

    // Por ultimo resolvemos por sustitucion hacia delante y hacia atras
    vect = sustitucion_adelante(vect, A, vect_len);
    X = sustitucion_atras(vect, X, C, B, vect_len);

    cout << "El resultado por el metodo de Thomas es: " << X << endl;
}

int main()
{
  mat A = {{1, 4, 0, 0},
           {3, 4, 1, 0},
           {0, 2, 3, 1},
           {0, 0, 1, 3}};
  mat vect = {1, 1, 1, 1};
  thomas(A, vect);
}
```

3.6. Método 5: Jacobi

Código 17: jacobi

```
import matplotlib.pyplot as plt
import numpy
import math

def jacobi(A, b, xk, tol, iterMax):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio del metodo de Jacobi.

    Sintaxis: jacobi(A, b, xk, tol, iterMax)

    Parametros Iniciales:
                A = es una matriz de dimension nxn.
                b = es un vector con los valores de las igualdades de las ecuaciones.
                xk = es un vector con los valores iniciales del sistema de ecuaciones.
                tol = tolerancia  para el  criterio  ||A * x - b||<tol.
                iterMax = cantidad de iteraciones maximas.

    Parametros de Salida:
                xk = es un vector con las soluciones del sistema de ecuaciones.
                k = numero de iteraciones realizados.
                error = ||A * xk - b||.
    """

    i = 0
    j = 0
    n = len(A)
    check = False

    while i < n:
        pivote = abs(A[i][i])
        temp_sum = 0

        while j < n:
            if i == j:
                j += 1

            else:
                temp_sum += abs(A[i][j])
                j += 1

        if temp_sum > pivote:
            check = True
            break

        i += 1
        j = 0

    if check:
        print("La matriz no es diagonalmente dominante.")
```

```
    else:

        i = 0
        j = 0

        k = 0
        error = tol + 1
        e = []
        it = []
        xk1 = []

        for i in xk:
            xk1.append(0)

        while error > tol and k < iterMax:

            while i < n:
                temp_sum = 0

                while j < n:
                    if i == j:
                        j += 1

                    else:
                        temp_sum += A[i][j] * xk[j]
                        j += 1

                xk1[i] = (1 / A[i][i]) * (b[i] - temp_sum)
                i += 1
                j = 0

            for i in range(len(xk1)):
                xk[i] = xk1[i]

            errMat = numpy.dot(A, xk) - b;
            errAdd = 0

            for i in errMat:
                errAdd += math.pow(i, 2)

            error = math.sqrt(errAdd)
            e.append(error)
            k = k + 1
            it.append(k)
            i = 0
            j = 0

        plt.plot(it, e)
        print ([xk, k, error])
        plt.show()
        return [xk, k, error]

help(jacobi)
```

3.7. Método 6: Gauss-Seidel

Código 18: gauss-seidel

```
import numpy
import math
import matplotlib.pyplot as plt

def gauss_seidel(A, b, x, tol, iterMax):

    """
    Esta funcion determina las soluciones de un sistema de ecuaciones representado
    matricialmente por medio del metodo de Gauss-Seidel.

    Sintaxis: gauss_seidel(A, b, x, tol, iterMax)

    Parametros Iniciales:
                A = es una matriz de dimension nxn
                b = es un vector con los valores de las igualdades de las ecuaciones.
                x = es un vector con los valores iniciales del sistema de ecuaciones.
                tol = tolerancia para el criterio ||A * x - b||<tol.
                iterMax = cantidad de iteraciones maximas.

    Parametros de Salida:
                x = es un vector con las soluciones del sistema de ecuaciones
                k = numero de iteraciones realizados
                error = ||A * x - b||
    """

    i = 0
    j = 0
    n = len(A)
    check = False

    while i < n:
        pivote = abs(A[i][i])
        temp_sum = 0

        while j < n:
            if i == j:
                j += 1

            else:
                temp_sum += abs(A[i][j])
                j += 1

        if temp_sum > pivote:
            check = True
            break

        i += 1
        j = 0

    if check:
        print("La matriz no es diagonalmente dominante.")
```

```
    else:

        i = 0
        j = 0

        k = 0
        error = tol + 1
        e = []
        it = []

        while error > tol and k < iterMax:

            while i < n:
                temp_sum = 0

                while j < n:
                    if i == j:
                        j += 1

                    else:
                        temp_sum += A[i][j] * x[j]
                        j += 1

                x[i] = (1 / A[i][i]) * (b[i] - temp_sum)
                i += 1
                j = 0

            errMat = numpy.dot(A, x) - b;
            errAdd = 0

            for i in errMat:
                errAdd += math.pow(i, 2)

            error = math.sqrt(errAdd)
            e.append(error)
            k = k + 1
            it.append(k)
            i = 0
            j = 0

        plt.plot(it, e)
        print ([x, k, error])
        plt.show()
        return [x, k, error]

help(gauss_seidel)
```

3.8. Método 7: Método de la Pseudoinversa

Código 19: pseudoinversa

```
#include <iostream>
#include <iomanip>
#include <armadillo>
#include "matplotlibcpp.h"

using namespace std;
using namespace arma;
namespace plt = matplotlibcpp;

/**
 * @brief Metodo de la pseudoinversa
 *
 * @param A: matriz de coeficientes
 * @param B: vector de terminos independientes
 * @param tol: tolerancia
 * @param iterMax: iteraciones maximas
 * @return error: error de la aproximacion
 *         xk: pseudoinversa aproximada
 */

void pseudoinversa(mat A, vec B, double tol, double iterMax){

    // Matrices
    mat x0 = (1 / (pow(norm(A,2),2))) * A.t();
    mat xk, xk1;

    // Matriz identidad
    int m = size(A)[0];
    mat I = eye(m,m);

    //Vector xk y xk1
    vec vecxk, vecxk1;

    //Error
    double error;
    vector <double> errors;

    vector <double>  iterations;

    for(int k = 1; k < iterMax; k++){

        // Pseudoinversa aproximada
        xk = x0 * (2 * I - A * x0);
        vecxk = xk * B;

        xk1 = xk * (2 * I - A * xk);
        vecxk1 = xk1 * B;

        // Formula para calcular el error
        error = (norm(vecxk1 - vecxk,2)) / (norm(vecxk1,2));
```

```
        // Condicion de parada
        if(abs(error) < tol){
            break;
        }
        x0 = xk;
        errors.push_back(error);
        iterations.push_back(k);
    }

    cout.precision(10);
    cout.setf(ios::fixed, ios::floatfield);
    cout << setw(15);
    xk.raw_print(cout, "Pseudoinversa aproximada: \n");
    cout << endl;
    vecxk1.raw_print(cout, "Vector aproximado: \n");
    cout << endl;
    cout << "Error: " << scientific << error << endl << endl;
    cout << endl;

    // Grafica de iteraciones vs error
    plt::plot(iteraciones,errores,"r-");
    plt::title("Iteraciones vs Error");
    plt::xlabel("Iteraciones");
    plt::ylabel("Error");
    plt::grid(true);
    plt::show();
}



int main(){
    mat A = {{1,2,-1}, {-3,1,5}};
    vec B = {1,4};

    pseudoinversa(A,B,1e-8,500);

    return 0;
}
```

4. Tema 4: Polinomio de interpolación.

4.1. Método 1: Método de Lagrange

Código 20: Lagrange

```
from sympy import Symbol
from sympy import expand
from sympy.plotting import plot

def lagrange(xk, yk):

    """
    Esta funciÃşn encuentra el polinomio de interpolaciÃşn de Lagrange

    Sintaxis: lagrange(xk, yk)

    ParÃąmetros Iniciales:
                xk = es una lista con pares x de las coordenadas
                yk = es una lista con pares y de las coordenadas

    ParÃąmetros de Salida:
                px = polinomio de interpolaciÃşn
    """

    X = Symbol('X')

    n = len(xk)
    L = []
    Ltemp = 1

    j = 0
    k = 0

    while k < n:
        while j < n:
            if j == k:
                j += 1
            else:
                Ltemp *= (X - xk[j]) / (xk[k] - xk[j])
                j += 1

        L.append(expand(Ltemp))
        Ltemp = 1
        k += 1
        j = 0

    px = 0

    for i in range(n):
        px += yk[i] * L[i]

    plot(px)
```

```
    print (px)
    return px

help(lagrange)

xk = [-2, 0, 1]
yk = [0, 1, -1]
lagrange(xk, yk)
```

4.2. Método 2: Método de Diferencias Divididas de Newton

Código 21: Diferencias Divididas de Newton

```
from sympy import Symbol, expand

def dd_newton(xk, yk):

    """
    Esta funciÃşn encuentra el px de interpolaciÃşn de Diferencias Divididas de Newton

    Sintaxis: dd_newton(xk, yk)

    ParÃąmetros Iniciales:
                xk = es una lista con pares x de las coordenadas
                yk = es una lista con pares y de las coordenadas

    ParÃąmetros de Salida:
                px = polinomio de interpolaciÃşn
    """

    par = []
    lista_puntos = []
    largo_lista = len(xk)
    z = 0

    while z < largo_lista:
        par.append(xk[z])
        par.append(yk[z])
        lista_puntos.append(par)
        par = []
        z += 1
    print (lista_puntos)

    n = len(lista_puntos)  # Cantidad de puntos
    px = lista_puntos[0][1]  # Polinomio de interpolacion
    resul_previos = []     # Lista para los resultados previos
    x = Symbol('x')        # Variable simbolica
    m = n - 1
    multi = 1  # Variable que almacena la multiplicacion (x-x0) * ... * (x-xn)

    for i in range(0, n): # Se agregan los "y" en la lista de resultados previos
        resul_previos += [lista_puntos[i][1]]

    for i in range(1, n):
        multi *= x - lista_puntos[i - 1][0]  # Se multiplica por (x - xi)
        resul_nuevos = []  # Lista para los resultados nuevos

        for j in range(0, m): # Se realiza el calculo de cada elemento del polinomio
            numerador = resul_previos[j] - resul_previos[j + 1]
            denominador = lista_puntos[j][0] - lista_puntos[j + i][0]
            resul_nuevos += [numerador / denominador]
        m -= 1

        # Se actualiza el polinomio
```

```
        px += resul_nuevos[0] * multi

        # Se actualiza la lista de resultados previos
        resul_previos = resul_nuevos.copy()

    return expand(px)

help(dd_newton)

xk = [-2, 0, 1]
yk = [0, 1, -1]
print(dd_newton(xk, yk))
```

4.3. Método 3: Trazador Cúbico Natural

Código 22: Trazador Cubico Natural

```
import numpy as np
import  sympy as sp

,,,
Esta funcion permite obtener los polinomios de interpolacion grado 3
de una funcion.

Sintaxis : traz_cubico(x_k, y_k)

Parametros Iniciales :
    x_k: corresponde al vector de las preimagenes en un intervalo
    y_k: vector de imagenes de x_k evaluada en una funcion
Parametros de Salida :
    trazadores: vector de polinomios de interpolacion
,,,
def traz_cubico(x_k, y_k):
    # Primero se construye el vector de h_k
    h_k = vectorH_k(x_k)

    # Segundo se construye el vector de u_k
    u_k = vectorU_k(y_k, h_k)

    #Tercero se contruye la tridiagonal
    tridiagonal = construirTridiagonal(h_k)

    #Cuarto se calcula la solucion M del sistema A*M=U por metodo de Thomas
    M = thomas(tridiagonal,u_k.transpose())

    #Quinto se calculan los coeficientes a,b,c,d y se calculan los trazadores S_i
    trazadores = vectorTrazadores(M, h_k, x_k, y_k)



def vectorH_k(x_k):
    h_k = []
    for i in range(len(x_k)-1):
        h_k.append(x_k[i+1]-x_k[i])
    return h_k

def vectorU_k(y_k, h_k):
    u_k = np.zeros((1,len(y_k)-2))
    for i in range(len(y_k)-2):
        tmp = (y_k[i+2]-y_k[i+1])/h_k[i+1]
        tmp2 = (y_k[i+1]-y_k[i])/h_k[i]
        u_k[0][i] = (6*(tmp - tmp2))
    return u_k

def construirTridiagonal(h_k):
    m = len(h_k)-1
    tridiagonal = np.zeros((m,m))
    for i in range(m):
```

```
        tmp = np.zeros(m)
        if (i==0):
            tmp[i+1] = h_k[i+1]
        elif (i==(m-1)):
            tmp[i-1] = h_k[i]
        else :
            tmp[i-1]=h_k[i]
            tmp[i+1] = h_k[i+1]
        tmp[i] = 2*(h_k[i]+h_k[i+1])
        tridiagonal[i] = tmp
    return tridiagonal


def is_tridiagonal(a):
    dims=a.shape

    for i in range(dims[0]):
        for j in range(dims[1]):
            if(j>i+1 and a[i,j]!=0):
                return False
            elif(j<i-1 and a[i,j]!=0):
                return False
    return True

def thomas(a,b):
    if(is_tridiagonal(a)):
        n=len(b)
        a_s=[]
        b_s=[]
        c_s=[]
        d_s=b.transpose();
        p_s=[]
        q_s=[]
        xk =np.zeros((1,n))
        for i in range(n):
            if(i==0):
                a_s.append(0)
                b_s.append(a[i,i])
                c_s.append(a[i,i+1])
            elif(i==n-1):
                a_s.append(a[i,i-1])
                b_s.append(a[i,i])
                c_s.append(0)
            else:
                a_s.append(a[i, i - 1])
                b_s.append(a[i, i])
                c_s.append(a[i, i + 1])
        n=len(b)
        qi=0
        pi=0
        for i in range(n):
            if(i==0):
                p_s.append(c_s[i]/b_s[i])
                q_s.append(d_s[0,i]/b_s[i])
```

```
            else:
                if(i!=n-1):
                    p_s.append(c_s[i]/(b_s[i]-p_s[i-1]*a_s[i]))
                    q_s.append((d_s[0,i]-q_s[i-1]*a_s[i])/(b_s[i]-p_s[i-1]*a_s[i]))
                else:
                    q_s.append((d_s[0, i] - q_s[i - 1] * a_s[i]) / (b_s[i] - p_s[i - 1] * a
        for j in range(n-1,-1,-1):
            if(j==n-1):
                xk[0,j]=q_s[j]
            else:
                xk[0,j]=q_s[j]-p_s[j]*xk[0,j+1]

        m_k = []
        m_k.append(0)
        for i in range(n):
            m_k.append(xk[0,i])
        m_k.append(0)

        return m_k
    else:
        print("La matriz no es tridiagonal")
        return None

def vectorTrazadores(M_k, h_k, x_k, y_k):
    trazadores = []
    x = sp.symbols('x')

    for i in range(len(h_k)):
        a= (M_k[i+1]-M_k[i])/(6*h_k[i])
        b= M_k[i]/2
        c= ((y_k[i+1]-y_k[i])/h_k[i])-((h_k[i]*(M_k[i+1]+2*M_k[i]))/6)
        d= y_k[i]
        polinomio= a*(x-x_k[i])**3+b*(x-x_k[i])**2+c*(x-x_k[i])+d
        trazadores.append(polinomio)

    return trazadores


# --------------- Ejemplo de la presentacion --------------
x_k = [1,1.05,1.07,1.1]
y_k = [2.718282, 3.286299, 3.527609, 3.905416]
traz_cubico(x_k, y_k)
```

4.4. Método 4: Cota de Error Polinomio de Interpolación

Código 23: Cota de Error Polinomio de Interpolación

```
function test
     pkg load symbolic
     clc;

     f = 'cos((x)/2)';

     ptos = [-5 0 5];

     valor = 0.4;

     cota = cota_poly_inter(f, ptos)
end



% Calcula el error de una aproximacion con un polinomio de interpolacion
% Param f: cadena de caracteres
% Param puntos: vector de los puntos de soporte
% Retorna: Cota del error

function cota = cota_poly_inter(f, puntos)

     n = length(puntos) - 1;

     fs = sym(f);

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

     alpha = alpha(fs, n + 1, a, b);

     xMax = polyMax(puntos, a, b);

     result = 1;
     for k = puntos
          result *= abs(xMax - k);
     end
     cota = alpha / factorial(n + 1) * result;

endfunction


% Calcula el alpha_max
% Retorna: funcion evaluada en punto maximo
function alpha = alpha(fs, m, a, b)

     fn = matlabFunction(fs);

     dfm_s = diff(fs, m);

     dfm_n = matlabFunction(dfm_s);
```

```
    fs_aux = -1 * abs(dfm_s);

    fn_aux = matlabFunction(fs_aux);

    xMax = fminbnd(fn_aux, a, b);

    alpha = abs(dfm_n(xMax));

endfunction


% Calcula el puntos maximo
% Retorna: Preimagen del punto maximo
function xMax = polyMax(puntos, a, b)

    syms x;
    f = 1;
    for i = puntos
        f *= (x - i);
    end

    f = expand(f);

    fs = sym(f);

    fn = matlabFunction(fs);

    f1 = diff(f, x);

    critical_pts = flip(solve(f1)');

    X = [fn(a), fn(b), critical_pts];

    Y = [];

    for k = 1 : length(X)

        punto = double(X(k));

        if (a < punto) & (punto < b)

            Y = [Y, abs(fn(punto))];

        else
            X(k) = [];

        endif
    end

    [yMax index] = max(Y);

    xMax = double(X(index));
```

```
endfunction
```

4.5. Método 5: Cota de Error Trazador Cúbico Natural

Código 24: Cota de Error Trazador Cubico Natural

```
import numpy as np
import  sympy as sp

'''
Esta funcion permite estimar el error del trazador cubico.

Sintaxis : cota_traz_cubico(funcion, x_k)

Parametros Iniciales :
    funcion: funcion de la cual se obtienen los trazadores
    x_k: vector de soporte definido en un intervalo [a,b]
Parametros de Salida :
    cota: error estimado
'''
def cota_traz_cubico(funcion, x_k):
    n = len(x_k)
    x = sp.symbols('x')
    fun = sp.simplify(funcion)

    #Calculamos la cuarta derivada
    derivada = cuartaDerivada(fun, x)

    #Obtenemos las imagenes
    imagenes = vectorImagenes(n, x_k, derivada, x)

    #Calculamos el vector h_k
    h_k = vectorH_k(x_k)

    #Calculamos el error
    h = max(h_k)
    tmp = max(imagenes)
    cota = tmp*(5*h**4)/384
    return cota

def cuartaDerivada(funcion, x):
    derivada = funcion.diff(x)
    derivada2 = derivada.diff(x)
    derivada3 = derivada2.diff(x)
    derivada4 = derivada3.diff(x)
    return derivada4


def vectorImagenes(n, x_k, derivada, x):
    imagenes = []
    for i in range(n):
        x_i = x_k[i]
        y_i = float(derivada.subs(x,x_i))
        imagenes.append(abs(y_i))
    return imagenes

def vectorH_k(x_k):
```

```
    h_k = []
    n = len(x_k)
    for i in range(n-1):
        h_i = x_k[i+1] - x_k[i]
        h_k.append(h_i)
    return h_k

x_k = [1, 1.05, 1.07, 1.1]
cota = cota_traz_cubico('3*x*exp(x) - 2*exp(x)', x_k)
print(cota)
```

5. Tema 5: Integración Numérica.

5.1. Método 1: Regla del Trapecio y Cota de Error.

Código 25: Regla del Trapecio y Cota de Error

```
pkg load symbolic;


function result = trapecio(funcion, intervalo)
  % Esta funcion aproxima la integral de una funcion en intervalo
  % mediante el metodo del trapecio.
  %
  % Sintaxis: trapecio(funcion, intervalo).
  %
  % Parametros iniciales:
  %  funcion = representa la funcion f.
  %  intervalo = vector que contiene los puntos a y b en que se evalua la funcion.
  %
  % Parametros de salida:
  %  area = aproximacion de la integral.
  %  error = cota de error del metodo.

  % Preparamos la funcion y los parametros iniciales
  ff = funcion;
  func = function_handle(sym(ff));
  syms x;
  a = intervalo(1);
  b = intervalo(2);

  % Calculamos el area
  h = b-a;
  area = (func(a)+func(b))*(h/2);

  % Calculamos el error
  fd=diff(func,x);
  fdd = diff(fd,x);
  fdd_aux = function_handle(-1*abs(fdd));
  max = fminbnd(fdd_aux,a,b);
  error = abs((h**3)*fdd_aux(max)/12);

  % Acoplamos el resultado
  result = [area, error];
endfunction

trapecio('ln(x)', [2,5])
```

5.2. Método 2: Regla de Simpson y Cota de Error

Código 26: Regla de Simpson y Cota de Error

```
from sympy import sympify
from sympy import diff

def simpson(f, a, b):

    """
    Esta funciÃşn encuentra una aproximaciÃşn de la integral de una funciÃşn
    en un intervalo dado con la regla de simpson.

    Sintaxis: simpson(f, a, b)

    ParÃąmetros Iniciales:
                f = funciÃşn a integrar.
                a = extremo izquierdo del intervalo de integraciÃşn.
                b = extremo derecho del intervalo de integraciÃşn.

    ParÃąmetros de Salida:
                aprox = valor aproximado de la integral en el intervalo especÃŋficado.
                error = h^5/90 * |f^4(eps)|.
    """
    func = sympify(f)
    primeraDerivada = diff(func)
    segundaDerivada = diff(primeraDerivada)
    terceraDerivada = diff(segundaDerivada)
    cuartaDerivada = diff(terceraDerivada)

    h = (b - a) / 2
    x1 = (a + b) / 2

    aprox = ((h / 3) * (func.subs({'x': a}) + 4 * func.subs({'x': x1}) + func.subs({'x': b}

    epsilon = a

    for i in range(a, b):
        if abs(cuartaDerivada.subs({'x': i})) > abs(cuartaDerivada.subs({'x': epsilon})):
            epsilon = i

    error = (((h**5) / 90) * abs(cuartaDerivada.subs({'x': epsilon}))).evalf()
    print ([aprox, error])
    return [aprox, error]

help(simpson)

simpson('ln(x)', 2, 5)
```

5.3. Método 3: Regla Compuesta del Trapecio y Cota de Error.

Código 27: Regla Compuesta del Trapecio y Cota de Error

```
pkg load symbolic;


function result = trapecio_compuesto(funcion, puntos, intervalo)
  %Esta funcion aproxima la integral de una funcion en un intervalo
  % mediante el metodo del trapecio compuesto.
  %
  % Sintaxis: trapecio_compuesto(funcion, puntos, intervalo).
  %
  % Parametros iniciales:
  %  funcion = representa la funcion f.
  %  puntos = numero entero en que se divide el intervalo.
  %  intervalo = vector que contiene los puntos a y b en que se evalua la funcion.
  %
  % Parametros de salida:
  %  sum = aproximacion de la integral.
  %  error = cota de error del metodo.

  % Preparamos la funcion y los parametros iniciales
  ff = funcion;
  func = function_handle(sym(ff));
  syms x;

  a = intervalo(1);
  b = intervalo(2);
  h = (b-a)/(puntos-1);

  % Calculamos el area
  sum = 0;
  for i=0:puntos-2
    x_k = a + i*h;
    x_k1 = a + (i+1)*h;
    sum = sum + (func(x_k)+func(x_k1));
  endfor
  sum = h*sum/2;

  % Calculamos el error
  fd=diff(func,x)
  fdd = diff(fd,x);
  fdd_aux = function_handle(-1*abs(fdd));
  max = fminbnd(fdd_aux,a,b);
  error = abs((h**2)*(b-a)*fdd_aux(max)/12);

  % Acoplamos el resultado
  result = [sum, error];
endfunction

trapecio_compuesto('ln(x)', 4, [2,5])
```

5.4. Método 4: Regla Compuesta de Simpson y Cota de Error

Código 28: Regla Compuesta de Simpson y Cota de Error

```
from sympy import sympify
from sympy import diff

def simpson_compuesto(f, a, b, m):

    """
    Esta funciÃşn encuentra una aproximaciÃşn de la integral de una funciÃşn en un interval

    Sintaxis: simpson_compuesto(f, a, b, m)

    ParÃąmetros Iniciales:
                f = funciÃşn a integrar
                a = inicio del intervalo de integraciÃşn
                b = final del intervalo de integraciÃşn
                m = cantidad de subintervalos a realizar

    ParÃąmetros de Salida:
                aprox = valor aproximado de la integral en el intervalo especÃŋficado
                error = ((b - a) * h^4)/180 * |f^4(eps)|
    """

    func = sympify(f)
    primeraDerivada = diff(func)
    segundaDerivada = diff(primeraDerivada)
    terceraDerivada = diff(segundaDerivada)
    cuartaDerivada = diff(terceraDerivada)

    h = (b - a) / (m - 1)

    i = 1
    x = [];
    x.append(a)
    while i < m:
        x.append(a + (i * h));
        i += 1;

    i = 1
    sumaPar = 0
    sumaImpar = 0
    cantVars = len(x) - 1

    while i < cantVars:
        if i % 2 == 0:
            sumaPar += (func.subs({'x': x[i]})).evalf()
            i += 1
        else:
            sumaImpar += (func.subs({'x': x[i]})).evalf()
            i += 1

    aprox = ((h / 3) * (func.subs({'x': a}) + 2 * sumaPar + 4 * sumaImpar + func.subs({'x':
```

```
    epsilon = a;

    for i in range(a, b):
        if abs(cuartaDerivada.subs({'x': i})) > abs(cuartaDerivada.subs({'x': epsilon})):
            epsilon = i

    error = ((((b - a) * (h**4)) / 180) * abs(cuartaDerivada.subs({'x': epsilon}))).evalf()
    print ([aprox, error])
    return [aprox, error]

help(simpson_compuesto)

simpson_compuesto('ln(x)', 2, 5, 7)
```

5.5. Método 5: Cuadratura Gaussiana y Cota de Error

Código 29: Cuadratura Gaussiana y Cota de Errorr

```
#{
Params
    f  =  Funcion f en forma de string
    n: numero de orden
    a : limite menor del intervalo
    b : limite mayor del intervalo

Parametros de Salida:
    aprox :  aproximacion
    cota : cota del error de la aproximacion
#}


function [I cota]  =  cuad_gaussiana(f,n,a,b)

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

  cota  =  0;
  I  =  0;
  fs  =  sym(f);
  y  =  ((b - a) * x + (b + a)) / 2;
  gs  =  (b - a) / 2 * subs(fs, x, y);
  gn  =  matlabFunction(gs);

  [x, w]  =  ceros_cuad_gaussiana(n);

 # Cuadratura gaussiana
  for i  =  1 : n
    I + =  w(i) * gn(x(i));
  endfor

  if n  =  =  2
    # Simbolica
    f2_s  =  abs(diff(gs, 4));
    # Numerica
    f2_n  =  matlabFunction(f2_s);


    # Simbolica
    fs_aux  =  (-1) * abs(f2_s);
    # Numerica
    fn_aux  =  matlabFunction(fs_aux);

    # Alpha max
    x_max  =  fminbnd(fn_aux, -1, 1);
    alpha  =  f2_n(x_max);

    #Calculo de la cota.
    cota  =  alpha / 135  # cota.
```

```
  end
end


function [x, m]  =  ceros_cuad_gaussiana(n)
  if n > 10 | n < 2
    x  =  0;
    m  =  0;

  else
    x  =  []; m  =  [];
    if n  =  =  2
      x(1)  =  -0.5773502692;
      x(2) = -x(1);

      m(1)  =  1;
      m(2)  =  1;

    elseif n == 3
      x(1) = -0.7745966692;
      x(2) = 0;
      x(3) = -x(1);

      m(1) = 0.5555555555;
      m(2) = 0.888888888;
      m(3) = m(1);

    elseif n == 4
      x(1) = -0.86113631159405;
      x(2) = -0.339981043584856;
      x(3) = -x(2);
      x(4) = -x(1);

      m(1) = 0.347854845137454;
      m(2) = 0.652145154862546;
      m(3) = m(2);
      m(4) = m(1);

    elseif n == 5
      x(1) = -0.9061798459;
      x(2) = -0.5384693101;
      x(3) = 0;
      x(4) = -x(2);
      x(5) = -x(1);

      m(1) = 0.2369268851;
      m(2) = 0.4786286705;
      m(3) = 0.5688888889;
      m(4) = m(2);
      m(5) = m(1);

    elseif n == 6
      x(1) = -0.9324695142;
```

```
    x(2) = -0.6612093865;
    x(3) = -0.2386191861;
    x(4) = -x(3);
    x(5) = -x(2);
    x(6) = -x(1);

    m(1) = 0.1713244924;
    m(2) = 0.3607615730;
    m(3) = 0.4679139346;
    m(4) = m(3);
    m(5) = m(2);
    m(6) = m(1);

  elseif n == 7
    x(1) = -0.9491079123;
    x(2) = -0.7415311856;
    x(3) = -0.4058451514;
    x(4) = 0;
    x(5) = -x(3);
    x(6) = -x(2);
    x(7) = -x(1);

    m(1) = 0.1294849662;
    m(2) = 0.2797053915;
    m(3) = 0.3818300505;
    m(4) = 0.4179591837;
    m(5) = m(3);
    m(6) = m(2);
    m(7) = m(1);

  elseif n == 8
    x(1) = -0.9602898565;
    x(2) = -0.7966664774;
    x(3) = -0.5255324099;
    x(4) = -0.1834346425;
    x(5) = -x(4);
    x(6) = -x(3);
    x(7) = -x(2);
    x(8) = -x(1);

    m(1) = 0.1012285363;
    m(2) = 0.2223810345;
    m(3) = 0.3137066459;
    m(4) = 0.3626837834;
    m(5) = m(4);
    m(6) = m(3);
    m(7) = m(2);
    m(8) = m(1);

  elseif n == 9
    x(1) = -0.9681602395;
    x(2) = -0.8360311073;
    x(3) = -0.6133714327;
    x(4) = -0.3242534234;
```

```
      x(5) = 0;
      x(6) = -x(4);
      x(7) = -x(3);
      x(8) = -x(2);
      x(9) = -x(1);

      m(1) = 0.0812743883;
      m(2) = 0.1806481607;
      m(3) = 0.2606106964;
      m(4) = 0.3123470770;
      m(5) = 0.3302393550;
      m(6) = m(4);
      m(7) = m(3);
      m(8) = m(2);
      m(9) = m(1);

    elseif n == 10
      x(1) = -0.9739065285;
      x(2) = -0.8650633667;
      x(3) = -0.6794095683;
      x(4) = -0.4333953941;
      x(5) = -0.1488743390;
      x(6) = -x(5);
      x(7) = -x(4);
      x(8) = -x(3);
      x(9) = -x(2);
      x(10) = -x(1);

      m(1) = 0.0666713443;
      m(2) = 0.1494513492;
      m(3) = 0.2190863625;
      m(4) = 0.2692667193;
      m(5) = 0.2955242247;
      m(6) = m(5);
      m(7) = m(4);
      m(8) = m(3);
      m(9) = m(2);
      m(10) = m(1);

    end
  end
end
```

6. Tema 6: Diferenciación Numérica.

6.1. Método 1: Método de Euler.

Código 30: Metodo de Euler

```
from sympy import sympify
from sympy import Symbol
from sympy import expand
import matplotlib.pyplot as plt

def euler(funcion, intervalo, pasoh, y_0):
    """
    Esta funcion permite resolver el problema de Cauchy

    Sintaxis: euler(funcion, intervalo, pasoh, y_0)

    Parametros Iniciales:
                funcion: string de la funcion a evaluar
                intervalo: de la forma [a,b] en la cual se evalua la funcion
                pasoh: numero de puntos dentro del intervalo
                y_0: valor inicial de los pares y_k

    Parametros de Salida:
                Una lista con los siguientes valores
                x_k: lista de los pares x_k
                y_k: lista de los pares y_k
                px: polinomios de interpolacion
    """

    # Se preparan los valores iniciales
    func = sympify(funcion)
    a = intervalo[0]
    b = intervalo[1]
    N = pasoh
    h = (b-a)/(N-1)

    #Arrays para almacenar resultados
    valores_xk = []
    valores_yk = []

    #Se ejecuta el metodo
    n = 0
    while (n < N):
        x_k = a + n*h
        valores_xk.append(x_k)
        valores_yk.append(y_0)

        f_x_y = func.subs({'x':x_k, 'y':y_0})
        y_n = y_0 + h*f_x_y
        y_0 = y_n
        n+=1

    #Se obtiene el polinomio de interpolacion
```

```
    pol = lagrange(valores_xk, valores_yk)

    #Se construye la grafica
    plt.scatter(valores_xk, valores_yk)
    plt.plot(valores_xk, valores_yk)

    plt.title("Polinomio de interpolaciÃ§n")
    plt.xlabel("xk")
    plt.ylabel("p(x)")

    plt.show()
    return [valores_xk, valores_yk, pol]


def lagrange(xk, yk):
    #Funcion para obtener polinomio de interpolacion
    k = 0
    polinomio = 0
    while k<len(xk):
        polinomio +=  yk[k]*L_k(k, xk)
        k+=1
    polinomio = expand(polinomio)

    return polinomio

def L_k(k, xk):
    #Funcion auxiliar para lagrange
    j = 0
    x = Symbol('x')
    Lk = 1
    while j < len(xk):
        if j != k:
            Lk = Lk*(x - xk[j])/(xk[k] - xk[j])

        j+=1
    Lk = expand(Lk)
    return Lk

result = euler('1+y-x**2', [0,2],11,0.5)
help(euler)
print("Los parametros de salida son:\n \nValores x_k de los pares ordenados:\n \n", result[
      "\n\nValores y_k de los pares ordenados:\n \n", result[1],
      "\n\nPolinomios de interpolacion:\n \n", result[2])
```

6.2. Método 2: Método Predictor-Corrector.

Código 31: Metodo Predictor-Corrector

```
pkg load symbolic

function [x, y] = predictor_corrector(f, a, b, y0, n)

  %Esta funciÃşn encuentra una aproximaciÃşn a la soluciÃşn del problema
  %de Cauchy con el mÃľtodo predictor-corrector.
  %
  %Sintaxis: predictor_corrector(f, a, b, y0, n)
  %
  %ParÃąmetros Iniciales:
  %            f = funciÃşn de dos variables x y y
  %            a = inicio del intervalo
  %            b = final del intervalo
  %            y0 = valor inicial de y
  %            n = cantidad de subintervalos a realizar
  %
  %ParÃąmetros de Salida:
  %            x = vector con las soluciones del par ordenado x
  %            y = vector con las soluciones del par ordenado y

  func = matlabFunction(sym(f));

  h = (b - a) / (n - 1);
  i = 1;
  x(i) = a;

  while i < n
    x(i + 1) = a + (i * h);
    i += 1;
  end

  y(1) = y0;
  i = 2;

  while i < n + 1
    yPrima(i) = y(i - 1) + (h * func(x(i - 1), y(i - 1)));
    temp = (func(x(i - 1), y(i - 1)) + func(x(i), yPrima(i))) / 2;
    y(i) = y(i - 1) + (h * temp);
    i += 1;
  end

  stem(x, y)

end

help predictor_corrector

[x, y] = predictor_corrector('y - x**2 + 1', 0, 2, 0.5, 11)
```

6.3. Método 3: Runge-Kutta de Orden 4.

Código 32: Runge-Kutta de Orden 4

```
from sympy import *
import  matplotlib.pyplot  as plt



def runge_kutta_4(y0, a, b, n, funcion):

    """
    Esta funcion aproxima la solucion de la ecuacion diferencial dx/dy,
    utilizando el metodo de Runge-Kutta de Orden 4.

    Sintaxis:
        runge_kutta_4(y0, a, b, n, funcion)

    Parametros Iniciales:
        y0 = va a ser el valor inicial
        a, b = son los extremos del intervalo [a, b]
        n = va a ser un nÃžmero que representa la cantidad de puntos
        funcion = es una cadena de caracteres (string) que representa a la ecuacion diferen

    Parametros  de  Salida:
        [x, y], vectores conteniendo los pares ordenados de la solucion
    """

    variableX = Symbol('x')
    variableY = Symbol('y')
    func = sympify(funcion)
    f = lambdify([variableX, variableY], func, "numpy")

    h = (b - a) / (n - 1)

    x = [a]
    y = [y0]

    k_4 = []
    k_3 = []
    k_2 = []
    k_1 = []

    i = 0

    while i < n-1:

        k_1.append(f(x[i], y[i]))
        k_2.append(f(x[i] + h / 2, y[i] + h * k_1[i] / 2))
        k_3.append(f(x[i] + h / 2, y[i] + h * k_2[i] / 2))
        k_4.append(f(x[i] + h, y[i] + h * k_3[i]))

        x.append(x[i] + h)
        y.append(y[i] + h / 6 * (k_1[i] + 2 * k_2[i] + 2 * k_3[i] + k_4[i]))
```

```
        i += 1

    plt.plot(x, y)
    plt.show()

    return [x, y]

help (runge_kutta_4)

print(runge_kutta_4(1, 0, 1, 11, '-x*y + 4*x/y'))
```

6.4. Método 4: Adam-Bashford de Orden 4.

Código 33: Adam-Bashford de Orden 44

```
% Funcion con los pasos para graficar y probar el metodo
function adam_bashford_4 ()
  clc; clear;

  f = '1 - (y - x + 2) ^ 2 - x + 4';
  n = 6;
  startValue = [1 1 0.5 2];
  interval = [2 4];
  [xv, yv, interPol] = adam_bashford_4_method(f, interval, n, startValue)
  hold on

  % Grafica de la solucion con el metodo
  % (rojo)
  xG = interval(1) : 0.0001 : interval(2);
  pol1 = matlabFunction(sym(interPol));
  yP = pol1(xG);
  plot(xG, yP, 'r')

  % Grafica analitica
  % (azul)
  yS = @(x) -1 * (x - 1).^(-1) + x
  xG = interval(1):0.0001:interval(2);
  yG = yS(xG);
  plot(xG, yG, 'b')
  title('Grafica de soluciones analitica (azul) y del polinomio de interpolacion (rojo)')
  xlabel('x')
  ylabel('y(x)')

endfunction


% Entradas:
%   f: funcion f.
%   interval: intervalo que contiene el valor inicial y final
%   num: corresponde al numero de puntos con el que se
%     realizara el analisis.
%   val_inicial: corresponde al conjunto de valores iniciales
%     que necesita este metodo para ejecutarse

% Salidas:
%   xv, yv: son lo vectores que componen los pares ordenados de los
%     valores aproximados de la solucion del problema.
%   interPol: es el polinomio de interpolacion calculado por medio del
%     metodo de lagrange para los puntos indicados.


function [xv, yv, interPol] = adam_bashford_4_method(f, interval, num, startValue)

  pkg load symbolic;
  syms x y;
```

```
  f1 = matlabFunction(sym(f));

  a = interval(1);
  b = interval(2);
  h = (b - a) / (num - 1);

  xv = a : h : b;

  y0 = startValue(1);
  y1 = startValue(2);
  y2 = startValue(3);
  y3 = startValue(4);

  yv = [y0 y1 y2 y3];

  for n = 4 : num - 1

    fk = f1(xv(n), yv(n));
    fk1 = f1(xv(n - 1), yv(n - 1));
    fk2 = f1(xv(n - 2), yv(n - 2));
    fk3 = f1(xv(n - 3), yv(n - 3));
    yv(n + 1)= yv(n)+ (h / 24) * (55 * fk - 59 * fk1 + 37 * fk2 - 9 * fk3);

  endfor


  interPol = metodo_lagrange(xv, yv);

endfunction


function Lk = lk(xv, k)

  syms x
  n = length(xv) - 1;
  Lk = 1;
  for j = 0 : n

    if j ~= k
      Lk = Lk * (x - xv(j + 1)) / (xv(k + 1) - xv(j + 1));

    endif

  endfor

  Lk = expand(Lk);

endfunction


% Metodo de Lagrange

% Parametros de entrada:
%   xv, yv: vetores de los pares ordenador
```

```
function p = metodo_lagrange(xv, yv)

  syms x
  n = length(xv) - 1;
  p = 0;

  for k = 0 : n
    p = p + yv(k + 1) * lk(xv, k);
  endfor

  p = expand(p);

endfunction
```

7. Tema 7: Valores y Vectores Propios.

7.1. Método 1: Método de la Potencia.

Código 34: Metodo de la Potencia

```
function ejecucion()
     clc

     x_0 = [1; 1; 1];
     A =[-3 1 0;1 -2 1;0 1 -3];

     [valor, vector]=potencia(A, x_0);

     fprintf('valor propio apropximado: ');
     valor
     fprintf('vector propio correspondiente: ');
     vector
endfunction

function [valor, vector] = potencia(matriz, vector_0)
  %Esta funcion aproxima el calculo de el
  %valor propio de mayor magnitud una matrix de entrada.
  %
  %Sintaxis: potencia(matriz, vector_0).
  %
  %Parametros iniciales:
  %
  %            matriz = matriz de entrada simetrica definida positiva.
  %            vector_0 = vector de valor inicial.
  %
  %Parametros de salida:
  %            valor = aproximaciÃşn de valor propio de mayor magnitud de la matriz.
  %            vector = vector propio correspondiente.

  % Definimos valores iniciales
  x_0 = vector_0;
  iterMax = 100;
  tol = 10**-10;
  error = 1;
  error_vect = [];

  %Iniciamos iteraciones del metodo
  for (n=1:iterMax)
    %Calculos carecteristicos
    y_k = matriz * x_0;
    c_k = norm(y_k,Inf);
    x_k = (1/c_k)*y_k;

    %Calculo del error
    err = norm(x_k-x_0);
```

```
    %Actualizacion de variables
    x_0 = x_k;
    error_vect = [ error_vect,  err ];

    %Condicion de parada
    if (error < tol)
      break
    endif

  endfor

  %Valores finales
  valor = c_k;
  vector = x_0';

  %Finalmente, graficamos
  plot(0:length(error_vect)-1,error_vect)
  title('Metodo de Potencia')
  xlabel('Iteracion')
  ylabel('Error')

endfunction
```

7.2. Método 2: Método de la Potencia Inversa.

Código 35: Metodo de la Potencia Inversa

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

using namespace std;
using namespace arma;

mat potencia_inversa(mat A, mat vector, int maxIter) {
    mat iterations;
    mat errors;
    double tol = 1e-5;
    mat yk1;
    float ck1;
    mat xk1;
    mat xk = vector;
    double error = 1.0 + tol;

    for (int k = 1; k < maxIter; k++){
        yk1 = solve(A, xk);
        ck1 = norm(yk1, "inf");
        xk1 = (1 / ck1) * yk1;
        error = norm(xk1 - xk);
        xk = xk1;

        if (error < tol) {
```

```
            cout << "Final error: "<< error << endl;
            cout << "Final k: "<< k << endl;
            cout << "Final ck: "<< ck1 << endl;
            cout << "Final xk: "<< xk << endl;
            break;
        }
        cout << "Error: "<< error << endl;
        cout << "k: "<< k << endl;
        cout << "ck: "<< ck1 << endl;
        cout << "xk: "<< xk << endl;
    }
    return A;
}

int main() {
    mat A = {{5, 4, -2},
            {4, -1, -7},
            {8, 2, 5}};

    mat b = {1, 1, 1};
    b = b.t();
    potencia_inversa(A, b, 15);

    return 0;
}
```

7.3. Método 3: Método QR.

Código 36: Metodo QR

```
import numpy as np


def producto_punto(x, y):
    result = 0
    for i in range(len(x)):
        result += x[i] * y[i]
    return result


def factorizacion_QR(A):
    A = np.array(A)
    Q = []
    for i in range(len(A)):
        Ak = A[:, i]
        if i == 0:
            U = Ak
        else:
            sum = 0
            for j in range(i):
                sum += producto_punto(Ak, Q[j]) * np.array(Q[j])
            U = A[:, i] - sum
        P = U / np.linalg.norm(U)
```

```
            Q.append(P.tolist())
        R = np.dot(Q, A)
        Q = np.transpose(Q)
        return Q, R


"""
Metodo QR para el calculo de autovalores de una matriz
Entradas: Matriz y tolerancia minima
Salidas: Valores propios de la matriz
"""
def metodo_QR(A, tol):
    error = tol
    while error >= tol:
        QR = factorizacion_QR(A)
        A = np.dot(QR[1], QR[0])
        error = 0
        for i in range(1, len(A)):
            for j in range(i):
                error += abs(A[i][j])
    return np.diagonal(A)


print(metodo_QR([[5, 0, 2], [1, 2, 3], [8, 1, 4]], 0.000005))
```