

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 | 2 |
| 1.1. Método 1: Bisección | 2 |
| 1.2. Método 2: Newton-Raphson | 3 |
| 1.3. Método 3: Secante | 5 |
| 1.4. Método 4: Falsa Posición | 6 |
| 1.5. Método 5: Punto Fijo | 7 |
| 1.6. Método 6: Muller | 8 |
| 2. Optimización | 9 |
| 2.1. Método 1: Descenso Coordinado | 9 |
| 2.2. Método 2: Gradiente Conjugado No Lineal | 11 |
| 3. Sistemas de Ecuaciones | 14 |
| 3.1. Método 1: Eliminación Gaussiana | 14 |
| 3.2. Método 2: Factorización LU | 16 |
| 3.3. Método 3: Factorización Cholesky | 17 |
| 3.4. Método 4: Método de Thomas | 19 |
| 3.5. Método 5: Método de Jacobi | 21 |
| 3.6. Método 6: Método de Gauss-Seidel | 23 |
| 3.7. Método 7: Método de Relajacion | 25 |
| 3.8. Método 8: Método de la Pseudoinversa | 27 |
| 4. Polinomio de Interpolación | 27 |

| | |
|-------------------------------|----|
| 5. Integración Numérica | 27 |
| 6. Diferenciación Numérica | 27 |
| 7. Valores y Vectores Propios | 27 |

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: tolerancia 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);
        end
    end
end
```

```

        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.

```

# Metodo de Newton-Raphson
# Entradas:
    #func: es la funcion a analizar
    #x0: valor inicial
    #MAXIT: es la cantidad de iteraciones maximas a realizar

```

```

#TOL: es la tolerancia del algoritmo
# Salidas:
#xAprox: es la solucion, valor aproximado de x
#error: pocentaje de error del resultado obtenido

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

def newton_raphson(func, x0, MAXIT, TOL):
    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

#Grafica
#Entradas:
#listaValoresX: valores que se graficaran en el eje 'x'
#listaValoresY: valores que se graficaran en el eje 'y'
#Salidas:
#Grafico con los valores ingresados
def grafica(listaValoresX, listaValoresY):
    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
func = lambda x: (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;
    symlist 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

#Punto Fijo
#Entradas: funcion - Funcion por aproximar - funcion lambda
#valor - inicial - Valor por el cual se empezara a aproximar - int, float, double
#iteraciones - maximas - Numero maximo de itreaciones - int
#
#

def punto_fijo(funcion, valor_inicial, iteraciones_maximas):
    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
    plt.plot(lista_error, label = 'errores por interaccion') #Construccion de tabla
    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

funcion = lambda x: np.exp(-x)
punto_fijo(funcion, 0, 15)

```

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: tolerancia 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\nError = %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 componen 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.

```

# Metodo del Gradiente Conjugado No Lineal
# Entradas:
    #func: string con la funcion a evaluar
    #vars: lista con las variables de la ecuacion
    #xk: vector con los valores iniciales
    #MAXIT: es la cantidad de iteraciones maximas a realizar
# Salidas:
    #xAprox: es la solucion, valor aproximado de x
    #error: pocentaje de error del resultado obtenido

#####
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):
    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 tamaño"

    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
        betakdk = [i * bk for i in dk] #Se calcula el nuevo valor del vector 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

# Evaluar Gradiente
# Entradas:
    #gradiente: gradiente a evaluar
    #:vars: lista con las variables de la ecuacion
    #:xk: vector con los valores iniciales
# Salidas:
    #gradResult: resultado de evaluar el vector en el gradiente

```

```

def evaluarGradiente(gradiente, variables, xk):
    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

# Calcular alpha k
# Entradas:
    #gradiente: gradiente a evaluar
    #:vars: lista con las variables de la ecuacion
    #:xk: vector con los valores iniciales
# Salidas:
    #gradResult: resultado de evaluar el vector en el gradiente
def calcularAlphaK(func, variables, xk, dk, gk):
    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 multiplicacion 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

# Evaluar en la funcion
# Entradas:
    #func: string con la funcion a evaluar
    #:vars: lista con las variables de la ecuacion
    #:xk: vector con los valores iniciales
# Salidas:
    #func: resultado de evaluar en la funcion
def evaluarFuncion(func, variables, xk):
    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

# Calcular beta k
# Entradas:
    #gk: vector gk
    #prevGK: vector gk de la iteracion anterior
    #dk: vector dk
    #reglaBK: regla utilizada para calcular el BK
# Salidas:
    #b: valor del Bk calculado
def calcularBetaK(gk, prevGK):
    #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

#Grafica
#Entradas:
    #listaValoresX: valores que se graficaran en el eje 'x'
    #listaValoresY: valores que se graficaran en el eje 'y'
#Salidas:
    #Grafico con lo valores ingresados
def grafica(listaValoresX, listaValoresY):
    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.

```

# Metodo de la Factorizacion LU
# Entradas:
#   matrizD: matriz de coeficientes
#   matrizI: matriz de terminos independientes
# Salidas:
#   X: solucion del sistema

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

def fact_lu(matrizD, matrizI):
    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;

/**Funcion que realiza la factorizacion de Cholesky
Entradas: Una matriz A de cualquier tama o , simetrica y positiva definida
Salidas: 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;
}

/**Funcion que realiza la sustitucion hacia atras
Entradas:
1) Una matriz L que es la factorizacion de Cholesky de otra matriz
2) Un vector d que es el vector de terminos independientes
Salidas: 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;
}

/**Funcion que realiza la sustitucion hacia atras
Entradas:
1) Una matriz L que es la transpuesta de la factorizacion de Cholesky de otra matriz
2) Un vector y que es el vector de terminos independientes
Salidas: 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;
}

/**Funcion que realiza la sustitucion hacia atras
Entradas:
1) Una matriz A de cualquier tama o
2) Un vector d que es el vector de terminos independientes
Salidas: Un vector x que es la solucion del sistema de ecuaciones**/
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.

```

# Metodo de Thomas
# Entradas:
#   matrizC: matriz de coeficientes
#   matrizTI: matriz de terminos independientes
# Salidas:
#   X: solucion del sistema

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

def thomas(matrizC, vectorTI):
    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 tamaño")

# Funcion para crear la matriz tridiagonal
# Entradas:
# N: tamaño de la matriz
# a: valor debajo de la diagonal principal
# b: valor de la diagonal principal
# c: valor sobre la diagonal principal
# Salidas:
# matriz: matriz tridiagonal
def creaTridiagonal(N, a, b, c):
    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

# Funcion para crear el vector d
# Entradas:
# N: tamano de la matriz
# ext: valor en los extremos del vector
# inte: valor en el interior del vector
# Salidas:
# d: vector d
def creaD(N, ext, inte):
    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

#Metodo de relajacion
#Entradas
#A matriz de cofactores
#b matriz de respuestas
#maxI maxima cantidad de iteraciones
#tol tolerancia para calcular error
#w constante por usar en el metodo

def relajacion(A,b,maxI,tol,w):
    #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)

        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

plt.plot(lista_error, label = 'errores por interacion') #Construccion de tabla

```

```

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()
return x,error

#ejemplo numerico
relajacion([[4,3,0],[3,4,-1],[0,-1,4]],[7,7,7],5,0.0001,1.24)

```

3.8. Método 8: Método de la Pseudoinversa

4. Polinomio de Interpolación

5. Integración Numérica

6. Diferenciación Numérica

7. Valores y Vectores Propios