FINAL FOLLOWAGE

José Joab Romero

Juan Camilo Guerrero

Santiago Moreno

Kevin Alexander Herrera

24 November, 2020

Subject:

Numerical Analysis

Professor:

Edwar Samir Posada Murillo

Semester:

6th

Name project:

Alpha Numeric

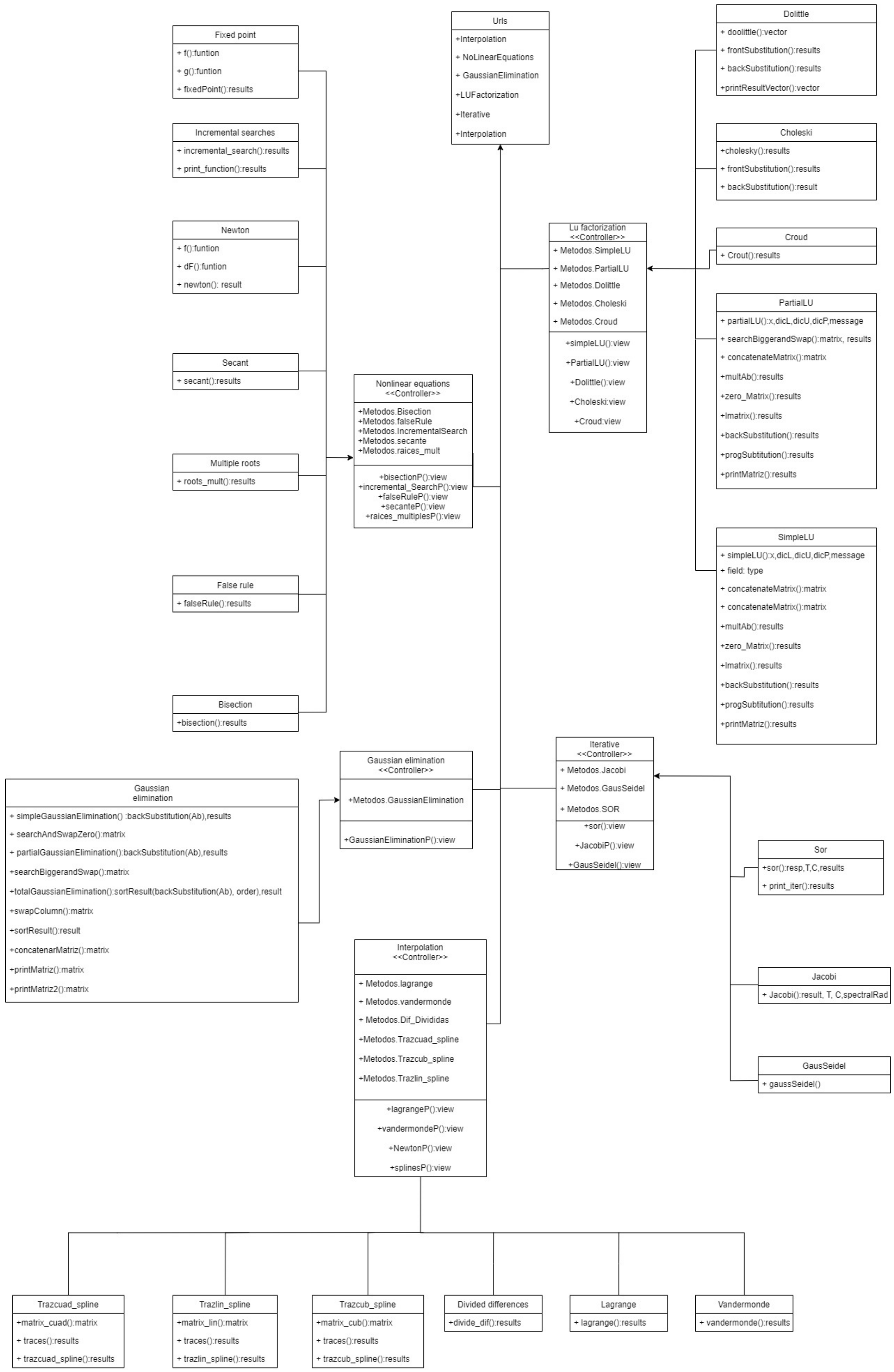
Repository from where we will work:

Page from where we will work:

https://dry-beyond-30251.herokuapp.com/

Contents

Class diagram



Use case diagram

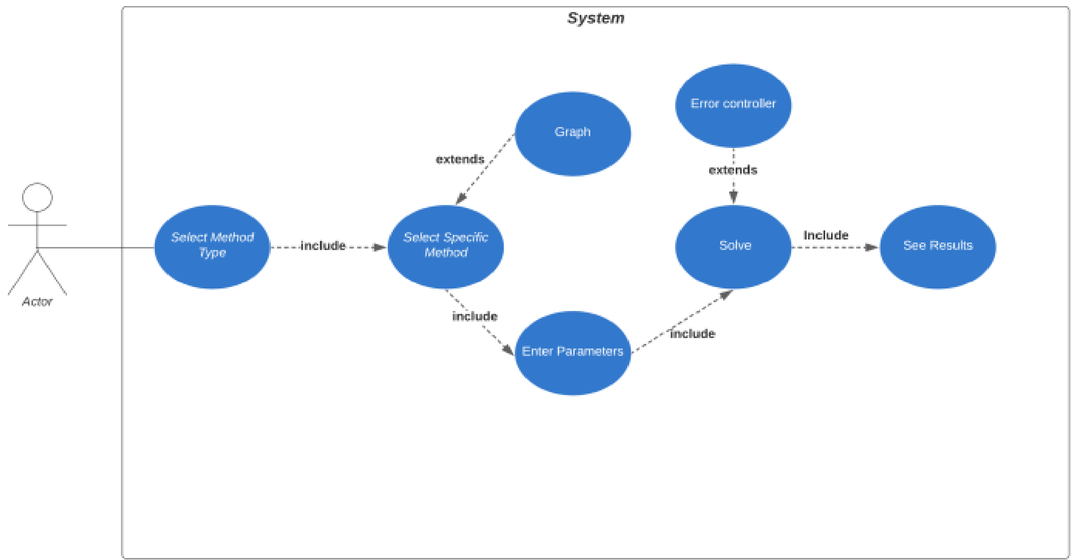
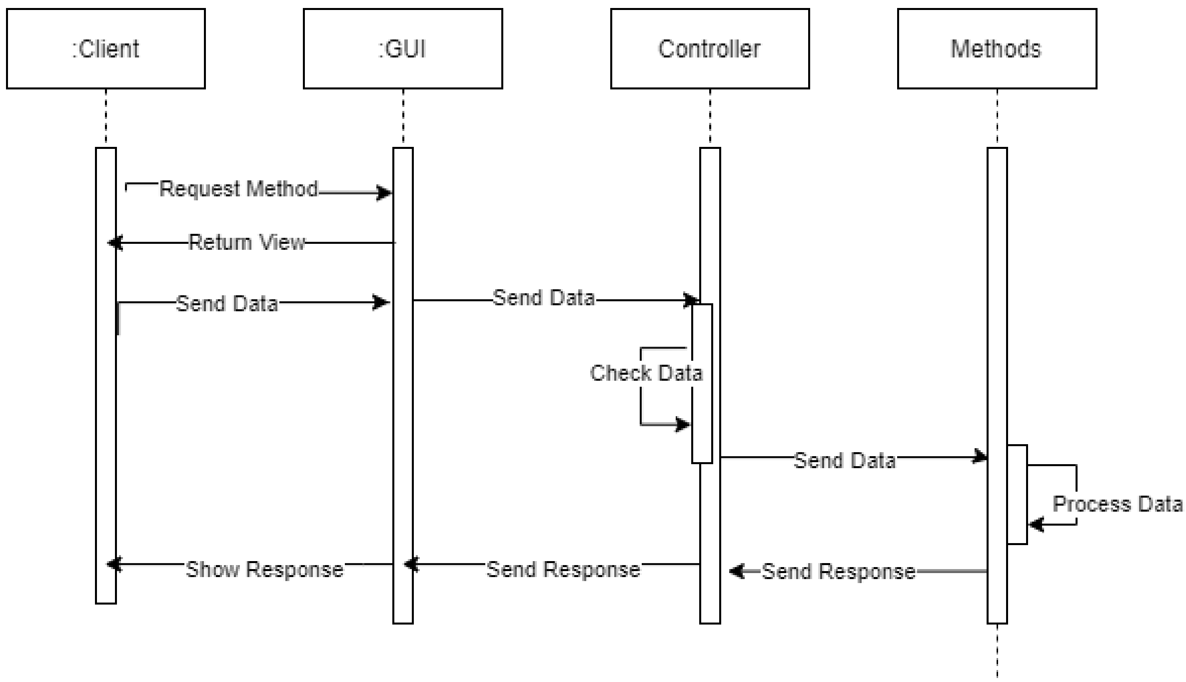


Diagram sequence



Pseudocodes

Bisection:

read xi, xs, tolerancia, niter, funcion

```
fxi = f(xi)
fxs = f(xs)

if (fxi == 0)
write "xi es raíz"
else if (fxs == 0)
write "xs es raíz"
else if (fxi * fxs < 0 then)

xm = (xi + xs)/2
fxm = f(xm)
error = tolerancia + 1
count = 1

while ( fxm != 0 and error > tolerancia and contador < niter)
if (fxi * fxm < 0 )
xs = xm
else
xi = xm
fxi = fxm
end if

xaux = xm
xm = (xi + xs) / 2
fxm = f(xm)
error = abs(xm - xaux)
count = count + 1
end while

else
write "el intervalo es inadecuado"
```

end if

end

Cholesky:

read A, b

determinante=det(A);

if determinante==0 then

return "El sistema no tiene una única solución"

end if

n = lenght A

L,U = Iniciar LU(n)

for k=1 hasta n

suma1=0;

for p=1:k-1

suma1=suma1+L(k,p)∗ U(p,k);

end for

L(k,k)∗ U(k,k)=A(k,k) - suma1;

```
for i=k+1 hasta n
suma2= 0;
for p = 1 hasta k-1
suma2= suma2+ L(i,p)* U(p,k);
end for
L(i,k) = (A(i,k)-suma2)/U(k,k);
end for

for j=k+1 hasta n
suma3= 0;
for p = 1hasta k-1
suma3=suma3+ L(k,p)* U(p,j);
end for
U(k,j) = (A(k,j)- suma3)/L(k,k);
end for
end for
return L,U
end
```

Crout:

```
read A, b
n = lenght A
L,U = Iniciar LU(n)
for k=0 hasta n
U(k,k)=1;
for i=k hasta n
suma1 = 0
for p=0 hasta k
suma1=suma1+L(k,p)* U(p,k);

end for
U(k,j)=A(k,j) - suma1;

for j=k+1 hasta n
suma2= 0;
for p = 0 hasta k
suma2= suma2+ L(k,p)* U(p,j);
end for
U(k,j) = (A(k,j)-suma2)/L(k,k);
end for
end for


detA = 1;
for i=1 hasta n
detA = detA* L(i,i)
end for


if (detA ≠ 0 then)
```

z = sustituir(L,b)

x = sustituir(U,z)

else

return "Hay infinitas soluciones o no tiene solución"

end if

return L,U

Divide diferencies:

leer x,y:

n = longitud(x)

D = matriz de 0

D[:,0] = y->traspuesta

para i hasta n:

aux0 = D[i-1:n,i-1]

```
aux1 = diferenciaAdyacente(aux0)
aux2 = restaVectorial(x[i:n],x[0:n-1-i+1])
D[i:n,i] = DivisionVectorial(aux1,traspuesta(aux2))
fin

res = diagonal(D)

r = res[0]
m = '(x' + (-x[0]) + ')'
para i hasta n:
r += res[i] + m
m += '(x' + -x[i] + ')'
fin
escribir('Matrix D: \ n',D)
escribir('Coef: ',res)
escribir('Newton Polinom : ', r)
fin
```

Doolittle:

```
read A, b
L,U = Iniciar LU(n)
for k=0 hasta n
L(k,k)=1;
for j=k hasta n
suma1 = 0
for p=0 hasta k
suma1=suma1+L(k,p)* U(p,k);

end for
U(k,j)=A(k,j) - suma1;
```

```
for i=k+1 hasta n
suma2= 0;
for p = 0 hasta k
suma2= suma2+ L(i,p)* U(p,k);
end for
L(i,k) = (A(i,k)-suma2)/U(k,k);
end for
end for

detA = 1;
for i=1 hasta n
detA = detA* U(i,i)
end for

if (detA ≠ 0 then)
z = sustituir(L,b)
x = sustituir(U,z)
else
return "Hay infinitas soluciones o no tiene solución"
end if

return L,U
```

Fixed point:

read xa, tol, iter

y = f(xa)

cont = 0

error = tol + 1

while(y != 0 & error > tol & cont < iter)

xn = g(xa)

y = f(xn)

error = abs(xn - xa)

xa = xn

cont += 1

end while

if (y == 0)

write "xa is root"

else if (error < tol)

write "xa approximate root with tolerance: tol)"

else

write "Fail in iteration: iter"

end if

Gaussian Elimination:

simpleGaussianElimination

read a, b

AB = concatenar(a, b)

n = len(AB)

while k < n do

write step k

write AB

if AB[k][k] == 0

AB = searchAndSwapZero(Ab, n, k)

while i = k+1 < n do

mult = Ab[i][k]/Ab[k][k]

while j = k < n+1 do

Ab[i][j] = Ab[i][j] - mult∗ Ab[k][j]

partialGaussianElimination

read a, b

AB = concatenar(a, b)

n = len(AB)

while k < n do

write step k

write AB

AB = searchBiggerandSwap(Ab, n, k)

while i = k+1 < n do

mult = Ab[i][k]/Ab[k][k]

while j = k < n+1 do

Ab[i][j] = Ab[i][j] - mult∗ Ab[k][j]

totalGaussianElimination

read a, b

AB = concatenar(a, b)

n = len(AB)

while k < n do

write step k

write AB

Ab, order = searchTheBiggestandSwap(Ab, n, k, order)

while i = k+1 < n do

mult = Ab[i][k]/Ab[k][k]

while j = k < n+1 do

Ab[i][j] = Ab[i][j] - mult$*$ Ab[k][j]

Gauss Seidel:

read A,b,x0,tol,iter

n = lenght A

cont = 0

error = tol + 1

while(error > tol & cont < iter)

for i=0 hasta n do

sum=0

for j=0 hasta i do

```
sum = sum + A(i,j)* x(j)
end for

for j=i+1 hasta n do
sum = sum + A(i,j)* x(j)
end for
x(i) = ((b(i)-sum)/A(i,i))
end for

error = errorAbsolute(x0,x)
cont++
for i=0 hasta n do
x0(i)= x(i)
end for
mostrar cont vector x error
end while

if(error < tol)
return "La solución del sistema es: "
mostrar vector x
else
return "Fracaso en " cont
mostrar vector x
end
```

IncrementalSearch:

read x_inicial, delta, limite_Iteraciones, funcion:

if delta <= 0:

write "El delta debe ser positivo"

sys.exit(1)

elif limite_Iteraciones > 0:

x_anterior = x_inicial

x_actual = x_anterior+delta

f_anterior = f(x_anterior)

f_actual = F(x_actual)

contador = 0

while (contador < limite_Iteraciones):

if f_actual∗ f_anterior<0:

resultados[contador] <- [x_anterior,x_actual]

x_anterior = x_actual

x_actual = x_actual + delta

f_anterior = f_actual

f_actual = f(x_actual)

contador = contador + 1

endwhile:

devolver resultados

write(aux)

endif:

else:

write "Las iteraciones deben ser un numero positivo"

sys.exit(1)

end else:

end

Jacobi:

Jacobi ():

read A

read b

read t

read iter

read x0

n = length of A

l = length of A [0]

if (n! = l):

```
write ("A is not a square matrix please check and run again.")
if not:
x = [with the size of nxn]
aux = 0
cont = 0
error = t + 1
iteration = 1
T = [with the size of nxn]
C = [with the size of n]
while (error> t and cont <o = iter):
write ("iteration: # " + str (iteration))
error = 0
for i from 0 to n:
sum = 0
for j from 0 to n:
if (i! = j):
sum = sum + A [i] [j] * x0 [j]
T [i] [j] = -A [i] [j] / A [i] [i]
C [i] = b [i] / A [i] [i]

x [i] = (b [i] - sum) / A [i] [i]
aux = x [i] - x0 [i]
error = error + math.pow (aux, 2)

error = error raised to 0.5

write (error)
```

```
for i from 0 to n:
    x0 [i] = x [i]
write ("x" + i + 1 + ":" + x0 [i])
```

LagrangeP:

```
read x, y

n = len(x)
l = []
ld = []

for i in x do
    l1 = ''
    l2 = 1
    for j in x do
        if i!=j then
            if j<0 then
                l1+= '(x + '+str(abs(j))+') '
            else:
                l1+= '(x - '+str(abs(j))+') '
            l2 = l2* (i-j)
    l.append(l1)
```

```
ld.append(l2)

i = 0
lf = []
l1 = 1
while i<len(y) do
l1 = y[i]/ld[i]
lf.append(l1)
i+=1

i = 0
polinomio = ”

while i<len(l) do
if(lf[i]>=0):
polinomio+= ' + '+str(lf[i])+l[i]
else:
polinomio+= ' '+str(lf[i])+l[i]
i+=1
print(polinomio)
```

Newton:

read x0, iter, tol

fx = f(x0)

dFx = dF(x0)

cont = 1

error = tol + 1

while fx != 0 and error > tol and dFx != 0 and cont < iter

x1 = x0 - fx/dFx

fx = f(x1)

dFx = df(x1)

error = abs(x1 -x0)

x0 = x1

cont += 1

end while

if fx = 0

write "X0 is root"

else if error < tolerancia

write "X0 approximate root with tolerance: tol"

else if dfx = 0

write "X0 is probably a multiple root"

else

write "Fail in iteration: iter"

end if

Partial LU:

leer A, b :

```
n = longitud(A)
u = MatrizdeCeros(n)
l = MatrizDiagonal(n)
p = MatrizDiagonal(n)

para k hasta (n):
A, p = IntercambiarFilas(A, n, k, p)

para i hasta(k + 1, n):
mult = A[i][k] / A[k][k]
l[i][k] = mult
para j hasta (k, n):
A[i][j] = A[i][j] - mult * A[k][j]

escribir('u step', k)
para i hasta (n):
u[k][i] = A[k][i]


Pb = Producto_Punto(p, b)
lpb = concatenarMatriz(l, Pb)
z = SustitucionProgresiva(lpb)
uz = concatenarMatriz(u, z)
x = SustitucionRegresiva(uz)
escribir('z', z)
escribir('x', x)
```

Multiple roots:

```
read tol, x0, nIteration, function1, function2, function3
fx = function1 (x0)
dfx = function2 (x0)
d2fx = function3 (x0)

counter = 0
error = tol + 1
den = (dfx ^ 2) - (fx * d2fx)
while (error> tol and fx <> 0 and den <> 0 counter <nIteration)
x1 = x0 - ((fx * dfx) / den)
fx = function1 (x1)
dfx = function2 (x1)
d2fx = function3 (x1)
den = (dfx ^ 2) - (fx * d2fx)
error = abs (x1-x0)
x0 = x1
counter = counter + 1
end while
if (fx == 0)
x0 "is a root"
else if (error <tol)
x1 "was found as an approximation to a root with a tolerance of =" tol
else if (den == 0)
"it is an indeterminacy"
Else
"method failed in" nIteraccion "iterations"
end if
```

Secant:

```
read x1, x0, tol, nIteration

fun0 = f (x0)

if (fun0 = 0)

write "x0 is root"

else

fun1 = f (x1)

cont = 0

error = tol + 1

while (fun1 <> 0 and error> tol and den <> 0 and counter <nIteration)

x2 = x1 - ((fun1 * (x1 x0)) / den)

error = absolute_value ((x2 - x1) / x2)

x0 = x1

fun0 = fun1

x1 = x2

fun1 = f (x1)

den = fun1-fun0

cont = cont + 1
```

end while

if (fun1 = 0)

write "x1 was found as root"

else if (error <tol)

write x1 + "Found as an approximation with a tolerance of" + tol

else if (den = 0)

write "There is a possible multiple root"

else

write "Failure in" + nIteration "+" iterations "

end if

end if

Simple LU:

leer A, b:

n = longitud(A)

u = MatrizCeros(n)

l = MatrizDiagonal(n)

para k hasta (n):

if (A[k][k] == 0):

A = IntercambiarFilas(A, n, k)

```
para i hasta(k + 1, n):
mult = A[i][k] / A[k][k]
l[i][k] = mult
para j hasta (k, n):
A[i][j] = A[i][j] - mult * A[k][j]

escribir('u step', k)
para i hasta(n):
u[k][i] = A[k][i]
escribir(u)
escribir('u', u)
lb = concatenarMatriz(l, b)
z = SustitucionProgresiva(lb)
uz = concatenarMatriz(u, z)
x = SustitucionRegresiva(uz)
escribir('z ', z)
escribir('x ', x)
```

Sor:

```
leer sor A, b, x0, w, tol, Nmax:

results = { }
D = Diagonal(A)
L = -TraingularIferior(A) + D
U = -TraingularSuperiror(A) + D
T = Inversa(D - (w * L)) * ((1 - w) * D + (w * U))
C = w * Inversa(D - (w * L))* (b)
xant = x0
E = 1000
```

cont = 0

val = ValoresPropios(T)

resp = max(abs(val))

Mientras E > tol and cont < Nmax:

xact = T* xant + C

E = Normal(xant - xact)

xant = xact

cont = cont + 1

results[cont] = [float(E), xact]

fin

x = xact

escribir('Radio espectral', resp)

escribir('X', x)

escribir('T', T)

escribir('C', C)

escribir(results)

fin

Spline Quadratic:

matrix_cuad ():

read x

read b

a = [[0 for i from 0 to ((length (x) -1) * 3)] for j from 0 to ((length (x) -1) * 3)]

a [0] [0] = x [0] raised to the 2

a [0] [1] = x [0]

a [0] [2] = 1

a [1] [0] = x [1] to the power of 2

a [1] [1] = x [1]

a [1] [2] = 1

j = 3

for i from 2 to length (x):

a [i] [j] = x [i] raised to the 2

a [i] [j + 1] = x [i]

a [i] [j + 2] = 1

j + = 3

i = 1

j = 0

for k of length (x) up to ((length (x) raised to the 2) -2)):

b + = [0]

a [k] [j] = x [i] raised to the 2

a [k] [j + 1] = x [i]

a [k] [j + 2] = 1

a [k] [j + 3] = - (x [i] to the power of 2)

a [k] [j + 4] = -x [i]

a [k] [j + 5] = -1

i + = 1

j + = 3

i = 1

j = 0

for k from (length (x) raised to the 2) -2) to length (a) -1):

b + = [0]

a [k] [j] = 2 * x [i]

a [k] [j + 1] = 1

a [k] [j + 2] = 0

a [k] [j + 3] = -2 * x [i]

a [k] [j + 4] = -1

a [k] [j + 5] = 0

i + = 1

j + = 3

b + = [0]

a [len (a) -1] [0] = 2

return a, b

traces ():

read x

result = empty

j = 0

for i in from 0 to length (x):

if j == 0:

if x [i]> = 0.0:

result + = x [i]) + "x * * 2"

else:

result + = x [i] + "x * * 2"

elif j == 1:

if x [i]> = 0.0:

result + = "+" + x [i] + "x"

else:

result + = x [i] + "x"

else:

if x [i]> = 0.0:

result + = "+" + x [i] + ""

else:

result + = x [i] + ""

j = -1

j + = 1

write ("Traces:")

for i in result.split (""):

write (i)

Spline cubic:

def matrix_cub ():

read x

read b

a = [[0 for i from 0 to ((length (x) -1) * 4)] for j from 0 to ((length (x) -1) * 4)]

a [0] [0] = x [0] raised to the 3

a [0] [1] = x [0] raised to the 2

a [0] [2] = x [0]

a [0] [3] = 1

a [1] [0] = x [1] raised to the 3

a [1] [1] = x [1] to the power of 2

a [1] [2] = x [1]

a [1] [3] = 1

j = 4

for i from 2 to length (x):

a [i] [j] = x [i] raised to the 3

a [i] [j + 1] = x [i] raised to the 2

a [i] [j + 2] = x [i]

a [i] [j + 3] = 1

j + = 4

i = 1

j = 0

for k of length (x) up to ((length (x) raised to the 2) -2)):

b + = [0]

a [k] [j] = x [i] raised to the 3

a [k] [j + 1] = x [i] raised to the 2

a [k] [j + 2] = x [i]

a [k] [j + 3] = 1

a [k] [j + 4] = - (x [i] to the power of 3)

a [k] [j + 5] = - (x [i] to the power of 2)

a [k] [j + 6] = -x [i]

a [k] [j + 7] = -1

i + = 1

j + = 4

i = 1

j = 0

for k from (length (x) raised to the 2) -2) to length (x) * 3 -4):

b + = [0]

a [k] [j] = 3 * (x [i] to the power of 2)

a [k] [j + 1] = 2 * x [i]

a [k] [j + 2] = 1

a [k] [j + 3] = 0

a [k] [j + 4] = - (3 ∗ (x [i] to the power of 2))

a [k] [j + 5] = - (2 ∗ x [i])

a [k] [j + 6] = -1

a [k] [j + 7] = 0

i + = 1

j + = 4

i = 1

j = 0

for k from (length (x) raised to the 3) -4) to length ((x) ∗ 4) -6):

b + = [0]

a [k] [j] = 6 ∗ x [i]

a [k] [j + 1] = 2

a [k] [j + 2] = 0

a [k] [j + 3] = 0

a [k] [j + 4] = -6 ∗ x [i]

a [k] [j + 5] = -2

a [k] [j + 6] = 0

a [k] [j + 7] = 0

i + = 1

j + = 4

b + = [0] ∗ 2

a [len (a) -2] [0] = 6 ∗ x [0]

a [len (a) -2] [1] = 2

a [len (a) -1] [len (a) -4] = 6 ∗ x [len (x) -1]

a [len (a) -1] [len (a) -3] = 2

```
return a, b

def traces (x):
result = empty
j = 0
for i from 0 to length (x):
if j == 0:
if x [i]> = 0.0:
result + = "+" + x [i] + "x raised to 3"
else:
result + = x [i] + "x raised to 3"
elif j == 1:
if x [i]> = 0.0:
result + = "+" + x [i] + "x raised to 2"
else:
result + = x [i] + "x raised to 2"
elif j == 2:
if x [i]> = 0.0:
result + = "+" + x [i] + "x"
else:
result + = x [i] + "x"
else:
if x [i]> = 0.0:
result + = "+" + x [i] + ""
else:
result + = x [i] + ""
j = -1
```

j + = 1

write ("Traces: \ n")

for i in result.split (""):

write (i)

Spline linear:

matrix_lin ():

read x

read b

a = [[0 for i from 0 to ((length (x) -1) ∗ 2)] for j from 0 to ((length (x) -1) ∗ 2)]

a [0] [0] = x [0]

a [0] [1] = 1

a [1] [0] = x [1]

a [1] [1] = 1

j = 2

for i from 2 to length (x):

a [i] [j] = x [i]

a [i] [j + 1] = 1

j + = 2

i = 1

j = 0

for k of length (x) up to ((length (x) raised to the 2) -2)):

b + = [0]

a [k] [j] = x [i]

a [k] [j + 1] = 1

```
a [k] [j + 2] = -x [i]
a [k] [j + 3] = -1
i + = 1
j + = 2

return a, b


traces ():
read x


result = empty


for i from 0 to length x:
if i% 2 == 0:
if x [i]> = 0.0:
result + = "+" + x [i] + "x"
else:
result + = x [i] + "x"
else:
if x [i]> = 0.0:
result + = "+" + x [i] + ""
else:
result + = x [i] + ""


write ("Traces:")
for i in result.split (""):
write (i)
```

Vandermonde:

read x, y

matriz = []

while i < len(x) do

matriz.append([])

i++

fila = 0

for i in x do

j = len(x)-1

while j >= 0 do

matriz[fila].append(i* * j)

fila++

totalGaussianElimination(matriz,y)

Codes

Bisection:

import sympy as sm

def bisection(funcion, xi, xs, nIter, iter):

```
results = { }
if nIter > 0:
x = sm.symbols('x')
fxi = sm.sympify(funcion).subs(x, xi)
fxs = sm.sympify(funcion).subs(x, xs)
sm.plot(funcion)
if (fxi == 0):
print(fxi)
elif (fxs == 0):
print(fxs)
elif (fxs * fxi < 0):
xm = (xi + xs) / 2
fxm = sm.sympify(funcion).subs(x, xm)
count = 1
error = iter + 1
results[count] = [float(xi), float(xm), float(xs), float(fxm), float(error)]
while ((error > iter) and (count < nIter)):
if (fxi * fxm < 0):
xs = xm
else:
xi = xm
xaux = xm
xm = (xi + xs) / 2
fxm = sm.sympify(funcion).subs(x, xm)
error = abs(xm - xaux)
count += 1
results[count] = [float(xi), float(xm), float(xs), float(fxm), float(error)]
print(results)
return results
```

```
else:
results['message'] = 'Error'
print('el intervalo no sirve')
return results
```

Cholesky:

```
import numpy as np
import math

def cholesky(A, b):
    # Inicialización
    n = len(A)
    L = np.eye(n)
    U = np.eye(n)

    # factorization
    for i in range(n-1):
        suma = 0
        for j in range(i):
    suma += (L[i][j] * U[j][i])
```

```
L[i][i] = math.sqrt(A[i][i] - suma)
U[i][i] = L[i][i]

for k in range(i+1,n):
suma = 0
for j in range(i):
suma += (L[k][j] * U[j][i])
L[k][i] = (A[k][i] - suma) / U[i][i]

for k in range(i+1,n):
suma = 0
for j in range(i):
suma += (L[i][j] * U[j][k])
U[i][k] = (A[i][k] - suma) / L[i][i]

suma = 0
for j in range(n-1):
suma += (L[n-1][j] * U[j][n-1])
L[n-1][n-1] = math.sqrt(A[n-1][n-1] - suma)
U[n-1][n-1] = L[n-1][n-1]

print("Matriz L")
print(L)
print("Matriz U")
print(U)
z = frontSubstitution(L, b)
x = backSubstitution(U, z)
print(x)

def frontSubstitution(A, b):
n = len(A)
```

```
x = np.zeros((n))
for i in range(n):
    sum = 0
    for j in range(i):
        sum += A[i][j] * x[j]
    x[i] = (b[i] - sum) / A[i][i]
return x

def backSubstitution(A, b):
    n = len(A)
    x = np.zeros((n))
    for i in range(n-1, -1, -1):
        sum = 0.0
        for j in range (i+1, n):
            sum += A[i][j] * x[j]
        x[i] = (b[i] - sum) / A[i][i]
    return x
```

Crout:

```
import numpy as np

def Crout(a, b):
cout = 0
m, n = a.shape
if (m !=n ):
print("Crout cannot be used.")
else:
l = np.zeros((n,n))
u = np.zeros((n,n))
s1 = 0
s2 = 0

for m in range(1,n+1):
print("Stage " + str(m) + ": ")
for i in range(n):
l[i][0] = a[i][0]
u[i][i] = 1
for j in range(1, n):
u[0][j] = a[0][j] / l[0][0]
for k in range(1, n):
for i in range(k, n):
for r in range(k): s1 += l[i][r] * u[r][k]
l[i][k] = a[i][k] - s1
s1 = 0
for j in range(k+1, n):
for r in range(k): s2 += l[k][r] * u[r][j]
u[k][j] = (a[k][j] - s2) / l[k][k]
s2 = 0
```

```
print("U: ")
print(u)
print("L: ")
print(l)


y = np.zeros(n)
s3 = 0
y[0] = b[0] / l[0][0
for k in range(1, n):
    for r in range(k):
        s3 += l[k][r] * y[r]
    y[k] = (b[k]-s3) / l[k][k]
    s3 = 0

x = np.zeros(n)
s4 = 0
x[n-1] = y[n-1]
for k in range(n-2, -1, -1):
    for r in range(k+1, n):
        s4 += u[k][r] * x[r]
    x[k] = y[k] - s4
    s4 = 0

for i in range(n):
print("x" + str(i + 1) + " = ", x[i])
```

Divide diferences:

```
import numpy as np
```

```
def divide_dif(x,y):
n = len(x)
D = np.zeros((n,n))

D[:,0] = np.conjugate(y)

for i in range(1,n):
aux0 = D[i-1:n,i-1]
aux1 = np.diff(aux0)
aux2 = np.subtract(x[i:n],x[0:n-1-i+1])
D[i:n,i] = np.divide(aux1,np.transpose(aux2))

res = np.diag(D)

r = ” + ’{ 0:+} ’.format(res[0])
m = ’(x’ + ’{ 0:+} ’.format(-x[0]) + ’)’
for i in range(1,n):
r += ’{ 0:+} ’.format(res[i]) + m
m += ’(x’ + ’{ 0:+} ’.format(-x[i]) + ’)’
r = r.replace(’x+0’,’x’)
print(’Matrix D: \ n’,D)
print(’Coef: ’,res)
print(’Newton Polinom : ’, r)

return (r,D)
```

Doolittle:

```
import sympy as sm
import math
import sys
import json
```

```
import base64
import numpy as np

def doolittle(A,b,size):
A = np.array(A)
b = np.array(b)
L = np.eye(size)
U = np.eye(size)
print("Etapa 0:")
print("Matriz L: ")
print(L)
print("Matriz U: ")
print(U)
for i in range(size):
print("Etapa " + str(i+1))
for k in range(i, size):
suma = 0;
for j in range(i):
suma += (L[i][j] * U[j][k]);
U[i][k] = A[i][k] - suma;
for k in range(i, size):
if (i == k):
L[i][i] = 1;
else:
suma = 0;
for j in range(i):
suma += (L[k][j] * U[j][i]);
L[k][i] = ((A[k][i] - suma)/U[i][i]);

print("Matriz L: ")
```

```
print(L)
print("Matriz U: ")
print(U)
z = frontSubstitution(L, b)
x = backSubstitution(U, z)
printResultVector(x)

def frontSubstitution(A, b):
n = len(A)
x = np.zeros((n))
for i in range(n):
sum = 0
for j in range(i):
sum += A[i][j] * x[j]
x[i] = (b[i] - sum) / A[i][i]
return x

def backSubstitution(A, b):
n = len(A)
x = np.zeros((n))
for i in range(n-1, -1, -1):
sum = 0.0
for j in range (i+1, n):
sum += A[i][j] * x[j]
x[i] = (b[i] - sum) / A[i][i]
return x

def printResultVector(vector):
n = len(vector)
for i in range(n):
```

```
print('x' + str(i + 1) + ': ' + str(vector[i]))
```

FalseRule:

```
import sympy as sm
import math
import sys

def falseRule(a, b, funcion, limite_iteraciones, tolerancia):
results = { }
x = sm.symbols('x')
funcion = sm.sympify(funcion)
fa = funcion.subs(x, a)
fb = funcion.subs(x, b)
if(fa == 0):
print('a es raiz')
elif(fb == 0):
print('b es raiz')
```

elif(fa ∗ fb < 0):

error = 1

cont = 1

c = (fb∗ a - fa∗ b)/(fb - fa)

fc = funcion.subs(x, c)

print('iter| a | c | b | fc | error')

while(fc != 0 and cont < limite_iteraciones and error > tolerancia):

results[cont]=[float(a),float(c),float(b),float(fc),float(error)]

print(cont,a,c,b,fc,error)

if (fa ∗ fc < 0):

b = c

fb = funcion.subs(x, c)

else:

a = c

fa = funcion.subs(x, c)

caux = c

c = (fb∗ a - fa∗ b)/(fb - fa)

fc = funcion.subs(x, c)

error = abs(caux - c)

cont+=1

if (fc == 0):

print('c is a root '+ str(c))

results['message']='c is a root '+ str(c)

elif (error < tolerancia):

print('c is an approximation of the root c: '+ str(c) +' error: '+ str(error)+ ' in the iteration '+str(cont))

results['message']='c is an approximation of the root c: '+ str(c) +' error: '+ str(error)+ ' in the iteration '+str(cont)

else:

print('number of maximum iterations reached, convergence was not reached')

```
results['message']='number of maximum iterations reached, convergence was not reached'
else:
print('inadequate interval, does not satisfy the theorem fa * fb < 0')
results['message']='inadequate interval, does not satisfy the theorem fa * fb < 0'
return results
```

Fixed Point:

```
import math
def f(x):
return x** 3 + 4 * x** 2 - 10
def g(x):
return math.sqrt(10/(x+4))

def fixedPoint(xa, iter, tol):
fx = f(xa)
cont = 0
error = tol + 1
xn = 0

while((fx != 0) and error > tol and cont < iter):
xn = g(xa)
fx = f(xn)
error = abs(xn - xa)
xa = xn
```

```
cont += 1

if fx == 0:
print("Xa: ", xa, " is a root")
elif error < tol :
print("Xa: ", xa, " approximate root with tolerance: ", tol)
else:
print("Fail in iteration: ", iter)
```

Gaussian Elimination:

```
import numpy as np
import math
import copy

def simpleGaussianElimination(A, b):
Ab = concatenarMatriz(A, b)
n = len(Ab)
result = { }
for k in range(n):
print('step ',k)
print(Ab)
result[k]=copy.deepcopy(Ab)
if(Ab[k][k]==0):
Ab = searchAndSwapZero(Ab, n, k)
for i in range(k+1, n):
mult = Ab[i][k]/Ab[k][k]
for j in range(k, n+1):
```

```
Ab[i][j] = Ab[i][j] - mult* Ab[k][j]
print('x ',backSubstitution(Ab))
return(backSubstitution(Ab),result)

def searchAndSwapZero(Ab, n, i):
for j in range(i+1,n):
if(Ab[j][i]!=0):
temp = Ab[i]
Ab[i] = Ab[j]
Ab[j] = temp
break
return Ab

def partialGaussianElimination(A, b):
Ab = concatenarMatriz(A, b)
order = []
result = { }
n = len(Ab)
for k in range(n):
print('step ',k)
printMatriz(Ab)
result[k]=copy.deepcopy(Ab)
Ab = searchBiggerandSwap(Ab, n, k)
for i in range(k+1, n):
mult = Ab[i][k]/Ab[k][k]
for j in range(k, n+1):
Ab[i][j] = Ab[i][j] - mult* Ab[k][j]
print('X ',backSubstitution(Ab))
return(backSubstitution(Ab),result)
```

```
def searchBiggerandSwap(Ab, n, i):
row = i
for j in range(i+1,n):
if(abs(Ab[row][i]) < abs(Ab[j][i])):
row = j
temp = Ab[i]
Ab[i] = Ab[row]
Ab[row] = temp
return Ab

def totalGaussianElimination(A, b):
order = []
result = { }
Ab = concatenarMatriz(A, b)
n = len(Ab)
for k in range(n):
print('step ',k)
printMatriz(Ab)
result[k]=copy.deepcopy(Ab)
Ab, order = searchTheBiggestandSwap(Ab, n, k, order)
for i in range(k+1, n):
mult = Ab[i][k]/Ab[k][k]
for j in range(k, n+1):
Ab[i][j] = Ab[i][j] - mult* Ab[k][j]
# retorna las x
return(sortResult(backSubstitution(Ab), order),result)

def searchTheBiggestandSwap(Ab, n, k, order):
row = k
column = k
```

```
for i in range(k,n):
    for j in range(k,n):
        if(abs(Ab[row][column]) < abs(Ab[i][j])):
            row = i
            column = j
temp = Ab[k]
Ab[k] = Ab[row]
Ab[row] = temp
Ab = swapColumn(Ab, k, column)
order.append((k, column))
return (Ab, order)

def swapColumn(Ab, c1, c2):
    for i in range(len(Ab)):
        temp = Ab[i][c1]
        Ab[i][c1] = Ab[i][c2]
        Ab[i][c2] = temp
    return Ab

def sortResult(x, order):
    for i in range(len(order)-1, -1, -1):
        temp = x[order[i][0]]
        x[order[i][0]] = x[order[i][1]]
        x[order[i][1]] = temp
    return x

def concatenarMatriz(A, b):
    n = len(A)
    for i in range(n):
        A[i].append(b[i])
```

```
return A

# b = [[4],[5],[6]] this is the format
def concatenar(a,b):
a = np.array(a)
b =np.array(b)
matriz = np.concatenate((a, b), axis=1)
return matriz

def backSubstitution(Ab):
n= len(Ab)
x = []
for i in range(n):
x.append(0)

for i in range(n-1, -1, -1):
sum = 0
for j in range(i+1, n):
sum+= Ab[i][j]* x[j]
x[i] = (Ab[i][n]-sum)/Ab[i][i]
return x

def printMatriz(M):
result=”
for i in range(len(M)):
print(M[i])

def printMatriz2(M,k,result):
for i in range(len(M)):
print(M[i])
result[k]=M
```

return result

Gauss Seidel:

```
import math
import numpy as np

def gaussSeidel(A, b, t, iter, x0) :
n = len(A)
l = len(A)
if (n != l) :
print("A is nor a square matrix.")
return 0
else:
x = [None]* n
aux = 0
cont = 0
E = t + 1
iteration = 1
while (E > t and cont <= iter):
print("iter: " , iteration)
E = 0
for i in range(0,n):
suma = 0
for j in range(0,n):
if (i != j):
suma = suma + A[i][j] * x0[j]
x[i] = ( ((b[i] - suma) / A[i][i]))
```

```
aux = x[i] - x0[i]
E = E + math.pow(aux, 2)
x0[i] = x[i]
print("x" , (i + 1) , ": " , x0[i])

E = math.pow(E, 0.5)
print("E = " , E)
print("")
iteration = iteration + 1
cont = cont+1

if (E < t):
    return x
else:
print("Can not find a solution in " , iter , " iterations")
    return 0



A = [[4, -1, 0, 3],
    [1, 15.5, 3, 8],
    [0, -1.3, -4, 1.1],
    [14, 5, -2, 30]]
b = [1, 1, 1, 1]
x0 = [0, 0, 0, 0]
t = math.pow(10, -7)
iter = 100
gaussSeidel(A, b, t, iter, x0)

D = np.diag(np.diag(A))
```

```
U = -np.triu(A,1)
L = -np.tril(A,-1)
T = (np.dot((np.linalg.inv(D-L)), U))
C = (np.dot((np.linalg.inv(D-L)), b))
print("T: ")
print(T)
print("C:")
print(C)
values, normalized_eigenvectors = np.linalg.eig(T) #  T es la matriz
spectral_radius = max(abs(values))
print("\ nSpectral Radius: ", spectral_radius)
```

Incremental search:

```
import sympy as sm
import sys
import pandas as pd
```

```
def incremental_search(funcion,xi, delta, nIter):
results = { }
if delta <= 0:
print("El delta debe ser positivo")
sys.exit(1)
elif nIter > 0:

x = sm.symbols('x')
x_a = xi
current_X = x_a+delta
f_a = sm.sympify(funcion).subs(x,x_a)
currentF = sm.sympify(funcion).subs(x,current_X)
contador = 0
while (contador < nIter):
if currentF* f_a<0:
results[contador] = [float(x_a),float(current_X)]
x_a = current_X
current_X = current_X + delta
f_a = currentF
currentF = sm.sympify(funcion).subs(x,current_X)
contador = contador + 1

return results

else:
print("Las iteraciones deben ser un numero positivo")
results['message'] = 'Error'
print('el intervalo no sirve')
return results
```

```
def print_function(results):
index = []
x1 = []
x2 = []
for i in results:
index.append(i)
x1.append(results[i][0])
x2.append(results[i][1])

data = { 'xi': x1,
'xs': x2,
}
df = pd.DataFrame(data, index=index)
print(df)
```

Jacobi:

```
import math
import numpy as np
def Jacobi(A, b, t, iter, x0):
n = len(A)
l = len(A[0])
result = { }
if (n!=l):
```

```
return("A is not a square matrix please check and run again.")
else:
x =[None] * n
aux=0
cont = 0
error = t + 1
iteration = 1
T = np.zeros((n, n))
C = np.zeros(n)

while(error > t and cont <= iter):
error = 0
for i in range(0,n):
sum = 0
for j in range(0,n):
if (i != j):
sum = sum + A[i][j] * x0[j]
T[i][j] = -A[i][j] / A[i][i]
C[i] = b[i] / A[i][i]
x[i] = (b[i] - sum) / A[i][i]
aux = x[i] - x0[i]
error = error + math.pow(aux, 2)

error = math.pow(error, 0.5)
for i in range(0,n):
x0[i] = x[i]
print("x"+str(i+1)+": "+str(round(x0[i],4)))
result[iteration]=(float(error),x)
```

```
iteration=iteration+1
cont = cont+1

print(result)
print("")
print("T: \ n"+str(T))
print("")
print("C: \ n"+str(C))
print("")
spectralRadius = np.amax(abs(T))
print("Spectral radius: \ n"+str(spectralRadius))


if (error < t):
return(result, T, C,spectralRadius )
else:
print ("no solution reached in " + str (iter) + " iterations")
return
```

Lagrange:

```
import math

def lagrange(x, y):
    n = len(x)
    l = []
    ld = []
    for i in x:
        l1 = ''
        l2 = 1
        for j in x:
            if(i!=j):
                if(j<0):
                    l1+= '(x + '+str(abs(j))+') '
                else:
                    l1+= '(x - '+str(abs(j))+') '
                l2 = l2* (i-j)
        l.append(l1)
        ld.append(l2)

    i = 0
    lf = []
    l1 = 1
    while(i<len(y)):
        l1 = y[i]/ld[i]
        lf.append(l1)
        i+=1


    i = 0
```

```
polinomio = ''
while(i<len(l)):
if(lf[i]>=0):
polinomio+= ' + '+str(lf[i])+l[i]
else:
polinomio+= ' '+str(lf[i])+l[i]
i+=1
return(polinomio)
```

Newton:

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

def f(x):
return x**3 - np.cos(x)

def dF(x):
```

```
return 3 ∗ x∗ ∗ 2 + np.sin(x)

def newton(x0, iter, tol):
fx = f(x0)
dFx = dF(x0)
cont = 1
error = tol + 1

while( (fx != 0) and error > tol and (dFx != 0) and cont < iter):
x1 = x0 - fx/dFx
fx = f(x1)
dFx = dF(x1)
error = abs(x1-x0)
x0 = x1
cont += 1

if fx == 0:
print("X0: " ,x0, " is root")
elif error < tol:
print("X0: ", x0, " approximate root with tolerance: ", tol)
elif dFx == 0:
print("X0: ", x0, " is probably a multiple root")
else:
print("Fail in iteration: ", iter)

def draw():
x = np.linspace(-2, 2, 100)
plt.plot(x, x∗ ∗ 3 - np.cos(x))
plt.grid()
plt.show()
```

```
draw()
```

Partial LU:

```
import copy
import numpy as np

def partialLU(A, b):
n = len(A)
message = ''
det = np.linalg.det(A)
if(det != 0):
u = zero_Matrix(n)
l = lmatrix(n)
p = lmatrix(n)
dicL = { }
dicU = { }
dicP = { }
for k in range(n):
```

```
print('step ', k)
printMatriz(A)
print('L step', k)
printMatriz(l)
dicL[k] = copy.deepcopy(l)
dicU[k] = copy.deepcopy(u)
A, p = searchBiggerandSwap(A, n, k, p)
for i in range(k + 1, n):
    mult = A[i][k] / A[k][k]
    l[i][k] = mult
    for j in range(k, n):
        A[i][j] = A[i][j] - mult * A[k][j]

print('u step', k)
for i in range(n):
    u[k][i] = A[k][i]
printMatriz(u)
dicU[k] = copy.deepcopy(u)
print('P step', k)
printMatriz(p)
dicP[k] = copy.deepcopy(p)

Pb = multAb(p, b)
lpb = concatenateMatrix(l, Pb)
z = progSubtitution(lpb)
uz = concatenateMatrix(u, z)
x = backSubstitution(uz)
print('det', det)
return (x,dicL,dicU,dicP,message)
else:
```

```
message = 'Error'
return ('','','',message)

def searchBiggerandSwap(Ab, n, i, p):
row = i

for j in range(i + 1, n):
if (abs(Ab[row][i]) < abs(Ab[j][i])):
row = j
temp = Ab[i]
aux = p[i]
Ab[i] = Ab[row]
p[i] = p[row]
Ab[row] = temp
p[row] = aux
return Ab, p

def concatenateMatrix(A, b):
n = len(A)
for i in range(n):
A[i].append(b[i], )
return A

def multAb(A, b):
n = len(A)
mult = []
for i in range(n):
suma = 0
for j in range(n):
suma += b[j] * A[i][j]
```

```
mult.append(suma)
return mult

def zero_Matrix(n):
u = []
for i in range(n):
u.append([0] * n)
return u

def lmatrix(n):
l = zero_Matrix(n)
for i in range(n):
l[i][i] = 1
return l

def backSubstitution(Ab):
n = len(Ab)
x = []
for i in range(n):
x.append(0)

for i in range(n - 1, -1, -1):
sum = 0
for j in range(i + 1, n):
sum += Ab[i][j] * x[j]
x[i] = (Ab[i][n] - sum) / Ab[i][i]
return x

def progSubtitution(Ab):
n = len(Ab)
x = []
```

```
for i in range(n):
    x.append(0)

for i in range(n):
    sum = 0
    for j in range(i):
    sum += Ab[i][j] * x[j]
x[i] = ((Ab[i][n] - sum) / Ab[i][i])
    return x
def printMatriz(M):
for i in range(len(M)):
    print(M[i])
```

Multiple roots:

```
import sympy as sm
import math
def roots_mult(x0, nInterations, tol, function, function2, function3):
results = { }
add = { }
x = sm.symbols ('x')
fx0 = sm.sympify (function) .subs (x, x0)
dfx0 = sm.sympify (function2) .subs (x, x0)
d2fx0 = sm.sympify (function3) .subs (x, x0)

cont = 0
error = tol + 1
xn = 0
det = (math.pow (dfx0,2)) - (fx0 * d2fx0)
results [cont] = [float (x0), float (fx0), float (0)]
while (fx0!= 0 and error> tol and det != 0 and cont <nInterations):
xn = x0 - ((fx0 * dfx0) / det)
fx0 = sm.sympify (function) .subs (x, xn)
dfx0 = sm.sympify (function2) .subs (x, xn)
d2fx0 = sm.sympify (function3) .subs (x, xn)
det = (math.pow (dfx0,2)) - (fx0 * d2fx0)
error = abs (xn-x0)
x0 = xn
cont = cont + 1
results [cont] =([round(float (x0),18), round(float (fx0),18), round(float (error),18)])
```

```
if (fx0 == 0):
add =(str (x0) + "is a root")

elif (error<tol):
add =(str (x0) + "was found as an approximation with a tolerance of =" + str (tol))

elif (det == 0):
add =("is an indeterminacy")

else:
add =("The method failed in" + str (nInterations) + "iterations")

return results,add
```

Secant:

```
import abc
import sympy as sm

def secant(x0, x1, nInterations, tol, function):
results = { }
```

```
add = { }

x = sm.symbols('x')
fx0 = sm.sympify(function).subs(x, x0)

if (fx0 == 0):
print(x0 + "is root")
else:
fx1 = sm.sympify(function).subs(x, x1)
cont = 1
error = tol + 1
det = fx1 - fx0
xi = 0
results [cont-1] = [float (x0), float (fx0), float (0)]
results [cont] = [float (x1), float (fx1), float (0)]
while (fx1 != 0 and error> tol and det!= 0 and cont <nInterations):

xi = x1 -((fx1 * (x1-x0)) / det)
error = abs(xi-x1)
x0 = x1
fx0 = fx1
x1 = xi
fx1 = sm.sympify(function).subs(x, x1)
det = fx1 - fx0
cont = cont + 1
results [cont] = ([round(float (xi),10), round(float (fx1),10), round(float (er-
ror),10)])
```

if (fx1 == 0):

add = (str(x0)+" is a root")

elif (error <tol):

add= (str(x1)+" was found as an approximation with a tolerance of = "+str(tol))

elif (det == 0):

add= (str(x1)+"is a possible multiple root")

else:

add=("The method failed in "+str(nInterations)+ " iterations")

return results,add

Simple LU:

import numpy as np

import math as math

import copy

def simpleLU(A, b):

n = len(A)

det = np.linalg.det(A)

message = ”

if(det != 0):

```
u = zero_Matrix(n)
l = lmatrix(n)
dicL = { }
dicU = { }

for k in range(n):
    print('step ', k)
    printMatriz(A)
    print('L step', k)
    printMatriz(l)
    dicL[k] = copy.deepcopy(l)
    dicU[k] = copy.deepcopy(u)
    if (A[k][k] == 0):
        A = searchAndSwapZero(A, n, k)
    for i in range(k + 1, n):
        mult = A[i][k] / A[k][k]

        l[i][k] = mult
        for j in range(k, n):
            A[i][j] = A[i][j] - mult * A[k][j]

    print('u step', k)
    for i in range(n):
        u[k][i] = A[k][i]
    printMatriz(u)
    dicU[k] = copy.deepcopy(u)

print('u', dicU)
lb = concatenateMatrix(l, b)
z = progSubtitution(lb)
```

```
uz = concatenateMatrix(u, z)
x = backSubstitution(uz)
print('x ', x)

return (x,dicL,dicU,message)
else:
message = 'Error'
return ('', '', '', message)
def searchAndSwapZero(Ab, n, i):
for j in range(i + 1, n):
if (Ab[j][i] != 0):
temp = Ab[i]
Ab[i] = Ab[j]
Ab[j] = temp
break
return Ab

def concatenateMatrix(A, b):
n = len(A)
for i in range(n):
A[i].append(b[i], )
return A

def multAb(A, b):
n = len(A)
mult = []
for i in range(n):
suma = 0
for j in range(n):
suma += b[j] * A[i][j]
```

```
mult.append(suma)
return mult

def zero_Matrix(n):
u = []
for i in range(n):
u.append([0] * n)
return u

def lmatrix(n):
l = zero_Matrix(n)
for i in range(n):
l[i][i] = 1
return l

def backSubstitution(Ab):
n = len(Ab)
x = []
for i in range(n):
x.append(0)

for i in range(n - 1, -1, -1):
sum = 0
for j in range(i + 1, n):
sum += Ab[i][j] * x[j]
x[i] = (Ab[i][n] - sum) / Ab[i][i]
return x

def progSubtitution(Ab):
n = len(Ab)
x = []
```

```
for i in range(n):
    x.append(0)

for i in range(n):
    sum = 0
    for j in range(i):
    sum += Ab[i][j] * x[j]
x[i] = ((Ab[i][n] - sum) / Ab[i][i])
return x
def printMatriz(M):
for i in range(len(M)):
    print(M[i])
```

Sor:

```
import numpy as np
from scipy import linalg as LA
import pandas as pd
import copy

def sor(A, b, x0, w, tol, Nmax):
results = { }
det = np.linalg.det(A)
if(det != 0):
d = np.diag(A)
D = np.diagflat(d)
L = -np.tril(A) + D
U = -np.triu(A) + D
T = np.linalg.inv(D - (w * L)).dot((1 - w) * D + (w * U))
C = w * np.linalg.inv(D - (w * L)).dot(b)
```

```
xant = x0
E = 1000
cont = 0
val, evec = np.linalg.eig(T)
resp = max(abs(val))
while E > tol and cont < Nmax:
xact = np.dot(T, xant) + C
short = ['{ :.6f} '.format(elem) for elem in xact]
E = np.linalg.norm(xant - xact)
xant = xact
cont = cont + 1
results[cont] = [float(E), short]

print_iter(results)
print('Spectral Radious ', resp)
print('T', T)
print('C', C)

return (resp,T,C,results)
else:
results['message'] = 'Error'
return ('','','',results)

def print_iter(results):
index = []
x = []
error = []
for i in results:
index.append(i)
error.append(results[i][0])
```

x.append(results[i][1])

data = { 'Error': error,

'X': x

}

df = pd.DataFrame(data, index=index)

print(df)

Spline Quadratic

from Metodos.GaussianElimination_splines import simpleGaussianElimination,partialGaussianElimination,totalGaussianElimination, backSubstitution, sortResult

def matrix_cuad(x, b):

a = [[0 for i in range((len(x)-1)* 3)] for j in range((len(x)-1)* 3)]

a[0][0] = x[0]* * 2

a[0][1] = x[0]

a[0][2] = 1

a[1][0] = x[1]* * 2

a[1][1] = x[1]

a[1][2] = 1

j = 3

for i in range(2,len(x)):

a[i][j] = x[i]* * 2

a[i][j+1] = x[i]

a[i][j+2] = 1

```
j += 3

i = 1
j = 0
for k in range(len(x),((len(x)* 2)-2)):
b += [0]
a[k][j] = x[i]* * 2
a[k][j+1] = x[i]
a[k][j+2] = 1
a[k][j+3] = -(x[i]* * 2)
a[k][j+4] = -x[i]
a[k][j+5] = -1
i += 1
j += 3

i = 1
j = 0
for k in range(((len(x)* 2)-2),len(a)-1):
b += [0]
a[k][j] = 2* x[i]
a[k][j+1] = 1
a[k][j+2] = 0
a[k][j+3] = -2* x[i]
a[k][j+4] = -1
a[k][j+5] = 0
i += 1
j += 3

b += [0]
```

a[len(a)-1][0] = 2

return a,b

def traces(x):

result = ""

j = 0

for i in range(len(x)):

if j == 0:

\# print(x[i])

if x[i] >= 0.0:

result += "+"+str(x[i])+"x* * 2"

else:

result += str(x[i])+"x* * 2"

elif j == 1:

if x[i] >= 0.0:

result += "+"+str(x[i])+"x"

else:

result += str(x[i])+"x"

else:

if x[i] >= 0.0:

result += "+"+str(x[i])+" "

else:

result += str(x[i])+" "

j = -1

j += 1

return(result)

def trazcuad_spline(x,y,d):

”’ x = [-1,0,3,4]

y = [15.5,3,8,1]”’

b = y

A, b = matrix_cuad(x,b)

if(d=="T"):

t1=totalGaussianElimination(A, b)

return traces(t1[0]),A,b,t1[1],t1[0]

elif(d=="P"):

t2=partialGaussianElimination(A, b)

return traces(t2[0]),A,b,t2[1],t2[0]

elif(d=="S"):

t3=simpleGaussianElimination(A, b)

return traces(t3[0]),A,b,t3[1],t3[0]

Spline Linear:

```
from Metodos.GaussianElimination_splines import simpleGaussianElimination,partialGaussianElimination,totalGaussianElimination

def matrix_lin(x, b):
a = [[0 for i in range((len(x)-1)* 2)] for j in range((len(x)-1)* 2)]
a[0][0] = x[0]
a[0][1] = 1
a[1][0] = x[1]
a[1][1] = 1

j = 2
for i in range(2,len(x)):
a[i][j] = x[i]
a[i][j+1] = 1
j += 2


i = 1
j = 0
for k in range(len(x),((len(x)* 2)-2)):
b += [0]
a[k][j] = x[i]
a[k][j+1] = 1
a[k][j+2] = -x[i]
a[k][j+3] = -1
i += 1
j += 2

return a,b
```

```
def traces(x):

result = ""
for i in range(len(x)):
if i % 2 == 0:
if x[i] >= 0.0:
result += "+"+str(x[i])+"x"
else:
result += str(x[i])+"x"
else:
if x[i] >= 0.0:
result += "+"+str(x[i])+" "
else:
result += str(x[i])+" "

return(result)

def trazlin_spline(x,y,d):

b = y
A, b = matrix_lin(x,b)

if(d=="T"):
t1=totalGaussianElimination(A, b)
return traces(t1[0]),A,b,t1[1],t1[0]

elif(d=="P"):
```

t2=partialGaussianElimination(A, b)

return traces(t2[0]),A,b,t2[1],t2[0]

elif(d=="S"):

t3=simpleGaussianElimination(A, b)

return traces(t3[0]),A,b,t3[1],t3[0]

Spline Cubic

from Metodos.GaussianElimination_splines import simpleGaussianElimination,partialGaussianElimination,totalGaussianElimination, backSubstitution, sortResult

def matrix_cub(x, b):

a = [[0 for i in range((len(x)-1)* 4)] for j in range((len(x)-1)* 4)]

a[0][0] = x[0]* * 3

a[0][1] = x[0]* * 2

a[0][2] = x[0]

```
a[0][3] = 1
a[1][0] = x[1]**3
a[1][1] = x[1]**2
a[1][2] = x[1]
a[1][3] = 1

j = 4
for i in range(2,len(x)):
    a[i][j] = x[i]**3
    a[i][j+1] = x[i]**2
    a[i][j+2] = x[i]
    a[i][j+3] = 1
    j += 4

i = 1
j = 0
for k in range(len(x),((len(x)* 2)-2)):
    b += [0]
    a[k][j] = x[i]**3
    a[k][j+1] = x[i]**2
    a[k][j+2] = x[i]
    a[k][j+3] = 1
    a[k][j+4] = -(x[i]**3)
    a[k][j+5] = -(x[i]**2)
    a[k][j+6] = -x[i]
    a[k][j+7] = -1
    i += 1
    j += 4


i = 1
```

```
j = 0
for k in range(((len(x)* 2)-2),((len(x)* 3)-4)):
b += [0]
a[k][j] = 3* (x[i]* * 2)
a[k][j+1] = 2* x[i]
a[k][j+2] = 1
a[k][j+3] = 0
a[k][j+4] = -(3* (x[i]* * 2))
a[k][j+5] = -(2* x[i])
a[k][j+6] = -1
a[k][j+7] = 0
i += 1
j += 4

i = 1
j = 0
for k in range(((len(x)* 3)-4),((len(x)* 4)-6)):
b += [0]
a[k][j] = 6* x[i]
a[k][j+1] = 2
a[k][j+2] = 0
a[k][j+3] = 0
a[k][j+4] = -6* x[i]
a[k][j+5] = -2
a[k][j+6] = 0
a[k][j+7] = 0
i += 1
j += 4
```

```
b += [0]* 2
a[len(a)-2][0] = 6* x[0]
a[len(a)-2][1] = 2
a[len(a)-1][len(a)-4] = 6* x[len(x)-1]
a[len(a)-1][len(a)-3] = 2

return a, b

def traces(x):
result = ""
j = 0
for i in range(len(x)):
if j == 0:
if x[i] >= 0.0:
result += "+"+str(x[i])+"x* * 3"
else:
result += str(x[i])+"x* * 3"
elif j == 1:
if x[i] >= 0.0:
result += "+"+str(x[i])+"x* * 2"
else:
result += str(x[i])+"x* * 2"
elif j == 2:
if x[i] >= 0.0:
result += "+"+str(x[i])+"x"
else:
result += str(x[i])+"x"
else:
if x[i] >= 0.0:
result += "+"+str(x[i])+" "
```

```
else:
result += str(x[i])+""
j = -1
j += 1

return(result)

def trazcub_spline(x,y,d):

b = y
A, b = matrix_cub(x,b)

if(d=="T"):
t1=totalGaussianElimination(A, b)
return traces(t1[0]),A,b,t1[1],t1[0]

elif(d=="P"):
t2=partialGaussianElimination(A, b)
return traces(t2[0]),A,b,t2[1],t2[0]
elif(d=="S"):
t3=simpleGaussianElimination(A, b)
return traces(t3[0]),A,b,t3[1],t3[0]
```

Vandermonde:

```
from Metodos.GaussianElimination import *

def vandermonde(x, y):
matriz = []
for i in range(len(x)):
matriz.append([])
fila = 0

for i in x:
for j in range(len(x)-1,-1,-1):
matriz[fila].append(i* * j)
fila+=1
print('A:')
printMatriz(matriz)
return totalGaussianElimination(matriz, y)[0]
```