A picture containing diagram

Description automatically generatedAlexandria University

Faculty of engineering

Computer and Systems Engineering Department

CSE-211: Numerical Computations

Project - phase 1

Solving system of linear equations

Team’s members:

|  |  |
| --- | --- |
| Name | ID |
| Aaser Fawzy Zakaria Hassan | 19015403 |
| Mohammed Ezzat Saad El-Shazly | 19016441 |
| Ahmed Hamdy Ahmed Osman | 19017253 |
| Ahmed Ashour El-Sayed Mohammed Soliman | 19015267 |

# Pseudo-code for each function:

This function takes the systems of equations

and check if they are right and return the number of lines

function validations (string)

loop over the entered string

add the newline and = signs in

separate array

if you find 2 signs or two dots after each

return false

if you find the newline did equal the equal sign

return false

if you find 2 or more = in the same line

return false

return number of lines

This function takes the string and find the variables array

function parsingVar (string)

flag =false

vars=[]

loop over the string

if the character is alphabetic

append it queue

flag = true

else if operation sign is found and flag = true

empty queue in x

if x is not in variables vars

append x in vars

else if flag = true and the character is number

append character to the queue

else if the character is weird operation then

return wrong format

if queue > 2

return wrong variables

return vars

Takes the string and the variables and return the coefficients and the values array

function parsingCoff (vars, string)

row=0

coff=[][]

value=[]

flag = false

loop over the string

if the character is numeric or. or -

append it queue1

else if the character is alphabetic

append it queue2

flag = true

else if operation sign is found and flag = true

empty queue2 in x

Colum = index of x from vars array

if queue1 is not empty

empty queue1 in y

if y = -

add -1 in coff[row][Colum]

else

add y in coff[row][Colum]

else

put 1 in coff[row][Colum]

else if flag = true and the character is number

append character to the queue2

else if operation sign is found and flag = false

empty queue1 in y

add y to value[row]

if character is new line

row++

return cofs,values

Make chelosky decomposition and return an array of upper and lower matrix and number of iterations

function cheloskey (x,n,rounds)

a=numpy array of x

if a is not positive defined or not symmetric

return “cannot do chelosky”

L=2d zeros numpy array(n)

for j in range(n):   
 for i in range(j,n):

if i==j:

sum=0

for l in range(j):

sum = round(sum+ L[i,l]\*\**2 , to rounds)*

*L[i,j] = round(np.sqrt(a[i,j]-summ), to rounds)*

*else:*

*sum=0*

*for l in range(j):*

*sum=round( sum+L[i,l]\**L[j,l],to rounds)

L[i,j]=round((a[i,j]-sum)/L[j,j],to rounds)

return L , L.T

*Forward\_Elimination()* function: performs the forward elimination step in Gauss & Gauss-Jordan methods.

Returns the coefficients matrix, the constants array, & the number of iterations

**Function** Forward\_Elimination(n, A, B, precision, iterations)

Factor ← 0

**For** i from 0 to n:

iterations ← iterations + 1

A, B ← pivoting(n, i, A, B) //calling

**For** j from i+1 to n-1:

iterations ← iterations + 1

**try**:

factor ← A[j][i] / A[i][i]

factor ← round(factor, precision) //rounding

**catch** *dividingByZeroException*:

**RETURN** A, B, 0 //zeroing the iterations as evidence

**EndTry**

**For** k from i to n-1:

iterations ← iterations + 1

A[j][k] ← A[j][k] – factor \* A[i][k]

A[j][k] ← round(A[j][k], precision) //rounding

**EndFor**

B[j] = B[j] – factor \* B[i]

B[j]← round(B[j], precision) //rounding

**EndFor**

**EndFor**

**RETURN** A, B, iterations

**Gauss Elimination** *solve()* function: takes the size of the matrix, the coefficients matrix, the constants array, and the precision.

Returns an array of 3 elements: (the solution array, time complexity, & number of iterations)

**Function** GaussE.solve(n, A, B, precision)

begin\_time ← timer() //start time of the function

iterations ← 0

//calling Forward\_Elimination()

A, B, iterations ← Forward\_Elimination(n, A, B, precision, iterations)

**IF** iterations = 0 **THEN** //if dividing by zero occurs

**RETURN** ‘The system has infinite number of solutions’

**EndIF**

**IF** A[n-1][n-1]= 0 **THEN**

**IF** B[n-1] = 0 **THEN**

**RETURN** ‘The system has infinite number of solutions’

**ELSE**

**RETURN** ‘The system has no solution’

**EndIF**

**EndIF**

X[n-1] ← B[n-1] / A[n-1][n-1]

X[n-1] ← round(X[n-1], precision) //rounding

**For** i from n-2 to 0:

sum ← 0

**For** j from i+1 to n-1:

iterations ← iterations + 1

sum ← sum + A[i][j]

sum ← round(sum, precision) //rounding

**EndFor**

X[i] ← (B[i] – sum) / A[i][i]

X[i] ← round(X[i], precision) //rounding

**EndFor**

time ← timer() – begin\_time //getting the run time

**RETURN** array of(solution Array X, time, and iterations)

**Gauss-Jordan** *solve()* function: takes the size of the matrix, the coefficients matrix, the constants array, and the precision.

Returns an array of 3 elements: (the solution array, time complexity, & number of iterations)

**Function** GaussJ.solve(n, A, B, precision)

begin\_time ← timer() //start time of the function

iterations ← 0

//calling Forward\_Elimination()

A, B, iterations ← Forward\_Elimination(n, A, B, precision, iterations)

**IF** iterations = 0 **THEN** //if dividing by zero occurs

**RETURN** ‘The system has infinite number of solutions’

**EndIF**

**IF** A[n-1][n-1]= 0 **THEN**

**IF** B[n-1] = 0 **THEN**

**RETURN** ‘The system has infinite number of solutions’

**ELSE**

**RETURN** ‘The system has no solution’

**EndIF**

**EndIF**

**For** i from n-1 to 1:

iterations ← iterations + 1

**For** j from i-1 to 0:

iterations ← iterations + 1

factor ← A[j][i] / A[i][i]

factor ← round(factor, precision) //rounding

**For** k from i to 0:

iterations ← iterations + 1

A[j][k] ← A[j][k] – factor \* A[i][k]

A[j][k] ← round(A[j][k], precision) //rounding

**EndFor**

B[j] ← B[j] – factor \* B[i]

B[j] ← round(B[j], precision) //rounding

**EndFor**

**EndFor**

**For** i from 0 to n-1:

iterations ← iterations + 1

X[i] ← B[i] / A[i][i]

X[i] ← round(X[i], precision) //rounding

**EndFor**

time ← timer() – begin\_time //getting the run time

**RETURN** array of(solution Array X, time, and iterations)

*pivoting()* function: takes the size of the matrix, the index of the pivot, the coefficients matrix, & the constants array.

Returns the coefficients matrix, & the constants array.

**Function** pivoting(n, p, A, B)

max\_index ← p

**For** i from p+1 to n-1:

**IF** A[max\_index][p] < A[i][p] **THEN**

max\_index ← i

**EndIF**

**EndFor**

temp ← 0

**For** i from 0 to n-1:

temp ← A[p][i]

A[p][i]← A[max\_index][i]

A[max\_index][i]← temp

**EndFor**

temp ← B[p]

B[p]← B[max\_index]

B[max\_index] ← temp

**RETURN** A, B

#pseudo code for crout

If not a square matrix

Return error

Upper = [] #prepare upper

Lower = [] #prepare lower

For I= 0 to length #matrix length

Upper[i][i] = 1 #diagonal of ones

For j=0 to length #fill lower

For k=0 to i

Calculate sum of lower[j][k] \* upper[k][i] #from formula

Lower[j][i] = matrix[j][i] – sum #from formula

For j = 0 to length

For k= 0 to i

Calculate sum of lower[i][k] \* upper[k][j] # from formula

Upper[i][j] = (matrix[i][j] – sum) / lower[i][i] #from formula

Return upper & lower

#pseudo code for jacobi and seidel iterations

Guess = initial guess

While I < iterations+1000 #more iterations to check for convergence

Prevguess = guess # save prev guess

Guess = getguesses(prevguess)

Calculate error

If error < tolerance

break

I = I + 1

If I > iterations #save final result to be displayed

lastGuess = guess

round errors and guesses

if I > iterations

if I = iterations +1000

the result will diverge

else

the result will converge

Return lastGuess

#pseudo code for jacobi guessing

prevGuess = prevGuess

Arr = [] # to store the solution

For i=0 to length #matrix length

If needs downward pivoting

Pivot()

Last = coffarray[i][length-1]

X = last

For j=0 to length

If diagonal element

Skip a loop

X = x – coffarray[i][j] \* prevGuess[j] #guess with the prevGuess

X = x/diagonal element

Arr.push(x)

Return arr #new guesses

#pseudo code for seidel guessing

prevGuess = prevGuess

Arr = [] # to store the solution

For i=0 to length #matrix length #pick a row

If needs downward pivoting

Pivot()

Last = coffarray[i][length-1]

prevGuess[i] = last # prepare for the guess of the current guess

For j=0 to length #loop inside the row

If diagonal element

Skip a loop

#update prevGuess with the new guess

prevGuess[i] = prevGuess[i] – coffarray[i][j] \* prevGuess[j]

prevGuess[i]= prevGuess[i]/diagonal element

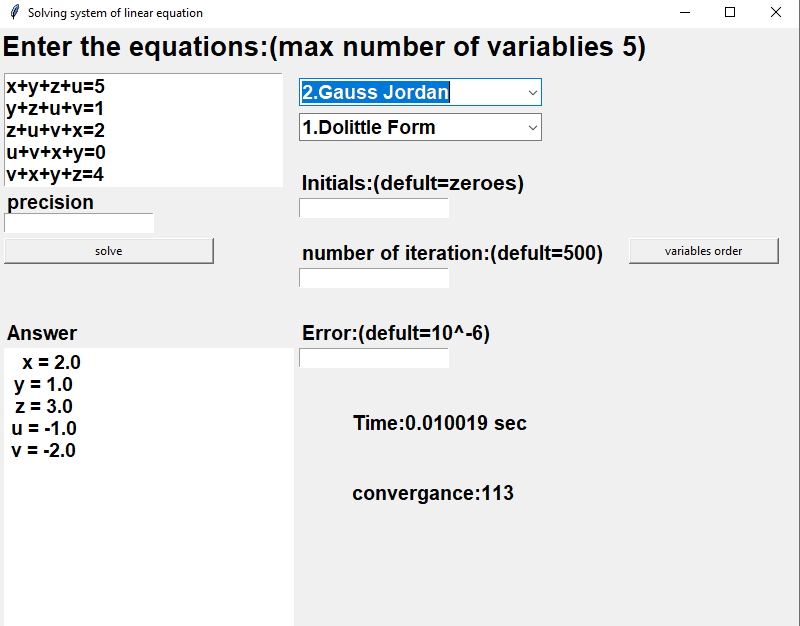
Arr.push(PrevGuess[i])

Return arr #new guesses

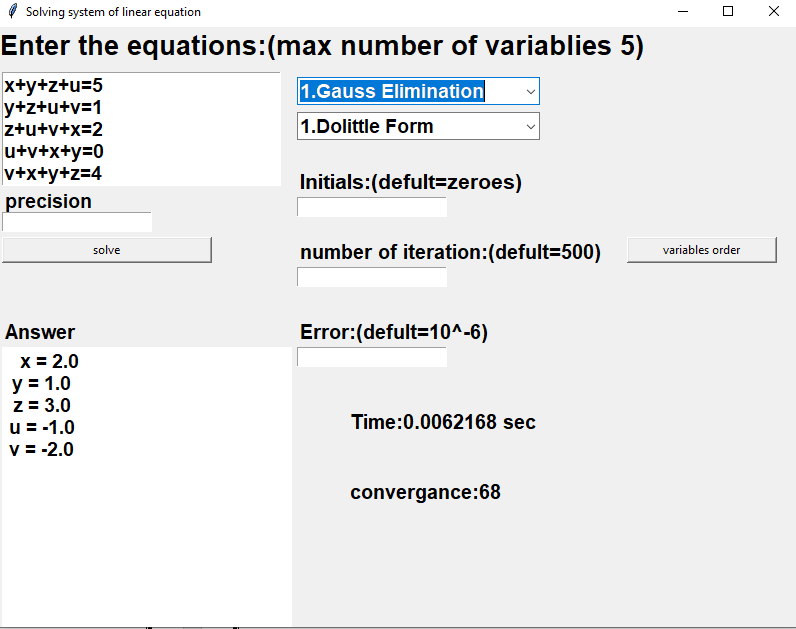
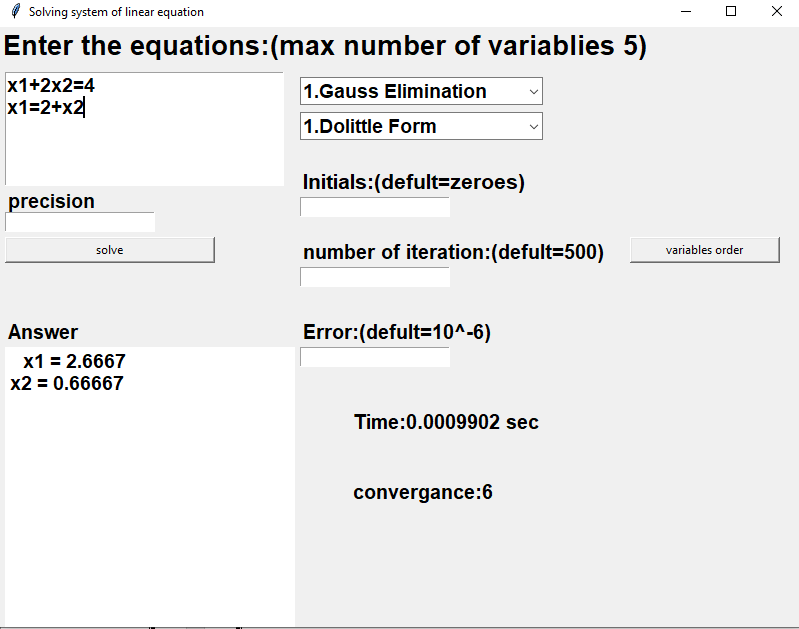
# Sample runs:

Graphical user interface, application, email

Description automatically generated

Graphical user interface, application, email

Description automatically generated



# 

# Comparison between different methods:

# Data Structure used: