TEC | Tecnológico de Costa Rica

Enseñanza de la Matemática con Entornos Tecnológicos

EM−4010 Programación Lineal

Proyecto Grupal

Integrantes:

Miguel Castillo Lopez, 2020209596

Jose Pablo Ruiz Badilla, 2020059180

Jose Manuel Sandoval Salazar, 2020097639

Andy Torres Alfaro, 2020171568

Yadir Vega Espinoza, 2016074093

Profesor:

Jeffry Chavarría Molina

II Semestre 2023

Índice

1. Descripción del proyecto

Este proyecto implica la colaboración de los 5 estudiantes del grupo de Programación Lineal del segundo semestre del 2023, en el cuál se va a desarrollar un sistema o programa computacional en Python. El enfoque principal es dividir el trabajo en asignaciones más manejables, que se abordarán semanalmente y se combinarán para crear un único producto final. Cada contribución será documentada en un solo archivo LaTeX, que incluirá el código Python, comentarios detallados y descripciones de parámetros y formatos de datos.

Objetivos del proyecto:

- Desarrollar un programa en Python el cuál pueda generar código LaTeX para la solución de programas lineales generales de manera paso a paso, con su debida documentación de cada función para una mejor comprensión.
- Dividir el trabajo en asignaciones manejables y garantizar la puntualidad en la entrega de tareas.
- Fomentar la colaboración y el esfuerzo en conjunto del equipo.

Contexto del proyecto: El proyecto se desarrolla en un entorno colaborativo, con un equipo de 5 personas trabajando en conjunto. Se enfoca en la programación en Python y la generación de código LaTeX para resolver programas lineales generales. Se trabaja con la subdivisión de tareas y la documentación detallada para procurar el éxito del proyecto dando énfasis en la importancia de la puntualidad. El objetivo es crear un sistema que facilite la generación de código LaTeX para soluciones de programas lineales de manera eficiente.

2. Código implementado

2.1. Tabla2Latex

Para comenzar, se quiere crear una función que tome una matriz de tamaño $M \times N$ y regrese el código LaTeX correspondiente.

```
def Tabla2LaTeX(A):
    m = len(A)      # Calcula el numero de filas de la matriz
    n = len(A[0])   # Calcula el numero de columnas de la matriz

    # Comienza a construir el codigo LaTeX
    latex_code = "\\begin{tabular}{|" + "c" * (n-1) + "|c|}\\hline\n"

    # Itera sobre las filas de la matriz
    for i, fila in enumerate(A):
        # Itera sobre los elementos de cada fila
        for elemento in fila:
            # Agrega el elemento a la tabla LaTeX
            latex_code += str(elemento) + " & "

        # Elimina el ultimo " & " de la fila y agrega una nueva linea
        latex_code = latex_code[:-2] + " \\\\\n"

        # Agrega una linea horizontal antes de la penultima fila
        if i == m - 2:
            latex_code += "\\hline\n"

    # Agrega la linea horizontal final y el cierre de la tabla LaTeX
    latex_code += "\\hline\n\\end{tabular}"

    return latex_code
```

Por ejemplo, esta tabla fue retornada del código anterior, veamos cómo se pudo haber obtenido

1	2	31	4	1	3	1	1
1	2	31	4	1	3	1	2
1	2	31	4	1	3	1	3
1	2	31	4	1	3	1	4
1	2	31	4	1	3	1	5
1	2	31	4	1	3	1	6
1	2	31	4	1	3	1	7
1	2	31	4	1	3	1	z

Sobre el mismo archivo al final se añade el siguiente código:

```
A = [[1,2,31,4,1,3,1,1],
     [1,2,31,4,1,3,1,2],
     [1,2,31,4,1,3,1,3],
     [1,2,31,4,1,3,1,4],
     [1,2,31,4,1,3,1,5],
     [1,2,31,4,1,3,1,6],
     [1,2,31,4,1,3,1,7],
     [1,2,31,4,1,3,1,"z"]]

# Convierte la matriz en una tabla LaTeX
codigo_latex = Tabla2LaTeX(A)

# Imprime el codigo LaTeX resultante
print(codigo_latex)
```

El programa retorna lo siguiente:

```
\begin{tabular}{|ccccccc|c|}\hline
1 & 2 & 31 & 4 & 1 & 3 & 1 & 1  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 2  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 3  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 4  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 5  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 6  \\
1 & 2 & 31 & 4 & 1 & 3 & 1 & 7  \\
\hline
1 & 2 & 31 & 4 & 1 & 3 & 1 & z  \\
\hline
\end{tabular}
```

Notas del código: La función solo recibe una matriz A como una lista de listas y retorna una cadena de texto pensada solo para ser copiada y pegada en LaTeX.

2.2. Permutación, Pivoteo, Escalamiento

Estas funciones llevan a cabo operaciones elementales sobre las filas de una matriz

```
def Escalamiento(matrix, k, i):
    if i<=len(matrix) and k!=0:
        for j in range(len(matrix[i-1])): #Toma cada valor de la fila i
            matrix[i-1][j] = k* matrix[i-1][j] #Multiplica el valor por k
    else:
        print('La entrada de la fila es invalida')
    return matrix # Devuelve la matriz ya con el escalamiento

```

```
def Permutacion(matrix,i,j):
    if (i and j ) <= len(matrix): # Valida las filas ingresadas

        # Guarda los datos de la fila i y cambia los valores de las filas
        auxiliar = matrix[i-1]
        matrix[i-1]=matrix[j-1]
        matrix[j-1]=auxiliar

    return matrix # Devuelve la matriz ya con la permutacion

def Pivoteo(matrix,i,k,j):
    if ((i and j ) <= len(matrix) and k!=0): # Valida los valores
        n = len(matrix[i-1])
        if n==len(matrix[j-1]):
            for elemento_n in range(0,n):

                #Toma el elemento de la fila j, lo multiplica por k y lo
                #suma con el elemento de la fila i
                matrix[i-1][elemento_n]=matrix[i-1][elemento_n]+
                                        k*matrix[j-1][elemento_n]

    return matrix #Devuelve la matriz con el pivoteo
```

En este caso los programas reciben el parámetro *matrix* es una matriz en formato lista de listas, los parámetros i y j son enteros y k es un double. Para ejemplificar su uso se pueden agregar los siguientes códigos al código y correrlo:

```
prueba_escalamiento = Escalamiento(matriz, 100,3)
print(prueba_escalamiento)
prueba_permutacion=Permutacion(matriz, 1, 2)
print(prueba_permutacion)
prueba_pivoteo=Pivoteo(matriz, 1,10,2)
print(prueba_pivoteo)
```

Lo cual imprime los siguientes resultados:

```
# Matriz ejemplo
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [700, 800, 900]]
[[4, 5, 6], [1, 2, 3], [700, 800, 900]]
[[14, 25, 36], [1, 2, 3], [700, 800, 900]]
```

Primero se multiplicó la fila 3 por 100, luego se permutaron la fila 1 y la fila 2 y por último se multiplicó la fila 1 por 10 veces la fila 2

2.3. Tabla Simplex Ampliada

Esta función recibe el sistema de restricciones de un programa de minimización y el vector c correspondiente a las constantes de $z = cx^T$, el programa retorna la tabla simplex estandar canónica y si es necesario devuelve la tabla ampliada con las variables artificiales.

Antes de definir esta función se necesita un paso antes para realizar el procedimiento de pivoteo en las variables artificiales, por lo que se crea la función que se ve a continuación.

```
def pivoteo(tabla):
    num_fil = len(tabla)
    num_col = len(tabla[0])

    # Buscar el penultimo elemento en la ultima fila
    penultimo_elem = tabla[num_fil - 1][num_col - 2]

    # Buscar la fila que tiene un "1" en la misma columna que el
   penultimo elemento
    pivot_row = -1
    for i in range(num_fil - 1):
        if tabla[i][num_col - 2] == 1:
            pivot_row = i
            break

    # Realizar la operacion F_ultima - F_encontrada
    if pivot_row != -1:
        for i in range(num_col):
            tabla[num_fil - 1][i] -= tabla[pivot_row][i]

    return tabla
```

Esta función recibe una tabla y le efectúa las operaciones necesarias para terminar el proceso de convertir la tabla simplex a estandar canónica. Ya con esta función, el código de Tabla Simplex Ampliada es el siguiente

```
def TablaSimplexAmpliada(c, T):

    num_restricciones = len(T)
    num_variables = 0
    num_artificiales = 0
    variables_artificiales = []

    # Contar la cantidad de variables:
    for restriccion in T:
        var_max = len(restriccion)-2
        if var_max > num_variables:
            num_variables = var_max # Toma el valor mas grande

    var_originales=num_variables # Guarda las variables originales

    # Verifica si hay variables de holgura o superfluas
    for restriccion in T:
        if '<=' in restriccion or '>=' in restriccion:
            num_variables += 1

    # Agrega las variables faltantes a las restricciones
    for restriccion in T:
        var_actuales = len(restriccion)-2
        var_faltantes = num_variables - var_actuales

        # Agrega ceros faltantes a la restriccion
        if var_faltantes > 0:
            for i in range(var_faltantes):
                restriccion.insert(var_actuales + i, 0)

        # Agrega 1 o -1 si es variable de holgura o superflua
        if '<=' in restriccion:
            restriccion[var_originales + T.index(restriccion)]=1
        if '>=' in restriccion:
            restriccion[var_originales + T.index(restriccion)]=-1

    # Ahora se va a comprobar si los vectores Identidad estan en T,
    primero se crea la matriz transpuesta, luego se verifica si los
    vectores identidad estan en ella. En caso que no, se agregan
    T_trans=[]
    for i in range(num_variables):
        fila = []
        for restriccion in T:
            fila.append(restriccion[i])
        T_trans.append(fila)

```

```
    # Crear los vectores (1,0,0...), (0,1,0,...)
    vector_Iden=[]
    for i in range(num_restricciones):
        vi=[0]*num_restricciones
        vi[i]=1
        vector_Iden.append(vi)

    # Comprobar si los vectores se encuentran en T_trans:
    for vector in vector_Iden:
        if vector not in T_trans:
            T_trans.append(vector)
            num_artificiales += 1

            # Agregar la variable artificial a la lista
            variables_artificiales.append(f'x{num_variables +
num_artificiales}')

    # Agrega el vector b a T_trans:
    b = []
    for restriccion in T:
        b.append(restriccion[-1])
    T_trans.append(b)

    #transpuesta de T_trans
    T1_trans = [[fila[i] for fila in T_trans] for i in range(len(T_trans
[0]))]

    # Creacion de la tabla simplex estandar canonica

    # Para el caso en que no hayan variables artificiales:
    if num_artificiales == 0:
        a=len(T1_trans[0])-len(c)-1
        for i in range(a):
            c.append(0)
        c.append("z")
        T1_trans.append(c)

        # Devolver la tabla normal
        return T1_trans, 0

    # En caso que si hayan variables artificiales:
    else:
        # Agregar al vector de constantes las variables que no se usan
        a=len(T1_trans[0])-len(c)-1
        for i in range(a):
            c.append(0)
```

```
        c.append("z")
        T1_trans.append(c)

        # Crear el vector de la tabla ampliada
        b=len(T1_trans[0])-num_artificiales-1
        M=[]
        for i in range(b):
            M.append(0)
        for i in range(num_artificiales):
            M.append(1)
        M.append("M")
        T1_trans.append(M)

        # Retornar la tabla ampliada y variables artificiales
        return pivoteo(T1_trans), variables_artificiales
```

Un ejemplo de uso es el siguiente:

```
c = [5, 7]
T = [[7, 2, '>=', 5], [8, 9, '<=', 4], [-2, 5, '=', 6]]
[tabla_ampliada, artificiales] = TablaSimplexAmpliada(c, T)
print(tabla_ampliada)
print(artificiales)
```

El cual imprime lo siguiente:

```
[[7, 2, -1, 0, 1, 0, 5], [8, 9, 0, 1, 0, 0, 4], [-2, 5, 0, 0, 0, 1, 6],
   [5, 7, 0, 0, 0, 0, 'z'], [0, 0, 0, 0, 1, 1, 'M']]
['x5', 'x6']
```

Se debe tomar en consideración que el vector c ingresado debe contener solo constantes, y el vector T debe ser una lista de listas el cuál cada sublista tiene los coeficientes de las primeras n variables no nulas de la restricción y sus últimos 2 elementos son la relación $(<, >, =)$ y el valor del vector b correspondiente, como el ejemplo que se muestra anteriormente.

2.4. Resolver simplex

En esta sección se encuentran 3 funciones, la primera implementa el algoritmo de simplex, la segunda maneja la primera fase del simplex para tratar con variables artificiales y la última es la función principal que resuelve el problema de optimización:

```
def algoritmo_simplex(T):
    m, n = len(T), len(T[0])  # Dimensiones de la matriz.

    # Mientras haya valores negativos en la fila de la funcion objetivo.
    while any(x < 0 for x in T[-1][:-1]):

        # Encuentra la columna con el valor mas negativo.
        columna_pivote = T[-1].index(min(T[-1][:-1]))

        # Si todos son no positivos, no hay solucion finita
        if all(x<=0 for x in [fila[columna_pivote] for fila in T[:-1]]):
            raise ValueError("No hay solucion optima finita.")

        # Calcula las proporciones para determinar la fila pivote.
        proporciones = [fila[-1] / fila[columna_pivote] if fila[
    columna_pivote] > 0 else float('inf') for fila in T[:-1]]

        fila_pivote = proporciones.index(min(proporciones))

        # Normaliza la fila pivote.
        valor_pivote = T[fila_pivote][columna_pivote]
        T[fila_pivote] = [x / valor_pivote for x in T[fila_pivote]]

        # Hace cero los demas valores de la columna pivote.
        for i in range(m):
            if i != fila_pivote:
                factor = T[i][columna_pivote]
                T[i] = [x-factor*y for x, y in zip(T[i], T[fila_pivote])]
    return T
```

Cabe rescatar que las funciones Permutación, Escalamiento y Pivoteo que fueron creadas anteriormente son utilizadas para la implementación de estos códigos. Ahora se crea otra función que realiza la primera fase para programas ampliados. En ese caso, se deberá recibir la tabla ampliada y realizar el trabajo de optimización de la función auxiliar o función de penalización.

```
def fase_inicial(tabla):
    m, n = len(tabla), len(tabla[0])  # Dimensiones de la matriz.
    numero_vars_artificiales = m - 1  # Artificiales necesarias.

    # Crea una fila de penalizacion.
    fila_penalizacion = [-1 if i < n - 1 - numero_vars_artificiales else
   0 for i in range(n)]
    tabla.insert(m - 1, fila_penalizacion)

    # Resuelve el problema de optimizacion con la funcion de penalizacion
    try:
        tabla = algoritmo_simplex(tabla)
    except Exception as e:
        raise ValueError(f"Error en la primera fase del Simplex: {e}")

    # Si el valor en la fila de penalizacion no es 0, el problema
   original no tiene solucion factible.
    if tabla[-2][-1] != 0:
        return None

    # Elimina la fila de penalizacion y las variables artificiales.
    tabla.pop(-2)
    for i in range(m - 1):
        del tabla[i][-2-numero_vars_artificiales:-2]
    del tabla[-1][-2-numero_vars_artificiales:-2]
    return tabla
```

Finalmente, se define la función que resuelve el problema de optimización:

```
def resolver_simplex(tabla):
    try:
        # Resuelve el problema usando el algoritmo de simplex.
        resultado = algoritmo_simplex(tabla)
        return resultado

    # Captura errores durante la ejecucion del algoritmo.
    except ValueError as e:
        return str(e)
```

Un ejemplo de uso podemos observarlo al añadir este código al final del programa:

```
if __name__ == '__main__':
    # Define una matriz con un ejercicio con variables artificiales
    matriz_prueba = [[1, 1, -1, 0, 0, 5],[2, 3, 0, 1, 0, 12],[0, 1, 0, 0, 1, 3],[-2, -3, 0, 0, 0, 0],[0, 0, 0, 0, -1, -3]]
    # Define una matriz prueba sin variables artificiales
    matriz_prueba2=[[2, 1, 1, 0, 18],[2, 3, 0, 1, 42],[-3, -1, 0, 0, 0]]

    # Ejecuta la primera fase del Simplex.
    matriz_transformada = fase_inicial(matriz_prueba)

    # Si la fase inicial fue exitosa y la matriz transformada no es None, ejecuta el algoritmo Simplex.
    if matriz_transformada:
        solucion = resolver_simplex(matriz_transformada)

        # Muestra la solucion.
        if isinstance(solucion, list):
            for fila in solucion:
                print(fila)
        else:
            print(solucion)  # Muestra el mensaje de error si hubo alguno
    else:
        print("El problema no tiene solucion factible.")

    # Para el ejercicio sin variables artificiales
    print('\n\nPara el ejercicio SIN variables artificiales')
    solucion2 = resolver_simplex(matriz_prueba2)
    # Muestra la solucion.
    if isinstance(solucion2, list):
        for fila in solucion2:
            print(fila)
    else:
        print(solucion2)  # Muestra el mensaje de error si hubo alguno.
```

El cual al correr el programa retorna lo siguiente:

```
[0.0, 5.0]
[0.0, 12.0]
[1.0, 3.0]
[0.0, 0.0]
[0.0, 0.0]
Para el ejercicio SIN variables artificiales
[1.0, 0.5, 0.5, 0.0, 9.0]
[0.0, 2.0, -1.0, 1.0, 24.0]
[0.0, 0.5, 1.5, 0.0, 27.0]
```

2.5. Mostrar resultados de Simplex

Durante el desarrollo del algoritmo de esta semana se generó un problema principal y corresponde a la utilidad del código, y otros diversos errores, así que se decidió crear otra versión del Algoritmo Simplex.

```
def AlgoritmoSimplex(Matriz, EsAmpliado):
    NFilas, NColumnas = len(Matriz), len(Matriz[0])
    Ejecutar = True
    while(Ejecutar):
        #Si todos los coeficientes de la funcion objetivo son no negativos, se finaliza automaticamente
        if min(Matriz[NFilas-1][0:NColumnas-1])>=0:
            Ejecutar = False

        if Ejecutar:
            #Encontrar columna pivote, recorre la funcion objetivo y devuelve el indice del menor elemento
            columna_pivote = 0
            for i in range(0, NColumnas-1):
                if (Matriz[NFilas-1][i] < Matriz[NFilas-1][columna_pivote]):
                    columna_pivote = i
            #Encontrar Pivote

            #Recordar que la ultima fila puede variar si es ampliado o no
            if EsAmpliado:
                CantidadFilasSimplex=NFilas-2
            else:
                CantidadFilasSimplex = NFilas - 1

            VectorCocientes = [-float('inf')]*CantidadFilasSimplex
            for i in range(CantidadFilasSimplex):
                if Matriz[i][columna_pivote]!=0:
                    VectorCocientes[i] = Matriz[i][NColumnas-1]/Matriz[i][columna_pivote]
            Cocientes_Postivos = [numero for numero in VectorCocientes if numero >= 0]
            if Cocientes_Postivos:
                # Encontrar el indice del menor numero positivo
                fila_pivote = VectorCocientes.index(min(Cocientes_Postivos))
            else:
                Ejecutar = False
                T = []

        if Ejecutar:
```

```
            #Proceso de pivote
            ElementoPivote = Matriz[fila_pivote][columna_pivote]
            #Primer paso pivote
            Matriz = Escalamiento(Matriz, 1/ElementoPivote, fila_pivote
    +1)
            #Segundo paso pivoteo
            for i in range(0, NFilas):
                if i != fila_pivote:
                    Valor_k = -Matriz[i][columna_pivote]
                    Matriz = Pivoteo(Matriz, i+1, Valor_k, fila_pivote+1)
    return Matriz
```

Básicamente el algoritmo anterior se encarga de realizar ambas versiones del método Simplex (con y sin variables artificiales), es decir, recibe la matriz inicial del método Simplex y si tiene variables articiales.

En caso de tener variables artificiales no se consideran las dos últimas filas de la matriz para la selección de la fila pivote, en caso de no tener variables artificiales, solo no se toma en cuenta la última linea.

Pero el procedimiento básicamente, para ambas versiones del Simplex pero se le debe dar la información de si es ampliado o no.

Por otro lado, se creó para una función para devolver el vector solución de una tabla dada:

```
def obtenerVector(T):
    #Obtiene numero de filas y columnas de la tabla
    NFilas, NColumnas = len(T), len(T[0])
    #Vector de ceros
    VectorSolucion = [0]*(NColumnas-1)
    #Llena los valores si los valores corresponden a a variables no
    basicas.
    for i in range(NFilas-1):
        for j in range(NColumnas-1):
            if T[i][j] == 1 and T[NFilas-1][j]==0:
                VectorSolucion[j] = T[i][NColumnas-1]
    return VectorSolucion
```

2.5.1. Estado 1

Se generó el siguiente código que permite detectar sin un programa es ampliado y en caso de serlo y tener solución, que realiza el proceso de poda, esta función es la base de la resolución del **estado 1:**

```
"""Estado 1 sera que el programa reciba un problema lineal y retorne unicamente
el resultado, la solucion optima o el mensaje de fracaso."""
def IniciarProgramaEstado1(c,T,z0):
    Tabla, TieneArtificiales, Artificiales = TablaSimplexAmpliada(c, T,z0)
    #print(TieneArtificiales)
    #print(Tabla)
    if TieneArtificiales:
        #Cuales son las variables artificiales
        VectorArtificiales =[]
        for _ in Artificiales:
            VectorArtificiales.append(int(_))
        VectorArtificiales.sort(reverse=True)

        #Simplex para prgramas aumentados
        AlgoritmoSimplex(Tabla, True)
        print(Tabla)
        if Tabla:
            #Commpara que el programa se encuentre en forma estandar y canonica para devolver el vector solucion

            SolucionProgramaAmplicado = obtenerVector(Tabla)
            Validacion = True
            for IndiceArtificial in VectorArtificiales:
                #Valida que el vector del sistema ampliado tenga las variables artificiales iguales a 0
                if SolucionProgramaAmplicado[(IndiceArtificial)-1] != 0:
                    Validacion = False
            if Validacion:

                #Proceso de poda
                del(Tabla[-1])
                for i in range(len(Tabla)):
                    for IndiceArtificial in VectorArtificiales:
                        del(Tabla[i][IndiceArtificial-1])
                #print(Tabla)
            else:
                Tabla = []
        else:
            pass
```

```
38      Tabla = AlgoritmoSimplex(Tabla,False)
39      return obtenerVector(Tabla),Tabla[-1][-1]
```

El anterior programa sirve para resolver cualquier programa lineal en caso de tener solución, se pueden modificar las dos funciones anteriores para recoger la información que se desea recoger.

2.5.2. Estado 2

Lo cual, la modificación del estado 1, permite obtener un estado 2 que es básicamente obtener la tabla inicial, intermedia y final, y la solución

```
"""Estado 2 sera que el programa reciba un problema lineal, muestre la tabla
simplex inicial y la tabla simplex final junto con la solucion o el mensaje
de fracaso. En caso de requerir variables artificiales, debera mostrar la
tabla simplex intermedia."""
def IniciarProgramaEstado2(c,T,z0):
    Tabla, TieneArtificiales, Artificiales = TablaSimplexAmpliada(c, T,z0)
    lista = [TieneArtificiales]
    lista.append(copy.deepcopy(Tabla))
    if TieneArtificiales:
        #Cuales son las variables artificiales
        VectorArtificiales =[]
        for _ in Artificiales:
            VectorArtificiales.append(int(_))
        VectorArtificiales.sort(reverse=True)
        #Simplex para sistemas aumentados
        AlgoritmoSimplex(Tabla, True)
        lista.append(copy.deepcopy(Tabla))
        if Tabla:
            #Commpara que cada solucion de las variables artificiales del programa ampliado sea 0, es decir que converja
            SolucionProgramaAmplicado = obtenerVector(Tabla)
            Validacion = True
            for IndiceArtificial in VectorArtificiales:
                if SolucionProgramaAmplicado[(IndiceArtificial)-1] != 0:
                    Validacion = False
            if Validacion:
                #Proceso de poda
                del(Tabla[-1])
                for i in range(len(Tabla)):
                    for IndiceArtificial in VectorArtificiales:
                        del(Tabla[i][IndiceArtificial-1])
                #print(Tabla)
                #Captura de tabla intermmedia
                lista.append(copy.deepcopy(Tabla))
            else:
                #print('Programa lineal ampliado no paga los pesos')
                Tabla = []
        else:
            #print('Programa sin solucion')
```

```
39              pass
40      #Simplex
41      Tabla = AlgoritmoSimplex(Tabla,False)
42      if Tabla:
43          #Obtener solucion y la captura
44          lista.append(Tabla)
45          Sol = obtenerVector(Tabla)
46          lista.append(Sol)
47          pass
48      else:
49
50      return lista
```

2.5.3. Estado 3

Se generó otra versión de Simplex que muestre la información capturada en cada paso, de esta manera se tiene la siguiente función, que permite capturar toda la información del proceso de Simplex:

```
def AlgoritmoSimplexPasoAPaso(Matriz, EsAmpliado):
    NFilas, NColumnas = len(Matriz), len(Matriz[0])
    Ejecutar = True

    #Primera iteracion
    print('Primera iteracion ')
    Matrices = []
    Matrices.append(copy.deepcopy(Matriz))
    print(Matrices)
    input()


    while(Ejecutar):
        #Si todos los coeficientes de la funcion objetivo son no negativos, se finaliza automaticamente
        if min(Matriz[NFilas-1][0:NColumnas-1])>=0:
            print('Se encontro que todos los coeficientes de la funcion objetivo son no negativos, fin del programa')
            input()
            Ejecutar = False

        if Ejecutar:
            #Encontrar columna pivote, recorre la funcion objetivo y devuelve el indice del menor elemento
            columna_pivote = 0
            for i in range(0, NColumnas-1):
                if (Matriz[NFilas-1][i] < Matriz[NFilas-1][columna_pivote]):
                    columna_pivote = i
            print('Columna pivote:', columna_pivote+1, ' dado que ', Matriz[NFilas-1][columna_pivote], ' es el menor coeficiente negativo de la funcion objetivo')
            #Encontrar Pivote

            #Recordar que la ultima fila puede variar si es ampliado o no
            if EsAmpliado:
                CantidadFilasSimplex=NFilas-2
            else:
                CantidadFilasSimplex = NFilas - 1

            VectorCocientes = [-float('inf')]*CantidadFilasSimplex
```

```
            for i in range(CantidadFilasSimplex):
                if Matriz[i][columna_pivote]!=0:
                    VectorCocientes[i] = Matriz[i][NColumnas-1]/Matriz[i
    ][columna_pivote]
            Cocientes_Postivos = [numero for numero in VectorCocientes if
     numero >= 0]
            if Cocientes_Postivos:
                # Encontrar el indice del menor numero positivo
                fila_pivote = VectorCocientes.index(min(
    Cocientes_Postivos))
                print('Fila pivote:', fila_pivote+1, ' dado que ', Matriz
    [fila_pivote][NColumnas-1], '/', Matriz[fila_pivote][columna_pivote]
    ,' es el menor cociente no negativo')
                print('De esta manera el elemento pivote ')
                input()
            else:
                Ejecutar = False
                T = []
        if Ejecutar:
            print('Proceso de pivoteo:')
            input()
            #Proceso de pivote
            print('Elemento pivote se encuentra en la posicion',
    fila_pivote+1, ',', columna_pivote+1)

            ElementoPivote = Matriz[fila_pivote][columna_pivote]

            print('Elemento pivote corresponde a', ElementoPivote)
            input()
            #Primer paso pivote
            Matriz = Escalamiento(Matriz, 1/ElementoPivote, fila_pivote
    +1)
            print('Se multiplica fila ', fila_pivote+1, ' por ', 1/
    ElementoPivote)
            input()
            #Segundo paso pivoteo
            for i in range(0, NFilas):
                if i != fila_pivote:
                    Valor_k = -Matriz[i][columna_pivote]
                    print('A la fila ', i+1, ' se le suma ', Valor_k, '
    veces la fila ', fila_pivote+1)
                    print('El resultado del proceso es: ')
                    Matriz = Pivoteo(Matriz, i+1, Valor_k, fila_pivote+1)
                    print(Matriz)
                    input()
            Matrices.append(copy.deepcopy(Matriz))
```

```
73      return Matriz
```

2.6. Solución método simplex paso a paso

A continuación se brindarán 2 ejemplos de la resolución de un ejercicio de programación lineal, uno con variables artificiales y otro sin.

2.6.1. Resolución sin variables artificiales

$$\text{Minimizar: } z = 3x_1 - 2x_2$$
$$\text{Sujeta a las restricciones:}$$

$$\begin{cases} 2x_1 + x_2 & \leq & 18 \\ 2x_1 + 3x_2 & \leq & 42 \\ 3x_1 - 2x_2 & \leq & 5 \\ x_1, x_2 & \geq & 0 \end{cases}$$

Primero, vamos a agregar a las restricciones las variables de holgura faltantes.

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 18 \\ 2x_1 + 3x_2 + x_4 & = & 42 \\ 3x_1 - 2x_2 + x_5 & = & 5 \\ x_1, x_2, x_3, x_4, x_5 & \geq & 0 \end{cases}$$

De esta forma, el programa final sería el siguiente:

$$\text{Minimizar: } z = 3x_1 - 2x_2 + 0x_3 + 0x_4 + 0x_5$$
$$\text{Sujeta a las restricciones:}$$

$$\begin{cases} 2x_1 + x_2 + x_3 + 0x_4 + 0x_5 & = & 18 \\ 2x_1 + 3x_2 + 0x_3 + x_4 + 0x_5 & = & 42 \\ 3x_1 - 2x_2 + 0x_3 + 0x_4 + x_5 & = & 5 \\ x_1, x_2, x_3, x_4, x_5 & \geq & 0 \end{cases}$$

Con esto, podemos construir la tabla simplex del programa:

2	1	1	0	0	18
2	3	0	1	0	42
3	-2	0	0	1	5
3	-2	0	0	0	z

Note que $c_2 = -2 < 0$, por lo que podemos mejorar la solución.

Se tomará el elemento $a_{22} = 3$ como pivote para convertir la columna 2 en el vector canónico e_2.
Donde aplicando operaciones elementales sobre la tabla simplex se obtiene:

$1.\overline{3}$	0	1	$-0.\overline{3}$	0	4
$0.\overline{6}$	1	0	$0.\overline{3}$	0	14
$4.\overline{3}$	0	0	$0.\overline{6}$	1	33
$4.\overline{3}$	0	0	$0.\overline{6}$	0	$z + 28$

De esta última tabla se puede concluir que la solución del programa corresponde a: $x_1 = 0, x_2 = 14, x_3 = 4, x_4 = 0, x_5 = 33$

2.6.2. Resolución con variables artificiales

Minimizar: $z = -2x_1 - x_2$
Sujeta a las restricciones:

$$\begin{cases} x_1 - x_2 & \leq & 4 \\ 2x_1 + x_2 & \geq & 3 \\ -x_1 + 8x_2 & \leq & 24 \\ x_1, x_2 & \geq & 0 \end{cases}$$

Primero, añadimos las variables de holgura y superfluas faltantes a las restricciones:

$$\begin{cases} x_1 - x_2 + x_3 & = & 4 \\ 2x_1 + x_2 - x_4 & = & 3 \\ -x_1 + 8x_2 + x_5 & = & 24 \\ x_1, x_2, x_3, x_4, x_5 & \geq & 0 \end{cases}$$

Notamos que falta el vector $(0, 1, 0)^T$ en el programa, entonces añadimos la variable artificial x_6, por lo que el programa final será:

Minimizar: $z = -2x_1 - x_2 + 0x_3 + 0x_4 + 0x_5 + Mx_6$
Sujeta a las restricciones:

$$\begin{cases} x_1 - x_2 + x_3 + 0x_4 + 0x_5 + 0x_6 & = & 4 \\ 2x_1 + x_2 + 0x_3 - x_4 + 0x_5 + x_6 & = & 3 \\ -x_1 + 8x_2 + 0x_3 + 0x_4 + x_5 + 0x_6 & = & 24 \\ x_1, x_2, x_3, x_4, x_5, x_6 & \geq & 0 \end{cases}$$

Por lo que a partir de esto podemos construir la tabla simplex ampliada:

1	-1	1	0	0	0	4
2	1	0	-1	0	1	3
-1	8	0	0	1	0	24
-2	-1	0	0	0	0	z
0	0	0	0	0	1	w

Antes de seguir, se deben realizar operaciones elementales para conseguir ceros en las columnas de los vectores identidad.

En este caso aplicamos las operaciones $F_5 - F_2$:

1	-1	1	0	0	0	4
2	1	0	-1	0	1	3
-1	8	0	0	1	0	24
-2	-1	0	0	0	0	z
-2	-1	0	1	0	-1	w-3

Aplicando el método simplex sobre la función de penalización.

0.0	-1.5	1.0	0.5	0.0	-0.5	2.5
1.0	0.5	0.0	-0.5	0.0	0.5	1.5
0.0	8.5	0.0	-0.5	1.0	0.5	25.5
0.0	0.0	0.0	-1.0	0.0	1.0	3.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0

Como las variables artificiales dejaron de ser básicas, el algoritmo puede continuar, se elimina la última columna de la tabla simplex así como las filas de todas las variables artificiales para obtener:

0.0	-1.5	1.0	0.5	0.0	2.5
1.0	0.5	0.0	-0.5	0.0	1.5
0.0	8.5	0.0	-0.5	1.0	25.5
0.0	0.0	0.0	-1.0	0.0	3.0

Seguidamente, se aplica el algoritmo simplex para obtener:

0.0	0.0	2.4285714285714284	1.0	0.42857142857142855	17.0
1.0	0.0	1.1428571428571428	0.0	0.14285714285714285	8.0
0.0	1.0	0.14285714285714285	0.0	0.14285714285714285	4.0
0.0	0.0	2.4285714285714284	0.0	0.42857142857142855	20.0

De esto se puede concluir que la solución del programa es: $x_1 = 8.0$, $x_2 = 4.0$, $x_3 = 0$, $x_4 = 17.0$, $x_5 = 0$

2.7. Simplex2Latex

Seguidamente, tomando como referencia el estado 3 de la sección anterior se creó el siguiente código que realiza el procedimiento del simplex paso a paso y devuelve el código LaTeX.

Primero, se definieron varias funciones auxiliares que permitirán que la función creada llamada Simplex2Latex trabaje de manera correcta. La primera corresponde a la función Tabla2LaTeX. La cual recibe la matriz que contiene los coeficientes asociados a las restricciones del programa lineal, los coeficientes de la función objetivo y el vector b y lo convierte en una tabla simplex en código LaTex. Básicamente, si se tiene un problema sin variables artificiales creará la tabla normal de simplex, en caso contrario si se tienen variables artificiales entonces creará una tabla simplex ampliada en código LaTeX. El código es el siguiente:

```
def Tabla2LaTeX(A, Ampl):
    m = len(A)      # Calcula el numero de filas de la matriz
    n = len(A[0])   # Calcula el numero de columnas de la matriz

    # Comienza a construir el codigo LaTeX
    latex_code = "\\begin{tabular}{|" + "c" * (n-1) + "|c|}\\hline\n"

    # Itera sobre las filas de la matriz
    for i, fila in enumerate(A):
        # Itera sobre los elementos de cada fila
        for elemento in fila:
            # Agrega el elemento a la tabla LaTeX
            latex_code += str(elemento) + " & "

        # Elimina el ultimo " & " de la fila y agrega una nueva linea
        latex_code = latex_code[:-2] + " \\\\\n"

        # Agrega una linea horizontal antes de la penultima fila
        if Ampl and i == m - 3:
            latex_code += "\\hline\n"
        if not Ampl and i == m - 2:
            latex_code += "\\hline\n"

    # Agrega la linea horizontal final y el cierre de la tabla LaTeX
    latex_code += "\\hline\n\\end{tabular}"

    return latex_code
```

Seguidamente, se definió la función LatexSimplexPasoAPaso, la cual tiene el objetivo de ir explicando paso a paso el proceso de resolución del programa utilizando el método

simplex. Lo primero que realiza la función es calcular el número de filas y columnas de la matriz en formato simplex y se crea una variable llamada Ejecutar que tendrá como valor True y se ejecutará hasta que se le de la instrucción contraria.

Lo primero que se revisa es que todas las entradas del vector de coeficientes sean positivas, si es el caso se le da la instrucción False a la variable Ejecutar y se termina el proceso. Si hay alguna entrada negativa se empieza el proceso de pivoteo. Primero, se busca la columna pivote, que corresponde a la columna que contenga al coeficiente de menor número. Luego se crea una variable llamada CantidadFilasSimplex que hace que si se tiene un problema con tabla simplex aumentada la cantidad de filas se reduzca en dos y si se tiene una tabla simplex normal se reduzca en una, esto se hace dado que el pivoteo solo toma en cuenta los coeficientes asociados a las restricciones del programa.

Seguidamente, se hace un vector de cocientes que irá añadiendo el cociente entre el vector b y cada una de las entradas de la columna pivote. Luego, se creará otro vector que almacene solamente las entradas positivas del vector de cocientes y tomará la menor como el elemento pivote.

A partir de esto, el elemento encontrado como la menor entrada positiva se convierte en 1 y los demás elementos de la columna se convierten en 0. Este proceso se repetirá hasta que no se tengan entradas negativas en el vector de coeficientes. El código que se utilizó para realizar la tarea fue el siguiente:

```
def LatexSimplexPasoAPaso(Matriz, EsAmpliado):
    NFilas, NColumnas = len(Matriz), len(Matriz[0])
    Matrices = []
    Matrices.append(copy.deepcopy(Matriz))
    Ejecutar = True

    #Primera iteracion
    print('Se comienza a resolver usando el metodo simplex. \\\ ')


    while(Ejecutar):
        #Si todos los coeficientes de la funcion objetivo son no
   negativos, se finaliza automaticamente
        if min(Matriz[NFilas-1][0:NColumnas-1])>=0:
            print('Como los coeficientes de la funcion objetivo son no
   negativos, se llega al fin del programa.')
            Ejecutar = False

        if Ejecutar:
            #Encontrar columna pivote, recorre la funcion objetivo y
   devuelve el indice del menor elemento
```

```
19            columna_pivote = 0
20            for i in range(0, NColumnas-1):
21                if (Matriz[NFilas-1][i] < Matriz[NFilas-1][columna_pivote
    ]):
22                    columna_pivote = i
23            print("\\textbf{Proceso de pivoteo}")
24            print("\\begin{itemize}")
25            print('\item La columna pivote es', columna_pivote+1, 'dado
    que', Matriz[NFilas-1][columna_pivote], ' es el menor coeficiente
    negativo de la funcion objetivo.')
26            #Encontrar Pivote
27
28            #Recordar que la ultima fila puede variar si es ampliado o no
29            if EsAmpliado:
30                CantidadFilasSimplex=NFilas-2
31            else:
32                CantidadFilasSimplex = NFilas - 1
33
34            VectorCocientes = [-float('inf')]*CantidadFilasSimplex
35            for i in range(CantidadFilasSimplex):
36                if Matriz[i][columna_pivote]!=0:
37                    VectorCocientes[i] = Matriz[i][NColumnas-1]/Matriz[i
    ][columna_pivote]
38            Cocientes_Postivos = [numero for numero in VectorCocientes if
     numero >= 0]
39            if Cocientes_Postivos:
40                # Encontrar el indice del menor numero positivo
41                fila_pivote = VectorCocientes.index(min(
    Cocientes_Postivos))
42                print('\item La fila pivote es', fila_pivote+1, 'dado que
    ', str(Matriz[fila_pivote][NColumnas-1])+'/'+ str(Matriz[fila_pivote
    ][columna_pivote]) ,'es el menor cociente no negativo.')
43
44            else:
45                Ejecutar = False
46                T = []
47
48        if Ejecutar:
49            print('\item Ahora se aplica el proceso de pivoteo con el
    elemento pivote en la posicion ['+str(fila_pivote+1)+', '+str(
    columna_pivote+1)+ "].")
50
51            ElementoPivote = Matriz[fila_pivote][columna_pivote]
52
53            print('\item El elemento pivote corresponde a',
    ElementoPivote)
```

```
            print("\end{itemize}")
            #Primer paso pivote
            Matriz = Escalamiento(Matriz, 1/ElementoPivote, fila_pivote
    +1)
            print('Se multiplica la fila ', fila_pivote+1, 'por', 1/
    ElementoPivote, " y se convierte en 0 los demas valores de la columna
     pivote en 0: ")

            #Segundo paso pivoteo
            for i in range(0, NFilas):
                if i != fila_pivote:
                    Valor_k = -Matriz[i][columna_pivote]
                    Matriz = Pivoteo(Matriz, i+1, Valor_k, fila_pivote+1)
            print("\\begin{center}")
            print(Tabla2LaTeX(Matriz, EsAmpliado))
            print("\end{center}")
    return Matriz
```

Luego, tenemos otra función llamada obtenerVector la cual recibirá el vector de restricciones y a partir de este se obtendrá el vector de soluciones del programa. Primero, se genera un for para buscar en que posición se tiene un 1 y a la vez se tienen ceros en las demás entradas del vector.

Al encontrar esto, se agrega en el vector de soluciones el número correspondiente a la entrada en el vector de restricciones que se encuentre en la misma fila del 1 y en la última columna.

Cabe recalcar que la posición de cada elemento que se agrega al vector de soluciones depende de la columna en la que se encuentre el 1 a partir del for. Luego, se repite el proceso hasta encontrar todas las soluciones. El código correspondiente es:

```
def obtenerVector(T):
    NFilas, NColumnas = len(T), len(T[0])
    VectorSolucion = [0]*(NColumnas-1)
    for i in range(NFilas-1):
        for j in range(NColumnas-1):
            if T[i][j] == 1 and T[NFilas-1][j]==0:
                VectorSolucion[j] = T[i][NColumnas-1]
    return VectorSolucion
```

La siguiente función se llama Coeficiente2Funcion, la cual tiene la funcionalidad de tomar el vector de coeficientes y el vector de restricciones y convertirlos en un criterio de función con las variables correspondientes. El código es el siguiente:

```python
def Coeficiente2Funcion(c, verCeros):
    #Esta funcion crea el string c_1x_1+c_2x_2...
    f=""
    numvariable=0 # Contador de variables
    for coeficiente in c:
        if isinstance(coeficiente, str):
            return f
        else:
            numvariable+=1
            # Verifica si quiere que se vean los 0x_i
            if verCeros:
                # Si el coeficiente es negativo o es la primera variable
                # no hace falta agregar el +
                if coeficiente<0 or numvariable==1:
                    if coeficiente==-1: #Hacer que el -1 no se vea
                        f+=f"-x_{numvariable}"
                    elif coeficiente==1: #Hacer que el 1 no se vea
                        f+=f"x_{numvariable}"
                    else:
                        f+=str(coeficiente)+f"x_{numvariable}"

                #Si el coeficiente es 0 no se agrega a la funcion
                else:
                    if coeficiente==1: #Hacer que el 1 no se vea
                        f+=f"+x_{numvariable}"
                    else:
                        f+="+"+str(coeficiente)+f"x_{numvariable}"
            else:
                if coeficiente<0 or numvariable==1:
                    if coeficiente==-1: #Hacer que el -1 no se vea
                        f+=f"-x_{numvariable}"
                    elif coeficiente==1: #Hacer que el 1 no se vea
                        f+=f"x_{numvariable}"
                    else:
                        f+=str(coeficiente)+f"x_{numvariable}"
                elif coeficiente>0:
                    if coeficiente==1: #Hacer que el 1 no se vea
                        f+=f"+x_{numvariable}"
                    else:
                        f+="+"+str(coeficiente)+f"x_{numvariable}"
    return f
```

La última función auxiliar se llama minimizarSujetoA, la cual crea el encabezado del programa lineal donde se tiene el minimizar la función sujeto a las restricciones en código LaTeX. A partir de la función Coeficiente2Funcion se crea la función objetivo y cada

uno de los criterios de las restricciones, luego de eso se crea el código LaTeX correspondiente a la forma usual de un programa lineal. El código se muestra a continuación:

```
def minimizarSujetoA(c,T,z0, canonica, soloT):
    #Hace el print de minizar z sujeto a T

    #Verifica si se tiene z0
    if z0==0:
        z="$z="+Coeficiente2Funcion(c, canonica)+"$"
    else:
        z="$z="+Coeficiente2Funcion(c, canonica)+str(z0)+"$"


    #Inicia el vector de restricciones:
    r="$$\left\{ \n \\begin{array}{rcl}\n"
    for restriccion in T:
        if not canonica:
            if "<=" in restriccion:
                igualdad="\leq"
            if ">=" in restriccion:
                igualdad="\geq"
            if "=" in restriccion:
                igualdad="="
        else:
            igualdad="="
        b=str(restriccion[-1]) # Toma el ultimo valor de la restriccion
        r+=Coeficiente2Funcion(restriccion, canonica)+" & "+igualdad+" &
   "+b+"\\\ "+"\n"
    r+="\end{array} \n\\right.$$"

    #Print del problema inicial
    if soloT: #Si solo se quiere ver el vector de restricciones
        print(f"{r}")
    else:
        print("\\begin{center}")
        print(f"Minimizar {z} \\\ \nSujeto a las restricciones:")
        print("\end{center}")
        print(f"{r}")
```

Por último, se tiene la función Simplex2LaTeX, que recibe el vector de coeficientes, el vector de restricciones y la constante perteneciente a la función objetivo en el caso que la tenga.

Lo que realiza esta función es devolver en código LaTeX la explicación del procedimiento de solución de un programa lineal dado. Primero utiliza la función minimizarSujetoA para colocar el encabezado del programa lineal. Seguidamente, coloca un texto explica-

tivo en el que se muestra el proceso de adición de variables de holgura y superfluas en el caso de que el programa lo necesite. Si añadiendo estas variables aún no se tienen los vectores canónicos, entonces se explica que se añadieron variables artificiales y se realiza la explicación de la solución (si tiene) del programa utilizando las funciones LatexSimplexPasoAPaso, obtenerVector y Tabla2LaTeX. El código es el siguiente:

```
def Simplex2Latex(c,T,z0):

    print("Resolucion del siguiente programa lineal:")
    minimizarSujetoA(c,T,z0, False, False)

    # Texto explicativo de variables de holgura y superfluas
    txt1=""
    cont=0
    for restriccion in T:
        if "<=" in restriccion:
            cont+=1
            txt1+=f"La restriccion {cont} es de $\leq$, se agrega la variable de holgura $x_{cont+len(c)}$. "
        if ">=" in restriccion:
            cont+=1
            txt1+=f"La restriccion {cont} es de $\geq$, se agrega la variable superflua $x_{cont+len(c)}$. "
    print(txt1)

    Tabla, TieneArtificiales, Artificiales = TablaSimplexAmpliada(c, T,z0)

    print("Al agregar las variables de holgura y superfluas a la restriccion se obtiene:")
    minimizarSujetoA(c,T,z0, True, True)

    #Verificar si es un ejercicio con artificiales o no
    if TieneArtificiales:
        print("Note que todavia no se tienen todas las columnas del vector identidad, se deben agregar variables artificiales \\\ ")
        print("Al agregar las variables artificiales se crea la siguiente tabla simplex apliada:")
        print("\\begin{center} \n" +Tabla2LaTeX(Tabla, True)+"\n\end{center}")

        VectorArtificiales =[]
        for _ in Artificiales:
            VectorArtificiales.append(int(_))
        VectorArtificiales.sort(reverse=True)
        #Simplex para sistemas aumentados
```

```
        LatexSimplexPasoAPaso(Tabla, True)
        if Tabla:
            #Commpara que cada
            SolucionProgramaAmplicado = obtenerVector(Tabla)
            Validacion = True
            for IndiceArtificial in VectorArtificiales:
                if SolucionProgramaAmplicado[(IndiceArtificial)-1] != 0:
                    Validacion = False
            if Validacion:
                print('Note que ampliado converge y como las variables artificiales son cero la penalizacion es nula. \\')
                #Proceso de poda, corregir
                print('Se elimina la ultima fila de la tabla y las columnas de las variables artificiales')
                del(Tabla[-1])

                for i in range(len(Tabla)):
                    for IndiceArtificial in VectorArtificiales:
                        del(Tabla[i][IndiceArtificial-1])
                print("\\begin{center}")
                print(Tabla2LaTeX(Tabla, False))
                print("\end{center}")
                print("Ahora se vuelve a realizar el metodo simplex con esta nueva tabla.")
            else:
                print('Note que el programa lineal ampliado converge pero todavia tiene variables artificiales, por lo que no tiene solucion')
                Tabla = []
        else:
            print('El programa no tiene solucion')

    print("Se crea la tabla simplex incial:")
    print("\\begin{center} \n" +Tabla2LaTeX(Tabla, False)+"\n\end{center}")
    Tabla = LatexSimplexPasoAPaso(Tabla,False)
    if Tabla:
        print('La solucion del programa lineal es:')
        print(obtenerVector(Tabla))
    else:
        print('Programa lineal sin solucion')
    return Tabla
```

Al ejecutar la función Simplex2Latex con los siguientes datos:

```
c=[2,-3]
T=[[-1,2,1,0,0,'=',8],[1,1,0,1,0,'=',10],[3,-2,0,0,1,'=',15]]
z0=0
```

Se obtiene lo siguiente:

```
Resolución del siguiente programa lineal:
\begin{center}
Minimizar $z=2x_1-3x_2$ \\
Sujeto a las restricciones:
\end{center}
$$\left\{
 \begin{array}{rcl}
-x_1+2x_2+x_3 & = & 8\\
x_1+x_2+x_4 & = & 10\\
3x_1-2x_2+x_5 & = & 15\\
\end{array}
\right.$$

Al añadir las variables de holgura y superfluas a la restricción se obtiene:
$$\left\{
 \begin{array}{rcl}
-x_1+2x_2+x_3+0x_4+0x_5 & = & 8\\
x_1+x_2+0x_3+x_4+0x_5 & = & 10\\
3x_1-2x_2+0x_3+0x_4+x_5 & = & 15\\
\end{array}
\right.$$
Se crea la tabla simplex incial:
\begin{center}
\begin{tabular}{|ccccc|c|}\hline
-1 & 2 & 1 & 0 & 0 & 8  \\
1 & 1 & 0 & 1 & 0 & 10  \\
3 & -2 & 0 & 0 & 1 & 15  \\
\hline
2 & -3 & 0 & 0 & 0 & 0  \\
\hline
\end{tabular}
\end{center}
Se comienza a resolver usando el método simplex. \\
\textbf{Proceso de pivoteo}
\begin{itemize}
\item La columna pivote es 2 dado que -3  es el menor coeficiente negativo de la función objetivo.
\item La fila pivote es 1 dado que 8/2 es el menor cociente no negativo.
\item Ahora se aplica el proceso de pivoteo con el elemento pivote en la posición [1, 2].
\item El elemento pivote corresponde a 2
\end{itemize}
Se multiplica la fila  1 por 0.5  y se convierte en 0 los demás valores de la columna pivote en 0:
\begin{center}
\begin{tabular}{|ccccc|c|}\hline
-0.5 & 1.0 & 0.5 & 0.0 & 0.0 & 4.0  \\
1.5 & 0.0 & -0.5 & 1.0 & 0.0 & 6.0  \\
2.0 & 0.0 & 1.0 & 0.0 & 1.0 & 23.0  \\
\hline
0.5 & 0.0 & 1.5 & 0.0 & 0.0 & 12.0  \\
\hline
\end{tabular}
\end{center}
Como los coeficientes de la función objetivo son no negativos, se llega al fin del programa.
La solucion del programa lineal es:
[0, 4.0, 0, 6.0, 23.0]
```

Y al copiar ese código y pegarlo en LaTeX se genera lo siguiente:
Resolución del siguiente programa lineal:

$$\text{Minimizar } z = 2x_1 - 3x_2$$
$$\text{Sujeto a las restricciones:}$$

$$\begin{cases} -x_1 + 2x_2 + x_3 &= 8 \\ x_1 + x_2 + x_4 &= 10 \\ 3x_1 - 2x_2 + x_5 &= 15 \end{cases}$$

Al añadir las variables de holgura y superfluas a la restricción se obtiene:

$$\begin{cases} -x_1 + 2x_2 + x_3 + 0x_4 + 0x_5 &= 8 \\ x_1 + x_2 + 0x_3 + x_4 + 0x_5 &= 10 \\ 3x_1 - 2x_2 + 0x_3 + 0x_4 + x_5 &= 15 \end{cases}$$

Se crea la tabla simplex incial:

-1	2	1	0	0	8
1	1	0	1	0	10
3	-2	0	0	1	15
2	-3	0	0	0	0

Se comienza a resolver usando el método simplex.
Proceso de pivoteo

- La columna pivote es 2 dado que -3 es el menor coeficiente negativo de la función objetivo.
- La fila pivote es 1 dado que 8/2 es el menor cociente no negativo.
- Ahora se aplica el proceso de pivoteo con el elemento pivote en la posición [1, 2].
- El elemento pivote corresponde a 2

Se multiplica la fila 1 por 0.5 y se convierte en 0 los demás valores de la columna pivote en 0:

-0.5	1.0	0.5	0.0	0.0	4.0
1.5	0.0	-0.5	1.0	0.0	6.0
2.0	0.0	1.0	0.0	1.0	23.0
0.5	0.0	1.5	0.0	0.0	12.0

Como los coeficientes de la función objetivo son no negativos, se llega al fin del programa. La solucion del programa lineal es: [0, 4.0, 0, 6.0, 23.0]

2.8. SimplexEntero

Ahora, se utiliza todo el código anterior para implementar la parte de programación entera, para esto primero se define una función aparte que lleva el conteo del número de nodo:

```
def contadorNodo():
    global nodoContador
    nodoContador+=1
```

El valor inicial de la variable `nodoContador` es 1 ya que el primer nodo es dado automáticamente por el programa y esta variable al ser aumentada antes de hacer la división de los nodos si comenzara en cero el primer hijo del nodo 1 también sería nodo 1.

Seguidamente se crea la función `simplexEntero()` la cuál recibe los siguientes parámetros:

- Matriz: Matriz en forma estandar y canónica lista para aplicar el método simplex
- Padre: Este es el valor del nodo del cual proviene (0 si es el primer nodo)
- T: Vector de restricciones en el formato definido previamente
- numNodo: Número del nodo actual (el cual es dado por la función auxiliar referenciada previamente como `contadorNodo()`:)
- c: El vector de los coeficientes. Aunque este vector en teoría es el mismo para todos, al ser utilizado en funciones como `TablaSimplexAmpliada` se le agregan ceros innecesarios para el proceso de programación lineal, por lo que siempre se están realizando copias de este valor para no perder su naturaleza original
- z0: El mismo valor de z0 original.
- EsAmpliado: Ya que al realizar la programación entera en la división con restricción mayor o igual se generan variables artificiales por lo que hay que verificar si el programa es ampliado o no.
- Artificiales: La lista de las posiciones de las variables artificiales

El código final de SimplexEntero es una función recursiva la cual se llama a sí misma en las ocasiones que encuentra un valor no entero en el vector solución del método simplex realizado. Por facilidad la división la hace tomando en cuenta el primer valor no entero encontrado por lo que los resultados pueden variar de los realizados por el usuario si este toma otro valor para añadir las restricciones necesarias. El código final es el siguiente:

```
def simplexEntero(Matriz, padre, T, numNodo, c, z0, EsAmpliado,
    Artificiales):
    global nodoContador
    NColumnas = len(T[0])

    #Proceso general para resolver simplex con artificiales
    if EsAmpliado:
        VectorArtificiales=[]
        for _ in Artificiales:
            VectorArtificiales.append(int(_))
        VectorArtificiales.sort(reverse=True)

        Matriz=AlgoritmoSimplex(Matriz, True)
        del(Matriz[-1])
        for i in range(len(Matriz)):
            for IndiceArtificial in VectorArtificiales:
                del(Matriz[i][IndiceArtificial-1])

    Matriz=AlgoritmoSimplex(Matriz, False)
    solucion=obtenerVector(Matriz)

    #Solucion con solo variables importantes
    vectorSoloVar=[]
    for i in range(NColumnas-2):
        vectorSoloVar.append(solucion[i])

    #Obtener el optimo de esta iteracion
    optimo=Matriz[-1][-1]

    #Print de los resultados del nodo
    print("-----------------------------------------------")
    print("\nNodo:", numNodo, "| Hijo de:",padre)
    print("T usado:", T, "\nResultados:")
    print("z*:",optimo)
    print("Vector de solucion:", vectorSoloVar)

    #Verificar que los valores son enteros
    todosEnteros=True
    valor=0
    var=0 #Almacena la posicion de la variable no entera
    for num in solucion:
        if num % 1 != 0: #Si se encuentra un valor no entero
            todosEnteros=False
            valor=num #Se toma el primer valor no entero
            var=solucion.index(num)
            break
```

```
46
47      if todosEnteros:
48          if optimo == 0: # Si la funcion no devuelve optimo
49              print("Solucion no factible")
50          else:
51              print("Converge con solucion:", optimo)
52      else: #El caso en que se tenga que seguir la programacion entera
53          v1 = math.floor(valor)
54          v2 = math.ceil(valor)
55          cCopiado=copy.deepcopy(c)
56          c2=copy.deepcopy(c)
57          #Crear las nuevas restricciones
58
59          #-----------------------------------------------------------
60
61          nuevaRest1 = [0]*(NColumnas) #Tamano de la restriccion
62          nuevaRest1[-2]="<=" #Agregar el menor o igual
63          nuevaRest1[-1]=v1 # ... al piso del numero no entero
64          nuevaRest1[var]=1 # Agregar un 1 a la posicion de la variable
65          T1=copy.deepcopy(T) #Tomar el T actual
66          T1.append(nuevaRest1) #Agregar el T al vector de restricciones
67          T11=copy.deepcopy(T1) #Guardar el T1 actual
68          [Matriz1, Amp1, Art1]=TablaSimplexAmpliada(c, T1, z0)
69
70          #Recalcular el simplex entero
71          contadorNodo()
72          simplexEntero(Matriz1, numNodo, T11, nodoContador, cCopiado, z0,
    Amp1, Art1)
73
74          #------------------ Division ---------------------------
75
76          nuevaRest2 = [0]*(NColumnas)
77          nuevaRest2[-2]=">=" #Agregar el mayor o igual
78          nuevaRest2[-1]=v2 # ... al techo del numero no entero
79          nuevaRest2[var]=1
80          T2=copy.deepcopy(T)
81          T2.append(nuevaRest2)
82          T22=copy.deepcopy(T2)
83          [Matriz2, Amp2, Art2]=TablaSimplexAmpliada(c2, T2, z0)
84
85          #Recalcular el simplex entero
86          contadorNodo()
87          simplexEntero(Matriz2, numNodo, T22, nodoContador, cCopiado, z0,
    Amp2, Art2)
88      return
```

Un ejemplo de utilización de código se puede ver añadiendo las siguientes lineas:

```
# Ejemplo
c=[-6,-8]
T=[[2, 3, '<=', 9], [2, 1, '<=', 6]]
z0=0
nodoContador=1
T_inicial=copy.deepcopy(T)
c_inicial=copy.deepcopy(c)

[MatrizA, Ampliado, Artificiales]=TablaSimplexAmpliada(c, T, z0)
simplexEntero(MatrizA, 0, T_inicial, 1, c_inicial, z0, Ampliado,
    Artificiales)
```

El cual retorna lo siguiente:

```
-----------------------------------------------

Nodo: 1 | Hijo de: 0
T usado: [[1, 6, '<=', 50], [12, 1, '<=', 60]]
Resultados:
z*: 51.267605633802816
Vector de solucion: [4.366197183098592, 7.605633802816901]
-----------------------------------------------

Nodo: 2 | Hijo de: 1
T usado: [[1, 6, '<=', 50], [12, 1, '<=', 60], [1, 0, '<=', 4]]
Resultados:
z*: 47.666666666666664
Vector de solucion: [4.0, 7.666666666666666]
-----------------------------------------------

Nodo: 3 | Hijo de: 2
T usado: [[1, 6, '<=', 50], [12, 1, '<=', 60], [1, 0, '<=', 4], [0, 1, '
    <=', 7]]
Resultados:
z*: 47.0
Vector de solucion: [4.0, 7.0]
Converge con solucion: 47.0
-----------------------------------------------

Nodo: 4 | Hijo de: 2
T usado: [[1, 6, '<=', 50], [12, 1, '<=', 60], [1, 0, '<=', 4], [0, 1, '
    >=', 8]]
Resultados:
z*: 28.0
Vector de solucion: [2.0, 8.0]
```

```
Converge con solucion: 28.0
-----------------------------------------------

Nodo: 5 | Hijo de: 1
T usado: [[1, 6, '<=', 50], [12, 1, '<=', 60], [1, 0, '>=', 5]]
Resultados:
z*: 50.0
Vector de solucion: [5.0, -0.0]
Converge con solucion: 50.0
```

Para este código fue necesario la utilización de la biblioteca copy ya los valores de c y T son modificados al usar funciones como `TablaSimplexAmpliada` la cual agrega espacios vacíos que alargan el código y entorpecen la optimización del mismo, por lo que muchas copias de estos atributos son creadas antes de ejecutar la función `TablaSimplexAmpliada`.

3. Resultados

El resultado de la realización de este proyecto es un código en Python el cual recibe un programa lineal, y puede retornar diferentes cosas, según lo necesitemos y le especifiquemos al programa el estado requerido.

- El primer estado nos retorna el valor óptimo y donde es alcanzado.
- El segundo estado además de lo anterior mencionado retorna la tabla simplex inicial, intermedia si utiliza variables artificiales y la tabla final del simplex.
- El tercer estado nos retorna un código LaTeX con la solución detallada paso a paso de como se resuelve el problema de programación lineal.

Además se cuenta con otra sección en donde se pueden resolver ejercicios o problemas de programación entera y nos muestra el proceso de ramificación y acotamiento como una lista. Cabe destacar que el programa puede resolver problemas en donde se incluyan variables de holgura, superfluas o artificiales, según sea necesario y este nos retornara lo anterior mencionado en caso de tener solución y en caso contrario indica que el programa no posee solución, ya sea porque no paga la penalización de las variables artificiales o no se puede optimizar la función objetivo.

Finalmente se generó un repositorio en Github, para así almacenar todo lo realizado durante la creación del programa, de manera que todos los integrantes tuvieran acceso al material generado y pudiéramos evidenciar un avance. Puede acceder al repositorio mediante el siguiente enlace https://github.com/vyadir/PL-Assignments

4. Problemas, mejoras, expansiones futuras

Una posible mejora para el futuro es tomar la función que resuelve la programación entera y con los datos retornados de los nodos, el valor óptimo, las variables, su estatus y los nodos hijos, almacenarlos y luego crear una función, la cual se encargara de crear el diagrama de árbol generado por el proceso de programación entera por ramificación y acotamiento, para finalmente obtener una imagen con dicha información organizada.

Otra posible mejora al programa puede ser la optimización del código, ya sea utilizando clases, programación orientada a objetos o incluso recursividad, de manera que mejore su eficiencia.

Por otro lado se podría realizar una expansión, la cual sea generar una interfaz gráfica, en donde un usuario pueda introducir el problema de programación lineal y solicitarle en especifico al programa cual de sus funcionalidades desea utilizar, de manera que sea mas agradable para el usuario el uso del programa generado.

5. Bibliografía

OpenAI. (2022). ChatGPT: Language Model based on GPT-3.5.

Stoer, J., Bulirsch, R., Bartels, R., Gautschi, W., & Witzgall, C. (1980). *Introduction to numerical analysis* (Vol. 1993). Springer.

Strang, G. (2016). *Introduction to Linear Algebra*. Wellesley. https://books.google.co.cr/books?id=efbxjwEACAAJ