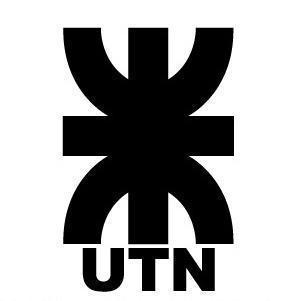
**UNIVERSIDAD TECNOLÓGICA NACIONAL**

**FACULTAD REGIONAL DE SANTA FE**

**TRABAJO PRÁCTICO N° 2**

**Sistemas de Ecuaciones No Lineales**

**Cátedra:** Matemática Superior

**Carrera:** Ingeniería en Sistemas de Información

**Comisión:** 3A

**Año:** 2021 - 2° Cuatrimestre

**Integrantes:** Albani, Juan - juanalbani48@gmail.com

Musso, Emilio - emilio.musso2898@gmail.com

Nuñez, Nahuel - nahuelnunezz14@gmail.com

Peiretti, Tomás - tomaspeiretti@gmail.com

Riquelme, Nahuel - nahuelriquelme00@gmail.com

**ÍNDICE**

Desarrollo…………………………………..…………………………………………...……………..…………1

Diseño del método ReboteMS…………………………………………….……………………..….1.1

Primer acercamiento…………………………………………………………...………....1.1.1

Problemas enfrentados……………………………………………………………..…….1.1.2

Método iterativo………………………………………………………….………………...1.1.3

Primeras pruebas………………………………………………………………………….1.1.4

Implementación de Newton-Raphson para sistemas de ecuaciones no lineales……………..1.2

Comparación entre ReboteMS y Newton-Raphson para SEC no lineales………….………....1.3

Observaciones y conclusiones………………...…………………………………………………………...….2

**1 - DESARROLLO**

Como consideración general, se informa que 3 códigos numéricos fueron utilizados a lo largo del trabajo práctico para identificar las condiciones de corte de los métodos iterativos:

* Código 0: Retorna por condición de tolerancia.
* Código 1: Retorna por condición de máximas iteraciones.
* Código -1: Retorna por condición de divergencia.

Durante el desarrollo, se observó que el método de sympy que permite sustituir las variables, al usar funciones provistas por sympy (por ejemplo, sympy.exp) nos retornaba valores exactos que ralentizaba los cálculos. Por esto, fue necesario castear todas las evaluaciones a float perdiendo precisión en el proceso pero ganando velocidad de cálculo.

**1.1 - DISEÑO DEL MÉTODO ReboteMS**

**1.1.1 - PRIMER ACERCAMIENTO**

Durante la primera interpretación del método, se pudo observar que los pasos del mismo eran muy similares en su estructura, y que las únicas diferencias entre ellos son:

* Las superficies de nivel que deben ser paralelas a la recta.
* El punto de paso de la recta.
* La superficie de nivel cuya intersección con la recta debe ser obtenida.

Por ello, se creó la función obtenerPuntoInterseccion que pide las 3 funciones, el punto inicial y la tolerancia para calcular la intersección con la tercera función:

def reboteMS(f1, f2, f3, Pn, tolerancia, max\_iter):

# max\_iter será la cantidad de iteraciones máximas para el método N-R

Q1 = obtenerPuntoInterseccion(f1,f2,f3,Pn,tolerancia,max\_iter)

msg = validarPunto(Q1)

if msg != "ok":

return msg

Q2 = obtenerPuntoInterseccion(f2,f3,f1,Q1,tolerancia,max\_iter)

msg = validarPunto(Q2)

if msg != "ok":

return msg

Pn1 = obtenerPuntoInterseccion(f1,f3,f2,Q2,tolerancia,max\_iter)

msg = validarPunto(Pn1)

if msg != "ok":

return msg

return Pn1

Siendo **r** la recta paralela a **f** y **g** que pasa por **P**, su vector director es calculado como el producto cruz de los vectores normales de los planos tangentes de **f** y **g** en **P**. Para encontrar el punto de contacto con la recta dentro de la función, se utilizó el método **N-R** para hallar una solución a la ecuación **h(r) = 0**.

# f y g son las funciones cuyas superficies de nivel seran paralelas

# a la recta r que pasa por el punto P, y h es la funcion cuya superficie

# de nivel se utiliza para intersectar con la recta r y obtener el punto Q

def obtenerPuntoInterseccion(f, g, h, P, tolerancia, max\_iter):

fx,fy,fz = derivadasParciales(f)

gx,gy,gz = derivadasParciales(g)

derivf\_evaluada = [float(evaluar(fx,P)), float(evaluar(fy,P)), float(evaluar(fz,P))]

derivg\_evaluada = [float(evaluar(gx,P)), float(evaluar(gy,P)), float(evaluar(gz,P))]

director1 = np.cross(derivf\_evaluada,derivg\_evaluada)

t = sym.symbols('t')

r = (director1[0]\*t + P[0], director1[1]\*t + P[1], director1[2]\*t + P[2])

#print(r)

intersec\_h\_r1 = evaluar(h,r)

#print(intersec\_h\_r1)

# Obtención del punto Q

# los argumentos son: t0, f, max\_iteraciones, tolerancia

tn, yn, codigo = newton\_raphson(1,intersec\_h\_r1,max\_iter,tolerancia\*0.01)

if codigo != 0:

return codigo

Q = (r[0].subs(t,tn), r[1].subs(t,tn), r[2].subs(t,tn))

return Q

**1.1.2 - PROBLEMAS ENFRENTADOS**

En el método N-R para la aproximación del punto **tn** en obtenerPuntoInterseccion(), en primera instancia se había planteado observar gráficamente la intersección de la recta con la superficie de nivel para dar un punto inicial aproximado, pero esto resultaba poco “programático”.

También se pensó la posibilidad de aplicar Bisección o Regula-Falsi para la obtención de un **t** que permita que el método N-R converja siempre, pero tampoco no se logró llegar a una solución que pueda ser implementada en código. Por ende, el valor inicial de **t** fue fijado a 1 para mantener la simplicidad del procedimiento.

Además, en la implementación del método newton\_raphson() (el cual puede leerse en el archivo ReboteMS.py que se adjunta), la siguiente expresión almacena una fracción en **xi**:

xi = xi - (f\_evaluada/dfdt.subs(t,xi))

Esto hacía que en cada iteración el método trabaje con fracciones con numeradores y denominadores muy grandes, requiriendo de mucho tiempo de procesamiento para calcular las evaluaciones de la función.

Para solucionar esto, se castea la expresión a un número de punto flotante de 64 bits:

xi = float(xi - (f\_evaluada/dfdt.subs(t,xi)))

Durante la fase de pruebas, se detectó un problema en la forma de evaluar si el método estaba divergiendo. Inicialmente, se había tomado la idea de calcular la distancia entre el punto actual y el punto hallado en la anterior iteración para observar la convergencia del método. Sin embargo se pensaron en algunos ejemplos en lo que esto podría ser erróneo, por lo cual se decidió cambiarlo por la norma infinita de un vector, un vector **X** cuyas componentes son , y .

**1.1.3 - MÉTODO ITERATIVO**

Para evaluar la tolerancia de las aproximaciones obtenidas en el método, se decidió utilizar la norma infinita de un vector **X** cuyas componentes son , y . El vector **ans** se encarga de almacenar este valor para cada una de las iteraciones y así poder verificar si el método diverge. Para este caso, se considera que el método diverge a partir de la segunda iteración si la norma obtenida en el paso actual es mayor a la del paso anterior.

La norma del vector utilizada fue la norma infinita ya que por definición es el valor máximo del vector en valor absoluto, esto nos permite asegurar que si ese valor es menor a la tolerancia el resto de las componentes del vector también lo serán (en valor absoluto). Esta elección fue completamente arbitraria y dependiendo del caso de aplicación del algoritmo otra norma podría resultar más apropiada.

def reboteMS\_iterativo(f1, f2, f3, Pn, tolerancia, max\_iter):

i=0

puntos = []

ans = [max(abs(float(evaluar(f1,Pn))), float(abs(evaluar(f2,Pn))), float(abs(evaluar(f3,Pn))))]

while not(isinstance(Pn,str)) and i<=max\_iter and ans[-1] > tolerancia:

puntos.append(Pn)

if i>1 and abs(ans[-2]) < abs(ans[-1]):

print("El metodo reboteMS diverge")

return

print("Iteracion:",i)

print("Pn+1",Pn)

Pn = reboteMS(f1,f2,f3, Pn, tolerancia, max\_iter\*2)

if not isinstance(Pn,str):

ans.append(max(abs(float(evaluar(f1,Pn))), float(abs(evaluar(f2,Pn))), float(abs(evaluar(f3,Pn)))))

i+=1

if isinstance(Pn,str):

print("El método falló en la iteración", i)

print("La causa del fallo es:", Pn)

return Pn

**1.1.4 - PRIMERAS PRUEBAS**

Para probar que el método funciona correctamente, se planteó en primer lugar el siguiente caso:

En este caso, se evaluó gráficamente paso por paso el método haciendo uso de Geogebra. En la siguiente imagen, se muestra el cálculo del punto **Q1** de la primera iteración (punto **D** en la imagen):

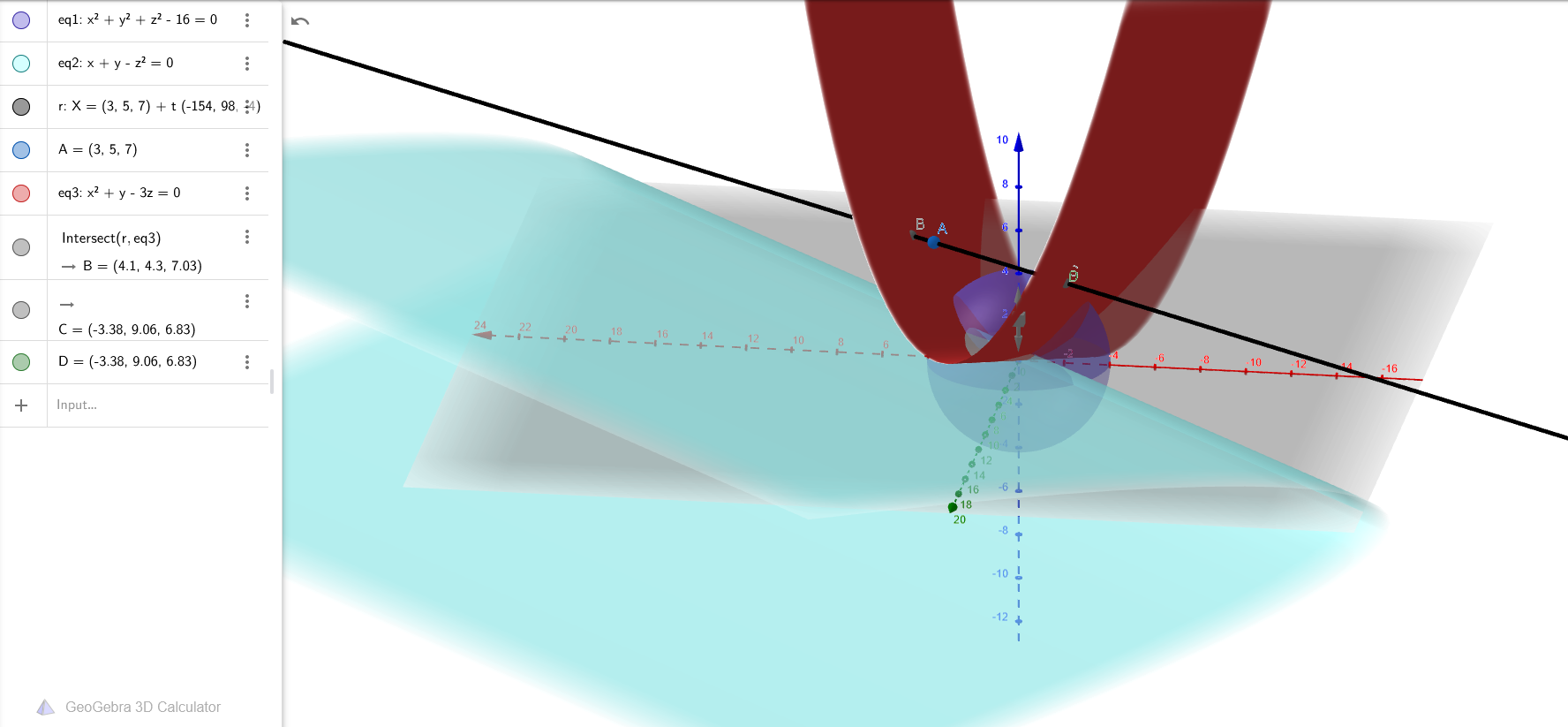


Imagen 1 - Obtención de Q1 a partir de la intersección de **r** con **f3** (eq3)

En la obtención de **Q2**, la recta **r2** paralela a f2 y f3, que pasa por **Q1** (D en el gráfico) no intersecta con la superficie de nivel f1 (la esfera). Por lo que, el método N-R diverge, al igual que en la ejecución del método.

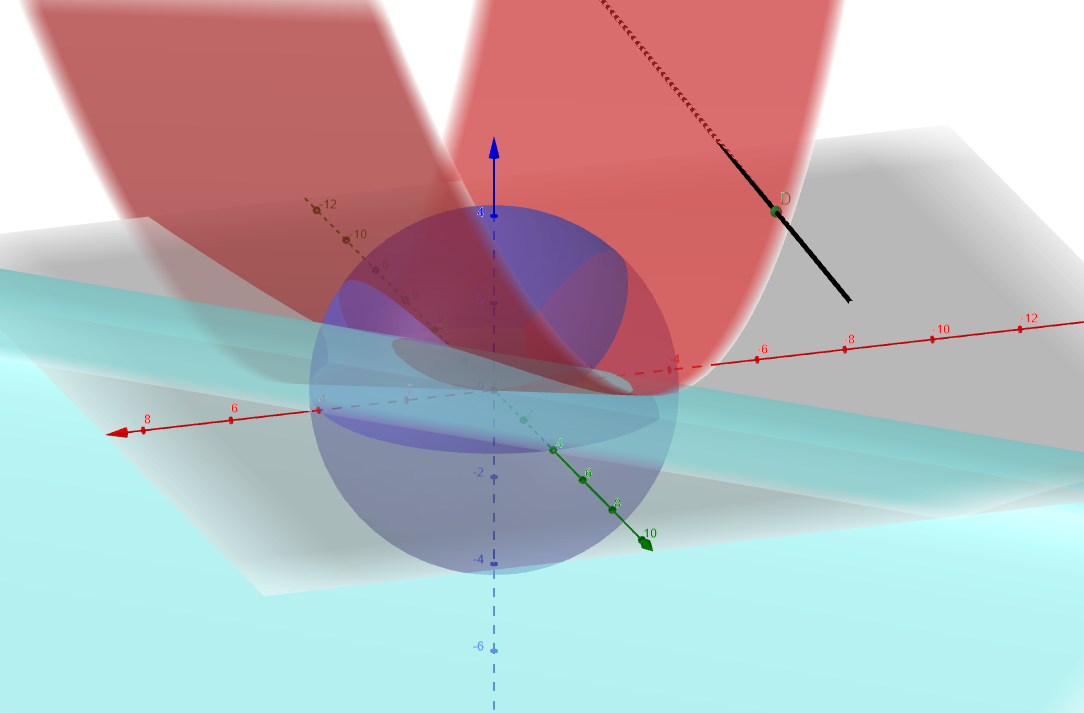


Imagen 2 - La recta **r2** no intersecta con la esfera.

Como segunda prueba, se cambió el punto inicial Pn a (1,1,1), manteniendo las funciones utilizadas previamente para ver si de esta forma el método converge. Los resultados fueron los siguientes:

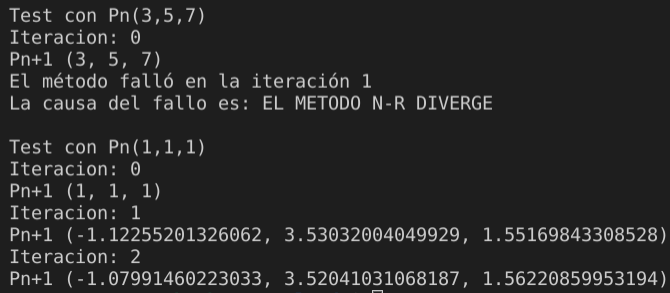


Imagen 3 - resultados del método ReboteMS iterativo

El punto obtenido Pn+1 (-1.08, 3.52, 1.56) es una buena aproximación a una de las soluciones al sistema dado, ya que gráficamente se puede visualizar que es uno de los puntos en donde las tres superficies de nivel se intersecan.

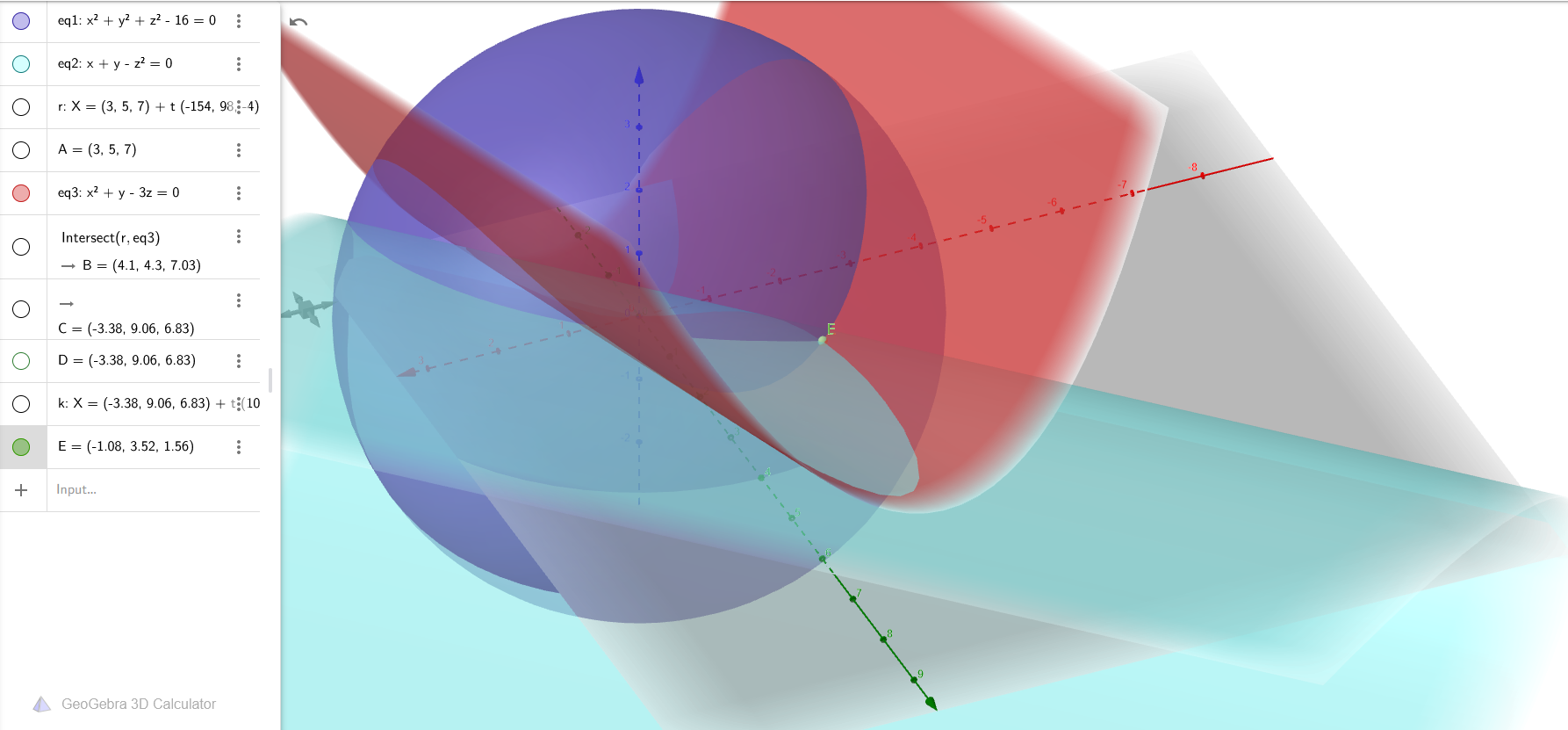


Imagen 4 - Intersección de las 3 superficies de nivel (punto E).

**1.2 - IMPLEMENTACIÓN DE NEWTON-RAPHSON PARA SISTEMAS DE ECUACIONES NO LINEALES**

Para este punto, se planteó el mismo procedimiento aplicado para resolver los ejercicios de sistemas de ecuaciones de la clase de práctica. Pero, inicialmente, para el método de búsqueda de la soluciones se decidió utilizar Gauss-Seidel, ya que se vió en la clase de teoría que este método no requiere calcular la inversa del Jacobiano y que es uno de los métodos iterativos que más rápido converge (si se plantea correctamente).

La implementación planteada fue realizada en base a los códigos brindados por la cátedra y la fórmula de Gauss-Seidel:

def gauss\_seidel(A,b,x0,max\_iter,tolerancia):

A = np.asarray(A,dtype=float)

b = np.asarray(b,dtype=float)

x0 = np.asarray(x0,dtype=float)

dim = x0.shape[0]

aux = A.diagonal()

M = aux\*np.eye((dim))-np.tril(A) # matriz triangular inferior opuesta (sin diagonal)

N = aux\*np.eye((dim))-np.triu(A) # matriz triangular superior opuesta (sin diagonal)

ans = [np.linalg.norm(np.dot(A,x0)-b)]

x = x0.copy()

iter=0

while ans[iter]>tolerancia and iter<max\_iter:

for i in range(dim):

sum1=0

sum2=0

for j in range( len(M[i]) ):

sum1+=M[i][j]\*x[j]

for j in range( len(N[i]) ):

sum2+=N[i][j]\*x[j]

x[i]=(b[i] + sum1 + sum2) / A[i][i]

ans.append(np.linalg.norm(np.dot(A,x)-b))

if ans[-1] > ans[-2]:

return "GAUSS SEIDEL FINALIZO POR DIVERGENCIA",ans,-1

iter+=1

# codigo = 0 -> retorna por condicion de tolerancia

# codigo = 1 -> retorna por condicion de maximas iteraciones

# codigo = -1 -> retorna por condicion de divergencia

if ans[-1] <= tolerancia:

return x,ans,0

return "GAUSS SEIDEL FINALIZO POR CANTIDAD DE ITERACIONES",ans,1

El problema que se tuvo con esta implementación, es que no se logró que converja para ningún caso de prueba planteado. Esto se debe a que hubo un error de concepto, al no conocer en el momento de la implementación las condiciones para que el método converja. No se planteó el acondicionamiento del Jacobiano para que sea una matriz diagonal dominante, ni se plantea la condición de la norma de la matriz.

Por ende, luego de realizar una consulta con los profesores, se decidió que es más conveniente y sencillo resolver el sistema calculando la inversa del Jacobiano, ya que para este trabajo, el tamaño de la matriz siempre será de 3x3.

Para la resolución del sistema, existe un método ya implementado en el módulo de numpy; **linalg.solve()**, pero se decidió codificar todos los pasos para entender mejor el procedimiento.

def newton\_raphson\_sistema(f, g, h, Pn, max\_iter, tolerancia):

jacobiano = []

jacobiano.append(np.asarray(derivadasParciales(f)))

jacobiano.append(np.asarray(derivadasParciales(g)))

jacobiano.append(np.asarray(derivadasParciales(h)))

iter = 0

ans = [max(abs(float(evaluar(f,Pn))), abs(float(evaluar(g,Pn))), abs(float(evaluar(h,Pn))))]

# codigo = 0 -> retorna por condicion de tolerancia

# codigo = 1 -> retorna por condicion de maximas iteraciones

# codigo = -1 -> retorna por condicion de divergencia

while max\_iter>iter and ans[-1] > tolerancia: # falta la tolerancia?

#evaluacion de las derivadas parciales en el punto

jacobiano\_evaluado = []

for deriv in jacobiano:

jacobiano\_evaluado.append([float(evaluar(deriv[0],Pn)),float(evaluar(deriv[1],Pn)),float(evaluar(deriv[2],Pn))])

#evaluacion de las funciones en el punto

funciones\_evaluadas = [[-1\*float(evaluar(f,Pn))],[-1\*float(evaluar(g,Pn))],[-1\*float(evaluar(h,Pn))]]

if np.linalg.det(jacobiano\_evaluado) == 0:

print("NO ES INVERTIBLE LA MATRIZ, NO SE PUEDE CONTINUAR")

return

inversa\_jacobiano = np.matrix(jacobiano\_evaluado)\*\*-1

delta = inversa\_jacobiano \* funciones\_evaluadas

Pn = (Pn[0] + delta[0,0], Pn[1] + delta[1,0], Pn[2] + delta[2,0])

ans.append(max(abs(float(evaluar(f,Pn))), abs(float(evaluar(g,Pn))), abs(float(evaluar(h,Pn)))))

if iter > 0 and ans[-1] > ans[-2]:

return Pn, ans, -1

iter+=1

if iter < max\_iter:

return Pn, ans, 0

return Pn, ans, 1

Al igual que para ReboteMS\_iterativo(), se aplica la norma infinita para evaluar la tolerancia por su simplicidad de cálculo.

Cabe resaltar que durante la implementación del N-R para sistemas también se tuvieron errores de concepto, ya que en principio se estaba considerando el **delta** como la nueva aproximación del **Pn** en vez de adicionarlo a la aproximación anterior. Causando que el método nunca converja y que los puntos siempre parecieran estar en la dirección contraria a la solución.

**1.3 - COMPARACIÓN ENTRE ReboteMS y NEWTON-RAPHSON PARA SEC NO LINEALES**

A continuación, se someten los algoritmos implementados a distintos sistemas de ecuaciones para observar su comportamiento. Para la prueba de los algoritmos se varía el Punto Inicial, la tolerancia admitida para la solución y la cantidad máxima de iteraciones permitidas. Serán aplicados sobre funciones de variada naturaleza (trigonométricas, exponenciales, etc.), y se evalúan los siguientes parámetros:

* Iteraciones.
* Tiempo.
* Resultado.

Los sistemas planteados pueden ser clasificados según la cantidad de soluciones que tienen, como se muestra en la tabla 1. La cantidad de soluciones de cada sistema fueron determinadas gráficamente haciendo uso del software Geogebra.

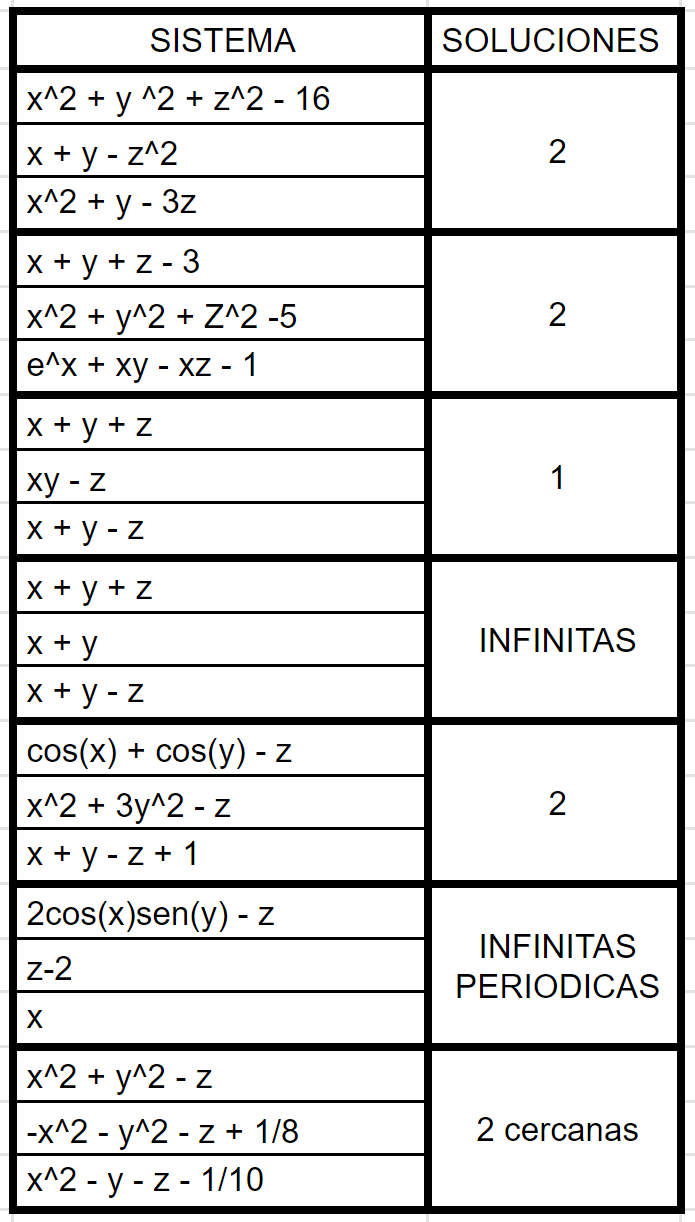


Tabla 1 - Cantidad de soluciones de cada sistema planteado.

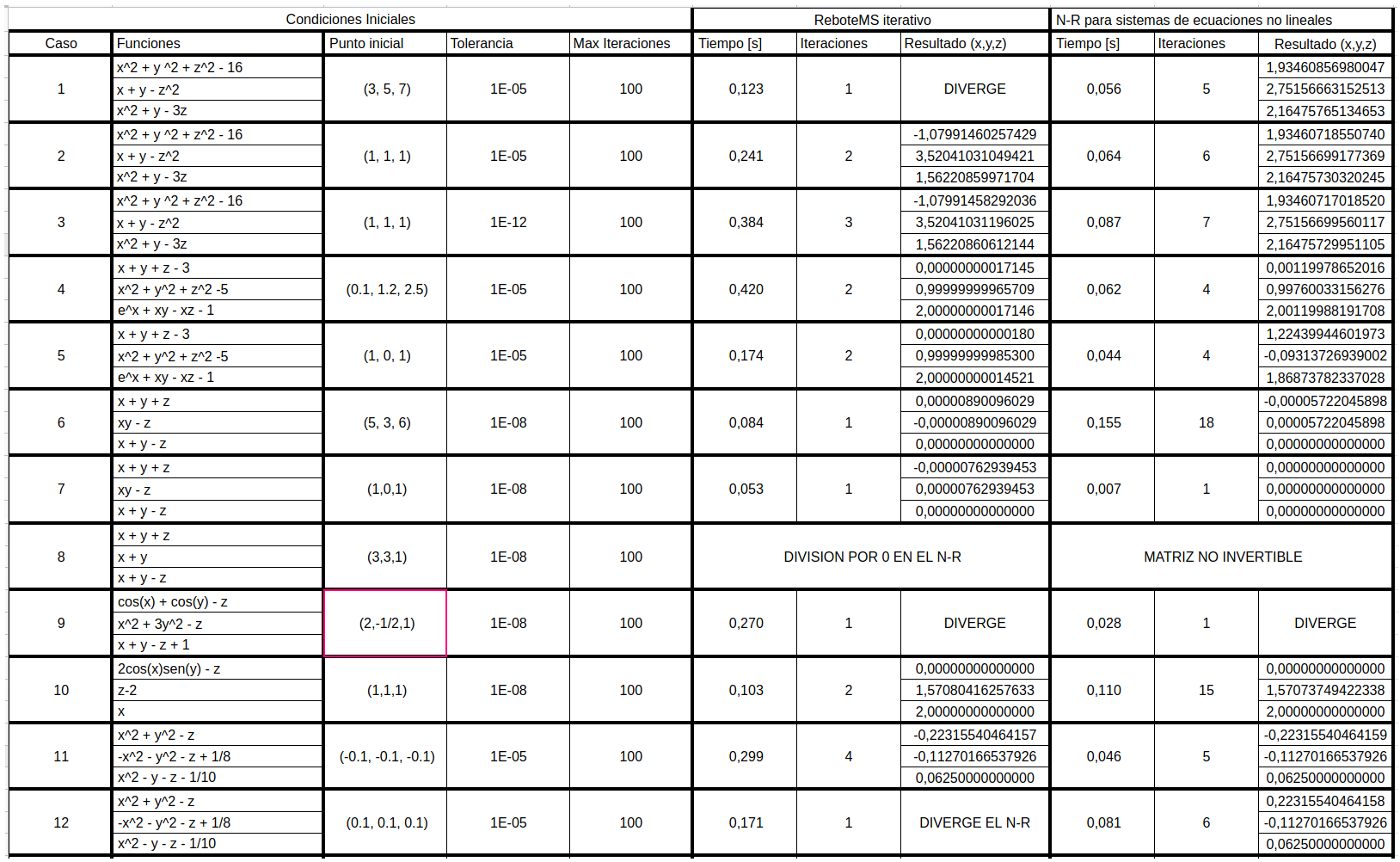
****

Tabla 2 - Prueba de los métodos.

Estas pruebas fueron realizadas utilizando el archivo pruebas.py que se adjunta, cambiando las variables f1, f2, f3, tol, it y P.

**2 - OBSERVACIONES Y CONCLUSIONES**

* En el caso 12 del punto 1.3, se verificó gráficamente que el método ReboteMS diverge por el mismo motivo que lo explicado en el punto 1.1.4. Parece ser que para éste método es especialmente delicado con su punto de inicio para asegurar la convergencia en los casos en que se tienen superficies de nivel cuya extensión es restringida respecto a alguno o varios de los ejes coordenados. (Por ejemplo, paraboloides, elipsoides, etc.). Parece ser importante que el punto de inicio se encuentre “dentro” de la superficie más restrictiva.
* En los ejemplos mostrados, se observa que el ReboteMS converge más rápido que el N-R debido a que necesita una menor cantidad de iteraciones, pero el tiempo por iteración que requiere el mismo es mucho mayor que el método N-R para sistemas.
* Una gran ventaja del método ReboteMS es que no se necesitan calcular y evaluar nxn derivadas parciales para resolver el sistema de ecuaciones en cada iteración (que puede ser una operación muy costosa), ahorrando memoria del sistema. Además, evita la necesidad de calcular la inversa del Jacobiano o del uso de un método iterativo para hallar las soluciones.
* Se observa que el Punto de Inicio tiene una gran importancia en ambos métodos para asegurar la Divergencia o la Convergencia, y en el que caso de que el algoritmo converja, cuál será la solución que se obtiene.
* Ambos métodos divergen cuando el sistema de ecuaciones dado posee infinitas soluciones, ya que, para ReboteMS dentro del N-R en algún punto ocurre una división por cero, y para N-R para sistemas el Jacobiano no es invertible. A pesar de esto, si el sistema posee infinitas soluciones periódicas, si es posible hallar una solución aplicando los métodos.
* Se cree que sería posible plantear como mejora al reboteMS que, para el cálculo del punto que surge como la intersección de la recta “paralela” y la superficie de nivel, que sea posible setear el valor inicial por parte del usuario para que coloque un valor que asegure la convergencia del N-R.
* Además, podría plantearse que el usuario en vez de brindar un valor inicial, brinde el intervalo en donde se encuentre la solución para aplicar Bisección o Regula-Falsi y con esto iniciar el N-R con un valor más cercano a la solución.
* Se considera que dependiendo del caso de aplicación del algoritmo la selección de la norma infinita como medida de la tolerancia puede no ser la más apropiada y debería analizarse para el caso.