# UNIVERSIDAD TECNOLOGICA NACIONAL
# FACULTAD REGIONAL SANTA FE

## Matemática Superior

Trabajo Práctico Nº 2:

Búsqueda de Raíces y Sistemas Lineales


Alumnos:

- Esposito Matias
- Gaucher Daniel
- Musuruana Lara


**2023**

# ***BUSQUEDA DE RAICES***

## ***Ejercicio 1a)***

Para comenzar utilizaremos el método cerrado de bisección (ya que es un método robusto al momento de converger) en cada una de las funciones para estimar el costo computacional de iteraciones necesarias para aproximarse a una raíz y respecto a un residuo que consideramos sería adecuado al momento del corte.

**Las entradas para el algoritmo de bisección serán las mismas para las tres funciones, y estas son:**

➤Extremo A: -5
➤Extremo B = 6
➤Cantidad máxima de iteraciones: 50
➤Tolerancia: 1 x$10^{-10}$

A partir de la gráfica de las tres funciones se determinó un intervalo válido cerrado en el cual se contempla cada una de las raíces de las funciones. De esta manera se podrá medir la eficiencia que tiene bisección con los mismos parámetros de entrada y es por ello que se fijó un extremo en -5 y el otro en 6.
Con esta información condicionamos el algoritmo para que el corte quede establecido al realizar un máximo de 50 iteraciones o al alcanzar un residuo menor a 1*x$10^{-10}$.

- $f_1(x) = x\, e^{-\frac{x}{2}}$
  **Raíz**: -1.4551915228366852 x$10^{-11}$
  **Iteraciones**: 36
  **Residuo**: -1.455191522847273 x$10^{-11}$

- $f_2(x) = \arctan(x)$
  **Raíz**: -1.4551915228366852 x$10^{-11}$
  **Iteraciones**: 36
  **Residuo**: -1.4551915228366852 x$10^{-11}$

- $f_3(x) = -\cos(|x| + \pi/4) + 0.8$
  **Raíz**: -4.85428603499895
  **Iteraciones**: 34
  **Residuo**: -6.057487844657317 x$10^{-12}$

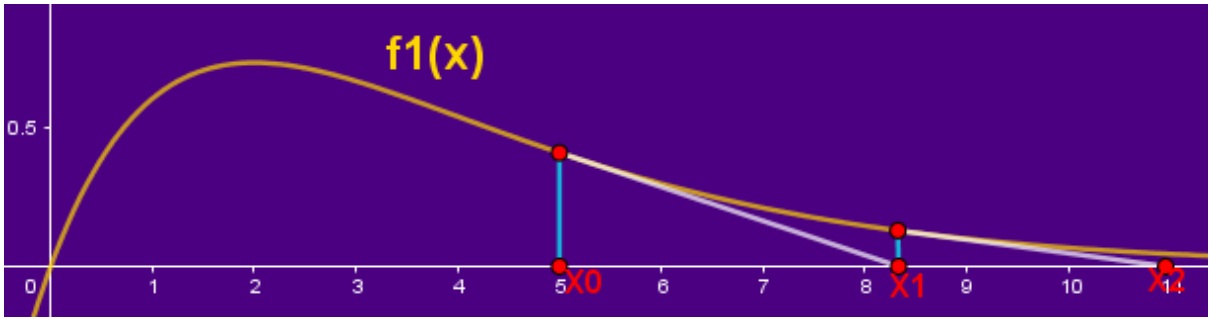
**Ahora para el método de Newton-Raphson las condiciones serán**:

➤Cantidad máxima de iteraciones: 50
➤Tolerancia: 1 x$10^{-10}$

- $f_1(x) = x\, e^{-\frac{x}{2}}$ $f_1'(x) = e^{-\frac{x}{2}} - (\frac{x}{2}).e^{-\frac{x}{2}}$ $x_0 = 5$

**Raíz**: 54.338594059201476
**Iteraciones**: 22
**Residuo**: 8.622488429750798 x$10^{-11}$

El método **diverge** a pesar de que el **residuo** parecería demostrar lo contrario ya que tiende a 0. Esto es causado por una mala elección de la condición inicial $x_0 = 5$ y además a la función $f_1(x)$ presenta una concavidad la que provoca que al utilizar el método con el $x_0$, los puntos sucesivos obtenidos se alejen cada vez más de la raíz.

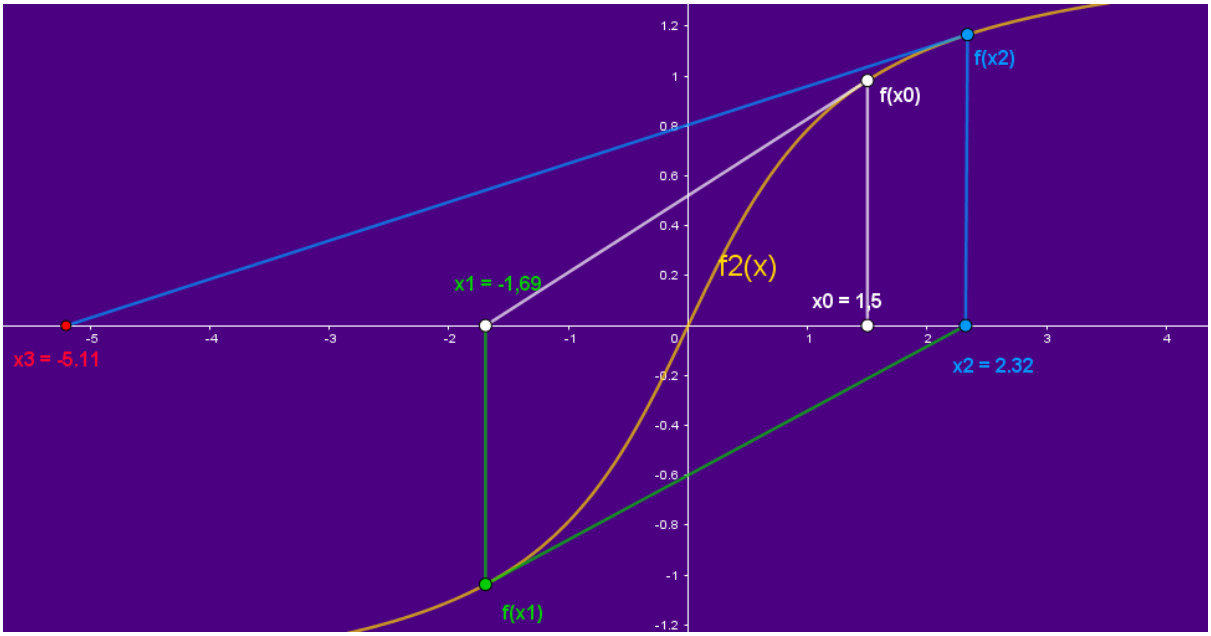



- $f_2(x) =$ arctan(x) $f_2'(x) = \frac{1}{1+x^2}$ $x_0 = 1.5$

**Raíz:** 2.4539946374984955 x$10^{108}$
**Iteraciones:** 10
**Residuo:** 1.5707963267948966

En este caso el método de Newton-Raphson **no converge dado el x0 inicial.** Al calcular la derivada de f(x) y al obtener el siguiente punto $x_1$, este se encuentra más alejado de la raíz, y al utilizar nuevamente el método se aleja aún más es decir el método **diverge.**

Esto se debe a una mala elección de la condición inicial y también a la propia naturaleza de la función **arctan(x)**, la cual tiene una pendiente que se vuelve horizontal a medida que nos alejamos del origen, generando que las segundas derivadas muy cercanas a 0 (pendiente horizontal) y estas a su vez produciendo rectas tangentes horizontales que finalmente desembocan en puntos cada vez más alejados de la raíz buscada.

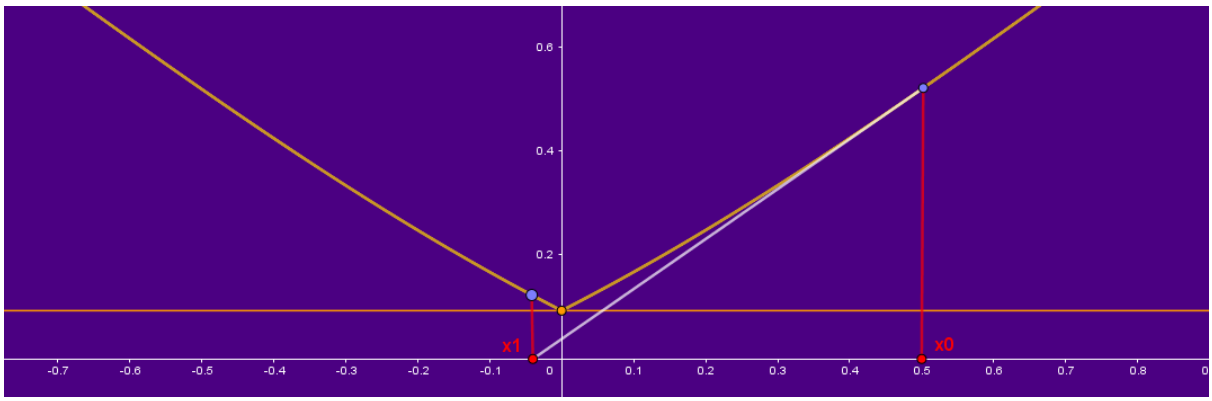



**Nota**: En este caso se modificó el número de iteraciones máximas a 10 dado que el método diverge demasiado rápido y al generar números tan grandes Python comienza a dar errores de ***Overflow.***

- $f_3(x) = -\cos(|x| + \frac{\pi}{4}) + 0.8$ $\quad f_2'(x) = \frac{x(\,sen(|x|) + cos(|x|)\,)}{\sqrt{2}\,|x|}$ $\quad x_0 = 0.5$

  **Raíz**: 0.11335677493066532
  **Iteraciones**: 50
  **Residuo**: 0.17741522347274286

En este caso si bien el método **no diverge**, tampoco se acerca a una posible raíz dado que la función tiene una asíntota horizontal la cual genera que el método se estanque gastando todas las iteraciones posibles sin acercarse a un resultado correcto. Al igual que en los dos casos anteriores con Newton-Raphson, la elección incorrecta de nuestras condiciones iniciales junto a una función con asíntotas o formas cóncavas, producen que este método no vaya a converger nunca o produzca raíces incorrectas.



Luego de examinar ambos métodos en las tres funciones, podemos concluir que **biseccion** es un método **robusto** que nos permite aproximar una raíz incluso cuando no se sabe nada de una función $f$, sin embargo se necesita establecer inicialmente un intervalo válido y el costo computacional en iteraciones es demasiado, ya que el método divide sucesivamente a la mitad el intervalo generando un proceso que se torna lento.

Por otro lado **Newton-Raphson** permite obtener una eficiencia computacional más óptima que bisección añadiendo a que no necesita un intervalo válido, sino que requiere un solo valor inicial. Pero como vimos en los tres casos en los cuales este método **diverge**, el proceso de seleccionar la condición inicial es fundamental para poder hallar una aproximación correcta. Además otro punto en contra de este método es que requiere utilizar la **derivada** de una función la cual no siempre se puede utilizar y hay que recurrir a una **derivada aproximada**.

Podemos combinar estos dos métodos a fin de optimizar el costo computacional, al igual que minimizar las posibilidades de que el método **diverga**

**Primero utilizaremos bisección para hallar un punto muy próximo a la raíz y una vez hecho esto tomaremos esta salida como entrada para Newton-Raphson y hallar un resultado con una aproximación 1 x$10^{-10}$. Utilizaremos las mismas condiciones iniciales en bisección del principio, solo que ahora la tolerancia será de 0.1**

➤Extremo A: -5
➤Extremo B = 6
➤Cantidad máxima de iteraciones: 50
➤Tolerancia: 0.1

- $f_1(x) = x\, e^{-\frac{x}{2}}$
  **Raíz**: -0.015625
  **Iteraciones**: 6
  **Residuo**: -0.01574754839385075

**Tomando esta salida como entrada para Newton-Raphson:**

- $f_1(x) = x\,.\,e^{-\frac{x}{2}}$ $\quad f_1'(x) = e^{-\frac{x}{2}} - (\frac{x}{2}).e^{-\frac{x}{2}}$ $\quad x_0 = -0.015625$
  **Raíz**: -2.6901634883079183 x$10^{-17}$
  **Iteraciones**: 3
  **Residuo**: -2.6901634883079183 x$10^{-17}$

ITERACIONES BISECCIÓN + Newton-Raphson: 9

- $f_2(x) = \arctan(x)$
  **Raíz**: -0.015625
  **Iteraciones**: 6

**Residuo**: -0.015623728620476831

**Tomando esta salida como entrada para Newton-Raphson:**

- $f_2(x)$ = arctan(x) $f_2'(x) = \frac{1}{1+x^2}$ $x_0 = -0.015625$
  Raíz: -1.0963570952786036 x$10^{-17}$
  Iteraciones: 2
  Residuo: -1.0963570952786036 x$10^{-17}$

ITERACIONES BISECCIÓN + Newton-Raphson: 8

- $f_3(x) = -\cos(|x| + \pi/4) + 0.8$
  **Raíz**: -4.828125
  **Iteraciones**: 6
  **Residuo**: 0.0159685748815932

**Tomando esta salida como entrada para Newton-Raphson:**

- $f_3(x) = -\cos(|x| + \frac{\pi}{4}) + 0.8$ $f_2'(x) = \frac{x(\,sen(|x|) + cos(|x|)\,)}{\sqrt{2}\,|x|}$ $x_0 = -4.828125$
  **Raíz**: -4.854286034988843
  **Iteraciones**: 3
  **Residuo**: 6.5503158452884236 x$10^{-15}$

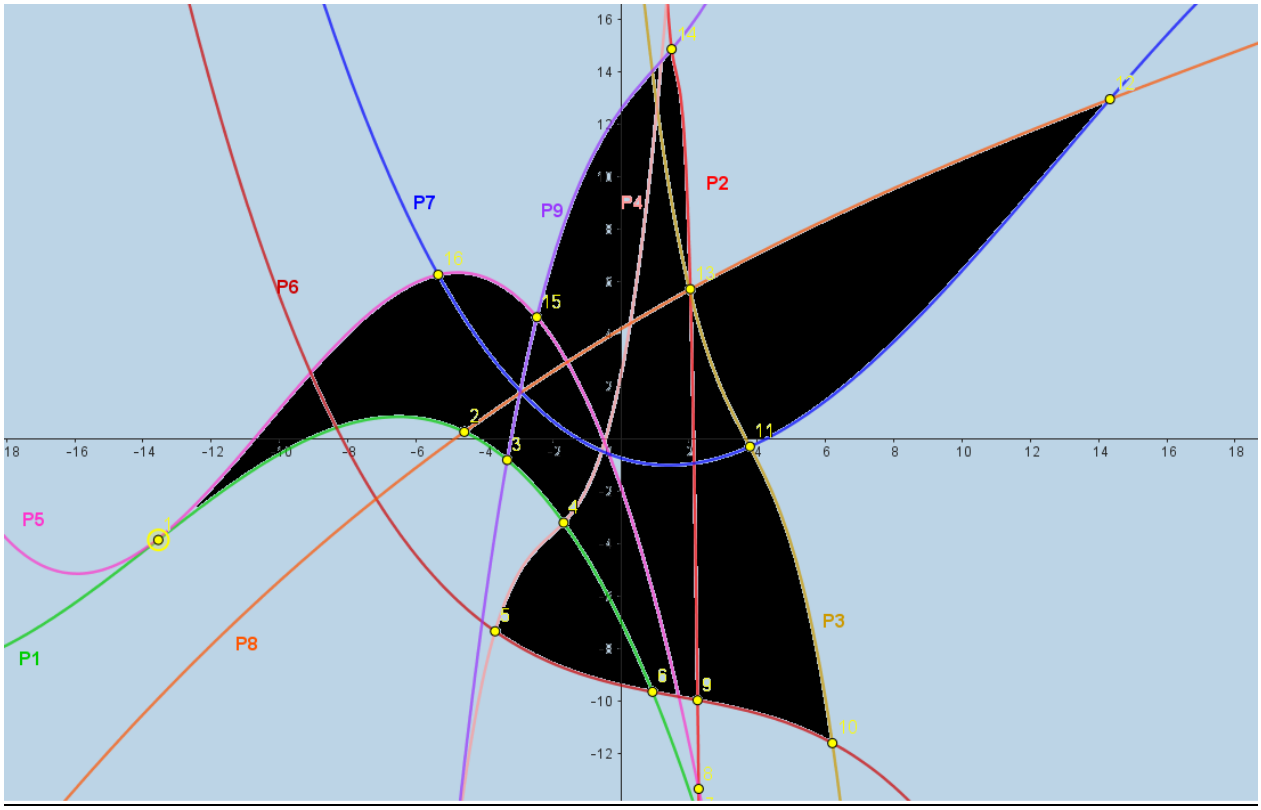
ITERACIONES BISECCIÓN + Newton-Raphson: 9

La desventaja de utilizar el algoritmo de **bisección** y el de **Newton-Raphson** por separados es que el primero asegura convergencia y obtiene una aproximación de la raíz pero a un alto costo computacional y el segundo es más rápido, pero una mala elección de las condiciones iniciales puede derivar resultados incorrectos.

La solución que consideramos es la mas **optima** es ejecutar inicialmente unas pocas iteraciones bisección para acercarse a la raíz salteando así todas las irregularidades que pueda tener la función y luego de esto utilizar este resultado como entrada para **Newton-Raphson** para ahora sí lograr una aproximación más precisa del resultado con un costo computacional menor

# ***Ejercicio 1b)***

***Gráfica de los polinomios :***




***Puntos intersección entre los polinomios necesarios para visualizar el colibrí:***
Para determinar el gráfico del colibrí determinamos **16 puntos** que son las intersecciones entre los polinomios:

- **Punto 1**: intersección entre los polinomios **P1** y **P5**
- **Punto 2**: intersección entre los polinomios **P1** y **P8**
- **Punto 3**: intersección entre los polinomios **P1** y **P9**
- **Punto 4**: intersección entre los polinomios **P1** y **P4**
- **Punto 5**: intersección entre los polinomios **P4** y **P6**
- **Punto 6**: intersección entre los polinomios **P1** y **P6**
- **Punto 7**: intersección entre los polinomios **P1** y **P2**
- **Punto 8**: intersección entre los polinomios **P2** y **P5**
- **Punto 9**: intersección entre los polinomios **P2** y **P6**
- **Punto 10**: intersección entre los polinomios **P3** y **P6**
- **Punto 11**: intersección entre los polinomios **P3** y **P7**
- **Punto 12**: intersección entre los polinomios **P7** y **P8**
- **Punto 13**: intersección entre los polinomios **P2** y **P8**
- **Punto 14**: intersección entre los polinomios **P2** y **P9**
- **Punto 15**: intersección entre los polinomios **P5** y **P9**
- **Punto 16**: intersección entre los polinomios **P5** y **P7**

Para el cálculo de la aproximación de los puntos, utilizaremos los mismos algoritmos de **Bisección y Newton-Raphson** empleados en el ejercicio anterior pero con algunas modificaciones para poder obtener las raíces de los puntos que necesitamos. Se agregaron algunas funciones:

➤**Se importó la biblioteca *mpmath*, la cual proporciona precisión arbitraria para números de punto flotante y permite realizar cálculos con un alto grado de precisión**

```
from mpmath import mp

# Configurar la precisión a 100 dígitos decimales
mp.dps = 100
```

➤**Evaluar una función polinómica Px en un valor x**

```
def f(Px,x):
    Px0 = mp.mpf(float(Px[0]))
    Px1 = mp.mpf(float(Px[1]))
    Px2 = mp.mpf(float(Px[2]))
    Px3 = mp.mpf(float(Px[3]))
    return Px0*x**3 + Px1*x**2 + Px2*x + Px3
```

➤**Recibe como parametro 2 polinomios, los iguala y lo retorna evaluado en un punto x**

```
def interseccion(Pa,Pb,x):
    C1 = mp.mpf(Pa[0]) - mp.mpf(Pb[0])
    C2 = mp.mpf(Pa[1]) - mp.mpf(Pb[1])
    C3 = mp.mpf(Pa[2]) - mp.mpf(Pb[2])
    C4 = mp.mpf(Pa[3]) - mp.mpf(Pb[3])
    return C1*x**3 + C2*x**2 + C3*x +C4
```

➤**Retorna la derivada de la intersección entre 2 polinomios en un punto x**

```
def derivada(Pa,Pb,x):
    C1 = mp.mpf(Pa[0]) - mp.mpf(Pb[0])
    C2 = mp.mpf(Pa[1]) - mp.mpf(Pb[1])
    C3 = mp.mpf(Pa[2]) - mp.mpf(Pb[2])
    C4 = mp.mpf(Pa[3]) - mp.mpf(Pb[3])
    return 3*C1*x**2 + 2*C2*x + C3
```

En el proceso del cálculo de las raíces utilizaremos primero **Bisección** con un máximo de **iteraciones de 50** un como punto de corte una **tolerancia de 0.01**
Consideramos que **0.01** es un punto ideal en el cual **Bisección** no requiere demasiadas iteraciones y se acerca a una distancia muy próxima de la raíz, para luego ser utilizada como entrada y que **Newton-Raphson** no diverga.
Una vez que se aproxime a **0.01,** utilizaremos **Newton-Raphson** con un criterio de corte de **50 iteraciones** o una **tolerancia del residuo de** $1x\mathbf{10^{-30}}$

**x,y,residuo,iteraciones_Newton,iteraciones_Biseccion=biseccion(extremo_A,extremo_b, max_iteraciones,tolerancia,Polinomio_a, Polinomio_b)**

- x1,y1,residuo_1,iteracionesNewton_1,iteracionesBiseccion_1 = biseccion(-16,-14,50,0.01,P1,P5)
- x2,y2,residuo_2,iteracionesNewton_2,iteracionesBiseccion_2 = biseccion(-5,-4,50,0.01,P1,P8)
- x3,y3,residuo_3,iteracionesNewton_3,iteracionesBiseccion_3 = biseccion(-4,-3,50,0.01,P1,P9)
- x4,y4,residuo_4,iteracionesNewton_4,iteracionesBiseccion_4 = biseccion(-2,-1,50,0.01,P1,P4)
- x5,y5,residuo_5,iteracionesNewton_5,iteracionesBiseccion_5 = biseccion(-4,-3,50,0.01,P4,P6)
- x6,y6,residuo_6,iteracionesNewton_6,iteracionesBiseccion_6 = biseccion(0,1,50,0.01,P1,P6)
- x7,y7,residuo_7,iteracionesNewton_7,iteracionesBiseccion_7 = biseccion(1,3,50,0.01,P1,P2)
- x8,y8,residuo_8,iteracionesNewton_8,iteracionesBiseccion_8 = biseccion(1,3,50,0.01,P2,P5)
- x9,y9,residuo_9,iteracionesNewton_9,iteracionesBiseccion_9 = biseccion(2,3,50,0.01,P2,P6)
- x10,y10,residuo_10,iteracionesNewton_10,iteracionesBiseccion_10 = biseccion(5,7,50,0.01,P3,P6)
- x11,y11,residuo_11,iteracionesNewton_11,iteracionesBiseccion_11 = biseccion(3,4,50,0.01,P3,P7)
- x12,y12,residuo_12,iteracionesNewton_12,iteracionesBiseccion_12 = biseccion(14,16,50,0.01,P7,P8)
- x13,y13,residuo_13,iteracionesNewton_13,iteracionesBiseccion_13 = biseccion(2,3,50,0.01,P2,P8)
- x14,y14,residuo_14,iteracionesNewton_14,iteracionesBiseccion_14 = biseccion(1,3,50,0.01,P2,P9)
- x15,y15,residuo_15,iteracionesNewton_15,iteracionesBiseccion_15 = biseccion(-3,-2,50,0.01,P5,P9)
- x16,y16,residuo_16,iteracionesNewton_16,iteracionesBiseccion_16 = biseccion(-6,-5,50,0.01,P5,P7)

***Nota***: La ejecución de **Newton-Raphson** se encuentra dentro de la función de **Bisección**. Una vez que termina, se retorna la raíz de **Newton-Raphson** junto con los demás datos(residuo y números de iteraciones de cada método) desde la misma función de **Bisección.**

## Los puntos y las iteraciones obtenidas son las siguientes:

- **Punto1(-14.54595087970377513407900912742337982782718620126330486732128128357580351574519105555298017991441624,-4.87192505440368567625986404255694969293896307317187762767068719988634631832915421945117926875302565 3)**

**Iteraciones Biseccion(50)**
**Iteraciones Newton(50)**

- **Punto2(-4.586830862604893406081804774387045851010676126463482178640444497444942177508782165576923789451302712,0.25064615734478533701180101225416697569099855160311015630081910456400549302640529666705982214453536 14)**

**Iteraciones Biseccion(7)**
**Iteraciones Newton(3)**

- **Punto3(-3.3282316533749791097216629292577894645593116271153083758653876619562696076341608028849501700012286 41,-0.8223710918508295010987828747445943242198738154794163215464477422252822659719549751788787735552689785)**

**Iteraciones Biseccion(6)**
**Iteraciones Newton(3)**

- **Punto4(-1.6778579081981033186118202587468907291767754549779597353157864278790056029075609566089643028232395 29,-3.2147202489609392427789483180834311522848326518189859256444179061403292884955000772381234620596764 94)**

**Iteraciones Biseccion(7)**
**Iteraciones Newton(4)**

- **Punto5(-3.6841546256547685557706891555007442721708705503360636751072223112906177527510702048909595308126496 58,-7.3523402158504687346009582053521991697346101080159365184285220572729511334190618940876565926897213 22)**

**Iteraciones Biseccion(8)**
**Iteraciones Newton(4)**

- **Punto6(0.9277652989566060484429045671207258814210668507110576085113585462601241059617188316339901326778430749,-9.6640501114171822017770932447629185063058930365465752265396846500473716545868898095628201725411545 08)**

**Iteraciones Biseccion(7)**

**Iteraciones Newton(4)**

- **Punto7(2.291031338384210014018578572368086870071133838840392471619461577266826527444440776578507244893765861,-14.539184555253334928773027312007490127867150611608385585550763121188504954365105566483390614142043 58)**

**Iteraciones Biseccion(10)**
**Iteraciones Newton(3)**

- **Punto8(2.280813089305562305792612315293568944267003788703143275780070552477506327492652006422030737321373764,-13.367091747162430961342724040663379645399085565621021385902072551104984625533081053368738662475287 79)**

**Iteraciones Biseccion(12)**
**Iteraciones Newton(3)**

- **Punto9(2.249474720274980758985660430277078593255808559956693864824127335804243563759284055724853502112554847,-9.982062915333978157510767860513445737519193046333647584258177403914244719003454556450936195270099898)**

**Iteraciones Biseccion(11)**
**Iteraciones Newton(3)**

- **Punto10(6.198050802761689632197190030287643528472948420669904333871105665076292907187227137313781565722652699,-11.618795880021603441392365804819979946274735553257485536202898082044279295475123465897413259503432 97)**

**Iteraciones Biseccion(9)**
**Iteraciones Newton(4)**

- **Punto11(3.784194906790015693835706796353040712144072012916614261139869883200131969261554150146264447191293172,-0.311110306498471712179697355764261217508688457702537279833665555909499370938238273318608633046779162 8)**

**Iteraciones Biseccion(5)**
**Iteraciones Newton(4)**

- **Punto12(14.33203670373925249899939973433793963941302226962720460112917474457647343003043867019574338127997253,12.939704432158379742424267596180574967730870709618056616724953870982350186285720787949615798007246 48)**

**Iteraciones Biseccion(7)**
**Iteraciones Newton(3)**

- **Punto13(2.037296567169569395116167499853112954715376319645918921604577788823009571113947139749910545245243414,5.694171622283284922702369010536442492709858573319042872187049273901022482733098448280044973669825908)**

**Iteraciones Biseccion(9)**
**Iteraciones Newton(4)**

- **Punto14(1.490140083273050885067494642204572932974774790162668368839713273731372527862521085378461442467923 82,14.847084144371099442714954274965774456285889517151839671798353040473243301848592004047079674172780 57)**

**Iteraciones Biseccion(10)**
**Iteraciones Newton(4)**

- **Punto15(-2.460190204533889664440771785993015132227042650467203223406528836138805942443059701303968631810459734,4.62584936594401221935176736284554494356835360886449430879369941078237491842643542832345688941226103 7)**

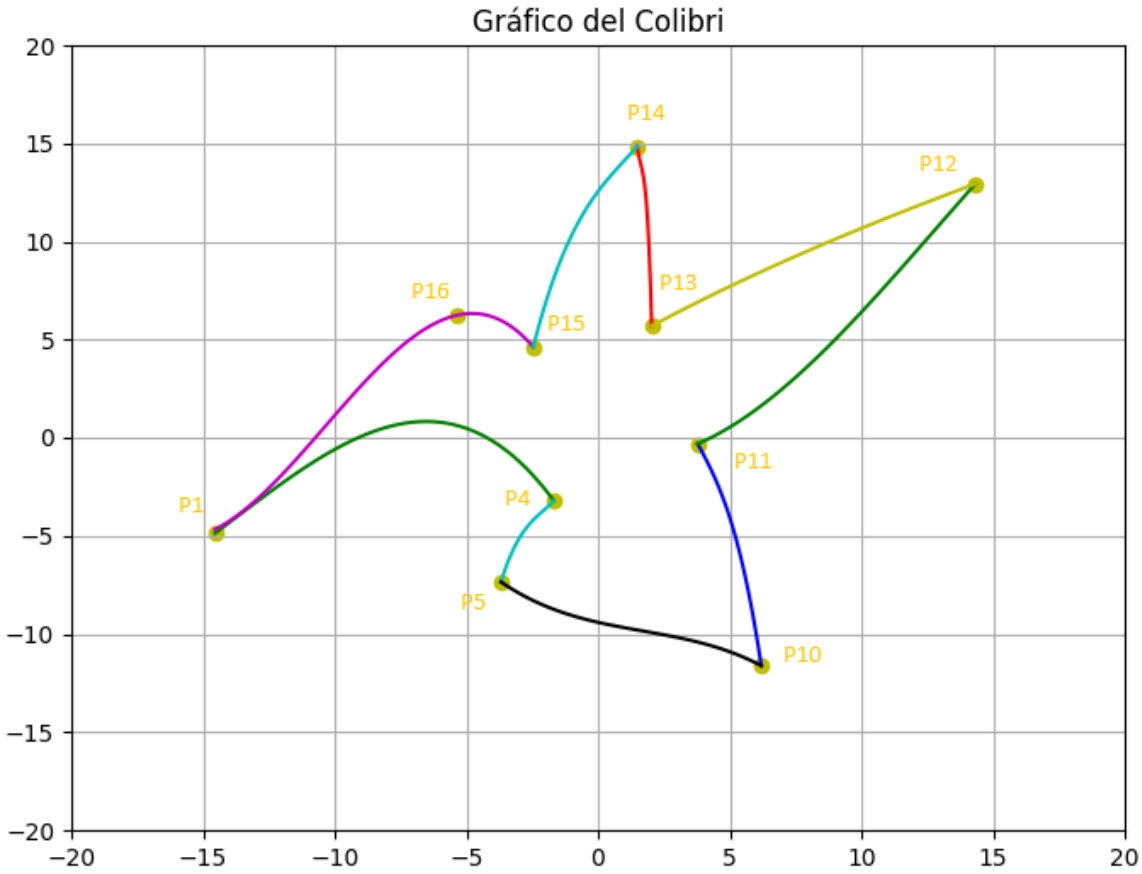
**Iteraciones Biseccion(7)**
**Iteraciones Newton(3)**

- **Punto16(-5.34939684304845507960172387184081672392470895217990081179450946125776075535028422368517470568012817 2,6.24542098064493535025841830860627352136616237019389619744443063710201982236342577410924663903969765 4)**

**Iteraciones Biseccion(7)**
**Iteraciones Newton(4)**

En el **Punto 1** el método de **Bisección** y **Newton-Raphson** gastaron todas las iteraciones máximas. Esto se debe a que en este punto intersección del **polinomio 1 y 5,** no existe una intersección como tal, sino que las curvas pasan muy cerca la una de la otra pero sin cortarse, de modo que los métodos no cortan por la tolerancia establecida.

Por último resta graficar cada uno de los polinomios, solo que esta vez estarán limitados por los puntos anteriormente calculados.




Al momento de graficar el colibrí, notamos que existían puntos que sobraran y que no eran necesarios de calcular para el gráfico. Estos puntos son el **2,3,6,7,8 y 9.**

Igualmente, dejamos el código en el cual se calculan estos puntos sobrantes para su evaluación, pero cabe aclarar que en la práctica esto no debería hacerse, puesto que es fundamental reducir lo máximo posible el costo computacional y el tiempo de ejecución.

Finalmente debemos calcular las distancias, y para ello hay que recorrer el listado con los residuos para evaluar cual es el menor y el mayor. De esta manera podremos determinar cuales son los puntos con la mayor y la menor distancia:

**Distancia mas corta**: 1.365425296150958071047980044999494416418049428071015117965658909736846162434164513126891495111789817x$\mathbf{10}^{-58}$
En el punto P11

**Distancia mas extensa**: 0.200867309929832498306339161692186334533638123765001561478406173542013140900436791536395857125103003 4
En el punto P1

## ***Ejercicio 2)***

• Tolerancia = $1x10^{-15}$ • max_iteraciones=100

• A = [5, -1, 1, 1, 1]
[-1, 6, -1, 1, 1]
[1, -1, 7, -1, 1]
[1, 1, -2, 8, -2]
[1, 1, 1, -3, 9]

• b = [1,1,1,1,1]

Para analizar el rendimiento de ambos métodos, nos enfocaremos en la cantidad de costo computacional que requiere cada uno en hallar una solución con una tolerancia de residuo menor a $1x10^{-15}$

**Jacobi**

| K | -1 | -0.9 | -0.8 | -0.7 | -0.6 | -0.5 | -0.4 | -0.3 | -0.2 | -0.1 | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| It | 49 | 52 | 54 | 56 | 59 | 62 | 65 | 68 | 71 | 75 | 79 | 75 | 71 | 68 | 65 | 62 | 59 | 56 | 54 | 52 | 49 |
| Res | 8.49 e-16 | 5.27 e-16 | 5.96 e-16 | 7.63 e-16 | 6.10 e-16 | 5.27 e-16 | 5.68 e-16 | 5.96 e-16 | 6.93 e-16 | 6.38 e-16 | 6.38 e-16 | 6.38 e-16 | 6.93 e-16 | 5.96 e-16 | 5.68 e-16 | 5.27 e-16 | 6.10 e-16 | 7.63 e-16 | 5.96 e-16 | 5.27 e-16 | 8.46 e-16 |

**Gauss-Seidel**

| K | -1 | -0.9 | -0.8 | -0.7 | -0.6 | -0.5 | -0.4 | -0.3 | -0.2 | -0.1 | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| It | 28 | 30 | 32 | 34 | 36 | 38 | 41 | 44 | 46 | 50 | 53 | 57 | 61 | 65 | 70 | 75 | 82 | 88 | 96 | 105 | 114 |
| Res | 9.71 e-17 | 1.11 e-16 | 1.52 e-16 | 1.80 e-16 | 2.77 | 3.33 | 2.22 e16 | 2.08 e-16 | 4.16 e-16 | 2.63 e-16 | 3.46 e-16 | 2.77 e-16 | 2.91 e-16 | 4.02 e-16 | 3.88 e-16 | 4.71 e-16 | 3.46 e-16 | 4.71 e-16 | 4.71 e-16 | 4.71 e-16 | 5.41 e-16 |

**Solución:** [0.15337423 0.17484663 0.15184049 0.14877301 0.10736196]

Las principales diferencias entre el método de **Jacobi** y el de **Gauss-Seidel** es que el primero en cada iteración, utiliza la información de las variables de la iteración anterior para actualizar todas las variables, mientras que **Gauss-Seidel** en cada iteración actualiza cada variable en función de los valores ya actualizados en esa misma iteración.

Esto genera que el método de Gauss-Seidel pueda converger más rápidamente que el de Jacobi si la matriz de coeficientes es diagonalmente dominante o simétrica definida positiva. Sin embargo, su convergencia no está garantizada en todos los casos.

Para el método de **Jacobi** el valor de **K** más conveniente es el **K=-1 o K=1**, ya que utiliza la menor cantidad de iteraciones para tener una cota de error menor a $\mathbf{1x10^{-15}}$**.**

Por el contrario al utilizar el **K=0** en el promedio ponderado provoca el peor rendimiento en cuanto a cantidad de iteraciones necesarias

En el método de **Gauss-Seidel** el mejor y el peor rendimiento se hallan en los extremos del **K.** El mejor rendimiento lo encontramos en **K=-1** y a medida que lo aumentamos, el rendimiento es peor al punto de llegar al peor rendimiento que está en **K=1**

## ***Ejercicio 3a)***

Dada una matriz **A** de dimensión **10x10** y con un término **p=1**:

$$A = \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 \\ 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 \\ 1/7 & 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 & 1/16 \\ 1/8 & 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 & 1/16 & 1/17 \\ 1/9 & 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 & 1/16 & 1/17 & 1/18 \\ 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 & 1/16 & 1/17 & 1/18 & 1/19 \end{pmatrix}$$

Para demostrar que la matriz es simétrica definida positiva debemos corroborar:

- $A = A^t$ ✓
- Todos los determinantes de las submatrices son mayores a 0

A partir del algoritmo que se encuentra anexado ***simetrica_definida_positiva.py*** calculamos los determinantes de todas las submatrices de la matriz 20x20 y de otras matrices de dimensiones como 30x30 , 40x40 y 50x50 y pudimos comprobar que todas estas tienen todas sus submatrices con determinantes mayores a 0.

Por lo tanto, la matriz es ***Simétrica Definida Positiva.***

## ***Ejercicio 3b)***

Para este inciso utilizaremos un **matriz A** de dimensión **10x10**, un vector **b =[]** con valores aleatorios mediante el método **random.randint** y un valor de **p = 1**

$$A = \begin{matrix} 1 & 1/2 & 1/3 & 1/4 & \dots & \dots & \dots & \dots & 1/9 & 1/10 \\ 1/2 & 1/3 & 1/4 & 1/5 & \dots & \dots & \dots & \dots & 1/10 & 1/11 \\ 1/3 & 1/4 & 1/5 & 1/6 & \dots & \dots & \dots & \dots & 1/11 & 1/12 \\ 1/4 & 1/5 & 1/6 & 1/7 & \dots & \dots & \dots & \dots & 1/12 & 1/13 \\ 1/5 & 1/6 & 1/7 & 1/8 & \dots & \dots & \dots & \dots & 1/13 & 1/14 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 1/9 & 1/10 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 1/19 \\ 1/10 & 1/11 & 1/12 & 1/13 & 1/14 & 1/15 & \dots & \dots & \dots & 1/20 \end{matrix} = \begin{matrix} x \\ x \\ x \\ x \\ x \\ \dots \\ \dots \\ \dots \\ x \\ x \end{matrix}$$

➢ Gauss-Seidel:

- tolerancia: 1e-5
- max iteraciones: 1000
- K : -1

- **Solución**:

[ 131.92958351 -502.65594699 -5110.34672269 5434.92528058
23685.3200434 -10344.519424 -19794.68134494 2019.20853816
-49730.58524088 55581.09984389]

- **Vector b**: [6, 8, 4, 3, 5, 4, 3, 4, 2, 6]

- **Errores**:

Error:(|6 - |5.940309245600474||)= 0.05969075439952576
Error:(|8 - |7.366106863337336||)= 0.6338931366626639
Error:(|4 - |5.924138893690724||)= 1.9241388936907242
Error:(|3 - |3.768668324679311||)= 0.7686683246793109
Error:(|5 - |2.5761570818222026||)= 2.4238429181777974
Error:(|4 - |2.377616655157908||)= 1.622383344842092
Error:(|3 - |2.8667196086303193||)= 0.13328039136968073
Error:(|4 - |3.7573642498027766||)= 0.2426357501972234
Error:(|2 - |4.8456115203894115||)= 2.8456115203894115
Error:(|6 - |6.0||)= 0.0

- **Iteraciones**: 1000

- **Tiempo de ejecución**: 0.40758330000244314 [seg]

- Gradientes Conjugados:
  - tolerancia: 1e-5
  - max iteraciones: 1000

  - **Solución**:

[ 2.20606714e+07 -1.89545645e+09  4.02056701e+10 -3.64355562e+11
  1.73357655e+12 -4.75597915e+12  7.79008403e+12 -7.51762810e+12
  3.94194661e+12 -8.65991514e+11]

  - **VECTOR b**: [8, 8, 4, 5, 3, 1, 8, 9, 2, 6]

  - **Errores**:

Error:(|8 - |7.999908447265625||)= 9.1552734375e-05
Error:(|8 - |7.99993896484375||)= 6.103515625e-05
Error:(|4 - |3.9999923706054688||)= 7.62939453125e-06
Error:(|5 - |4.999946594238281||)= 5.340576171875e-05
Error:(|3 - |2.9999961853027344||)= 3.814697265625e-06
Error:(|1 - |0.9999752044677734||)= 2.47955322265625e-05
Error:(|8 - |7.999973297119141||)= 2.6702880859375e-05
Error:(|9 - |8.999946594238281||)= 5.340576171875e-05
Error:(|2 - |1.9999446868896484||)= 5.53131103515625e-05
Error:(|6 - |6.000001907348633||)= 1.9073486328125e-06

  - **Iteraciones**: 89

  - **Tiempo de ejecución**: 0.00181439999869326135 [seg]

- Factorización LU:
-

  - **Solución**:

[ 3.46745173e+07 -2.92670721e+09  6.11648767e+10 -5.47358925e+11
  2.57630529e+12 -7.00204562e+12  1.13753825e+13 -1.08984112e+13
  5.67808496e+12 -1.24024908e+12]

  - **VECTOR b**: [3, 7, 6, 8, 9, 4, 9, 6, 3, 6]

  - **Errores**:

Error:(|3 - |3.000015258789062||)= 1.52587890625e-05
Error:(|7 - |7.000045776367188||)= 4.57763671875e-05
Error:(|6 - |6.00030517578125||)= 0.00030517578125
Error:(|8 - |8.000045776367188||)= 4.57763671875e-05
Error:(|9 - |8.999969482421875||)= 3.0517578125e-05
Error:(|4 - |4.000030517578125||)= 3.0517578125e-05
Error:(|9 - |9.00018310546875||)= 0.00018310546875
Error:(|6 - |6.000045776367188||)= 4.57763671875e-05
Error:(|3 - |3.0||)= 0.0

Error:(|6 - |6.0000457763671875||)= 4.57763671875e-05

- **Tiempo de ejecución**: 0.0004442999997991137 [seg]

Luego de ejecutar los 3 métodos con la matriz dada, podemos observar como el método menos eficiente es **Gauss-Seidel** ya que el error que arroja es muy superior al de los otros dos. Además el tiempo de ejecución es demasiado largo consumiendo todas las iteraciones disponibles.

El método de **Gradientes Conjugados** se comportó de manera más eficiente que **Gauss-Seidel** ya que logró una aproximación mucho más precisa con menos iteraciones.

Y por último, el método que funcionó con el menor costo computacional, fue la **Factorización LU** con un tiempo de ejecución muy acotado y cotas de error similares a **Gradientes Conjugados:** $1x10^{-5}$**.**

## ***Ejercicio 3c)***

Luego de ejecutar nuevamente los tres algoritmos y modificando el **p** = $\frac{\sqrt{5}-1}{2} \simeq 0.61803398\ldots$ pudimos observar cómo el método de **Gauss-Seidel** comienza a diverger rápidamente

➢ Gauss-Seidel:

- **Solución**:

[-1.49611914e+33  4.68870534e+33 -3.03757300e+33 -6.89346516e+32
 -7.72975256e+31  9.64096114e+31  1.37148203e+32  1.34879645e+32
  1.19527936e+32  1.01613840e+32]

- **VECTOR b**: [3, 8, 4, 6, 6, 8, 9, 7, 5, 8]

- **Errores**:

Error:(|3 - |1.1416549762412716e+32||)= 1.1416549762412716e+32
Error:(|8 - |-6.6103396305577175e+31||)= 6.6103396305577175e+31
Error:(|4 - |-5.089451546575267e+30||)= 5.089451546575267e+30
Error:(|6 - |3.2856751382951446e+30||)= 3.2856751382951446e+30
Error:(|6 - |3.650124542625266e+30||)= 3.650124542625266e+30
Error:(|8 - |2.709898375166913e+30||)= 2.709898375166913e+30
Error:(|9 - |1.743191748167811e+30||)= 1.743191748167811e+30
Error:(|7 - |9.761106811543991e+29||)= 9.761106811543991e+29
Error:(|5 - |4.089099199608337e+29||)= 4.089099199608337e+29
Error:(|8 - |0.0||)= 8.0

- **Iteraciones**: 1000

- **Tiempo de ejecución**: 0.38541609999811044 [seg]

➢ Gradientes conjugados:

- **Solución:**

[ 6.61916788e+05  2.62491925e+08 -8.12899720e+09  8.52871555e+10
 -4.36632907e+11  1.24746808e+12 -2.09164062e+12  2.04584613e+12
 -1.08070452e+12  2.38245789e+11]

- **Vector b:** [5, 5, 4, 2, 1, 8, 1, 2, 9, 8]

- **Error**:

Error:(|5 - |4.9998321533203125||)= 0.0001678466796875
Error:(|5 - |4.999881744384766||)= 0.000118255615234375
Error:(|4 - |3.9998741149902344||)= 0.000125885009765625

Error:(|2 - |1.9999046325683594||)= 9.5367431640625e-05
Error:(|1 - |0.9998779296875||)= 0.0001220703125
Error:(|8 - |7.999879837036133||)= 0.0001201629638671875
Error:(|1 - |0.9998836517333984||)= 0.0001163482666015625
Error:(|2 - |1.9999122619628906||)= 8.7738037109375e-05
Error:(|9 - |8.999900817871094||)= 9.918212890625e-05
Error:(|8 - |7.999931335449219||)= 6.866455078125e-05

- **Iteraciones**: 72
- **Tiempo de ejecución**: 0.0016619999951217324 [seg]

Factorización LU y Gradientes Conjugados se comportan de manera similar al caso previo donde p=1

Podemos concluir que este método diverge dado que al hacer que **p** valga $\frac{\sqrt{5}-1}{2} \simeq 0.61803398..$ y no 1, como en el inciso anterior, introducimos un valor el cual tendrá un impacto en los errores de redondeo, problema el cual Gauss-Seidel le afecta mucho puesto que ahora cada valor de la matriz estará dividido por un número con varios dígitos decimales en lugar de un entero y esto se acarrea en cada iteración.