# Tarea 2: Métodos Numéricos

Rafael Alejandro García Ramírez

5 de septiembre de 2023

1. Escribe un programa para calcular la constante matemática $e$, considerando la definición

$$e = \lim_{n\to\infty} (1 + 1/n)^n$$

es decir, calcula $(1 + 1/n)^n$ para $n = 10^k, k = 1, 2, \ldots, 20$. Determina el error relativo y el error absoluto de las aproximaciones comparándolas con $exp(1)$.

Recordemos que para $p$ el valor de teórico o real y $p^*$ nuestro valor numérico:

$$\epsilon_{\text{relativo}} = \frac{|p - p^*|}{|p|}, \quad \epsilon_{\text{absoluto}} = |p - p^*|$$

Dado que tenemos valores bastante grandes para $n$ recordemos el limite

$$\lim_{n\to\infty} \frac{1}{n} = 0$$

Por lo tanto para un numero muy grande la función original se hace

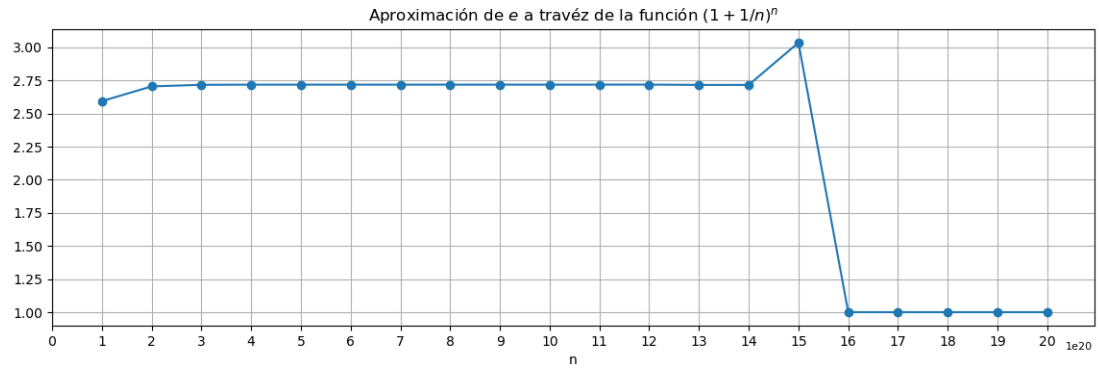
$$e \sim \left(1 + \frac{1}{n}\right)^n \approx 1^n = 1$$

Esto para algún $n$ que sea

$$\frac{1}{n} \leq \epsilon_{\text{machine}}$$

Entonces en algún punto de las evaluaciones los valores se desplomarán a cero. Creamos una función que evalué esta función para los valores especificados en el problema, después graficamos cada valor:

```
def euler_aproximacion(n):
    return (1 + n**-1)**n

datos = [euler_aproximacion(10**i) for i in range(1,21)]

plt.figure(figsize=(14, 4))
plt.plot(range(1,21), datos, marker = "o")
plt.annotate('1e20', xy=(0.985, -0.08), xycoords='axes fraction', fontsize=8, ha='center')
plt.xticks(range(0,21))
plt.title("Aproximacion de $e$ a traves de la funcion $(1 + 1/n)^{n}$")
plt.xlabel("n")
plt.grid()
plt.show()
```

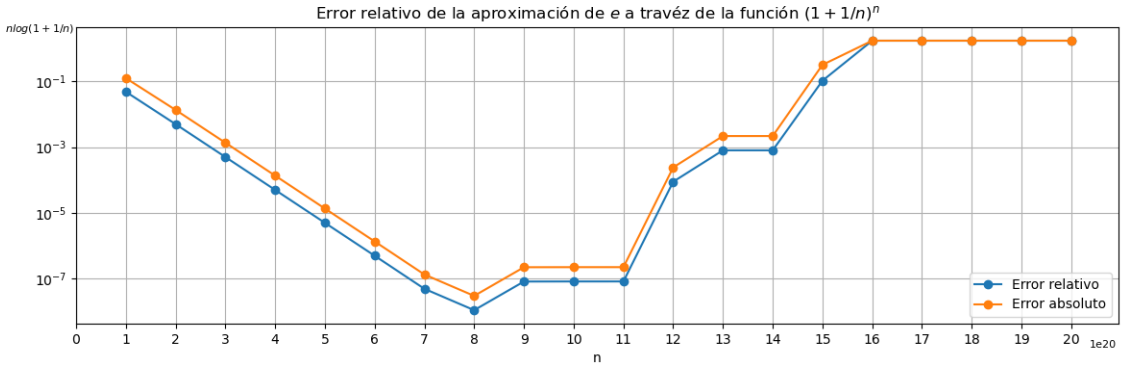



Graficamos el error relativo y el error absoluto calculando los valores previamente en una lista:

```
error_relativo = [abs(np.exp(1) - dato) / abs(dato) for dato in datos]
error_absoluto = [abs(np.exp(1) - dato) for dato in datos]
plt.figure(figsize=(14, 4))
plt.plot(range(1,21), error_relativo, marker = "o", label = "Error relativo")
plt.plot(range(1,21), error_absoluto, marker = "o", label = "Error absoluto")
plt.annotate('1e20', xy=(0.985, -0.08), xycoords='axes fraction', fontsize=8, ha='center')
plt.xticks(range(0,21))
plt.title("Error relativo de la aproximacion de $e$ a travez de la funcion $(1 + 1/n)^{n}$")
plt.xlabel("n")
plt.legend(); plt.grid(); plt.show()
```


Desde esta perspectiva pareciera que los ambos errores tienen la misma tendencia desde los valores 2 hasta 14, por lo que vamos a escalar logarítmicamente los valores para destacar cualquier ligera diferencia :

```
plt.figure(figsize=(14, 4))
plt.plot(range(1,21), error_relativo, marker = "o", label = "Error relativo")
plt.plot(range(1,21), error_absoluto, marker = "o", label = "Error absoluto")
plt.annotate('1e20', xy=(0.985, -0.08), xycoords='axes fraction', fontsize=8, ha='center')
plt.annotate('$n log(1 + 1/n)$', xy=(-0.035, 0.985), xycoords='axes fraction', fontsize=8, ha='
    center')
plt.xticks(range(0,21))
plt.title("Error relativo de la aproximacion de $e$ a travez de la funcion $(1 + 1/n)^{n}$")
plt.xlabel("n")
plt.yscale("log")
plt.legend(); plt.grid(); plt.show()
```




Al analizar el gráfico junto con los valores presentados, se observa que el error relativo es inferior al error absoluto. Esta observación es coherente, ya que el numerador del error relativo resulta de dividir el error absoluto entre el valor real. Esta tendencia sugiere un buen ajuste de los datos y una relación entre el error y el valor real que es considerablemente baja.

Al explorar la representación logarítmica de la gráfica, se destaca que el valor óptimo se alcanza cuando $n = 8$. Este resultado contrasta con lo que se podría esperar, es decir, que para una aproximación más precisa se requieran valores más elevados de acuerdo con la definición que estamos considerando. En la tabla subsiguiente se detallan con mayor claridad los valores correspondientes a cada punto de las gráficas.

| Valor n | Aproximacion | Error relativo | Error Absoluto |
|---|---|---|---|
| 1e1 | 2.593742 | 4.801532e-02 | 1.245394e-01 |
| 1e2 | 2.704814 | 4.979270e-03 | 1.346800e-02 |
| 1e3 | 2.716924 | 4.997918e-04 | 1.357896e-03 |
| 1e4 | 2.718146 | 4.999792e-05 | 1.359016e-04 |
| 1e5 | 2.718268 | 4.999973e-06 | 1.359127e-05 |
| 1e6 | 2.718280 | 5.000821e-07 | 1.359363e-06 |
| 1e7 | 2.718282 | 4.941613e-08 | 1.343270e-07 |
| 1e8 | 2.718282 | 1.107747e-08 | 3.011169e-08 |
| 1e9 | 2.718282 | 8.224037e-08 | 2.235525e-07 |
| 1e10 | 2.718282 | 8.269037e-08 | 2.247757e-07 |
| 1e11 | 2.718282 | 8.273537e-08 | 2.248981e-07 |
| 1e12 | 2.718523 | 8.889663e-05 | 2.416676e-04 |
| 1e13 | 2.716110 | 7.995973e-04 | 2.171794e-03 |
| 1e14 | 2.716110 | 7.995973e-04 | 2.171794e-03 |
| 1e15 | 3.035035 | 1.043656e-01 | 3.167534e-01 |
| 1e16 | 1.000000 | 1.718282e+00 | 1.718282e+00 |
| 1e17 | 1.000000 | 1.718282e+00 | 1.718282e+00 |
| 1e18 | 1.000000 | 1.718282e+00 | 1.718282e+00 |
| 1e19 | 1.000000 | 1.718282e+00 | 1.718282e+00 |
| 1e20 | 1.000000 | 1.718282e+00 | 1.718282e+00 |

---

2. La ecuación $x^3 + x = 6$ tiene una raíz en el intervalo [1,55, 1,75]. ¿Cuántas iteraciones se necesitan para obtener una aproximación de la raíz con error menor a 0,0001 con el método de bisección? Verifica con el método de la bisección tu predicción de la raíz.

   Tenemos que las diferencias entre el el punto $p$ y el calculado es

$$|p_n - p| \leq \frac{b-a}{2^N} < 10^{-4}$$

   para $N \geq 1$. Entonces para $a = 1{,}55$ y $b = 1{,}75$, sustutiyendo tenemos

$$\frac{0{,}2}{2^N} < 10^{-4} \rightarrow 2^N > (0{,}2)10^4 = 2000$$

   Esto es

$$N > \frac{log(2000)}{log(2)} \approx 10{,}96$$

   Y dado que $N$ tiene que se mayor a esta cantidad tenemos que $N = 11$. Para poder demostrar esto vamo a programar el método de la bisección y contar la candidad de iteraciones que se requiere para llegar al resultado dentro del nivel de tolerancia:

```python
def metodo_biseccion(inicio_intervalo, fin_intervalo, tolerancia, max_iteraciones, funcion_objetivo):
    """
    Calcula la raiz de una funcion mediante el m todo de bisecci n.

    Parametros:
    inicio_intervalo (float): Extremo izquierdo del intervalo inicial.
    fin_intervalo (float): Extremo derecho del intervalo inicial.
    tolerancia (float): Tolerancia para detener las iteraciones cuando el intervalo sea lo suficientemente pequenio.
    max_iteraciones (int): N mero maximo de iteraciones permitidas.
    funcion_objetivo (function): La funcion para la cual se busca la raiz.

    Retorna:
    tuple: Una tupla (raiz, iteraciones) que contiene la raiz aproximada encontrada y el n mero de iteraciones realizadas.
           En caso de fracaso, retorna un mensaje indicando que el m todo no tuvo  xito .
    """
    valor_inicio = funcion_objetivo(inicio_intervalo)

    for iteracion in range(1, int(max_iteraciones + 1)):
        punto_medio = inicio_intervalo + (fin_intervalo - inicio_intervalo) / 2
        valor_punto_medio = funcion_objetivo(punto_medio)
```

```
        if valor_punto_medio == 0 or abs(fin_intervalo - inicio_intervalo) / 2 < tolerancia:
            return punto_medio, iteracion

        if valor_inicio * valor_punto_medio > 0:
            inicio_intervalo = punto_medio
            valor_inicio = valor_punto_medio
        else:
            fin_intervalo = punto_medio

    return "El metodo no tuvo exito en encontrar una raiz dentro del intervalo dado."

def funcion1(x):
    return x**3 + x - 6


raiz, cantidad_iteraciones = metodo_biseccion(inicio_intervalo = 1.55, fin_intervalo = 1.75,
                                              tolerancia = 1e-4, max_iteraciones = 10**5,
                                              funcion_objetivo = funcion1)
print(f"La raiz es: {raiz}, que se encontro en la iteracion: {cantidad_iteraciones}.")
```

```
La raíz es: 1.63427734375, que se encontró en la iteración: 11.
```

Confirmamos que efectivamente la cantidad de iteraciones necesarias son 11.

---

3. Hallar una raíz de de $f(x) = x^4 + 3x^2 - 2$ por medio de las siguientes 4 formulaciones de punto fijo utilizando $p_0 = 1$:

$$a) \quad = \sqrt{\frac{2 - x^4}{3}}, \quad b) \quad x = (2 - 3x^2)^{\frac{1}{4}}, \quad c) \quad x = \frac{2 - x^4}{3x}, \quad d) \quad x = \left(\frac{2 - 3x^2}{x}\right)^{\frac{1}{3}}$$

   - Las raíces de $f(x)$ de coincidir con las raíces de $x = g(x)$. Gráfica $f(x)$ y $x - g(x)$. Comenta lo observado.
   - Crea una tabla comparativa para comparar el resultado de las raíces el resultado de las raíces de $f(x)$ con la raíz alcanzada con cada una de las 4 formulaciones. Usa máximo 20 iteraciones y tol = 0,0001. Explica lo sucedido.

   Comenzaremos creando el código para el método tomando precauciones para cada tipo de error debido a la naturaleza del método, esto es tomando en cuando que el proceso puede divergir, comenzar a buscar alguna raíz imaginaria o simplemente no completarse en el número de iteraciones máxima.

```
def punto_fijo(punto_inicial, tolerancia, max_iteraciones, funcion_objetivo):
    """
    Encuentra una aproximacion de la raiz de una funcion utilizando el metodo de punto fijo.

    Parametros:
    punto_inicial (float): Valor inicial para comenzar las iteraciones.
    tolerancia (float): Valor minimo de cambio en el resultado para considerar convergencia.
    max_iteraciones (int): Numero maximo de iteraciones permitidas.
    funcion_objetivo (function): Funcion objetivo para la cual se busca la raiz.

    Retorna:
    tuple: Una tupla que contiene:
        - float: Aproximacion de la raiz encontrada.
        - int: Numero de iteraciones realizadas.
        - list: Lista de valores aproximados en cada iteracion.
        - En caso de error de Overflow:
            - None: Indicador de error.
            - str: Mensaje de error.
            - list: Lista de valores aproximados hasta el momento del error.
    """

    datos = []
    for iteracion in range(max_iteraciones):
        try:
            punto_nuevo = funcion_objetivo(punto_inicial)
            datos.append(punto_nuevo)
        except OverflowError:
            return None, "Error de overflow. El metodo no tuvo exito debido a un calculo que
    resulto en un valor demasiado grande.", datos

        if abs(punto_nuevo - punto_inicial) < tolerancia:
            return punto_nuevo, iteracion + 1, datos
        else:
            punto_inicial = punto_nuevo
```

```
    return f"El metodo fallo despues de {max_iteraciones} iteraciones.", f"Ultimo valor: {
    punto_nuevo}.", datos
```

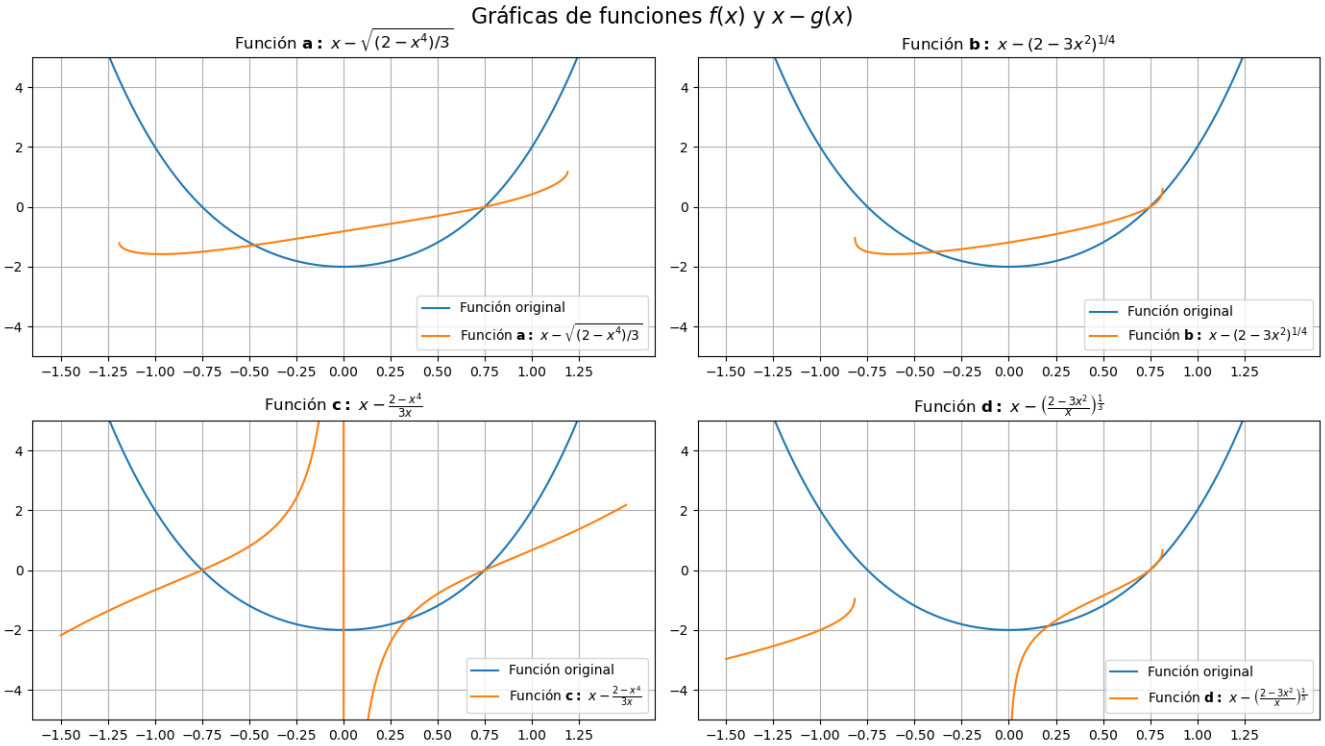
Para los parámetros que nos especifica el problema, haremos el calculo después de definir cada función.

```
funcion_original = lambda x: x**4 + 3*x**2 - 2
funcion_a = lambda x: ((2 - x**4) / 3)**0.5
funcion_b = lambda x: (2 - 3*(x**2))**0.25
funcion_c = lambda x: ((2 - x**4) / (3*x))
funcion_d = lambda x: ((2 - 3*x**2) / x)**(1/3)

raiz_a, iteraciones_a, datos_a = punto_fijo(punto_inicial = 1, tolerancia = 0.0001,
    max_iteraciones =  100, funcion_objetivo = funcion_a)
raiz_b, iteraciones_b, datos_b = punto_fijo(punto_inicial = 1, tolerancia = 0.0001,
    max_iteraciones =  100, funcion_objetivo = funcion_b)
raiz_c, iteraciones_c, datos_c = punto_fijo(punto_inicial = 1, tolerancia = 0.0001,
    max_iteraciones =  100, funcion_objetivo = funcion_c)
raiz_d, iteraciones_d, datos_d = punto_fijo(punto_inicial = 1, tolerancia = 0.0001,
    max_iteraciones =  100, funcion_objetivo = funcion_d)

print(f"La raiz de la funcion a es: {raiz_a} - Iteraciones = {iteraciones_a}")
print(f"La raiz de la funcion b es: {raiz_b} - Iteraciones = {iteraciones_b}")
print(f"La raiz de la funcion c es: {raiz_c} - Iteraciones = {iteraciones_c}")
print(f"La raiz de la funcion d es: {raiz_d} - Iteraciones = {iteraciones_d}")
```

```
La raíz de la funcion a es: 0.7493865308123768 - Iteraciones = 10
La raíz de la funcion b es: El método falló después de 100 iteraciones.
    - Iteraciones = Último valor: (1.4397484695363614-0.7568854969757942j).
La raíz de la funcion c es: None
    - Iteraciones = Error de overflow. El método no tuvo éxito debido a un
      cálculo que resultó en un valor demasiado grande.
La raíz de la funcion d es: El método falló después de 100 iteraciones.
    - Iteraciones = Último valor: (1.2909944487358054-1.1547005383792512j).
```

Parece ser que solamente la reformulacion $a$ converge, mientras que las reformulaciones $b$ y $d$ se van a valores imaginarios y la reformulacion $c$ diverge completamente. Grafiquemos cada reformulacion con la formulacion original para apreciar el comportamiento de cada una de ellas

```
funcion_original = lambda x: x**4 + 3*x**2 - 2
funcion_a = lambda x: x - ((2 - x**4) / 3)**0.5
funcion_b = lambda x: x - (2 - 3*(x**2))**0.25
funcion_c = lambda x: x - ((2 - x**4) / (3*x))
funcion_d = lambda x: x - ((2 - 3*x**2) / x)**(1/3)

x = np.arange(-1.5,1.5,0.001)

a = [funcion_a(i) for i in x]
b = [funcion_b(i) for i in x]
c = [funcion_c(i) for i in x]
d = [funcion_d(i) for i in x]


yOriginal = [funcion_original(i) for i in x]

funciones = [a,b,c,d]
nombres = ["Funcion $\mathbf{a:}$ $x - \\sqrt{(2 - x^{4})/3}$", "Funcion $\mathbf{b:}$ $x - (2 - 3
    x^{2})^{1/4}$", "Funcion $\mathbf{c:}$ $x - \\frac{2 - x^{4}}{3x}$", "Funcion $\mathbf{d:}$ $x
     - \\left ( \\frac{2 - 3x^{2}}{x} \\right )^{\\frac{1}{3}}$"]
plt.figure(figsize=(14, 8))

for i, (y_func, label) in enumerate(zip(funciones, nombres), start=1):
    plt.subplot(2, 2, i)
    plt.plot(x, yOriginal, label="Funcion original")
    plt.plot(x, y_func, label=label)
    plt.xticks(np.arange(-1.5,1.5,0.25))
    plt.ylim(-5, 5)
    plt.title(label)
    plt.legend(loc = "lower right")
    plt.grid()

plt.tight_layout()
plt.suptitle("Graficas de funciones $f(x)$ y $x - g(x)$", fontsize = 16)
plt.subplots_adjust(top=0.91)
plt.show()
```




Se puede observar que definitivamente existe una raíz alrededor de $x \approx 0{,}75$, y en las cuatro formulaciones la función comparte la misma raíz que la función original, siendo que la función $c$ comparte ambas raíces reales. Sin embargo, es el método el que conduce a la divergencia en el caso de la función $c$, así como a la búsqueda de raíces imaginarias en las funciones $b$ y $d$. Además, la función $c$ introduce una singularidad en $x = 0$. Aunque este no sea el caso, podría eliminar alguna raíz dependiendo de la función original, además de que complica sustancialmente el cálculo.

Mostraremos entonces algunos valores encontrados durante el cálculo para la tolerancia $tol = 0{,}0001$ y con un máximo de 20 iteraciones:

| Iteracion | Funcion a | Funcion b | Funcion c | Funcion d |
|---|---|---|---|---|
| 1 | 0.57735 | (0.70711+0.70711j) | 0.33333 | (0.5+0.86603j) |
| 2 | 0.79349 | (1.3366-0.33517j) | 1.98765 | (1.38234-0.87036j) |
| 3 | 0.73111 | (1.1675+0.80505j) | -2.28218 | (1.17743+1.15881j) |
| 4 | 0.75593 | (1.42002-0.5989j) | 3.67003 | (1.30626-1.11802j) |
| 5 | 0.74688 | (1.34927+0.7808j) | -16.29574 | (1.27701+1.15578j) |
| 6 | 0.7503 | (1.43521-0.70374j) | 1442.41051 | (1.29292-1.15019j) |
| 7 | 0.74902 | (1.4104+0.76551j) | -1000334796.86732 | (1.28929+1.15484j) |
| 8 | 0.7495 | (1.43849-0.73956j) | 3.3366824230210896e+26 | (1.29123-1.15415j) |
| 9 | 0.74932 | (1.43029+0.75976j) | -1.2382928520514783e+79 | (1.29079+1.15472j) |
| 10 | 0.74939 | (1.43937-0.75129j) | | (1.29102-1.15463j) |
| 11 | | (1.43671+0.75782j) | | (1.29097+1.1547j) |
| 12 | | (1.43963-0.75508j) | | (1.291-1.15469j) |
| 13 | | (1.43877+0.75719j) | | (1.29099+1.1547j) |
| 14 | | (1.43971-0.75631j) | | (1.29099-1.1547j) |
| 15 | | (1.43943+0.75698j) | | (1.29099+1.1547j) |
| 16 | | (1.43974-0.7567j) | | (1.29099-1.1547j) |
| 17 | | (1.43965+0.75692j) | | (1.29099+1.1547j) |
| 18 | | (1.43974-0.75683j) | | (1.29099-1.1547j) |
| 19 | | (1.43972+0.7569j) | | (1.29099+1.1547j) |
| 20 | | (1.43975-0.75687j) | | (1.29099-1.1547j) |

En lo general, estos resultados se deben a las características de cada función, pues en las formulaciones $b$ y $d$ se tiene una expresión de la forma $\sim \sqrt{a - bx^n}$ , lo que hace que justo cuando a $a > bx^n$ los resultados se hagan complejos. Mientras que en el caso $c$ la función tiene el comportamiento $\sim x^4$ lo que que hace que al momento de revalorar la expresión se tenga ahora este valor de nuevo elevado a la cuarta, lo que hace que muy rápidamente se escape del valor máximo representable por la máquina. Por último, la función $a$ no presenta este problema dado al punto inicial $p_0 = 1$ que se otorga pues si se le otorgara un valor inicial mayor o igual a $\sqrt[4]{2}$ esta función comenzaría

lléndose a los imaginarios o se haría cero; y a la fracción que contiene, esto hará que los valores no tiendan fuera de 2 y no se hagan complejos.

---

4. Utiliza el método de bisección, método de newton, método de la secante y método de la falsa posición para comparar los resultados del siguiente problema:

   Encontrar $\lambda$ con una precisión de $10^{-4}$ y $N_{iter,max} = 100$, para la ecuación de la población en términos de la tasa de natalidad $\lambda$,

   $$P(\lambda) = 1,000,000e^{\lambda} + \frac{435,000}{\lambda}(e^{\lambda} - 1)$$

   para $P(\lambda) = 1,564,000$ in dividuos por años. Usa $\lambda_0 = 0{,}1$. (Sugerencia: graficar $P(\lambda) - N$)

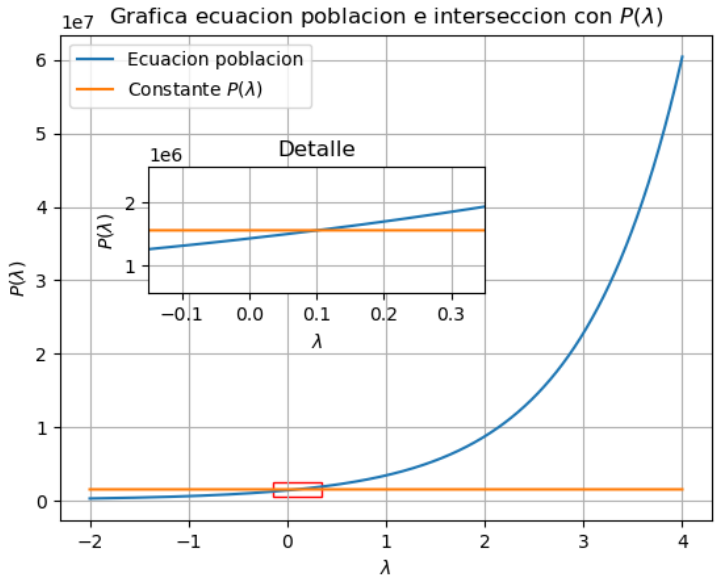
   Comenzamos creando una función para cada método restante que no hemos hecho aún:

```python
def metodo_newton(punto_inicial, tolerancia, max_iteraciones, funcion_objetivo, funcion_derivada):
    """
    Calcula la raiz de una funcion mediante el metodo de Newton.

    Parametros:
    punto_inicial (float): Valor inicial para comenzar el metodo.
    tolerancia (float): Tolerancia para detener las iteraciones cuando la aproximacion sea lo suficientemente cercana.
    max_iteraciones (int): Numero maximo de iteraciones permitidas.
    funcion_objetivo (function): La funcion para la cual se busca la raiz.
    funcion_derivada (function): La derivada de la funcion objetivo.

    Retorna:
    tuple: Una tupla (raiz, iteraciones) que contiene la raiz aproximada encontrada y el numero de iteraciones realizadas.
           En caso de fracaso, retorna un mensaje indicando que el metodo no tuvo exito.
    """
    datos = []
    for iteracion in range(max_iteraciones):
        punto_nuevo = punto_inicial - (funcion_objetivo(punto_inicial) / funcion_derivada(punto_inicial))
        datos.append(punto_nuevo)
        if abs(punto_nuevo - punto_inicial) < tolerancia:
            return punto_nuevo, iteracion + 1, datos
        else:
            punto_inicial = punto_nuevo

    return f"El metodo de Newton fallo despues de {max_iteraciones} iteraciones."

def metodo_secante(punto_0, punto_1, tolerancia, max_iteraciones, funcion_objetivo):
    """
    Calcula la raiz de una funcion mediante el metodo de la secante.

    Parametros:
    punto_0 (float): Primer valor inicial.
    punto_1 (float): Segundo valor inicial.
    tolerancia (float): Tolerancia para detener las iteraciones cuando la aproximacion sea lo suficientemente cercana.
    max_iteraciones (int): Numero maximo de iteraciones permitidas.
    funcion_objetivo (function): La funcion para la cual se busca la raiz.

    Retorna:
    tuple: Una tupla (raiz, iteraciones) que contiene la raiz aproximada encontrada y el numero de iteraciones realizadas.
           En caso de fracaso, retorna un mensaje indicando que el metodo no tuvo exito.
    """
    q0 = funcion_objetivo(punto_0)
    q1 = funcion_objetivo(punto_1)
    datos = []

    for iteracion in range(max_iteraciones):
        punto_nuevo = punto_1 - q1 * ((punto_1 - punto_0) / (q1 - q0))
        datos.append(punto_nuevo)
        if abs(punto_nuevo - punto_1) < tolerancia:
            return punto_nuevo, iteracion + 1, datos
        else:
            punto_0 = punto_1
            q0 = q1
            punto_1 = punto_nuevo
            q1 = funcion_objetivo(punto_nuevo)

    return f"El metodo de Secante fallo despues de {max_iteraciones} iteraciones."
```

```python

def metodo_falsa_posicion(punto_0, punto_1, tolerancia, max_iteraciones, funcion_objetivo):
    """
    Calcula la raiz de una funcion mediante el metodo de falsa posicion.

    Parametros:
    punto_0 (float): Primer valor inicial.
    punto_1 (float): Segundo valor inicial.
    tolerancia (float): Tolerancia para detener las iteraciones cuando la aproximacion sea lo suficientemente cercana.
    max_iteraciones (int): Numero maximo de iteraciones permitidas.
    funcion_objetivo (function): La funcion para la cual se busca la raiz.

    Retorna:
    tuple: Una tupla (raiz, iteraciones) que contiene la raiz aproximada encontrada y el numero de iteraciones realizadas.
           En caso de fracaso, retorna un mensaje indicando que el metodo no tuvo exito.
    """
    q0 = funcion_objetivo(punto_0)
    q1 = funcion_objetivo(punto_1)
    datos = []

    for iteracion in range(max_iteraciones - 1):
        punto_nuevo = punto_1 - q1 * ((punto_1 - punto_0) / (q1 - q0))
        datos.append(punto_nuevo)
        if abs(punto_nuevo - punto_1) < tolerancia:
            return punto_nuevo, iteracion + 1, datos

        q_nuevo = funcion_objetivo(punto_nuevo)

        if q_nuevo * q1 < 0:
            punto_0 = punto_1
            q0 = q1

        punto_1 = punto_nuevo
        q1 = q_nuevo

    return f"El metodo Falsa Posicion fallo despues de {max_iteraciones} iteraciones."
```

Dado que el método de la bisección ocupa un intervalo donde se encuentre la raíz y el método de la secante y posición falsa requieren de dos puntos, graficaremos previamente las intersecciones de de ambos valores para encontrar un estimado. Después de jugar un poco con los intervalos de la función, podemos encontrar una zona donde se intesectan ambas rectas:

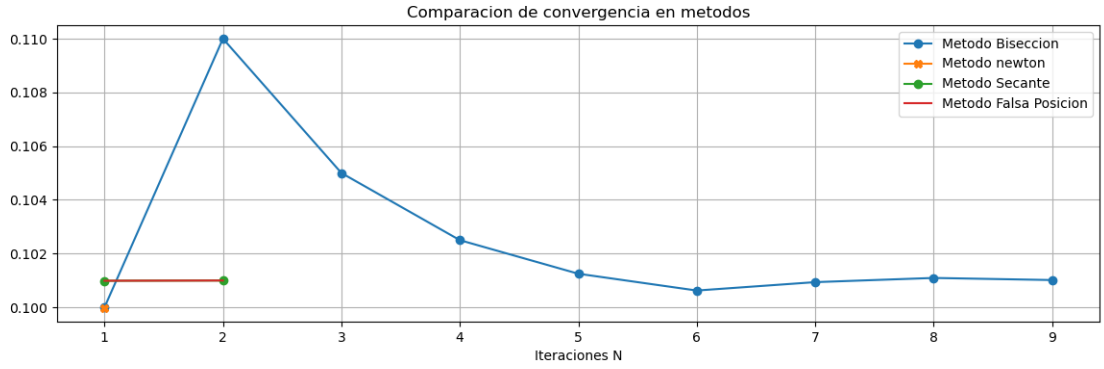
```python
def ecuacion_poblacion(lambd):
    return (1e6 * np.exp(lambd)) + ((np.exp(lambd) - 1)*(1e3*435 / lambd))

valores_grafica = np.arange(-2, 4, 1e-5)
data_grafica = [ecuacion_poblacion(i) for i in valores_grafica]

fig, ax_main = plt.subplots()
ax_main.plot(valores_grafica, data_grafica, label="Ecuacion poblacion")
ax_main.plot(valores_grafica, [1564000]*len(valores_grafica), label=f"Constante $P(\\lambda)$")
ax_main.set_title(f"Grafica ecuacion poblacion e interseccion con $P(\\lambda)$")
ax_main.set_xlabel(f"$\\lambda$")
ax_main.set_ylabel(f"$P(\\lambda)$")
ax_main.legend()
ax_main.grid()

# Definimos la region de la lupa
x_zoom, y_zoom = 0.1, 1564000  # Coordenadas del punto de zoom
width, height = 0.5, 2000000    # Tamanio de la lupa
rect = plt.Rectangle((x_zoom - width / 2, y_zoom - height / 2), width, height, edgecolor='red', facecolor='none')
ax_main.add_patch(rect)

# Creamos la region de la lupa
ax_zoom = plt.axes([0.23, 0.47, 0.4, 0.2]) # Coordenadas de impresion
ax_zoom.plot(valores_grafica, data_grafica, label="Ecuacion poblacion")
ax_zoom.plot(valores_grafica, [1564000]*len(valores_grafica), label=f"Constante $P(\\lambda)$")
ax_zoom.set_xlim(x_zoom - width / 2, x_zoom + width / 2)
ax_zoom.set_ylim(y_zoom - height / 2, y_zoom + height / 2)
ax_zoom.grid()
ax_zoom.set_title("Detalle")
ax_zoom.set_xlabel(f"$\\lambda$")
ax_zoom.set_ylabel("$P(\\lambda)$")
plt.savefig("graficos41.png")

plt.show()
```




Como se puede apreciar en el gráfico de la izquierda, la función presenta una raíz que se encuentra ubicada aproximadamente en el intervalo de estimación $[0{,}08, 0{,}12]$. Aunque parece indicar que esta raíz podría estar en el punto 0,1, o al menos en sus cercanías, no podemos aún considerarlo como una solución definitiva. Sin embargo, reconocemos la inclinación de considerarlo como nuestro punto inicial $\lambda_0$. Por lo tanto, establecemos este rango como el intervalo en el cual buscaremos la solución de la ecuación.

Con este enfoque en mente, emplearemos las funciones que hemos definido previamente y generaremos una representación gráfica que ilustre la evolución de la solución a lo largo de las iteraciones requeridas por cada método:

```python
def ecuacion_poblacion(lambd): # modificamos la funcion
    return (1e6 * np.exp(lambd)) + ((np.exp(lambd) - 1)*(1e3*435 / lambd)) - 1.564e6

def ecuacion_poblacion_derivada(lambd):
    return (5000*(np.exp(lambd)*(200*lambd**2 + 87*lambd - 87)) + 87 ) / lambd**2

raiz_biseccion, iteraciones_biseccion, datos_biseccion = metodo_biseccion(inicio_intervalo = 0.08, fin_intervalo = 0.12, tolerancia = 1e-4, max_iteraciones = 100, funcion_objetivo = ecuacion_poblacion)
raiz_newton, iteraciones_newton , datos_newton = metodo_newton(p0 = 0.1, tolerancia = 1e-4, max_iteraciones = 100, funcion_objetivo = ecuacion_poblacion, funcion_derivada = ecuacion_poblacion_derivada)
raiz_secante, iteraciones_secante, datos_secante = metodo_secante(p0 = 0.1, p1 = 0.12, tolerancia = 1e-4, max_iteraciones = 100, funcion_objetivo = ecuacion_poblacion)
raiz_falsa_posicion, iteraciones_falsa_posicion, datos_falsa_posicion = metodo_falsa_posicion(p0 = 0.1, p1 = 0.12, tolerancia = 1e-4, max_iteraciones = 100, funcion_objetivo = ecuacion_poblacion)

plt.figure(figsize=(14, 4))
plt.plot(range(1, len(datos_biseccion) + 1), datos_biseccion, marker = "o", label = "Metodo Biseccion")
plt.plot(range(1, len(datos_newton) + 1), datos_newton, marker = "X", label = "Metodo newton")
plt.plot(range(1, len(datos_secante) + 1), datos_secante, marker = "o", label = "Metodo Secante")
plt.plot(range(1, len(datos_falsa_posicion) + 1), datos_falsa_posicion, label = "Metodo Falsa Posicion")
plt.title("Comparacion de convergencia en metodos")
plt.xlabel("Iteraciones N")
plt.legend()
plt.grid()
plt.show()
```

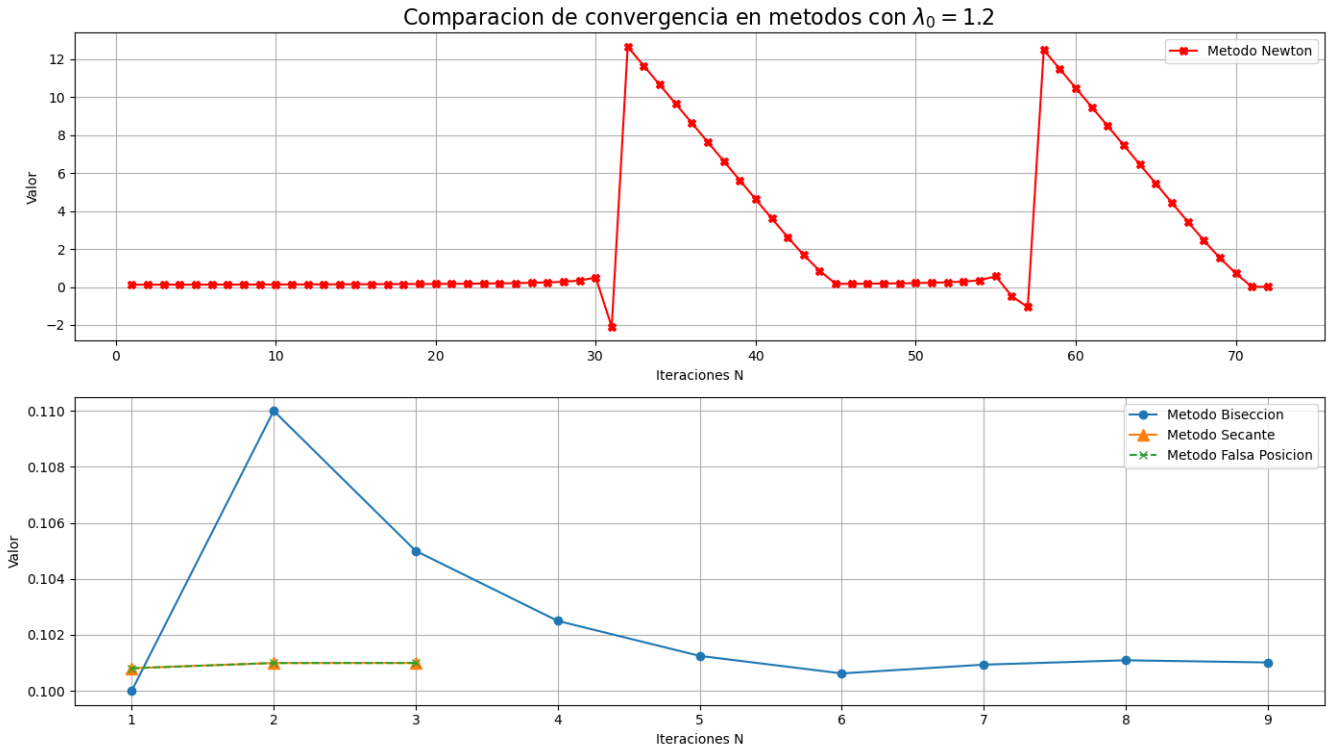



Nuestras sospechas fueron ciertas al momento de analizar el rango de conversión y se soluciona para algún $\lambda \approx 0{,}1$, tabulemos los valores obtenidos en cada método:

| Iteración | Newton | Secante | Falsa Posicion | Bisección |
|---|---|---|---|---|
| 1 | 0.09996831612889101 | 0.10098901232392817 | 0.10098901232392817 | 0.1 |
| 2 | | 0.10099785001305919 | 0.10099785001305919 | 0.11 |
| 3 | | | | 0.105 |
| 4 | | | | 0.1025 |
| 5 | | | | 0.10125 |
| 6 | | | | 0.100625 |
| 7 | | | | 0.1009375 |
| 8 | | | | 0.10109375 |
| 9 | | | | 0.101015625 |

Podemos ver en general hubo un performance excelente para los métodos de Newton y sus extensiones debido a la elección de punto el cuál se pudo estimar bastante bien (y quizás con suerte) a mera vista. Sin embargo vemos que a pesar de la 9 veces más tardada método de bisección, se tiene un avance mucho más constante y seguro debido a que admite posibles un rango mayor de error al momento de elegir el rango que contiene a la raíz.

Vamos a replicar el experimento pero ahora con un $\lambda_0 = 1{,}2$, un valor ligeramente mayor del proporcionado en un inicio para apreciar el desempeño de cada método




Ahora el método de Newton toma 72 iteraciones para llegar a la solución sin mencionar en varias iteraciones el valor se aleja bastante de la solución, mientras que sus extensiones siguen presentando un desempeño bastante similar. Esto solo deja más en claro el peso que tiene sobre el cálculo la elección del punto inicial que puede dependiendo, en general, de la formulación del método que se esté tratando.