FACULTAD DE INGENIERIA
Universidad de Buenos Aires

[75.13 - 95.04] ANÁLISIS NUMÉRICO
TRABAJO PRÁCTICO 1
2DO CUATRIMESTRE 2020

Búsqueda de raíces

AUTORES

Alonso Cristeff, Agustina. <aalonsoc@fi.uba.ar>	102.175
Buceta, M. Belén. <bbuceta@fi.uba.ar>	102.121
Caceres Alegre, Facundo. <fjcaceres@fi.uba.ar>	97.124
Perez, Ezequiel. <emperez@fi.uba.ar>	101.267
Robles, Ezequiel. <erobles@fi.uba.ar>	97.021

CÁTEDRA	Sassano
DOCENTE CORRECTOR	Payva, Matías
FECHA DE ENTREGA	01 de enero de 1900
LENGUAJE ELEGIDO	Python

Índice

1. Introduccion

El presente informe reune la documentación de la solución del primer trabajo práctico de la materia Análisis Numérico I, que estudia y analiza los distintos métodos de búsqueda de raíces vistos en clase.

2. Objetivos

El presente trabajo tiene los siguientes objetivos:

- Modelar y programar un algoritmo de fuerza bruta que halle la clave de un candado.
- Diseñar un algoritmo que permite calcular las raices reales de un polinomio de orden dos, utilizando sólo 32 bits.
- Definir dos funciones que modelen el volumen de un tanque de agua según cierto llenado que tengan, e implementar los distintos métodos de búsqueda de raíces vistos en clases para hallar las raíces de estas funciones.

3. Candado

3.1. Desarrollo

Para este problema, lo primero que hicimos fue plantear de qué forma íbamos a descubrir cuál era la clave, la que sería definida inicialmente de forma aleatoria con la función random.

Sabemos que la clave del candado es un número entero de 4 cifras, así que nuestra contraseña debe tener 4 dígitos y debe estar incluida en el intervalo entre 0000 y 9999.

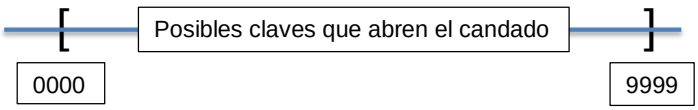

Figura 3.1: Representación del intervalo en el que debemos trabajar para hallar las claves.

Por lo tanto, decidimos tomar como contraseña inicial la que está compuesta de la forma 0000 y de esa forma 'barrer' todas las posibilidades hasta hallar la clave deseada. A esa contraseña inicial la comparamos con la clave del candado. Si esa coincidía, el candado abriría. Caso contrario, a la contraseña inicial le sumaríamos una unidad (0000+1=0001) y volveríamos a comparar con la clave. Ese proceso se repetiría hasta que finalmente la contraseña sea igual a la clave definida y por lo tanto, hayamos logrado encontrar la clave que abriría el candado.

3.2. Resultados

Los siguientes gráficos son dos ejemplos de los histogramas que son generados por el programa candado.py. Notamos que cada histograma generado es completamente distinto a los demás y a simple vista, en ningún caso podría hablarse de convergencia ya que las barras están distribuidas heterogéneamente en el histograma.

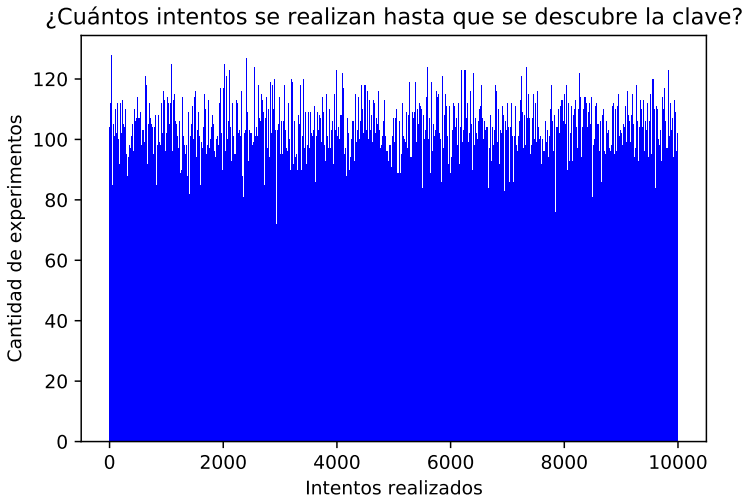

Figura 3.2: Ejemplo de histograma generado por el programa *candado.py*.

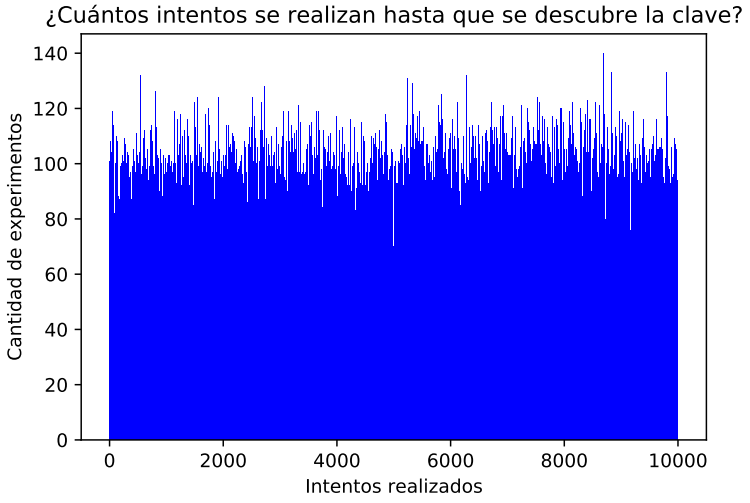

Figura 3.3: Otro ejemplo de histograma generado por el programa *candado.py*.

3.3. Conclusiones

En primer lugar, se cumplió con el objetivo de implementar un algoritmo de fuerza bruta capaz de hallar la clave de un candado cuya configuración fue aleatoria. Sin embargo, luego de realizar distintas pruebas podemos afirmar que no sería útil implementarlo como método de búsqueda de raíces ya que no se establece ninguna función que defina el comportamiento de la clave del candado. Es decir, ésta no resulta raíz de ninguna función. Además, al desconocer cúal es, se vuelve imposible definir una. De modo que la única solución que resta, es comparar clave a clave hasta acertar con la correcta. Por otro lado, a partir de lo anterior expuesto, concluímos que tampoco se puede considerar convergencia en esta forma de implementación. Ya que no se converge a ningún punto, sino que para cada clave generada -siendo ésta de un número de cuatro cifras, además de definirse aleatoriamente- deberán realizarse exactamente ese número de intentos hasta encontrarla. Se sugiere observar los histogramas adjuntados en (donde sea que estén) para visualizar cómo lo anterior explicado se refleja en ellos. Ver que en cada figura, el histograma que se genera es único y distinto. Por último, recomendamos utilizar el método descripto anteriormente en situaciones donde el intervalo de claves posibles o la cantidad de dígitos permitidos sean sumamente acotado. Tal que el tiempo aproximado de búsqueda no tienda a ser extenuante.

4. Calculadora

4.1. Desarrollo

Mediante nuestro programa *Resolvente.Py* se realizó el código que permite encontrar la o las raíces de un polinomio de grado 2, de la forma $ax^2 + bx + c$, para una calculadora de 32 bits. La lógica utilizada es la del método de la resolvente:

$$x_{1,2} = \frac{b^2 \pm \sqrt{b^2 + 4ac1}}{2a}$$

De esta manera, se pueden obtener las raíces de manera convencional con hasta 7 cifras significativas (máximo de precisión permitido por los 32 bits). Sin embargo, las matemáticas de la computadora no siguen siempre nuestra lógica llegando a resultados que no son del todo correctos. Esto pasa cuando el coeficiente "b^2" es mucho mayor a los otros dos, entonces sucede lo siguiente:

$$x_1 = \frac{b \pm \sqrt{b^2}}{2a} \tag{4.1}$$

$$x_1 = \frac{b \pm b}{2a} \tag{4.2}$$

$$x_1 = 0 \tag{4.3}$$

Como se puede observar esto sucede sólo en el caso de la suma y nos simplifica una de las raíces a cero, siendo esta respuesta no necesariamente la correcta. De manera similar ocurre para la segunda raíz en el caso de que el coeficiente "b" sea negativo. Para solucionar este inconveniente, se realizó el programa *Calculadora.Py*. El mismo cuenta con una condición para que cuando "b" sea mucho mayor a "a" y "c", se calcule la primer raíz empleando otro método:

$$x_1 = \frac{-2c}{b \pm \sqrt{b^2}} \tag{4.4}$$

Hay que aclarar que en las mismas condiciones este método tiene problemas en la segunda raíz. También, si bien Python puede calcular raíces complejas, los programas fueron realizados para que no lo hicieran y muestren el mensaje "Math error". Esto se debe a que si se calculara una raíz con una parte real de 7 cifras significativas, la calculadora no sería capaz de mostrar el resultado de la parte imaginaria.

¿Qué ventaja hay en una calculadora de 64 bits? En este caso 52 bits serían destinados a la mantisa, lo que significa una precisión de hasta 15 dígitos significativos. De esta manera la calculadora podría calcular valores de raíces más cercanos a los verdaderos sin perder información. Por ejemplo, en la tabla I se observa que para la tercer fila el valor de X1 con 32 bits es -0.010001, cuando con 64 sería -0.010001000200048793. En cambio ¿qué ventaja se obtiene con 16 bits? En este caso sólo 10 bits están destinados a la mantisa, dando una precisión de hasta 3 dígitos significativos. Al emplearse menos bits se utiliza menos memoria RAM permitiendo que las operaciones se lleven a cabo con mayor velocidad

4.2. Resultados

a	b	c	x_1	x_2	x_1^*	x_2^*
2	5	2	-0.5	-2	-0.5	-2
1	10	3	-0.3095842	-9.6904158	-0.3095842	-9.6904158
1	100	1	-0.010001	-99.989999	-0.010001	-99.989999
1	1000	1	-0.01	-999.999	-0.001	-999.999
1	$-1x10^{-8}$	1	0.0	100000000	1e-08	134217728.0
1	$1x10^{8}$	1	-0.0	-100000000.0	-1e-08	-134217728.0
1	1	100	*MATH ERROR*	*MATH ERROR*	*MATH ERROR*	*MATH ERROR*
100	1	1	*MATH ERROR*	*MATH ERROR*	*MATH ERROR*	*MATH ERROR*
0.01	0.5	0.01	-0.020008	-49.979992	-0.020008	-49.979992

Tabla 1: Tabla de valores de prueba

4.3. Conclusiones

Se puede decir que trabajando con 32 bits se logró obtener un algoritmo que calcula las raíces reales de un polinomio de grado 2. Sin embargo, en algunos casos, el método de la fórmula resolvente es incompatible con la lógica del programa. Estos casos son aquellos en los que la magnitud del coeficiente "b2" supera altamente a los demás coeficientes del polinomio. Por ello, se implementó un algoritmo que establece condiciones que analizan tanto al módulo como al signo de "b", para determinar si es necesario re-calcular alguna de las dos raíces mediante un método alternativo. Realizando una comparación de los resultados con los distintos métodos, se observa que, en ocasiones, cuando el primero encuentra una raíz igual a cero, el segundo encuentra su verdadero valor. En el caso de querer conseguir los componentes con el fin de aumentar la capacidad a 64 bits, lograríamos elevar la precisión de la calculadora hasta 1x1015 dígitos significativos. Lo cual resultaría particularmente útil en el caso de implementarse el primer algoritmo desarrollado, puesto que permite considerar coeficientes "b" más grandes, sin la necesidad de recurrir al segundo método. También, existe la posibilidad de emplear a 16 bits para el segundo algoritmo implementado.
Esta última opción, no la recomendamos ya que al hacerlo se perdería precisión y no calcularía correctamente alguna de las raíces.

5. Tanque

5.1. Desarrollo

Empleando la ecuación de la esfera y seleccionando el intervalo correcto se puede modelizar un sistema como lo es el tanque de agua mediante la función $v(x) = \frac{\pi x^2(3R-x)}{3}$. De esta manera, se pueden aplicar métodos numéricos, en este caso búsqueda de raíces, para hallar

soluciones del sistema, o dicho en otras palabras, hallar la altura del tanque para que este contenga un volumen determinado.

En primer lugar, partiendo de la ecuación 5.1 , buscamos la fórmula que nos permitiera calcular el volumen a un porcentaje determinado,

$$v(x) = \frac{\pi x^2(3R-x)}{3} \tag{5.1}$$

$$porcentaje = \frac{1+1+4+5+7}{5 \cdot 9{,}5} \tag{5.2}$$

$$vmax(x) = \frac{\pi x^2(3R-x)}{3} \cdot porcentaje \tag{5.3}$$

Luego, graficamos dichas funciones y sus derivadas.

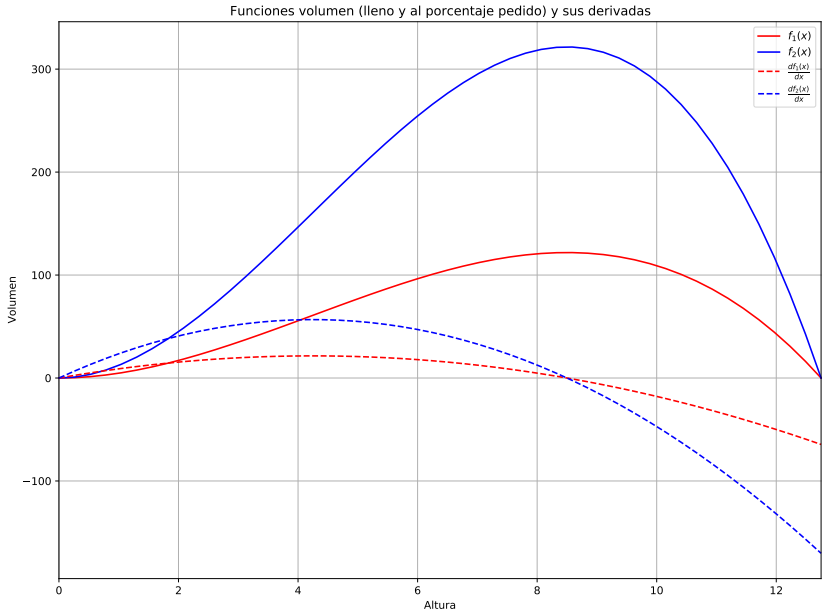

Figura 5.1: Funciones y sus derivadas.

Con un corto análisis de la derivada de la ecuación 5.3, hallamos el volumen máximo que podia contener el tanque cuando este estaba lleno a un porcentaje calculado. Este volumen era de la forma

$$vmax = \frac{4}{3}\pi \cdot R^3 \cdot porcentaje$$

.

Con ese nuevo dato, ahora el problema consiste en averiguar qué altura debe tener el tanque para poder contener ese volumen completamente. Es decir, resolver la siguiente ecuación:

$$V(x) = \frac{\pi x^2(3R - x)}{3} = vmax \tag{5.4}$$

Desarrollando 5.5, llegamos a la ecuación a la que debemos calcularle las raíces, utilizando los métodos vistos en clase.

$$V(x) = \frac{\pi x^2(3R - x)}{3} - vmax = 0 \tag{5.5}$$

$$V(x) = \pi R \cdot x^2 - \frac{\pi}{3} \cdot x^3 - \frac{4}{3}\pi \cdot R^3 \cdot porcentaje = 0 \tag{5.6}$$

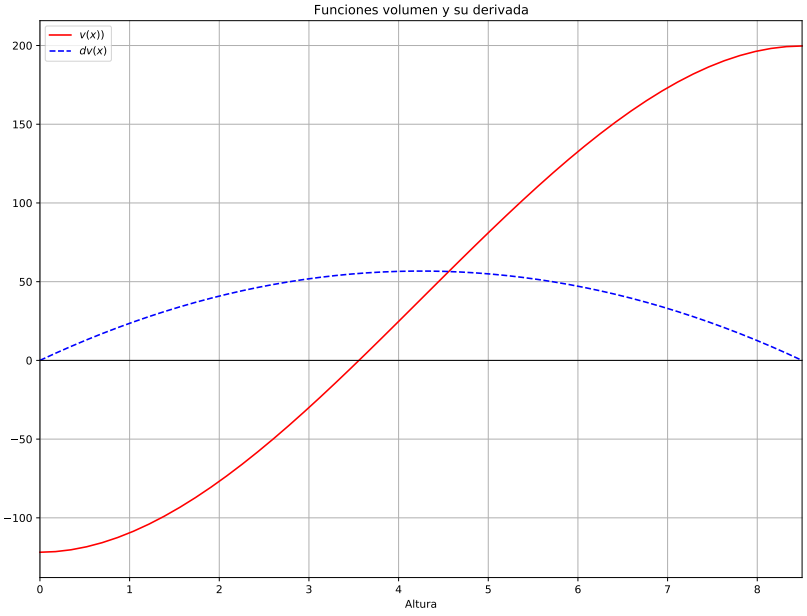

Figura 5.2: Volúmen y su derivada.

5.2. Resultados

A continuación se presentan los resultados obtenidos.

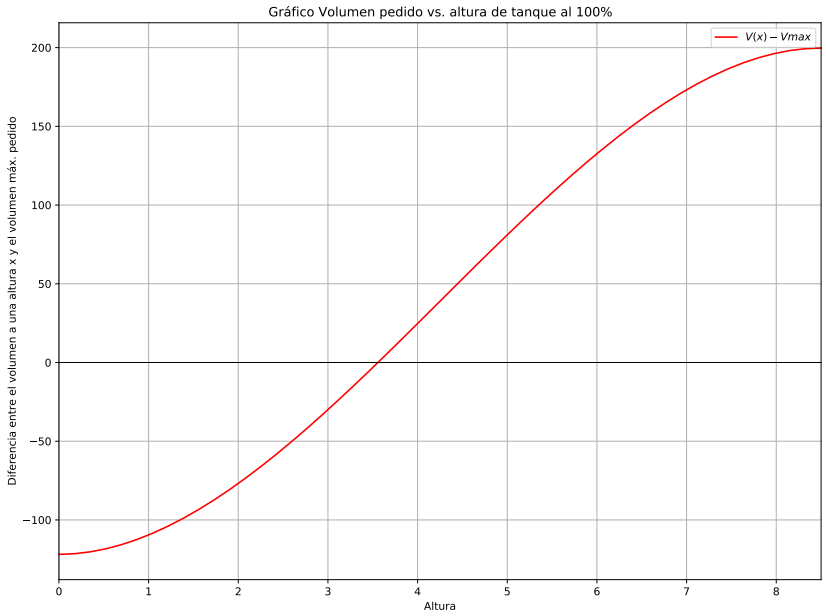

Figura 5.3: Diferencia entre el volumen requerido y el volumen máximo.

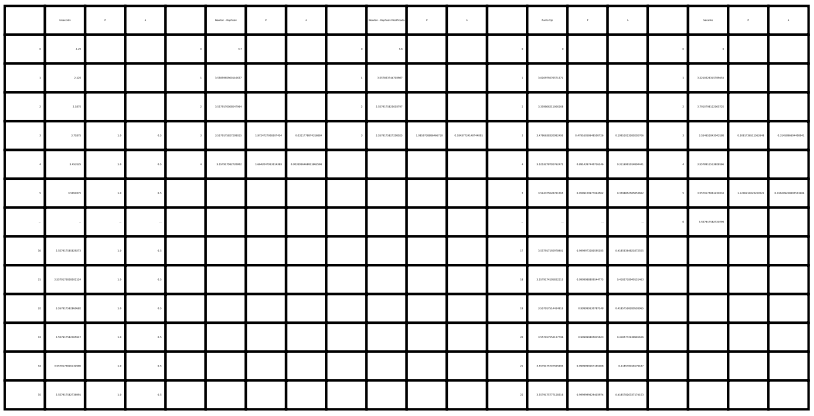

Figura 5.4: Resultados de los métodos

5.3. Conclusión

Tras realizar las pruebas con cinco métodos distintos y dos cotas de error se puede observar que todos los métodos convergen al mismo valor, mas no con la misma rapidéz. Siendo el de bisección el más lento en cuanto a iteraciones, los métodos de Newton-Raphson, Newton-Raphson Modificado y de la Secante son los que se aproximan a la raíz más rápido. En cuanto a la convergencia, notamos que a medida que la tolerancia utilizada aumenta, los resultados del cálculo de alfa

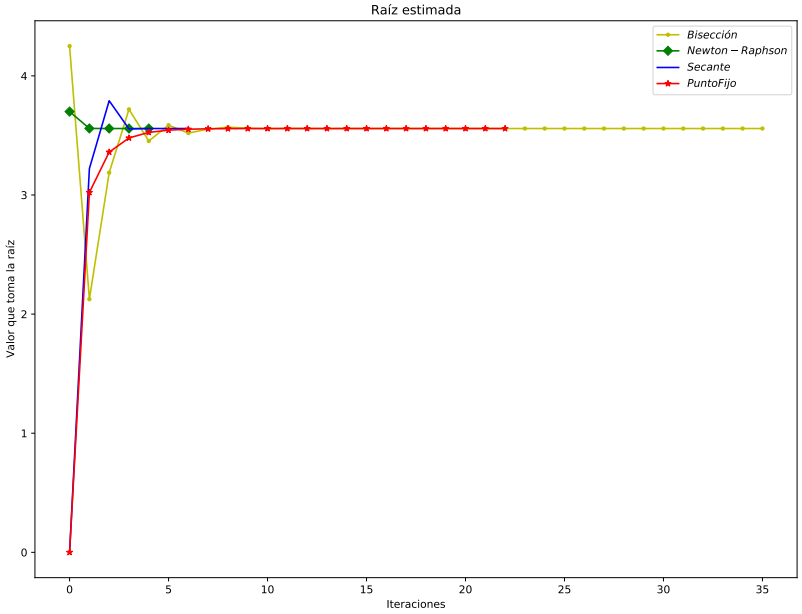

Figura 5.5: Raiz estimada

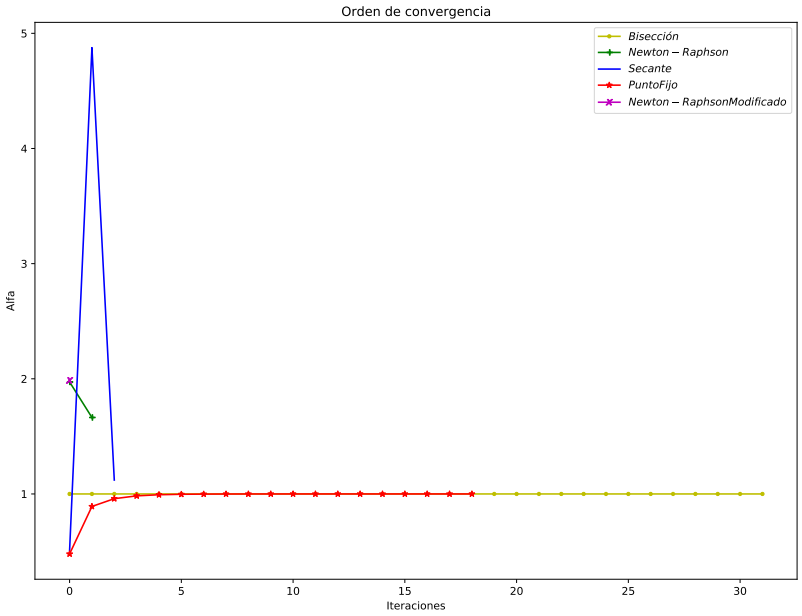

Figura 5.6: Convergencia de los métodos utilizados.

Referencias

[1] *Apuntes del curso Análisis numérico 1 - curso Sassano* - Facultad de Ingeniería - Universidad de Buenos Aires - 2020.

[2] *Análisis numérico* - Burden, R. L.; Faires, J.D.; Burden, A. M. - 10ma ed. Mexico, Iberoamérica - 2017