MARCELINO III BEDAÑA
MAJELYN MHAE ENERIO
KIM CARLO GONZALES
JOVIAN ASHER PARCO

Brute Force Algorithm (In terms of X)

Moduleone.brute_forcef (funcs,n_roots,epochs=100,tol=1.0e-05)

The brute force function (In terms of X) is a method to get the roots of any function by using a different X order with the same equation.

Parameters: **funcs : list data type with functions as elements**

The inputted collections of equations must be in a **list** data type that has a function as elements where the elements are stored, e.g., **def f(x): return "equation"** or **f = lambda x: "equation"**

n_roots : integer data type

The expected number of roots in the imputed equation.

epochs : integer data type, optional

The number of cycles that the algorithm will do.

tol : integer data type, optional

The tolerance or closeness of the answer compared to the output

Return: **roots: list data type with integer element**

The inputted equation's roots, the number of elements are based on the input n_roots, and the output will be raw data, which means the user has an option to use a decorator.

epoch : list data type with integer element

The output is in a list where the elements correspond to the total number of cycles per root output.

Notes:

The user input for the "funcs" must be in a list and has an accurately computed equation for the elements; using the brute force method requires more space and time to solve complex equations.

Usage:

Step one : Set the function containing the equation e.g. def f(x): return "equation" or f = lambda: "equation"

Step two : Call the function moduleone.brute_forcef and set the parameters of the desired equations

Step three : Get the function's answer using "roots" to output the list of roots and "epoch" to output the list of total cycles used per root.

Brute Force Algorithm (F(x)=0)

Moduleone.brute_forcefx (funct,start,n_roots,epochs=100,tol=1.0e-06,incre=1)

The brute force function (F(x)=0) is a method to get the roots of any function by finding the zeros or approximately the zeros of that function.

Parameters: **funcs : function data type**

The inputted equation must be inside of a function e.g. **def f(x): return "equation"** or **f = lambda x: "equation"** change the variable "x" depending on the equation inputted.

start : integer data type

The initial value where the algorithm will start.

n_roots : integer data type

The expected number of roots in the imputed equation.

epochs : integer data type, optional

The number of cycles that the algorithm will do.

tol : integer data type, optional

The tolerance or closeness of the answer compared to the output.

incre : integer data type, optional

The number that will be searched in the equation per cycle, to have an accurate result it must have 1>incre.

Return: **roots: list data type with integer element**

The inputted equation's roots, the number of elements are based on the input n_roots, and the output will be raw data, which means the user has an option to use a decorator.

epoch : integer data type

The total number of cycles that the algorithm runs to find the roots.

Notes:
To use the brute force method (F(x)=0) in a transcendental function, adjust the "epoch, "tol", and "incre" close to the expected answer to have an accurate output; using the brute force method requires more space and time to solve complex equations.

Usage:

Step one : Set the function containing the equation e.g. def f(x): return "equation" or f = lambda: "equation"

Step two : Call the function moduleone.brute_forcefx and set the parameters of the desired equations

Step three : Get the function's answer using "roots" to output the list of roots and "epoch" to output the total cycle used.

Newton-Raphson Algorithm
Moduleone.newton_raphson(funct, N_roots, epochs = 100, start = 0, end = 100, rnd_off = 3, Print = False)
The Newton-Raphson method is a method to get the roots of any function using the Newton-Raphson equation, which is represented by this equation: $x' = x - \frac{f(x)}{f'(x)}$.

Parameters:

funct : Function data type
A function that contains the equation that needs to find its roots. E.g. def f(x): return "equation" or f = lambda x: "equation", the parameter x could be changed depending on the variables in the equation.

N_roots: Integer data type
The estimated or desired number of roots to be found.

epochs: Integer data type
The number of cycles/loops the algorithm would do. The default epochs value is 100.

start: Integer data type, optional.
The starting value of **x** in the equation. The default start value is 0.

end: Integer data type, optional.
The final value of **x** in the equation. Also, serve as the parameter for the loop to end. The default end value is 100.

rnd_off: Integer data type, optional.
The desired number of decimal places that a user would want in the roots. The default round off value is 3

Print: Boolean data type, optional.
A statement that would print the root/s and its epoch/s. The default print value is false.

Returns: **roots : List data type**
A list that contains the roots for the given equation.

epochs_re : List data type
A list that contains the epoch (number of cycles) in which each root was found.

Notes:
If the desired number of roots (N_roots) is greater than the actual number of roots it would only return the actual number of roots.

Usage:

Step one: Create a function that contains the equation that needs to find its roots. e.g. def f(x): return “equation” or f = lambda: “equation.”

Step two: Call the function moduleone.newton_raphson and set the parameters of the desired equations.

Step three: Get the function’s answer using “[1]” to output the list of roots, “[0]” to see the list of cycles when the roots were found, or set "Print= True" to see a more detailed output. E.g. `newton_raphson(f,2, Print = True)[0]`.

Bisection Algorithm

Moduleone.bisection(funct, intval, n_roots, rnd_off = 3, bilim = 100, tol = 1.0e-06)
The Bisection algorithm is a method to get the roots of an equation; it uses the numerical sign to its advantage, where the difference between the signs can indicate how close or far the roots of the equations are.

Parameters: **funcs : function data type**
The inputted equation must be inside of a function e.g.
def f(x): return “equation” or **f = lambda x: “equation”** change the variable “x” depending on the equation inputted.

interval : list data type with integer element
The interval consists of a list with two integer elements; the first element is the estimated starting points for the roots, while the element is the estimated last point of the roots.

rnd_off : integer data type, optional
The desired number of decimal places that a user would want in the roots. The default round off value is 3.

bilim : integer data type, optional
The maximum number of cycles per root.

tol : integer data type, optional
The tolerance or closeness of the answer compared to the output.

Return: **roots: list data type with integer element**
The inputted equation's roots, the number of elements are based on the input n_roots, and the output will be raw data, which means the user has an option to use a decorator.

end_bisect: list data type with integer element
The output is in a list where the elements correspond to the total number of cycles per root output.

Usage:

Step one : Set the function containing the equation e.g. def f(x): return "equation" or f = lambda: "equation"

Step two : Call the function moduleone.bisection and set the parameters of the desired equations

Step three : Get the function's answer using "roots" to output the list of roots and "end_bisect" to output the total cycle used.

Regula Falsi Method

Moduleone.regula_falsi(funct, N_roots, pos = (0,100), a_range = (0,10,1), b_range = (0,10,1), rnd_off = 3, Print = False)

The Regula falsi or false position method is a method to find the roots of an equation by taking interval a and b in finding c and if f(c) is equal or extremely close to 0 then c is a root of the equation. The equation used to find c is $c = b - \frac{f(b) \cdot (b-a)}{f(b) - f(a)}$.

Parameters: **funct : function data type**

A function that contains the equation that needs to find its roots. E.g. def f(x): return “equation” or f = lambda x: “equation", the parameter x could be changed depending on the variables in the equation.

N_roots: integer data type
The estimated or desired number of roots to be found.

pos: range data type, optional
The number of cycles/loops the algorithm would do. The default epochs value is range(0,100).

a_range: range data type, optional
The range of value of **a** in the equation. The default value is range(1,10).

b_range: range data type, optional
The range of value of **b** in the equation. The default value is range(1,10).

rnd_off: integer data type, optional
The desired number of decimal places that a user would want in the roots. The default round off value is 3.

Print: boolean data type, optional
A statement that would print the root/s and its epoch/s. The default print value is false.

Return: **roots: list data type**
A list that contains the roots for the given equation.

posn: list data type
A list that contains the posn (position) where each root was found.

Usage:

Step one : Set the function containing the equation e.g. def f(x): return “equation” or f = lambda: “equation”

Step two : Call the function moduleone.regula_falsi and set the parameters of the desired equations

Step three : Get the function’s answer using “[1]” to output the list of roots, “[0]” to see the list of positions where the roots were found, or or set "Print= True" to see a more detailed output. E.g. `regula_falsi(f,2, Print = True)[0]`.

Secant Method :

[7] The idea underlying the secant method is the same as the one underlying Newton's method: to find an approximate zero of a function $f(x)$. we find instead a zero for a linear function $f(x)$ that corresponds to a "best straight line of fit" to $f(x)$

$$[8] \quad x_2 = x_1 - \frac{f(x_1)\cdot(x_1 - x_0)}{f(x_1) - f(x_0)}$$

def secant(f,a,b,epochs=100,tol=1.0e-06)

Parameters : **f : function**

a function for which we are trying to approximate a solution f(x)=0. This parameter can be changed depending on the equation that is given or inputted.

a,b: number int/float

An interval in which the root is expected

Epochs : int

A number of iterations in which the algorithm would do

Tol : Tolerance: float/int

Tolerance

Returns **: ROOTS : data type**

A list that contains the roots for the given equation.

Epoch : List data type

A list that contains the epoch (number of cycles) in which each root was found

Usage:

Step one : Set the function containing the equation, it depends on the given equation

Step two : call the function secant(). And set the given parameter.

Step Three : Get the function’s answer using root to output the list of roots so that it will show the roots answer

REFERENCES

[1] M. RASHEED, S. SHIHAB, T. RASHID, and M. Enneffati, “Some Step Iterative Method for Finding Roots of a Nonlinear Equation”, JQCM, vol. 13, no. 1, pp. Math Page 95-, Feb. 2021.

[2] M. R. King and N. A. Mody, “Root-finding techniques for nonlinear equations,” Numerical and Statistical Methods for Bioengineering, pp. 310–353, Jun 2012.

[3] G. Corliss, “Which Root Does the Bisection Algorithm Find?,” SIAM Review, vol. 19, no. 2, pp. 325–327, 1977.

[4] Lopez, D.J.D.”Roots of equation”.2021.

[5] E. Chopra. “Regula Falsi Method for finding root of a polynomial,” October 24, 2019. https://iq.opengenus.org/regula-falsi-method/ (accessed March 04, 2021)

[6] O. Veliz, "False Position Method - Regula Falsi", 2019. [Online]. Available: https://www.youtube.com/watch?v=pg1I8AG59Ik. [Accessed: 04- Mar- 2021].

[7] P. Walls (2019) *Roots and Optimization: Secant Method*. Mathematical Python. https://www.math.ubc.ca/~pwalls/math-python/roots-optimization/secant/

[8] B. Binegar. (1998) *Secant Method*. Fall Lecture onn Introduction to Numerical Analysis. Oklahoma State University. https://math.okstate.edu/people/binegar/4513-F98/4513-l08.pdf