1η Υποχρεωτική Εργασία Στο Μάθημα της Αριθμητικής Ανάλυσης: Άσκηση 2

Ονοματεπώνυμο: Μπαρακλιλής Ιωάννης
AEM: 3685

17 Δεκεμβρίου 2020

1 Ζητούμενο (α): Τροποποιημένη μέθοδος Newton-Raphson

Ζητείται η υλοποίηση μίας τροποποιημένης μεθόδου Newton-Raphson με

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)^2 f''(x_n)}{f'(x_n)^3}$$

.

Η μέθοδος αυτή υλοποιείται προγραμματιστικά στην γλώσσα python (3.7) στο αρχείο a_Modified_Newton_Raphson.py το οποίο φαίνεται παρακάτω:

```
def modified_newton_raphson(function, function_derivative,
    function_second_derivative, starting_point,
    digits_of_precision):
    iteration_counter = 0

    f = function
    f_d = function_derivative
    f_dd = function_second_derivative

    x = starting_point

    while True:
        if f(x) == 0:
            return x, iteration_counter
        else:
            iteration_counter += 1
            x_next = x - (f(x) / f_d(x)) - (1/2) * ( (f(x) ** 2
                * f_dd(x)) / (f_d(x)**3) )
```

```
        if abs(x_next - x) < 0.5 * 10 ** (-1.0 *
            digits_of_precision):
            return x_next, iteration_counter
        else:
            x = x_next
```

Στον παραπάνω κώδικα:
Ορίζω την συνάρτηση modified_newton_raphson η οποία δέχεται την συνάρτηση, την πρώτη και δεύτερη παράγωγο της συνάρτησης, το αρχικό σημείο της ακολουθίας και τα ζητούμενα ψηφία ακρίβειας:

Αρχικά, ορίζω ψευδώνυμα για την συνάρτηση και την πρώτη και δεύτερη παράγωγο της (με ονόματα f, f_d και f_dd αντίστοιχα). Επίσης, ορίζω και αρχικοποιώ την μεταβλητή iteration_counter, σε 0, που "μετράει" τις επαναλήψεις της μεθόδου και την μεταβλητή x, στο αρχικό σημείο της ακολουθίας (που δίνεται ως όρισμα), που αποθηκεύει την τρέχουσα εκτίμηση της ρίζας.

Στην συνέχεια, αρχίζει ατέρμονος βρόχος (που θα τερματισει αργότερα όταν "πετύχω" την επιθυμιτή ακρίβεια) όπου γίνονται οι ενέργειες:
Πρώτα, ελέγχω αν η τρέχουσα εκτίμηση αποτελεί ρίζα της συνάρτησης, επειδή αν ισχύει αυτό δεν χρειάζεται να συνεχίσω την αναζήτηση γιατί βρέθηκε η ζητούμενη ρίζα. Άν είναι τερματίζω την εκτέλεση της συνάρτησης και επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί.
Στην συνέχεια (η τρέχουσα εκτίμιση δεν αποτελεί ρίζα): ενημερώνω την μεταβλητή που αποθηκεύει τον αριθμό των επαναλήψεων, υπολογίζω το επόμενο στοιχείο ακολουθίας που αποτελεί την επόμενη εκτίμηση της ρίζας ($x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)^2 f''(x_n)}{f'(x_n)^3}$) και το αποθηκεύω στην (νέα) μεταβλητή x_next.
Τέλος, ελέγχω άν η διαφορά της νέας και παλιάς εκτίμησης (διαφορά μεταβλητών x και x_next είναι μικρότερη του ανεκτού σφάλματος και αν είναι επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί (τερματίζοντας την εκτέλεση της συνάρτησης).

2 Ζητούμενο (β): Τροποποιημένη μέθοδος διχοτόμησης

Ζητείται η υλοποίηση μίας τροποποιημένης μεθόδου διχοτόμησης όπου η εκτίμηση για την ρίζα δεν είναι το μέσο του διαστήματος αναζήτησης σε κάθε βήμα αλλά ένα τυχαίο σημείο που επιλέγεται με την χρήση μιας συνάρτησης παραγωγής τυχαίων αριθμών εντός του διαστήματος αναζήτησης. Η επιλογή τυχαίου σημείου εντός διαστήματος μπορεί να γίνει με την χρήση της συνάρτησης uniform, του module random, που επιστρέφει έναν τυχαίο πραγματικό αριθμό εντός του διαστήματος τα άκρα του οποίου που δίνονται ως παράμετροι.

Η μέθοδος αυτή υλοποιείται προγραμματιστικά στην γλώσσα python στο αρχείο b_Modified_Bisection.py το οποίο φαίνεται παρακάτω:

```
import random

def modified_bisection(function, range_beginning, range_ending
    , digits_of_precision):
    iteration_counter = 0

    f = function
    a = range_beginning
    b = range_ending
    fa = f(a)
    fb = f(b)

    r = random.uniform(a, b)
    fr = f(r)

    while True:
        if fr == 0:
            return r, iteration_counter
        elif fa * fr < 0:
            fb = fr
            b = r
        else:
            fa = fr
            a = r

        iteration_counter += 1

        old_r = r
        r = random.uniform(a, b)
        fr = f(r)

        if abs(old_r - r) < 0.5 * (10 ** (-1 *
            digits_of_precision)):
            return r, iteration_counter
```

Ο παραπάνω κώδικας είναι πανομοιότυπος με τον αντίστοιχο του ερωτήματος (α) της άσκησης 1, που υλοποιεί την κλασσική μέθοδο διχοτόμησης, με την διαφορά ότι η νέα εκτίμηση ρίζας υπολογίζεται ως τυχαίο σημείο εντός του διαστήματος αναζήτησης αντί του μέσου αυτού του διαστήματος.

Παρ' όλα αυτά, είναι χρήσιμο να αναλυθεί (έκ νέου) η λειτουργία του παραπάνω κώδικα:
Ορίζω την συνάρτηση modified_bisection η οποία δέχεται την συνάρτηση, το

αριστερό άκρο του αρχικού διαστήματος, το δεξί άκρο του αρχικού διαστήματος και τα ζητούμενα ψηφία ακρίβειας:

Αρχικά, ορίζω ψευδώνυμα για την συνάρτηση, αρχή και τέλος διαστήματος αναζήτησης ρίζας (με ονόματα f, a, b αντίστοιχα). Ακόμη, ορίζω και αρχικοποιώ την μεταβλητή iteration_counter, σε 0, που "μετράει" τις επαναλήψεις της μεθόδου. Επίσης, ορίζω και δίνω τιμές στις μεταβλητές που αποθηκεύουν την τιμή συνάρτησης στην αρχή και τέλος διαστήματος, το τυχαίο σημείο του διαστήματος και την τιμή συνάρτησης στο τυχαίο σημείο διαστήματος (με ονόματα fa, fb, r, fr αντίστοιχα).

Στην συνέχεια, αρχίζει ατέρμονος βρόχος (που θα τερματισει αργότερα όταν "πετύχω" την επιθυμιτή ακρίβεια) όπου γίνονται οι ενέργειες:
Πρώτα, ελέγχω αν η τρέχουσα εκτίμηση (τυχαίο σημείο του διαστήματος) αποτελεί ρίζα της συνάρτησης, επειδή αν ισχύει αυτό δεν χρειάζεται να συνεχίσω την αναζήτηση γιατί βρέθηκε η ζητούμενη ρίζα. Άν είναι τερματίζω την εκτέλεση της συνάρτησης και επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί.
Στην συνέχεια (η τρέχουσα εκτίμιση δεν αποτελεί ρίζα), υπολογίζω το νέο διάστημα αναζήτησης: Άν το γινόμενο της συνάρτησης στο αριστερό άκρο (a) με την συνάρτηση στο τυχαίο σημείο του διαστήματος (r) είναι αρνητικό ($fa * fr < 0$) τότε έχω ρίζα στο διάστημα $[a, r]$ και ορίζω νέο διάστημα αναζήτησης ορίζοντας ως νέο δεξί άκρο διαστήματος το τυχαίο σημείο (θέτω b = r και για να μην το ξαναυπολογίσω θέτω fb = fr). Διαφορετικά (το πάνω γινόμενο είναι θετικό), έχω ρίζα στο διάστημα $[r, b]$ (όπου b δεξί άκρο του αρχικού διαστήματος) και ορίζω νέο διάστημα αναζήτησης ορίζοντας ως νέο αριστερό άκρο διαστήματος το τυχαίο σημείο (θέτω a = r και για να μην το ξαναυπολογίσω θέτω fa = fr).
Μετά, ενημερώνω την μεταβλητή που αποθηκεύει τον αριθμό των επαναλήψεων και εκείνη που αποθηκεύει την παλιά τιμή του τυχαίου σημείου που θα χρησιμοποιήσω αργότερα για τον έλεγχο του σφάλματος, υπολογίζω την νέα εκτίμηση ρίζας και υπολογίζω την τιμή της συνάρτησης σε αυτή την εκτίμηση.
Τέλος, ελέγχω άν η διαφορά της νέας και παλιάς εκτίμησης είναι μικρότερη του ανεκτού σφάλματος και αν είναι επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί (τερματίζοντας την εκτέλεση της συνάρτησης).

3 Ζητούμενο (γ): Τροποποιημένη μέθοδος της τέμνουσας

Ζητείται η υλοποίηση μίας τροποποιημένης μεθόδου της τέμνουσας που χρειάζεται 3 αρχικά σημεία x_n, x_{n+1}, x_{n+2} που υπολογίζει επόμενες εκτιμήσεις της ρίζας με τον τύπο:

$$x_{n+3} = x_{n+2} - \frac{r(r-q)(x_{n+2} - x_{n+1}) + (1-r)s(x_{n+2} - x_n)}{(q-1)(r-1)(s-1)}$$

, με $q = \dfrac{f(x_n)}{f(x_{n+1})}$, $r = \dfrac{f(x_{n+2})}{f(x_{n+1})}$ και $s = \dfrac{f(x_{n+2})}{f(x_n)}$.

Η μέθοδος αυτή υλοποιείται προγραμματιστικά στην γλώσσα python στο αρχείο c_Modified_Secant.py το οποίο φαίνεται παρακάτω:

```
def modified_secant(function, point_one, point_two,
    point_three, digits_of_precision):
    iteration_counter = 0

    f = function

    x1 = point_one
    x2 = point_two
    x3 = point_three

    while True:
        if f(x3) == 0:
            return x3, iteration_counter
        else:
            iteration_counter += 1

            q = f(x1) / f(x2)
            r = f(x3) / f(x2)
            s = f(x3) / f(x1)

            x_next = x3 - ( r * (r - q) * (x3 - x2) + (1 - r) *
                 s * (x3 - x1) ) / ( (q - 1) * (r - 1) * (s - 1)
                 )

            if abs(x_next - x3) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x1 = x2
                x2 = x3
                x3 = x_next
```

Στον παραπάνω κώδικα:

Ορίζω την συνάρτηση modified_secant η οποία δέχεται την συνάρτηση, τα τρία αρχικά σημεία και τα ζητούμενα ψηφία ακρίβειας:

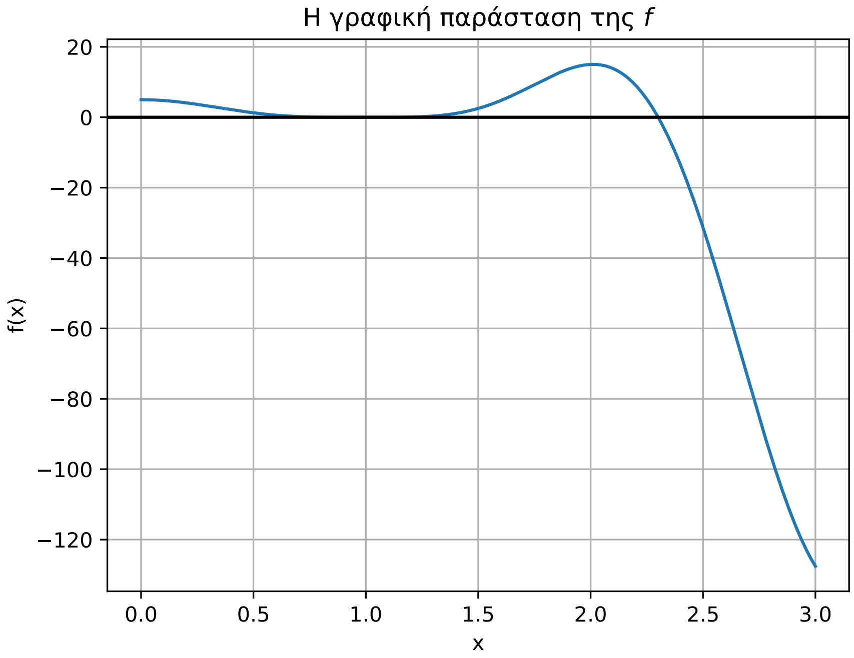
Αρχικά, ορίζω ψευδώνυμο για την συνάρτηση (με όνομα f), ορίζω και αρχικοποιώ την μεταβλητή iteration_counter, σε 0, που "μετράει" τις επαναλήψεις της μεθόδου και ορίζω και αρχικοποιώ τις μεταβλητές x1, x2 και x3, που αποθηκεύουν τα τρία σημεία απο τα οποία υπολογίζεται εκτίμηση της ρίζας, στις τιμές αρχικών σημείων point_one, point_two και point_three αντίστοιχα που δίνοται ως ορίσματα.

Στην συνέχεια, αρχίζει ατέρμονος βρόχος (που θα τερματισει αργότερα όταν "πετύχω" την επιθυμιτή ακρίβεια) όπου γίνονται οι ενέργειες:
Πρώτα, ελέγχω αν η τρέχουσα εκτίμηση (που βρίσκεται στην μεταβλητή x3) αποτελεί ρίζα της συνάρτησης, επειδή αν ισχύει αυτό δεν χρειάζεται να συνεχίσω την αναζήτηση γιατί βρέθηκε η ζητούμενη ρίζα. Άν είναι τερματίζω την εκτέλεση της συνάρτησης και επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί.
Στην συνέχεια (η τρέχουσα εκτίμιση δεν αποτελεί ρίζα): ενημερώνω την μεταβλητή που αποθηκεύει τον αριθμό των επαναλήψεων, υπολογίζω τα q, r, s που δίνονται στον παραπάνω τύπο ($q = \frac{f(x_n)}{f(x_{n+1})}$, $r = \frac{f(x_{n+2})}{f(x_{n+1})}$ και $s = \frac{f(x_{n+2})}{f(x_n)}$), υπολογίζω το επόμενο στοιχείο ακολουθίας που αποτελεί την επόμενη εκτίμηση της ρίζας ($x_{n+3} = x_{n+2} - \frac{r(r-q)(x_{n+2}-x_{n+1}) + (1-r)s(x_{n+2}-x_n)}{(q-1)(r-1)(s-1)}$) και το αποθηκεύω στην (νέα) μεταβλητή x_next.
Τέλος, ελέγχω άν η διαφορά της νέας και παλιάς εκτίμησης (διαφορά μεταβλητών x_next και x3 είναι μικρότερη του ανεκτού σφάλματος και άν είναι επιστρέφω την εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων που χρειάστηκαν για να βρεθεί (τερματίζοντας την εκτέλεση της συνάρτησης). Διαφορετικά (η διαφορά μεταβλητών x_next και x3 δεν είναι μικρότερη του ανεκτού σφάλματος), ενημερώνω τις μεταβλητές x1, x2 και x3 (το x1 παίρνει την τιμή του x2, το x2 παίρνει την τιμή του x3 και το x3 την τιμή του x_next).

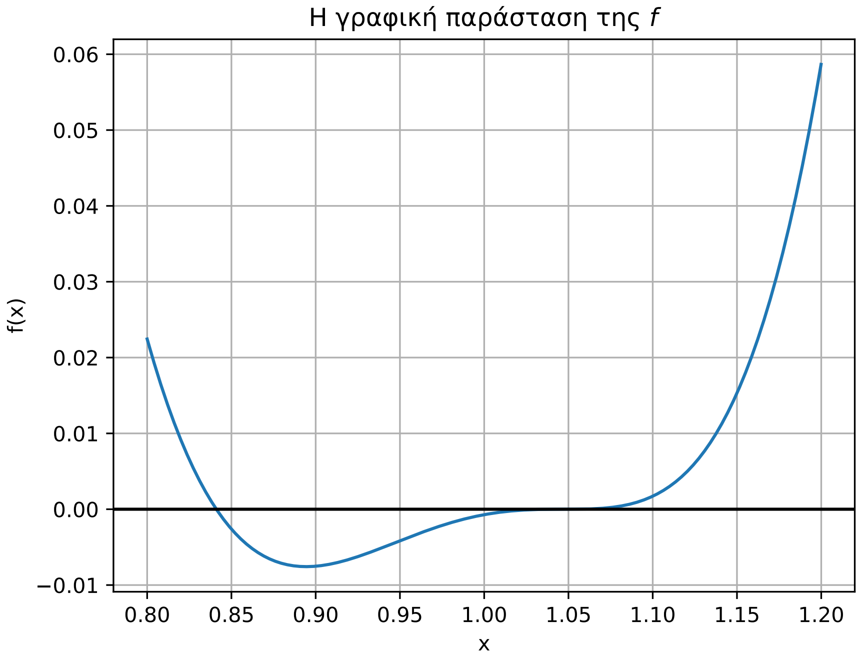
4 Ερώτημα (1): Εύρεση ριζών με χρήση προηγούμενων μεθόδων

Ζητείται να βρεθούν όλες οι ρίζες της συνάρτησης $f(x) = 94cos^3x - 24cosx + 177sin^2x - 108sin^4x - 72cos^3xsin^2x - 65$ στο διάστημα $[0, 3]$ με ακρίβεια 5ου δεκαδικού ψηφίου.

Αρχικά, πρέπει να γίνει η γραφική παράσταση της f για να βρούμε τα διαστήματα και σημεία "κοντά" στα οποία βρίσκονται οι ρίζες:



Μπορούμε να παρατηρήσουμε ρίζες στα διαστήματα $[0.5, 1.5]$ και $[2.0, 2.5]$. Αν μελετήσουμε την συνάρτηση πιο αναλυτικά θα παρατηρήσουμε ότι στο διάστημα $[0.5, 1.5]$ δεν έχει μία ρίζα αλλά δύο. Για αυτό τον λόγο είναι χρήσιμο να κάνουμε την γραφική παράσταση της συνάρτησης στο διάστημα $[0.8, 1.2]$ όπου οι δύο αυτές ρίζες φαίνονται πιο ξεκάθαρα:



Οπότε έχουμε μία αρχική εκτίμηση της τοποθεσίας των ριζών με την οποία μπορούμε να τροφοδοτήσουμε τους παραπάνω αλγορίθμους για την εύρεση τους.

Η εύρεση των ριζών με κάθε έναν απο τους παραπάνω αλγορίθμους γίνεται προγραμματιστικά (σε γλώσσα python) στο αρχείο d_1_find_f_roots.py το οποίο φαίνεται παρακάτω:

```
import math
from math import sin, cos
import random


def modified_newton_raphson(function, function_derivative,
    function_second_derivative, starting_point,
    digits_of_precision):
    iteration_counter = 0

    f = function
    f_d = function_derivative
    f_dd = function_second_derivative

    x = starting_point
```

```
    while True:
        if f(x) == 0:
            return x, iteration_counter
        else:
            iteration_counter += 1
            x_next = x - (f(x) / f_d(x)) - (1/2) * ( (f(x) ** 2
                 * f_dd(x)) / (f_d(x)**3) )

            if abs(x_next - x) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x = x_next


def modified_bisection(function, range_beginning, range_ending
    , digits_of_precision):
    iteration_counter = 0

    f = function
    a = range_beginning
    b = range_ending
    fa = f(a)
    fb = f(b)

    r = random.uniform(a, b)
    fr = f(r)

    while True:
        if fr == 0:
            return r, iteration_counter
        elif fa * fr < 0:
            fb = fr
            b = r
        else:
            fa = fr
            a = r

        iteration_counter += 1

        old_r = r
        r = random.uniform(a, b)
        fr = f(r)
```

```
        if abs(old_r - r) < 0.5 * (10 ** (-1 *
            digits_of_precision)):
            return r, iteration_counter


def modified_secant(function, point_one, point_two,
    point_three, digits_of_precision):
    iteration_counter = 0

    f = function

    x1 = point_one
    x2 = point_two
    x3 = point_three

    while True:
        if f(x3) == 0:
            return x3, iteration_counter
        else:
            iteration_counter += 1

            q = f(x1) / f(x2)
            r = f(x3) / f(x2)
            s = f(x3) / f(x1)

            x_next = x3 - ( r * (r - q) * (x3 - x2) + (1 - r) *
                 s * (x3 - x1) ) / ( (q - 1) * (r - 1) * (s - 1)
                 )

            if abs(x_next - x3) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x1 = x2
                x2 = x3
                x3 = x_next


def f(x):
    return 94 * (cos(x) ** 3) - 24 * cos(x) + 177 * (sin(x) **
        2) - 108 * (sin(x) ** 4) - 72 * (cos(x) ** 3) * (sin(x)
         ** 2) - 65
```

```
def f_derivative(x):
    return - 282 * (cos(x) ** 2) * sin(x) + 24 * sin(x) + 354 *
         sin(x) * cos(x) - 432 * (sin(x) ** 3) * cos(x) + 216 *
         (cos(x) ** 2) * (sin(x) ** 3) - 144 * (cos(x) ** 4) *
        sin(x)


def f_second_derivative(x):
    return 564 * cos(x) * (sin(x) ** 2) - 282 * (cos(x) ** 3) +
         24 * cos(x) + 354 * (cos(x) ** 2) - 354 * (sin(x) **
        2) - 1296 * (cos(x) ** 2) * (sin(x) ** 2) + 432 * (sin(
        x) ** 4) - 432 * cos(x) * (sin(x) ** 4) \
           + 648 * (cos(x) ** 3) * (sin(x) ** 2) + 576 * (cos(x
               ) ** 3) * (sin(x) ** 2) - 144 * (cos(x) ** 5)


print("Modified␣Newton-Raphson␣roots:")
root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 0.8, 5)
print("\tThe␣root␣near␣0.8:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 1.0, 5)
print("\tThe␣root␣near␣1.0:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 2.3, 5)
print("\tThe␣root␣near␣2.3:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))


print("\nModified␣Bisection␣roots:")
root, loops_counter = modified_bisection(f, 0.8, 1.0, 5)
print("\tThe␣root␣in␣[0.8,␣1.0]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_bisection(f, 1.0, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_bisection(f, 2.2, 2.4, 5)
```

```
print("\tThe␣root␣in␣[2.2,␣2.4]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))


print("\nModified␣Secant␣roots:")
root, loops_counter = modified_secant(f, 0.8, 0.85, 0.9, 5)
print("\tThe␣root␣in␣[0.8,␣0.9]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_secant(f, 1.0, 1.1, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_secant(f, 2.0, 2.25, 2.5, 5)
print("\tThe␣root␣in␣[2.0,␣2.5]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))
```

Στον παραπάνω κώδικα, ορίζω εκ νέου τις συναρτήσεις που υλοποιούν τις τροποιημένες μεθόδους.

Στην συνέχεια ορίζω τις συναρτήσεις f, f_derivative, f_second_derivative που υλοποιούν την συνάρτηση f και την πρώτη και δεύτερη παράγωγο της αντίστοιχα. Ακολούθως, για κάθε ρίζα καλώ την συνάρτηση κάθε μεθόδου (με αντίστοιχες παραμέτρους) και αποθηκεύω τα αποτελέσματα στις μεταβλητές root, loops_counter που αποθηκεύουν την τελική εκτίμηση της ρίζας και τον αριθμό των επαναλήψεων αντίστοιχα. Μετά, χρησιμοποιώντας αυτές τις μεταβλητές τυπώνω τα αποτελέσματα.

Για την τροποποιημένη μέθοδο Newton-Raphson χρησιμοποιώ ως αρχικές εκτιμήσεις ριζών τα σημεία: 0.8, 1.0 και 2.3.
Για την τροποιημένη μέθοδο διχοτόμησης χρησιμοποιώ ως αρχικά διαστήματα τα: $[0.8, 1.0]$, $[1.0, 1.2]$ και $[2.2, 2.4]$.
Για την τροποιημένη μέθοδο τέμνουσας χρησιμοποιώ ως αρχικές 3αδες σημείων (x_n, x_{n+1}, x_{n+2}) τα: (0.8, 0.85, 0.9), (1.0, 1.1, 1.2) και (2.0, 2.25, 2.5).

Άν εκτελέσουμε τον κώδικα θα τυπωθεί:

```
Modified Newton-Raphson roots:
    The root near 0.8: 0.841069. It was calculated in 4 repetitions
    The root near 1.0: 1.047187. It was calculated in 14 repetitions
    The root near 2.3: 2.300524. It was calculated in 2 repetitions

Modified Bisection roots:
    The root in [0.8, 1.0]: 0.841082. It was calculated in 18 repetitions
    The root in [1.0, 1.2]: 1.047188. It was calculated in 20 repetitions
    The root in [2.2, 2.4]: 2.300528. It was calculated in 13 repetitions

Modified Secant roots:
    The root in [0.8, 0.9]: 0.841069. It was calculated in 6 repetitions
    The root in [1.0, 1.2]: 1.047182. It was calculated in 23 repetitions
    The root in [2.0, 2.5]: 2.300524. It was calculated in 5 repetitions
```

Μπορούμε να δούμε ότι οι μέθοδοι συγκλίνουν για όλες τις επιλεγμένες αρχικές προσεγγίσεις και:

- Η τροποποιημένη μέθοδος Newton-Raphson δίνει ως αποτέλεσμα τις ρίζες: 0.841069 (σε 4 επαναλήψεις), 1.047187 (σε 14 επαναλήψεις) και 2.300524 (σε 2 επαναλήψεις).
- Η τροποιημένη μέθοδος διχοτόμησης δίνει ως αποτέλεσμα τις ρίζες: 0.841082 (σε 18 επαναλήψεις), 1.047188 (σε 20 επαναλήψεις) και 2.300528 (σε 13 επαναλήψεις) (τα αποτελέσματα σε επόμενη εκτέλεση του ίδιου αρχείου αναμένεται να διαφέρουν λόγω της τυχαιότητας που λαμβάνει μέρος στον υπολογισμό της ρίζας).
- Η τροποιημένη μέθοδος τέμνουσας δίνει ως αποτέλεσμα τις ρίζες: 0.841069 (σε 6 επαναλήψεις), 1.047182 (σε 23 επαναλήψεις) και 2.300524 (σε 5 επαναλήψεις).

5 Ερώτημα (2): Έλεγχος σύγκλισης τροποιημένης μεθόδου διχοτόμησης

Ζητείται να γίνει εκτέλεση του αλγορίθμου τροποιημένης μεθόδου διχοτόμησης και να διαπιστωθεί το αν συγκλίνει πάντα σε ίδιο αριθμό επαναλήψεων.

Προγραμματιστικά αυτο γίνεται (σε γλώσσα python) στο αρχείο d_2_modified_bisection_convergence.py το οποίο φαίνεται παρακάτω:

```
import random
import math
from math import sin, cos
```

```
def modified_bisection(function, range_beginning, range_ending
    , digits_of_precision):
    iteration_counter = 0

    f = function
    a = range_beginning
    b = range_ending
    fa = f(a)
    fb = f(b)

    r = random.uniform(a, b)
    fr = f(r)

    while True:
        if fr == 0:
            return r, iteration_counter
        elif fa * fr < 0:
            fb = fr
            b = r
        else:
            fa = fr
            a = r

        iteration_counter += 1

        old_r = r
        r = random.uniform(a, b)
        fr = f(r)

        if abs(old_r - r) < 0.5 * (10 ** (-1 *
            digits_of_precision)):
            return r, iteration_counter


def f(x):
    return 94 * (cos(x) ** 3) - 24 * cos(x) + 177 * (sin(x) **
        2) - 108 * (sin(x) ** 4) - 72 * (cos(x) ** 3) * (sin(x)
         ** 2) - 65


print("First␣root:")
for i in range(10):
    root, loops_counter = modified_bisection(f, 0.8, 1.0, 5)
```

```
    print("\tRoot:␣{:f}␣calculated␣in␣{:d}␣repetitions".format(
        root, loops_counter))

print("Second␣root:")
for i in range(10):
    root, loops_counter = modified_bisection(f, 1.0, 1.2, 5)
    print("\tRoot:␣{:f}␣calculated␣in␣{:d}␣repetitions".format(
        root, loops_counter))

print("Third␣root:")
for i in range(10):
    root, loops_counter = modified_bisection(f, 2.2, 2.4, 5)
    print("\tRoot:␣{:f}␣calculated␣in␣{:d}␣repetitions".format(
        root, loops_counter))
```

Στον παραπάνω κώδικα, ορίζω εκ νέου την συνάρτηση που υλοποιεί την τροποποιημένη μέθοδο διχοτόμησης και στην συνέχεια ορίζω την συνάρτηση f που υλοποιεί την f της οποίας τις ρίζες ζητήθηκε να βρώ στο ερώτημα 1 ($f(x) = 94cos^3x - 24cosx + 177sin^2x - 108sin^4x - 72cos^3xsin^2x - 65$). Στην συνέχεια εκτελώ για κάθε ρίζα την τροποποιημένη μέθοδο διχοτόμησης 10 φορές (με τα ίδια ορίσματα με τα οποία κλήθηκε στο ερώτημα 1) και τυπώνω τα αποτελέσματα.

Άν εκτελέσουμε τον κώδικα θα τυπωθεί:

```
First root:
    Root: 0.841067 calculated in 22 repetitions
    Root: 0.841056 calculated in 14 repetitions
    Root: 0.841067 calculated in 29 repetitions
    Root: 0.841053 calculated in 25 repetitions
    Root: 0.841245 calculated in 12 repetitions
    Root: 0.841115 calculated in 15 repetitions
    Root: 0.841068 calculated in 18 repetitions
    Root: 0.841020 calculated in 19 repetitions
    Root: 0.841074 calculated in 20 repetitions
    Root: 0.841087 calculated in 17 repetitions
Second root:
    Root: 1.047195 calculated in 11 repetitions
    Root: 1.047206 calculated in 7 repetitions
    Root: 1.047191 calculated in 19 repetitions
    Root: 1.047191 calculated in 11 repetitions
    Root: 1.047200 calculated in 16 repetitions
    Root: 1.047190 calculated in 18 repetitions
    Root: 1.047197 calculated in 16 repetitions
    Root: 1.047206 calculated in 20 repetitions
    Root: 1.047181 calculated in 21 repetitions
    Root: 1.047213 calculated in 19 repetitions
Third root:
    Root: 2.300524 calculated in 17 repetitions
    Root: 2.300521 calculated in 19 repetitions
    Root: 2.300522 calculated in 18 repetitions
    Root: 2.300522 calculated in 14 repetitions
    Root: 2.300491 calculated in 20 repetitions
    Root: 2.300525 calculated in 14 repetitions
    Root: 2.300527 calculated in 24 repetitions
    Root: 2.300530 calculated in 20 repetitions
    Root: 2.300507 calculated in 20 repetitions
    Root: 2.300518 calculated in 24 repetitions
```

Σύμφωνα με το παραπάνω διαπιστώνουμε ότι δεν συγκλίνει σε ίδιο σταθερό αριθμό επαναλήψεων.

6 Ερώτημα (3): Σύκριση ταχύτητας σύκλισης τροποποιημένων μεθόδων σε σχέση με τις κλασσικές

Ζητείται να γίνει συγκριση ως προς την ταχύτητα σύγκλισης των τροποιημένων μεθόδων σε σχέση με τις κλασικές με πειραματικό τρόπο.

Προγραμματιστικά αυτο γίνεται (σε γλώσσα python) στο αρχείο d_3_compare_with_classic.py το οποίο φαίνεται παρακάτω:

```
import math
from math import sin, cos
```

```python
import random


def modified_newton_raphson(function, function_derivative,
    function_second_derivative, starting_point,
    digits_of_precision):
    iteration_counter = 0

    f = function
    f_d = function_derivative
    f_dd = function_second_derivative

    x = starting_point

    while True:
        if f(x) == 0:
            return x, iteration_counter
        else:
            iteration_counter += 1
            x_next = x - (f(x) / f_d(x)) - (1/2) * ( (f(x) ** 2
                 * f_dd(x)) / (f_d(x)**3) )

            if abs(x_next - x) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x = x_next


def modified_bisection(function, range_beginning, range_ending
    , digits_of_precision):
    iteration_counter = 0

    f = function
    a = range_beginning
    b = range_ending
    fa = f(a)
    fb = f(b)

    r = random.uniform(a, b)
    fr = f(r)

    while True:
        if fr == 0:
```

```
            return r, iteration_counter
        elif fa * fr < 0:
            fb = fr
            b = r
        else:
            fa = fr
            a = r

        iteration_counter += 1

        old_r = r
        r = random.uniform(a, b)
        fr = f(r)

        if abs(old_r - r) < 0.5 * (10 ** (-1 *
            digits_of_precision)):
            return r, iteration_counter


def modified_secant(function, point_one, point_two,
    point_three, digits_of_precision):
    iteration_counter = 0

    f = function

    x1 = point_one
    x2 = point_two
    x3 = point_three

    while True:
        if f(x3) == 0:
            return x3, iteration_counter
        else:
            iteration_counter += 1

            q = f(x1) / f(x2)
            r = f(x3) / f(x2)
            s = f(x3) / f(x1)

            x_next = x3 - ( r * (r - q) * (x3 - x2) + (1 - r) *
                 s * (x3 - x1) ) / ( (q - 1) * (r - 1) * (s - 1)
                 )

            if abs(x_next - x3) < 0.5 * 10 ** (-1.0 *
```

```
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x1 = x2
                x2 = x3
                x3 = x_next


def classic_bisection(function, range_beginning, range_ending,
     digits_of_precision):
    iteration_counter = 0

    f = function
    a = range_beginning
    b = range_ending
    fa = f(a)
    fb = f(b)

    m = (a + b) / 2
    fm = f(m)

    while True:
        if fm == 0:
            return m, iteration_counter
        elif fa * fm < 0:
            fb = fm
            b = m
        else:
            fa = fm
            a = m

        iteration_counter += 1

        old_m = m
        m = (a + b) / 2
        fm = f(m)

        if abs(old_m - m) < 0.5 * (10 ** (-1 *
            digits_of_precision)):
            return m, iteration_counter

def classic_newton_raphson(function, function_derivative,
    starting_point, digits_of_precision):
    iteration_counter = 0
```

```
    f = function
    f_d = function_derivative

    x = starting_point

    while True:
        if f(x) == 0:
            return x, iteration_counter
        else:
            iteration_counter += 1
            x_next = x - f(x) / f_d(x)

            if abs(x_next - x) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x = x_next


def classic_secant(function, point_one, point_two,
    digits_of_precision):
    iteration_counter = 0

    f = function

    x1 = point_one
    x2 = point_two

    while True:
        if f(x2) == 0:
            return x2, iteration_counter
        else:
            iteration_counter += 1
            x_next = x2 - (f(x2) * (x2 - x1)) / (f(x2) - f(x1))

            if abs(x_next - x2) < 0.5 * 10 ** (-1.0 *
                digits_of_precision):
                return x_next, iteration_counter
            else:
                x1 = x2
                x2 = x_next
```

```
def f(x):
    return 94 * (cos(x) ** 3) - 24 * cos(x) + 177 * (sin(x) **
        2) - 108 * (sin(x) ** 4) - 72 * (cos(x) ** 3) * (sin(x)
         ** 2) - 65


def f_derivative(x):
    return - 282 * (cos(x) ** 2) * sin(x) + 24 * sin(x) + 354 *
         sin(x) * cos(x) - 432 * (sin(x) ** 3) * cos(x) + 216 *
         (cos(x) ** 2) * (sin(x) ** 3) - 144 * (cos(x) ** 4) *
        sin(x)


def f_second_derivative(x):
    return 564 * cos(x) * (sin(x) ** 2) - 282 * (cos(x) ** 3) +
         24 * cos(x) + 354 * (cos(x) ** 2) - 354 * (sin(x) **
        2) - 1296 * (cos(x) ** 2) * (sin(x) ** 2) + 432 * (sin(
        x) ** 4) - 432 * cos(x) * (sin(x) ** 4) \
           + 648 * (cos(x) ** 3) * (sin(x) ** 2) + 576 * (cos(x
              ) ** 3) * (sin(x) ** 2) - 144 * (cos(x) ** 5)


print("Bisection␣Comparison:\n")
print("Classic␣Bisection:")

root, loops_counter = classic_bisection(f, 0.8, 1.0, 5)
print("\tThe␣root␣in␣[0.8,␣1.0]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = classic_bisection(f, 1.0, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = classic_bisection(f, 2.2, 2.4, 5)
print("\tThe␣root␣in␣[2.2,␣2.4]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

print("Modified␣Bisection:")
root, loops_counter = modified_bisection(f, 0.8, 1.0, 5)
print("\tThe␣root␣in␣[0.8,␣1.0]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_bisection(f, 1.0, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
```

```
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_bisection(f, 2.2, 2.4, 5)
print("\tThe␣root␣in␣[2.2,␣2.4]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

print("
    --------------------------------------------------------------------------------
    ")

print("Newton-Raphson␣Comparison:\n")

print("Classic␣Newton-Raphson:")
root, loops_counter = classic_newton_raphson(f, f_derivative,
    0.8, 5)
print("\tThe␣root␣near␣0.8:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = classic_newton_raphson(f, f_derivative,
    1.0, 5)
print("\tThe␣root␣near␣1.0:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = classic_newton_raphson(f, f_derivative,
    2.3, 5)
print("\tThe␣root␣near␣2.3:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

print("Modified␣Newton-Raphson:")
root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 0.8, 5)
print("\tThe␣root␣near␣0.8:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 1.0, 5)
print("\tThe␣root␣near␣1.0:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))

root, loops_counter = modified_newton_raphson(f, f_derivative,
     f_second_derivative, 2.3, 5)
print("\tThe␣root␣near␣2.3:␣{:f}.␣It␣was␣calculated␣in␣{:d}␣
    repetitions".format(root, loops_counter))
```

```
print("
    ------------------------------------------------------------------------------------
    ")

print("\nSecant␣Comparison:\n")
print("Classic␣Secant:")
root, loops_counter = classic_secant(f, 0.8, 0.9, 5)
print("\tThe␣root␣in␣[0.8,␣0.9]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = classic_secant(f, 1.0, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = classic_secant(f, 2.0, 2.5, 5)
print("\tThe␣root␣in␣[2.0,␣2.5]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

print("Modified␣Secant:")
root, loops_counter = modified_secant(f, 0.8, 0.85, 0.9, 5)
print("\tThe␣root␣in␣[0.8,␣0.9]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_secant(f, 1.0, 1.1, 1.2, 5)
print("\tThe␣root␣in␣[1.0,␣1.2]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))

root, loops_counter = modified_secant(f, 2.0, 2.25, 2.5, 5)
print("\tThe␣root␣in␣[2.0,␣2.5]:␣{:f}.␣It␣was␣calculated␣in␣{:
    d}␣repetitions".format(root, loops_counter))
```

Στον παραπάνω κώδικα, αρχικά ορίζονται εκ νέου οι συναρτήσεις που υλοποιούν τις τροποποιημένες μεθόδους.
Στην συνέχεια, ορίζω τις μεθόδους classic_bisection, classic_newton_raphson και classic_secant που υλοποιούν τις κλασσικές μεθόδους διχοτόμησης, Newton-Raphson και τέμνουσας αντίστοιχα. Ο κώδικας τον οποίον η κάθε μία υλοποιεί είναι πανομοιότυπος με εκείνον της αντίστοιχης μεθόδου του αντίστοιχου ερωτήματος της άσκησης 1 και για αυτό παραλείπεται η ανάλυση του.
Στην συνέχεια ορίζω τις συναρτήσεις f, f_derivative, f_second_derivative που υλοποιούν την συνάρτηση f και την πρώτη και δεύτερη παράγωγο της αντίστοιχα. Ως f ορίζεται η συνάρτηση της οποίας οι ρίζες ζητήθηκε να βρεθούν στο ερώτημα 1 ($f(x) = 94cos^3x - 24cosx + 177sin^2x - 108sin^4x - 72cos^3xsin^2x - 65$)

Τέλος, για κάθε μέθοδο (κλασσική και έπειτα τροποποιημένη) βρίσκω κάθε ρίζα με τις κατάλληλες παραμέτρους που είναι ίδιες με αυτές που χρησιμοποιήθηκαν για εύρεση των ριζών της f στο ερώτημα 1 και τυπώνω τα αποτελέσματα κάθε εκτέλεσης.

Άν εκτελέσουμε το αρχείο θα εμφανιστεί:

```
Bisection Comparison:

Classic Bisection:
    The root in [0.8, 1.0]: 0.841068. It was calculated in 15 repetitions
    The root in [1.0, 1.2]: 1.047211. It was calculated in 14 repetitions
    The root in [2.2, 2.4]: 2.300522. It was calculated in 15 repetitions
Modified Bisection:
    The root in [0.8, 1.0]: 0.841057. It was calculated in 24 repetitions
    The root in [1.0, 1.2]: 1.047206. It was calculated in 16 repetitions
    The root in [2.2, 2.4]: 2.300525. It was calculated in 15 repetitions
---------------------------------------------------------------------------
Newton-Raphson Comparison:

Classic Newton-Raphson:
    The root near 0.8: 0.841069. It was calculated in 5 repetitions
    The root near 1.0: 1.047184. It was calculated in 20 repetitions
    The root near 2.3: 2.300524. It was calculated in 2 repetitions
Modified Newton-Raphson:
    The root near 0.8: 0.841069. It was calculated in 4 repetitions
    The root near 1.0: 1.047187. It was calculated in 14 repetitions
    The root near 2.3: 2.300524. It was calculated in 2 repetitions
---------------------------------------------------------------------------


Secant Comparison:

Classic Secant:
    The root in [0.8, 0.9]: 0.841069. It was calculated in 11 repetitions
    The root in [1.0, 1.2]: 1.047186. It was calculated in 30 repetitions
    The root in [2.0, 2.5]: 2.300524. It was calculated in 7 repetitions
Modified Secant:
    The root in [0.8, 0.9]: 0.841069. It was calculated in 6 repetitions
    The root in [1.0, 1.2]: 1.047182. It was calculated in 23 repetitions
    The root in [2.0, 2.5]: 2.300524. It was calculated in 5 repetitions
```

Για την μέθοδο της διχοτόμησης:

- Για την ρίζα στο $[0.8, 1.0]$ η κλασσική μέθοδος συγκλίνει $24 - 15 = 9$ επαναλήψεις πιο γρήγορα σε σχέση με την τροποποιημένη μέθοδο.
- Για την ρίζα στο $[1.0, 1.2]$ η κλασσική μέθοδος συγκλίνει $16 - 14 = 2$ επαναλήψεις πιο γρήγορα σε σχέση με την τροποποιημένη μέθοδο.

- Για την ρίζα στο $[2.2, 2.4]$ η κλασσική μέθοδος συγκλίνει με τον ίδιο αριθμό επαναλήψεων σε σχέση με την τροποποιημένη μέθοδο.

Επομένως, απο τα παραπάνω διαπιστώνουμε ότι για την μέθοδο της διχοτόμησης η κλασσική μέθοδος είναι, εν γένει, πιο γρήγορη απο την τροποποιημένη.

Για την μέθοδο Newton-Raphson:

- Για την ρίζα κοντά στο 0.8 η τροποποιημένη μέθοδος συγκλίνει $5-4=1$ επανάληψη πιο γρήγορα σε σχέση με την κλασσική μέθοδο.
- Για την ρίζα κοντά στο 1.0 η τροποποιημένη μέθοδος συγκλίνει $20-14=6$ επαναλήψεις πιο γρήγορα σε σχέση με την κλασσική μέθοδο.
- Για την ρίζα κοντά στο 2.3 η τροποποιημένη μέθοδος συγκλίνει με τον ίδιο αριθμό επαναλήψεων με την κλασσική μέθοδο.

Επομένως, απο τα παραπάνω διαπιστώνουμε ότι για την μέθοδο Newton-Raphson η τροποποιημένη μέθοδος είναι, εν γένει, πιο γρήγορη απο την κλασσική.

Για την μέθοδο της τέμνουσας:

- Για την ρίζα στο $[0.8, 0.9]$ η τροποποιημένη μέθοδος συγκλίνει $11-6=5$ επαναλήψεις πιο γρήγορα σε σχέση με την κλασσική μέθοδο.
- Για την ρίζα στο $[1.0, 1.2]$ η τροποποιημένη μέθοδος συγκλίνει $30-23=7$ επαναλήψεις πιο γρήγορα σε σχέση με την κλασσική μέθοδο.
- Για την ρίζα στο $[2.2, 2.5]$ η τροποποιημένη μέθοδος συγκλίνει $7-5=2$ επαναλήψεις πιο γρήγορα σε σχέση με την κλασσική μέθοδο.

Επομένως, απο τα παραπάνω διαπιστώνουμε ότι για την μέθοδο της τέμνουσας η τροποποιημένη μέθοδος είναι πιο γρήγορη απο την κλασσική.