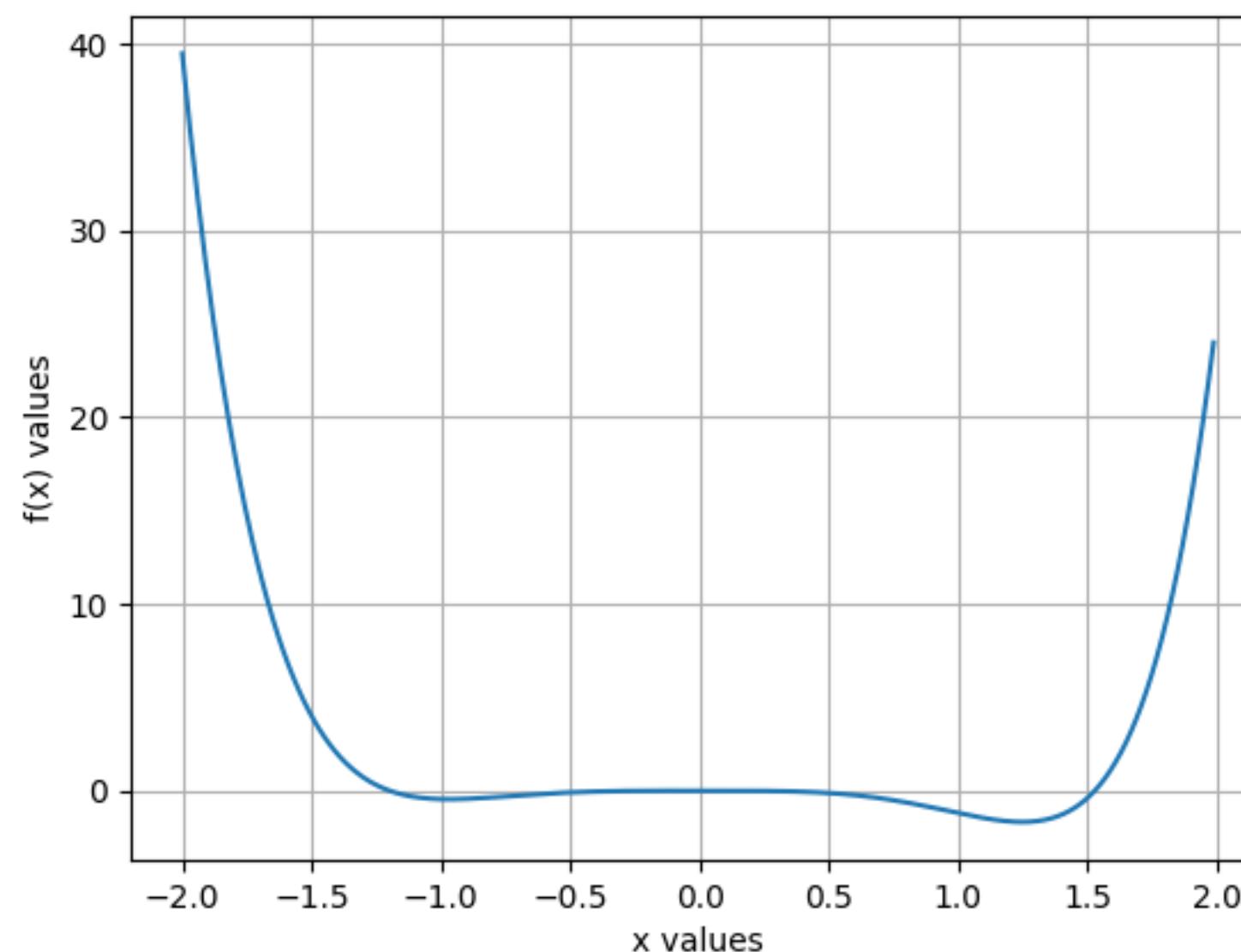


## ΑΣΚΗΣΗ 1



α) Στην διχοτόμηση δημιουργησα μία συνάρτηση bisection η οποία δέχεται ένα a και ένα b ως παραμέτρους ώστε να έχουμε το αρχικό μας διάστημα. Στην συνάρτηση αυτήν πρώτα αρχικοποίησα μια μεταβλητή m ( οποία θα σε κάθε επανάληψη θα πέρνει την τιμή του μέσου του διαστήματος ) ως None. Επίσης έθεσα μια μεταβλητή N=1 η οποία θα αποθηκεύει τον αριθμό των επαναλήψεων όπως και μια μεταβλητή error\_bound=0.001 ώστε να ξέρουμε πότε βρέθηκε η προσέγγιση της ρίζας, και τέλος μια μεταβλητή root η οποία θα αποθηκεύσει την ρίζα της συνάρτησης όταν βρεθεί και η αρχική τιμή αυτής της μεταβλητής είναι επίσης None. Μετά την αρχικοποίηση αυτών των μεταβλητών μπαίνουμε μέσα σε ένα while loop το οποίο σταματά μόνο άμα  $(a-b)/2 \leq 2^{**N}$  ή άμα  $f(m)=0$ . Σε αυτό το while loop πρώτα κάνουμε το  $m = (a+b)/2$  και μετά καλούμε την συνάρτηση f όπου για x στέλνουμε το m. Αν  $f(m)==0$  θέτουμε  $root=m$  και σπάμε την επανάληψη αλλιώς συνεχίζουμε στο επόμενο βήμα το οποίο είναι να ελέγχουμε το πρόσημο της  $f(m)*f(a)$  και αν αυτό είναι μικρότερο του 0 τότε θέτουμε το  $b = m$  αλλιώς  $a = m$  και αυξάνουμε κατά 1 το N. Όταν βγούμε από το loop ελέγχουμε αν το  $root==None$  και αν είναι το θέτουμε στο m ( την πιο κοντινή προσέγγιση στην ρίζα δηλαδή ). Και στο τέλος επιστρέφουμε την ρίζα και τον αριθμό των επαναλήψεων που κάναμε. Η ρίζα που βρέθηκε με την συνάρτηση είναι το 0 το οποίο βρέθηκε μετά από 1 επανάληψη.

```

def bisection(a, b):
    m = None
    N = 1

    error_bound = 0.001

    root = None

    while (b - a) / 2**N > error_bound:
        m = (a + b) / 2
        fm = f(m)

        if fm == 0:
            root = m
            break

        if fm * f(a) < 0:
            b = m
        else:
            a = m

        N += 1

    if root == None:
        root = m

    return (keepTheFirstNDecimals(5, root), N)

```

```

def keepTheFirstNDecimals(n, num):
    if len(str(num).split(".")) == 1 or len(str(num).split(".")[1]) <= n:
        return num
    else:
        before_decimal = str(num).split(".")[0]
        after_decimal = str(num).split(".")[1]
        return np.float(before_decimal + "." + after_decimal[0:n])

```

β) Στην Newton-Raphson δημιούργησα μία συνάρτηση newton η οποία η μόνη παράμετρος που δέχεται είναι το αρχικό x. Στην συνάρτηση αυτή θέτουμε μια μεταβλητή x\_current = x0, μια μεταβλητή x\_new = None, έναν άδειο πίνακα x\_list = [] ο οποίος θα αποθηκεύει σε κάθε επανάληψη τα x ώστε να μπορούμε να βρούμε την σύγκλιση, μια μεταβλητή error\_bound = 0.001 και μια μεταβλητή N = 1. Μετά μπαίνουμε σε while loop. Στο while loop κάθε φορά θα παίρνουμε το καινούργιο x με τον τύπο του Newton ( $x_{n+1} = x_n - f(x)/f'(x)$ ). Το x αυτό θα το παίρνουμε μέσα σε try catch block γιατί υπάρχει περίπτωση το  $f'(x)$  να είναι 0 ( που στην περίπτωση αυτήν είναι και ρίζα της εξίσωσης το 0 αλλά είναι και τοπικό ακρότατο οπότε το  $f'(0)$  θα μας δώσει ZeroDivisionError). Αφού πάρουμε το καινούργιο x ελέγχουμε αν η διαφορά του με το προηγούμενο είναι  $\leq$  error\_bound ή αν  $f(x_{new}) = 0$  και αν ισχύει αυτό καλούμε την συνάρτηση getConverganceRate για να πάρουμε την σύγκλιση της ρίζας και επιστρέφουμε την ρίζα, τον αριθμό των επαναλήψεων και την σύγκλιση, αλλιώς βάζουμε το x\_new στο x\_list, θέτουμε το x\_current = x\_new και αυξάνουμε το N κατά 1. Εγώ κάλεσα 2 φορές την συνάρτηση με αρχικά x το -2 και το 1. Στο -2 η ρίζα που βρέθηκε ήταν το -1.1976 με σύγκλιση 2 μετά από 7 αριθμούς επαναλήψεων ενώ για το 1 η ρίζα που βρέθηκε ήταν το 0.0023 με σύγκλιση 1.5 μετά από 20 αριθμούς επαναλήψεων.

```
def newton(x_guess):
    x_current = x_guess
    error_bound = 0.001
    x_new = None
    N = 1
    x_list = []

    while True:
        try:
            x_new = x_star(x_current)
        except ZeroDivisionError:
            print("Local max or local min. f'(x) = 0")
            return (None, None, None)

        if(np.abs(x_current - x_new) <= error_bound) or f(x_new) == 0:
            x_new = keepTheFirstNDecimals(5, x_new)
            convergence = getConvergeRate(getFixedValues(x_list), x_new)
            return (x_new, N, convergence[len(convergence) - 1])

        x_list.append(x_new)
        x_current = x_new
        N += 1
```

```
def getFixedValues(arr):
    temp = []

    for num in arr:
        temp.append(keepTheFirstNDecimals(5, num))

    return temp
```

```
def x_star(x):
    return x - (f(x) / df(x))
```

```
def getConvergeRate(x_list, root):
    errors = [np.abs(x - root) for x in x_list]
    q = [np.log(errors[n+1] / errors[n]) / np.log(errors[n] / errors[n-1]) for n in range(1, len(errors)-1, 1)]
    return q
```

γ) Στην Secant δημιουργησα μια συνάρτηση secant η οποία δέχεται 2 x ως παραμέτρους τα οποία θα είναι τα 2 αρχικά μας σημεία. Στην συνάρτηση αυτή αρχικά θέτω ένα  $x_0$  και ένα  $x_1$  ίσα με τις δύο παραμέτρους που δέχεται η συνάρτηση και μετά θέτω ένα error\_bound = 0.001 και ένα N=1. Μετά μπαίνουμε στο while loop το οποίο πρώτα υπολογίζει το καινούργιο x και έπειτα ελέγχει αν η απόλυτη τιμή της διαφοράς του καινούργιου x μείον το προηγούμενο x είναι  $\leq$  error\_bound ή  $f(x_{new}) = 0$  και άμα ισχύει ένα από τα δύο τότε επιστρέφουμε το  $x_{new}$  και το N. Αν δεν ισχύει απλά θέτουμε το  $x_0 = x_1$  και το  $x_1 = x_{new}$  και αυξάνουμε το N κατά 1. Στο πρόγραμμα μου κάλεσα την συνάρτηση 2 φορές, την πρώτη φορά με αρχικά σημεία τα -2 και -1 τα οποία μου έδωσαν την ρίζα -1.19762 μετά από 13 επαναλήψεις και την δεύτερη φορά με αρχικά σημεία 1 και 2 τα οποία μου έδωσαν την ρίζα 0.00379 μετά από 28 επαναλήψεις.

```
def secant(x0_guess, x1_guess):
    x0 = x0_guess
    x1 = x1_guess
    N = 1

    error_bound = 0.001

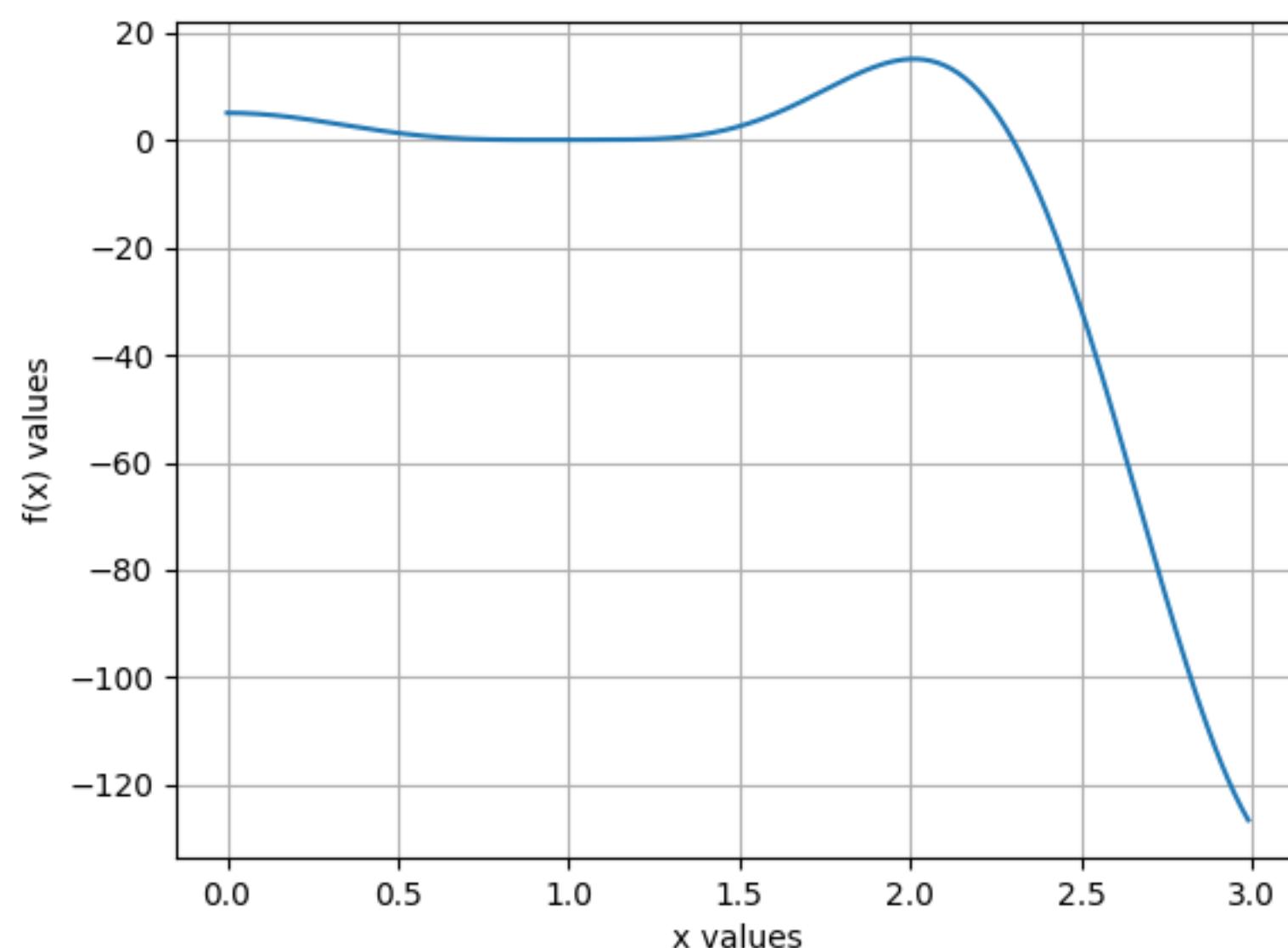
    while True:
        x_new = next_x(x1, x0)

        if np.abs(x1 - x_new) <= error_bound or f(x_new) == 0:
            return (keepTheFirstNDecimals(5, x_new), N)

        x0 = x1
        x1 = x_new
        N += 1

def next_x(x, prev_x):
    return x - ((f(x) * (x - prev_x)) / (f(x) - f(prev_x)))
```

## ΑΣΚΗΣΗ 2



1) Στον αλγόριθμο μου το μόνο που άλλαξε είναι για κάθε μέθοδο ο τρόπος που δίνει το επόμενο  $x$ , όλα τα άλλα έμειναν ακριβώς το ίδιο. Οπότε απλά χρησιμοποιώντας τις τροποποιημένες μεθόδους σε κάθε συνάρτηση (τις οποίες ονόμασα `bisection_changed`, `newton_changed` και `secant_changed` ώστε να μπορώ να χρησιμοποιήσω και τις παλιές για να συγκρίνω ως προς την σύγκλιση) πήρα τις εξεις ρίζες:

Η `newton_changed` μου έδωσε για τα σημεία 1.5 και 1.8 τις ρίζες 2.30027 και 2.30072.

Η `bisection_changed` μου έδωσε (αφού πήρα την μέση τιμή από τις ρίζες που επέστρεψε εφόσον το δεύτερο ερώτημα ήθελε να την εκτελέσω 10 φορές) την ρίζα 1.2306439999999998

Η `secant_changed` μου έδωσε για τα σημεία (1.5, 1.8, 2) και (.7, 1.98, 2.4) τις ρίζες 1.04907 και 0.84106

2) Μετά την εκτέλεση του αλγορίθμου 10 φορές, παρατήρησα αφού πήρα και την μέση τιμή όλων των ριζών που επιστράφηκαν από την συνάρτηση ότι τις περισσότερες φορές ο αλγόριθμος συγκλίνει στον ίδιο αριθμό επαναλήψεων.

3) Η τροποποιημένη μέθοδος `newton` έκανε λιγότερες επαναλήψεις από την μη τροποποιημένη βρίσκοντας μια αρκετά καλή προσέγγιση της ρίζας αλλά ο αριθμός σύγκλισης της ήταν αρκετά μικρός σε σχέση με την μη τροποποιημένη, δηλαδή για την τιμή 1.5 η τροποποιημένη είχε σύγκλιση 0.026 ενώ η μη τροποποιημένη είχε σύγκλιση 1.66312 και για την τιμή 1.8 η τροποποιημένη είχε σύγκλιση 0.10886 ενώ η μη τροποποιημένη είχε σύγκλιση 1.66801.

Στην `secant` και η τροποποιημένη και η μη τροποποιημένη είχαν σχεδόν την ίδια σύγκλιση και τον ίδιο αριθμό επαναλήψεων στα ίδια  $x$  που πήραν ως παραμέτρους.

Στην `bisection` η τροποποιημένη μέθοδος κάνει κατά μέσο όρο λιγότερες επαναλήψεις από την μη τροποποιημένη και αρκετές φορές βρίσκει μια καλή προσέγγιση της ρίζας οπότε έχει μεγαλύτερη σύγκλιση από την μη τροποποιημένη

```
def newton_changed(x_guess):
    x_current = x_guess
    error_bound = 0.001
    x_new = None
    N = 1
    x_list = []

    while True:
        try:
            x_new = x_star_changed(x_current)
        except ZeroDivisionError:
            print("Local max or local min. f'(x) = 0")
            return (None, None, None)

        if(np.abs(x_current - x_new) <= error_bound) or f(x_new) == 0:
            x_new = keepTheFirstNDecimals(5, x_new)
            convergence = getFixedValues(getConvergeRate(x_list, x_new))
            return (x_new, N, convergence[len(convergence) - 1])

        x_list.append(x_new)
        x_current = x_new
        N += 1

def x_star_changed(x):
    return x - 0.5*((f(x)**2)*d2f(x)) / df(x)**3
```

### ΑΣΚΗΣΗ 3

1) Στην αρχή της συνάρτησης μου αρχικοποιώ μια μεταβλητή  $n = \text{len}(b)$  για να ξέρω πόσες γραμμές έχει ο πίνακας μου. Μετά θέτω τον πίνακα  $A$  που δέχεται η συνάρτηση ίσο με το αποτέλεσμα της συνάρτησης `correctPivots` η οποία δέχεται και αυτή ως παραμέτρους έναν πίνακα  $A$  και έναν  $b$  και αλλάζει τις σειρές του πίνακα αν χρειαστεί ώστε στην διαγώνιο να μην υπάρχουν μηδενικά. Δηλαδή ο πίνακας που χρησιμοποίησα εγώ θα γίνει :

$$\begin{array}{l}
 \text{Αρχικός} \\
 \text{πίνακας } A
 \end{array}
 \xrightarrow{\text{correctPivots}(A, b)}
 \begin{bmatrix}
 0 & 7 & -1 & 3 & 1 \\
 0 & 3 & 4 & 1 & 7 \\
 6 & 2 & 0 & 2 & -1 \\
 2 & 1 & 2 & 0 & 2 \\
 3 & 4 & 1 & -2 & 1
 \end{bmatrix}
 \rightarrow
 \begin{bmatrix}
 6 & 2 & 0 & 2 & -1 \\
 0 & 3 & 4 & 1 & 7 \\
 0 & 7 & -1 & 3 & 1 \\
 3 & 4 & 1 & -2 & 1 \\
 2 & 1 & 2 & 0 & 2
 \end{bmatrix}$$

Αφού ο πίνακας έχει διορθωθεί έτσι ώστε να μην έχει κανένα μηδενικό στην διαγώνιο η συνάρτηση συνεχίζει με ένα διπλό for loop τα οποία θα κάνουν 0 όλες τις τιμές κάτω από την διαγώνιο. Το πρώτο πράγμα μέσα που γίνεται μέσα στο loop είναι να ελέγξει αν η τιμή του πίνακα που βρισκόμαστε τώρα είναι 0, και αν ισχύει αυτό θα χρησιμοποιήσουμε την εντολή `continue` ώστε να μην γίνει κάτι στο συγκεκριμένο index του loop. Αν δεν ισχύει τότε θα δημιουργήσει μια μεταβλητή  $factor = A[k, k] / A[i, k]$  (όπου  $k$  το index της πρώτης επανάληψης το οποίο πάει μέχρι και το προτελευταίο στοιχείο του πίνακα, και  $i$  το index της δεύτερης επανάληψης το οποίο ξεκινάει από  $k+1$  και πάει μέχρι και το τελευταίο στοιχείο του πίνακα). Όταν πάρουμε αυτήν την μεταβλητή τότε θα χρειαστούμε και ένα τρίτο for loop το οποίο θα έχει ένα  $j$  index το οποίο θα πηγάινει από  $k$  μέχρι το τελευταίο στοιχείο του πίνακα και θα θέτει  $A[i][j] = A[k, j] - A[i, j] * factor$  το οποίο στην ουσία απλά θα μηδενίζει την συγκεκριμένη θέση και αφού τελειώσει αυτό το τρίτο loop τότε θα αλλάξουμε και τον πίνακα  $b$ , δηλαδή  $b[i] = b[k] - b[i] * factor$ . Ένα μικρό παράδειγμα του αλγορίθμου είναι αυτό:

$k=0$

$i=3$

τότε  $factor = A[0, 0] / A[3, 0] = 6 / 3 = 2$

οπότε το τρίτο for loop θα κάνει το εξής:

`for j in range(k, n):` (όπου  $n = 5$ )

$A[3, j] = A[0, j] - A[3, j] * factor$  (το οποίο θα μηδενίσει μόνο τις τιμές κάτω από το index  $k$  ενώ τις άλλες απλά θα τις αλλάξει)

Για  $j=0$ :  $A[3, 0] = A[0, 0] - A[3, 0] * 2 = 6 - 3 * 2 = 0$

Για  $j=1$ :  $A[3, 1] = A[0, 1] - A[3, 1] * 2 = 2 - 4 * 2 = -6$

Για  $j=2$ :  $A[3, 2] = A[0, 2] - A[3, 2] * 2 = 0 - 1 * 2 = -2$

Για  $j=3$ :  $A[3, 3] = A[0, 3] - A[3, 3] * 2 = 2 + 2 * 2 = 6$

Για  $j=4$ :  $A[3, 4] = A[0, 4] - A[3, 4] * 2 = -1 - 1 * 2 = -3$

Οπότε από ότι φαίνεται ο αλγόριθμος μηδένισε σε αυτήν την σειρά μόνο το  $k$  index. Στην τελευταία σειρά θα γινόταν πάλι το ίδιο και το 2 στο index 0 θα μηδενιζόταν.

Οπότε αφού τελειώσουν αυτά τα loops στην συνέχεια θα δημιουργηθεί ένας πίνακας  $x$  όπου στην αρχή θα είναι όλα 0. Μετά θα χρησιμοποιήσουμε back substitution για να βρίσκουμε ένα ένα τα  $x$  ξεκινώντας από το τελευταίο. Πρώτα θα πάρουμε απευθείας το τελευταίο  $x$  με την πράξη  $x[n-1] = b[n-1] / A[n-1, n-1]$  και στην συνέχεια θα μπούμε σε ακόμα ένα for loop με index  $i$  το οποίο θα ξεκινά από  $n-2$  και θα πηγαίνει μέχρι 0 με βήμα -1. Μέσα σε αυτό το loop πρώτα θα αρχικοποιήσουμε μια μεταβλητή  $sum_ax = 0$  στην οποία με ένα loop θα προσθέσουμε όλα τα στοιχεία του πίνακα της συγκεκριμένης σειράς πολλαπλασιάζοντας τα με το  $x$  τους και στο τέλος το  $x[i]$  θα είναι ίσο με το συγκεκριμένο  $b$  μείον το  $sum_ax$  και αυτό θα το διαιρέσουμε με το  $A[i, i]$  δηλαδή  $x[i] = (b[i] - sum_ax) / A[i, i]$ .

```

def gauss(A, b):
    n = len(b) # Matrix size

    A = correctPivots(A, b) ←

    for k in range(n-1):
        for i in range(k+1, n):
            if A[i, k] == 0: continue

            factor = A[k, k] / A[i, k]

            for j in range(k, n):
                A[i, j] = A[k, j] - A[i, j]*factor

            b[i] = b[k] - b[i]*factor

```

```

def correctPivots(A, b):
    n = len(b) # Matrix size

    for k in range(n-1):
        if np.fabs(A[k, k]) < 1.0e-10:
            for i in range(k+1, n):
                if np.fabs(A[i, k]) > np.fabs(A[k, k]):
                    A[[k, i]] = A[[i, k]]
                    b[[k, i]] = b[[i, k]]
                    break

    return A

```

```

# Initialize our x variables
x = np.zeros(n, float)

```

```

# Back-substitution
x[n-1] = b[n-1] / A[n-1, n-1]
for i in range(n-2, -1, -1):
    sum_ax = 0
    for j in range(i+1, n):
        sum_ax += A[i, j]*x[j]
    x[i] = (b[i] - sum_ax) / A[i, i]

```

```

return x

```

2) Στην συνάρτηση cholesky που δημιούργησα η οποία απλά δέχεται ως παράμετρο έναν συμμετρικό και θετικά ορισμένο πίνακα A απλά αρχικοποιεί έναν πίνακα L ο οποίος έχει ίδιο μέγεθος σειρών και στήλων με τον A με μηδενικά. Στην συνέχεια γίνεται ένα διπλό for loop ώστε στην συγκεκριμένη σειρά και στήλη του L να βάλουμε την σωστή τιμή με τον τύπο της αποσύνθεσης του Cholesky ο οποίος είναι ο εξής:

$$L_{j,j} = (\pm) \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2},$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \quad \text{for } i > j.$$

```

def cholesky(A):
    A = np.array(A, float)
    L = np.zeros_like(A)

    n,_ = np.shape(A)

    for j in range(n):
        for i in range(j, n):
            if i == j:
                L[i, j] = np.sqrt(A[i, j] - np.sum(L[i, :j]**2))
            else:
                L[i, j] = (A[i, j] - np.sum(L[i, :j]*L[j, :j])) / L[j, j]

    return L

```

3) Η συνάρτηση gaussSeidel που δημιούργησα (η οποία δέχεται δύο παραμέτρους, η μία είναι ο πίνακας A και η άλλη είναι ο πίνακας b) ξεκινά αρχικοποιώντας κάποιες μεταβλητές, οι οποίες είναι το error\_bound=0.005, το n = len(b), τον πίνακα x = np.zeros\_like(b, float) στον οποίο θα αποθηκεύουμε τις λύσεις μας, το prev\_x = np.copy(x) το οποίο θα το χρησιμοποιούμε για να ελέγξουμε αν τα επόμενα x μας είναι ίδια με τα προηγούμενα για να σταματήσουμε τις επαναλήψεις και την loops = 1 η οποία θα κρατάει τον αριθμό των επαναλήψεων. Στην συνέχεια μπαίνουμε σε while loop το οποίο θα εκτελεστεί το πολύ 50 φορές, και μέσα σε αυτό το while loop θα έχουμε και ένα for loop το οποίο θα ξεκινά από 0 και θα πηγαίνει μέχρι το τελευταίο στοιχείο του πίνακα x γιατί στην ουσία σε κάθε επανάληψη θα λύνουμε ως προς  $x_i$ . Μέσα σε αυτό το for loop θα αρχικοποιούμε κάθε φορά μια μεταβλητή sum\_x στην οποία θα προσθέτουμε όλα τα υπόλοιπα x με ένα άλλο for loop και μετά το x στο οποίο βρισκόμαστε θα γίνει:

$x[i] = \text{keepTheFirstNDecimals}(4, (b[i] - \text{sum}_x) / A[i, i])$  (Αυτή η συνάρτηση αναφέρθηκε και πιο πάνω)

και αφού τελειώσει το αρχικό for loop τότε πρωτού ξαναμπούμε στην επόμενη επανάληψη θα ελέγξουμε αν η διαφορά του αθροίσματος των πινάκων x και prev\_x σε απόλυτη τιμή είναι  $\leq \text{error\_bound}$  και άμα ισχύει αυτό τότε απλά θα σπάμε το loop και θα εκτυπώσουμε τα x αλλιώς θα αυξήσουμε την μεταβλητή loops κατά ένα και ο πίνακας prev\_x θα γίνει prev\_x = np.copy(x).

```

def gaussSeidel(A, b):
    error_bound = 0.005
    n = len(b)
    x = np.zeros_like(b, float)
    prev_x = np.copy(x)
    loops = 1

    while loops < 50:
        for i in range(n):
            sum_x = 0

            for j in range(n):
                if i != j:
                    sum_x = sum_x + (A[i, j] * x[j])

            x[i] = keepTheFirstNDecimals(4, (b[i] - sum_x) / A[i, i])

            if np.abs(np.sum(x) - np.sum(prev_x)) <= error_bound:
                break

        prev_x = np.copy(x)
        loops += 1

    print("Οι προσεγγίσεις των λύσεων είναι: ")
    print(x)
    print("Επαναλήψεις που έγιναν: ", loops)

```

## ΑΣΚΗΣΗ 4

1) Κάθε επόμενο στάδιο του πίνακα των τάξεων των σελίδων εξαρτάται μόνο από το τωρινό στάδιο. Δηλαδή ο πίνακας των τάξεων μετά από 30 επαναλήψεις, για να μας δώσει το επόμενο στάδιο που θα υπολογισθεί στην 31 επανάληψη θα εξαρτάται μόνο από την επανάληψη 30, δεν θα μας νοιάζει πως ήταν ο πίνακας στις υπόλοιπες. Οπότε από αυτό καταλαβαίνουμε πως ο πίνακας είναι Μαρκοβιανός, και ξέρουμε πως ένας Μαρκοβιανός πίνακας είναι στοχαστικός, οπότε ο πίνακας G είναι στοχαστικός.

2) Αφού έφτιαξα τον αλγόριθμο και τον έτρεξα είδα πως όντως οι τιμές ήταν σχεδόν ίδιες (είχαν υπερβολικά μικρό σφάλμα εφόσον η python δεν μπορεί να χειριστεί άφογα τους δεκαδικούς.) Πρωτού τρέξω τον αλγόριθμο όμως έκανα κάποιες αλλαγές στον αρχικό πίνακα A, δηλαδή έπρεπε να αλλάξω τις τιμές που ήταν 1 ώστε το συνολικό άθροισμα κάθε σειράς να ήταν 1. Επίσης δημιούργησα και έναν πίνακα R ο οποίος είχε μέγεθος 15 (όσο οι σειρές του A) και όλες του οι τιμές ήταν  $1 / \text{μέγεθος } A$ , γιατί στην αρχή όλες οι σελίδες έχουν ίδια τάξη (αυτό το έκανα για όλα τα ερωτήματα).

Οι τιμές που βρήκε ο αλγόριθμος είναι αυτές:

```
[0.02682453, 0.02986105, 0.02986105, 0.02682453, 0.03958715, 0.03958714, 0.03958715, 0.03958715, 0.07456423,  
0.10631975, 0.10631973, 0.07456423, 0.12509139, 0.11632765, 0.1250914]
```

```
def PageRank(A, R, n, q):  
    power = 1  
    PR = np.copy(R)  
    prev_PR = np.copy(PR)  
  
    while True:  
        for i in range(n):  
            linked_websites = []  
  
            for k, link in enumerate(A[:, i]):  
                if link != 0:  
                    linked_websites.append(k)  
  
            PR[i] = getPageRank(A, PR, linked_websites, q)  
  
        if stopIterations(PR, prev_PR):  
            break  
  
        prev_PR = np.copy(PR)  
        power += 1  
  
    return (PR, power)
```

```
def getPageRank(A, PR, linked_websites, q):  
    rank = (1 - q) / n  
  
    if len(linked_websites) != 0:  
        for j in linked_websites:  
            rank += q * (PR[j] / len(list(filter(lambda x: x != 0, A[j]))))  
  
    return rank
```

```
def stopIterations(PR, prev_PR):  
    count = 0  
    margin_error = 1.0e-7  
  
    for i in range(len(PR)):  
        if np.fabs(PR[i] - prev_PR[i]) <= margin_error:  
            count += 1  
  
    return count == len(PR)
```

3) Οι νέες συνδέσεις που έβαλα είναι στα σημεία (14, 0), (13, 0), (12, 0) και (9, 0), και η διαγραφή σύνδεσης που έκανα ήταν στο σημείο (3, 7). Σκοπός μου ήταν να βάλω στην καλύτερη τάξη την ιστοσελίδα 1 και αφού έτρεξα το πρόγραμμα και πήρα πίσω τον πίνακα με τις τάξεις κάλεσα μία συνάρτηση που έφτιαξα η οποία δέχεται έναν πίνακα με τάξεις και επιστρέφει ένα λεξιλόγιο με κλειδί της ιστοσελίδας και τιμή την τάξη της και αυτό ταξινομημένο, και παρατήρησα πως όντως η ιστοσελίδα 1 πήγε στην καλύτερη τάξη.

Τα αποτελέσματα της συνάρτησης ήταν αυτά:

```
{0: 0.1357669409655188, 9: 0.11098247863329593, 14: 0.09014716489066976, 8: 0.08651664566652875,
1: 0.08581157425288524, 10: 0.08341930347723986, 12: 0.06640831358386824, 4: 0.05882634082341906,
13: 0.054357393735387574, 5: 0.052623679995002476, 6: 0.046735082388783114, 11: 0.04384164408897119,
2: 0.04261322010172763, 7: 0.022421803017132293, 3: 0.019529268711403036}
```

Ενώ τα προηγούμενα αποτελέσματα πρωτού κάνω τις αλλαγές ήταν αυτά:

```
{14: 0.12509139798407176, 12: 0.1250913914925863, 13: 0.11632764815781536, 9: 0.10631974738474384,
10: 0.1063197326476894, 11: 0.07456423039737362, 8: 0.07456422639149599, 6: 0.039587146617658775,
7: 0.03958714600874552, 4: 0.039587145064175405, 5: 0.03958714445526215, 1: 0.029861050394935206,
2: 0.02986104824582962, 3: 0.026824526173899837, 0: 0.02682452562582292}
```

Η συνάρτηση που έδωσε τα αποτελέσματα ταξινομημένα ήταν η εξής:

```
def getPagesAndTheirRankings(ranks):
    n = len(ranks)
    temp_ranks = np.copy(ranks)
    ranks_dict = {}

    for i in range(n):
        max_rank = temp_ranks[i]
        index = i

        for j in range(n):
            if i != j:
                if temp_ranks[j] >= max_rank:
                    max_rank = temp_ranks[j]
                    index = j

        ranks_dict[index] = max_rank
        temp_ranks[index] = 0

    return ranks_dict
```

4) Το πρώτο πράγμα που παρατηρούμε είναι πως όσο πιο μεγάλο το  $q$  (έβαλα  $(1 - q)/n$  αντί  $q/n$  για στον τύπο μου εγώ) τόσο πιο αργά συγκλίνει ο αλγόριθμος και όσο πιο μικρό τόσο πιο γρήγορα. Όταν έτρεξα τον αλγόριθμο είδα πως με  $q=0.85$  έκανε 30 επαναλήψεις, με  $q=0.98$  έκανε 182 επαναλήψεις και με  $q=0.4$  έκανε 8 επαναλήψεις. Οι τάξεις άλλαξαν σε όλες τις σελίδες εκτός από την πρώτη σελίδα (αυτήν επέλεξα να βελτιώσω στο 3 ερώτημα). Στο  $q=0.98$  οι σελίδες οι οποίες έχουν μεγαλύτερη τάξη έχουν αρκετά μεγάλη διαφορά σε σχέση με τις σελίδες που βρίσκονται σε χαμηλότερη τάξη. Ενώ στο  $q=0.4$  οι σελίδες που έχουν μεγαλύτερη τάξη δεν έχουν μεγάλη διαφορά σε σχέση με τις σελίδες που βρίσκονται σε χαμηλότερη τάξη.

Αυτά είναι τα αποτελέσματα με  $q=0.85$ :

{0: 0.1357669409655188, 9: 0.11098247863329593, 14: 0.09014716489066976, 8: 0.08651664566652875,  
1: 0.08581157425288524, 10: 0.08341930347723986, 12: 0.06640831358386824, 4: 0.05882634082341906,  
13: 0.054357393735387574, 5: 0.052623679995002476, 6: 0.046735082388783114, 11: 0.04384164408897119,  
2: 0.04261322010172763, 7: 0.022421803017132293, 3: 0.019529268711403036}

Αυτά είναι τα αποτελέσματα με  $q=0.98$ :

{0: 0.1542556626060627, 9: 0.12237100408188506, 8: 0.10028633818032819, 1: 0.09440455479274928,  
14: 0.08429745897097711, 10: 0.07420962398321851, 12: 0.07153382544394403, 4: 0.0649323785379344,  
13: 0.05223823537800837, 5: 0.05157949403025425, 6: 0.042750916885874284, 2: 0.03568558515228151,  
11: 0.03238394806997989, 7: 0.011912095653576184, 3: 0.007170263272775313}

Αυτά είναι τα αποτελέσματα με  $q=0.4$ :

{0: 0.09304872106511494, 9: 0.08895068697285932, 10: 0.08603608749201316, 14: 0.07912511159826188,  
1: 0.070473545024386, 8: 0.06694318665790881, 12: 0.06250081368209702, 5: 0.060789555632843446,  
11: 0.060472536158993506, 2: 0.05931899122039506, 13: 0.058883457518207755, 4: 0.05832223005868257,  
6: 0.05745947662987219, 3: 0.049612602325326785, 7: 0.04806300395995406}

5) Αυτή η στρατιγική δεν θα δουλέψει γιατί τα web crawlers της Google θα δουν αυτά τα links ως duplicates και πιθανότατα να τα αγνοήσουν. Αυτό δεν είναι καλή τεχνική για SEO κιόλας γιατί μπορεί να κάνει ζημιά στην κυκλοφορία και τάξη της ιστοσελίδας.

6) Όταν τρέξουμε τον κώδικα και κάνουμε την σύγκριση θα παρατηρήσουμε ότι οι σελίδες 8, 9 και 11 βελτιώθηκαν αρκετά σε σχέση με πριν που υπήρχε και η σελίδα 10.

Τα αποτελέσματα πριν την διαγραφή της σελίδας 10:

```
{14: 0.12509139798407176, 12: 0.1250913914925863, 13: 0.11632764815781536, 9: 0.10631974738474384,  
10: 0.1063197326476894, 11: 0.07456423039737362, 8: 0.07456422639149599, 6: 0.039587146617658775,  
7: 0.03958714600874552, 4: 0.039587145064175405, 5: 0.03958714445526215, 1: 0.029861050394935206,  
2: 0.02986104824582962, 3: 0.026824526173899837, 0: 0.02682452562582292}
```

Τα αποτελέσματα μετά την διαγραφή (όλα τα indices μετά την σελίδα 10 κατέβηκαν κατά 1 πχ η σελίδα 11 τώρα είναι στο index 9 αντί για το index 10 που ήταν πριν διαγραφεί η σελίδα 10):

```
{13: 0.18647990277286766, 9: 0.17096241811296542, 12: 0.1074619823384518, 10: 0.10359795641419242,  
6: 0.05165858156265789, 7: 0.05024877385466982, 8: 0.04822340189635783, 0: 0.04709493756809271,  
4: 0.04280078113837178, 5: 0.04139097343038371, 11: 0.041161829709641176, 1: 0.04091138144953402,  
2: 0.03593558953898786, 3: 0.03207000400550146}
```