Αριθμητική Ανάλυση - 1η Εργασία

Δημήτρης Βενιόπουλος

22 Δεκεμβρίου 2020

Άσκηση 1

Στην 1η άσκηση έχουν υλοποιηθεί οι εξής τρεις αλγόριθμοι για υπολογισμό ριζών μιας εξίσωσης:

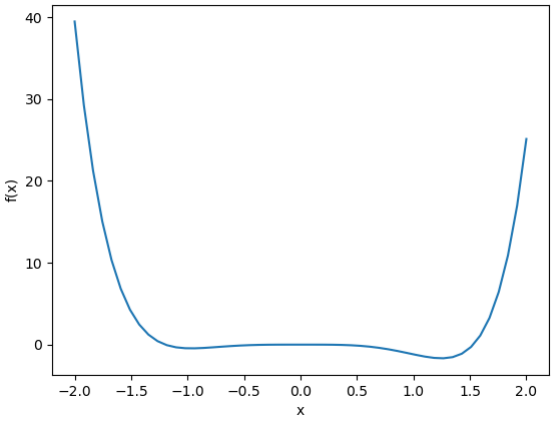
1. Μέθοδος διχοτόμησης
2. Μέθοδος Newton - Raphson
3. Μέθοδος τέμνουσας

Οι παραπάνω μέθοδοι χρησιμοποιήθηκαν για να βρούμε τις ρίζες της συνάρτησης

$$f(x) = e^{sin^3 x} + x^6 - 2x^4 - x^3 - 1$$

στο διάστημα [-2,2] με ακρίβεια 5 δεκαδικών ψηφίων. Όπως φαίνεται στο γράφημα της συνάρτησης που ακολουθεί, υπάρχουν συνολικά 3 ρίζες σε αυτό το διάστημα:

$$x_1 = -1.19762,\ x_2 = 0,\ x_3 = 1.53013$$



Μέθοδος διχοτόμησης

Στην μέθοδο της διχοτόμησης, η ρίζα φράσσεται μεταξύ ενός διαστήματος στο οποίο ισχύει το θέωρημα Bolzano. Το διάστημα αυτό διχοτομείται διαρκώς μέχρι να προσσεγγιστεί η ρίζα εντός του επιθυμητού σφάλματος ή εντός ορισμένου αριθμού επαναλήψεων. Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```python
# a: start of interval
# b: end of interval
# tol: tolerance of error
def bisection(a, b, tol):
    iterations = 0
    # Calculating the minimum number of iterations required
    # in order to approach root within tolerance of error
    iterations_needed = (np.log(b - a) - np.log(tol)) / np.log(2)
    while True:
        iterations += 1
        f_a = f(a)
        f_b = f(b)
        # Calculating current root approximation
        m = (a + b) / 2
        f_m = f(m)
        if f_m == 0 or iterations > iterations_needed:
            return m, iterations
        elif f_m * f_a < 0:
            b = m
        elif f_m * f_b < 0:
            a = m
```

Στην πραγματικότητα, γνωρίζουμε εκ των προτέρων πόσες επαναλήψεις θα χρειαστούν για να προσεγγίσουμε την λύση με το επιθυμητό σφάλμα με την βοήθεια του εξής τύπου: $N > \frac{ln(b-a)-lnc}{ln2}$, όπου N ο αριθμός των επαναλήψεων, a και b τα άκρα του διαστήματος και c το επιθυμητό σφάλμα.

Η συνάρτηση bisection καλείται τρεις φορές, μία για κάθε ρίζα με τα εξής αποτελέσματα:

- στο διάστημα [-2,-1] για να βρούμε την ρίζα x_1

```python
bisection_x1, iterations = bisection(-2, -1, tol)
print("Root x1 found with Bisection method: " + str(bisection_x1))
print("Iterations needed: " + str(iterations))
```

```
Root x1 found with Bisection method: -1.1976203918457031
Iterations needed: 18
```

- στο διάστημα [-1.5,1.5] για να βρούμε την ρίζα x_2

```python
bisection_x2, iterations = bisection(-1.5, 1.5, tol)
print("Root x2 found with Bisection method: " + str(bisection_x2))
print("Iterations needed: " + str(iterations))
```

```
Root x2 found with Bisection method: 0.0
Iterations needed: 1
```

- στο διάστημα [1.5,2] για να βρούμε την ρίζα x_3

```
bisection_x3, iterations = bisection(1.5, 2, tol)
print("Root x3 found with Bisection method: " + str(bisection_x3))
print("Iterations needed: " + str(iterations))
    Root x3 found with Bisection method: 1.5301322937011719
    Iterations needed: 17
```

Μέθοδος Newton - Raphson

Η μέθοδος Newton-Raphson, αποτελεί μια επαναληπτική μέθοδο σταθερού σημείου. Αν η συνάρτηση f της εισόδου είναι δυο φορές παραγωγίσιμη σε ένα διάστημα [a,b], με μη μηδενικές παραγώγους στο διάστημα αυτό, και ισχύει το θεώρημα Bolzano, τότε η f έχει μοναδική ρίζα το όριο της παρακάτω αναδρομικής σχέσης.

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, n = 1, ...$$

Παρατηρούμε ότι χρειαζόμαστε ένα αρχικό σημείο x_0, για το οποίο μάλιστα για να εγγυηθεί η σύγκλιση της μεθόδου πρέπει να ισχύει $f(x_0)f"(x_0) > 0$.

Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```
# a: start of interval
# b: end of interval
# tol: tolerance of error
# max_iterations: maximum number of iterations
def newton(a, b, x, tol, max_iterations):
    iterations = 0
    while True:
        iterations += 1
        # Storing previous value of x
        previous_x = x
        # Calculating current root approximation
        x = x - f(x) / df(x)
        # Iteration stops when exact solution is found
        # or when current error is within tolerance of error
        # or when the maximum number of iterations has been reached
        if f(x) == 0 or np.abs(x - previous_x) < tol or iterations >= max_iterations:
            return x, iterations
```

Η συνάρτηση newton καλείται τρεις φορές, μία για κάθε ρίζα με τα εξής αποτελέσματα:

- στο διάστημα [-2,-1] με αρχικό σημείο $x_0 = -2$ για να βρούμε την ρίζα x_1

```
newton_x1, iterations = newton(-2, -1, -2, tol, max_iterations)
print("Root x1 found with Newton - Raphson method: " + str(newton_x1))
print("Iterations needed: " + str(iterations))
   Root x1 found with Newton - Raphson method: -1.19762372213359
   Iterations needed: 8
```

- στο διάστημα [-1.5,1.5] με αρχικό σημείο $x_0 = 1$ για να βρούμε την ρίζα x_2

```
newton_x2, iterations = newton(-1.5, 1.5, 1, tol, max_iterations)
print("Root x2 found with Newton - Rapshon method: " + str(newton_x2))
print("Iterations needed: " + str(iterations))
   Root x2 found with Newton - Rapshon method: 4.54251675107278e-05
   Iterations needed: 33
```

- στο διάστημα [1.5,2] με αρχικό σημείο $x_0 = 2$ για να βρούμε την ρίζα x_3

```
newton_x3, iterations = newton(1.5, 2, 2, tol, max_iterations)
print("Root x3 found with Newton - Rapshon method: " + str(newton_x3))
print("Iterations needed: " + str(iterations))
    Root x3 found with Newton - Rapshon method: 1.5301335081666154
    Iterations needed: 7
```

Μέθοδος τέμνουσας

Η μέθοδος της τέμνουσας αποτελεί μια εναλλακτική της μεθόδου Newton - Raphson. Χρησιμοποιείται συνήθως όταν δεν είναι γνωστή η 1η παράγωγος της συνάρτησης που εξετάζουμε. Αν ισχύουν οι ίδιες προϋποθέσεις της Newton - Raphson σε ένα διάστημα [a,b],τότε θα υπάρχει μοναδική ρίζα της συνάρτησης στο διάστημα αυτό που δίνεται από το όριο της παρακάτω αναδρομικής σχέσης:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, n = 1, ...$$

Παρατηρούμε πως στην μέθοδο της τέμνουσας χρειάζονται δυο αρχικά σημεία x_0 και x_1.

Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```
# a: start of interval
# b: end of interval
# tol: tolerance of error
#max_iterations: maximum number of iterations
def secant(a, b, tol, max_iterations):
    iterations = 0
    x1 = a
    x2 = b
    while True:
        iterations += 1
        # Storing previous value of x
        previous_x = x2
        # Calculating current root approximation
        x = x2 - f(x2) * (x2 - x1) / (f(x2) - f(x1))
        # Iteration stops when exact solution is found
        # or when current error is within tolerance of error
        # or when the maximum number of iterations has been reached
        if f(x) == 0 or np.abs(x - previous_x) < tol or iterations >= max_iterations:
            return x, iterations
        x1 = x2
        x2 = x
```

Η συνάρτηση secant καλείται τρεις φορές, μία για κάθε ρίζα με τα εξής αποτελέσματα:

- στο διάστημα [-2,-1] για να βρούμε την ρίζα x_1

```
secant_x1, iterations = secant(-2, -1, tol, max_iterations)
print("Root x1 found with Secant method: " + str(secant_x1))
print("Iterations needed: " + str(iterations))
   Root x1 found with Secant method: -1.1976237224693556
   Iterations needed: 14
```

- στο διάστημα [-1.5,1] για να βρούμε την ρίζα x_2

```
secant_x2, iterations = secant(-1.5, 1, tol, max_iterations)
print("Root x2 found with Secant method: " + str(secant_x2))
print("Iterations needed: " + str(iterations))
  Root x2 found with Secant method: 6.593985448210844e-05
  Iterations needed: 44
```

- στο διάστημα [1.5,2] για να βρούμε την ρίζα x_3

```
secant_x3, iterations = secant(1.5, 2, tol, max_iterations)
print("Root x3 found with Secant method: " + str(secant_x3))
print("Iterations needed: " + str(iterations))
   Root x3 found with Secant method: 1.5301335082029042
   Iterations needed: 6
```

Σχολιασμός αποτελεσμάτων και σύγκριση μεθόδων

Και οι τρεις αλγόριθμοι κατάφεραν να προσεγγίσουν ικανοποιητικά τις ρίζες της συνάρτησης f. Η κύρια διαφορά τους είναι ως προς τον αριθμό των επαναλήψεων που πραγματοποιήθηκαν για να επιτευχθεί σύγκλιση. Γενικότερα, τα αποτελέσματα ήταν αναμενόμενα ως προς την ταχύτητα, με την μέθοδο Newton - Raphson να είναι συνήθως η γρηγορότερη, έπειτα να ακολουθεί η μέθοδος της τέμνουσας και τέλος η μέθοδος της διχοτόμησης. Παρουσιάστηκαν, ωστόσο, κάποιες ιδιαιτερότητες οι οποίες θα σχολιαστούν παρακάτω.

- προσέγγιση της ρίζας $x_1 = -1.19762$: Και στις τρεις μεθόδους η αναζήτηση της ρίζας έγινε στο διάστημα [-2,1] επιτυχώς, με ακρίβεια 5 δεκαδικών ψηφίων. Η μέθοδος Newton - Raphson χρειάστηκε 8 επαναλήψεις, η μέθοδος της τέμνουσας χρειάστηκε 14 επαναλήψεις και η μέθοδος της διχοτόμησης χρειάστηκε 18 επαναλήψεις.

- προσέγγιση της ρίζας $x_2 = 0$: Αυτή η περίπτωση παρουσιάζει κάποιες ιδιαιτερότητες. Ξεκινώντας από την μέθοδο της διχοτόμησης, το διάστημα που εξετάστηκε ήταν το [-1.5,1.5]. Παρατηρούμε ότι το διάστημα αυτό είναι πολύ βολικό για την συγκεκριμένη μέθοδο, αφού η μηδενική ρίζα που αναζητούμε είναι το μέσο του αρχικού διαστήματος, επομένως βρίσκει την ακριβή ρίζα με μόλις 1 επανάληψη. Αυτό δεν σημαίνει πως ο αλγόριθμος θα συγκλίνει τόσο γρήγορα αν τροποποιηθεί το αρχικό διάστημα. Η μέθοδος Newton - Raphson εκτελέστηκε επίσης στο διάστημα [-1.5,1.5]. Η προσέγγιση της ρίζα που βρήκε ήταν το $x^* = 0.00004$, ωστόσο ήταν αρκετά αργή αφού χρειάστηκε 33 επαναλήψεις. Αυτό συμβαίνει λόγω του μηδενισμού της πρώτης παραγώγου της f κοντά στο 0. Η τάξη σύγκλιση της Newton - Raphson θα δειχτεί αναλυτικά παρακάτω. Τέλος, η μέθοδος της τέμνουσας εκτελέστηκε στο διάστημα [-1.5,1] και η ρίζα που βρήκε ήταν το $x^* = 0.00006$ σε 44 επεναλαήψεις. Επιλέχθηκε διαφορετιό διάστημα καθώς στο [-1.5,1.5] η μέθοδος της τέμνουσας συγκλινει στην ρίζα x_3.

- προσέγγιση της ρίζας $x_3 = 1.53013$: Και στις τρεις μεθόδους η αναζήτηση της ρίζας έγινε στο διάστημα [1.5,2] επιτυχώς, με ακρίβεια 5 δεκαδικών ψηφίων. Αυτή τη φορά, η μέθοδος της τέμνουσας ήταν κατά 1 επανάληψη γρηγορότερη από την μέθοδο Newton - Raphson , καθώς η πρώτη χρειάστηκε 7 επαναλήψεις ενώ δεύτερη χρειάστηκε 8. Η μέθοδος της διχοτόμησης ήταν η πιο αργή, αφού χρειάστηκε 17 επαναλήψεις.

Τάξη σύγκλισης της μεθόδου Newton - Raphson

Όπως είδαμε και παραπάνω, η μέθοδος Newton - Raphson ήταν αρκετά γρήγορη στην προσέγγιση των ριζών $x_1 = -1.19762$ και $x_3 = 1.53013$, ενώ για την προσέγγιση της ρίζας $x_3 = 0$ η σύκγλιση ήταν αργή. Αυτό συμβαίνει γιατί η τάξη σύγκλισης της μεθόδου διαφέρει σε αυτά τα σημεία. Όταν η πρώτη παράγωγος της f είναι μη μηδενική σε μια ρίζα x, η μέθοδος Newton - Raphson συγκλίνει τετραγωνικά, άρα γρήγορα. Όταν όμως η παράγωγος μηδενίζεται κοντά στην ρίζα, τότε η σύγκλιση είναι γραμμική, άρα σαφώς πιο αργή. Εξετάζουμε λοιπόν την παράγωγο της f για κάθε μια από τις ρίζες.

- Για την $x_1 = -1.19762$ ισχύει ότι $f'(x_1) = -4.92 \neq 0$ άρα η σύκλιση είναι τετραγωνική.
- Για την $x_2 = 0$ ισχύει ότι $f'(x_2) = 0$ άρα η σύκλιση είναι γραμμική.
- Για την $x_3 = 1.53013$ ισχύει ότι $f'(x_3) = 14.97 \neq 0$ άρα η σύκλιση είναι τετραγωνική.

Άσκηση 2

Στην 2η άσκηση έχουν υλοποιηθεί ελαφρώς τροποποιημένες οι τρεις επαναληπτικές μέθοδοι της 1ης άσκησης, για να βρεθούν οι ρίζες της συνάρτησης

$$f(x) = 94cos^3x - 24cosx + 177sin^2x - 108sin^4x + 72cos^3xsin^2x - 65$$

στο διάστημα [0,3] με ακρίβεια 5 δεκαδικών ψηφίων. Σε αυτό το διάστημα η συνάρτηση παρουσιάζει τις εξής 3 ρίζες:

$$x_1 = 0.84106,\ x_2 = 1.04719,\ x_3 = 2.30052$$

Τροποοιημένη μέθοδος διχοτόμησης

Αυτή η τροποποιημένη μέθοδος της διχοτόμησης εκτελεί σχεδόν την ίδια λειτουργία με την μέθοδο διχοτόμισης της 1ης άσκησης με μία διαφορά. Για τον χωρισμό του διαστήματος δεν επιλέγεται το μέσο, αλλά ένα τυχαίο σημείο εντός του διαστήματος. Η μέθοδος τερματίζει είτε όταν προσεγγιστεί η ρίζα εντός του επιθυμητού σφάλματος, είτε όταν φτάσουμε στον μέγιστο αριθμό επαναλήψεων που είμαστε διατεθιμένοι να εκτελέσουμε. Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```python
# a: start of interval
# b: end of interval
# tol: tolerance of error
# max_iterations: maximum number of iterations
def bisection(a, b, tol, max_iterations):
    iterations = 0
    while True:
        iterations += 1
        f_a = f(a)
        f_b = f(b)
        # Selecting random point within the interval
        x = random.uniform(a, b)
        f_x = f(x)
        # Iteration stops when exact solution is found
        # or when current error is within tolerance of error
        # or when the maximum number of iterations has been reached
        if f_x == 0 or abs(a - b) < tol or iterations == max_iterations:
            return x, iterations
        elif f_x * f_a < 0:
            b = x
        elif f_x * f_b < 0:
            a = x
```

Εκτελώντας την μέθοδο της διχοτόμησης 10 φορές για κάθε ένα από τα διαστήματα [0,1], [1,2] και [2,3], ώστε να βρούμε προσεγγιστικά τις ρίζες x_1, x_2 και x_3 αντίστοιχα, αναμένουμε να δούμε διαφορετικά αποτελέσματα λόγω της τυχαιότητας που διέπει τον συγκεκριμένο αλγόριθμο.

- στο διάστημα [0,1] για να βρούμε την ρίζα x_1 παρατηρούμε ότι ο αριθμός επαναλήψεων διαφαίρει σε κάθε εκτέλεση της μεθόδου, καθώς κυμαίνεται από 14 έως και 44:

```
Root x1 found with Bisection method: 0.8410664842408127
Iterations needed: 24
Root x1 found with Bisection method: 0.8410684476570279
Iterations needed: 22
Root x1 found with Bisection method: 0.8410690376849466
Iterations needed: 20
Root x1 found with Bisection method: 0.8410659990953125
Iterations needed: 31
Root x1 found with Bisection method: 0.841069270361201
Iterations needed: 22
Root x1 found with Bisection method: 0.8410683073735331
Iterations needed: 14
Root x1 found with Bisection method: 0.8410672498697901
Iterations needed: 23
Root x1 found with Bisection method: 0.8410686015852933
Iterations needed: 44
Root x1 found with Bisection method: 0.8410682083991023
Iterations needed: 28
Root x1 found with Bisection method: 0.8410673900959724
Iterations needed: 18
```

- στο διάστημα [1,2] για να βρούμε την ρίζα x_2 επίσης υπάρχει κύμανση του αριθμού επαναλήψεων, καθώς έχουμε από 12 έως 30 επαναλήψεις:

```
Root x2 found with Bisection method: 1.0471835455188672
Iterations needed: 27
Root x2 found with Bisection method: 1.0471927764116133
Iterations needed: 22
Root x2 found with Bisection method: 1.047200324749796
Iterations needed: 27
Root x2 found with Bisection method: 1.0472022459697348
Iterations needed: 12
Root x2 found with Bisection method: 1.047201381842826
Iterations needed: 15
Root x2 found with Bisection method: 1.0472021650936396
Iterations needed: 23
Root x2 found with Bisection method: 1.0472020991552853
Iterations needed: 21
Root x2 found with Bisection method: 1.0472091354477715
Iterations needed: 23
Root x2 found with Bisection method: 1.0471944458429252
Iterations needed: 16
Root x2 found with Bisection method: 1.0472014128223999
Iterations needed: 30
```

- στο διάστημα [2,3] για να βρούμε την ρίζα x_3 παρατηρούμε ότι ο αριθμός των επαναλήψεων κυμαίνεται από 19 έως 37:

```
Root x3 found with Bisection method: 2.3005261191285262
Iterations needed: 20
Root x3 found with Bisection method: 2.300526054586601
Iterations needed: 35
Root x3 found with Bisection method: 2.3005228942388007
Iterations needed: 24
Root x3 found with Bisection method: 2.3005245310441014
Iterations needed: 35
Root x3 found with Bisection method: 2.3005210147785986
Iterations needed: 30
Root x3 found with Bisection method: 2.3005222548085773
Iterations needed: 37
Root x3 found with Bisection method: 2.300523317473091
Iterations needed: 26
Root x3 found with Bisection method: 2.300524596126276
Iterations needed: 27
Root x3 found with Bisection method: 2.3005274686712287
Iterations needed: 29
Root x3 found with Bisection method: 2.300523370180977
Iterations needed: 19
```

Εκτελώντας την κλασική μέθοδο διχοτόμισης στα ίδια διαστήματα έχουμε τα αποτελέσματα που φαίνονται στην παρακάτω εικόνα. Παρατηρούμε ότι, τουλάχιστον στα συγκεκριμένα διαστήματα και για την συγκεκριμένη συνάρτηση, η τροποποιημένη μέθοδος της διχοτόμησης φαίνεται πως δεν είχε καλύτερη απόδοση, καθώς στις περισσότερες εκτελέσεις ήταν πιο αργή από την κλασική μέθοδο διχοτόμησης.

```
Root x1 found with Bisection method: 0.8410682678222656
Iterations needed: 18
Root x2 found with Bisection method: 1.047210693359375
Iterations needed: 15
Root x3 found with Bisection method: 2.300525665283203
Iterations needed: 18
```

Τροποποιημένη μέθοδος Newton - Raphson

Η τροποποιημένη μέθοδος Newton - Raphson εκτελεί παρόμοια λειτουργία με την μέθοδο Newton - Raphson της 1ης άσκησης. Η διαφορά βρίσκεται στην αναδρομική σχέση που δίνει την προσεγγιστική ρίζα. Η τροποιοποιημένη αναδρομική σχέση είναι η εξής:

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

Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```python
# a: start of interval
# b: end of interval
# tol: tolerance of error
# max_iterations: maximum number of iterations
def newton(a, b, x, tol, max_iterations):
    iterations = 0
    while True:
        iterations += 1
        # Storing previous value of x
        previous_x = x
        # Calculating current root approximation
        x = x - f(x) / df(x) - (f(x) ** 2 * df2(x)) / (2 * df(x) ** 3)
        # Iteration stops when exact solution is found
        # or when current error is within tolerance of error
        # or when the maximum number of iterations has been reached
        if f(x) == 0 or np.abs(x - previous_x) < tol or iterations >= max_iterations:
            return x, iterations
```

Η τροποποιημένη μέθοδος Newton - Raphson εκτελέστηκε στα διαστήματα [0,1] (με $x_0 = 0.5$), [1,2] (με $x_0 = 1$) και [2,3] (με $x_0 = 2.5$) και προσέγγισε με επιτυχία τις 3 ζητούμενες ρίζες x_1, x_2 και x_3 αντίστοιχα. Τα αποτελέσματα φαίνονται στην παρακάτω εικόνα:

```
Root x1 found with Newton - Raphson method: 0.841068670567939
Iterations needed: 6
Root x2 found with Newton - Raphson method: 1.0471910380074236
Iterations needed: 15
Root x3 found with Newton - Raphson method: 2.3005239830218627
Iterations needed: 4
```

Στην παρακάτω εικόνα φαίνονται και τα αποτελέσματα της κλασικής μεθόδου Newton - Raphson . Παρατηρούμε πως η τροποποιημένη μέθοδος πετυχαίνει γρηγορότερη σύγκλιση στην προσέγγιση και των 3 ριζών της συνάρτησης f συγκριτικά με την κλασική μέθοδο.

```
Root x1 found with Newton - Raphson method: 0.8410686705678883
Iterations needed: 8
Root x2 found with Newton - Raphson method: 1.0471873116326482
Iterations needed: 29
Root x3 found with Newton - Raphson method: 2.300523983021863
Iterations needed: 5
```

Τροποποιημένη μέθοδος τέμνουσας

Η τροποποιημένη εκδοχή της μεθόδου της τέμνουσας χρησιμοποιεί 3 αρχικά σημεία x_n, x_{n+1} και x_{n+2} και η προσέγγιση της ρίζας είναι το όριο της παρακάτω αναδρομικής ακολουθίας:

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

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

Ο αλγόριθμος υλοποιείται από την παρακάτω συνάρτηση σε Python, η οποία επιστρέφει την προσεγγιστική ρίζα και τον αριθμό των επαναλήψεων που χρειάστηκαν.

```python
# a: start of interval
# b: end of interval
# tol: tolerance of error
# max_iterations: maximum number of iterations
def secant(a, b, tol, max_iterations):
    iterations = 0
    # Initiating starting points
    x1 = a
    x2 = (a + b) / 2
    x3 = b
    while True:
        iterations += 1
        # Storing previous value of x
        previous_x = x3
        q = f(x1) / f(x2)
        r = f(x3) / f(x2)
        s = f(x3) / f(x1)
        # Calculating current root approximation
        x = x3 - (r * (r - q) * (x3 - x2) + (1 - r) * s * (x3 - x1)) / ((q - 1) * (r - 1) * (s - 1))
        # Iteration stops when exact solution is found
        # or when current error is within tolerance of error
        # or when the maximum number of iterations has been reached
        if f(x) == 0 or np.abs(x - previous_x) < tol or iterations >= max_iterations:
            return x, iterations
        x1 = x2
        x2 = x3
        x3 = x
```

Η τροποποιημένη μέθοδος της τέμνουσας εκτελέστηκε στα διαστήματα [0.8,2], [1,2] και [2,3] και προσέγγισε με επιτυχία τις 3 ζητούμενες ρίζες x_1, x_2 και x_3 αντίστοιχα. Τα αποτελέσματα φαίνονται στην παρακάτω εικόνα:

```
Root x1 found with Secant method: 0.8410686705657576
Iterations needed: 8
Root x2 found with Secant method: 1.0471818278995637
Iterations needed: 26
Root x3 found with Secant method: 2.300523982944295
Iterations needed: 6
```

Εκτελόντας την κλασική μέθοδο τέμνουσας, παρατηρούμε πως η τροποποιημένη μέθοδος είναι ελάχιστα πιο γρήγορη από την κλασική. Τα αποτελέσματα της εκτέλεσης της κλασικής μεθόδου φαίνονται στην παρακάτω εικόνα.

```
Root x1 found with Secant method: 0.8410686699813509
Iterations needed: 8
Root x2 found with Secant method: 1.0471848516802726
Iterations needed: 30
Root x3 found with Secant method: 2.3005239825772943
Iterations needed: 8
```

Άσκηση 3.1

Στην άσκηση 3.1 της εργασίας έχει υλοποιηθεί η παραγοντοποίηση $PA = LU$ σε Python, για την επίλυση ενός γραμμικού συστήματος της μορφής $Ax = b$. Για τον αλγόριθμο αυτόν έχουν υλοποιηθεί οι εξής συναρτήσεις:

1. Συνάρτηση που καλείται pivot και δέχεται σαν όρισμα ένα πίνακα A. Η συνάρτηση επιστρέφει τον πίνακα P, ο οποίος αν πολλαπλασιάσει τον πίνακα A τον ταξινομεί ώστε στην διαγώνιο του να βρίσκονται τα στοιχεία με την μεγαλύτερη απόλυτη τιμή. Ο πίνακας P ουσιαστικά είναι μια μετάλλαξη του μοναδιαίου πίνακα, του οποίου οι γραμμές έχουν διαταχθεί με τέτοιο τρόπο ώστε να επιτυγχάνεται ο σκοπός που περιγράφτηκε πιο πάνω. Στην παρακάτω εικόνα φαίνεται η υλοποίηση του αλγορίθμου σε Python.

```python
# Function that creates P matrix
def pivot(array):
    n = len(array)
    A = array.copy()

    # Initializing P as a diagonal matrix with ones on the main diagonal
    P = np.zeros((n,n))
    for i in range(n):
        P[i][i] = 1

    # Sorting rows to get the final form of P matrix
    for j in range(n):
        max = abs(A[j][j])
        max_pos = j
        for i in range(j, n):
            if abs(A[i][j]) > max:
                max = abs(A[i][j])
                max_pos = i
        A[j], A[max_pos] = A[max_pos], A[j]
        P[[j, max_pos]] = P[[max_pos, j]]

    return P
```

2. Συνάρτηση που καλείται $lu\_factorization$ και δέχεται σαν όρισμα έναν πίνακα A. Η συνάρτηση εκτελεί τον αλγόριθμο της LU αποσύνθεσης και επιστρέφει τον πίνακα L (κάτω τριγωνικός) και τον πίνακα U (κάτω τριγωνικός). Τα στοιχεία u_{ij} του πίνακα U και τα στοιχεία l_{ij} του πίνακα L προκύπτουν από τις σχέσεις:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik}$$

$$l_{ij} = \frac{1}{u_{ij}} (a_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik})$$

Το γινόμενο τον δύο αυτών πινάκων είναι ο πίνακας A. Η παραγοντοποίηση του A σε δυο τριγωνικούς πίνακες βοηθάει στην μείωση του υπολογιστικού κόστους όταν θέλουμε να λύσουμε πολλά συστήματα με τον ίδιο πίνακα συντελεστών. Στην παρακάτω εικόνα φαίνεται η υλοποίηση του αλγορίθμου σε Python.

```python
# Function that performs the LU decomposition of matrix A
def lu_factorization(A):
    n = len(A)

    # Initializing L and U matrices
    L = np.zeros((n, n))
    U = np.zeros((n, n))

    # Applying u_ij and l_ij formulas
    for j in range(n):
        L[j][j] = 1

        for i in range(j+1):
            s = 0
            for k in range(i):
                s += U[k][j] * L[i][k]
            U[i][j] = A[i][j] - s

        for i in range(j, n):
            s = 0
            for k in range(j):
                s += U[k][j] * L[i][k]
            L[i][j] = (A[i][j] - s) / U[j][j]

    return L, U
```

3. Συνάρτηση που καλείται gauss και δέχεται σαν ορίσματα τον πίνακα A των συντελεστών και τον πίνακα b των σταθερών όρων. Η συνάρτηση αυτή επιλύει το σύστημα $Ax = b$ ως προς x και επιστρέφει το διάνυσμα x. Η επίλυση γίνεται με την παραγοντοποίηση $PA = LU$, συνδυάζοντας δηλαδή τις δύο προηγούμενες συναρτήσεις. Αφού έχουν δημιουργηθεί οι πίνακες P, L και U, λύνουμε πρώτα το σύστημα $Lc = Pb$ ως προς c και στη συνέχεια λύνουμε το σύστημα $Ux = c$ ως προς x. Τέλος επιστρέφουμε το διάνυσμα x, το οποίο είναι και η λύση του συστήματος $Ax = b$. Στην εικόνα που ακολουθεί φαίνεται η υλοποίηση του αλγορίθμου σε Python.

```python
# Function that solves a system of equations using PA=LU factorization
def gauss(A, b):
    # Creating P, L and U matrices
    P = pivot(A)
    L, U = lu_factorization(np.dot(P, A))
    Pb = np.dot(P, b)

    n = len(A)

    # Solving Lc = Pb for c
    c = np.zeros(n)
    c[0] = Pb[0]
    for i in range(1, n):
        c[i] = Pb[i]
        for j in range(0, i):
            c[i] -= L[i][j] * c[j]

    # Solving Ux = c for x
    x = np.zeros(n)
    x[n - 1] = c[n - 1] / U[n - 1][n - 1]
    for i in range (n - 2, -1, -1):
        x[i] = c[i]
        for j in range (n - 1, i, -1):
            x[i] -= U[i][j] * x[j]
        x[i] /= U[i][i]

    return x
```

Άσκηση 3.2

Στην άσκηση 3.2 έχει υλοποιηθεί ο αλγόριθμος για την αποσύνθεση Cholesky ενός συμμετρικού και θετικά ορισμένου πίνακα A. Η αποσύνθεση Cholesky είναι ουσιαστικά μια πιο αποδοτική παραγοντοποίηση LU. Όταν εφαρμόζεται σε έναν πίνακα A παράγει έναν κάτω τριγωνικό πίνακα L με θετικά στοιχεία διαγωνίου για τον οποίον ισχύει $A = LL^T$. Τα στοιχεία l_{ij} του πίνακα L προκύπτουν ως εξής:

$$L_{i,j} = \frac{1}{L_{j,j}}(A_{i,j} - \sum_{k=0}^{j-1} L_{i,k}L_{j,k}) \text{ για } i \neq j$$

$$L_{i,j} = \sqrt{A_{i,j} - \sum_{k=0}^{j-1} L_{i,k}^2} \text{ για } i = j$$

Ακολουθεί η συνάρτηση που υλοποιεί τον αποσύνθεση Cholesky σε Python.

```python
# Function that performs choesky decomposition
def cholesky(A):
    n = len(A)
    L = np.zeros((n,n))

    # Applying l_ij formula
    for j in range (n):
        for i in range (j, n):
            sum = 0
            for k in range(j):
                sum += L[i][k] * L[j][k]
            if i == j:
                L[i][j] = np.sqrt(A[i][j] - sum)
            else:
                L[i][j] = (A[i][j] - sum) / L[j][j]
    return L
```

Για παράδειγμα αν εκτελεστεί το παρακάτω κομμάτι κώδικα

```python
A = [[5.2, 3, 0.5, 1, 2],
     [3, 6.3, -2, 4, 0],
     [0.5, -2, 8, -3.1, 3],
     [1, 4, -3.1, 7.6, 2.6],
     [2, 0, 3, 2.6, 15]]

L = cholesky(A)
print(L)
```

εκτυπώνεται ο ακόλουθος πίνακας L:

```
[[ 2.28035085  0.          0.          0.          0.        ]
 [ 1.31558703  2.13757591  0.          0.          0.        ]
 [ 0.2192645  -1.07058726  2.60878631  0.          0.        ]
 [ 0.43852901  1.60138263 -0.5679783   2.12618594  0.        ]
 [ 0.87705802 -0.5397919   0.85472619  1.67683543  3.22444724]]
```

Άσκηση 3.3

Στην άσκηση 3.3 έχει υλοποιηθεί ο αλγόριθμος που εκτελεί την μέθοδο Gauss - Seidel για την λύση ενός γραμμικού συστήματος εξισώσεων της μορφής $Ax = b$ με ακρίβεια 4 δεκαδικών ψηφίων. Ο πίνακας συντελεστών Α είναι ένας αραιός τετραγωνικός πίνακας διάστασης n για τον οποίο ισχύει $A_{i,i} = 5, A_{i+1,i} = A_{i,i+1} = -2$. Ο πίνακας των σταθερών όρων b είναι της μορφής $b = [3, 1, 1, ..., 1, 1, 3]^T$.

Η μέθοδος Gauss - Seidel για την επίλυση γραμμικών συστημάτων εξισώσεων της μορφής $Ax = b$ βασίζεται στην εξής αναδρομική σχέση, δεδομένου ενός αρχικού διανύσματος x_0:

$$x_{k+1} = (L + D)^{-1}(-Ux_k + b)$$

όπου L το κάτω τριγωνικό μέρος του Α, D η κύρια διαγώνιος του Α και U το άνω τριγωνικό μέρος του Α, ώστε $A = L + D + U$.

Για την υλοποίηση του αλγορίθμου υλοποιήθηκαν οι παρακάτω συναρτήσεις σε Python :

1. Συνάρτηση που καλείται get_norm η οποία δέχεται ως όρισμα ένα διάνυσμα και επιστρέφει την άπειρη νόρμα του. Η συνάρτηση αυτή χρησιμοποιείται για τον υπολογισμό του σφάλματος της προσέγγισης της λύσης.

```python
# Function that calculates infinity norm of vector x
def get_norm(x):
    max = 0
    for i in range(len(x)):
        if abs(x[i]) > max:
            max = abs(x[i])
    return max
```

2. Συνάρτηση που καλείται create_arrays και δημιουργεί τους πίνακες Α και b ανάλογα με το μέγεθος που θέλουμε.

```python
# Functions that creates A and b matrices according to the given size
def create_arrays(n):
    A = np.zeros((n, n))
    for i in range(n):
        A[i][i] = 5
    for i in range(n - 1):
        A[i + 1][i] = -2
        A[i][i + 1] = -2

    b = np.ones(n)
    b[0] = 3
    b[n - 1] = 3

    return A, b
```

3. Συνάρτηση που καλείται split_array δέχεται ως όρισμα έναν πίνακα και τον χωρίζει στους πίνακες L, D και U με τον τρόπο που εξηγήθηκε προηγουμένως.

```python
# Function that splits matrix A into a diagonal, a lower triangular and an upper triangular matrix
# according to Gauss - Seidel method
def split_array(A):
    n = len(A)

    L = np.zeros((n, n))
    for i in range(n):
        for j in range(i):
            L[i][j] = A[i][j]

    D = np.zeros((n, n))
    for i in range(n):
        D[i][i] = A[i][i]

    U = np.zeros((n, n))
    for i in range (n):
        for j in range (i + 1, n):
            U[i][j] = A[i][j]

    return L, D, U
```

4. Συνάρτηση που καλείται gauss_seidel και υλοποιεί τον αλγόριθμο Gauss - Seidel , που δέχεται ως όρισμα τον πίνακα των συντελεστών A και τον πίνακα των σταθερών όρων b και επιλύει προσεγγιστικά το σύστημα.

```python
# Function that performs Gauss - Seidel method
# in order to solve the given equation system
def gauss_seidel(A, b, tol, max_iterations):
    # Creating L, D and U matrices
    L , D, U = split_array(A)
    # Initial solution is a zero matrix
    previous_x = np.zeros(len(A))
    iterations = 0
    while True:
        iterations += 1
        # Calculating approximate solution with Gauss - Seidel formula
        x = np.dot(np.linalg.inv(L + D), np.dot(-U, previous_x) + b)
        # Iteration ends when we find a solution within the tolerance of error
        # or when we have reached the maximum number of iterations
        if abs(get_norm(x) - get_norm(previous_x)) < tol or iterations == max_iterations:
            return x
        previous_x = x
```

Για παράδειγμα αν εκτελεστεί ο αλγόριθμος για $n = 10$

```python
A, b = create_arrays(10)
x = gauss_seidel(A, b, 0.00005, 25)
print("Solution found: ")
print(np.array([x]).T)
```

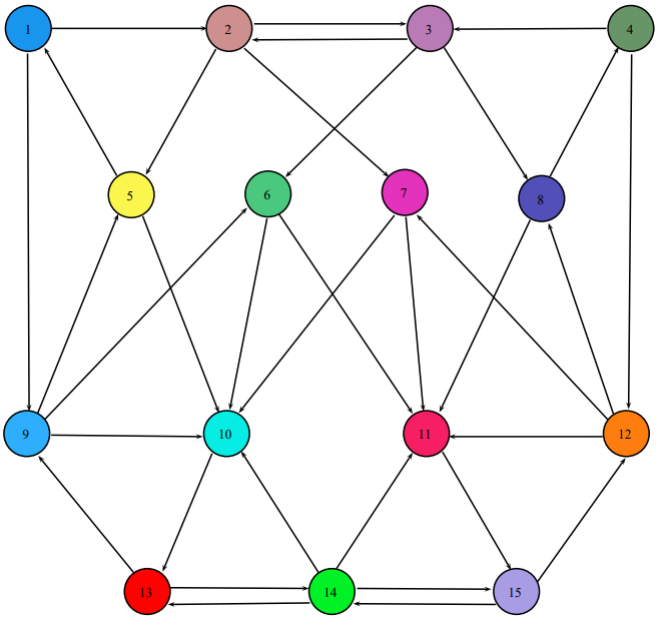
εκτυπώνεται η ακόλουθη λύση x:

```
Solution found:
[[0.99943503]
 [0.99915176]
 [0.99907157]
 [0.99912607]
 [0.99925817]
 [0.99942322]
 [0.99958935]
 [0.9997366 ]
 [0.99985498]
 [0.99994199]]
```

Για $n = 10000$ οι συνιστώσες εξακολουθούν να συγκλίνουν προς το 1, αλλά ο χρόνος εκτέλεσης πολλαπλασιάζεται.

Άσκηση 4

Στην 4η άσκηση ζητείται η μελέτη της αλγορίθμου Page Rank της Google για ταξινόμηση των σελίδων στο διαδίκτυο. Θεωρούμε το παρακάτω γράφημα, στο οποίο απεικονίζονται n ιστοσελίδες (κόμβοι), μεταξύ των οποίων υπάρχουν ακμές (*links*), που συνδέουν τις ιστοσελίδες μεταξύ τους.



Το γράφημα αυτό αποτυπώνεται και στον πίνακα γειτνίασης A, όπου $A_{ij} = 1$ αν υπάρχει σύνδεσμος από την ιστοσελίδα i στην ιστοσελίδα j και $A_{ij} = 0$ αν δεν υπάρχει σύνδεσμος μεταξύ τους.

$$\mathbf{A} = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}$$

Ακόμη, έχουμε τον πίνακα G, ο οποίος καλείται και πίνακας $Google$. Τα κελιά του πίνακα G υπολογίζονται από τον εξής τύπο:

$$G_{ij} = \frac{q}{n} + \frac{A_{ij}(1-q)}{n_j}$$

όπου q η πιθανότητα μεταπήδησης του χρήστη από μια σελίδα σε μια άλλη και n_j το άθροισμα της στήλης n του πίνακα A. Τέλος, έχουμε το κανονικοποιημένο ιδιοδιάνυσμα p που αντιστοιχεί στην (μέγιστη) ιδιοτιμή 1 του πίνακα G, το οποίο είναι το διάνυσμα με τις n πιθανότητες του να βρίσκεται ο χρήστης στις n ιστοσελίδες, δηλαδή η τάξη της σελίδας.

Άσκηση 4.1

Στην άσκηση 4.1 παρουσιάζεται η απόδειξη πως ο πίνακας G είναι στοχαστικός. Για να είναι ένας πίνακας στοχαστικός πρέπει (a) όλα τα στοιχεία του να είναι μη μηδενικά και (b) το άθροισμα της κάθε στήλης να είναι ίσο με 1.
(a)Θα αποδειχτεί πρώτα πως $G_{ij} \geq 0 \implies \frac{q}{n} + \frac{A_{ij}(1-q)}{n_j} \geq 0$. Ισχύουν τα εξής:

$$0 \leq q \leq 1 \implies 1 - q \geq 0 \text{ (ως στοιχείο πιθανότητας)}$$

$$n > 0 \text{ (ως η διάσταση του πίνακα)}$$

$$A_{ij} \geq 0 \text{ (ως στοιχείο του πίνακα γειτνίασης)}$$

$$n_j = \sum_{i=1}^{n} A_{ji} \implies n_j \geq 0 \text{ (ως άθροισμα μη αρνητικών στοιχείων)}$$

Συνεπώς, προκύπτει ότι $G_{ij} \geq 0$ αφού όλοι οι όροι του είναι μη αρνητικοί.
(b) Θα αποδειχτεί πως το άθροισμα της κάθε στήλης να είναι ίσο με 1:

$$\sum_{i=1}^{n} G_{ij} = \sum_{i=1}^{n} \frac{q}{n} + \frac{A_{ij}(1-q)}{n_j} = \sum_{i=1}^{n} \frac{q}{n} + \sum_{i=1}^{n} \frac{A_{ij}(1-q)}{n_j}$$

$$= n\frac{q}{n} + (1-q)\sum_{i=1}^{n} \frac{A_{ij}}{n_j} = q + (1-q)\frac{\sum_{i=1}^{n} A_{ji}}{\sum_{i=1}^{n} A_{ji}} = q + 1 - q = 1$$

Επομένως έχει αποδειχτεί ότι ο πίνακας G είναι στοχαστικός.

Άσκηση 4.2

Στην άσκηση 4.2 έχει υλοποιηθεί σε Python ο αλγόριθμος που κατασκευάζει τον πίνακα G, με βάση τον πίνακα γειτνίασης A και $q = 0.15$. Η υλοποίηση έγινε με τις εξής συναρτήσεις:

1. Συνάρτηση που καλείται get_norm η οποία υπολογίζει και επιστρέφει την l_1 νόρμα ενός διανύσματος.

```python
# Function that calculates l1 norm of a vector
def get_norm(x):
    sum = 0
    for i in range(len(x)):
        sum += abs(x[i])
    return sum
```

2. Συνάρτηση που καλείται create_g η οποία δημιουργεί και επιστρέφει τον πίνακα G για κάποια τιμή της πιθανότητας μεταπήδησης q.

```python
# Function that creates the Google matrix
def create_g(A, q):
    n = len(A)
    G = np.zeros((n, n))

    # Applying g_ij formula
    for i in range(n):
        for j in range(n):
            G[i][j] = q / n + (A[j][i] * (1 - q)) / sum(A[j])

    return G
```

3. Συνάρτηση που καλείται power_method η οποία εκτελεί την μέθοδο της δυνάμεως, ώστε να υπολογιστεί το κανονικοποιημένο ιδιοδιάνυσμα του πίνακα.

```python
# Function that performs the power method
# in order to find the eigenvector of a matrix
# A: the matrix for which we calculate the eigenvector
# x: the starting eigenvector
# tol: tolerance of error
# max_iterations: maximum number of iterations
def power_method(A, x, tol, max_iterations):
    iterations = 0
    previous_l = 1
    while True:
        iterations += 1
        # Applying power method steps
        x = np.dot(A, x)
        l = max(abs(x))
        x = x / l
        # Iteration ends when the value of the eigenvalue is within the tolerance of error
        # or when the maximum number of iterations has been reached
        if abs(l - previous_l) < tol or iterations == max_iterations:
            # Normalizing eigenvector
            x = x / get_norm(x)
            return x
```

Εκτελούμε τον κώδικα για $q = 0.15$ και παίρνουμε το παρακάτω ιδιοδιάνυσμα p

```
Eigenvector p found.
[[0.02682457]
 [0.02986108]
 [0.02986108]
 [0.02682457]
 [0.03958722]
 [0.03958722]
 [0.03958722]
 [0.03958722]
 [0.07456439]
 [0.10631996]
 [0.10631996]
 [0.07456439]
 [0.12509163]
 [0.11632789]
 [0.12509163]]
```

Άσκηση 4.3

Στην άσκηση 4.3 ζητείται να αυξήσουμε τον βαθμό σημαντικότητας μιας σελίδας της επιλογής μας, προσθέτοντας 4 ακμές στο γράφημα και αφαιρώντας 1. Η σελίδα που επιλέχθηκε είναι η 11, της οποίας την τάξη συμβολίζουμε p_{11}. Η αρχική τάξη της σελίδας 11 ήταν $p_{11} = 0.10631996$. Οι 4 ακμές που προστέθηκαν ήταν από σελίδες με μεγάλη σημαντικότητα. Πιο συγκεκριμένα προστέθηκαν οι ακμές (9,11), (10,11), (13,11) και (15,11). Η ακμή που αφαιρέθηκε ήταν η (5,1). Μετά τις αλλαγές, ο βαθμός της σελίδας 11 αυξήθηκε σημαντικά, καθιστώντας την ως τη σελίδα με την μεγαλύτερη σημαντικότητα με $p_{11} = 0.21821449$. Το ανανεωμένο ιδιοδιάνυσμα p φαίνεται παρακάτω.

```
Eigenvector p found.
[[0.01      ]
 [0.02218135]
 [0.027993  ]
 [0.02754891]
 [0.02317306]
 [0.02481969]
 [0.03964492]
 [0.04129156]
 [0.03241573]
 [0.08287526]
 [0.21821449]
 [0.08244777]
 [0.06411435]
 [0.08890522]
 [0.21437468]]
```

Άσκηση 4.4

Στην άσκηση 4.4 δοκιμάζουμε διαφορετικές τιμές για την πιθανότητα μεταπήδησης για να βρούμε τις μεταβολές που προκαλεί το q στην τάξη των σελίδων. Στην παρακάτω εικόνα φαίνονται οι τιμές του ιδιοδιανύσματος p για $q = 0.02$, για $q = 0.15$ και για $q = 0.6$. Όσο μειώνεται η πιθανότητα μεταπήδησης q τόσο αυξάνεται το χάσμα μεταξύ της τάξης των σελίδων. Αυτό συμβαίνει γιατί η πιθανότητα $1-q$, που εκφράζει την πιθανότητα να μεταβεί ο χρήστης σε μια νέα σελίδα επιλέγοντας έναν τυχαίο σύνδεσμο της σελίδας που βρίσκεται τώρα, αυξάνεται. Επομένως εποφελούνται οι σελίδες με τα περισσότερα links προς αυτές. Αντίθετα, αν αυξηθεί το q, τότε επωφελούνται οι σελίδες στις οποίες οδηγούν λίγα links.

```
   q = 0.02              q = 0.15               q = 0.6
Eigenvector p found.  Eigenvector p found.  Eigenvector p found.
[[0.01710634]         [[0.02682457]         [[0.05133212]
 [0.01442887]          [0.02986108]          [0.05799972]
 [0.01442887]          [0.02986108]          [0.05799972]
 [0.01710634]          [0.02682457]          [0.05133212]
 [0.0321898 ]          [0.03958722]          [0.05666061]
 [0.0321898 ]          [0.03958722]          [0.05666061]
 [0.0321898 ]          [0.03958722]          [0.05666061]
 [0.0321898 ]          [0.03958722]          [0.05666061]
 [0.08002971]          [0.07456439]          [0.06695487]
 [0.10957603]          [0.10631996]          [0.09026137]
 [0.10957603]          [0.10631996]          [0.09026137]
 [0.08002971]          [0.07456439]          [0.06695487]
 [0.14349852]          [0.12509163]          [0.08344224]
 [0.14196188]          [0.11632789]          [0.0733769 ]
 [0.14349852]]         [0.12509163]]         [0.08344224]]
```

Άσκηση 4.5

Στην άσκηση 4.5 θέτουμε τα κελιά $A_{8,11}$ και $A_{12,11}$ ίσα με 3. Αυτή η στρατηγική βοηθά την σελίδα 11 να αυξήσει της τάξη της έναντι της σελίδας 10. Αυτό συμβαίνει γιατί κατά τον υπολογισμό της πιθανότητας ο συντελεστής $\frac{A_{ij}}{n_i}$ (όπου A_{ij} το αντίστοιχο κελί και n_i το άθροισμα της στήλης i του πίνακα A) αυξάνεται και στις δυο περιπτώσεις. Επομένως, αυξάνεται και η πιθανότητα ο χρήστης σε οποιαδήποτε χρονική στιγμή να είναι στην σελίδα 11. Από την άλλη, η τάξη της σελίδας του ανταγωνιστή (σελίδα 10) παραμένει σχεδόν αναλλοίωτη εφόσον οι σελίδες 8 και 12 δεν δείχνουν σε αυτήν.

Άσκηση 4.6

Στην άσκηση 4.6 διαγράφουμε την σελίδα 10 από τον γράφο άρα διαγράφονται η 10η γραμμή και η 10η στήλη από τον πίνακα γειτνίασης. Αυτό αυξομειώνει την τάξη όλων των σελίδων. Ωστόσο, για κάποιες σελίδες η μεταβολή που προκλήθηκε στην τάξη τους ήταν αρκετά μεγάλη, κι αυτό επειδή η σελίδα 10 ήταν από τις πιο σημαντικές σελίδες. Η σελίδα 10 έδειχνε μόνο προς την σελίδα 13, η οποία πλέον έχει υποστεί μεγάλη μείωση στην τάξη της (από 0.125 μειώθηκε σε 0.041). Ακόμη, η διαγραφή της σελίδας 10, πλέον έχει αυξήσει σημαντικά την τάξη της σελίδας 11 (από 0.106 σε 0.17), όντας πλέον η σελίδα με τους περισσότερους συνδέσμους προς το μέρος της. Αυτό είχε ως συνέπεια να αυξηθεί σημαντικά και η τάξη της σελίδας 15 (από 0.125 σε 0.186) αφού αποτελεί την μοναδική σελίδα προς την οποία έχει σύνδεσμο η σελίδα 11. Αυτές οι μεγάλες μεταβολές έχουν επηρεάσει αναμενόμενα και τις υπόλοιπες σελίδες, όχι όμως σε τόσο μεγάλο βαθμό. Όλες οι μεταβολές παρουσιάζονται στον παρακάτω πίνακα.

Σελίδα	Πριν	Μετά
1	0.026	0.047
2	0.029	0.040
3	0.029	0.035
4	0.026	0.032
5	0.039	0.042
6	0.039	0.041
7	0.039	0.051
8	0.039	0.050
9	0.074	0.048
10	0.106	del
11	0.106	0.170
12	0.074	0.103
13	0.125	0.041
14	0.116	0.107
15	0.125	0.186

Τέλος Εργασίας

Αριθμητική Ανάλυση - 2η Εργασία

Δημήτριος Βενιόπουλος - ΑΕΜ 3610

Ιανουάριος 2021

Άσκηση 5

Στην 5η άσκηση έχουν υλοποιηθεί οι εξής τρεις μέθοδοι για την προσέγγιση μιας συνάρτησης:

1. Πολυώνυμο Lagrange
2. Παρεμβολή Splines
3. Μέθοδος ελαχίστων τετραγώνων

Οι παραπάνω μέθοδοι χρησιμοποιήθηκαν για να βρούμε μια προσέγγιση της συνάρτησης $f(x) = sin(x)$ στο διάστημα [-π,π] (με στρογγυλοποίηση στο 5ο δεκαδικό ψηφίο). Για την υλοποίηση των μεθόδων έχουν επιλεχθεί 10 αρχικά σημεία x_i, ανομοιόμορφα κατανεμημένα, τα οποία φαίνονται παρακάτω:

i	x_i	y_i
0	-π	0
1	-2.73	-0.40007
2	-2.15	-0.83690
3	-1.24	-0.94578
4	-0.12	-0.11971
5	0	0
6	0.73	0.66687
7	1.87	0.95557
8	3	0.14112
9	π	0

Πολυώνυμο Lagrange

Έστω $n+1$ σημεία $(x_i, y_i), i = 0, 1, 2, ..., n$, τότε το πολώνυμο Lagrange που διέρχεται από αυτά τα σημεία δίνεται από τον τύπο

$$p_n(x) = \sum_{i=0}^{n} y_i L_i(x)$$

όπου

$$L_i(x) = \frac{(x-x_0)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)}, i = 0, ...n$$

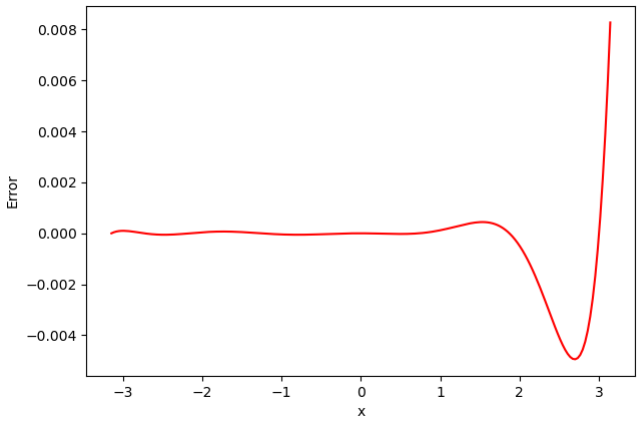
Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου σε Python. Η συνάρτηση lagrange δέχεται ως ορίσματα τις αρχικές τιμές των x_i και y_i και ένα σημείο x για το οποίο θα υπολογιστεί η τιμή της ζητούμενης συνάρτησης.

```
def lagrange(x_values, y_values, x):
    n = len(x_values) - 1
    output = 0
    for i in range(n):
        p = 1
        for j in range(n):
            if (i != j):
                p *= (x - x_values[j]) / (x_values[i] - x_values[j])
        output += y_values[i] * p
    return output
```

Το πολυώνυμο Lagrange κατάφερε να προσεγγίσει σε πολύ καλό βαθμό την συνάρτηση του ημιτόνου. Παρακάτω φαίνονται συγκριτικά τα αποτελέσματα για τις 10 αρχικές τιμές x_i.

i	x_i	y_i	Έξοδος Lagrange
0	-π	0	0
1	-2.73	-0.40007	-0.40007
2	-2.15	-0.83690	-0.83690
3	-1.24	-0.94578	-0.94578
4	-0.12	-0.11971	-0.11971
5	0	0	0
6	0.73	0.66687	0.66687
7	1.87	0.95557	0.95557
8	3	0.14112	0.14112
9	π	0	-0.00827

Παρατηρούμε ότι το σφάλμα (ως προς την στρογγυλοποίηση στο 5ο ψηφίο) είναι μηδενικό σε όλες τις τιμές εκτός από την $x = \pi$, όπου παρατηρείται μια μικρή απόκλιση της τάξης του 0.008. Αυτό συμβαίνει επειδή τα αρχικά σημεία x που έχουν επιλεχθεί είναι πιο αραιά σε εκείνη την περιοχή. Παρακάτω φαίνεται το γράφημα του σφάλματος για 200 σημεία του διαστήματος [-π,π], με $RMSE = 0.00156$. Ομοίως, το σφάλμα είναι σχεδόν μηδενικό στις περιοχές όπου τα σημεία x είναι πυκνά, ενώ στην αραιή περιοχή, μεταξύ του 1.9 και π, το σφάλμα μπορεί να πάρει τιμές έως και 0.008.



Παρεμβολή Splines

Έστω τα σημεία $x_0 < x_1 < ... < x_n$, τα οποία διαμερίζουν το διάστημα $[x_0, x_n]$. Κάθε συνάρτηση $s \in C^2[x_0, x_n]$ τέτοια ώστε $s_{[x_i, x_{i+1}]}$ να είναι πολυώνυμο βαθμού 3, τότε η s καλείται πολυωνιμική Spline 3ου βαθμού ή κυβική Spline.

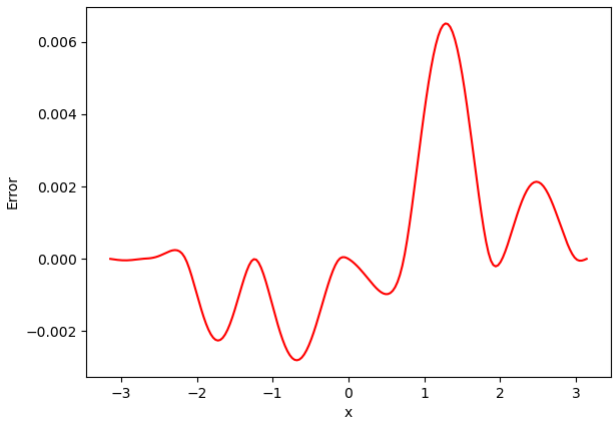
Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου σε Python. Η συνάρτηση splines δέχεται ως ορίσματα τις αρχικές τιμές των x_i και y_i και ένα σημείο x για το οποίο θα υπολογιστεί η τιμή της ζητούμενης συνάρτησης.

```
def splines(x_values, y_values, x):
    n = len(x_values)

    dx = np.zeros(n - 1)
    dy = np.zeros(n - 1)
    for i in range(n - 1):
        dx[i] = x_values[i + 1] - x_values[i]
        dy[i] = y_values[i + 1] - y_values[i]

    A = np.zeros((n, n))
    B = np.zeros(n)
    A[0, 0] = 1
    A[n - 1, n - 1] = 1
    for i in range(1, n - 1):
        A[i, i - 1] = dx[i - 1]
        A[i, i + 1] = dx[i]
        A[i, i] = 2 * (dx[i - 1] + dx[i])
        B[i] = 3 * (dy[i] / dx[i] - dy[i - 1] / dx[i - 1])

    a = y_values.copy()
    c = np.linalg.solve(A, B)
    d = np.zeros(n)
    b = np.zeros(n)
    for i in range(n - 1):
        d[i] = (c[i + 1] - c[i]) / (3 * dx[i])
        b[i] = (dy[i] / dx[i]) - (dx[i] / 3) * (2 * c[i] + c[i + 1])

    for i in range (n - 1):
        if (x_values[i] <= x <= x_values[i + 1]):
            return a[i] + b[i] * (x - x_values[i]) + c[i] * (x - x_values[i]) ** 2 + d[i] * (x - x_values[i]) ** 3
```

Η κυβική Splines κατάφερε να προσεγγίσει σε πολύ καλό βαθμό την συνάρτηση του ημιτόνου. Παρακάτω φαίνονται συγκριτικά τα αποτελέσματα για τις 10 αρχικές τιμές x_i.

i	x_i	y_i	Έξοδος Splines
0	-π	0	0
1	-2.73	-0.40007	-0.40007
2	-2.15	-0.83690	-0.83689
3	-1.24	-0.94578	-0.94578
4	-0.12	-0.11971	-0.11970
5	0	0	0
6	0.73	0.66687	0.66687
7	1.87	0.95557	0.95557
8	3	0.14112	0.14112
9	π	0	0

Παρατηρούμε ότι το σφάλμα (ως προς την στρογγυλοποίηση στο 5ο ψηφίο) είναι σχεδόν μηδενικό σε όλες τις τιμές των αρχικών x. Ακολουθεί το γράφημα του σφάλματος για 200 σημεία του διαστήματος [-π,π], με $RMSE = 0.00215$.



Μέθοδος Ελαχίστων Τετραγώνων

Η μέθοδος των ελαχίστων τετραγώνων ουιαστικά σκοπεύει στην ελαχιστοποίηση του αθροίσματος των τετραγωνικών σφαλμάτων ενός μοντέλου που εμείς ορίζουμε. Ο υπολογισμός του ημιτόνου έγινε χρησιμοποιώντας ως μοντέλο ένα πολυώνυμο βαθμού $m = 4$ της μορφής $y = c_1 + c_2x + c_3x^2 + c_4x^3 + c_5x^4$. Ο προσδιορισμός των παραμέτρων c_i προκύπτει από την επίλυση του συστήματος $A^TAc = A^Tb$ ως προς c, όπου A ένας πίνακας στον οποίον τοποθετούνται οι εξισώσεις που προκύπτουν από τα δεδομένα και b ο πίνακας των τιμών του y.

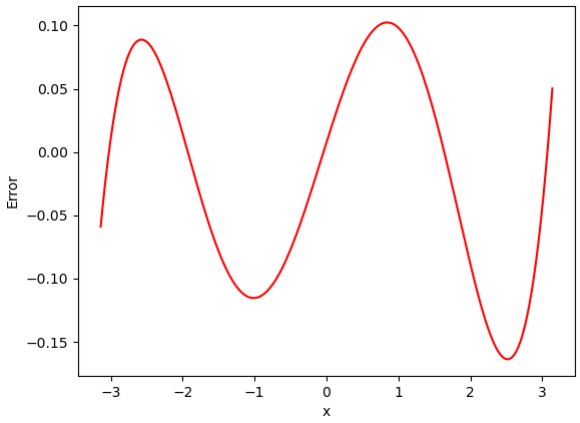
Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου σε Python. Η συνάρτηση least_squares δέχεται ως ορίσματα τις αρχικές τιμές των x_i και y_i, ένα σημείο x για το οποίο θα υπολογιστεί η τιμή της ζητούμενης συνάρτησης και τον βαθμό του πολυωνύμου μέσω του οποίου θα γίνει ο υπολογισμός.

```
def least_squares(x_values, y_values, x, degree):
    n = len(x_values)
    k = degree + 1
    b = y_values.copy()
    A = np.zeros((n,k))

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

    c = np.linalg.solve(np.dot(A.transpose(), A), np.dot(A.transpose(), b))
    result = 0
    for i in range(k):
        result += c[i] * x ** i

    return result
```

Η μέθοδος ελαχίστων τετραγώνων προσέγγισε σε ικανοποιητικό βαθμό την συνάρτηση του ημιτόνου, όχι όμως όσο καλά όσο οι προηγούμενες μέθοδοι. Παρακάτω φαίνονται συγκριτικά τα αποτελέσματα για τις 10 αρχικές τιμές x_i.

i	x_i	y_i	Έξοδος Ελαχίστων Τετραγώνων
0	-π	0	0.05892
1	-2.73	-0.40007	-0.48004
2	-2.15	-0.83690	-0.88012
3	-1.24	-0.94578	-0.83858
4	-0.12	-0.11971	-0.10563
5	0	0	-0.00769
6	0.73	0.66687	0.56661
7	1.87	0.95557	1.01230
8	3	0.14112	0.18560
9	π	0	-0.05026

Παρατηρούμε ότι τα σφάλματα (ως προς την στρογγυλοποίηση στο 5ο ψηφίο) γίνονται αισθητά, διατηρώντας συνήθως 1 μόλις ψηφίο ακρίβειας. Ακολουθεί το γράφημα του σφάλματος για 200 σημεία του διαστήματος [-π,π], με $RMSE = 0.08481$.



Συγκρίνωντας τις τρεις μεθόδους, καταλήγουμε στα αναμενόμενα συμπεράσματα. Η πιο καλή προσέγγιση έχει γίνει από το πολυώνυμο Lagrange , έπειτα από την κυβική Splines και τέλος από την μέθοδο των ελαχίστων τετραγώνων.

Άσκηση 6

Στην 6η άσκηση έχουν υλοποιηθεί οι κανόνες τραπεζίου και Simpson για τον υπολογισμό του ολοκληρώματος $\int_0^{\pi/2} sin(x)\,dx$.

Κανόνας Τραπεζίου

Έστω τα σημεία $a = x_0 < x_1 < ... < x_n = b$, τα οποία διαμερίζουν το διάστημα $[a, b]$ σε n ισομήκη υποδιαστήματα. Με τον κανόνα τραπεζίου μπορούμε να υπολογίσουμε το ολοκλήρωμα $I = \int_a^b f(x)\,dx$ ως εξής:

$$I = \frac{b-a}{2n}(f(x_0) + f(x_n) + 2\sum_{i=1}^{n-1} f(x_i))$$

Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου σε Python. Η συνάρτηση trapezoidal_rule δέχεται ως ορίσματα τα άκρα του διαστήματος a και b όπου θα υπολογιστεί το ολοκλήρωμα και τον επιθυμητό αριθμό των διαμερίσεων του διαστήματος.

```
def trapezoidal_rule(a, b, n):
    x = np.linspace(a, b, n + 1)
    sum = 0
    for i in range (1, n):
        sum += f(x[i])

    integral = (b - a) / (2 * n) * (f(x[0]) + f(x[n]) + 2 * sum)
    return integral
```

Ακολουθούν τα αποτελέσματα του αλγορίθμου για τον υπολογισμό του ολοκληρώματος $\int_0^{\pi/2} sin(x)\,dx$ με 10 διαμερίσεις.

$\int_0^{\pi/2} sin(x)\,dx$	1
Έξοδος κανόνα τραπεζίου	0.9979429863543572
Αριθμητικό Σφάλμα	$\|e\| \approx 0.00206$
Θεωρητικό σφάλμα	$\|e\| \leq 0.00323$
Αριθμός διαμερίσεων	10

Το θεωρητικό σφάλμα e προκύπτει από τον τύπο $e \leq \frac{(b-a)^3}{12n^2}M$ όπου M το μέγιστο της δεύτερης παραγώγου της συνάρτησης στο διάστημα $[a, b]$, το οποίο στην περίπτωση μας είναι 1 αφού η συνάρτηση μας είναι το ημίτονο. Κάνοντας αντικατάσταση όπου $a = 0, b = \frac{\pi}{2}$ και $n = 10$, προκύπτει τελικά $|e| \leq 0.00323$. Πράγματι το αριθμητικό μας σφάλμα ικανοποιεί αυτή την συνθήκη.

Κανόνας Simpson

Η λογική του κανόνα Simpson είναι παρόμοια με αυτή του κανόνα τραπεζίου. Με τον κανόνα Simpson μπορούμε να υπολογίσουμε το ολοκλήρωμα $I = \int_a^b f(x)\,dx$ ως εξής:

$$I = \frac{b-a}{3n}(f(x_0) + f(x_n) + 2\sum_{i=1}^{\frac{n}{2}-1} f(x_{2i}) + 4\sum_{i=1}^{\frac{n}{2}} f(x_{2i-1}))$$

Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου σε Python. Η συνάρτηση simpson_rule δέχεται ως ορίσματα τα άκρα του διαστήματος a και b όπου θα υπολογιστεί το ολοκλήρωμα και τον επιθυμητό αριθμό των διαμερίσεων του διαστήματος.

```python
def simpson_rule(a, b, n):
    x = np.linspace(a, b, n + 1)

    sum1 = 0
    for i in range(2, n, 2):
        sum1 += f(x[i])

    sum2 = 0
    for i in range(1, n, 2):
        sum2 += f(x[i])

    return (b - a) / (3 * n) * (f(x[0]) + f(x[n]) + 2 * sum1 + 4 * sum2)
```

Ακολουθούν τα αποτελέσματα του αλγορίθμου για τον υπολογισμό του ολοκληρώματος $\int_0^{\pi/2} sin(x)\,dx$ με 10 διαμερίσεις.

$\int_0^{\pi/2} sin(x)\,dx$	1
Έξοδος κανόνα Simpson	1.0000033922209004
Αριθμητικό Σφάλμα	$\|e\| \approx 0.000003$
Θεωρητικό σφάλμα	$\|e\| \leq 0.000005$
Αριθμός διαμερίσεων	10

Το θεωρητικό σφάλμα e προκύπτει από τον τύπο $e \leq \frac{(b-a)^5}{180n^4}M$ όπου M το μέγιστο της τέταρτης παραγώγου της συνάρτησης στο διάστημα $[a, b]$, το οποίο

στην περίπτωση μας είναι 1 αφού η συνάρτηση μας είναι το ημίτονο. Κάνοντας αντικατάσταση όπου $a = 0, b = \frac{\pi}{2}$ και $n = 10$, προκύπτει τελικά $|e| \leq 0.000005$. Πράγματι το αριθμητικό μας σφάλμα ικανοποιεί αυτή την συνθήκη.

Τα αποτελέσματα είναι αναμενόμενα, αφού ο κανόνας Simpson έχει πετύχει καλύτερη προσέγγιση από τον κανόνα του τραπεζίου.

Άσκηση 7

Στην 7η ζητείται η πρόβλεψη των μετοχών δύο εταιρειών του χρηματιστηρίου Αθηνών με χρήση της μεθόδου ελαχίστων τετραγώνων για πολυώνυμο 2ου, 3ου και 4ου βαθμού Η ημερομηνία γενεθλίων μου είναι 22/3, οπότε για το έτος 2020 οι προηγούμενες 10 συνεδριάσεις που χρησιμοποιήθηκαν ως δεδομένα είναι οι εξής: 09/3, 10/3, 11/3, 12/3, 13/3, 16/3, 17/3, 18/3, 19/3 και 20/3, ενώ οι 5 επόμενες συνεδριάσεις για τις οποίες έγινε πρόβλεψη κλεισίματος μετοχής είναι οι εξής: 23/3, 24/3, 26/3, 27/3 και 30/3.

Η πρώτη εταιρεία που επιλέχθηκε είναι η Eurobank με κωδικό ΕΥΡΩΒ. Οι μετοχές για τις 10 προηγούμενες συνεδριάσεις φαίνονται στον παρακάτω πίνακα.

Αρ. Συνεδρίασης	Ημερομηνία	Μετοχή
1	09/3	0,4408
2	10/3	0,5000
3	11/3	0,4780
4	12/3	0,3950
5	13/3	0,4228
6	16/3	0,3800
7	17/3	0,3400
8	18/3	0,3234
9	19/3	0,3400
10	20/3	0,3654

Και στον παρακάτω πίνακα φαίνονται οι προβλέψεις για τις επόμενες 5 συνεδριάσεις.

Αρ. Συνεδρίασης	Ημερομηνία	Μετοχή	2ου βαθμού	3ου βαθμού	4ου βαθμού
11	23/3	0,3150	0,3309	0,4426	0,3957
12	24/3	0,3420	0,3276	0,5612	0,4292
13	26/3	0,4180	0,3267	0,7329	0,4519
14	27/3	0,3970	0,3280	0,9655	0,4486
15	30/3	0,4160	0,3316	1,2667	0,4010

Βαθμός Πολυωνύμου	RMSE
2	0,0643
3	0,4920
4	0,0603

Όπως παρατηρούμε, τα πολυώνυμο 2ου βαθμού έκανε μια αρκετά καλή εκτίμηση με $RMSE = 0.0643$. Οριακά καλύτερη ήταν η εκτίμηση του πολυωνύμου 4ου βαθμού με $RMSE = 0.0603$. Από την άλλη το πολυώνυμο 3ου βαθμού δεν ήταν καθόλου αποδοτικό αφού είχε $RMSE = 0.4920$, κάτι που είναι αρκετά απογοητευτικό.

Η δεύτερη εταιρεία που επιλέχθηκε είναι η Τράπεζα Πειραιώς με κωδικό ΠΕΙΡ. Οι μετοχές για τις 10 προηγούμενες συνεδριάσεις φαίνονται στον παρακάτω πίνακα.

Αρ. Συνεδρίασης	Ημερομηνία	Μετοχή
1	09/3	1,4460
2	10/3	1,5000
3	11/3	1,3800
4	12/3	1,1500
5	13/3	1,2250
6	16/3	1,0060
7	17/3	1,0000
8	18/3	0,9505
9	19/3	0,9655
10	20/3	1,0400

Και στον παρακάτω πίνακα φαίνονται οι προβλέψεις για τις επόμενες 5 συνεδριάσεις.

Αρ. Συνεδρίασης	Ημερομηνία	Μετοχή	2ου βαθμού	3ου βαθμού	4ου βαθμού
11	23/3	0,9630	1,0044	1,2168	1,0916
12	24/3	1,1010	1,0399	1,4840	1,1312
13	26/3	1,2400	1,0917	1,8639	1,1128
14	27/3	1,1980	1,1596	2,3714	0,9901
15	30/3	1,2350	1,2438	3,0214	0,7078

Βαθμός Πολυωνύμου	RMSE
2	0,0761
3	1,0167
4	0,2664

Όπως παρατηρούμε, τα πολυώνυμο 2ου βαθμού και αυτή την φορά έκανε μια πάρα πολύ καλή εκτίμηση του κλεισίματος των μετοχών με $RMSE = 0.0761$. Ικανοποιητική ήταν και η εκτίμηση του πολυωνύμου 4ου βαθμού με $RMSE = 0.2664$. Τέλος, το πολυώνυμο 3ου βαθμού δεν ήταν αποδοτικό για άλλη μια φορά, αφού έιχε $RMSE = 1.0167$.

Γενικότερα, αναμέναμε το πολυώνυμο 2ου βαθμού να είναι το πιο αποδοτικό από τα υπόλοιπα, αφού οι μετοχές είχαν συνήθως μικρές αυξομειώσεις.

Τέλος Εργασίας