Υπολογιστική Φυσική - 2η Σειρά Ασκήσεων

Αναστάσιος Τζαβέλλας / AM:1110 2016 00255

18/11/2017

1η Άσκηση

Ζητείται να βρεθούν όλες οι ρίζες της συνάρτησης $f(x) = x^2 + 10\cos x$ με τη μέθοδο του σταθερού σημείου. Καταρχάς, είναι σημαντικό να σημειωθεί ότι η f είναι άρτια (1), αφού είναι άθροισμα άρτιων συναρτήσεων. Επομένως, αν $\rho_1 > 0$ είναι ρίζα της εξίσωσης, τότε και η $-\rho_1$ θα είναι ρίζα.

$$f(-x) = (-x)^2 + 10\cos(-x) = x^2 + 10\cos x = f(x) \tag{1}$$

Επειδή η x^2 απειρίζεται για $x \to \infty$ και $|10\cos x| \leq 10$, αναμένεται ότι οι όποιες ρίζες της f θα βρίσκονται γύρω από την αρχή των αξόνων. Στο Σχήμα 1 δίνεται το γράφημα της f, όπου φαίνεται ότι τέμνει τον x άξονα 4 φορές και μάλιστα οι ρίζες είναι ανά δύο συμμετρικές ως προς το O.

Για επίλυση με τη μέθοδο σταθερού σημείου, υπάρχουν διάφορες επιλογές για την $g(x)$, όπου $x = g(x) \leftrightarrow f(x) = 0$. Αυτές είναι:

1.
$$g(x) = x - f(x) \to g(x) = x - x^2 - 10\cos x \tag{2}$$

2.
$$\begin{aligned} x^2 + 10\cos x = 0 \to \\ x^2 = -10\cos x \to \\ x = -10\frac{\cos x}{x} \to \\ g(x) = -10\frac{\cos x}{x} \end{aligned} \tag{3}$$

3.

$$
\begin{aligned}
x^2 + 10\cos x &= 0 \rightarrow \\
x^2 &= -10\cos x \rightarrow \\
x &= \pm\sqrt{-10\cos x} \rightarrow \\
g(x) &= \pm\sqrt{-10\cos x}
\end{aligned} \tag{4}
$$

4.

$$
\begin{aligned}
x^2 + 10\cos x &= 0 \rightarrow \\
\cos x &= -\frac{x^2}{10} \rightarrow \\
x &= \arccos\left(-\frac{x^2}{10}\right) \rightarrow \\
g(x) &= \arccos\left(-\frac{x^2}{10}\right)
\end{aligned} \tag{5}
$$

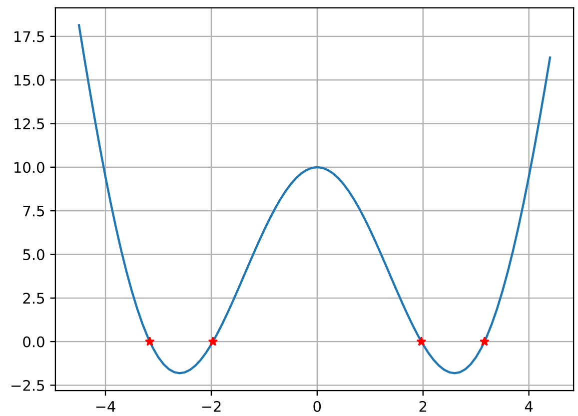
Για να συγκλίνει η μέθοδος του σταθερού σημείου σε μια ρίζα της f για κάποια από τις παραπάνω συναρτήσεις, πρέπει $|g'(x)| < 1$ για $x \in (a, b)$, όπου το διάστημα (a, b) περιέχει τη ρίζα της f.

Από τη γραφική παράσταση της f, μπορούν να αναζητηθούν οι θετικές ρίζες στα διαστήματα $(1.95, 2.5)$ και $(2.5, 3, 2)$. Σε αυτά τα διαστήματα δεν πληρούν όλες οι πιθανές g, το κριτήριο σύγκλισης. Στο διάστημα $I_1 = (1.95, 2.5)$ επιλέγεται η $g_1(x) = \arccos\left(-\frac{x^2}{10}\right)$ και στο διάστημα $I_2 = (2.5, 3, 2)$ επιλέγεται η $g_2(x) = \sqrt{-10\cos x}$.

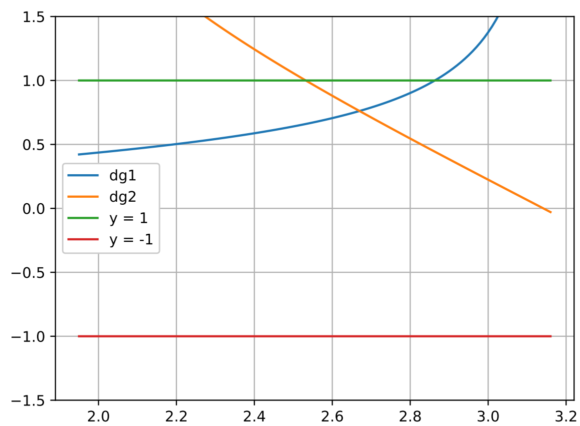
Στο Σχήμα 2 δίνεται η γραφική παράσταση των παραγώγων g_1' και g_2'. Όπως φαίνεται, και για τις δύο συναρτήσεις ισχύει $|g_i'(x)| < 1$, στα διαστήματα I_1 και I_2.

Ο αλγόριθμος της μεθόδου σταθερού σημείου εκτελέστηκε για g_1 και g_2 και τα αποτελέσματα του δίνονται στον Πίνακα 1 με ακρίβεια 10^{-4}. Όπως προαναφέρθηκε, ο προσδιορισμός των δύο θετικών ριζών επιτρέπει τον ταυτόχρονο προσδιορισμό και των αρνητικών και μάλιστα χωρίς επιπλέον υπολογιστικό κόστος.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση



Σχήμα 1: Γράφημα $x^2 + 10\cos x$ γύρω από το $O(0,0)$



Σχήμα 2: g_1' και g_2' κοντά στις ρίζες της f

g_i	$\rho > 0$	$\rho < 0$
$\arccos\left(-\frac{x^2}{10}\right)$	1.9689	-1.9689
$\sqrt{-10\cos x}$	3.1619	-3.1619

Πίνακας 1: Αποτελέσματα Προσομοίωσης

της βιβλιοθήκης Numpy. Ο αλγόριθμος της μεθόδου σταθερού σημείου δίνεται στο Παράρτημα.

ex1.py

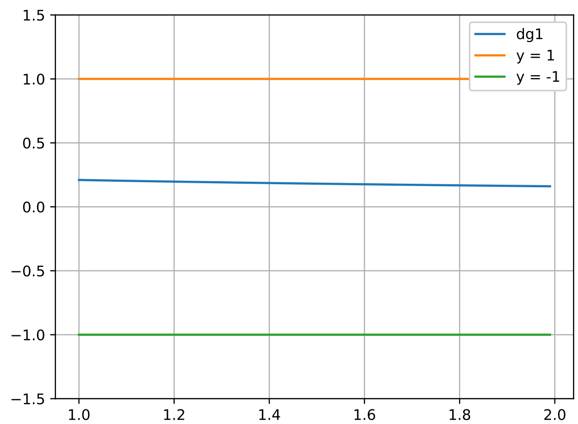
```
import numpy as np
import matplotlib.pyplot as plt
import equation


def f(x):
    return np.power(x, 2) + 10 * np.cos(x)


def g2(x):
    return np.sqrt(-10 * np.cos(x))


def dg2(x):
    return 10*np.sin(x) / (2 * np.sqrt(-10*np.cos(x)))


def g1(x):
    return np.arccos(- np.power(x, 2) / 10)


def dg1(x):
    return 2*x/np.sqrt(100 - np.power(x, 4))


x0 = 2.6
roots = np.zeros(4)
iterations = np.zeros(2)
```

```python
result = equation.FixedPoint(x0, g1, 1e-4)
roots[0] = result['p']
roots[1] = - roots[0]
iterations[0] = result['iterations']
print('Roots +-', roots[0],
      ' found after ', iterations[0], ' iterations')

result = equation.FixedPoint(x0, g2, 1e-4)
roots[2] = result['p']
roots[3] = - roots[2]
iterations[1] = result['iterations']
print('Roots +-', roots[2],
      'found after ', iterations[1], ' iterations')

x = np.arange(-4.5, 4.5, 0.1)
plt.close('all')
plt.figure(1)
plt.plot(x, f(x),
         roots[0], f(roots[0]), '*r',
         roots[1], f(roots[1]), '*r',
         roots[2], f(roots[2]), '*r',
         roots[3], f(roots[3]), '*r')
plt.grid()

x = np.arange(1.95, 3.17, 0.01)
plt.figure(2)
plt.plot(x, dg1(x), label='dg1')
plt.plot(x, dg2(x), label='dg2')
plt.plot(x, np.ones(x.shape), label='y = 1')
plt.plot(x, -np.ones(x.shape), label='y = -1')
plt.ylim(-1.5, 1.5)
plt.legend()
plt.grid()
```

2η Άσκηση

Για την εύρεση της ρίζας της $f(x) = x^3 - x - 1$ στο διάστημα $I = [1, 2]$, πρώτα πρέπει να βρεθεί κατάλληλη g τέτοια ώστε $x = g(x) \leftrightarrow f(x) = 0$. Πιθανές επιλογές για την g είναι η $g_1(x) = (x+1)^{\frac{1}{3}}$ και $g_2(x) = x^3 - 1$.

'Οπτικά' η g_2 φαίνεται πιο εύκολη επιλογή αλλά $g_2'(x) = 3x^2$ και $g_2' > 1$ για $x \in I$, οπότε απορρίπτεται. Επομένως, μένει η g_1 η οποία ικανοποιεί το κριτήριο σύγκλισης της ακολουθίας $x_{k+1} = g(x_k)$. Γραφικά, στο Σχήμα 3 δίνεται η παράγωγος της g_1.



Σχήμα 3: g_1' στο I

Ο αλγόριθμος σταθερού σημείου εκτελείται και συγκλίνει στη ρίζα $\rho = 1.32471$ με ακρίβεια 10^{-5} όπως ζητείται. Για την επίτευξη της συγκεκριμένης ακρίβειας, απαιτήθηκαν $N = 8$ επαναλήψεις. Γραφικά, η ρίζα της εξίσωσης φαίνεται στο Σχήμα 4.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Ο αλγόριθμος της μεθόδου σταθερού σημείου δίνεται στο Παράρτημα

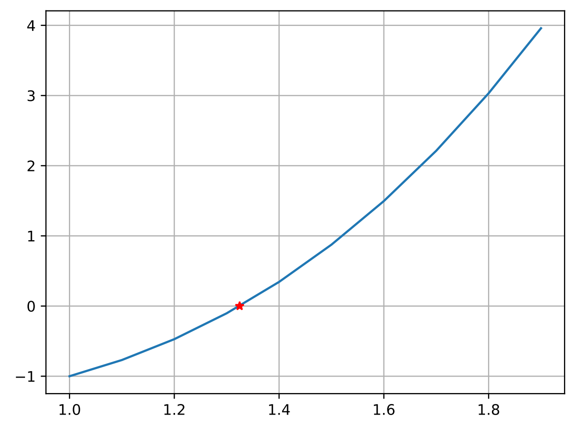
ex2.py

```
import numpy as np
import matplotlib.pyplot as plt
import equation


def f(x):
```

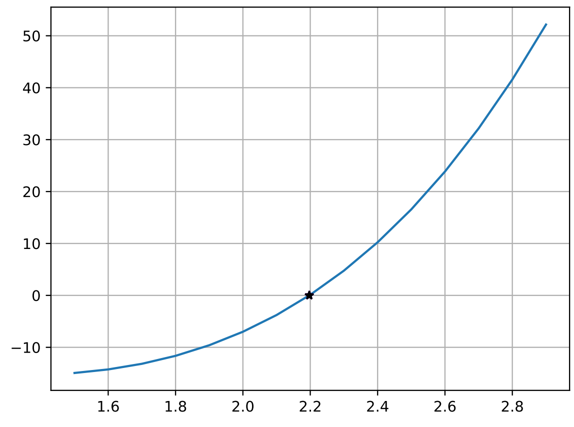
```python
    return np.power(x, 3) + -x - 1


def g1(x):
    return np.power(x + 1, 1/3)


def dg1(x):
    return np.power(x + 1, -2/3) / 3


x0 = 1
result = equation.FixedPoint(x0, g1, 1e-5)
root = result['p']
iterations = result['iterations']
print('Root', root, 'found after', iterations, 'iterations')

x = np.arange(1, 2, 0.1)
plt.close('all')
plt.figure(1)
plt.plot(x, f(x), root, f(root), '*r')
plt.grid()

x = np.arange(1, 2, 0.01)
plt.figure(2)
plt.plot(x, dg1(x), label='dg1')
plt.plot(x, np.ones(x.shape), label='y = 1')
plt.plot(x, -np.ones(x.shape), label='y = -1')
plt.ylim(-1.5, 1.5)
plt.legend()
plt.grid()
```

3η Άσκηση

Η συνάρτηση $f(x) = x^4 + 2x^3 - 5x^2 - 7x - 5$ στο διάστημα $I = [1.5, 3]$ φαίνεται στο Σχήμα 5. Η f είναι γνησίως αύξουσα στο I και ισχύει ότι $f(1.5)f(3) < 0$, επομένως η $f(x) = 0$ έχει μοναδική λύση στο I.



Σχήμα 4: Γράφημα $f(x) = x^3 - x - 1$ στο I



Σχήμα 5: Γράφημα $f(x) = x^4 + 2x^3 - 5x^2 - 7x - 5$ στο I

Μέθοδος Newton-Raphson Η μέθοδος Newton-Raphson συγκλίνει σε ρίζα μιας f, δεδομένου ότι η αρχική εκτίμηση x_0 είναι αρκετά κοντά στη ρίζα. Η εκτέλεση της μεθόδου έγινε για $x_0 = 2$ και βρέθηκε η ρίζα $\rho = 2.19714$ μετά από $N = 4$ επαναλήψεις.

Μέθοδος εσφαλμένης θέσης Η μέθοδος της εσφαλμένης θέσης επίσης συγκλίνει στη ρίζα της f στο I, αλλά πιο αργά από την Newton-Raphson. Συγκεκριμένα, έδωσε $\rho = 2.19714$ μετά από $N = 8$ επαναλήψεις.

Μέθοδος διχοτόμησης Η μέθοδος της διχοτόμησης είναι η πιο αργή από όλες τις μεθόδους και συνέκλινε στην τιμή $\rho = 2.19714$ μετά από $N = 18$ επαναλήψεις.

Από τη σύγκριση των τριών μεθόδων, η πιο γρήγορη είναι η Newton-Raphson και ο λόγος είναι ότι η ταχύτητα σύγκλισής της είναι τετραγωνική. Συγκρίνοντας την μέθοδο εσφαλμένης θέσης με αυτή της διχοτόμησης, η πρώτη είναι πιο γρήγορη αλλά όχι πάντα. Στη συγκεκριμένη περίπτωση, είναι αρκετά πιο γρήγορη και μάλιστα για διαφορετική επιλογή x_0, η Newton-Raphson μπορεί να συγκλίνει στον ίδιο αριθμό επαναλήψεων με την μέθοδο εσφαλμένης θέσης. Γενικά, η μέθοδος της διχοτόμησης είναι η πιο αργή, αλλά εγγυάται πάντα τη σύγκλιση σε μια ρίζα μιας συνάρτησης.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Η υλοποίηση των αλγορίθμων Newton-Raphson, εσφαλμένης θέσης και διχοτόμησης δίνεται στο Παράρτημα.

ex3.py

```python
import numpy as np
import matplotlib.pyplot as plt
import equation


def f(x):
    return np.power(x, 4) + 2 * np.power(x, 3) - 5 *\
           np.power(x, 2) - 7 * x - 5


```

```
def df(x):
    return 4 * np.power(x, 3) + 6 * \
           np.power(x, 2) - 10 * x - 7



x0 = 2
roots = np.zeros(3)

result = equation.NewtonRaphson(x0, f, df, 1e-5)
roots[0] = result['p']
iterations = result['iterations']
print('Newton Raphson: Root', roots[0],
      'found after', iterations, 'iterations')

result = equation.RegulaFalsi(1.5, 3, f, 1e-5)
roots[1] = result['p']
iterations = result['iterations']
print('RegulaFalsi Method: Root', roots[1],
      'found after', iterations, 'iterations')

result = equation.Bisection(1.5, 3, f, 1e-5)
roots[2] = result['p']
iterations = result['iterations']
print('Bisection Method: Root', roots[2],
      'found after', iterations, 'iterations')

x = np.arange(1.5, 3, 0.1)
plt.close('all')
plt.figure(1)
plt.plot(x, f(x),
         roots[0], f(roots[0]), '*r',
         roots[1], f(roots[1]), '*b',
         roots[2], f(roots[2]), '*k')
plt.grid()

x = np.arange(1.5, 3, 0.01)
plt.figure(2)
plt.plot(x, df(x), label='df')
plt.legend()
plt.grid()
```

4η Άσκηση

Ο πίνακας A παραγοντοποιείται σε LU ως:

$$\begin{bmatrix} 1 & 1 & 2 \\ -1 & 0 & 2 \\ 3 & 2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & -3 \end{bmatrix} \tag{6}$$

Κατόπιν, μπορεί να λυθεί το αρχικό σύστημα $Ax = b$ ως $Ly = b$ με εμπρός αντικατάσταση και στη συνέχεια λύνοντας $Ux = b$ με πίσω αντικατάσταση. Για το πρώτο σύστημα είναι

$$y = \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix} \tag{7}$$

και για το δεύτερο

$$x = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} \tag{8}$$

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Οι ρουτίνες παραγοντοποίησης LU, εμπρός και πίσω αντικατάστασης δίνονται στο Παράρτημα.

ex4.py

```python
import numpy as np
import solver


A = np.array([[1., 1., 2.],
              [-1., 0., 2.],
              [3., 2., -1.]])
b = np.array([1., -3., 8.])
ret = solver.LU(A)
L = ret['L']
U = ret['U']
print('L\n', L)
print('\nU\n', U)
```

```
y = solver.ForwardSubstitution(L, b)
print('\ny=', y)
x = solver.BackSubstitution(U, y)
print('x=', x)
```

5η Άσκηση

Για την επίλυση αυτής της άσκησης, υλοποιήθηκε ο αλγόριθμος απαλοιφής Gauss με ένα επιπλέον όρισμα, το οποίο καθορίζει το είδος της οδήγησης που θα εφαρμοστεί.

Το όρισμα είναι προαιρετικό οπότε σε περίπτωση που δεν δοθεί, εκτελείται ο αλγόριθμος χωρίς οδήγηση, παρά μόνο για έλεγχο μη μηδενικού στοιχείου οδηγού.

Η ορίζουσα του δοσμένου πίνακα μπορεί να υπολογιστεί εύκολα αφού εκτελεστεί ο αλγόριθμος απαλοιφής. Μετά την εκτέλεση του αλγορίθμου, ο αρχικός πίνακας είναι άνω τριγωνικός και επομένως η ορίζουσά του είναι το γινόμενο των διαγώνιων στοιχείων.

Τελικά είναι $det(A) = -8$ είτε εφαρμοστεί μερική είτε πλήρης οδήγηση. Η λύση με μερική οδήγηση είναι

$$x_{pp} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \tag{9}$$

και για πλήρη οδήγηση

$$x_{fp} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} \tag{10}$$

Η διαφορά στη λύση οφείλεται στις μεταθέσεις γραμμών που προέκυψαν κατά την εκτέλεση του αλγορίθμου.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Η υλοποίηση του αλγορίθμου απαλοιφής Gauss δίνεται στο Παράρτημα.

ex5.py

```
import numpy as np
import solver


def determinant(A):
    rows = A.shape[0]
    det = 1.  # Det of a triangular matrix is equal
    for i in range(rows):  # to the product of the
        det = det * A[i, i]  # diagonal entries
    return det


A = np.array([[1., 1., 2.], [1., -1., 0.], [-2., 2., 4.]])
B = np.array([7., 1., 2.])
x = np.zeros(3)

result1 = solver.GaussElimination(A, B, 'partial')
det1 = determinant(result1['E'])
print('Partial Pivot Determinant is:', det1)
print('Partial Pivot Solution is:', result1['x'])

result2 = solver.GaussElimination(A, B, 'complete')
det2 = determinant(result2['E'])
print('Complete Pivot Determinant is:', det2)
print('Complete Pivot Solution is:', result2['x'])
```

6η Άσκηση

Σε αυτή την άσκηση, είναι συνετό πριν εφαρμοστεί οποιοδήποτε επαναληπτικό σχήμα να εκτελεστούν κάποιες αλγεβρικές πράξεις. Ο στόχος είναι το σχήμα να μπορεί να προγραμματιστεί πιο εύκολα σε υπολογιστή και ενδεχομένως να αποφευχθούν επαναλήψεις υπολογισμών.

$$\begin{aligned}
x^{k+1} &= (1-\tau)x^k + Ux^{k+1} + (\tau-1)Ux^k + \tau Lx^k + \tau D^{-1}d \rightarrow \\
(I-U)x^{k+1} &= [(1-\tau)I - (1-\tau)U]\, x^k + \tau Lx^k + \tau D^{-1}d \rightarrow \\
x^{k+1} &= (1-\tau)Ix^k + \tau(I-U)^{-1}Lx^k + \tau(I-U)^{-1}D^{-1}d \rightarrow \\
x^{k+1} &= \left[(1-\tau)I + \tau(I-U)^{-1}L\right] x^k + \tau(I-U)^{-1}D^{-1}d \xrightarrow{M=I-U} \\
x^{k+1} &= Gx^k + c
\end{aligned} \tag{11}$$

όπου $G = (1-\tau)I + \tau M^{-1}L$, $c = \tau M^{-1}D^{-1}d$. Όπως φαίνεται από την εξίσωση (11), τα G και c είναι σταθερά και δεν αλλάζουν σε κάθε επανάληψη. Επομένως μπορούν να υπολογιστούν εξαρχής και απλά να χρησιμοποιούνται στο σώμα του υπολογιστικού βρόχου.

Επίσης, σχετικά με τον πίνακα $M = (I-U)$, από τον ορισμό του επαναληπτικού σχήματος, ο πίνακας U είναι αυστηρά άνω τριγωνικός, ο I είναι ο μοναδιαίος πίνακας, άρα η διαφορά τους θα είναι άνω τριγωνικός πίνακας, με μοναδιαία διαγώνια στοιχεία.

Ο υπολογισμός του αντιστρόφου είναι πλέον εύκολη υπόθεση αφού μπορεί να υπολογιστεί η j-στήλη του M^{-1}, ως η λύση ενός συστήματος $Mc_j = e_j$, όπου e_j η j-στήλη του μοναδιαίου πίνακα. Η επίλυση του τελευταίου συστήματος γίνεται με πίσω αντικατάσταση, αξιοποιώντας την ιδιότητα του M να είναι άνω τριγωνικός.

Αντίστοιχα εύκολος είναι και ο υπολογισμός του D^{-1}. Ο D είναι διαγώνιος πίνακας και δεδομένου ότι κανένα στοιχείο της διαγωνίου δεν είναι 0, ο αντίστροφός του είναι πάλι ένας διαγώνιος πίνακας μου τα διαγώνια στοιχεία τα αντίστροφα του D. Αξιοποιώντας τα παραπάνω, υπολογίζονται εκτός επαναληπτικού βρόχου D^{-1}, L, U, $(I-U)^{-1}$, G και τέλος c.

Για την επίδειξη της συμπεριφοράς του αλγορίθμου, χρησιμοποιήθηκαν ένας τριδιαγώνιος πίνακας $4x4$ και ένα διάνυσμα d. Ο αλγόριθμος εκτελέστηκε για $\tau = 0.1, 0.2, \ldots, 1.9$ με σκοπό να μελετηθεί η συμπεριφορά του επαναληπτικού σχήματος.

Για τα συγκεκριμένα A, d ο αλγόριθμος συνέκλινε στη ίδια λύση για κάθε

τ. Όμως, η ταχύτητα σύγκλισης του αλγορίθμου ήταν διαφορετική για κάθε τ. Παρατηρήθηκε ότι χρειάστηκαν λιγότερες επαναλήψεις (23) για $\tau = 1.4$ ενώ για τις ακραίες τιμές, χρειάστηκαν ως και 326 επαναλήψεις.

Για άλλα ζεύγη A και d υπήρχε ανάλογη συμπεριφορά με τη διαφορά ότι για μεγάλα τ, ο αλγόριθμος δεν συνέκλινε. Επομένως, μπορεί να εξαχθεί το συμπέρασμα ότι η παράμετρος τ επηρεάζει αφενός τη σύγκλιση του αλγορίθμου και αφετέρου την ταχύτητά της.

Θεωρητική ανάλυση της σύγκλισης ή μη καθώς και της ταχύτητας μπορεί να γίνει υπολογίζοντας την φασματική ακτίνα του G, η οποία σε περίπτωση σύγκλισης πρέπει να είναι $\rho(G) < 1$.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Οι ρουτίνες υπολογισμού δίνονται στο Παράρτημα.

ex6.py

```
import numpy as np
import solver


A = np.array([[2., -1., 0., 0.],
              [-1., 2., -1., 0.],
              [0., -1., 2., -1.],
              [0., 0., -1., 2.]])
d = np.array([1., 0., 0., 1.])

taus = np.arange(.1, 2., .1)
for tau in taus:
    result = solver.IterativeMethod(A, d, .5e-6, tau, 1000)
    x = result['x']
    iterations = result['iterations']
    print('tau=', tau, ': ', iterations, ', x=', x)
    delta = np.dot(A, x) - d
    print('Norm (Ax-d) =', np.sqrt(np.dot(delta, delta)))
```

Παραρτήματα

Κώδικας Αριθμητικής επίλυσης εξισώσεων

equation.py

```
import numpy as np


def FixedPoint(x0, g, tol, N=None):
    i = 1
    p0 = x0
    while (True if N is None else (i < N)):
        p = g(p0)
        if np.abs(p-p0) < tol:
            break
        i = i + 1
        p0 = p
    return {'p': p, 'iterations': i}


def NewtonRaphson(x0, f, df, tol, N=None):
    i = 1
    p0 = x0
    while (True if N is None else (i < N)):
        p = p0 - f(p0)/df(p0)
        if np.abs(p-p0) < tol:
            break
        i = i + 1
        p0 = p
    return {'p': p, 'iterations': i}


def RegulaFalsi(a, b, f, tol, N=None):
    i = 1
    p0 = a
    p1 = b
    q0 = f(a)
    q1 = f(b)
    while (True if N is None else (i < N)):
        p = p1 - q1 * (p1 - p0)/(q1 - q0)
        if np.abs(p - p1) < tol:
            break
        i = i + 1
        p0 = p1
```

```
        q0 = q1
        p1 = p
        q1 = f(p)
    return {'p': p, 'iterations': i}


def Bisection(a, b, f, tol, N=None):
    i = 1
    c_old = 0
    while (True if N is None else (i < N)):
        c = (a + b) / 2
        if np.isclose(f(c), 0) or (i > 1 and (np.abs(c-c_old) <
    tol)):
            break
        i = i + 1
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
        c_old = c
    return {'p': c, 'iterations': i}
```

Κώδικας Επίλυσης Γραμμικών Συστημάτων

solver.py

```
import numpy as np


def swap(r, p, E, cols=None):
    if cols is None:
        b = np.copy(E[r, :])
        E[r, :] = E[p, :]
        E[p, :] = np.copy(b)
    else:
        b = np.copy(E[:, r])
        E[:, r] = E[:, p]
        E[:, p] = np.copy(b)


```

```
def ForwardSubstitution(A, b):
    rows = A.shape[0]
    y = np.zeros(rows)
    for i in range(rows):
        s = 0.
        for j in range(i):
            s = s + A[i, j] * y[j]
        y[i] = (b[i] - s) / A[i, i]
    return y


def BackSubstitution(A, b):
    rows = A.shape[0]
    x = np.zeros(rows)
    for i in reversed(range(rows)):
        s = 0
        for j in range(i + 1, rows):
            s = s + A[i, j] * x[j]
        x[i] = (b[i] - s) / A[i, i]
    return x


def foundNonZeroPivot(r, E):
    rows = E.shape[0]
    PivotFound = False
    for p in range(r, rows - 1):
        if np.isclose(E[p, r], 0):  # Keep looking for non-zero
   pivot
            continue
        else:  # if pivot is found
            PivotFound = True
            if p > r:  # Only swap if p>r
                swap(r, p, E)
            break
    return PivotFound


def partialPivot(r, A):
    rows = A.shape[0]  # Number of rows
    Amax = np.abs(A[r, r])
```

```
    rmax = r
    for p in range(r, rows):
        Apr = np.abs(A[p, r])
        if Apr > Amax:
            Amax = Apr
            rmax = p
    if rmax != r:
        swap(r, rmax, A)
    return


def completePivot(r, A):
    rows, cols = A.shape
    cols = cols - 1  # ignore the last column of the augmented matrix
    Amax = np.abs(A[r, r])
    rmax = r
    cmax = r
    for i in range(r, rows):
        for j in range(r, cols):
            Aij = np.abs(A[i, j])
            if Aij > Amax:
                Amax = Aij
                rmax = i
                cmax = j
    if (rmax != r) and (cmax != r):
        swap(r, rmax, A)
        swap(r, cmax, A, True)
    return


def GaussElimination(A, b, pivot=None):
    isSingular = False
    rows = A.shape[0]
    b = b.reshape(rows, 1)
    E = np.append(A, b, axis=1)  # Append b as extra column
    for r in range(rows - 1):
        if pivot == 'partial':
            partialPivot(r, E)
        elif pivot == 'complete':
```

```python
                completePivot(r, E)
        else:  # Simple pivot is required to avoid division by 0
            isSingular = not foundNonZeroPivot(r, E)
            if isSingular:
                break
        for i in range(r + 1, rows):
            if np.isclose(E[i, r], 0):  # skip line if pivot is already 0
                continue
            m = - E[i, r]/E[r, r]
            E[i][r] = 0
            for j in range(r + 1, rows + 1):
                E[i, j] = E[i, j] + m * E[r, j]
    if isSingular:
        print("Matrix is singular")
    elif np.isclose(E[rows-1, rows-1], 0):
        print("There is no unique solution")
    else:
        y = E[:, rows]
        E = np.delete(E, [rows], axis=1)
        x = BackSubstitution(E, y)
    return {'E': E, 'x': x, 'isSingular': isSingular}


def LU(A):
    rows, cols = A.shape
    assert (rows == cols)
    U = np.copy(A)
    L = np.identity(rows)
    for k in range(rows-1):
        for j in range(k+1, rows):
            L[j, k] = U[j, k] / U[k, k]
            for p in range(k, rows):
                U[j, p] = U[j, p] - L[j, k] * U[k, p]
    return {'L': L, 'U': U}


def inverseU(U):
    rows = U.shape[0]
    Uinv = np.zeros(U.shape)
```

```python
    unit = np.identity(rows)
    for j in range(rows):
        b = unit[:, j]
        Uinv[:, j] = BackSubstitution(U, b)
    return Uinv


def IterativeMethod(A, d, tol, tau=0.1, N=None):
    rows = A.shape[0]
    Dinv = np.diag(1./A.diagonal())
    CL = -np.tril(A, -1)
    CU = -np.triu(A, 1)

    L = np.dot(Dinv, CL)
    U = np.dot(Dinv, CU)

    xk = d
    M = np.identity(rows) - U
    Minv = inverseU(M)
    c = tau * np.dot(Minv, np.dot(Dinv, d))
    G = (1.-tau) * np.identity(rows) + tau * np.dot(Minv, L)
    i = 1
    while (True if N is None else (i < N)):
        x = np.dot(G, xk) + c
        dx = x - xk  # Check if scheme converges
        if np.sqrt(np.dot(dx, dx)) < tol:
            break
        i = i + 1
        xk = x  # Prepare for next iteration
    return {'x': x, 'dx': dx, 'iterations': i}
```

Εκτέλεση Προγραμμάτων

```
In [9]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex1.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: equation
Roots +- 1.96891033395  found after  12.0  iterations
Roots +- 3.16195016268 found after  5.0  iterations

In [10]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex2.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: equation
Root 1.32471737244 found after 8 iterations

In [11]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex3.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: equation
Newton Raphson: Root 2.19714054608 found after 4 iterations
RegulaFalsi Method: Root 2.19714054607 found after 8 iterations
Bisection Method: Root 2.19714546204 found after 18 iterations

In [12]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex4.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: equation
L
 [[ 1.  0.  0.]
 [-1.  1.  0.]
 [ 3. -1.  1.]]

U
 [[ 1.  1.  2.]
 [ 0.  1.  4.]
 [ 0.  0. -3.]]

y= [ 1. -2.  3.]
x= [ 1.  2. -1.]

In [13]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex5.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: solver
Partial Pivot Determinant is: -8.0
Partial Pivot Solution is: [ 3.  2.  1.]
Complete Pivot Determinant is: -8.0
Complete Pivot Solution is: [ 1.  3.  2.]

In [14]:         'C:/Users/extern.a.Tzavellas/Downloads/ComputationalPhysics-master/
ComputationalPhysics-master/Assignment2/ex6.py'      ='C:/Users/extern.a.Tzavellas/
Downloads/ComputationalPhysics-master/ComputationalPhysics-master/Assignment2'
Reloaded modules: solver
tau= 0.1 :  326 , x= [ 0.99999647  0.99999294  0.99999128  0.99999334]
Norm (Ax-d) = 6.20896179109e-06
tau= 0.2 :  170 , x= [ 0.99999827  0.99999653  0.99999571  0.99999672]
Norm (Ax-d) = 3.05234604818e-06
tau= 0.3 :  116 , x= [ 0.99999897  0.99999794  0.99999745  0.99999805]
Norm (Ax-d) = 1.8135575875e-06
```

1

Σχήμα 6: Εκτέλεση Προγραμμάτων