Αριθμιτική Ανάλυση
1η Εργασία

Γραμμένος Αναστάσης
AEM:2345

28 Δεκεμβρίου 2016

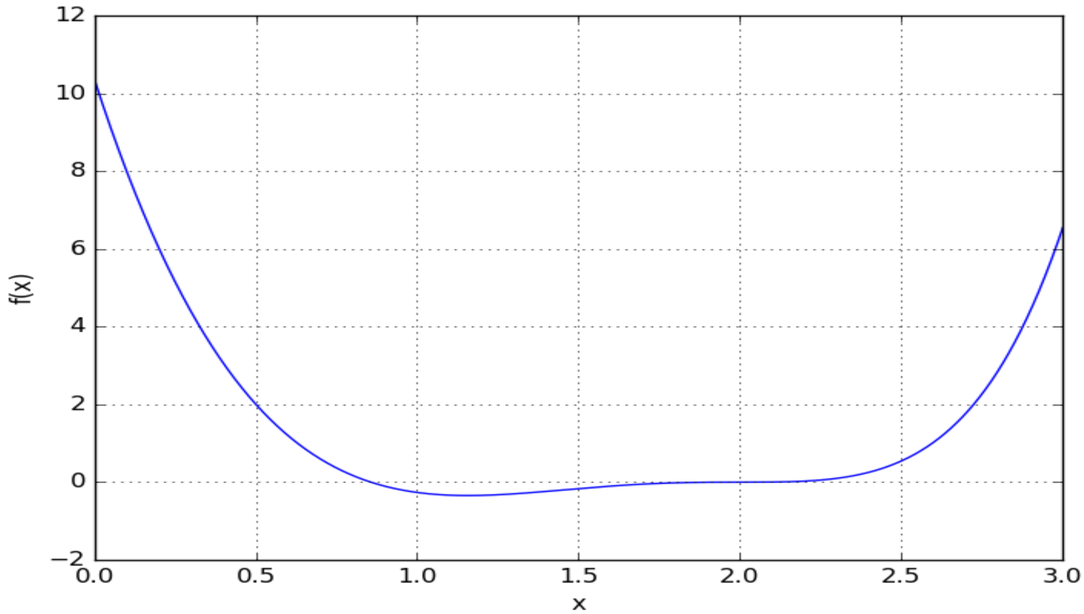
1 Άσκηση 1

Στην αρχή έχω τα απαραίτητα imports για `numpy`, `scipy` και `matplotlib`:

```
from numpy import *
import numpy

from scipy import *
import scipy

import matplotlib.pyplot as plt
```

Η συνάρτηση και οι 2 πρώτες παράγωγοι ορίζονται και γίνεται το plot της $f(x)$:

```
t = arange(0,3.001,step=.001)

def f(t):
  return (14*t)*(e**(t-2)) - 12*(e**(t-2)) - 7*(t**3) + 20*(t**2) - 26*t + 12

def f_der(t):
  return (14*t)*(e**(t-2)) + 2*(e**(t-2)) - 21*(t**2) + 40*t - 26

def f_der_2(t):
  return (14*t)*(e**(t-2)) - 16*(e**(t-2)) - 42*t + 40

plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(t, f(t))
plt.grid(True)
plt.show()
```



Καθώς η άσκηση δεν το απαιτεί δεν υπάρχουν έλεγχοι για σφάλματα ούτε συνθήκες τερματισμού σε περίπτωση που δεν υπάρχουν ρίζες.

1.1 Διχοτόμηση

Ακολουθεί η συνάρτηση για διχοτόμηση:

```
def bisect(a, b):
  a_1, b_1 = a, b
  root = (a + b) / 2

  # number of times to repeat to achieve 6 points accuracy
  N = int(ceil((log(b - a) - log(0.0000005)) / log(2)))

  for i in range(0, N):
    if (f(a) < 0 and f(root) > 0) or (f(a) > 0 and f(root) < 0):
      b = root
    elif (f(b) < 0 and f(root) > 0) or (f(b) > 0 and f(root) < 0):
      a = root
    root = (a + b) / 2

  return root, N, a_1, b_1
```

Απλά γίνετε η εγαρμογή του θεωρήματος για $N = \frac{\ln{(b-a)} - \ln{(error)}}{\ln 2}$ φορές

1.2 Newton-Raphson

Στην συνέχεια έχω τη συνάρτηση για την μέθοδο Newton-Raphson:

```
def new_raph(start):
  temp_l = [start, start - (f(start)/f_der(start))]

  N = 1
  while round(f(temp_l[-1]), 6) != 0.000000:
    temp =  temp_l[N] - (f(temp_l[N])/f_der(temp_l[N]))
    temp_l.append(temp)
    N = N + 1

  root = temp_l[N]

  return root, N, start
```

Εδώ αρχικοποιώ τον πίνακα `temp_l` με την τιμή εισόδου της συνάρτησης και το αποτέλεσμα μιας πρώτης εφαρμογής της αναδρομικής συνάρτησης

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

Στην συνέχεια με τον έλεγχο στο while η συνάρτηση θα τρέχει μέχρι να επιτευχθεί η επιθυμιτή ακρίβεια (6 δεκαδικά ψηδία)

Επιστρέφω την ρίζα, τον αριθμό επαναλήψεων και το σημείο εκκίνησης.

1.3 Τέμνουσα

Ακολουθέι η συνάρτηση της μεθόδου της τέμνουσας:

```
def interpolation(a, b):
  temp_l = [a, b, b - ((f(b)*(b - a))/f(b) - f(a))]

  N = 1
  while round(f(temp_l[-1]), 6) != 0.000000:
    temp =  temp_l[N] - (f(temp_l[N])*
                         (temp_l[N] - temp_l[N-1]))/(f(temp_l[N]) - f(temp_l[N
   -1]))
    temp_l.append(temp)
    N = N + 1

  root = temp_l[-1]

  return root, N, a, b
```

Η συνάρτηση είναι αντίστοιχη με αυτή της Newton-Raphson. Χρειάζεται δύο αρχικές τιμές, και για να γίνει ο έλεγχος του `while`, υπολογίζω και την τρίτη σύμφωνα με την αναδρομική συνάρτηση

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

Ο έλεγχος είναι ο ίδιος με την Newton-Raphson για να πετύχω τα 6 δεκαδικά ψηφία
Επιστρέφω την ρίζα, τον αριθμό επαναλήψεων καθως και τις 2 αρχικές τιμές.

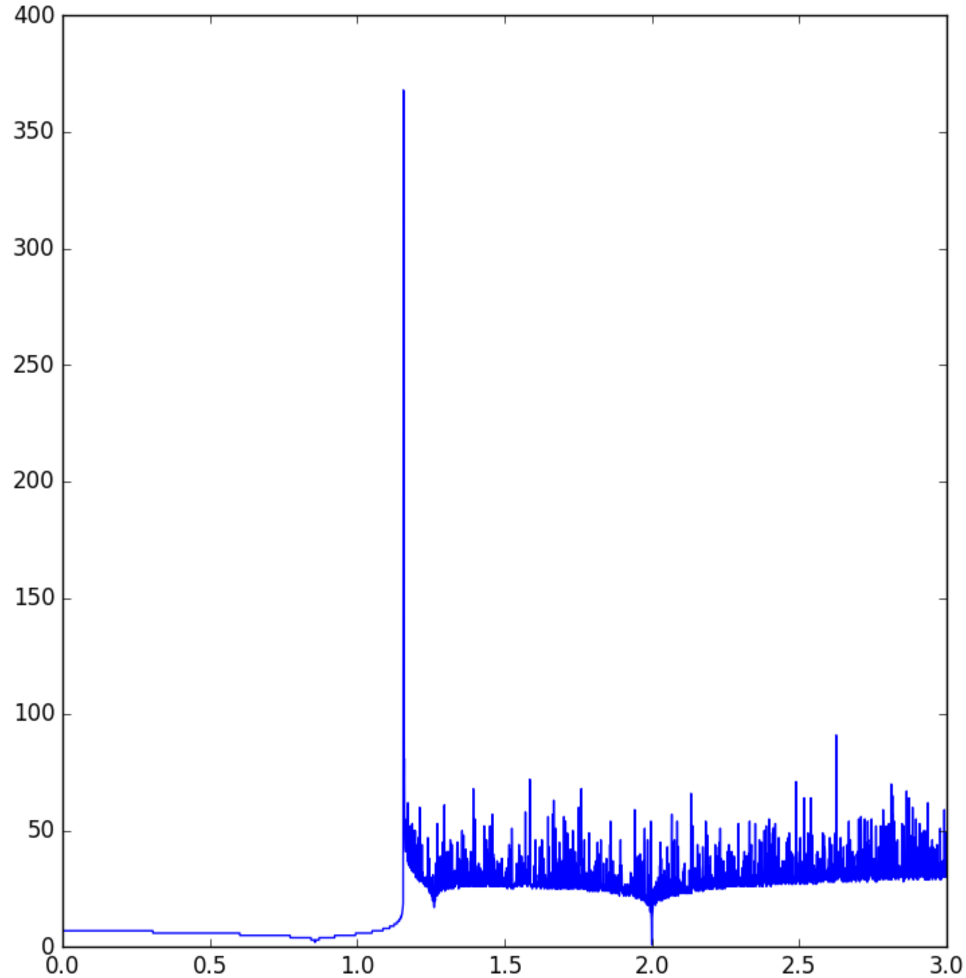
1.4 Αποτελέσματα

Ακολουθεί η έξοδος του προγράμματος όταν το τρέχω με τις κατάλληλες αρχικές τιμές (βασισμένες στο διάγραμμα την συνάρτησης):

```
$ python ex1/ex1.py
++++ Bisection ++++

Root in [0.00,1.50] after 22 loops: f(0.857143) = -0.000000

Root in [1.50,3.00] after 22 loops: f(2.000004) = 0.000000

++++ Newton - Raphson ++++

Starting at 1.00:
after 5 iterations the root is: f(0.857143) = 0.000000

Starting at 3.00:
after 14 iterations the root is: f(2.005224) = 0.000000

++++ Interpolation ++++

Root found in [0.70,0.90] after 10 iterations:
f(0.857143) = 0.000000

Root found in [1.70,2.10] after 20 iterations:
f(2.004977) = 0.000000
```

1.5 Παρατηρήσεις

Στο διάγραμμα που ακολουθεί βλέπουμε τις επαναλήψεις που χρειάζετε η μέθοδος Newton-Raphson σε σχέση με την αρχική τιμή που δίνεται



Παρατηρώ οτι για την ρίζα στο 0.857143 η μέθοδος συγκλίνει τετραγωνικά ενώ για την ρίζα στο 2.000004 θέλει εμφανώς περισσότερες επαναλήψεις.

Συγκρίνοντας αυτό το διάγραμμα με το plot της συνάρτησης στην 1η σελίδα παρατηρώ οτι η ταχύτητα σύγκλισης στην πρώτη ρίζα είναι μεγάλη γιατι η συνάρτηση διατηρεί την ίδια κυρτότητα ενώ στην δεύτερη η συνάρτηση αλλάζει.

Καθώς η μέθοδος N-R χρησιμοποιεί την κλίση της παραγόγου για να βρεί το επόμενο x στην αναδρομή, όταν η κυρτότητα αλλάζει γύρω απο την ρίζα ο αλγόριθμος ταλαντώνεται και αργεί να συγκλίνει σε έναν αριθμό που να ικανοποιεί το σφάλμα που θέσαμε.

2 Άσκηση 2

Ο κώδικας σε αυτήν την άσκηση είναι παρόμοιος με τον κώδικα της πρώτης με το μόνο άξιο σχολιασμού να είναι η τροποποιημένη μέθοδος της τέμνουσας:

```
def al_interpolation(a, b, c):
  x = [a, b, c]

  N = 0
  root = x[0]
  while round(f(root),6) != 0:
    r = f(x[(N+2)%3])/f(x[(N+1)%3])
    q = f(x[(N)%3])/f(x[(N+1)%3])
    s = f(x[(N+2)%3])/f(x[(N)%3])
    tmp = x[(N+2)%3] - (r*(r - q)*(x[(N+2)%3]-x[(N+1)%3]) +
                        (1 - r)*s*(x[(N+2)%3] - x[(N)%3]))/((q - 1)*(r - 1)*(s
    - 1))
    x[N%3] = tmp
    N = N + 1
    root = x[(N)%3]

  return root, N - 1, a, b, c
```

καθως η άσκηση λέει πως το x_{n+3} αντικαθιστά το x_n δημιουργώ έναν πίνακα 3 θέσεων και χρησιμοποιώ τον τελεστή % για να γίνεται η αντικατάσταση κυκλικά

2.1 Ερώτημα 1

Ακολουθούν τα αποτελέσματα με κατάλληλες αρχικοποιήσεις:

```
$ python ex2/ex2.py
++++ almost-Bisection ++++

Root in [0.80,0.90] after 25 loops: f(0.841067) = 0.000000

Root in [0.95,1.10] after 7 loops: f(1.047667) = 0.000000

Root in [2.30,2.80] after 25 loops: f(2.300524) = 0.000000

++++ almost-Newton - Raphson ++++

Starting at 0.80:
after 4 iterations the root is: f(0.841069) = 0.000000

Starting at 1.00:
after 6 iterations the root is: f(1.044162) = -0.000000
```

```
Starting at 2.50:
after 4 iterations the root is: f(2.300524) = -0.000000

++++ almost-Interpolation ++++

Starting points: [1.00, 2.00, 3.00]. After 10 iterations:
f(1.043381) = -0.000000

Starting points: [0.70, 0.80, 0.90]. After 10 iterations:
f(0.841069) = -0.000000

Starting points: [2.20, 2.30, 2.40]. After 4 iterations:
f(2.300524) = 0.000000
```

2.2 Ερώτημα 2

Εκτελώντας τον αλγόριθμο 10 φορές παρατηρώ οτι κάθε φορά συγκλίνει σε μια ρίζα με διαφορετικό αριθμό επαναλήψεων, πράγμα αναμενόμενο καθως διαλέγει την ρίζα τυχαία.

```
0: Root in [0.00,3.00] after 32 loops: f(0.841069) = -0.000000
1: Root in [0.00,3.00] after 48 loops: f(2.300524) = 0.000000
2: Root in [0.00,3.00] after 28 loops: f(0.841069) = -0.000000
3: Root in [0.00,3.00] after 25 loops: f(2.300524) = 0.000000
4: Root in [0.00,3.00] after 34 loops: f(2.300524) = -0.000000
5: Root in [0.00,3.00] after 40 loops: f(2.300524) = -0.000000
6: Root in [0.00,3.00] after 48 loops: f(2.300524) = -0.000000
7: Root in [0.00,3.00] after 36 loops: f(2.300524) = -0.000000
8: Root in [0.00,3.00] after 19 loops: f(0.841070) = -0.000000
9: Root in [0.00,3.00] after 34 loops: f(2.300524) = 0.000000
```

2.3 Ερώτημα 3

Παρατηρώ πως η μέθοδος διχοτόμησης είναι σαφώς καλύτερη όταν επιλέγω το μεσαίο σημείο αντί κάποιου τυχαίου.

Η μέθοδος N-R είναι παρόμοια χωρίς μεγάλες διαφορές και η μέθοδος της τέμνουσας είναι καλύτερη στην εναλακτική της μορφή με τα 3 αρχικά σημεία καθως γίνεται καλύτερη προσέγκιση με καμπύλη αντί μιας ευθείας.

Ακολουθούν αναλυτικά τα αποτελέσματα της σύγκρισης:

```
++++ almost-Bisection ++++
Root in [0.80,0.90] after 18 loops: f(0.841069) = -0.000000
Root in [0.95,1.10] after 19 loops: f(1.047192) = 0.000000
Root in [2.30,2.80] after 20 loops: f(2.300524) = 0.000020

++++ almost-Bisection ++++
Root in [0.80,0.90] after 20 loops: f(0.841069) = -0.000000
```

```
Root in [0.95,1.10] after 8 loops: f(1.045441) = -0.000000
Root in [2.30,2.80] after 35 loops: f(2.300524) = -0.000000

++++ Newton - Raphson ++++
Starting at 0.80:
after 4 iterations the root is: f(0.841069) = 0.000000
Starting at 1.00:
after 6 iterations the root is: f(1.043596) = -0.000000
Starting at 2.50:
after 4 iterations the root is: f(2.300524) = -0.000000

++++ almost-Newton - Raphson ++++
Starting at 0.80:
after 4 iterations the root is: f(0.841069) = 0.000000
Starting at 1.00:
after 6 iterations the root is: f(1.044162) = -0.000000
Starting at 2.50:
after 4 iterations the root is: f(2.300524) = -0.000000

++++ Interpolation ++++
Starting points: [1.00, 3.00]. After 17 iterations:
f(1.043599) = -0.000000
Starting points: [0.70, 0.90]. After 31 iterations:
f(0.841069) = 0.000000
Starting points: [2.20, 2.40]. After 29 iterations:
f(13.610043) = -0.000000

++++ almost-Interpolation ++++
Starting points: [1.00, 2.00, 3.00]. After 10 iterations:
f(1.043381) = -0.000000
Starting points: [0.70, 0.80, 0.90]. After 10 iterations:
f(0.841069) = -0.000000
Starting points: [2.20, 2.30, 2.40]. After 4 iterations:
f(2.300524) = 0.000000
```

3 Άσκηση 3

3.1 Ερώτημα 1

Έχω υλοποιήσει σε `python` ένα πρόγραμμα που διαβάζει 2 αρχεία: το `a.csv` με έναν $n \times n$ πίνακα $\mathbf{A}$ και το `b.csv` με έναν $1 \times n$ πίνακα $\mathbf{b}$ και κάνει την **LU** ανάλυση του $\mathbf{A}$ και στην συνέχεια βρίσκει την λύση στο σύστημα $\mathbf{Ax} = \mathbf{b}$

Αφού φτιάξει τους πίνακες $\mathbf{A}$ και $\mathbf{b}$ καλέι την συνάρτηση `gauss(A, b)` η οποία επιστρέφει το x.

Η συνάρτηση αρχικοποιεί τους πίνακες $\mathbf{L}$, $\mathbf{U}$ και $\mathbf{x}$ και ξεκινάει υπολογίζοντας τους 2 πρώτους. Στην συνέχεια βρίσκει το ενδιάμεσο y. Μετά βρίσκει το ανάποδο x και το αντιστρέφει για να επιστραφεί σωστά.

Το αρχείο `a.csv` πρέπει να περιέχει n νούμερα ανα σειρά χωρισμένα με `space` και n σειρές ενώ το `b.csv` n σειρές απο 1 νούμερο.

3.2 Ερώτημα 2

Σε αυτό το ερώτημα το πρόγραμμα διαβάζει τον πίνακα απο ένα αρχείο `cho.csv` και καλεί την μέθοδο `cholesky(A)` η οποία τυπώνει την ανάλυση Cholesky αυτού του πίνακα.

3.3 Ερώτημα 3

Εδώ ο πίνακας **A** και **b** δημιουργείτε απο το πρόγραμμα.

Η μεταβλητή `size` στην γραμμή 38 ελέγχει το n. Στην συνέχεια καλείτε η συνάρτηση `GS(A, b)` η οποία επιστρέφει τον πίνακα x.