Set 1

Σεϊτανίδου Δήμητρα

30 Ιανουαρίου 2020

Άσκηση 1

Για να υπολογίσουμε τις ρίζες των δύο εξισώσεων χρησιμοποιήσαμε την μέθοδο της διχοτόμησης και την μέθοδο Newton-Rapshon.

Για την μέθοδο της διχοτόμησης ως αρχικές τιμές παίρνουμε τα άκρα του διαστήματος $[x_1, x_2]$ και υπολογίζουμε σαν πιθανή ρίζα το:

$$x_3 = (x_1 + x_2)/2 \tag{1}$$

Έπειτα υπολογίζουμε τα $f(x_3) * f(x_1)$ και $f(x_3) * f(x_2)$ και ανάλογα ποιο από τα δύο γινόμενα είναι μικρότερο του μηδενός, γνωρίζουμε ότι ανάμεσα σε εκείνα τα x βρίσκεται η ρίζα και θέτουμε $x_2 = x_3$ ή $x_1 = x_3$ αντίστοιχα και υπολογίζουμε εκ νέου τη ρίζα από την σχέση (1). Επαναλαμβάνουμε την διαδικασία μέχρι να φτάσουμε στην επιθυμητή ακρίβεια. Εδώ και για τις δύο περιπτώσεις θέλουμε 5 σημαντικά ψηφία δηλαδή συνθήκη εξόδου από τον βρόχο είναι η σχέση $e < 0.0005$.

Με την μέθοδο Newton-Rapshonπροσεγγίζουμε την ρίζα της εξίσωσης σύμφωνα με την σχέση:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2}$$

Εργαζόμαστε ως εξής: Θέτουμε μια αρχική τιμή για το x_i και υπολογίζουμε αναδρομικά μέσω της σχέσης (2) την ρίζα μέχρι να φτάσουμε την επιθυμητή ακρίβεια, ίδια με αυτή της μεθόδου διχοτόμησης.

Εφαρμόζοντας την παραπάνω διαδικασία στις δύο εξισώσεις που μας δίνονται βρήκαμε τα εξής:

Για την εξίσωση $\mathbf{e^x - 2xcos(x) = 0}$

- Μέθοδος διχοτόμησης: χρησιμοποιήσαμε σαν αρχικές τιμές τα άκρα του διαστήματος $x_1 = 0, x_2 = 2$ και μετά από 12 επαναλήψεις βρήκαμε $\mathbf{x_r = 1.3042}$.
- Μέθοδος Newton-Rapshon: επιλέγουμε τιμή εκκίνησης το $x_1 = 0$ και μετά από 13 επαναλήψεις βρίσκουμε $\mathbf{x_r = 1.3046}$.

Για την εξίσωση $\mathbf{x^2 - sin(x) + e^x - 2 = 0}$

- Μέθοδος διχοτόμησης: χρησιμοποιήσαμε σαν αρχικές τιμές τα άκρα του διαστήματος $x_1 = 0, x_2 = 1$ και μετά από 11 επαναλήψεις βρήκαμε $\mathbf{x_r = 0.7505}$.
- Μέθοδος Newton-Rapshon: επιλέγουμε τιμή εκκίνησης το $x_1 = 0.5$ και μετά από 4 επαναλήψεις βρίσκουμε $\mathbf{x_r = 0.7507}$.

Συμπέρασμα: Στην πρώτη εξίσωση οι μέθοδοι έχουν σχεδόν ίδια απόδοση, ενώ στην δεύτερη η Newton-Rapshon είναι πολύ πιο γρήγορη. Παρακάτω δίνεται ο κώδικας.

```
import math as m

#define the function as given in the instactions
fa= lambda x: m.exp(x)-2*x*m.cos(x)-3 #x[0,2]
fb= lambda x: pow(x,2)-m.sin(x)+m.exp(x)-2 #x[0,1]
dfa= lambda x: m.exp(x)-2*m.cos(x)+2*x*m.sin(x)
dfb= lambda x: 2*x-m.cos(x)+m.exp(x)

i_ba=0; i_nra=0

#bisection method for fa
x1=0; x2=2
while (abs(x2-x1)>0.0005):
    i_ba=i_ba+1
    x3=(x1+x2)/2
    if fa(x1)*fa(x3)<0:
        x2=x3
    elif fa(x1)*fa(x3)>0:
        x1=x3
    else:
        break
print('bisection fa: %.4f' %x3,'\n iterations:',i_ba)

#Newton-Raphson method for fa
x1=0; e_a=1
while (e_a>0.0005):
    i_nra=i_nra+1
    x2=x1-(fa(x1)/dfa(x1))
    e_a=abs(fa(x1)/dfa(x1))
    x1=x2
print('Newton-Raphson fa: %.4f' %x2,'\n iterations:',i_nra)


i_bb=0; i_nrb=0

#bisection method for fb
x1=0; x2=1
while (abs(x2-x1)>0.0005):
    i_bb=i_bb+1
    x3=(x1+x2)/2
    if fb(x1)*fb(x3)<0:
        x2=x3
    elif fb(x1)*fb(x3)>0:
        x1=x3
    else:
        break
print('bisection fb: %.5f' %x3,'\n iterations:',i_bb)

#Newton-Raphson method for fb
x1=0.5; e_a=1
while (e_a>0.0005):
    i_nrb=i_nrb+1
    x2=x1-(fb(x1)/dfb(x1))
    e_a=abs(fb(x1)/dfb(x1))
    x1=x2
print('Newton-Raphson fb: %.5f' %x2,'\n iterations:',i_nrb)
```

Άσκηση 2

Σε αυτή την άσκηση πρέπει να χρησιμοποιήσουμε την μέθοδο $x = g(x)$ και την μέθοδο Newton-Rapshon βρούμε την ρίζα της εξίσωσης:

$$2e^x - 3x^2 = 0 \tag{3}$$

Την Newton-Rapshon θα την εφαρμόσουμε όπως περιγράψαμε στην προηγούμενη άσκηση, σύμφωνα με την αναδρομική σχέση (2).

Για την μέθοδο $x = g(x)$ πρέπει πρώτα να βρούμε την κατάλληλη $g(x)$. Από την (1) αρκετά εύκολα καταλήγουμε στην σχέση $x = \pm\sqrt{(\frac{2}{3}e^x)}$ και διαλέγουμε το (-), έτσι ώστε να καταλήξουμε στην:

$$g(x) = -\sqrt{(\frac{2}{3}e^x)} \tag{4}$$

Ο λόγος που διαλέξαμε το (-) αντί για το (+) είναι το κριτήριο σύγκλισης $g'(x) < 1$. Η παράγωγος της $g(x)$ είναι $g'(x) = -\sqrt{e^x/6}$, για την οποία ισχύει $g'(x) < 1$. Εύκολα μπορούμε να δούμε ότι αν είχαμε επιλέξει το θετικό πρόσημο δεν θα ίσχυε το κριτήριο σύγκλισης.

Έχοντας βρει λοιπόν την σωστή $g(x)$ μπορούμε να υπολογίσουμε την ρίζα μέσω της $x_{i+1} = g(x_i)$. Τιμή εκκίνησης και για τις δύο μεθόδους βάλαμε το $x_1 = 0$. Οι βρόχοι σταματάνε να εκτελούνται όταν το σφάλμα γίνει μικρότερο από $e < 0.0005$. Βρήκαμε:

- Μέθοδος $x = g(x)$: μετά από 8 επαναλήψεις, $\mathbf{x_r = -0.6037}$.
- Μέθοδος Newton-Rapshon: μετά από 5 επαναλήψεις, $\mathbf{x_r = -0.6037}$.

Συμπέρασμα: οι δύο μέθοδοι βρήκαν την ίδια ακριβώς ρίζα αλλά με ταχύτερη σύγκλιση η Newton-Rapshon όπως περιμέναμε.

```
import math as m
import numpy as np
import bigfloat as bf

f= lambda x: 2*np.exp(x)-3*pow(x,2) #x[-1,1]
df= lambda x: 2*np.exp(x)-6*x
g= lambda x: -m.sqrt(2/3*(bf.exp(x,bf.precision(100))))
#used bigfloat because of overflow error

i_g=0; i_nr=0

#x=g(x) method
x1=0; e_a=1
while (e_a>0.0005):
    i_g=i_g+1
    x2=g(x1)
    e_a=abs(x2-x1)
    x1=x2
print('x=g(x) method: x=%.4f' %x2,'\n iterations: ',i_g)

#Newton-Raphson method
x1=0; e_a=1
while (e_a>0.0005):
    i_nr=i_nr+1
    x2=x1-(f(x1)/df(x1))
    e_a=abs(f(x1)/df(x1))
    x1=x2
print('Newton-Raphson method: x=%.4f' %x2,'\n iterations: ',i_nr)
```

Άσκηση 3

Εδώ καλούμαστε να υπολογίσουμε το πολυώνυμο Lagrange για δύο διαφορετικές εξισώσεις. Ο κώδικας που χρησιμοποιήσαμε ορίζει μια συνάρτηση που την ονομάσαμε Lagrange και υπολογίζει το πολυώνυμο Lagrange με τον εξής τρόπο.
Οι συντελεστές του πολυωνύμου Lagrange δίνονται από την σχέση:

$$L_j = \prod_{0 \le m \le k, m \neq k} \frac{x - x_m}{x_j - x_m} \tag{5}$$

και το πολυώνυμο από την:

$$P_n(x) = \sum_{j=0}^{k} y_j L_j \tag{6}$$

Η συνάρτηση δέχεται σαν είσοδο τις τιμές για τα x και $y(x)$ στα οποία θα κάνουμε την παρεμβολή και υπολογίζει πρώτα το κλάσμα της σχέσης (5) και μέσα σε ένα βρόχο πολλαπλασιάζει κάθε φορά τον επόμενο όρο για αυξανόμενο k και μετά σε έναν εξωτερικό βρόχο προσθέτει έναν-έναν τους όρους της σχέσης (6) για αυξανόμενο j. Τέλος η συνάρτηση επιστρέφει το πολυώνυμο Lagrange για τα δεδομένα τα οποία της δώσαμε. Οι εξισώσεις για τις οποίες πρέπει να κάνουμε την παρεμβολή Lagrange είναι οι:

$$y(x) = 1 + sin(\pi x), \quad x = [-1, 0, 1]$$
$$f(x) = 1/(1 + x^2), \quad x \in [-5, 5] \text{ με βήμα } 1$$

Μετά την εκτέλεση του κώδικα βρήκαμε τα πολυώνυμα Py για την εξίσωση $y(x)$ και Pf για την εξίσωση $f(x)$:

$$Py = 1$$

$$Pf = -2 \cdot 10^{-5} x^{10} + 1,27 \cdot 10^{-3} x^8 - 2,44 \cdot 10^{-2} x^6 + 1,97 \cdot 10^{-1} x^4 - 6,74 \cdot 10^{-1} x^2 + 1$$

Τέλος φτιάξαμε τα διαγράμματα για την κάθε εξίσωση ξεχωριστά, μαζί με το πολυώνυμο της, τα οποία δίνονται στο τέλος. Από τα διαγράμματα βλέπουμε ότι η προσέγγιση που κάνουμε δεν είναι και πολύ καλή, ειδικά από τη στιγμή που ξέρουμε τη μορφή των συναρτήσεων. Παρακάτω δίνεται ο κώδικας.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy._lib.six import xrange

#calculate the Lagrange coefficients
def Lagrange(y,x,n):

    L = np.poly1d(0.0)

    for j in xrange(n):
        pt=np.poly1d(y[j])
        for k in xrange(n):
            if k==j:
                continue
            factor= x[j]-x[k]
            pt *= np.poly1d([1.0,-x[k]])/factor
        L += pt
    return L
```

```
19
20
21 y_x = lambda x: 1+ np.sin(np.pi*x)
22 f_z = lambda z: 1/(1+pow(z,2))
23
24 #calculate values for y(x) and f(x) and make lists of the data
25 xi=[]
26 yi=[]
27 for i in range(-1,2):
28     xi.append(i)
29     yi.append(y_x(i))
30 x=np.array(xi) #we make the list into arrays so they work with poly1d
31 y=np.array(yi)
32
33 zi=[]
34 fi=[]
35 for i in range(-5,6):
36     zi.append(i)
37     fi.append(f_z(i))
38 z=np.array(zi)
39 f=np.array(fi)
40
41 #find the polynomials
42 Py=Lagrange(y,x,3)
43 Pf=Lagrange(f,z,11)
44
45 #keep significant terms
46 Py1=np.zeros(3)
47 Pf1=np.zeros(11)
48 for i in range(3):
49     Py1[i]=round(Py.c[i],5)
50 for i in range(11):
51     Pf1[i]=round(Pf.c[i],5)
52
53
54 Pol_y=np.poly1d(Py1)
55 print(Pol_y)
56 Pol_f=np.poly1d(Pf1)
57 print(Pol_f)
58
59
60 #plot the polynomials with the function they approach
61 x1=np.linspace(-1.0,1.0,100)
62 z1=np.linspace(-5.0,5.0,100)
63 plt.figure(1)
64 plt.axhline(y=1, color='r', linestyle='-',label='Py')
65 plt.plot(x1,y_x(x1),label='y = 1 + sin(pi*x)')
66 plt.xlabel('x')
67 plt.ylabel('y(x)')
68 plt.grid(alpha=.4,linestyle='--')
69 plt.legend()
70 plt.show()
71 plt.figure(2)
72 plt.plot(z,Pol_f,label='Pf')
73 plt.plot(z1,f_z(z1),label='f = 1/(1+x^2)')
74 plt.xlabel('x')
75 plt.ylabel('f(x)')
76 plt.grid(alpha=.4,linestyle='--')
77 plt.legend()
78 plt.show()
```

Άσκηση 4

Στην τελευταία άσκηση πρέπει να βρούμε το πολυώνυμο Newton-Gregory για ισαπέχοντα x για δεδομένα τιμές x και $f(x)$ που δίνονται την εκφώνηση. Ο τύπος που δίνει το πολυώνυμο Newton-Gregory είναι ο :

$$P_n(x_0+hs) = f(x_0) + s\Delta f(x_0) + \frac{s(s-1)}{2!}\Delta^2 f(x_0) + ... + \frac{s(s-1)(s-1)...(s-n+1)}{n!}\Delta^n f(x_0) \quad (7)$$

Ο κώδικας για να υπολογίσει την σχέση (7) ορίζει μερικές συναρτήσεις. Μια συνάρτηση υπολογίζει τα γινόμενα στον $s(s-1)(s-1)...(s-n+1)$ και μία άλλη τα παραγοντικά. Για να βρούμε τις διαφορές προς τα εμπρός φτιάχνουμε ένα array δύο διαστάσεων το οποίο σε κάθε στήλη n έχει όλες τις διαφορές n τάξης. Τέλος πολλαπλασιάζουμε και προσθέτουμε όλα τα κομμάτια μεταξύ τους σύμφωνα με τη σχέση (7) και έτσι υπολογίζουμε το πολυώνυμο, το οποίο είναι ίσο με:

$$P_5(x) = -432.5x^5 + 839.17x^4 - 618.54x^3 + 221.71x^2 - 39.54x + 2.97$$

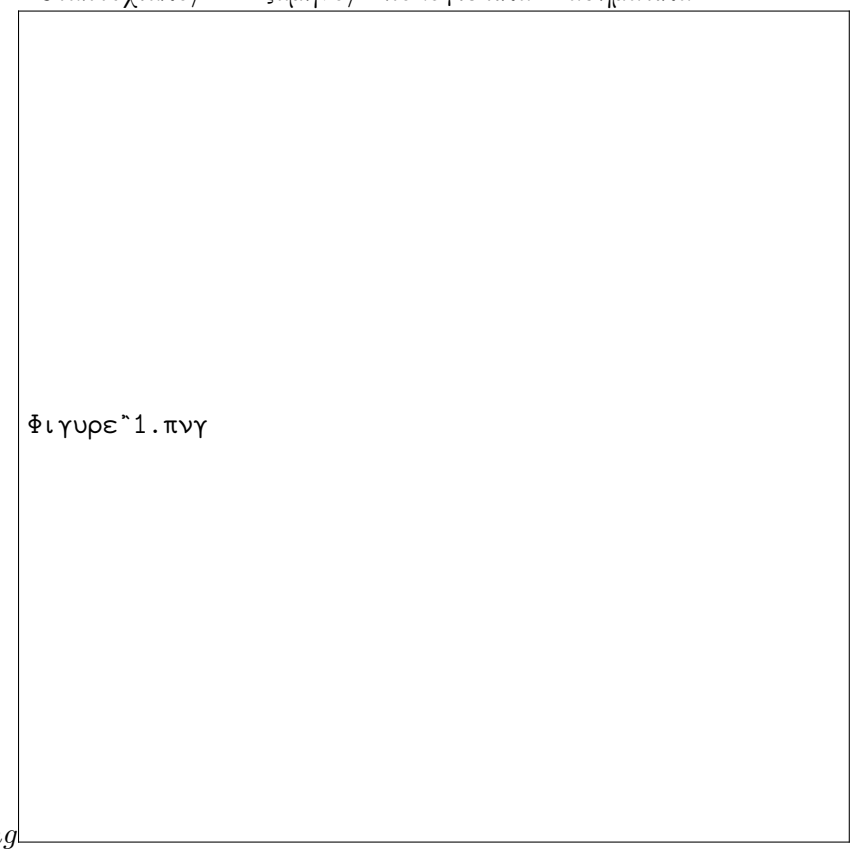
Στον κώδικα χρησιμοποιήσαμε συμβολικές συναρτήσεις για να μπορούμε να κάνουμε πιο εύκολα το διάγραμμα του πολυωνύμου που δίνεται στο τέλος μαζί με τα διαγράμματα της άσκησης 3. Ο κώδικας δίνεται παρακάτω.

```
from sympy import symbols,simplify
from sympy.plotting import plot as symplot


def s_cal(s, n):
    temp=s;
    for i in range(1, n):
        temp= temp*(s - i);
    return temp;

#function for factorial
def fact(n):
    f=1
    for i in range(2, n + 1):
        f *= i
    return f

x=symbols('x')
n=6
x0=0.2
h=0.1
s=(x-x0)/h

#initial values for f(x)
x_v=[0.2,0.3,0.4,0.5,0.6,0.7]
f=[[0 for i in range(n)]
        for j in range(n)]
f[0][0]=0.185
f[1][0]=0.106
f[2][0]=0.093
f[3][0]=0.24
f[4][0]=0.579
f[5][0]=0.561

```

```
#calculate forward differances
for i in range(1, n):
    for j in range(n - i):
        f[j][i] = f[j + 1][i - 1] - f[j][i - 1];

#find the polynomial
P = f[0][0]
for i in range(1,n):
    P += (s_cal(s,i)*f[0][i])/fact(i)
P=simplify(P)
print(P)
symplot(P)
```

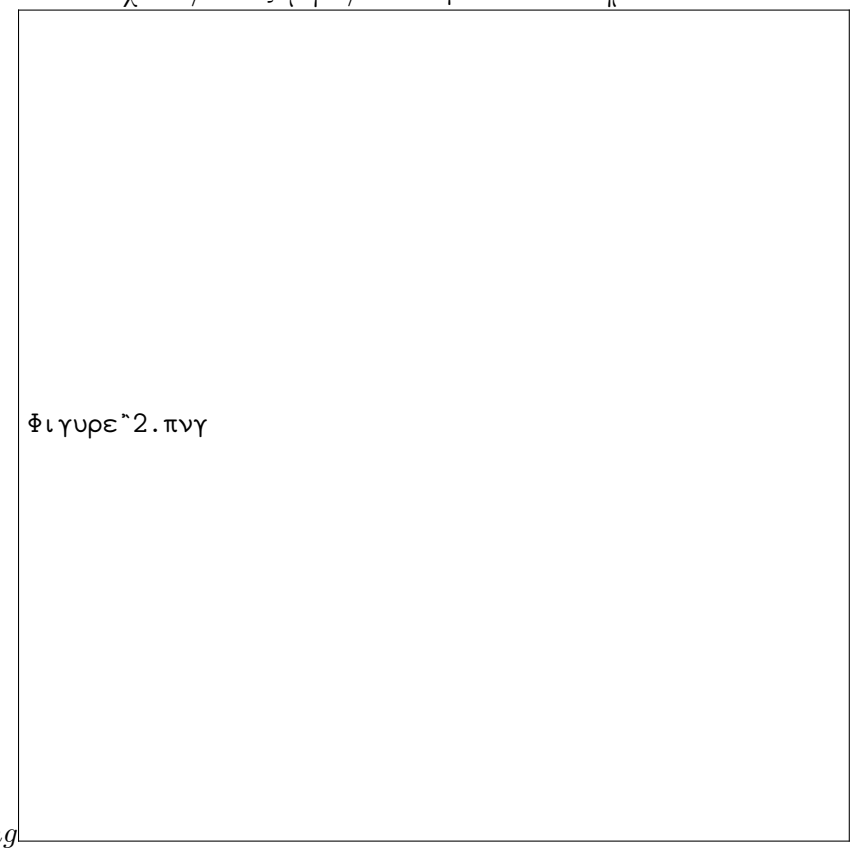
Διαγράμματα

Σχήμα 1: Άσκηση 3: Διάγραμμα για $y(x)$ και Py
Μεταπτυχιακό/Α΄ Εξάμηνο/Υπολογιστικά Μαθηματικά



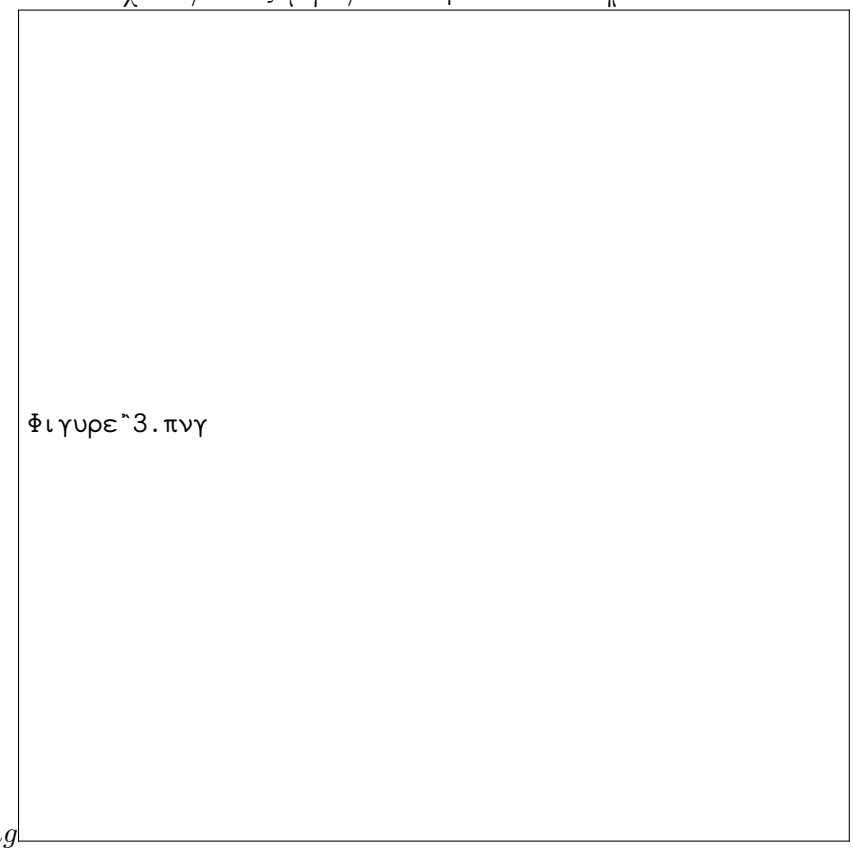
IΦιγυρε$_1$.*png*

Σχήμα 2: Άσκηση 3: Διάγραμμα για $f(x)$ και Pf
Μεταπτυχιακό/Α΄ Εξάμηνο/Υπολογιστικά Μαθηματικά



IΦιγυρε$_2$.*png*

Σχήμα 3: Άσκηση 4: Διάγραμμα για $P_5(x)$



IΦιγυρε3.*png*