Αριθμητικές λύσεις εξισώσεων

Επίλυση συστήματος γραμμικών εξισώσεων

Με την PYTHON θα μπορούσαμε να λύσουμε ένα σύστημα γραμμικών εξισώσεων

Γραμμική εξίσωση είναι μια αλγεβρική εξίσωση στην οποία κάθε όρος είναι είτε σταθερή ποσότητα ή σταθερή ποσότητα πολλαπλασιασμένη με την πρώτη δύναμη μιας μεταβλητής.

Σύστημα γραμμικών εξισώσεων είναι μια συλλογή από τουλάχιστον δύο γραμμικές εξισώσεις που θα πρέπει να θεωρηθούν συλλογικά και όχι μεμονωμένα

Για παράδειγμα, θα μπορούσαμε να έχουμε το ακόλουθο σύστημα γραμμικών εξισώσεων:

$$\begin{cases} 3x_1 - 2x_2 - x_3 = 2 \\ 2x_1 - 2x_2 + 4x_3 = 0 \\ -x_1 + 0.5x_2 - 1.5x_3 = -1 \end{cases}$$

Η παραπάνω εξίσωση μπορεί να γραφεί σε μορφή πινάκων ως: $A\boldsymbol{x} = \boldsymbol{B}$

όπου: $A = \begin{pmatrix} 3 & -2 & -1 \\ 2 & -2 & +4 \\ -1 & 0.5 & -1.5 \end{pmatrix}$ $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ και: $\boldsymbol{B} = \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix}$

Το σύστημα αυτό λύνεται αλγεβρικά με κάποια μέθοδο όπως η μέθοδος Cramer, η μέθοδος της απαλοιφής μεταβλητών, αντικατάστασης κλπ.

Επίλυση συστήματος γραμμικών εξισώσεων

Με την PYTHON θα μπορούσαμε να χρησιμοποιήσουμε μία μέθοδο από την βιβλιοθήκη numpy:

x=numpy.linalg.solve(A,B) Όπου **A** είναι ο πίνακας των όρων που πολ/ζουν τις μεταβλητές και **B** ο πίνακας των σταθερών συντελεστών των εξισώσεων.

Η μέθοδος θα υπολογίσει την ακριβή λύση αρκεί η ορίζουσα του A να μην είναι 0

```
#!/usr/bin/python3
import numpy as np
A = np. array([ [3.0, -2.0. -1.0], [2.0, -2.0, 4.0] , [-1.0, 0.5, -1.5] ] )
b = np.array( [2.0, 0.0, -1.0] )
x = np.linalg.solve(A,B)
print("x = ",x)
bb= np.dot(A,x)
print ("bb = ", bb)
```

Δημιουργία πίνακα από πίνακες

Επαλήθευση του αποτελέσματος

Η συνάρτηση dot() υπολογίζει το αποτέλεσμα του πολλαπλασιασμού γραμμής x στήλης για πράξεις μεταξύ πινάκων (Ένας πίνακας A (n x m) πολ/ζεται με πίνακα B(m x 1) και το αποτέλεσμα είναι ένας πίνακας C (n x 1)

Να σημειωθεί ότι το αποτέλεσμα της επαλήθευσης μπορεί να μη δώσει ακριβώς τους ίδιους τους σταθερούς όρους, λόγω στρογγυλοποίησης.

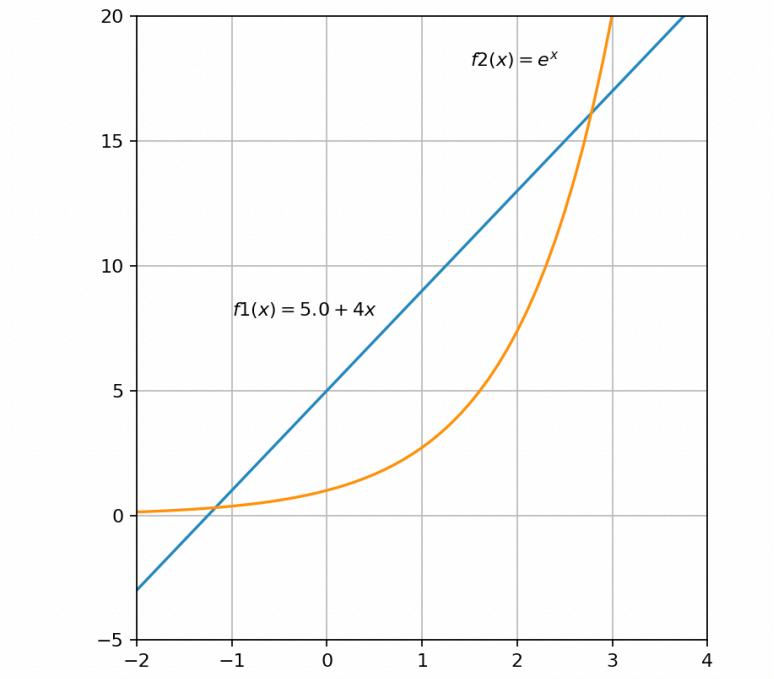
Επίλυση μή γραμμικών εξισώσεων

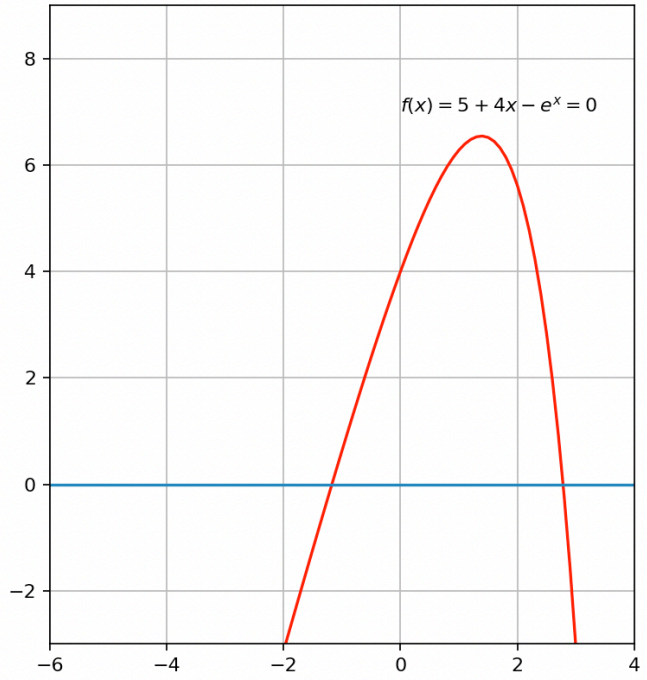
Πολύ συχνά ωστόσο αντιμετωπίζουμε το πρόβλημα της λύσης μιας μη γραμμικής εξίσωσης

Στις περιπτώσεις αυτές μεταφέρουμε το δεξί μέρος της εξίσωσης στο αριστερό και επομένως έχουμε το πρόβλημα της εύρεσης των τιμών της μεταβλητής που μηδενίζουν την $f(x)$ =0

Για παράδειγμα, έστω ότι έχουμε μια εξίσωση της μορφής: $5 + 4x = e^x$

Η εξίσωση αυτή δεν μπορεί να λυθεί αναλυτικά





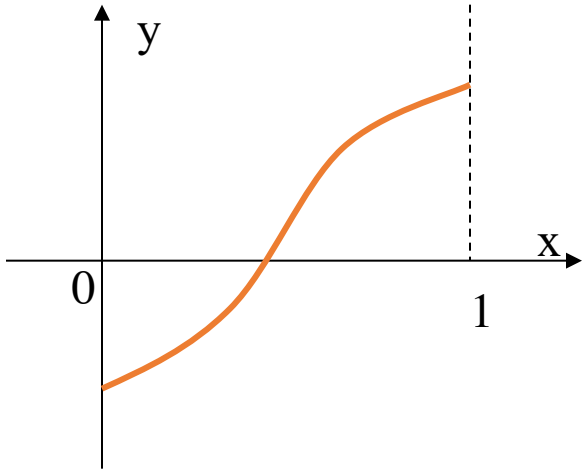
Μπορείτε να βρείτε το py file που κάνει τα δύο γραφήματα στο:
http://www2.ucy.ac.cy/~fotis/phy140/Lectures/lect14_graph.py

Εύρεση ριζών – Bisection Method

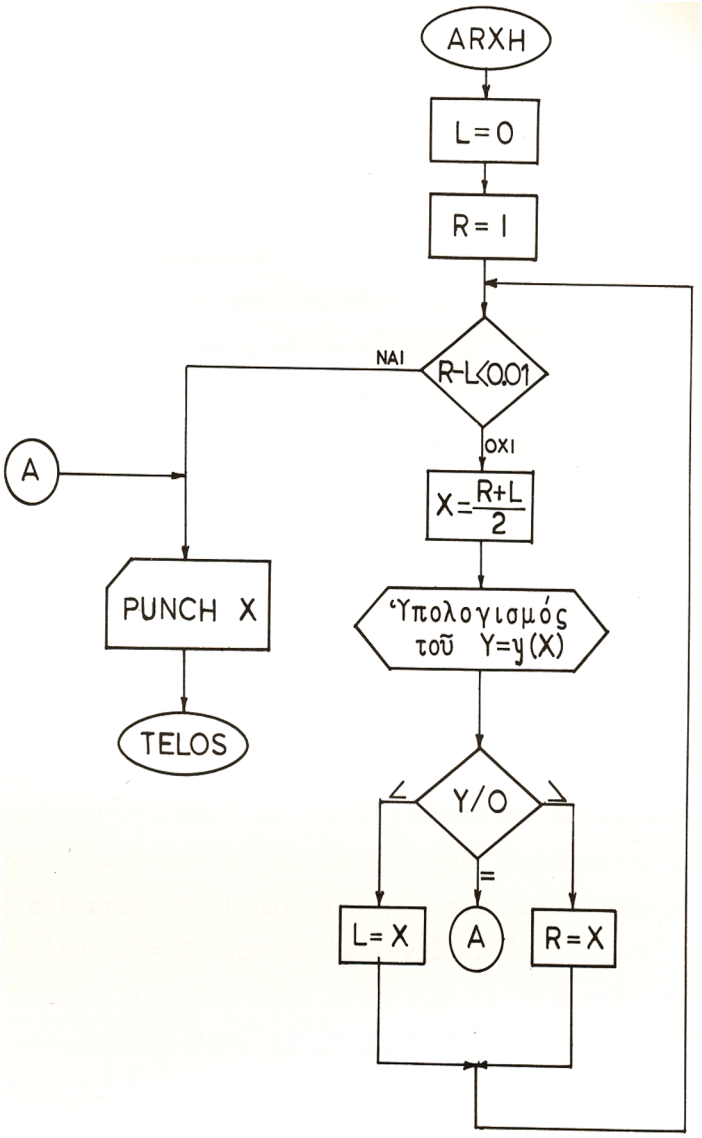
Η γραφική παράσταση του παρακάτω σχήματος μιας μονότονης συναρτήσεως f(x) κόβει τον άξονα των x σε ένα σημείο μεταξύ x=0 και x=1.
Η αναλυτική εξάρτηση του y από το x δεν είναι γνωστή.

Για κάθε x όμως στο διάστημα $0 \le x \le 1$ μπορούμε να βρούμε με τον Η/Υ το αντίστοιχο y με ένα δεδομένο πρόγραμμα.

Θέλουμε το πρόγραμμα που να βρίσκει τη ρίζα x_0 της εξίσωσης $f(x) = 0$ με ακρίβεια 0.01.



Εύρεση ριζών – Bisection Method



Στην αρχή ορίζουμε 2 αριθμούς L=0 και R=1 (όρια)
Οι L και R θα αλλάξουν πολλές τιμές αλλά πάντα

$$L \le x_0 \le R$$

Ρωτάμε αν η διαφορά R-L ≤ 0.01 (προσέγγιση)
Την πρώτη φορά η διαφορά είναι 1.

Προχωρούμε προς τα κάτω και ορίζουμε σα x το μέσο του διαστήματος μεταξύ των σημείων με τετμημένες L και R.

Υπολογίζουμε το y (με το H/Y) που αντιστοιχεί στο x.

Αν το y > 0, θέτουμε R=x και έχουμε ένα νέο διάστημα (L,R) μέσα στο οποίο είναι το x_0.

Αν το y < 0 θέτουμε L=x. Πάλι x_0 είναι στο διάστημα (L,R). Ρωτάμε R-L < 0.01?

Έτσι υποδιπλασιάζουμε το διάστημα (L,R) συνεχώς έως R-L<0.01 οπότε και κρατάμε την τελευταία τιμή του x για τον οποίο y=0.

Bisection program

```
#!/usr/bin/python3
import numpy as np
def  fun(x):
    return 5.0 + 4.0*x – np.exp(x)
#
a = 0.0; b=10.0
eps = 1.0e-15
#
fa = fun(a); fb = fun(b)
#
if  fa*fb > 0:
   print(“wrong interval!!!”, fa, fb)
   exit()
#
iter = 1
while (b-a) > eps:
     c = (a+b)/2.0
     fc = fun(c)
     if fc == 0:
        print (“ x= “, c)
        exit()
     if  fc * fa > 0 :
         a = c
         fa = fc
     else:
         b  = c
         fb =fc
     iter = iter + 1
# the output:
print (‘x = ‘, c)
print(‘accuracy = ,{:.2e}’.format(b-a))
print(”f(“,c,”) = ”,fun(c))
print(iter,’ iterations needed’)
```

Μπορείτε να κατεβάσετε το πρόγραμμα αυτό από:
http://www2.ucy.ac.cy/~fotis/phy140/Lectures/bisection.py

Bisection Method – Σύγκλιση της μεθόδου

Αν το αρχικό διάστημα είναι ε_i το σφάλμα θα είναι $\varepsilon_i/2$.
Μετά από κάθε επανάληψη το αρχικό σφάλμα διαιρείται με 2 οπότε
μετά από N επαναλήψεις το σφάλμα είναι $\varepsilon_f = \varepsilon_i/2^N$ όπου ε_f η επιθυμητή ακρίβεια
Επομένως ο αριθμός των βημάτων για την επίτευξη της επιθυμητής ακρίβειας είναι

$$N = \log\left(\varepsilon_i/\varepsilon_f\right)/\log(2)$$

Εύρεση ριζών – Newton's Method (Newton-Raphson)

Μέθοδος εύρεσης λύσης μη γραμμικής εξίσωσης.

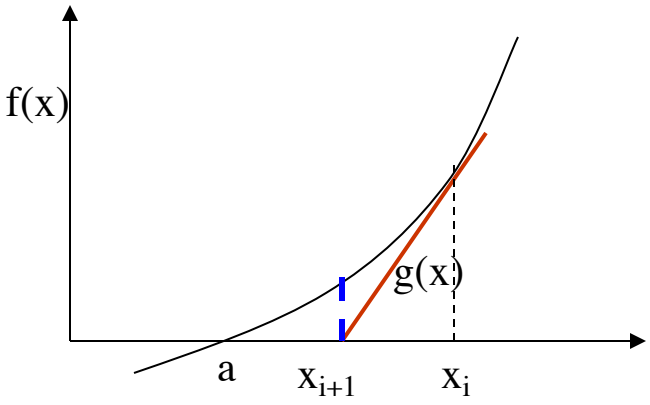
Συγκλίνει αν η αρχική προσεγγιστική τιμή για την ρίζα είναι κοντά στην πραγματική τιμή.

Η σύγκλιση είναι ανάλογη του τετραγώνου του αρχικού σφάλματος $\varepsilon_i = x - x_i$

Γρηγορότερη από τη μέθοδο της διχοτόμησης

Μειονέκτημα ----> Πρέπει να υπολογισθεί η παράγωγος της μη γραμμικής συνάρτησης g(x)=f'(x)

Η μέθοδος στηρίζεται στην χρησιμοποίηση των παραγώγων f'(x) της συνάρτησης f(x) για ταχύτερη σύγκλιση στην εύρεση των ριζών της f(x) =0



Η βασική ιδέα: Μια συνεχής παραγωγίσιμη, συνάρτηση f(x) που έχει συνεχή δεύτερη παράγωγο, μπορεί να αναπτυχθεί κατά Taylor ως προς ένα σημείο x_n που είναι κοντά στη ρίζα.

Έστω ότι η πραγματική ρίζα αυτή είναι α

$$f(a) = f(x_n) + (a - x_n) f'(x_n) + (a - x_n)^2 \frac{f''(x_n)}{2!} + \cdots$$

Εφόσον x=α είναι ρίζα της f, τότε f(α) = 0 οπότε από τους 2 πρώτους όρους του αναπτύγματος έχουμε:

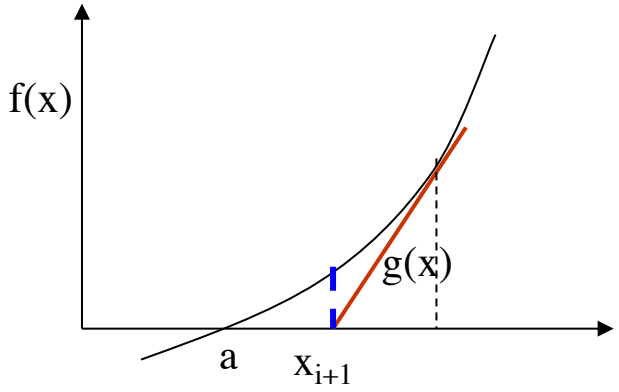
$$0 = f(a) = f(x_n) + (a - x_n) f'(x_n) \quad \Rightarrow a = x_n - \frac{f(x_n)}{f'(x_n)}$$

Εύρεση ριζών – Newton's Method

Επομένως χρειαζόμαστε f(x) και f'(x) για την μέθοδο.

Κάθε επανάληψη της μεθόδου βρίσκει: $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$

Σφάλμα: $x_{k+1} - a = x_k - a - \dfrac{f(x_k)}{f'(x_k)} \Rightarrow \varepsilon_{k+1} = \varepsilon_k + \dfrac{f(x_k)}{f'(x_k)}$ (A)



Χρησιμοποιώντας τους 3 πρώτους όρους από Taylor:

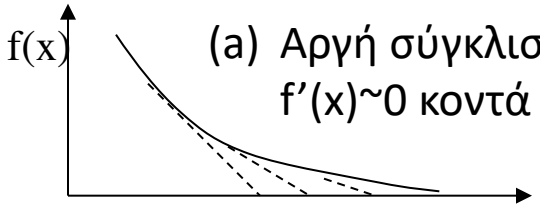
$$0 = f(a) = f(x_k) + (a - x_k) f'(x_k) + (a - x_k)^2 \frac{f''(x_k)}{2}$$

$$\Rightarrow f(x_k) = -(a - x_k) f'(x_k) - (a - x_k)^2 \frac{f''(x_k)}{2}$$

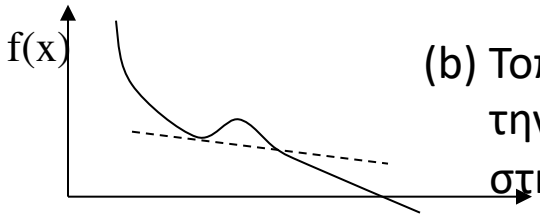
$$\Rightarrow f(x_k) = -\varepsilon_k f'(x_k) - \varepsilon_k^2 \frac{f''(x_k)}{2}$$ Αντικαθιστώντας στην (A)

Ρυθμός σύγκλισης: $\varepsilon_{k+1} = -\dfrac{\varepsilon_k^2 f''(x_k)}{2 f'(x_k)}$

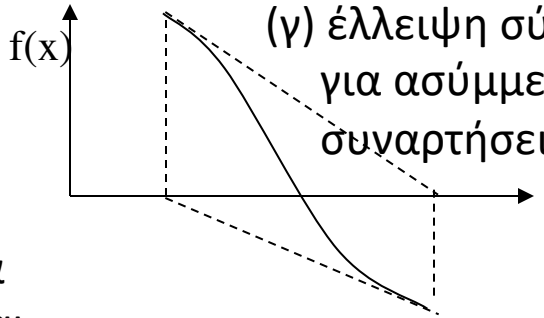
Προβλήματα της μεθόδου:



(a) Αργή σύγκλιση προς τη ρίζα όταν f'(x)~0 κοντά στη ρίζα



(b) Τοπικό ακρότατο στέλνει μακριά την τιμή x_{k+1} που χρησιμοποιείται στην επόμενη επανάληψη



(γ) έλλειψη σύγκλισης για ασύμμετρες συναρτήσεις

$f(x + a) = -f(x - a)$

Μέθοδος Newton's - πρόγραμμα

```
#!/usr/bin/python3

import numpy as np
def func(x):
        return 5.0 + 4.0*x – np.exp(x)
#
def deriv(x):
        return 4.0 - np.exp(x)
#
x0 = float(input(”give the initial guess ”))
eps = float(input(“give the precision “))
istepmax = int(input(“max iterations ”))
error = func(x0) / deriv(x0)
while np.abs(error) > eps :
     x0 = x0 – error
     error = func(x0) / deriv(x0)
     istep = istep + 1
     if istep > istepmax :
        flag = 1
        break
if flag !=1:
      print(“root is “, x0)
else:
      print(“not convergence”)
```

Μπορείτε να κατεβάσετε το πρόγραμμα αυτό από:
http://www2.ucy.ac.cy/~fotis/phy140/Lectures/newton.py