Προσεγγιστικές μέθοδοι εύρεσης ριζών μιας εξίσωσης

Οι μέθοδοι:

Newton – Raphson

secant

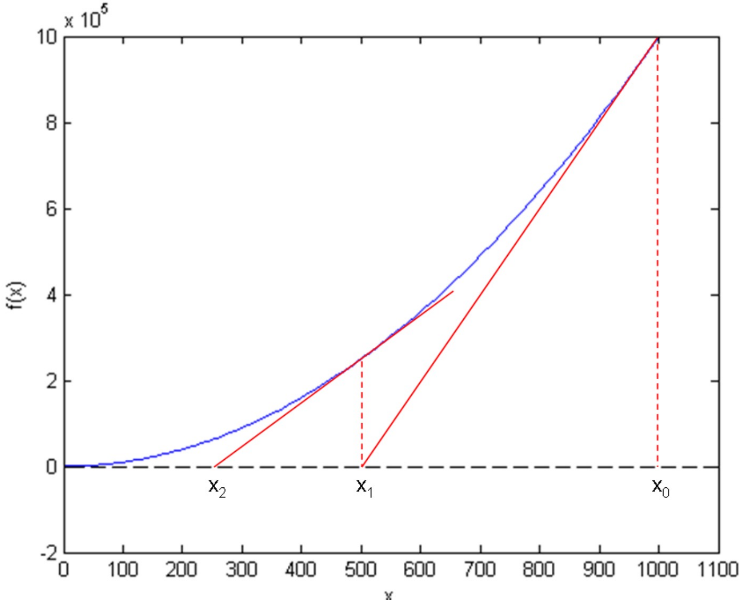
bisection

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

Αποτελεί την ταχύτερη μέθοδο εύρεσης λύσης μη γραμμικής εξίσωσης, ωστόσο δεν εγγυάται πάντοτε η εύρεση λύσης

Συγκλίνει πολύ γρήγορα αν η αρχική προσεγγιστική τιμή για τη ρίζα είναι κοντά στη πραγματική τιμή.

Η ιδέα των προσεγγιστικών μεθόδων εύρεσης της λύσης μιας μη γραμμικής εξίσωσης στηρίζεται στη δημιουργία μιας σειράς γραμμικών εξισώσεων (τις οποίες ξέρουμε να λύσουμε) ελπίζοντας ότι οι λύσεις των ενδιάμεσων αυτών γραμμικών εξισώσεων θα μας φέρει πιο κοντά στην τελική λύση.



Έστω ότι έχουμε την εξίσωση $f(x) = x^2 - 9$, την οποία προσπαθούμε να λύσουμε στο διάστημα [0,1000] βρίσκοντας επαναληπτικά το σημείο τομής της εφαπτόμενης της καμπύλης με τον x-άξονα. Από το σημείο τομής βρίσκουμε την τιμή της συνάρτησης και την εφαπτόμενη της καμπύλης της συνάρτησης και κατόπιν και πάλι το σημείο τομής με τον x-άξονα. Η διαδικασία συνεχίζεται έως ότου στο σημείο τομής $x_i,\ f(x_i) = 0$.

Η παραπάνω διαδικασία προϋποθέτει ότι δίνουμε μια αρχική τιμή x_0, για να μπορέσει να ξεκινήσει η ακολουθία

Η βασική ιδέα είναι να προσεγγίσουμε την αρχική συνάρτηση σαν μια ευθεία. Υπάρχουν άπειρες επιλογές για το πως να προσεγγίσουμε την $f(x)$ με μια ευθεία

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

Ωστόσο πως βρίσκουμε την εφαπτομένη της καμπύλης της συνάρτησης στο σημείο x_0;

Η εφαπτόμενη είναι γραμμική και έχει δύο χαρακτηριστικά:

- Η κλίση της είναι ίση με την παράγωγο της συνάρτησης, $f'(x_0)$, στο σημείο x_0
- Η εφαπτόμενη εφάπτεται της καμπύλης της συνάρτησης στο σημείο $(x_0, f(x_0))$

Η εξίσωση της εφαπτομένης θα είναι επομένως: $f_{\varepsilon\varphi}(x) = ax + b$

αλλά με βάση τα 2 παραπάνω χαρακτηριστικά: $\alpha = f'(x_0)$ και $f_{\varepsilon\varphi}(x_0) = f(x_0)$

Επομένως θα έχουμε:

$$f_{\varepsilon\varphi}(x) = f'(x_0)x + b$$

$$f_{\varepsilon\varphi}(x_0) = f(x_0) \Rightarrow f'(x_0)x_0 + b = f(x_0) \Rightarrow b = f(x_0) - f'(x_0)x_0$$

Για να καταλήξουμε ότι: $f_{\varepsilon\varphi}(x) = f(x_0) + f'(x_0)(x - x_0)$

Το σημείο κλειδί της μεθόδου Newton είναι να βρεθεί σε ποιο σημείο η εφαπτομένη τέμνει τον x-άξονα. Στο σημείο όμως αυτό, έστω x_1 θα έχουμε $f_{\varepsilon\varphi}(x_1) = 0$.

Επομένως: $0 = f(x_0) + f'(x_0)(x_1 - x_0) \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

Αν ξεκινήσουμε την εξίσωση $f(x) = x^2 - 9$ με $x_0 = 1000$, βρίσκουμε ότι $x_1 \approx 500$

Συνεχίζοντας τη διαδικασία, βρίσκουμε ότι: $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \approx 250$

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

Το γενικό σχήμα της μεθόδου είναι ότι : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ όπου *n*-= 0,1, 2, ...,

Η διαδικασία επαναλαμβάνετα έως ότου $f(x_n)$ είναι αρκετά κοντά στο 0.

Στην πραγματικότητα ελέγχουμε αν $|f(x_n)|$< ε όπου ε ένας πολύ μικρός αριθμός που καθορίζει την ακρίβεια της ρίζας που θα βρούμε.

Στο προηγούμενο παράδειγμα βρήκαμε ότι με δύο βήματα προσεγγίσαμε τη λύση από 1000 σε 250. Είναι επομένως ενδιαφέρον να δούμε πόσο γρήγορα θα βρούμε τη λύση x=3

Αρχικά γράφουμε ένα απλό πρόγραμμα που εφαρμόζει την μέθοδο Newton-Raphson

```
def naive_Newton(f, dfdx, x, eps):
     while abs(f(x)) > eps:
          x = x - f(x)/dfdx(x)
     return x
```

```
def f(x):
    return x*x - 9
def dfdx(x):
    return 2*x
print(naive_Newton(f,dfdx, 1000, 0.001)
```

`f, dfdx` είναι η συνάρτηση και η παράγωγος της συνάρτησης

Το όρισμα `x` είναι η αρχική τιμή x_0 που αναφέραμε προηγουμένως

`eps` είναι το ε για τερματισμό της διαδικασίας

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

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

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 μέθοδος μπορεί να εφαρμοστεί αν έχουμε μια συνεχή παραγωγίσιμη συνάρτηση $f(x)$, που έχει συνεχή δεύτερη παράγωγο, και μπορεί να αναπτυχθεί κατά Taylor ως προς ένα σημείο x_n που είναι κοντά στη ρίζα.

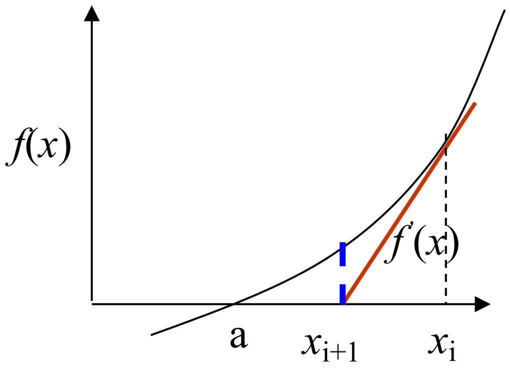
Έστω ότι η πραγματική ρίζα αυτή είναι x_{n+1}:

$$f(x_{n+1}) = f(x_n) + (x_{n+1} - x_n)f'(x_n) + (x_{n+1} - x_n)^2 \frac{f''(x_n)}{2!} + (x_{n+1} - x_n)^3 \frac{f'''(x_n)}{3!} + \cdots$$

Εφόσον $x = x_{n+1}$ είναι ρίζα της $f(x)$, τότε $f(x_{n+1}) = 0$, οπότε κρατώντας τους 2 πρώτους όρους του αναπτύγματος θα έχουμε (προσέγγιση της συνάρτησης γραμμικά):

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

Σφάλμα της μεθόδου Newton-Raphson



Αν α είναι η πραγματική ρίζα τότε το σφάλμα της μεθόδου είναι:

$$x_{k+1} - \alpha = x_k - \alpha - \frac{f(x_k)}{f'(x_k)} \Rightarrow \varepsilon_{k+1} = \varepsilon_k + \frac{f(x_k)}{f'(x_k)} \qquad \textbf{(A)}$$

Χρησιμοποιώντας τους 3 πρώτους όρους από Taylor:

$$f(x_{n+1}) = f(x_n) + (x_{n+1} - x_n)f'(x_n) + (x_{n+1} - x_n)^2 \frac{f''(x_n)}{2!}$$

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

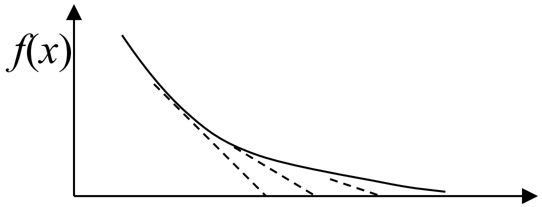
$$\Rightarrow f(x_k) = -(a - x_k)f'(x_k) - (a - x_k)^2 \frac{f''(x_k)}{2!} \Rightarrow f(x_k) = -\varepsilon_k f'(x_k) - \varepsilon_\kappa^2 \frac{f''(x_k)}{2!}$$

Αντικαθιστούμε την τελευταία στην (A):

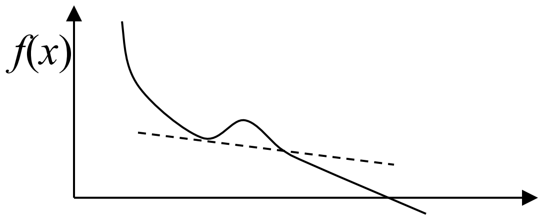
Ρυθμός σύγκλισης

$$\varepsilon_{k+1} = \varepsilon_k + \frac{\varepsilon_k f'(x_k) - \varepsilon_\kappa^2 \frac{f''(x_k)}{2!}}{f'(x_k)} \Rightarrow \varepsilon_{k+1} = \varepsilon_k - \varepsilon_k - \varepsilon_\kappa^2 \frac{f''(x_k)}{2f'(x_k)} \quad \Rightarrow \varepsilon_{k+1} = -\varepsilon_\kappa^2 \frac{f''(x_k)}{2f'(x_k)}$$

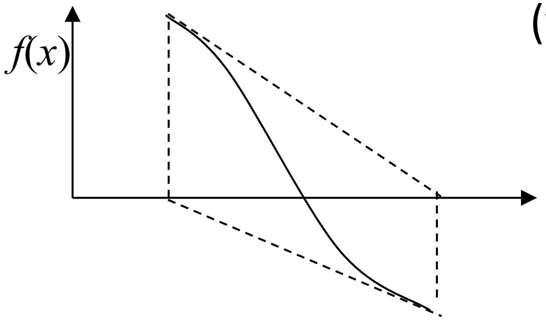
Προβλήματα της μεθόδου Newton-Rapshon



(α) Αργή σύγκλιση προς τη ρίζα όταν $f'(x) \sim 0$ κοντά στη ρίζα



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



(γ) έλλειψη σύγκλισης για ασύμμετρες συναρτήσεις

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

Βελτιστοποίηση της μεθόδου Newton

Θα πρέπει να βελτιώσουμε το πρόγραμμα εύρεσης της λύσης εξίσωσης με την μέθοδο Newton ώστε να αποφύγουμε:

- Διαίρεση με 0 στον υπολογισμό της παραγώγου της συνάρτησης
- Καθορισμό όριου στον αριθμό των επαναλήψεων για αποφυγή μη σύγκλισης
- Αποφυγή αχρείαστων υπολογισμών της συνάρτησης

```python
def Newton(f, dfdx, x, eps):
   f_value = f(x)
   iteration_counter = 0
   while abs(f_value) > eps and iteration_counter < 100:
      try:
        x = x - f_value/dfdx(x)
      except:
        print("Error! - derivative zero for x = ", x)
        exit(1)       # Abort with error
      f_value = f(x)
      iteration_counter += 1
   #Here, either a solution is found, or too many iterations
   if abs(f_value) > eps:
      iteration_counter = -1
   return x, iteration_counter
```

Ολοκλήρωση του προγράμματος για τη μέθοδο Newton

```
def f(x):
   return x**2 - 9
def dfdx(x):
   return 2*x

solution, no_iterations = Newton(f, dfdx, x=1000, eps=1.0e-6)
if no_iterations > 0: # Solution found
   print("Number of function calls: %d" % (1 + 2*no_iterations))
   print("A solution is: %f" % (solution))
else:
   print "Solution not found!"
```

Μπορείτε να κατεβάσετε το πρόγραμμα αυτό από:
https://ptohos.github.io/phy140/Lectures/newton.py

Μέθοδος Newton – Raphson χρησιμοποιώντας sympy

Όπως έχουμε δει, η μέθοδος Newton απαιτεί τον υπολογισμό της παραγώγου της συνάρτησης, κάτι που δεν είναι πάντοτε εύκολο.

Θα μπορούσαμε όμως να συνδυάσουμε τη μέθοδο Newton με το πακέτο συμβολικών υπολογισμών, sumpy, το οποίο μπορούμε να χρησιμοποιήσουμε για να ορίσουμε την συνάρτηση *dfdx* που χρησιμοποιείται στην μέθοδο Newton-Raphson.

```
from sympy import *
x = Symbol("x")              # orismos toy x ws mathematikou symbol
f_expr = x**2 - 9             # symbolic ekfrasi gia f(x)
dfdx_expr = diff(f_expr, x) # ypologismos tis f'(x) symbolically
# Metatropi f_expr kai dfdx_expr se Python functions
f = lambdify(x, f_expr)       # x einai to orisma stin f
# symbolic ekfrasi gia ypologismo
dfdx = lambdify(x, dfdx_expr)
print(dfdx(5)) # will print 10
```

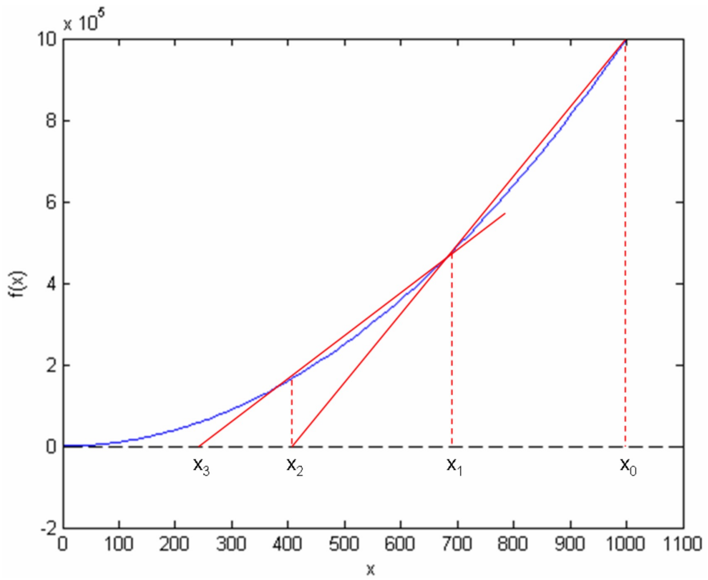
Η δομή ***lambdify*** χρησιμοποιείται για την μετατροπή μιας έκφρασης με σύμβολα, όπως είναι η *dfdx_expr*, σε κανονική συνάρτηση της Python με x ως όρισμα.

Για περισσότερα από 1 όρισμα στην symbolic έκφραση, θα πρέπει να τα περάσουμε ως λίστα [x,y,z]. Το ίδιο θα μπορούσαμε να κάνουμε και για 1 μόνο όρισμα και να το περάσουμε ως [x] και στην παραπάνω περίπτωση θα είχαμε: f=lambdify([x],f_expr)

Μέθοδος της Χορδής – Secant method

Όταν η εύρεση της παραγώγου της συνάρτησης είναι προβληματική ή ο υπολογισμός της συνάρτησης απαιτεί αρκετά βήματα και είναι χρονοβόρος, χρησιμοποιούμε μια παραλλαγή της μεθόδου του Newton – **την μέθοδο Secant**.

Αντί να χρησιμοποιήσουμε τις εξισώσεις της εφαπτομένης της καμπύλης της συνάρτησης χρησιμοποιούμε εξισώσεις χορδών, ευθυγράμμων τμημάτων που ενώνουν δύο σημεία της καμπύλης της συνάρτησης.



Η ιδέα είναι ίδια με αυτή της μεθόδου Newton, αλλά στην προκειμένη περίπτωση προσεγγίζουμε την παράγωγο της συνάρτησης σαν πεπερασμένη διαφορά μεταξύ δύο σημείων, ουσιαστικά μια χορδή.

Επομένως προσεγγίζουμε την παράγωγο με την κλίση της ευθείας που περνά από τα δύο πιο πρόσφατα σημεία της προσέγγισης της λύσης x_nκαι x_{n-1}.

Η κλίση θα είναι: $\dfrac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$

Εισάγουμε στη θέση της παραγώγου στη μέθοδο Newton, την παραπάνω προσέγγιση:

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

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

Η μέθοδος χρειάζεται δύο αρχικά σημεία x_0 και x_1

Η μέθοδος Secant

```
def secant(f, x0, x1, eps):
   f_x0 = f(x0)
   f_x1 = f(x1)
   iteration_counter = 0
   while abs(f_x1) > eps and iteration_counter < 100:
      try:
        denominator =(f_x1 - f_x0)/(x1 - x0)
        x = x1 - (f_x1)/denominator
      except:
        print("Error! - denominator zero for x = ", x)
        exit(1) # Abort with error
      x0 = x1
      x1 = x
      f_x0 = f_x1
      f_x1 = f(x1)
      iteration_counter += 1
   # Here, either a solution is found, or too many iterations
   if abs(f_x1) > eps:
      iteration_counter = -1
   return x, iteration_counter
def f(x):
   return x**2 - 9
x0 = 1000; x1 = x0 - 1
solution, no_iterations = secant(f, x0, x1, eps=1.0e-6)
if no_iterations > 0: # Solution found
   print("Number of function calls: %d" % (2 + no_iterations))
   print("A solution is: %f" % (solution))
else:
   print("Solution not found!")
```

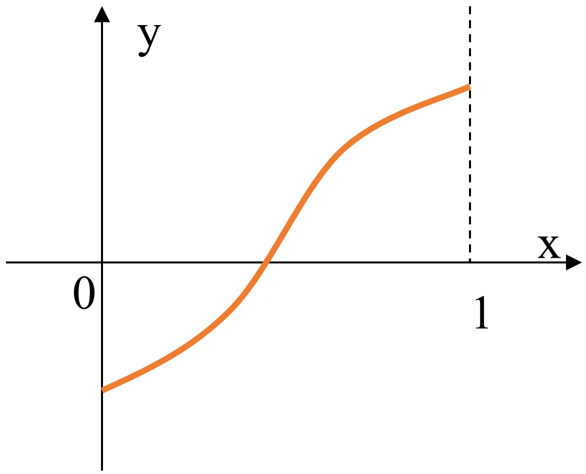
Μπορείτε να κατεβάσετε το πρόγραμμα αυτό από:
https://ptohos.github.io/phy140/Lectures/secant.py

Εύρεση ριζών – Bisection Method

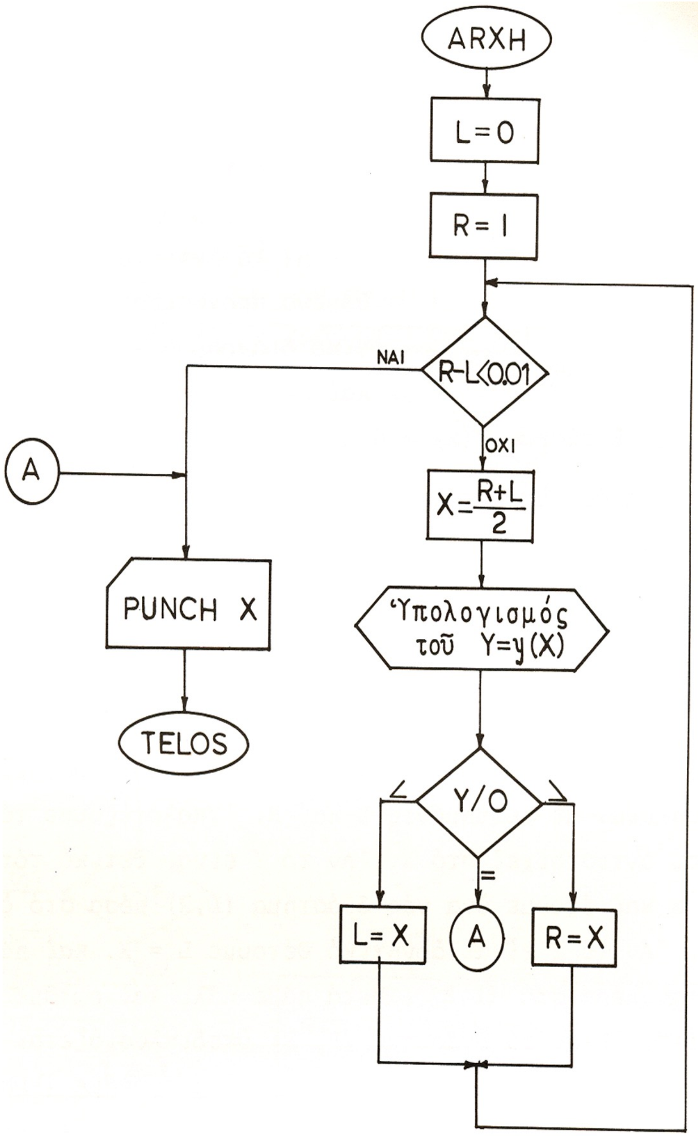
Η γραφική παράσταση του παρακάτω σχήματος μιας μονότονης συναρτήσεως f(x) κόβει τον άξονα των x σε ένα σημείο μεταξύ x=0 και x=1.
Η αναλυτική εξάρτηση του y από το x δεν είναι γνωστή.

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

$$L \leq x_0 \leq R$$

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 Η/Υ) 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

```
def bisection(f, x_L, x_R, eps, return_x_list=False):
    f_L = f(x_L)
    if f_L*f(x_R) > 0:
       print("Error! Function does not have opposite \
              signs at interval endpoints!")
       exit(1)
    x_M =(x_L + x_R)/2.0
    f_M = f(x_M)
    iteration_counter = 1
    if return_x_list:
      x_list = []
    while abs(f_M) > eps:
      if f_L*f_M > 0: # i.e. same sign
         x_L = x_M
         f_L = f_M
      else:
         x_R = x_M
      x_M = (x_L + x_R)/2
      f_M = f(x_M)
      iteration_counter += 1
      if return_x_list:
        x_list.append(x_M)
    if return_x_list:
      return x_list, iteration_counter
    else:
      return x_M, iteration_counter
def f(x):
   return x**2 - 9
a = 0; b = 1000
solution, no_iterations = bisection(f, a, b, eps=1.0e-6)
print("Number of function calls: %d" % (2 + no_iterations))
print("A solution is: %f" % (solution))
```

Μπορείτε να κατεβάσετε το πρόγραμμα αυτό από:
https://ptohos.github.io/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

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