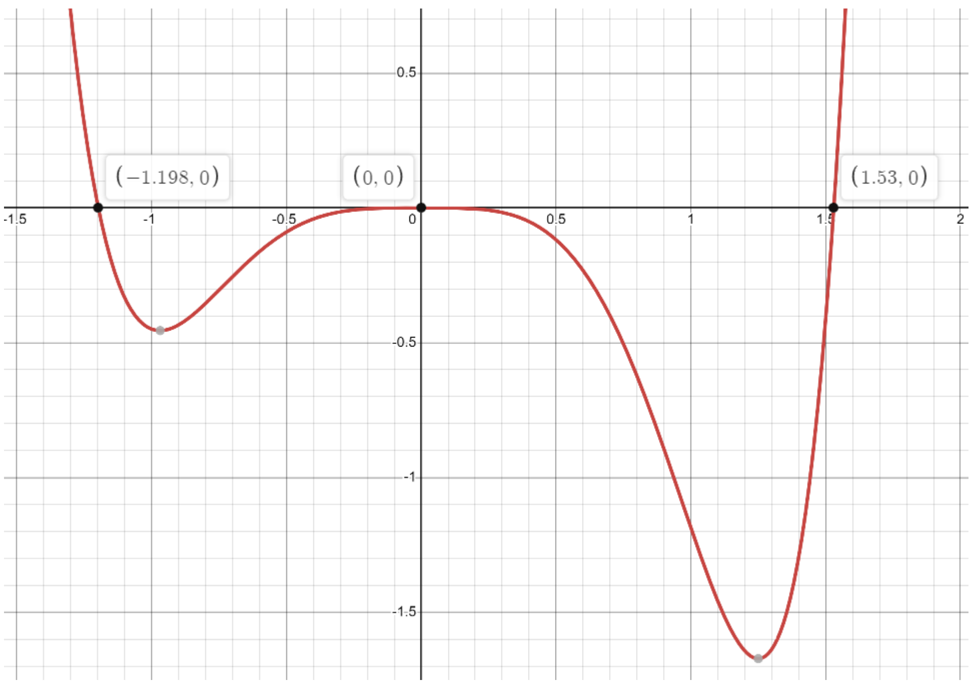
1η υποχρεωτική εργασία ΑΡΑΝ

Καρυοφυλλιά Πριάκου

December 29, 2022

1 Bisection Method



Με την βοήθεια του γραφήματος οι ρίζες μου με ακρίβεια 5 δεκαδικών είναι οι $x_0 = -1.19762, x_1 = 0, x_2 = 1.53013$
Μεγαλύτερη ακρίβεια: $\chi_0 = -1.19762372, \chi_1 = 0, \chi_2 = 1.5301335$

```python
def f(x):
    result = math.e**((math.sin(x))**3) + (x**6) - (2*(x)**4) - (x**3) - 1
    return result
print(f(-2),' f(-2)')
print(f(2),' f(2)')
print(f(1),' f(1)')
```

```
39.471504353021075  f(-2)
25.12087119364365  f(2)
-1.1854758870945021  f(1)
```

- Μεταφέρω την συνάρτηση $f(x) = e^{\sin x^3} + x^6 - 2x^4 - x^3 - 1$ στο notebook σε μορφή function. Παρατηρώ ότι f(2) και f(-2) ομόσημα θετικά, άρα αναζητώ $x\epsilon[-2,2]$ τέτοιο ώστε $f(x) < 0$ για να σπάσω το διάστημα σε 2 υποδιαστήματα και να αναζητήσω ρίζες στο καθένα απαυτά.Με δοκιμή βρίσκω οτι για x=1 ισχύει $f(1) < 0$. Στο 2ο διάστημα χάνω την ακρίβεια του 5ου δεκαδικού της ρίζας μου.

> **Bisection Method**
>
> Given initial interval $[a, b]$ such that $f(a)f(b) < 0$
> **while** $(b - a)/2 >$ TOL
> $c = (a + b)/2$
>   **if** $f(c) = 0$, **stop**, **end**
>   **if** $f(a)f(c) < 0$
> $b = c$
>   **else**
> $a = c$
>   **end**
> **end**
> The final interval $[a, b]$ contains a root.
> The approximate root is $(a + b)/2$.

```
In [64]: e = 1/2 * 10**-5

def bisection_method(a,b,iterat):


    if  f(a)*f(b)<0:
        m = (a + b ) / 2
        if (np.abs(f(m)) < e):
            m= m + e
            return m, iterat

        if f(a)*f(m)<0:
            iterat=iterat+1
            return bisection_method(a,m,iterat)
        else:
            iterat=iterat+1
            return bisection_method(m,b,iterat)
```

```
In [65]: bisection_method(-2,1,0)

Out[65]: (-1.1976182528686523, 19)

In [66]: bisection_method(1,2,0)

Out[66]: (1.5301382473754883, 19)
```

Ο **αριθμός επαναλήψεων** μέχρι την σύγκλιση είναι **19** και για τα 2 διαστήματα που εφάρμοσα την μέθοδο.

2 Newton-Raphson Method

Μέθοδος εύρεσης ριζών με τετραγωνική σύγκληση.Αν η μέθοδος ξεκινήσει μακριά από την επιθυμητή λύση υπάρχει πιθανότητα να μην συγκλίνει. Με δεδομένη την συνάρτηση f(x) και την παράγωγό της f'(x), ξεκινώντας με ένα τυχαίο x_0 μία καλύτερη προσέγγιση x_1 δίνεται από την σχέση:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

. Η γενική αναδρομική σχέση της μεθόδου του Νεύτωνα είναι:

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

όπου x_{n+1} η προσεγγιστική τιμή της ρίζας της συνάρτησης f(x) μετά από n+1 επαναλήψεις.

```
def derf(x):
    result = 3*math.e**((math.sin(x))**3)*math.cos(x)*(math.sin(x))**2 + 6*x**5 -8*x**3 - 3*x**2
    return result
```

```
def newton_raphson_method(x):
    iterat=0
    while (np.abs(f(x)) > e):
        iterat=iterat+1
        x = x - f(x)/derf(x)
    return x, iterat
```

```
newton_raphson_method(2)
```

```
(1.530133508276014, 6)
```

```
newton_raphson_method(-2)
```

```
(-1.1976237963358904, 7)
```

```
newton_raphson_method(0)
```

```
(0, 0)
```

Ο **αριθμός επαναλήψεων** μέχρι την σύγκλιση είναι **6** για x=2, **7** για x=-2 και **0** για x=0.
Για να βρώ και τις 3 ρίζες της συνάρτησης, πρέπει να τρέξω τον newton raphson για τα δύο άκρα του διαστήματος [-2,2] και το 0 ξεχωριστά. Επιτυγχάνεται η επιθυμητή ακρίβεια 5 δεκαδικών.

3 Secant Method

Η μέθοδος τέμνουσας είναι ένας αλγόριθμος εύρεσης ρίζας που χρησιμοποιεί μία διαδοχή ριζών τμηματκών γραμμών για να προσεγγίσει την ρίζα. Η επαναληπτική σχέση που χρησιμοποιεί είναι:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}$$

In [50]:
```
def secant_method(a,b):
    iterat=0
    while (np.abs(f(a)- f(b)) > e):
        iterat=iterat+1
        temp=a
        a = a - (a-b) * f(a)/(f(a)-f(b))
        b= temp
    return a,iterat
```

In [51]:
```
secant_method(-2,2)
```

Out[51]:
```
(-1.1976237221339254, 14)
```

In [52]:
```
secant_method(-2,0)
```

Out[52]:
```
(0.0, 2)
```

Ο **αριθμός επαναλήψεων** μέχρι την σύγκλιση είναι **14** για a=-2 και b=2 , και **2** για x=0.
Μπόρεσα να βρώ 2 ρίζες. με επιτυχής ακρίβεια.

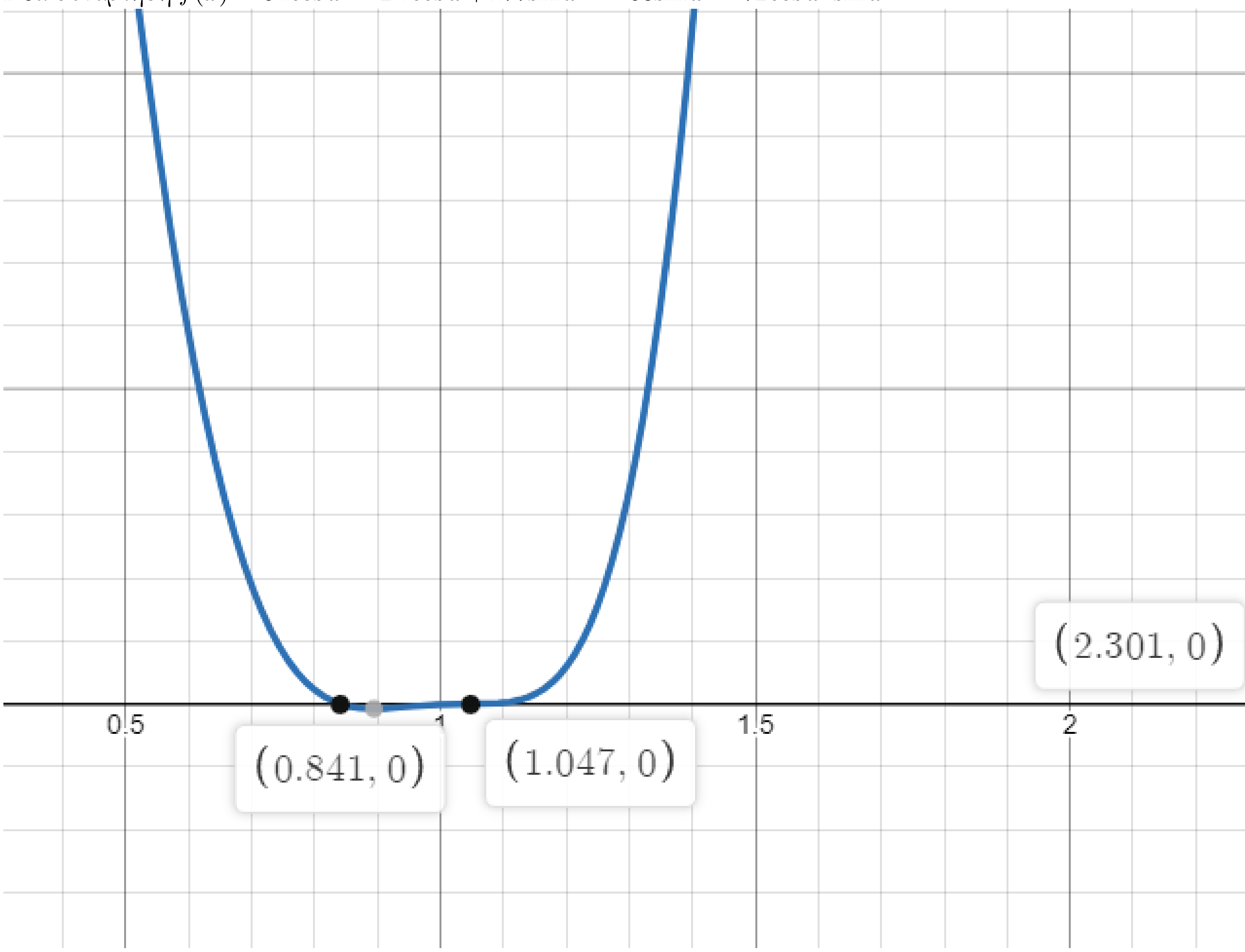
Παρατηρώ οτι γρηγορότερα συγκλίνει ο αλγόριθμος newton-raphson. Αυτό είναι κάτι που περιμέναμε καθώς γνωρίζουμε οτι η μέθοδος*συγκλίνει τετραγωνικά*. Οι περιπτώσεις για τις οποίες δεν συγκλίνει τετραγωνικά είναι:

1. **Zero derivative**: Αν η πρώτη παράγωγος είναι 0 στην ρίζα, τότε η σύγκλιση δεν είναι τετραγωνική. Πχ $f(x) = x^2$ όπου $f'(x) = 2\chi$ άρα $x - \frac{f(x)}{f'(x)} = \frac{x}{2}$. Στην περίπτωση μας $f'(x) = 3\mathrm{e}^{\sin^3(x)}\cos(x)\sin^2(x) + x^2 \cdot \left(6x^3 - 8x - 3\right)$ όπου f'(0) != 0

2. **No second derivative**: Αν δεν υπάρχει 2η παράγωγος στην ρίζα, τότε πάλι δεν συγκλίνει τετραγωνικά.

4 Modified Newton-Raphson

Νέα συνάρτηση $f(x) = 94\cos x^3 - 24\cos x + 177\sin x^2 - 108\sin x^4 - 72\cos x^3 \sin x^2$



Τροποποιημένη μέθοδος Newton-Raphson.

```python
#ASKISI 2 #A
def modified_NR(f2,derf2,dderf2,x):
    iterat=0
    while (np.abs(f2(x)) > e):
        iterat=iterat+1
        x= x - 1/((derf2(x)/f2(x)) - 1/2*(dderf2(x)/derf2(x))) #modified
        #print(iterat, x)
    return x,iterat
```

```python
modified_NR(f2,derf2,dderf2,1)
```

```
(1.0388850369295233, 243)
```

```python
modified_NR(f2,derf2,dderf2,3)
```

```
(2.300523985643211, 6)
```

```python
modified_NR(f2,derf2,dderf2,2)
```

```
(5.2442785114121895, 294)
```

Παρατηρώ οτι επιτυγχάνεται ακρίβεια 1, 2 και 1 δεκαδικών αντίστοιχα (για x=1, x=3 και x=2) με αρκετά μεγάλο αριθμό επαναλήψεων (143,6 και 294).

5 Modified Bisection Method

Τροποποιημένη Bisection Method όπου αντί για την χρήση του μέσου για την εύρεση της ρίζας επιλέγεται ένα τυχαίο σημείο (*import random*).

```python
#B
def modified_BM(a,b,iterat):

    if  f2(a)*f2(b)<0:
        m = random.uniform(a,b) #modified
        if (np.abs(f2(m)) < e):
            m= m + e
            return m, iterat

        if f2(a)*f2(m)<0:
            iterat=iterat+1
            return modified_BM(a,m,iterat)
        else:
            iterat=iterat+1
            return modified_BM(m,b,iterat)
```

```python
modified_BM(0.9,2,0)
```

```
(1.046039061301219, 6)
```

Παρατηρώ ότι η σύκλιση είναι γρήγορη (*6 iterations*) με ακρίβεια 2 δεκαδικών.

6 Modified Secant Method

```
def modified_SM(x1,x2,x3):
    iterat=0
    q=f2(x1)/f2(x2)
    r=f2(x3)/f2(x2)
    s=f2(x3)/f2(x1)
    while (np.abs(f2(x3)) > e):
        iterat=iterat+1
        x4 = x3 - (r*(r-q)*(x3-x2)+(1-r)*s*(x3-x1))/((q-1)*(r-1)*(s-1))
        x1=x2
        x2=x3
        x3=x4
    return x3,iterat
```

```
modified_SM(0,1,3)
```

```
(1.0391315493829685, 291)
```

Παρατηρώ αργή σύγκλιση (*291 iterations*) με ακρίβεια 1ος δεκαδικού.

7 Επίλυση γραμμικού συστήματος Ax=b με την μορφή PA=LU

Σπάω τον πίνακα A σε γινόμενο Lower Upper δηλ. A= LU. Γνωρίζω οτι Ax = b -¿ LUx = b θέτω Ux = y , οπότε λύνω:

- Ly= b
- Ux=y

Δημιουργώ αρχικά συνάρτηση **findLPU(A)** που δέχεται ως όρισμα τον πίνακα A και επιστρέφει τους πίνακες L, P, U. O πίνακας L είναι η αποθηκευμένες τιμές με τις οποίες πολλαπλασιάζεται κάθε γραμμή κατα την μέθοδο Gauss. O πίνακας U είναι ο πίνακας A μετά απο Gauss και ο πίνακας P είναι ο μοναδιαίος πίνακας τροποποιημένος με βάση τον πίνακα A.

```python
#evresi x me methodo PA=LU
def find_LPU(A):

    L = np.zeros((A.shape[0],A.shape[1]))  #arxikopoiw me midenika
    P= np.diag(np.ones(A.shape[0]))        #arxikopoiw me 1 diagwnio 0 allou

    for j in range(A.shape[0]-1):
            if (A[j][1]==0 or A[j][1]<A[j+1][1]):
                Pnew= np.diag(np.ones(A.shape[0]))
                A[[j+1,j],:]=A[[j,j+1],:] #vazw ta midenika kai tous mikroterous arithmous katw

                Pnew[[j+1,j],:]=Pnew[[j,j+1],:] #gia kathe allagi grammis tou A, allazw kai ton P
                P = np.dot(Pnew,P)              #gia kathe allagi pollaplasiazetai o kainourgios me ton palio P

    for i in range(0, A.shape[0]):


        L[i][i]=1

        for j in range(i + 1, A.shape[0]): #Gauss
            if (A[i][j]):
                l=A[j][i]/A[i][i]
                A[j] = A[j] - A[i] * (A[j][i]/A[i][i])
                L[j][i] = l
        U=A

    return L,P,U
```

Η τελική μου συνάρτηση είναι η **PALU(A,b)** η οποία λύνει το σύστημα εξισώσεων και επιστρέφει τον πίνακα x.

```python
def PALU(A,b):
    L,P,U= find_LPU(A)
    b = np.dot(P,b)
    #print("L\n",L)
    #print("P\n",P)
    #print("U\n",U)
    #print(b)
    y = solve(L,b) #Ly=b
    print("y= ",y)
    x=solve(U,y)   #Ux=y

    return x
```

Δοκιμάζω για πίνακα A=[[2,1,5],[4,4,-4],[1,3,1]] και πίνακα b=[5,0,6]

```
M= np.array([[2,1,5],[4,4,-4],[1,3,1]])
b=np.array([5,0,6])
PALU(M,b)
```

```
L
 [[ 1.    0.    0.  ]
 [ 0.25  1.    0.  ]
 [ 0.5  -0.5   1.  ]]
P
 [[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
U
 [[ 4  4 -4]
 [ 0  2  2]
 [ 0  0  8]]
y=  [0. 6. 8.]

array([-1.,  2.,  1.])
```

8 Αποσύνθεση Cholesky

Έστω A (nxn) συμμετρικός και θετικά ορισμένος πίνακας. Τότε υπάρχει μοναδικός κάτω τριγωνικός πίνακας L με θτεικά διαγώνια στοιχεία (όχι αναγκαστικά μονάδες) τέτοιος ώστε $A = LL^T$

Γενικά ισχύει:

$$\alpha_{ij} = \sum_{k=1}^{n} l_{ik} l_{jk} = \sum_{k=1}^{j-1} l_{ik} l_{jk} + l_{ij} l_{jj}$$

Οπότε για i=1,...,n και j=1,...,i-1:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right)$$

```python
def cholesky(A):
    L = np.array([[0.0] * A.shape[0] for i in range(A.shape[0])])

    for i in range(A.shape[0]):
        for j in range(i+1):
            sum_ch = sum(L[i][k] * L[j][k] for k in range(j))

            if (i == j):
                L[i][j] = math.sqrt(A[i][i] - sum_ch)
            else:
                L[i][j] = (1.0 / L[j][j] * (A[i][j] - sum_ch))
    return L
```

Δοκιμή για A=[[6, 3, 4, 8], [3, 6, 5, 1], [4, 5, 10, 7], [8, 1, 7, 25]]

```python
A = np.array([[6, 3, 4, 8], [3, 6, 5, 1], [4, 5, 10, 7], [8, 1, 7, 25]])
L = cholesky(A)
print("A:\n",A)


print("L:\n",L)
```

```
A:
 [[ 6  3  4  8]
 [ 3  6  5  1]
 [ 4  5 10  7]
 [ 8  1  7 25]]
L:
 [[ 2.44948974  0.          0.          0.        ]
 [ 1.22474487  2.12132034  0.          0.        ]
 [ 1.63299316  1.41421356  2.30940108  0.        ]
 [ 3.26598632 -1.41421356  1.58771324  3.13249102]]
```