ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

5/03/2021

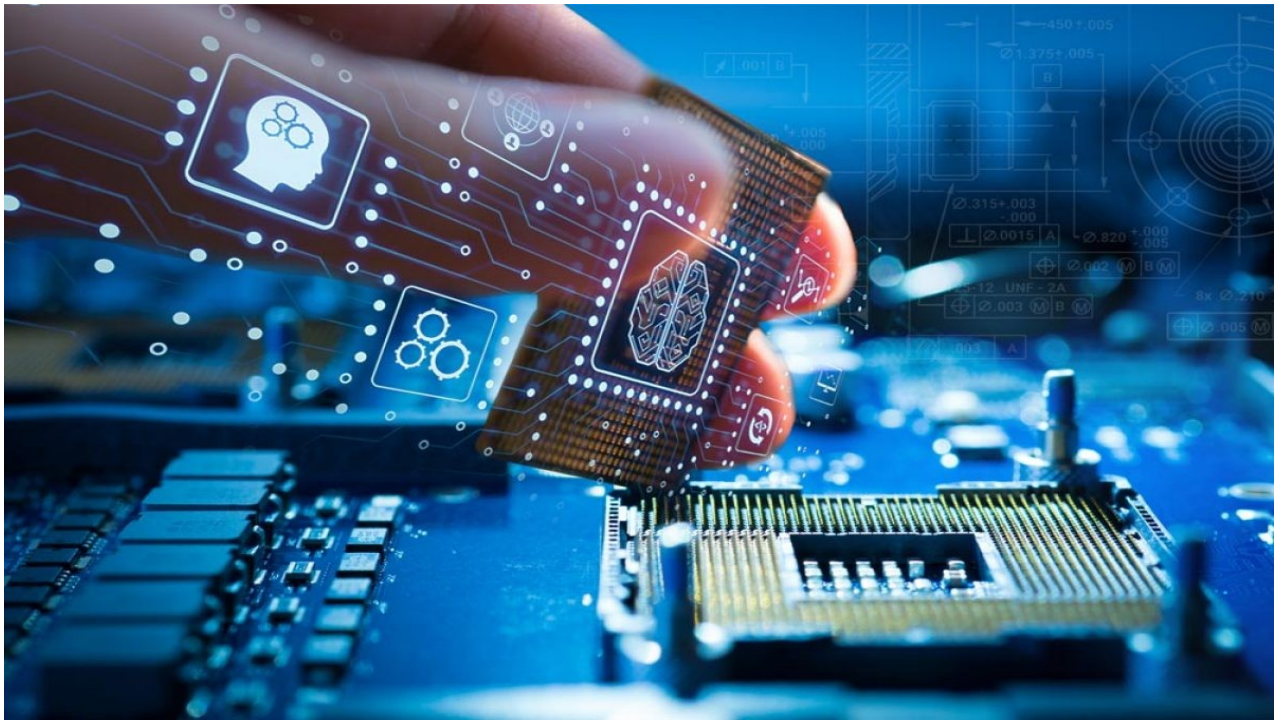
ΗΡΥ608/419-ΑΝΑΠΤΥΞΗ ΕΡΓΑΛΕΙΩΝ CAD ΓΙΑ ΣΧΕΔΙΑΣΗ ΟΛΟΚΛΗΡΩΜΕΝΩΝ ΚΥΚΛΩΜΑΤΩΝ

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2019

ΑΣΚΗΣΗ 1-ΜΕΘΟΔΟΣ NEWTON-RAPHSON

ΑΝΑΦΟΡΑ

Μυλωνάκης
Εμμανουήλ
2015030079



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σκοπός-Περιγραφή

Σκοπός της πρώτης άσκησης στο μάθημα Ανάπτυξη Εργαλείων CAD για Σχεδίαση Ολοκληρωμένων Κυκλωμάτων, ήταν η υλοποίηση ενός κώδικα σε γλώσσα προγραμματισμού C, ο οποίος υπολογίζει την ρίζα ενός πολυωνύμου με την χρήση της μεθόδου Newton-Raphson. Ο χρήστης εισάγει στο πρόγραμμα το επιθυμητό πολυώνυμο το πολύ 5ου βαθμού και στο τέλος εκτυπώνονται κάποια στατιστικά όπως:

- Το πλήθος επαναλήψεων που χρεάστηκε το πρόγραμμα μέχρι την εύρεση της ρίζας.
- Το πλήθος προσθέσεων και αφαιρέσων που πραγματοποιήθηκαν.
- Το πλήθος πολλαπλασιασμών που πραγματοποιήθηκαν.
- Το πλήθος διαίρεσεων που πραγματοποιήθηκαν.

Η παράγωγος της συνάρτησης που απαιτεί η μέθοδος Newton-Raphson υπολογίστηκε με δύο διαφορετικούς τρόπους, αναλυτικά και αριθμητικά. Ο αναλυτικός τρόπος πραγματοποιήθηκε με την κλασσική μέθοδο παραγώγισης πολυωνύμου που έχουμε διδαχθεί από το Λύκειο, αφαιρούμε τον αριθμό ένα από τον εκθέτη και πολλαπλασιάζουμε τον αρχικό εκθέτη με τον αντίστοιχο συντελεστή του πολυωνύμου. Για την αριθμητική μέθοδο αξιοποιήσαμε τον ορισμό της έννοιας "παράγωγος", δηλαδή τον ρυθμό μεταβολής της συνάρτησης. Διαλέγοντας μία μικρή τιμή "δ", πρώτα υπολογίζουμε την τιμή της συνάρτησης στο x_0, έπειτα στο $x_0+\delta$, και τέλος, τον λόγο διαφορών $\Delta y/\Delta x$.

Υλοποίηση-Κώδικας

Η μέθοδος Newton-Raphson είναι μία επαναληπτική διαδικασία για την προσέγγιση μιας ρίζας σε μία συνάρτηση. Δεν απαιτεί αναγκαστικά πολυώνυμο, αρκεί να υπάρχει η πρώτη παράγωγος της συναρτήσης και να είναι συνεχής. Στην άσκηση αυτή δουλέψαμε με πολυώνυμο καθώς τέτοιες συναρτήσεις περιγράφουν την τάση και το ρεύμα στα κυκλώματά μας κατά πλειοψηφία. Η διαδικασία αυτή μπορεί να περιγραφεί από τον παρακάτω αλγόριθμο.

- Αρχικά με βάση τον τύπο προβλήματος που λύνουμε, επιλέγουμε μία τιμή ανωχής (tolerance) που αντιπροσωπεύει μια αποδεκτή απόκλιση από την πραγματική ρίζα της συνάρτησης. Η τιμή αυτή ήταν "error = 0.001 ή 0.1%" σύμφωνα με την εκφώνηση.

- Διαλέγουμε την τιμή x_0 για την προσέγγιση της ρίζας. Στον κώδικα, επιλέχθηκε η τιμή 2.5 με το σκεπτικό ότι στα ολοκληρωμένα κυκλώματα η τάση παίρνει τιμές στο διάστημα [0-5] Volt. Διαλέγοντας την μέση τιμή του διαστήματος, βοηθάμε τον ίδιο τον αλγόριθμο να συγκλίσει πιο γρήγορα.

- Έπειτα ακολουθάμε επαναληπτικά τα παρακάτω βήματα:
    1. Υπολογισμός του $f(x_n)$. Στην πρώτη επανάληψη $x_n=x_0$.
    2. Υπολογισμός του $f'(x_n)$. Στην πρώτη επανάληψη $x_n=x_0$.
    3. $h = - f(x_n) / f'(x_n)$.
    4. $x_{n+1} = x_n + h$.
- Μέχρις ότου h < error. Επιπρόσθετα, ο κώδικας υλοποιεί το πολύ είκοσι επαναλήψεις.

Έχοντας αναλύση το θεωρητικό της εργασίας, προχωράμε στην επεξήγηση του κώδικα. Σε αυτόν, υλοποιούνται δύο συναρτήσεις εν ονόματι **question_a** και **question_b**, για τον αναλυτικό και αριθμητικό υπολογισμό της παραγώγου αντίστιοχα. Για την εκτίμηση του πολυωνύμου και της παραγώγου του για μία συγκεκριμένη τιμή της μεταβλητής x, χρησιμοποιήθηκαν αντίστοιχα οι συναρτήσεις **funct_calc()** και **deriv_calc().** Παραθέτουμε παρακάτω και τους αντίστοιχους κώδικες.

question_a

```
95  while(max>0)
96  {
97      stats[0]++;
98
99      funct = funct_calc(deg, C, x_0);      // Step
100     deriv = deriv_calc(deg, C, x_0);      // Step
101     h = -(funct / deriv);                 // Step
102
103     stats[2]++; // For the multiplication with -1
104     stats[3]++;
105
106     x_0 = x_0 + h; //Step 4. Calculate x_(n+1) =
107     stats[1]++;
108
109     printf("Loop %d. x_0 = %f\n", stats[0], x_0);
110
111     if(fabs(h)<error) break;
112
113     max--;
114 }
```

Εικόνα 1.

question_b

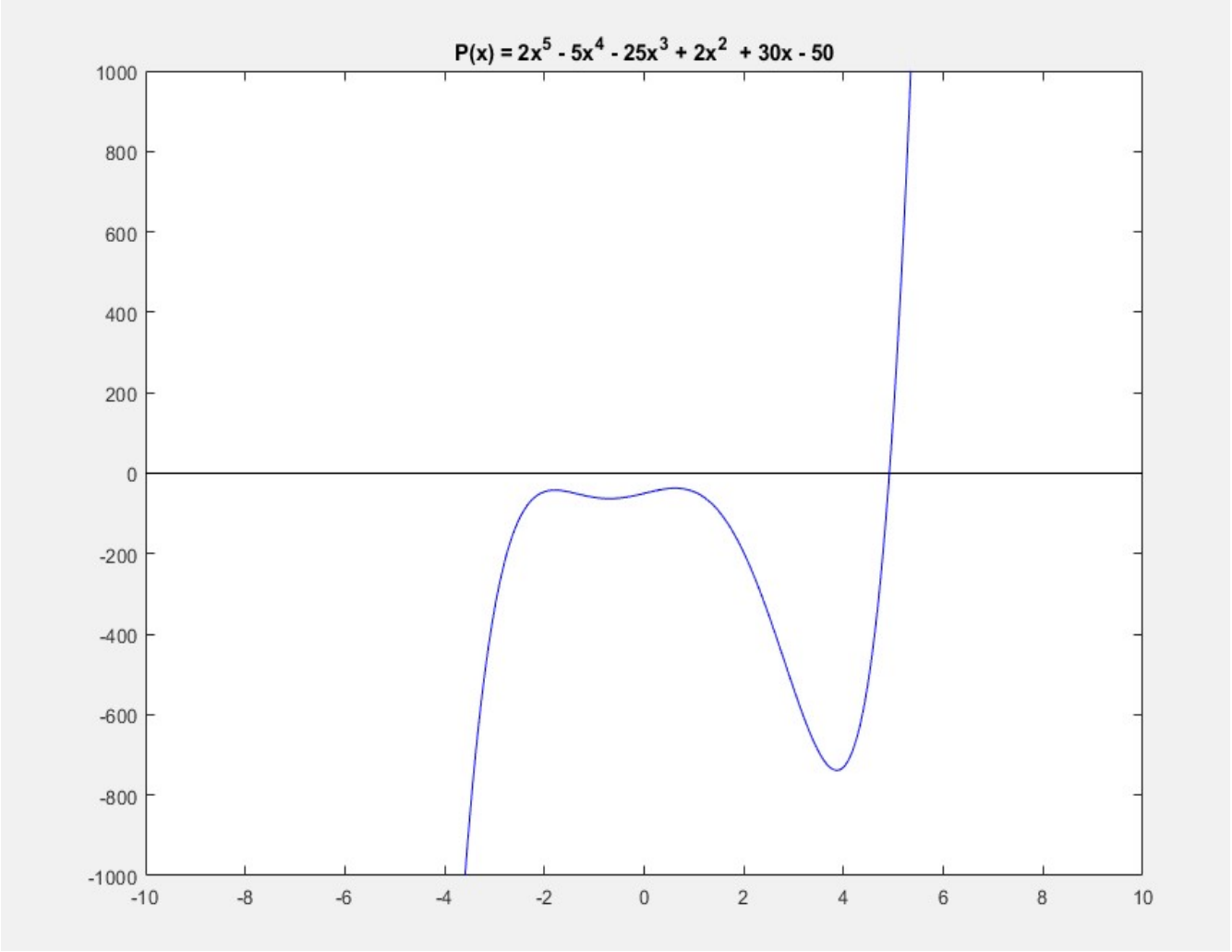
```
137 while (max > 0)
138 {
139     stats[0]++;
140
141     funct1 = funct_calc(deg, C, x_0);       // Step
142     funct2 = funct_calc(deg, C, x_0 + dx); // Step
143     stats[1]++;                             //One a
144
145     deriv = (funct2 - funct1) / dx;         // Step
146     stats[1]++;                             //One s
147     stats[3]++;                             //One d
148
149     h = -(funct1 / deriv);
150     stats[2]++; // For the multiplication with -1.
151     stats[3]++;
152
153     x_0 = x_0 + h; //Step 4. Calculate x_(n+1) = x
154     stats[1]++;
155
156     printf("Loop %d. x_0 = %f\n", stats[0], x_0);
157
158     if (fabs(h) < error) break;
159
160     max--;
161 }
```

Εικόνα 2.

Η **Εικόνα 1** αφορά το υποερώτημα για τον αναλυτικό υπολογισμό της παραγώγου, ενώ η **Εικόνα 2** τον αριθμητικό υπολογισμό. Στην πρώτη καλούμε την συνάρτηση derive_calc() που υπολογίζει αναλυτικά την τιμή της παραγώγου, ενώ στην δεύτερη περίπτωση, καλάμε για δεύτερη φορά την συνάρτηση func_calc() αλλά με όρισμα $\mathbf{x_0+\delta}$. Έπειτα υπολογίζουμε τον λόγο διαφορών $\mathbf{\Delta x/\Delta y}$ (βλ. γραμμή 145).

Εκτέλεση Κώδικα-Αποτελέσματα.

Για του σκοπούς της άσκησης, χρησιμοποιήθηκε το πολυώνυμο **$P(x) = 2x^5 - 5x^4 - 25x^3 + 2x^2 + 30x - 50$.** Σύμφωνα με το εργαλείο MATLAB, το πολυωνύμο αυτό έχει ακριβώς μία πραγματική ρίζα στην τιμή x = 4.9173 με την παρακάτω γραφική παράσταση.



Για το παραπάνω πολυώνυμο, εναρκτήρια τιμή x_0=2.5 και e=0.1%, έχουμε τα παρακάτω αποτελέσματα για διάφορες τιμές του **'δ'** που αφορά τον αριθμητικό υπολογισμό της παραγώγου:

δ = 0.1

```
========== Question a. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.492870 => P(x_0) = -353.125000
Loop 2. x_0 = 0.858366 => P(x_0) = -93.938766
Loop 3. x_0 = -0.531450 => P(x_0) = -40.368736
Loop 4. x_0 = 5.388817 => P(x_0) = -62.109715
Loop 5. x_0 = 5.033184 => P(x_0) = 1129.734375
Loop 6. x_0 = 4.926421 => P(x_0) = 215.409332
Loop 7. x_0 = 4.917356 => P(x_0) = 15.666840
Loop 8. x_0 = 4.917295 => P(x_0) = 0.105244
Final. x_0 = 4.917295 => P(x_0) = 0.105244
Number of Loops:           8
Number of Adds/Subtracts:  136
Number of Multiplications: 264
Number of Divisions:       8
========== Question b. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.508507 => P(x_0) = -353.125000
Loop 2. x_0 = 0.916624 => P(x_0) = -96.280861
Loop 3. x_0 = -0.011067 => P(x_0) = -42.310074
Loop 4. x_0 = 1.667970 => P(x_0) = -50.331730
Loop 5. x_0 = 1.047758 => P(x_0) = -123.289398
Loop 6. x_0 = 0.332457 => P(x_0) = -48.627655
Loop 7. x_0 = 2.415021 => P(x_0) = -40.776836
Loop 8. x_0 = 1.479449 => P(x_0) = -323.795685
Loop 9. x_0 = 0.890567 => P(x_0) = -91.971573
Loop 10. x_0 = -0.107287 => P(x_0) = -41.379448
Loop 11. x_0 = 1.696819 => P(x_0) = -53.165409
Loop 12. x_0 = 1.069670 => P(x_0) = -128.790283
Loop 13. x_0 = 0.375323 => P(x_0) = -49.964382
Loop 14. x_0 = 2.741530 => P(x_0) = -39.864662
Loop 15. x_0 = 1.553733 => P(x_0) = -440.566528
Loop 16. x_0 = 0.955726 => P(x_0) = -103.360306
Loop 17. x_0 = 0.111691 => P(x_0) = -43.902519
Loop 18. x_0 = 1.746645 => P(x_0) = -46.659893
Loop 19. x_0 = 1.106395 => P(x_0) = -138.737549
Loop 20. x_0 = 0.441203 => P(x_0) = -52.395119
Final. x_0 = 0.441203 => P(x_0) = -52.395119
Number of Loops:           20
Number of Adds/Subtracts:  300
Number of Multiplications: 660
Number of Divisions:       40
```

δ = 0.01

```
Starting with x_0 = 2.5
Loop 1. x_0 = 1.492870 => P(x_0) = -353.125000
Loop 2. x_0 = 0.858366 => P(x_0) = -93.938766
Loop 3. x_0 = -0.531450 => P(x_0) = -40.368736
Loop 4. x_0 = 5.388817 => P(x_0) = -62.109715
Loop 5. x_0 = 5.033184 => P(x_0) = 1129.734375
Loop 6. x_0 = 4.926421 => P(x_0) = 215.409332
Loop 7. x_0 = 4.917356 => P(x_0) = 15.666840
Loop 8. x_0 = 4.917295 => P(x_0) = 0.105244
Final. x_0 = 4.917295 => P(x_0) = 0.105244
Number of Loops:           8
Number of Adds/Subtracts:  136
Number of Multiplications: 264
Number of Divisions:       8
========== Question b. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.494596 => P(x_0) = -353.125000
Loop 2. x_0 = 0.864674 => P(x_0) = -94.194588
Loop 3. x_0 = -0.457122 => P(x_0) = -40.554821
Loop 4. x_0 = 3.586530 => P(x_0) = -61.165966
Loop 5. x_0 = -0.238280 => P(x_0) = -710.475159
Loop 6. x_0 = 2.006181 => P(x_0) = -56.714268
Loop 7. x_0 = 1.251192 => P(x_0) = -199.623505
Loop 8. x_0 = 0.594300 => P(x_0) = -64.422173
Loop 9. x_0 = 15.807336 => P(x_0) = -37.187626
Loop 10. x_0 = 12.842262 => P(x_0) = 1563891.000000
Loop 11. x_0 = 10.498238 => P(x_0) = 510330.781250
Loop 12. x_0 = 8.661934 => P(x_0) = 165866.968750
Loop 13. x_0 = 7.248192 => P(x_0) = 53488.054688
Loop 14. x_0 = 6.198404 => P(x_0) = 16963.291016
Loop 15. x_0 = 5.479599 => P(x_0) = 5177.716309
Loop 16. x_0 = 5.075140 => P(x_0) = 1433.759155
Loop 17. x_0 = 4.934607 => P(x_0) = 302.580505
Loop 18. x_0 = 4.917644 => P(x_0) = 29.902748
Loop 19. x_0 = 4.917296 => P(x_0) = 0.597130
Final. x_0 = 4.917296 => P(x_0) = 0.597130
Number of Loops:           19
Number of Adds/Subtracts:  285
Number of Multiplications: 627
Number of Divisions:       38
```

δ = 0.0001

```
========== Question a. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.492870 => P(x_0) = -353.125000
Loop 2. x_0 = 0.858366 => P(x_0) = -93.938766
Loop 3. x_0 = -0.531450 => P(x_0) = -40.368736
Loop 4. x_0 = 5.388817 => P(x_0) = -62.109715
Loop 5. x_0 = 5.033184 => P(x_0) = 1129.734375
Loop 6. x_0 = 4.926421 => P(x_0) = 215.409332
Loop 7. x_0 = 4.917356 => P(x_0) = 15.666840
Loop 8. x_0 = 4.917295 => P(x_0) = 0.105244
Final. x_0 = 4.917295 => P(x_0) = 0.105244
Number of Loops:           8
Number of Adds/Subtracts:  136
Number of Multiplications: 264
Number of Divisions:       8
========== Question b. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.491177 => P(x_0) = -353.125000
Loop 2. x_0 = 0.856883 => P(x_0) = -93.688446
Loop 3. x_0 = -0.541420 => P(x_0) = -40.325813
Loop 4. x_0 = 5.755226 => P(x_0) = -62.211281
Loop 5. x_0 = 5.213763 => P(x_0) = 2565.868896
Loop 6. x_0 = 4.969877 => P(x_0) = 628.173340
Loop 7. x_0 = 4.919357 => P(x_0) = 93.253143
Loop 8. x_0 = 4.917298 => P(x_0) = 3.522324
Loop 9. x_0 = 4.917294 => P(x_0) = 0.007645
Final. x_0 = 4.917294 => P(x_0) = 0.007645
Number of Loops:           9
Number of Adds/Subtracts:  135
Number of Multiplications: 297
Number of Divisions:       18
```

δ = 0.000001

```
========== Question a. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.492870 => P(x_0) = -353.125000
Loop 2. x_0 = 0.858366 => P(x_0) = -93.938766
Loop 3. x_0 = -0.531450 => P(x_0) = -40.368736
Loop 4. x_0 = 5.388817 => P(x_0) = -62.109715
Loop 5. x_0 = 5.033184 => P(x_0) = 1129.734375
Loop 6. x_0 = 4.926421 => P(x_0) = 215.409332
Loop 7. x_0 = 4.917356 => P(x_0) = 15.666840
Loop 8. x_0 = 4.917295 => P(x_0) = 0.105244
Final. x_0 = 4.917295 => P(x_0) = 0.105244
Number of Loops:           8
Number of Adds/Subtracts:  136
Number of Multiplications: 264
Number of Divisions:       8
========== Question b. ==========
Starting with x_0 = 2.5
Loop 1. x_0 = 1.535733 => P(x_0) = -353.125000
Loop 2. x_0 = 0.877174 => P(x_0) = -100.488136
Loop 3. x_0 = -0.464368 => P(x_0) = -40.940636
Loop 4. x_0 = 4.889667 => P(x_0) = -61.272072
Loop 5. x_0 = 4.912920 => P(x_0) = -46.125572
Loop 6. x_0 = 4.916631 => P(x_0) = -7.432461
Loop 7. x_0 = 4.917534 => P(x_0) = -1.129513
Final. x_0 = 4.917534 => P(x_0) = -1.129513
Number of Loops:           7
Number of Adds/Subtracts:  105
Number of Multiplications: 231
Number of Divisions:       14
```

Συμπεράσματα

Έπειτα από πειραματισμούς και σύμφωνα με τα παραπάνω αποτελέσματα μπορούμε να αποφανθούμε τα εξής:

- ✔ Εάν θέλουμε καλύτερη ακρίβεια μειώνουμε την ανωχή, δηλαδή την τιμή του "e". Παραδείγματος χάριν, για ακρίβεια 5 δεκαδικών ψηφίων θα χρησιμοποιούσαμε $e = 0.00001 = 10^{-5} = 0.001\%$. Μια τέτοια υλοποίηση απαιτεί παραπάνω χρόνο, καθώς θα χρειαστούν περισσότερες επαναλήψεις και περισσότερες πράξεις, και κατ'επέκταση κατανάλωση περισσότερων πόρων του εκάστοτε συστήματος που τρέχει ο κώδικας.
- ✔ Δεν υπάρχει καλύτερη ή χειρότερη μέθοδος από άποψη χρόνου ή πλήθος πράξεων, καθώς για την περίπτωση του αριθμητικού υπολογισμού της παραγώγου, μεγάλη σημασία καταλαμβάνει η τιμή που θα επιλέξουμε για το **'δ'.**
- ✔ Όσο μεγαλύτερη είναι η τιμή του **'δ'** τόσο μειώνεται η απόδοση της αριθμητικής μεθόδου. Αυτό διότι η ακρίβεια υπολογισμών ανά επανάληψη μειώνεται και το πρόγραμμα θα χρειαστεί περισσότερες επαναλήψεις μέχρι να συγκλίνει στην επιθυμητή ανοχή. Πράγμα που γίνεται αντιληπτό από τα στατιστικά του προγράμματος. Για την τιμή του "δ=0.1" οι 20 επαναλήψεις δεν ήταν αρκετές ώστε η μέθοδος να συγκλίνει, για **"δ=0.01"** χρείαστηκαν 19 επαναλήψεις για την σύγκλιση, και τέλος, για μία αρκετά μικρή τιμή όπως **"δ=0.000001"** πετύχαμε σύγκλιση γρηγορότερα και με λιγότερες πράξεις σε σχέση με τον αναλυτικό υπολογισμό της παραγώγου.

Φιλοσοφώντας λίγο ακόμα, ας παρατηρήσουμε την γραφική παράσταση του πολυωνύμου παραπάνω. Περίπου στο διάστημα του **D1 = [-2.5, 1,5]** ο ρυθμός μεταβολής της καμπύλης είναι αρκετά ομαλός, ενώ στο διάστημα **D2 = [4, 10)** η καμπύλη του πολυωνύμου φαίνεται να έχει υπερ-εκθετική συμπρεφορά. Αν η ρίζα βρισκόταν στο διάστημα D1, ενδεχομένως για δ = 0.1 να είχαμε σύγκλιση σε λιγότερο απο 20 επαναλήψεις. Καθώς η μέθοδος Newton-Raphson βασίζεται στην παράγωγο της συνάρτησης άρα και κατ'εξοχήν στο ρυθμό μεταβολής της. Στο διάστημα D1, για διάφορες μεταβολές του x λαμβάνουμε "ανόλογες προς ίδιες" μεταβολής της f(x) σύμφωνα με την γραφική παράσταση. Όμως στο διάστημα D2, για μεταβολές του x λαμβάνουμε υπερ-πολλαπλάσιες μεταβολές της f(x) λόγο της εκθετικής της συμπεριφοράς στο διάστημα αυτό. Έτσι μπορεί φαινομενικά ο αριθμός 0.1 να είναι μικρός, αλλά για μία μεταβολή κατά 0.1 του x να πάρουμε 10 και 20 φορές μεγαλύτερη τιμή της f(x). Πράγμα που σημαίνει ότι μπορεί να είμαστε κόντα προς σύγκλιση, αλλά με μία μικρή μεταβολή του x να "βρεθούμε έτη φωτός μακριά από την ρίζα". Για τον λόγο αυτό για δ = 0.1 δεν συγκλίναμε εντός 20 επαναλήψεων και σίγουρα υπάρχει περίπτωση για μία "κακιά" τιμή δ, να μην καταφέρουμε να συγκλίνουμε πότε σε κάποια ρίζα. Εν κατακλείδι, το τελευταίο απο τα παράπανω συμπεράσματα είναι καθαρό υποκειμενικό και αφορά το συγκεκριμένο πολυώνυμο. Όσον αφορά τι τιμή θα διαλέξουμε για το **δ**... `***Master the art of observing***`.