LAB REPORT 4 Group-2

Rupali Sharma, Chhavi Chahar, Smriti Chhibber,
Disha Nareda, Amlan Nayak

February 2021

1 Birge Vieta Method

Birge-Vieta method is an application of Newton Raphson method to find the real roots of an nth degree polynomial $P_n(x) = 0$. The method extracts a linear factor of the form $(x - p)$ from the polynomial $P_n(x)$, such that p is a real root of the polynomial. Starting with p_0, it obtains a sequence of iterates p_k from,

$$p_{k+1} = p_k - \frac{P_n(p_k)}{P_n'(p_k)} \tag{1}$$

$$p_{k+1} = p_k - \frac{b_n}{c_{n-1}}$$

which is same as the Newton Raphson Method, and is the step nested in the loop for this scheme.
The value for b_n and c_{n-1} are obtained from recurrence relations, found using synthetic division procedure.

$$b_i = a_i + p_k b_{i-1}$$

$$c_i = b_i + p_k c_{i-1}$$

$$c_0 = b_0 = a_0, c_{-1} = b_{-1} = 0$$

In order to determine the next root, it obtains a deflated polynomial. Then uses the Newton Raphson method and finds the next root.

1.1 Problem

Prepare code for Birge Vieta method, solving the for roots of:

$$P_n(x) = 3x^5 + 20x^4 - 10x^3 - 240x^2 - 250x + 200$$

1.2 Algorithm

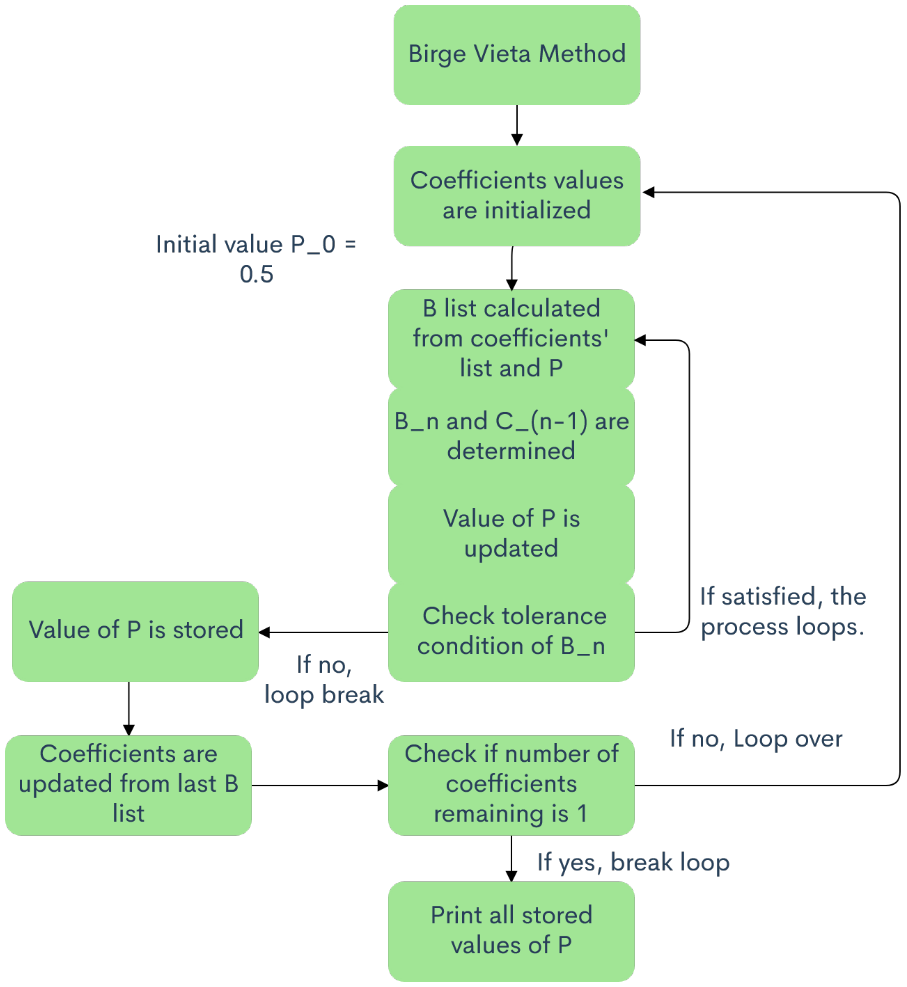
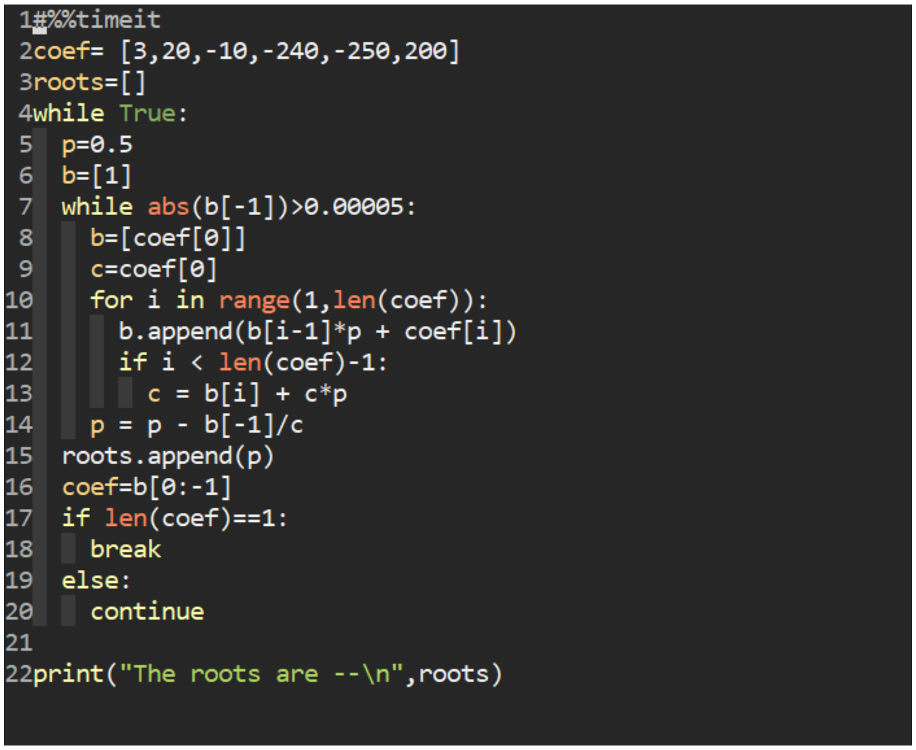

Birge Vieta Method
Coefficients values are initialized
Initial value P_0 = 0.5
B list calculated from coefficients' list and P
B_n and C_(n-1) are determined
Value of P is updated
Check tolerance condition of B_n
If satisfied, the process loops.
Value of P is stored
If no, loop break
Coefficients are updated from last B list
Check if number of coefficients remaining is 1
If no, Loop over
If yes, break loop
Print all stored values of P

Figure 1: Algorithm for Birge-Vieta Method

1.3 Code

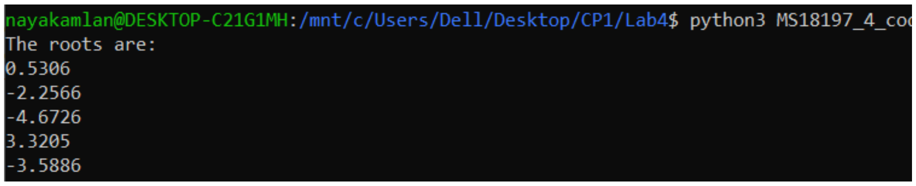


```
#%%timeit
coef= [3,20,-10,-240,-250,200]
roots=[]
while True:
    p=0.5
    b=[1]
    while abs(b[-1])>0.00005:
        b=[coef[0]]
        c=coef[0]
        for i in range(1,len(coef)):
            b.append(b[i-1]*p + coef[i])
            if i < len(coef)-1:
                c = b[i] + c*p
        p = p - b[-1]/c
    roots.append(p)
    coef=b[0:-1]
    if len(coef)==1:
        break
    else:
        continue

print("The roots are --\n",roots)
```

Figure 2: Birge-Vieta method

1.4 Output



```
nayakamlan@DESKTOP-C21G1MH:/mnt/c/Users/Dell/Desktop/CP1/Lab4$ python3 MS18197_4_coc
The roots are:
0.5306
-2.2566
-4.6726
3.3205
-3.5886
```

Figure 3: Output

1.5 Comparison of Chebyshev and Newton raphson method applied in nested loop

In the algorithm of Birge-Vieta method, Newton-Raphson method was used for finding the roots. Moving one step further we can also use Chebyshev method by including the second order derivative term. The general Chebyshev method has the form,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{1}{2}\frac{f(x_k)^2}{f'(x_k)^3}f''(x_k)$$

Modifying this for nth degree polynomial,

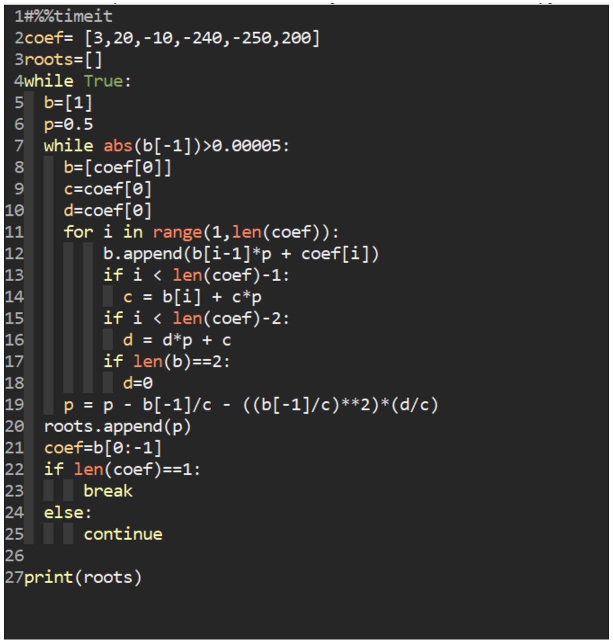
$$p_{k+1} = p_k - \frac{P_k(p_k)}{P'(p_k)} - \frac{1}{2}\frac{P(p_k)^2}{P'(p_k)^3}P''(p_k)$$

Taking the order argument one step further to find the second order coefficent we get,

$$d_k = c_k + pd_{k-2}$$

and using this we can find the roots using Chebyshev modification.

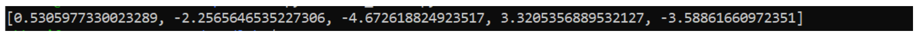
1.6 Code



```python
#%%timeit
coef= [3,20,-10,-240,-250,200]
roots=[]
while True:
    b=[1]
    p=0.5
    while abs(b[-1])>0.00005:
        b=[coef[0]]
        c=coef[0]
        d=coef[0]
        for i in range(1,len(coef)):
            b.append(b[i-1]*p + coef[i])
            if i < len(coef)-1:
                c = b[i] + c*p
            if i < len(coef)-2:
                d = d*p + c
            if len(b)==2:
                d=0
        p = p - b[-1]/c - ((b[-1]/c)**2)*(d/c)
    roots.append(p)
    coef=b[0:-1]
    if len(coef)==1:
        break
    else:
        continue

print(roots)
```

Figure 4: Using Chebyshev method in Birge-Vieta



```
[0.5305977330023289, -2.2565646535227306, -4.672618824923517, 3.3205356889532127, -3.58861660972351]
```

Figure 5: Chebyshev method Output in Birge-Vieta

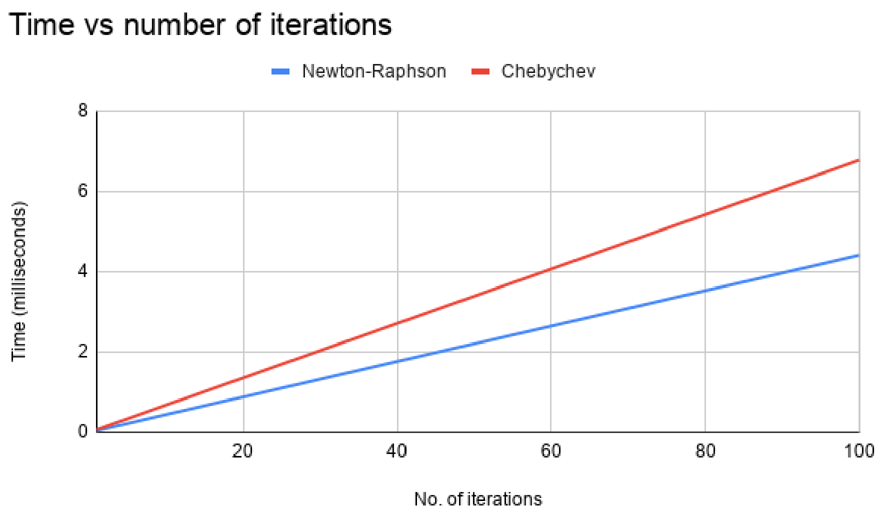


Figure 6: Time and Number of iterations in Birge-Vieta method using Newton-Raphson and Chebyshev Method

1.7 Round-off error

Throughout this process we work in nested loops. This is because this reduces the round off error that will be caused during the process. We can also check the root of an equation by directly putting it in the polynomial. But that gives rise to induced instability which can be demonstrated through the following example.Normally, we are used to substitute the value into the polynomial and do the math. We use this to simplify the process of evaluating a by dividing the polynomial into monomials. Each monomial involves a maximum of one multiplication and one addition processes. the computer processing time for a single multiplication process is 5 to 20 times the processing time of an addition process, and hence will have significant improvement on the execution time taken by the computer. The error accumulation in the direct evaluation of the polynomial vs synthetic division is quite significant due to the fact that there will be truncation error at every step

2 Bairstow's Method

In Bairstow's method, the polynomials is divided by a quadratic factor $x^2 - rx - s$. The degree of the resulting polynomial is two degrees less than the previous polynomial. If the remainder is zero, the roots r and s can be determined using the quadratic formula.

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$P_{n-2}(x) = b_2 + b_3 x + \cdots + b_{n-1} x^{n-3} + b_n x^{n-2}$$

$$R = b_1(x - r) + b_0$$

The recurrence relations are -

$$b_n = a_n$$

$$b_{n-1} = a_{n-1} + r b_n$$

$$b_i = a_i + r b_{i+1} + s b_{i+2}$$

For remainder to be zero, b_1 and b_0 should be zero. b_1 and b_0 are functions of s and r. Using Taylor series expansion-

$$b_1(r + \Delta r, s + \Delta s) = b_1 + \frac{\partial b_1}{\partial r}\Delta r + \frac{\partial b_1}{\partial s}\Delta s$$

$$b_0(r + \Delta r, s + \Delta s) = b_0 + \frac{\partial b_0}{\partial r}\Delta r + \frac{\partial b_0}{\partial s}\Delta s$$

The partial derivatives are obtained by synthetic division of b's. Δr and Δs can be calculated from the above two equations. The values of Δr and Δs are used to improve the initial guesses. The roots can then be determined by solving for x

$$x^2 - rx - s = 0$$

$$x = \frac{r \pm (r^2 + 4s)^{1/2}}{2}$$

If the quotient is a first order polynomial, $x = \frac{-s}{r}$

2.1 Problem

Prepare code for Bairstow's method and solve for roots of:

$$P_n(x) = x^6 - 3x^5 - 15x^4 + 22x^3 - 30x^2 - 30x + 180$$

2.2 Algorithm

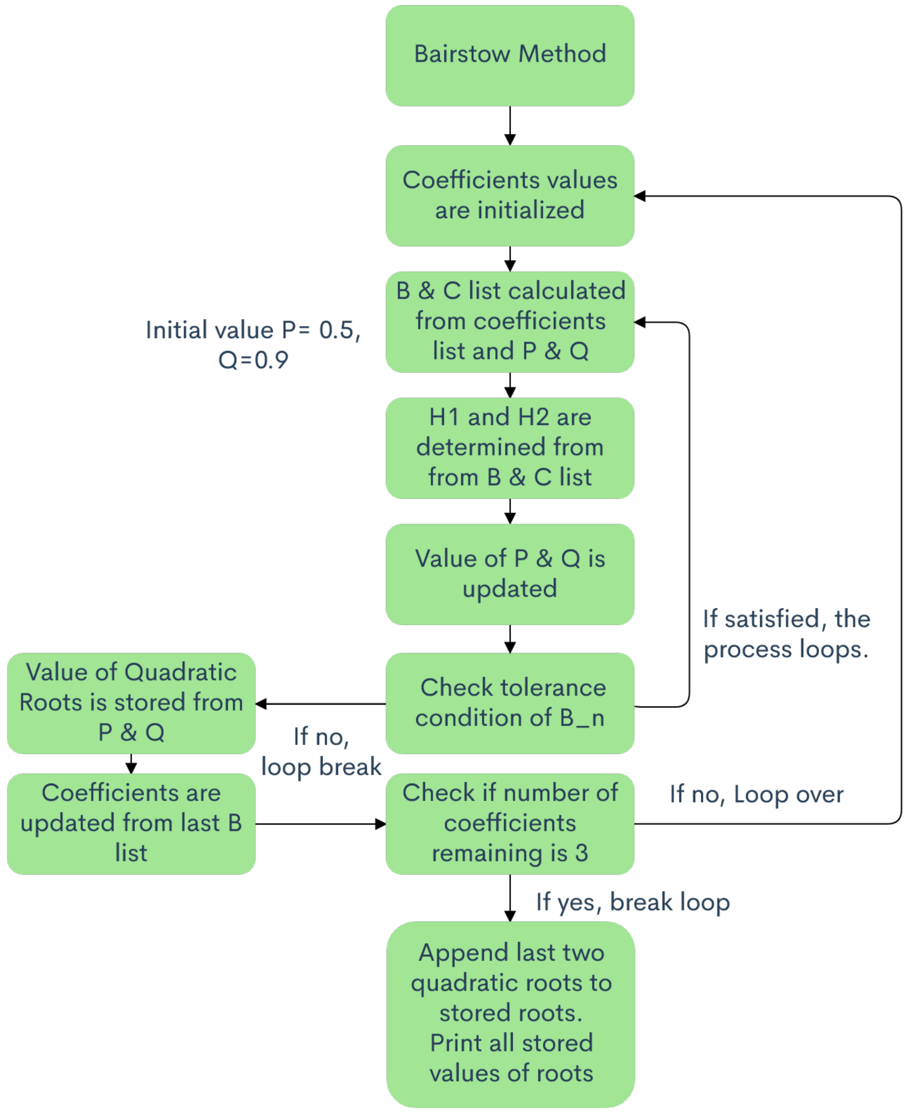
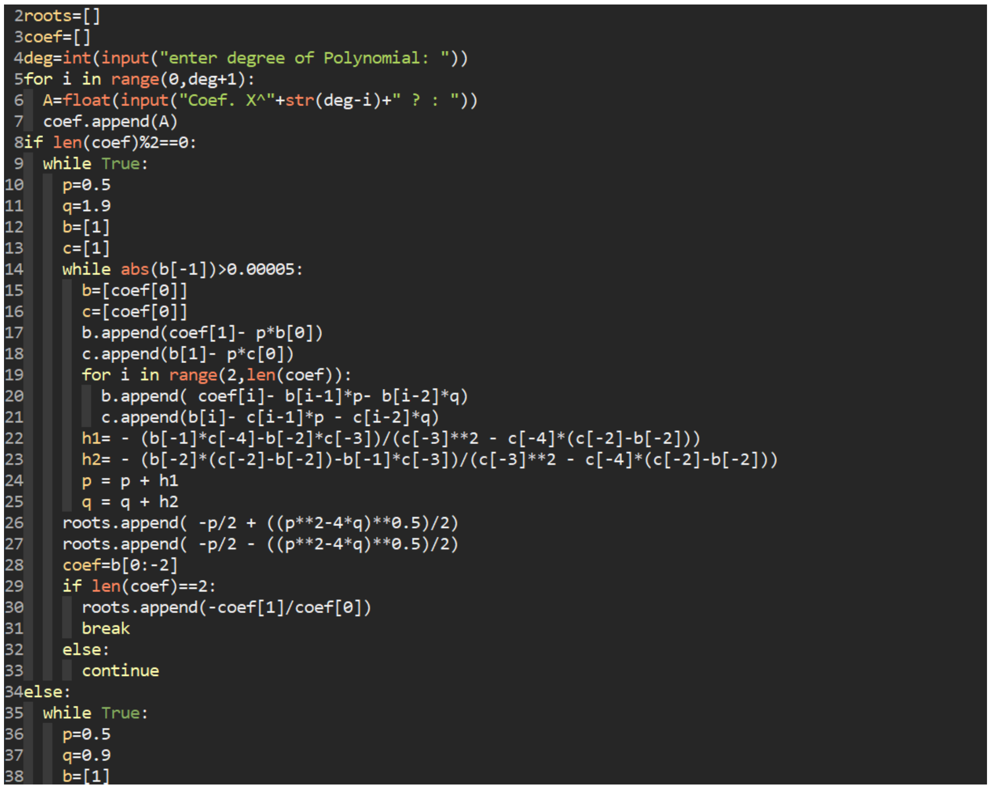

Bairstow Method
Coefficients values are initialized
B & C list calculated from coefficients list and P & Q
Initial value P= 0.5, Q=0.9
H1 and H2 are determined from from B & C list
Value of P & Q is updated
If satisfied, the process loops.
Value of Quadratic Roots is stored from P & Q
Check tolerance condition of B_n
If no, loop break
Coefficients are updated from last B list
Check if number of coefficients remaining is 3
If no, Loop over
If yes, break loop
Append last two quadratic roots to stored roots. Print all stored values of roots

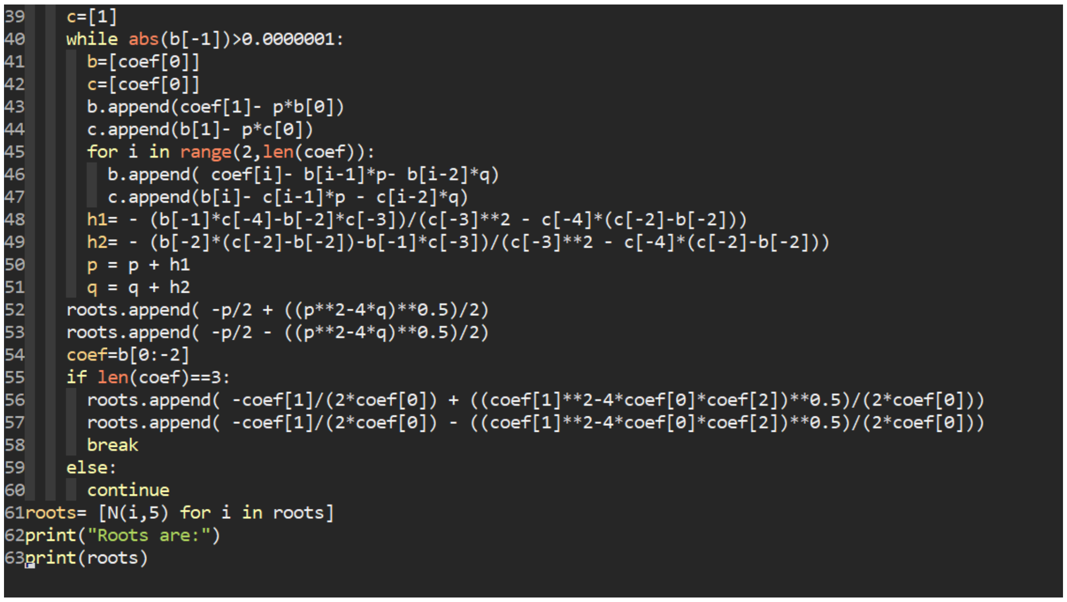
Figure 7: Algorithm for Bairstow Method -General

2.3 Code



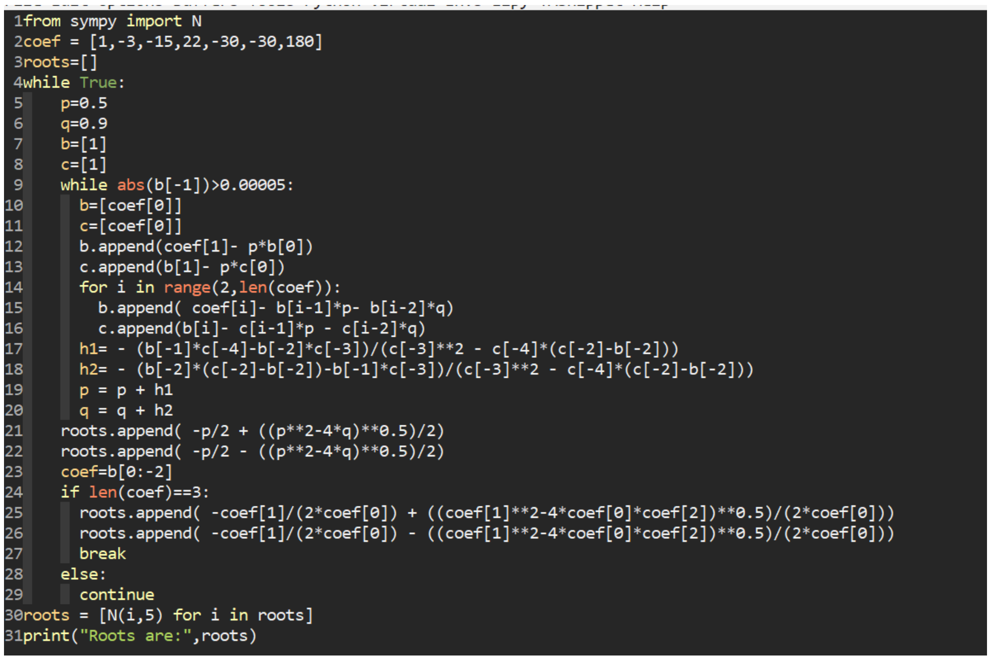
```
roots=[]
coef=[]
deg=int(input("enter degree of Polynomial: "))
for i in range(0,deg+1):
  A=float(input("Coef. X^"+str(deg-i)+" ? : "))
  coef.append(A)
if len(coef)%2==0:
  while True:
    p=0.5
    q=1.9
    b=[1]
    c=[1]
    while abs(b[-1])>0.00005:
      b=[coef[0]]
      c=[coef[0]]
      b.append(coef[1]- p*b[0])
      c.append(b[1]- p*c[0])
      for i in range(2,len(coef)):
        b.append( coef[i]- b[i-1]*p- b[i-2]*q)
        c.append(b[i]- c[i-1]*p - c[i-2]*q)
      h1= - (b[-1]*c[-4]-b[-2]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
      h2= - (b[-2]*(c[-2]-b[-2])-b[-1]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
      p = p + h1
      q = q + h2
    roots.append( -p/2 + ((p**2-4*q)**0.5)/2)
    roots.append( -p/2 - ((p**2-4*q)**0.5)/2)
    coef=b[0:-2]
    if len(coef)==2:
      roots.append(-coef[1]/coef[0])
      break
    else:
      continue
else:
  while True:
    p=0.5
    q=0.9
    b=[1]
```

Figure 8: Bairstow Method- General



```
        c=[1]
        while abs(b[-1])>0.0000001:
            b=[coef[0]]
            c=[coef[0]]
            b.append(coef[1]- p*b[0])
            c.append(b[1]- p*c[0])
            for i in range(2,len(coef)):
                b.append( coef[i]- b[i-1]*p- b[i-2]*q)
                c.append(b[i]- c[i-1]*p - c[i-2]*q)
            h1= - (b[-1]*c[-4]-b[-2]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
            h2= - (b[-2]*(c[-2]-b[-2])-b[-1]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
            p = p + h1
            q = q + h2
        roots.append( -p/2 + ((p**2-4*q)**0.5)/2)
        roots.append( -p/2 - ((p**2-4*q)**0.5)/2)
        coef=b[0:-2]
        if len(coef)==3:
            roots.append( -coef[1]/(2*coef[0]) + ((coef[1]**2-4*coef[0]*coef[2])**0.5)/(2*coef[0]))
            roots.append( -coef[1]/(2*coef[0]) - ((coef[1]**2-4*coef[0]*coef[2])**0.5)/(2*coef[0]))
            break
        else:
            continue
roots= [N(i,5) for i in roots]
print("Roots are:")
print(roots)
```

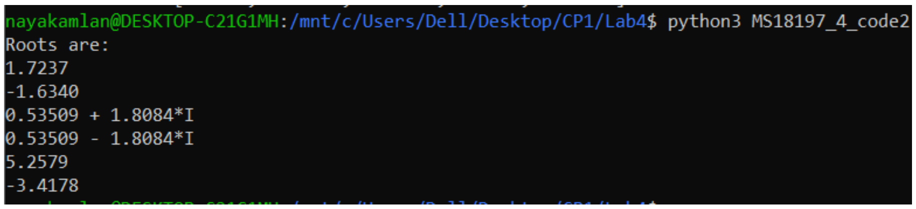
Figure 9: Bairstow Method - General



```python
from sympy import N
coef = [1,-3,-15,22,-30,-30,180]
roots=[]
while True:
    p=0.5
    q=0.9
    b=[1]
    c=[1]
    while abs(b[-1])>0.00005:
      b=[coef[0]]
      c=[coef[0]]
      b.append(coef[1]- p*b[0])
      c.append(b[1]- p*c[0])
      for i in range(2,len(coef)):
        b.append( coef[i]- b[i-1]*p- b[i-2]*q)
        c.append(b[i]- c[i-1]*p - c[i-2]*q)
      h1= - (b[-1]*c[-4]-b[-2]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
      h2= - (b[-2]*(c[-2]-b[-2])-b[-1]*c[-3])/(c[-3]**2 - c[-4]*(c[-2]-b[-2]))
      p = p + h1
      q = q + h2
    roots.append( -p/2 + ((p**2-4*q)**0.5)/2)
    roots.append( -p/2 - ((p**2-4*q)**0.5)/2)
    coef=b[0:-2]
    if len(coef)==3:
      roots.append( -coef[1]/(2*coef[0]) + ((coef[1]**2-4*coef[0]*coef[2])**0.5)/(2*coef[0]))
      roots.append( -coef[1]/(2*coef[0]) - ((coef[1]**2-4*coef[0]*coef[2])**0.5)/(2*coef[0]))
      break
    else:
      continue
roots = [N(i,5) for i in roots]
print("Roots are:",roots)
```

Figure 10: Bairstow Method for the given Polynomial

2.4 Output



```
nayakamlan@DESKTOP-C21G1MH:/mnt/c/Users/Dell/Desktop/CP1/Lab4$ python3 MS18197_4_code2
Roots are:
1.7237
-1.6340
0.53509 + 1.8084*I
0.53509 - 1.8084*I
5.2579
-3.4178
```

Figure 11: Bairstow Method- General

3 Summary

In this exercise, we used Birge-Vieta and Bairstow method for finding zeroes of a polynomial. We also tried to Use Chebyshev method. Chebyshev method took less number of iterations but it took more time to converge but the difference was not very significant for the problem given to us.

All five members of the group collaborated on these problems. We had discussion sessions to discuss the algorithms and make them better.

We also tried to work on the fractal representation of Bairstow method to demonstrate the convergence and divergence based on the initial guesses, but the work could not be completed and hence we decided on not including it in our reports.