Alunos: Luís Fernando Mendonça Junior
Professora: Priscila Cardoso Calegari
INE5202 - Cálculo Numérico

Relatório Exercício Programa 1

Ambos os programas foram escritos utilizando a linguagem Python, e foram utilizadas as bibliotecas ‘cmath’, ‘numpy’ e ‘matplotlib’ para a execução de algumas operações matemáticas e construção dos gráficos.

1 - A primeira atividade requer que seja implementado o método de Newton e que ele utilize o método de Horner para avaliar o polinômio e sua derivada. O código implementado ficou da seguinte forma:

```python
#função que calcula o valor do polinômio e sua derivada em x0
def horner(coeficientes, x0):
    n = len(coeficientes) - 1
    y = coeficientes[0]
    z = coeficientes[0]
    for j in range(1, n):
        k = coeficientes[j]
        y = x0 * y + k
        z = x0 * z + y
    y = x0 * y + coeficientes[n]
    return y, z

#função que calcula a raiz de um polinômio pelo método de Newton
def newton(coeficientes, x0, itmax):
    tolerancia = 10**-7
    xn = x0
    for _ in range(itmax):
        p_xn, dp_xn = horner(coeficientes, xn)
        if abs(p_xn) < tolerancia:
            break
        xn -= p_xn / dp_xn
    return xn
```

Primeiramente a função **horner** implementa o método de Horner para avaliar um polinômio em um ponto x0, ela itera pelos coeficientes do polinômio, atualizando dois valores, y e z, e retorna y e z no final, que são respectivamente os valores de p(x0) e p`(x0). Então função **newton** implementa o método de Newton para encontrar a raiz do polinômio, dentro de um loop de iteração, ela calcula o valor do polinômio e sua derivada usando o método de Horner, se o valor do polinômio for menor que uma tolerância pré definida, o loop para e retorna a estimativa final da raiz.

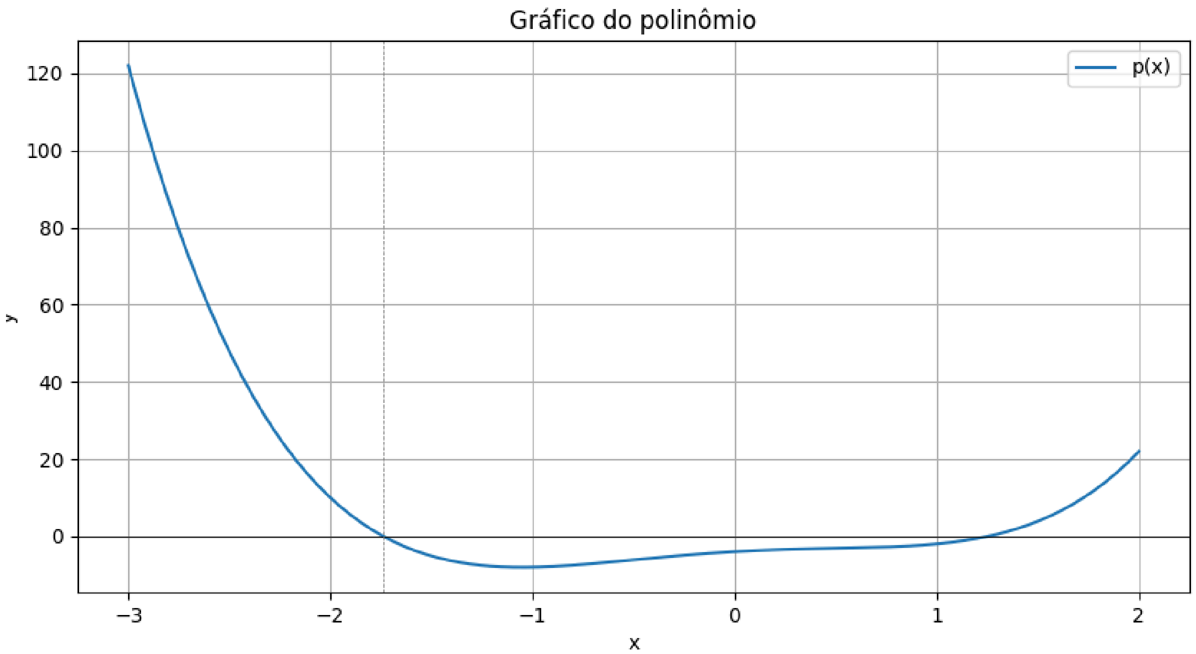
O algoritmo foi executado com os seguintes valores:

```python
if __name__ == "__main__":
    coeficientes = [2, 0, -3, 3, -4]   # Coeficientes do polinômio p(x) = x^4 + x^3 + x^2 + x + C
    x0 = -2                            # Valor inicial para x0
    itmax = 10                         # Número de iterações
```

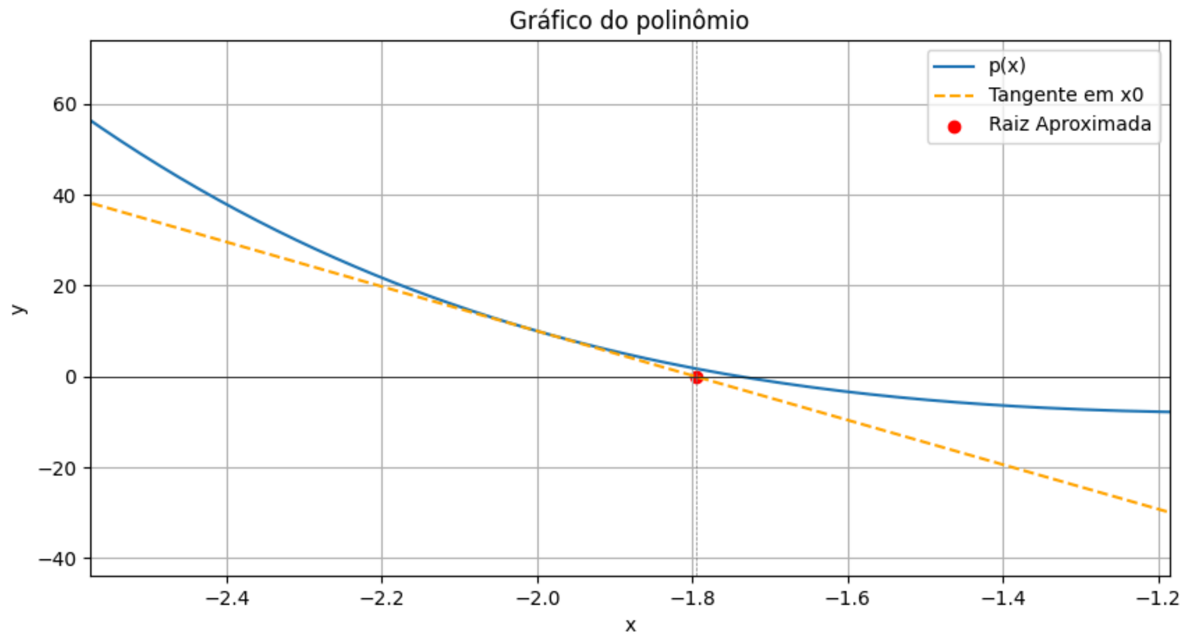
Então, testando com diferentes números de iterações, conseguimos extrair os seguintes dados:

i	raiz
1	-1.795918
2	-1.742432
3	-1.738970
4	-1.738956
5	-1.738956
10	-1.738956

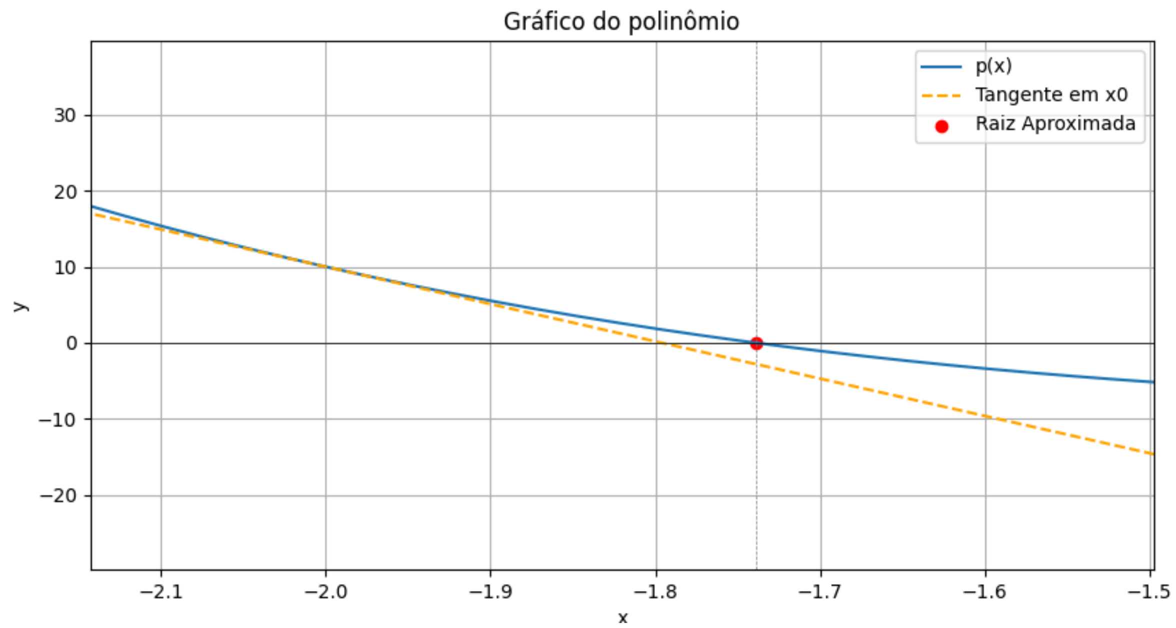
A conclusão que podemos tirar deles é que, para esse caso em específico, conseguimos ter um valor muito próximo para a raiz na quarta iteração do método, e como o loop é quebrado após uma certa tolerância, ele sempre irá acabar nesse resultado. Agora, analisando o resultado pelos gráficos.



Esse é o polinômio utilizado como exemplo no plano cartesiano, iremos analisar como o algoritmo se comporta ao executar o método com uma iteração e utilizado -2 como o valor inicial de x0.



Vemos então que em uma iteração do método a reta tangente ao polinômio no ponto x0 corta o eixo das abscissas exatamente no valor aproximado da raiz, onde será o valor de x1, mas como conseguimos analisar, quanto mais iterações do método, mais precisa é a nossa aproximação.



Então, é possível chegar a conclusão que após 3 iterações da função para esse polinômio, conseguimos chegar com um valor para uma de suas raízes com bastante precisão.

1 - A segunda atividade requer que seja implementar o método de muller, o código final ficou da seguinte forma:

```
#função que calcula a raiz de um polinômio pelo método de Muller
def muller (funcao, x0, x1, x2, itmax):
    f = funcao
    tolerancia = 10**-9
    x = x2
    it = 2

    while ((abs(f(x)) > tolerancia) and (it < itmax)):
        c = f(x2)
        q0 = (f(x0) - f(x2))/(x0 - x2)
        q1 = (f(x1) - f(x2))/(x1 - x2)
        a = (q0-q1)/(x0-x1)
        b = q0*((x2-x1)/(x0-x1)) + q1*((x0-x2)/(x0-x1))

        #Passo 9 tratando dos numeros complexos
        discriminante = cmath.sqrt(b**2 - 4*a*c)
        if abs(b + discriminante) > abs(b - discriminante):
            den = b + discriminante
        else:
            den = b - discriminante

        x = x2 - ((2*c) / den)
        x0, x1, x2 = x1, x2, x
        it = it+1
    return x
```

A função **muller** implementa o método de Muller para encontrar uma raiz de uma função polinomial. Dentro dela seguimos os passos originais do algoritmo, iniciamos determinando os valores iniciais e estabelecemos uma tolerância para a convergência. Em seguida, iteramos usando um loop while, calculando os valores necessários para a próxima iteração com base nos três pontos iniciais (x0, x1 e x2) e atualizando esses pontos a cada iteração.O loop continua até atingirmos a convergência desejada ou o número máximo de iterações (itmax) é alcançado.Finalmente, retornamos a estimativa final da raiz.

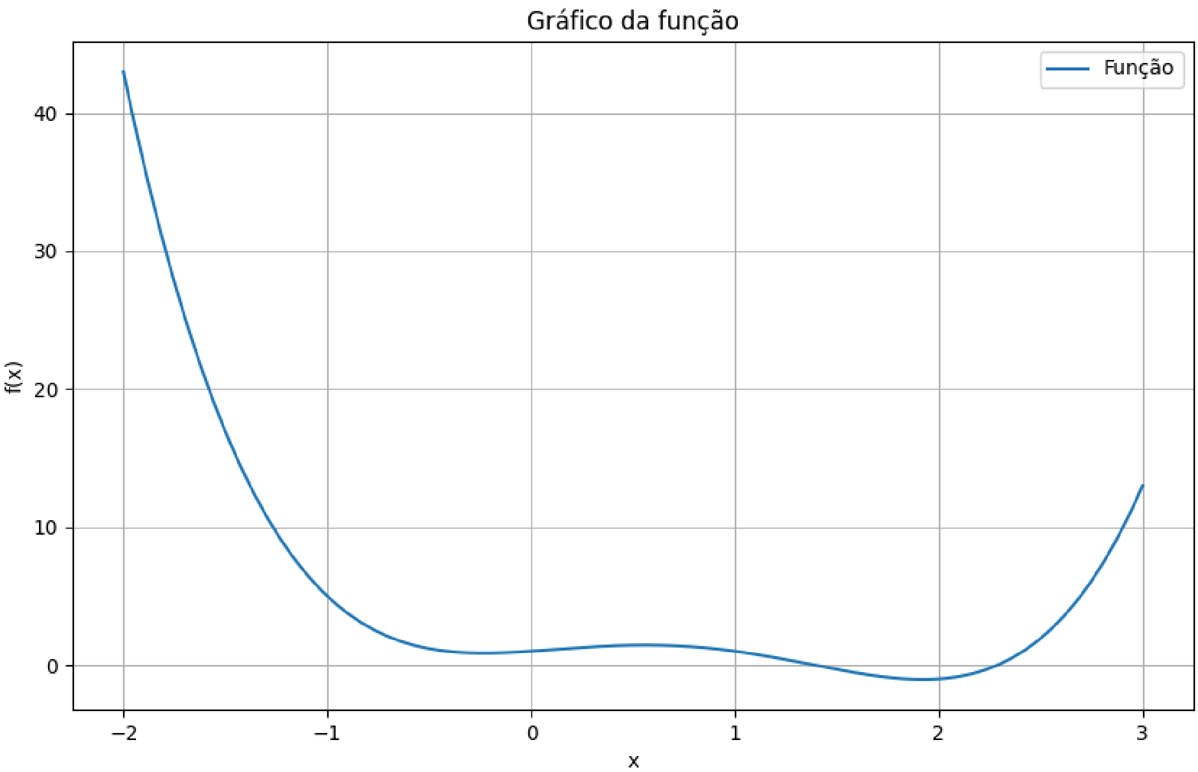
O algoritmo foi executado com os seguintes valores:

```
if __name__ == "__main__":
    funcao = lambda x: x**4 - 3*x**3 + x**2 + x + 1        # Polinomio p(x) = x^4 - 3x^3 + x^2 + x + 1
    f = funcao
    x0 = -0.5                                              # Valor inicial para x0
    x1 = 0                                                 # Valor inicial para x1
    x2 = 0.5                                               # Valor inicial para x2
    intervalos_arbitrario = [(0.2, 1,5), (1.6, 2.5)]       # Intervalos arbitrários para encontrar as raízes reais
    itmax = 10                                             # Número de iterações
```

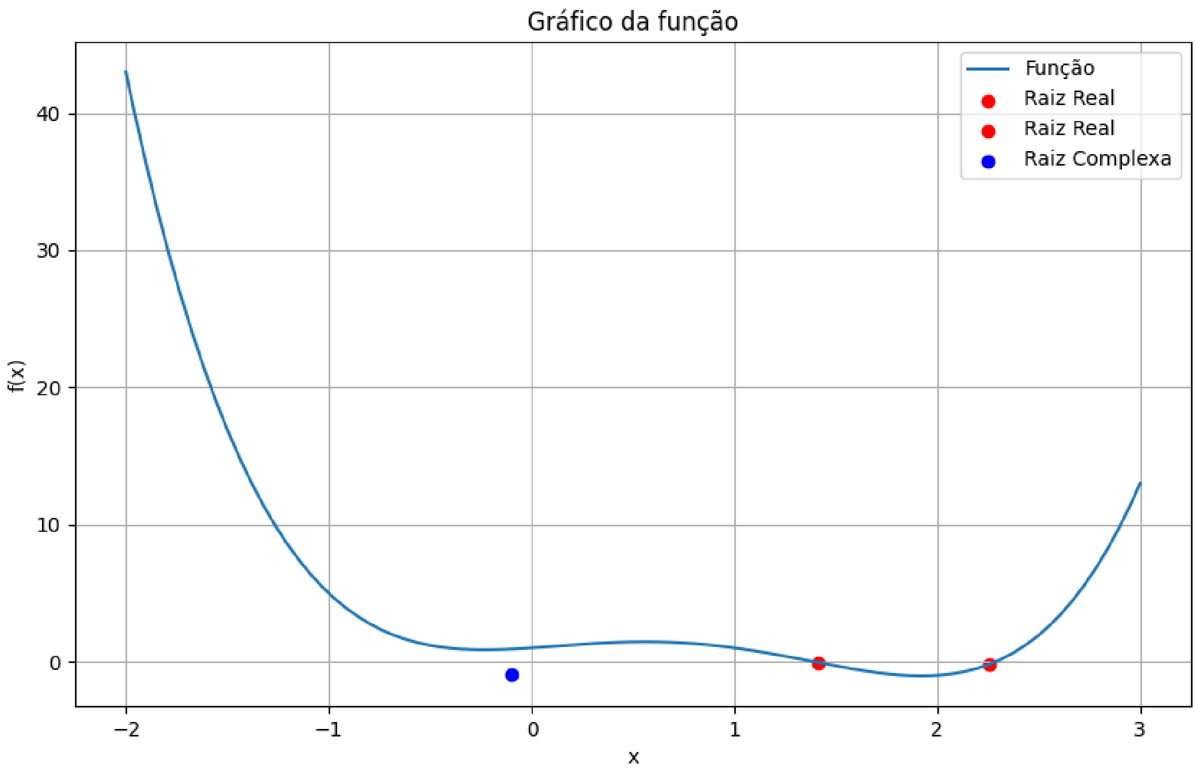
Então, testando com diferentes números de iterações, conseguimos extrair os seguintes dados:

i	raiz real 1	raiz real 2	raiz complexa
2	0.6	2.05	0.5
3	1.369691	2.259085	-0.0999-0.8888j
4	1.386909	2.287426	-0.2880-0.2382j
5	1.389367	2.288803	-0.3744-0.3742j
10	1.389390	2.288803	-0.3744-0.3742j

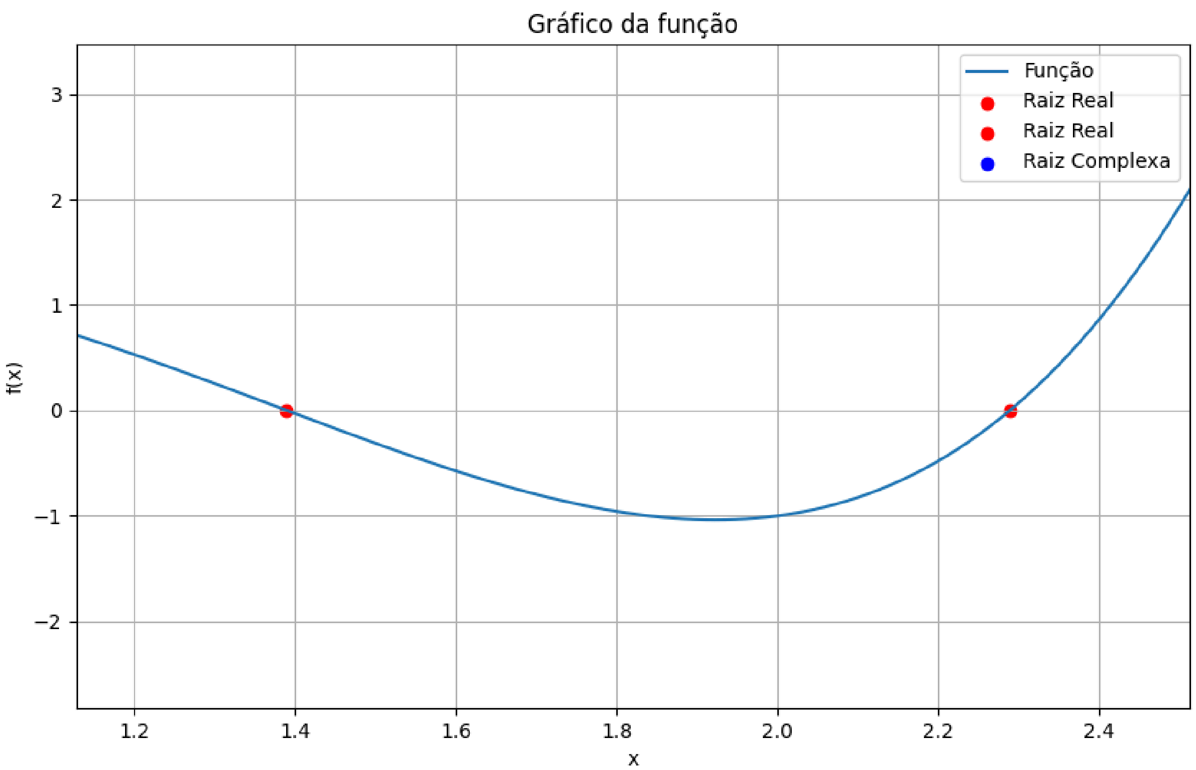
A conclusão que podemos tirar deles é que, para esse caso em específico, conseguimos ter um valor muito próximo para a raiz na quinta iteração do algoritmo, e cada vez mais que iteramos as raízes vão ficando mais precisas, principalmente a raiz real 1. Agora, analisando o resultado pelos gráficos.



Esse é o polinômio utilizado como exemplo no plano cartesiano, iremos analisar como o algoritmo se comporta ao executar o método com 10 iterações e utilizando -0.5, 0 e 0.5 para x0, x1 e x2 respectivamente, para encontrar a raiz complexa, e também foram definidos intervalos arbitrários entre (0.2, 1.5) e (1.6 e 2.5), para obter as raízes reais. Chegando assim no seguinte resultado



Podemos analisar que, após 3 iterações, as raízes reais aproximadas estão muito próximas de onde a função corta o eixo das abscissas, e como o gráfico fica impossibilitado de demonstrar a raiz complexa, como o algoritmo conseguiu encontrar as raízes reais, podemos concluir que ele foi igualmente capaz de encontrar a raiz complexa.



Concluímos que, após 10 iterações, neste caso em específico podemos alcançar valores para as raízes reais muito precisos, podemos então considerar que o valor da raíz complexa também apresenta essa precisão, já que utiliza do mesmo algoritmo.