Numerical Analysis Project Fernando Thaureaux: 6236069

Writing the Code

-Language: Python

-Input:

a - the start of interval [a, b] where f(a) and f(b) must have opposite signs

b - the end of interval [a, b] where f(a) and f(b) must have opposite signs

tol - the tolerance for root approximation, meaning the method should stop

max_iterations - makes sure the method does not run indefinitely

-Output: The root of a function (x) evaluated by using the Bisection method.

First, start with an interval [a, b] such that f(a) and f(b) have opposite signs to maintain the Intermediate Value Theorem (can be done analytically). Calculate the midpoint, check signs and update the interval, then calculate the error to determine whether the result is within desired tolerance.

Choice of Parameters

For a) f(x) = x^3 -5x^2 + 2x

[a, b] = [0.3, 0.5] and [4.5, 4.6] intervals contain root x = 0.438 and x = 4.56

For b) f(x) = x^3 -2x^2 - 5

[a, b] = [1, 3] interval contains root x = 2.69

Tol = 10^-4 is the desired tolerance, meaning error ≤ 10^-4

Max_iterations = 1000 is large enough to test convergence and stops infinite loop

Results and Analysis

Tables as output by the program for f(x) = x^3 - 5x^2 + 2x in [0.3, 0.5]

Iteration	Approximation	f(p)	Error
1	0.40000000	0.06400000	N/A
2	0.45000000	-0.02137500	0.05000000
3	0.42500000	0.02364062	0.02500000
4	0.43750000	0.00170898	0.01250000
5	0.44375000	-0.00968970	0.00625000
6	0.44062500	-0.00395444	0.00312500
7	0.43906250	-0.00111374	0.00156250
8	0.43828125	0.00029987	0.00078125
9	0.43867188	-0.00040637	0.00039063
10	0.43847656	-0.00005311	0.00019531
11	0.43837891	0.00012342	0.00009766

For interval [4.5, 4.6]

Iteration	Approximation	f(p)	Error

1 | 4.55000000 | -0.21612500 | N/A | 2 | 4.57500000 | 0.25448437 | 0.02500000 | 3 | 4.56250000 | 0.01782227 | 0.01250000 | 4 | 4.55625000 | -0.09948999 | 0.00625000 |
5 | 4.55937500 | -0.04091861 | 0.00312500 | 6 | 4.56093750 | -0.01156937 | 0.00156250 | 7 | 4.56171875 | 0.00312115 | 0.00078125 | 8 | 4.56132812 | -0.00422544 | 0.00039063 |
9 | 4.56152344 | -0.00055248 | 0.00019531 | 10 | 4.56162109 | 0.00128425 | 0.00009766 |

Tables as output by the program for f(x) = x^3 - 2x^2 - 5 in [1, 3]
Iteration | Approximation | f(p) | Error |
----
1 | 2.00000000 | -5.00000000 | N/A | 2 | 2.50000000 | -1.87500000 | 0.50000000 | 3 | 2.75000000 | 0.67187500 | 0.25000000 | 4 | 2.62500000 | -0.69335938 | 0.12500000 |
5 | 2.68750000 | -0.03442383 | 0.06250000 | 6 | 2.71875000 | 0.31271362 | 0.03125000 | 7 | 2.70312500 | 0.13765335 | 0.01562500 | 8 | 2.69531250 | 0.05124331 | 0.00781250 |
9 | 2.69140625 | 0.00831705 | 0.00390625 | 10 | 2.68945312 | -0.01307654 | 0.00195312 | 11 | 2.69042969 | -0.00238553 | 0.00097656 | 12 | 2.69091797 | 0.00296431 | 0.00048828 |
13 | 2.69067383 | 0.00028903 | 0.00024414 | 14 | 2.69055176 | -0.00104834 | 0.00012207 | 15 | 2.69061279 | -0.00037968 | 0.00006104 |

For the first function (a) in the interval [0.3, 0.5] the method converges to about 0.438 after 11 iterations, in the interval [4.5, 4.6] the method converges to about 4.56 after 10 iterations. Note that this function also has a root at x = 0, but this program fails to converge to it due to the IVT requiring opposite signs for f(a) and f(b), while this program is unable to handle negatives in [a, b]. For the second function (b) in the interval [1, 3] the method converges to about 2.69 after 15 iterations.
Both with the final error less than 10^-4 and decreasing steadily. The function f(p) evaluates very close to 0, meaning that the root is accurate.

Code

Bisection method:

```
# Function for the cubic polynomial f(x) = x^3 - 5x^2 + 2x (edit to f(x) = x^3 -2x^2 - 5 for function b)
def f(x):
  return x**3 - 5*x**2 + 2*x

# Bisection method implementation with f(p) and |P(n+1) - P(n)| as the error measure
def bisection_method_with_f_values(func, a, b, tol=1e-4, max_iterations=1000):
  iterations = []
```

```
    prev_c = None  # To store previous midpoint for error calculation

    if func(a) * func(b) >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    for i in range(max_iterations):
        c = (a + b) / 2  # Midpoint

        # Calculate error as |P(n+1) - P(n)|
        if prev_c is not None:
            error = abs(c - prev_c)
        else:
            error = None  # No error for the first iteration

        f_c = func(c)  # Calculate f(c) for the current midpoint

        iterations.append({
            'iteration': i + 1,
            'approximation': c,
            'f_value': f_c,
            'error': error
        })

        if error is not None and error < tol:  # Check if the error is within tolerance
            break

        prev_c = c  # Store the current midpoint for the next iteration

        # Update interval based on the sign of f(c)
        if func(c) == 0:
            break
        elif func(a) * func(c) < 0:
            b = c
        else:
            a = c

    return iterations

# Print results including f(p) for each iteration
def print_results_with_f(iterations, items_per_row=4):
    print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    for i in range(1, items_per_row):
        print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    print("\n" + "-" * (items_per_row * 50))
```

```
    for idx in range(0, len(iterations), items_per_row):
        for i in range(items_per_row):
            if idx + i < len(iterations):
                iteration = iterations[idx + i]
                f_value_str = f"{iteration['f_value']:12.8f}"
                error_str = f"{iteration['error']:12.8f}" if iteration['error'] is not None else "     N/A     "
                print(f"{iteration['iteration']:10d} | {iteration['approximation']:13.8f} | {f_value_str} |
{error_str}", end=" | ")
            else:
                break
        print()  # New line after each row

# Adjusted intervals that satisfy the bisection method condition
valid_intervals = [(0.3, 0.5), (4.5, 4.6)]  # For roots 0.438 and 4.56 (edit to (1, 3) for function b)

# Perform bisection method with the updated output (f(p)) for the valid intervals
valid_results_with_f = []
for interval in valid_intervals:
    a, b = interval
    result = bisection_method_with_f_values(f, a, b)
    valid_results_with_f.append(result)

# Print the results for each interval with f(p) and the updated error calculation
for i, result in enumerate(valid_results_with_f):
    print(f"Results for interval {valid_intervals[i]} (including f(p) and |P(n+1) - P(n)| as error):\n")
    print_results_with_f(result)
    print("\n\n")  # Separate the results for each interval
```

Writing the Code

-Language: Python

-Input:

func - the function f(x)

func_prime - the derivative of f(x)

initial_guess - an initial guess close to the root of the function

tol - the tolerance for root approximation, meaning the method should stop

max_iterations - makes sure the method does not run indefinitely

-Output: The root of a function (x) evaluated by using Newton's method.

First determine an initial guess close to the expected value, then take the function value of f(x) at each iteration, trying to find f(x) = 0, next evaluate the derivative to use the formula of Newton's method for the next approximation. Repeat this process until the error is within the range of tolerance.

Choice of Parameters

For a) f(x) = x^3 -5x^2 + 2x

func = f(x) = x^3 - 5x^2 + 2x

func_prime = f’(x) = 3x^2 - 10x + 2

initial_guess = 0.5 which is close to the root at about x=0.438

For b) f(x) = x^3 -2x^2 - 5

func = f(x) = x^3 -2x^2 - 5

func_prime = 3x^2 - 4x

initial_guess = 2 which is close to the root at about x=2.69

tol = 10^-4 is the desired tolerance, meaning error ≤ 10^-4

max_iterations = 1000 is large enough to test convergence and stops infinite loop

Results and Analysis

Tables as output by the program

Results for f(x) = x^3 - 5x^2 + 2x initial guess x = 0.5

Iteration | Approximation | f(p) | Error |

----
1 | 0.44444444 | -0.12500000 | 0.05555556 | 2 | 0.43851852 | -0.01097394 | 0.00592593 | 3 | 0.43844720 | -0.00012897 | 0.00007132 |

Initial guess x = 4 (trying for x = 4.56)

Iteration | Approximation | f(p) | Error |

----
1 | 4.80000000 | -8.00000000 | 0.80000000 | 2 | 4.58408304 | 4.99200000 | 0.21591696 | 3 | 4.56178360 | 0.42816316 | 0.02229944 | 4 | 4.56155284 | 0.00434110 | 0.00023077 | 5 | 4.56155281 | 0.00000046 | 0.00000002 |

Initial guess x = 0.1 (trying for x = 0)

Iteration | Approximation | f(p) | Error |
----
1 | -0.04660194 | 0.15100000 | 0.14660194 | 2 | -0.00447360 | -0.10416380 | 0.04212835 | 3 | -0.00004902 | -0.00904735 | 0.00442457 | 4 | -0.00000001 | -0.00009806 | 0.00004902 |

Results for f(x) = x^3 - 2x^2 - 5:

```
 Iteration | Approximation |        f(p) |       Error |
------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------
1 |    3.25000000 |  -5.00000000 |   1.25000000 |         2 |    2.81103679 |   8.20312500 |   0.43896321 |
3 |    2.69798950 |   1.40875418 |   0.11304729 |         4 |    2.69067715 |   0.08076844 |   0.00731235 |
5 |    2.69064745 |   0.00032546 |   0.00002970 |
```

For the first function (a) with initial guess x = 0.5 the method converges to about 0.438 after 3 iterations, with initial guess x = 4 the method converges to about 4.56 after 5 iterations, while initial guess x = 0.1 converges to the root at 0 after 4 iterations; so changing the initial guess will also change which root the method converges to. For the second function (b) with initial guess x = 2, the method converges to about 2.69 after 5 iterations. Newton's method is at least twice as fast as the bisection method, even with a small interval for the bisection, but it does rely on a closer initial approximation.
Both with the final error less than 10^-4 and decreasing rapidly. The function f(p) evaluates very close to 0, meaning that the root is accurate.

Newton's Method:

```
# Function for the cubic polynomial f(x) = x^3 - 5x^2 + 2x
def f1(x):
    return x**3 - 5*x**2 + 2*x

# Derivative of the cubic polynomial f'(x) = 3x^2 - 10x + 2
def f1_prime(x):
    return 3*x**2 - 10*x + 2

# Function for the cubic polynomial f(x) = x^3 - 2x^2 - 5
def f2(x):
    return x**3 - 2*x**2 - 5

# Derivative of the cubic polynomial f'(x) = 3x^2 - 4x
def f2_prime(x):
    return 3*x**2 - 4*x

# Newton's method implementation with |P(n+1) - P(n)| as the error measure and f(p) output
def newtons_method_with_f_values(func, func_prime, initial_guess, tol=1e-4, max_iterations=1000):
    iterations = []
    p_n = initial_guess  # Initial guess for the root

    for i in range(max_iterations):
        f_p_n = func(p_n)
        f_prime_p_n = func_prime(p_n)
```

```
        if f_prime_p_n == 0:
            raise ValueError(f"Derivative is zero at x = {p_n}. Newton's method fails.")

        # Compute the next approximation
        p_n_plus_1 = p_n - f_p_n / f_prime_p_n

        # Calculate error as |P(n+1) - P(n)|
        error = abs(p_n_plus_1 - p_n)

        iterations.append({
            'iteration': i + 1,
            'approximation': p_n_plus_1,
            'f_value': f_p_n,
            'error': error
        })

        # Check if the error is within tolerance
        if error < tol:
            break

        p_n = p_n_plus_1  # Update p_n for the next iteration

    return iterations

# Print results in a landscape format with the error calculation and f(p)
def print_newtons_results_with_f(iterations, items_per_row=5):
    print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    for i in range(1, items_per_row):
        print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    print("\n" + "-" * (items_per_row * 50))

    for idx in range(0, len(iterations), items_per_row):
        for i in range(items_per_row):
            if idx + i < len(iterations):
                iteration = iterations[idx + i]
                f_value_str = f"{iteration['f_value']:12.8f}"
                error_str = f"{iteration['error']:12.8f}" if iteration['error'] is not None else "     N/A     "
                print(f"{iteration['iteration']:10d} | {iteration['approximation']:13.8f} | {f_value_str} |
{error_str}", end=" | ")
            else:
                break
        print()  # New line after each row
```

```
# Test Newton's method for the first cubic function f(x) = x^3 - 5x^2 + 2x
print("Results for f(x) = x^3 - 5x^2 + 2x:\n")
initial_guess_f1 = 0.5  # Example initial guess (could also be x = 4)
results_f1 = newtons_method_with_f_values(f1, f1_prime, initial_guess_f1)
print_newtons_results_with_f(results_f1)

# Test Newton's method for the second cubic function f(x) = x^3 - 2x^2 - 5
print("\nResults for f(x) = x^3 - 2x^2 - 5:\n")
initial_guess_f2 = 2  # Example initial guess
results_f2 = newtons_method_with_f_values(f2, f2_prime, initial_guess_f2)
print_newtons_results_with_f(results_f2)
```

Writing the Code

-Language: Python

-Input:

func - the function f(x)

p0 - first point initial guess

p1 - second point initial guess

tol - the tolerance for root approximation, meaning the method should stop

max_iterations - makes sure the method does not run indefinitely

-Output: The root of a function (x) evaluated by using the Secant method.

Starting with 2 initial points, which are guesses fairly close to the root we want to approximate, then using the iterative method/iteration formula to compute the new approximation of the root.

Choice of Parameters

For a) f(x) = x^3 -5x^2 + 2x

func = f(x) = x^3 - 5x^2 + 2x

p0 = 0.5 first initial guess close to the root x = 0.438

p1 = 1.5 second initial guess close to the point

For b) f(x) = x^3 -2x^2 - 5

func = f(x) = x^3 -2x^2 - 5

p0 = 1 first initial guess

p1 = 2 second initial guess (close to root x = 2.69)

tol = 10^-4 is the desired tolerance, meaning error ≤ 10^-4

Results and Analysis

Tables as output by the program

Results for f(x) = x^3 - 5x^2 + 2x:
p0 = 0.5, p1 = 1.5

```
 Iteration | Approximation |        f(p) |       Error |
---------------------------------------------------------------------------------------------------------------------------
       1 |    0.47368421 |  -4.87500000 |   1.02631579 |         2 |    0.45911578 |  -0.06823152 |   0.01456843
|         3 |    0.43976113 |  -0.03892916 |   0.01935464 |         4 |    0.43849986 |  -0.00238166 |
0.00126127 |          5 |    0.43844733 |  -0.00009524 |   0.00005254 |
```

Results for initial guesses p0 = 4.4 and p1 = 4.6:

```
 Iteration | Approximation |        f(p) |       Error |
---------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
       1 |    4.55855856 |   0.73600000 |   0.04144144 |         2 |    4.56150030 |  -0.05623739 |   0.00294174
|         3 |    4.56155289 |  -0.00098760 |   0.00005258 |
```

Results for initial guess p0 = -0.1 and p1 = 0.1:

```
 Iteration | Approximation |        f(p) |       Error |
---------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
       1 |    0.02487562 |   0.15100000 |   0.07512438 |         2 |   -0.00873258 |   0.04667265 |   0.03360820
|         3 |    0.00056394 |  -0.01784711 |   0.00929652 |         4 |    0.00001208 |   0.00112629 |
0.00055186 |          5 |   -0.00000002 |   0.00002417 |   0.00001210 |
```

Results for f(x) = x^3 - 2x^2 - 5:
p0 = 1, p1 = 2

```
 Iteration | Approximation |        f(p) |       Error |
---------------------------------------------------------------------------------------------------------------------------
       1 |    7.00000000 |  -5.00000000 |   5.00000000 |         2 |    2.10204082 | 240.00000000 |
4.89795918 |          3 |    2.19315310 |  -4.54912494 |   0.09111228 |         4 |    2.96883707 |  -4.07094896 |
0.77568397 |          5 |    2.60808828 |   3.53932366 |   0.36074879 |
       6 |    2.67885351 |  -0.86370784 |   0.07076523 |         7 |    2.69120757 |  -0.12837330 |   0.01235406
|         8 |    2.69064377 |   0.00613873 |   0.00056380 |         9 |    2.69064745 |  -0.00004029 |
0.00000368 |
```

For the first function (a) with initial guess p0 = 0.5 and p1 = 1.5 the method converges to about 0.438 after 5 iterations, with initial guess p0 = 4.4 and p1 = 4.6 the method converges to about 4.56 after 3 iterations, while initial guess p0 = -0.1 and p1 = 0.1 converges to the root at 0 after 5 iterations; so changing the initial guess will also change which root the method converges to. For the second function (b) with initial guess p0 = 1 and p1 = 2, the method converges to about 2.69 after 9 iterations. The Secant method is

about as fast as Newton's method and faster than the bisection method, but it is worth noting that the initial guesses for this Secant algorithm could be much better, for example changing the points in function (b) to p0 = 2 and p1 = 3 converges in 5 iterations.
Both with the final error less than 10^-4 and decreasing rapidly. The function f(p) evaluates very close to 0, meaning that the root is accurate.

Secant Method:

```
# Function for the cubic polynomial f(x) = x^3 - 5x^2 + 2x
def f1(x):
    return x**3 - 5*x**2 + 2*x

# Function for the cubic polynomial f(x) = x^3 - 2x^2 - 5
def f2(x):
    return x**3 - 2*x**2 - 5

# Secant method implementation with |P(n+1) - P(n)| as the error measure, including f(p) output
def secant_method_with_f_values(func, p0, p1, tol=1e-4, max_iterations=1000):
    iterations = []

    for i in range(max_iterations):
        f_p0 = func(p0)
        f_p1 = func(p1)

        if f_p1 == f_p0:
            raise ValueError(f"f(P(n)) and f(P(n-1)) are equal at iteration {i+1}, division by zero will occur.")

        # Compute the next approximation using the secant method formula
        p2 = p1 - f_p1 * (p1 - p0) / (f_p1 - f_p0)

        # Calculate error as |P(n+1) - P(n)|
        error = abs(p2 - p1)

        iterations.append({
            'iteration': i + 1,
            'approximation': p2,
            'f_value': f_p1,
            'error': error
        })

        # Check if the error is within tolerance
        if error < tol:
            break
```

```
        # Update points for the next iteration
        p0, p1 = p1, p2

    return iterations

# Print results with f(p) and error calculation
def print_secant_results_with_f(iterations, items_per_row=5):
    print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    for i in range(1, items_per_row):
        print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}", end=" | ")
    print("\n" + "-" * (items_per_row * 50))

    for idx in range(0, len(iterations), items_per_row):
        for i in range(items_per_row):
            if idx + i < len(iterations):
                iteration = iterations[idx + i]
                f_value_str = f"{iteration['f_value']:12.8f}"
                error_str = f"{iteration['error']:12.8f}" if iteration['error'] is not None else "     N/A    "
                print(f"{iteration['iteration']:10d} | {iteration['approximation']:13.8f} | {f_value_str} |
{error_str}", end=" | ")
            else:
                break
        print()  # New line after each row

# Test Secant method for the first cubic function f(x) = x^3 - 5x^2 + 2x
print("Results for f(x) = x^3 - 5x^2 + 2x:\n")
initial_guess_f1_p0 = 0.5  # Example initial guess p0
initial_guess_f1_p1 = 1.5  # Example initial guess p1
results_f1 = secant_method_with_f_values(f1, initial_guess_f1_p0, initial_guess_f1_p1)
print_secant_results_with_f(results_f1)

# Test Secant method for the second cubic function f(x) = x^3 - 2x^2 - 5
print("\nResults for f(x) = x^3 - 2x^2 - 5:\n")
initial_guess_f2_p0 = 1  # Example initial guess p0
initial_guess_f2_p1 = 2  # Example initial guess p1
results_f2 = secant_method_with_f_values(f2, initial_guess_f2_p0, initial_guess_f2_p1)
print_secant_results_with_f(results_f2)
```

Writing the Code

-Language: Python

-Input:

func - the function f(x)

p0 - first point initial guess

p1 - second point initial guess
p2 - third point initial guess
tol - the tolerance for root approximation, meaning the method should stop
max_iterations - makes sure the method does not run indefinitely

-Output: The root of a function (x) evaluated by using Muller's method.

Starting with 3 initial points, which are guesses fairly close to the root we want to approximate, construct a quadratic function using the 3 points for each step, solve for the root using the quadratic formula, and select the root that gives a valid approximation based on the discriminant. Repeat this process until the error is less than 10^-4.

Choice of Parameters

For a) f(x) = x^3 -5x^2 + 2x
func = f(x) = x^3 - 5x^2 + 2x
p0 = 0.0 first initial guess close to the root x = 0.438
p1 = 0.5 second initial guess close to the point
p2 = 1.0 third initial guess

For b) f(x) = x^3 -2x^2 - 5
func = f(x) = x^3 -2x^2 - 5
p0 = 1.0 first initial guess
p1 = 2.0 second initial guess (close to root x = 2.69)
p2 = 3.0 third initial guess close to the root

tol = 10^-4 is the desired tolerance, meaning error ≤ 10^-4

Results and Analysis

Tables as output by the program

```
Results for f(x) = x^3 - 5x^2 + 2x:
p0 = 0, p1 = 0.5, p2 = 1.0
 Iteration | Approximation |       f(p) |       Error
--------------------------------------------------
        1 |    0.42857143 |  0.01749271 |  0.57142857
        2 |    0.43826122 |  0.00033605 |  0.00968979
        3 |    0.43844776 | -0.00000103 |  0.00018654
        4 |    0.43844719 | -0.00000000 |  0.00000057

Results near x = 4.56:
p0 = 4.4, p1 = 4.5, p2 = 4.6
 Iteration | Approximation |       f(p) |       Error
--------------------------------------------------
```

```
        1 |    4.56153248 |  -0.00038235 |   0.03846752
        2 |    4.56155281 |  -0.00000005 |   0.00002033
```

Results near x = 0:
p0 = -0.1, p1 = 0.1, p2 = 0.2

```
 Iteration | Approximation |        f(p) |       Error
--------------------------------------------------
        1 |    0.00099740 |   0.00198983 |   0.19900260
        2 |   -0.00001008 |  -0.00002015 |   0.00100748
        3 |    0.00000000 |   0.00000000 |   0.00001008
```

Results for f(x) = x^3 - 2x^2 - 5:

```
 Iteration | Approximation |        f(p) |       Error
--------------------------------------------------
        1 |    2.65586885 |  -0.37373772 |   0.34413115
        2 |    2.68998120 |  -0.00729688 |   0.03411235
        3 |    2.69064679 |  -0.00000716 |   0.00066560
        4 |    2.69064745 |   0.00000000 |   0.00000065
```

For the first function (a) with initial guess p0 = 0.0, p1 = 0.5, and p2 = 1.0 the method converges to about 0.438 after 4 iterations, with initial guesses p0 = 4.4, p1 = 4.5, and p2 = 4.6, the method converges to about 4.56 after 2 iterations, while initial guess p0 = -0.1, p1 = 0.1, and p2 = 0.2 converges to the root at 0 after 3 iterations; so changing the initial guess will also change which root the method converges to. For the second function (b) with initial guess p0 = 1, p1 = 2, and p2 = 3, the method converges to about 2.69 after 4 iterations. Muller's method is extremely fast, seemingly faster than the previous, but it is worth noting that the initial guesses for this algorithm were very good, very close to the values of the roots.
Both with the final error less than 10^-4 and decreasing rapidly. The function f(p) evaluates very close to 0, meaning that the root is accurate.

Muller's Method:

```
import math

# Function for the cubic polynomial f(x) = x^3 - 5x^2 + 2x
def f1(x):
   return x**3 - 5*x**2 + 2*x

# Function for the cubic polynomial f(x) = x^3 - 2x^2 - 5
def f2(x):
```

```
    return x**3 - 2*x**2 - 5

# Muller's method implementation with |P(n+1) - P(n)| as the error measure
def mullers_method_with_f_values(func, p0, p1, p2, tol=1e-4, max_iterations=100):
    iterations = []

    for i in range(max_iterations):
        # Evaluate the function at the three points
        f_p0 = func(p0)
        f_p1 = func(p1)
        f_p2 = func(p2)

        # Calculate the differences
        h0 = p1 - p0
        h1 = p2 - p1
        d0 = (f_p1 - f_p0) / h0
        d1 = (f_p2 - f_p1) / h1

        # Calculate the coefficients of the quadratic
        a = (d1 - d0) / (h1 + h0)
        b = a * h1 + d1
        c = f_p2

        # Calculate the discriminant
        discriminant = math.sqrt(b**2 - 4 * a * c)

        # Use the sign of b to determine which root to use
        if abs(b + discriminant) > abs(b - discriminant):
            denominator = b + discriminant
        else:
            denominator = b - discriminant

        # Calculate the next approximation
        p3 = p2 - (2 * c) / denominator

        # Calculate error as |P(n+1) - P(n)|
        error = abs(p3 - p2)

        # Store the iteration results
        iterations.append({
            'iteration': i + 1,
            'approximation': p3,
            'f_value': func(p3),
            'error': error
```

```
        })

        # Check if the error is within the tolerance
        if error < tol:
            break

        # Update the points for the next iteration
        p0, p1, p2 = p1, p2, p3

    return iterations

# Print results with f(p) and error calculation
def print_muller_results_with_f(iterations):
    print(f"{'Iteration':>10} | {'Approximation':>13} | {'f(p)':>12} | {'Error':>12}")
    print("-" * 50)
    for iteration in iterations:
        f_value_str = f"{iteration['f_value']:12.8f}"
        error_str = f"{iteration['error']:12.8f}"
        print(f"{iteration['iteration']:10d} | {iteration['approximation']:13.8f} | {f_value_str} | {error_str}")

# Test Muller's method for the first cubic function f(x) = x^3 - 5x^2 + 2x
print("Results for f(x) = x^3 - 5x^2 + 2x:\n")
initial_guess_f1_p0 = 0.0  # Example initial guess p0
initial_guess_f1_p1 = 0.5  # Example initial guess p1
initial_guess_f1_p2 = 1.0  # Example initial guess p2
results_f1 = mullers_method_with_f_values(f1, initial_guess_f1_p0, initial_guess_f1_p1,
initial_guess_f1_p2)
print_muller_results_with_f(results_f1)

# Test Muller's method for the second cubic function f(x) = x^3 - 2x^2 - 5
print("\nResults for f(x) = x^3 - 2x^2 - 5:\n")
initial_guess_f2_p0 = 1.0  # Example initial guess p0
initial_guess_f2_p1 = 2.0  # Example initial guess p1
initial_guess_f2_p2 = 3.0  # Example initial guess p2
results_f2 = mullers_method_with_f_values(f2, initial_guess_f2_p0, initial_guess_f2_p1,
initial_guess_f2_p2)
print_muller_results_with_f(results_f2)
```