+2 for presentation

CS375 HW3

Ryan Scherbarth, University of New Mexico

September 2024

1. Root finding Algorithms

(a) Bisection Method

```
function x_arr = bisection(fnc, a, b, tol, Nmax)
    x_arr = [];
    n = 0;
    epsilon_m = eps;

    % Bisection
    while n < Nmax
        n = n + 1;
        % Compute the midpoint
        c = (a + b) / 2;
        x_arr(end+1) = c;

        if abs(b - a) < tol || abs(fnc(c)) < epsilon_m
            break;
        end

        if fnc(a) * fnc(c) < 0
            b = c;
        else
            a = c;
        end
    end

    if n == Nmax
        disp('Hit NMax.');
    end
end
```

(b) Secant Method

```
function x_arr = secant(fnc, a, b, tol, Nmax)
    x_arr = [a, b];
    n = 2;
    epsilon_m = eps;

    while n < Nmax
        xk = x_arr(end);
        xk_1 = x_arr(end-1);

        f_xk = fnc(xk);
        f_xk_1 = fnc(xk_1);

        % Prevent division by a very small number
        if abs(f_xk - f_xk_1) < epsilon_m
            break;
        end

        x_next = xk - (f_xk * (xk - xk_1)) / (f_xk - f_xk_1);

        x_arr(end+1) = x_next;

        if abs(x_next - xk) < tol || abs(fnc(x_next)) < epsilon_m
            break;
        end

        n = n + 1;
    end

    if n == Nmax
        disp('Hit NMax.');
    end
end
```

(c) Newton's Method

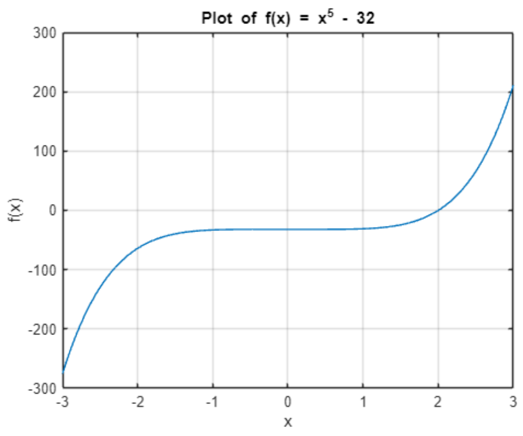
```
function x_arr = newton(fnc, df, a, tol, Nmax)
    x_arr = a;
    n = 1;
    epsilon_m = eps;

    while n < Nmax
        f_xk = fnc(x_arr(end));
        df_xk = df(x_arr(end));

        % Check for bad values
        if abs(df_xk) < epsilon_m
            disp('dx too small.');
            break;
        end

        x_next = x_arr(end) - f_xk / df_xk;

        x_arr(end+1) = x_next;

        if abs(x_next - x_arr(end-1)) < tol || abs(fnc(x_next)) < epsilon_m
            break;
        end

        n = n + 1;
    end

    if n == Nmax
        disp('Hit NMax.');
    end
end
```

-8

There is a bug in your Newton cod

2. Finding roots

(a) $p = 5,\ a = 32$



There is only a single root here.
It is a simple root.
We should use the interval [1..3]

(b) Evaluating with root finding algorithms

Method	**root**	**value**	**Iterations**
Bisection	2	-1.1642e-09	36
Secant	2	0	20
Newton	2	0	9

3. Newton's Method to find root of $f(x) = 8x(x - \frac{1}{2})^3$

(a) Find the positive root of $f(x)$ using Newton's method

Initial guess: 0.6
Root: 0.5
function value: 9.6134e-17
Iterations: 27

(b) Report computed iterates x_k and error ratios: $\frac{e_{k+1}}{e_k^2}$

Iteration	**Iterate** x_k	**Error Ratio:** $\frac{e_{k+1}}{e_k^2}$
1	0.5000	1.5489×10^5
2	0.5000	1.8587×10^5
3	0.5000	2.2305×10^5
4	0.5000	2.6766×10^5
5	0.5000	3.2120×10^5
6	0.5000	3.8544×10^5
7	0.5000	4.6254×10^5
8	0.5000	5.5505×10^5
9	0.5000	6.6606×10^5
10	0.5000	7.9928×10^5
11	0.5000	9.5914×10^5

The error ratio $\frac{e_{k+1}}{e_k^2}$ appears to be growing for each of the final 10 iterates. This suggests that Newton's method is not converging quadratically for this particular function due to the small derivative near the root, which slows down the convergence.

(c) Report computed iterates x_k and error ratios: $\dfrac{e_{k+1}}{e_k}$

Iteration	**Iterate** x_k	**Error Ratio:** $\frac{e_{k+1}}{e_k}$
1	0.5000	0.8333
2	0.5000	0.8333
3	0.5000	0.8333
4	0.5000	0.8333
5	0.5000	0.8333
6	0.5000	0.8333
7	0.5000	0.8333
8	0.5000	0.8333
9	0.5000	0.8333
10	0.5000	0.8333

These ratios are a little large...

The error ratio $\frac{e_{k+1}}{e_k}$ appears to stabilize around 0.8333 for each of the final 10 iterates. This suggests that Newton's method is converging linearly for this particular function as the ratio is bounded by a constant $C < 1$, indicating that the convergence rate has slowed down, but remains stable.

4. Consider the function $f(x) = e^{sin^3(x)} + x^6 - 2x^4 - x^3 - 1$ on the interval $[-2, 2]$. this function has three roots. One root is $r = 0$, the others are difficult to find exact formulas for.

(a) Do some initial matlab things etc.

optimset('TolX', 3e-16)

was used to set the tolerance on x.

root1 = fzero(f, -1.5, optimset('TolX', 3e-16))

where f is defined as the given function. We do the same for root2, and we get the following;

root1 $= -1.1976237221335699$
root2 $= 1.5301335081666154$

(b) Now use root-finding algorithms from prob 1 on the given function.

-1

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
1	0.0000000000000000	0.0000000000000000e+00	N/A

Table 1: Bisection Method for Root r0

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
15	0.0959674721116183	9.5967472111618254e-02	1.0420553306506743e+01
16	0.0786579553561796	7.8657955356179574e-02	1.2712960264235123e+01
17	0.0644727338115587	6.4472733811558741e-02	1.5509760477939359e+01
18	0.0528443866085141	5.2844386608514121e-02	1.8922424448092574e+01
19	0.0433114589580795	4.3311458958079516e-02	N/A
20	0.0354962444459980	3.5496244445997971e-02	N/A

Table 2: Secant Method for Root r0

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
15	-1.1976237255539619	1.1976237255539619e+00	8.3498680006175297e-01
16	-1.1976237224989903	1.1976237224989903e+00	8.3498680049254903e-01
17	-1.1976237221726098	1.1976237221726098e+00	8.3498680053857310e-01
18	-1.1976237221377408	1.1976237221377408e+00	8.3498680054349050e-01
19	-1.1976237221340154	1.1976237221340154e+00	N/A
20	-1.1976237221336175	1.1976237221336175e+00	N/A

Table 3: Newton's Method for Root r0

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
15	-1.1975708007812500	5.2921352319934556e-05	7.9994351150634293e+03
16	-1.1976013183593750	2.2403774194934556e-05	1.4235048652043262e+04
17	-1.1976165771484375	7.1449851324345559e-06	9.4887696138072388e+03
18	-1.1976242065429688	4.8440939881544409e-07	1.4192426478162300e+07
19	-1.1976203918457031	3.3302878668095559e-06	1.2829886189717129e+05
20	-1.1976222991943359	1.4229392339970559e-06	N/A

Table 4: Bisection Method for Root r1

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
1	-2.0000000000000000	8.0237627786643007e-01	8.3498680054407848e-01
2	0.0000000000000000	1.1976237221335699e+00	N/A
3	0.0000000000000000	1.1976237221335699e+00	N/A

Table 5: Secant Method for Root r1

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
15	-1.1976237221554544	2.1884494216806161e-11	4.5702298822454636e+10
16	-1.1976237221359078	2.3379076452556546e-12	4.2700796466920258e+11
17	-1.1976237221338197	2.4980018054066022e-13	4.0657496635983643e+12
18	-1.1976237221335966	2.6645352591003757e-14	2.6648518505150863e+13
19	-1.1976237221335728	2.8865798640254070e-15	N/A
20	-1.1976237221335702	2.2204460492503131e-16	N/A

Table 6: Newton's Method for Root r1

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
15	1.5300903320312500	4.3176135365419199e-05	6.7904206165709293e+03
16	1.5301208496093750	1.2658557240419199e-05	1.6227202262326975e+04
17	1.5301361083984375	2.6002318220808007e-06	7.4382633214214514e+05
18	1.5301284790039062	5.0291627091691993e-06	4.8016863547232926e+04
19	1.5301322937011719	1.2144654435441993e-06	4.6977478325306240e+05
20	1.5301342010498047	6.9288318926830073e-07	N/A

Table 7: Bisection Method for Root r2

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
1	0.0000000000000000	1.5301335081666154e+00	6.9307522767671133e+00
2	2.0000000000000000	4.6986649183338458e-01	N/A
3	0.0000000000000000	1.5301335081666154e+00	N/A

Table 8: Secant Method for Root r2

Iteration	**Iterate** x_k	**Error** e_k	$\frac{e_{k+1}}{e_k^2}$
9	1.5301335084519443	2.8532887164089971e-10	3.5103065562349358e+09
10	1.5301335081604497	6.1657345895582694e-12	1.4962083584609665e+11
11	1.5301335081667489	1.3344880755994382e-13	3.1274997412295109e+13
12	1.5301335081666128	2.6645352591003757e-15	0.0000000000000000e+00
13	1.5301335081666156	2.2204460492503131e-16	N/A
14	1.5301335081666154	0.0000000000000000e+00	N/A

Table 9: Newton's Method for Root r2