Newton's Method, Error, Newton's Fractal and Explorations

David Tran and Spencer Kelly

February 11, 2024

Abstract

In the following, we explore and provide an implementation of Newton's method for both simple roots and roots of multiple multiplicity. We discuss how to determine the order of convergence, as well as how to approximate the order of convergence for functions whose roots are unknown. Also, due to differences in convergence for roots of multiple multiplicity, we show a modified version of Newton's method which can speed up convergence up to the expected quadratic speed for multiplicity $m > 1$. In the final sections, we show how Newton's method can be used to generate Newton's fractals, and generate fractals for the fourth roots of unity and the Mandelbrot set.

1 Introduction

(See References section for references used here.)

This is the second lab in a series of 4 labs exploring various numerical algorithms and analyzing their properties. Finding roots of polynomials has been a significant area of interest throughout the entirety of the history of mathematics. As evidence to this claim, consider the fact that entire branches of mathematics were born out of the desire to find roots of polynomials. For example, group theory takes its roots from Galois' proof of the insolvability of the quintic.

In this lab, we discuss Newton's method, an algorithm discovered by Issac Newton and Joseph Raphson. The algorithm comes from a family of iterative root-finding algorithms: algorithms which generate successively better approximations of the root of a function. Another type of root-finding algorithms are bracketing methods, which successively narrow the range in which a root may be found until a desired precision. However, many of these methods, such as the bisection method, converges much slower than Newton's method (e.g., only a single extra bit of precision for each iteration in the case of the bisection method), so algorithms such as Newton's method are often used in practice.

Newton's method is especially important due to the prevalance of the problem of finding roots in numerous applications. In particular, it is useful for when only an approximate solution to an equation is needed, especially when the exact solution would require far more computational power. For example, consider the problem of rendering the shadows of certain voxels in the context of computer graphics. To the naked eye (and more importantly, the resolution of the screen), only a finite amount of precision is needed to calculate the reflection and physics of light for a realistic image. Thus, we see Newton's method effectively being used to calculate raytracing and pixel shadowing for millions, sometimes billions, of voxels, which would otherwise be computationally intractable if we desired exact solutions.

2 Newton's Method

2.1 Implementation

Newton's method is a fixed-point iteration method used to approximate the roots of a real-valued function. We implement a method that returns the iterates

```
function [x,flag] = mynewton(f,fx,x0,tol,maxiter)
% Author : Spencer Kelly and David Tran
% Date : 2024.02.11
% Purpose : Compute approximate solution to f(x)=0 via Newton's Method
%
% Inputs :
% f -- A function handle for f(x) being solved
% fx -- A function handle for the f'(x)
% x0 -- Initial guess for the fixed point
% tol -- Tolerance of the solution .
% maxiter -- Maximum number of iterations .
%
% Outputs :
% x -- Vector containing the iterates of the Newton's Method
% flag -- Flag specifying if the solution has been obtained :
% = The number of iterations taken to converge .
% = -1 If the algorithm has not converged in maxiter iterations .
%
flag = -1;                                  %give inital value of flag and xold
x(1) = x0;
for i = 1:maxiter                           %for loop do the iteration with '
    maxiter' given for the maximum steps
    x(i+1) = x(i) - f(x(i))/fx(x(i));       %Newton
    %fprintf('%f\n',xs);
    if abs(x(i+1)-x(i)) <= tol              %if x_new - x_old <= tol, break the
    loop, let flag = # of steps
        flag = i;
        break
    end                                     %if not, do next iteration
end
end
```

mynewton.m

2.2 Testing

We test the method with the function:

$$f(x) = (x+1)(x-1/2)$$

for which we expect the roots $x = -1$ and $x = 1/2$. The derivative is

$$f'(x) = 2x + 1/2$$

so we test with the code below which returns the iterates as follows

```
f = @(x)(x+1).*(x-1/2);
df = @(x)2*x+1/2;
[x1,fl]=mynewton(f,df,-1.2,0.001,10);
[x2,fl]=mynewton(f,df,0.6,0.001,10);
```

```
% the roots are x1 = -1 and x2 = 0.5, we expect approximately these values
disp(x1);
disp(x2);
```

test_mynewton.m

$$x1 = [-1.2000, -1.0211, -1.0003, -1.000]$$
$$x2 = [0.6000, 0.5059, 0.5000, 0.5000]$$

which we see properly approaches -1 and 0.5.

2.3 Determining Order of Convergence

We use the code in Figure 1 to plot the logarithms of the errors against each other for $x = -1$ in Figure 2 and $x = 1/2$ in Figure 3.

$$\log_e |e_{k+1}| \approx \alpha \log_e |e_k| + \log_e(C)$$

Estimating a line-of-best-fit for both plots as seen in Figure 4 and Figure 5 shows an approximate slope $\alpha = 2$, corresponding to a quadratic order of convergence.

This matches the expected value of 2 that we should observe for Newton's method, since we expect it to have quadratic convergence. We expect this since Newton's method approximates the next iteration x_{n+1} with the solution to

$$0 \approx f(x_{n+1}) \approx f(x_n) + f'(x_n)(x_{n+1} - x_n)$$

which, solving for $x_{n+1} - x_n$ and taking the error at iteration n as $e_n = r - x_n$ (where r is the root), gives

$$e_{n+1} \approx -\frac{f(r - e_n)}{f'(r - e_n)}$$

and thus Taylor-expanding the right-hand side demonstrates a quadratic relationship between e_{n+1} and e_n for some constant C dependent on the second derivative of f:

$$e_{n+1} \approx Ce_n^2.$$

2.4 Approximating Order of Convergence

Determining the order of convergence is only possible when we know in advance the exact roots. However, often this is not the case. To detect quadratic convergence using only the information provided from our iterative approximates, we can compute the ratios of consecutives differences between iterates x_n and x_{n+1}. If the ratios are constant, then we are most likely observing quadratic convergence. This is a result of the fact that the second derivative of a quadratic function is constant.

We use the code in Figure 6 to demonstrate the ratios for quadratic convergence of the original function of interest $f(x) = (x + 1)(x - 1/2)$ in Figure 7. We use an initial guess far from the true value, $x_0 = -1000$ to show that the ratios are constant as we converge. Notice how the ratios are constant around 0.5 (they fall rapidly towards 0 as we approach the exact root of -1.)

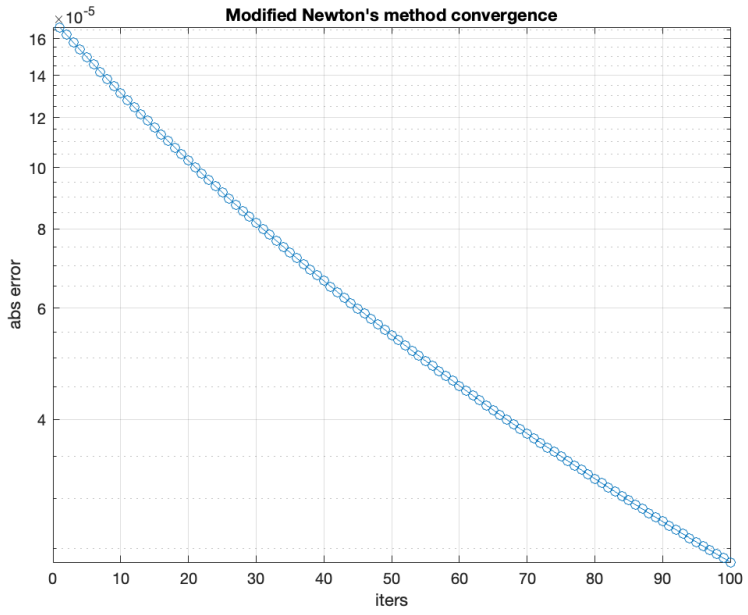
If instead our root has multiplicity greater than 1, quadratic convergence is not guaranteed, and the method will most likely converge slower. To detect when converging to a multiple root one can observe the convergence and see if convergence is linear, or even slower. To approximately

determine the multiplicity m of such a root, one can use the fact that a root of multiplicity m converges m times slower than a simple root. Thus, a root of multiplicity 2 converges linearly, a root of multiplicity 3 converges sublinearly (square-root), etc. So, we can observe the higher-order ratios, and observe when their behaviour (either quadratic, linear, sublinear, etc.), to approximately determine the multiplicity.

If the multiplciity is known, say m, we can modify Newton's method as the following to converge to the multiple root with quadratic convergence:

$$x_{n+1} = x_n - m\left(\frac{f(x_n)}{f'(x_n)}\right)$$

See the implementation in Figure 8 which results in the following convergence for a $f(x) = (x-2)^3$, a function with root of multiplicity 3.



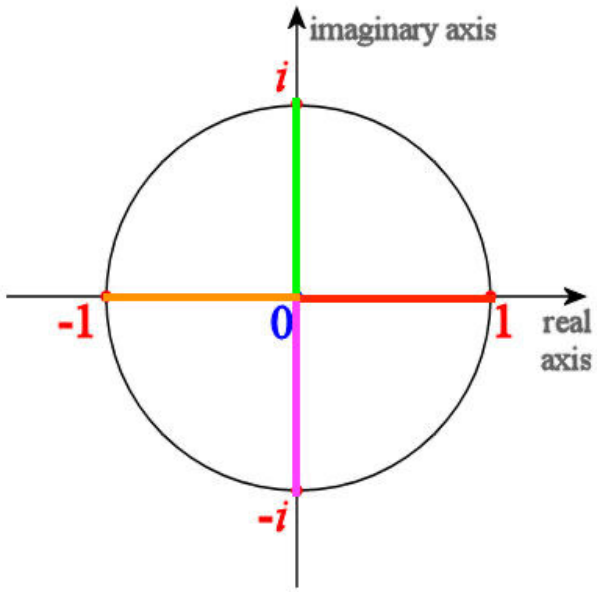
3 Newton's Fractals

3.1 Fourth Roots of Unity

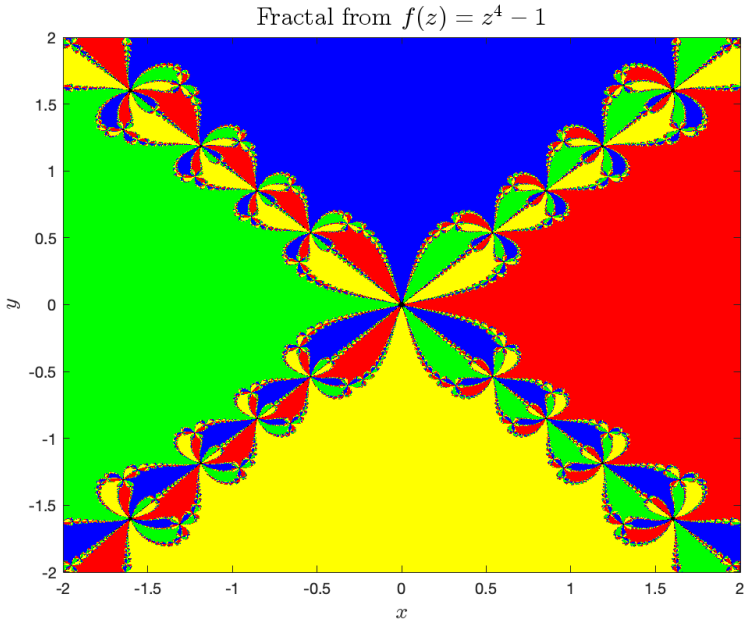
The solutions to $z^4 = 1$ can be solved using Euler's formula: since $\exp(i\theta) = \cos\theta + i\sin\theta$, $z^4 = 1$ is equivalent to $z^4 = \exp(i2\pi n)$ with n an integer. Taking the fourth root gives $z = \exp(i2\pi n/4)$, which for $n = 0, 1, 2, 3$ yields the roots

$$r_1 = 1$$
$$r_2 = i$$
$$r_3 = -1$$
$$r_4 = -i$$

and can be graphed on the complex plane by the red, green, orange, and pink lines, respectively below.



3.2 Fractal for the Fourth Roots of Unity



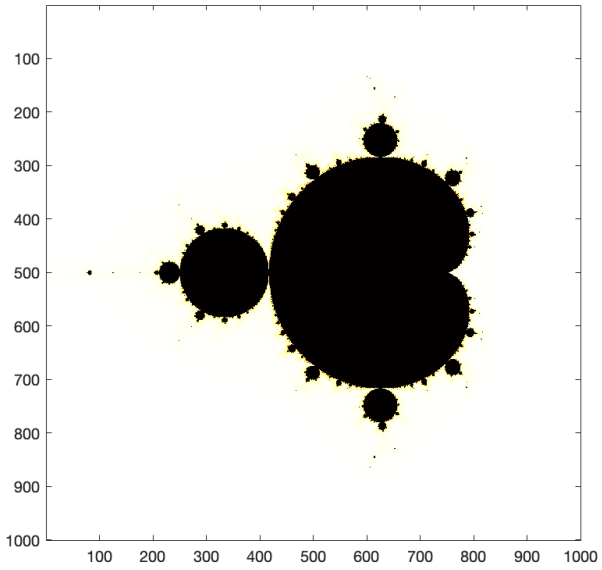

```
clear
clc
f = @(z) z.^4-1; fp = @(z) 4*z.^3;
root1 = 1; root2 = -1; root3 = 1i; root4 = -1i;

nx=2000; ny=2000;
xmin=-2; xmax=2; ymin=-2; ymax=2;

x=linspace(xmin,xmax,nx); y=linspace(ymin,ymax,ny);
[X,Y]=meshgrid(x,y);
```

```
Z=X+1i*Y;

nit=40;
for n=1:nit
Z = Z - f(Z) ./ fp(Z);
end

eps=0.001;
Z1 = abs(Z-root1) < eps; Z2 = abs(Z-root2) < eps;
Z3 = abs(Z-root3) < eps; Z4 = abs(Z-root4) < eps;
Z5 = ~(Z1+Z2+Z3+Z4);

figure;
map = [1 0 0 ; 0 1 0 ; 0 0 1; 1 1 0; 0 0 0]; colormap(map);
Z=(Z1+2*Z2+3*Z3+4*Z4+5*Z5);
image([xmin xmax], [ymin ymax], Z); set(gca,'YDir','normal');
xlabel('$x$', 'Interpreter', 'latex', 'FontSize',14);
ylabel('$y$', 'Interpreter', 'latex', 'FontSize',14);
title('Fractal from $f(z)=z^4-1$', 'Interpreter', 'latex','FontSize', 16)
```

z4.m

3.3 Mandelbrot Fractal

Using a similar approach, we demonstrate the Mandelbrot fractal, a fractal generated using the iteration of $f_c(z) = z^2 + c$.



4 Summary

4.1 Results

In conclusion, our exploration of Newton's method has covered its application to both simple roots and roots of multiple multiplicity. We delved into methods for determining the order of

convergence and outlined approaches to approximate the order when dealing with functions whose roots are unknown. Recognizing the distinct convergence behavior for roots of multiple multiplicity, we introduced a modified version of Newton's method tailored to accelerate convergence up to the anticipated quadratic speed for roots with a multiplicity $m > 1$.

In the final sections, we showcased the broader applications of Newton's method, extending beyond root finding to the realm of fractals. Specifically, we demonstrated how Newton's method can be harnessed to generate fractals, illustrating its significance in both mathematical exploration and visualization.

4.2 Team Description

Our team encountered problems with the convergence of Newton's method: we did not initially set a tolerance for when to halt the iteration. Thus, Newton's method would run for much longer than needed, attempting to find the exact value of the root. While we initially believed it was due to bad performance of the code, our profiling revealed that we were iterating much more than needed. Adding tolerance to the function let us specify the desired precision of the iteration method.

4.3 Future Explorations

Our team would like to further explore different fractals that can be generated using Newton's method. It would be interesting to see fractals of not just polynomial functions, but also of transcendental functions. We would like to see if there are any discernible differneces between the fractal images of functions according to their properties, such as if they are analytic, holomorphic, etc.

4.4 References

1. Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing (3rd ed.)*. New York: Cambridge University Press. ISBN 978-0-521-88068-8.

2. Atkinson, Kendall E. (1989). *An Introduction to Numerical Analysis*. John Wiley & Sons, Inc. ISBN 0-471-62489-6.

3. Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing (3rd ed.)*. New York: Cambridge University Press. ISBN 978-0-521-88068-8. See especially Sections 9.4, 9.6, and 9.7.

4. Blinn, Jim (July 1997). "Floating Point Tricks". *IEEE Computer Graphics & Applications*. 17 (4): 80. doi:10.1109/38.595279.

5. Eberly, David (2001). *3D Game Engine Design*. Morgan Kaufmann. ISBN 978-1-55860-593-0.

Appendix

```
clear
clc
f = @(x)(x+1).*(x-1/2);
df = @(x)2*x+1/2;
profile on
[x1 , fl1] = mynewton (f ,df , -1.2 ,0.001 ,10)
[x2 , fl2] = mynewton (f ,df ,0.6 ,0.001 ,10)
profile off
profile viewer
for i = 1: fl1
    ex(i) = abs(x1(i)+1);                          %error for |x_n-x|
    ex1(i) = abs(x1(i+1)+1);                       %error for |x_(n+1)-x|
    logex(i) = log(abs(x1(i)+1));                  %error after log for |x_n-x|
    logex1(i) = log(abs(x1(i+1)+1));               %error after log for |x_x(n+1)-
   x|
end

% Compute errors for the second set of iterations (x2)
for i = 1:fl2
    ex2(i) = abs(x2(i) - 0.5);  % Adjust the expected root value if necessary
    ex3(i) = abs(x2(i + 1) - 0.5);
    logex2(i) = log(abs(x2(i) - 0.5));
    logex3(i) = log(abs(x2(i + 1) - 0.5));
end

% Plot errors for the first set of iterations
figure(1)
plot(logex, logex1, '.', 'MarkerSize', 15)
title('Convergence for Root x = -1')
xlabel('log(|x_n + 1|)')
ylabel('log(|x_{n+1} + 1|)')

% Plot errors for the second set of iterations
figure(2)
plot(log(ex2), log(ex3), '.', 'MarkerSize', 15)
title('Convergence for Root x = 0.5')
xlabel('log(|x_n - 0.5|)')
ylabel('log(|x_{n+1} - 0.5|)')
```

newtontest.m

Figure 1: newtontest.m

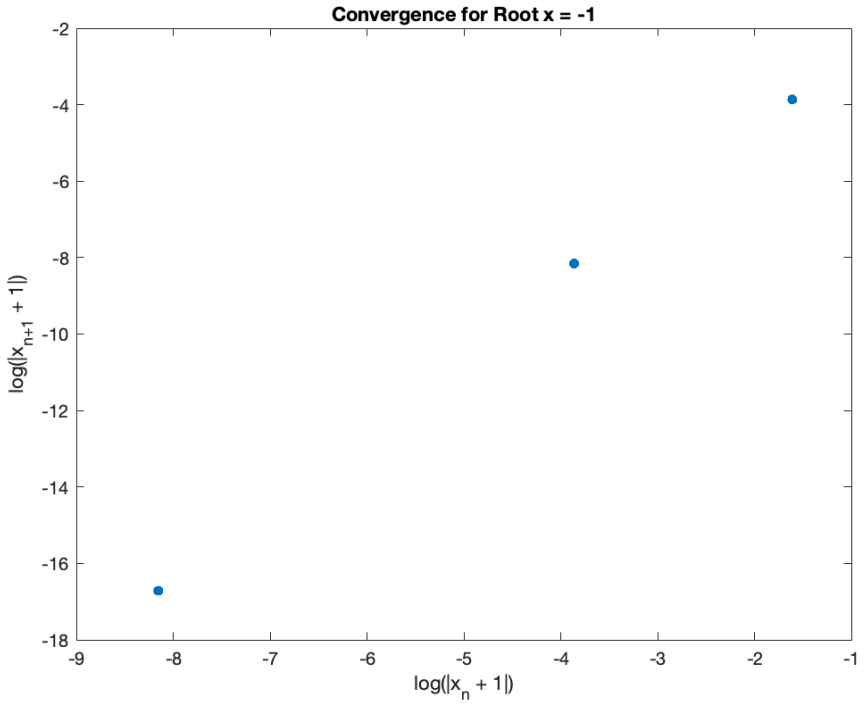


Figure 2: Log errors for $x = -1$

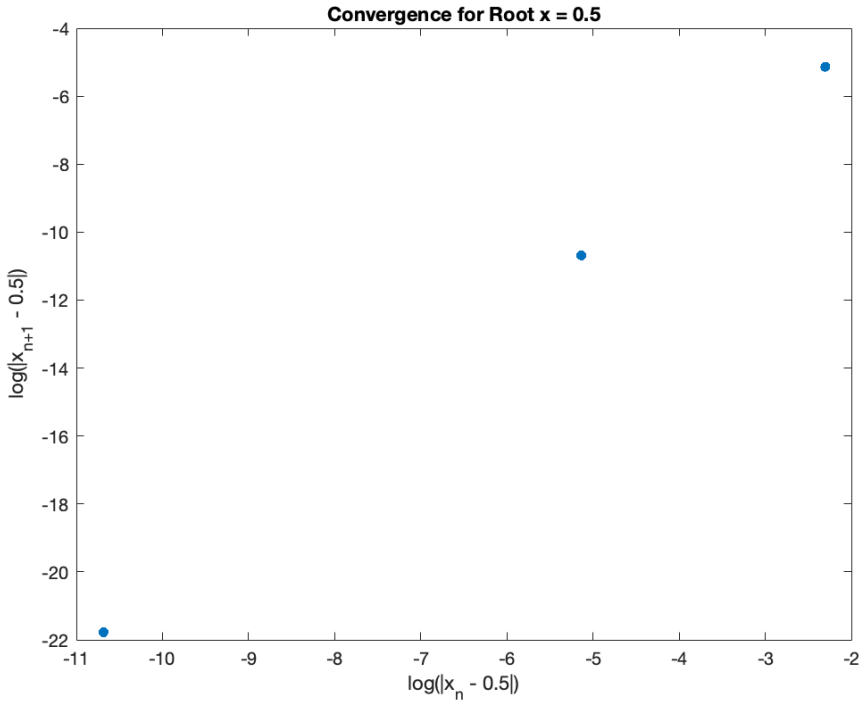


Figure 3: Log errors for $x = 1/2$

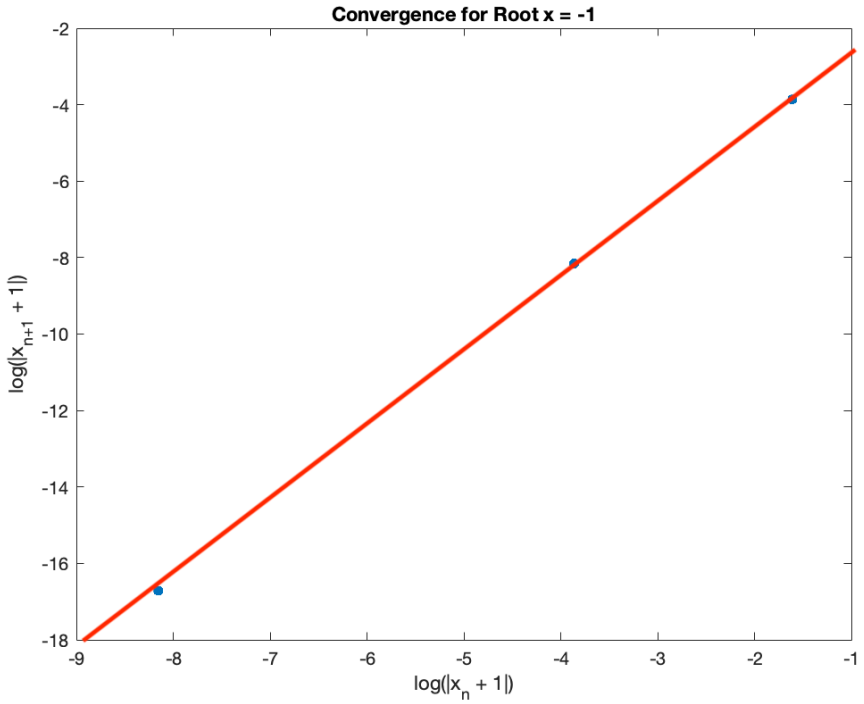


Figure 4: Log errors for $x = -1$

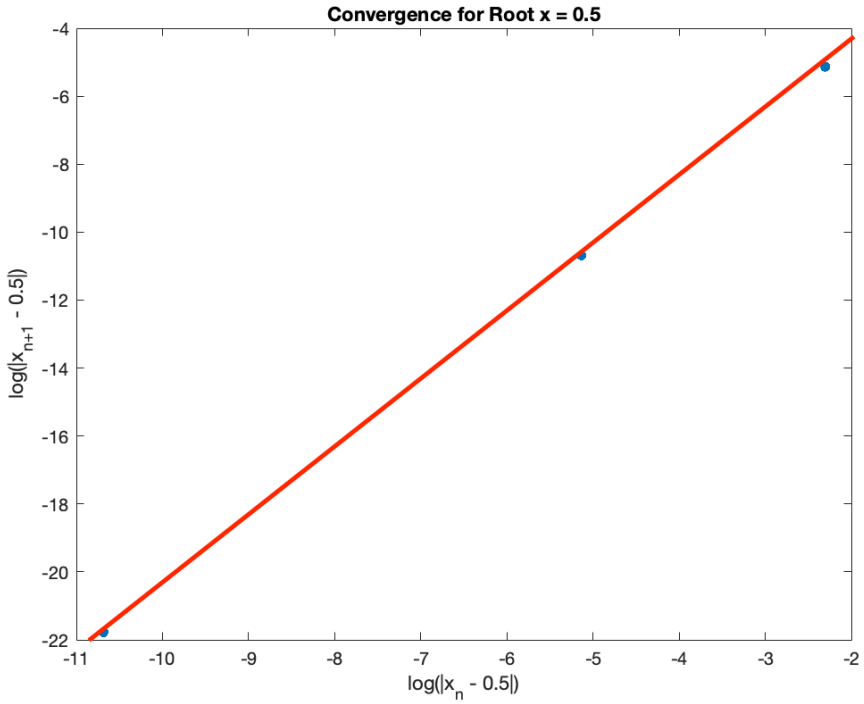


Figure 5: Log errors for $x = 1/2$

```
clear
clc

% Define the function and its derivative
f = @(x) (x + 1) .* (x - 1/2);
df = @(x) 2 * x + 1/2;

profile on
% Perform Newton's method
[x, ~] = mynewton(f, df, -100, 1e-10, 100);

% Compute differences
d = diff(x);

% Compute ratios
r = d(2:end) ./ d(1:end-1);
profile off
profile viewer

% Plot ratios
figure;
plot(r, 'o-', 'MarkerSize', 8);
title('Difference Ratios for Initial Guess x0 = -1000');
xlabel('Iteration');
ylabel('Ratio (r_k)');
grid on;
```

estimate_convergence.m

Figure 6: estimate_convergence.m

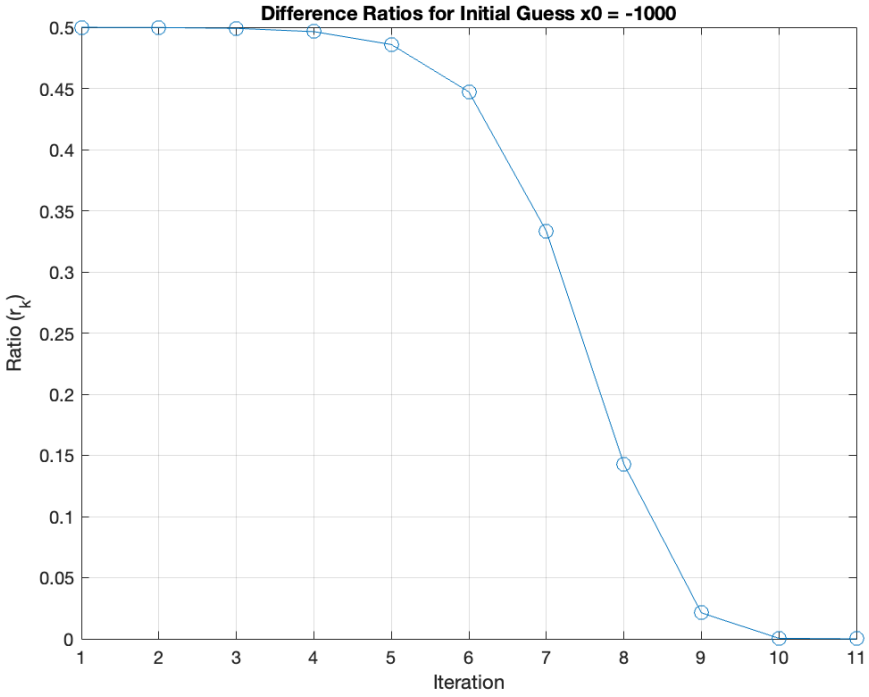


Figure 7: Difference Ratios for an Initial Guess $x_0 = -1000$

```
function [root, iterations] = modifiednewton(f, df, d2f, x0, tol, max_iter)
    iterations = 0;
    x = x0;
    while iterations < max_iter
        fx = f(x);
        dfx = df(x);
        d2fx = d2f(x);

        if abs(fx) < tol
            root = x;
            return;
        end
        x = x - (dfx / d2fx) * (fx / dfx);
        iterations = iterations + 1;
    end
    root = x
end
```

modifiednewton.m

Figure 8: modifiednewton.m