Newton's Iteration

Presented:

Yuchen Jin

Background

The problem: How can we calculate the roots of a function?

Newton's Iteration (also known as Newton's Method or the Newton-Raphson Method) is a numerical analysis method of approximating the roots of a function.

Applications of this method:

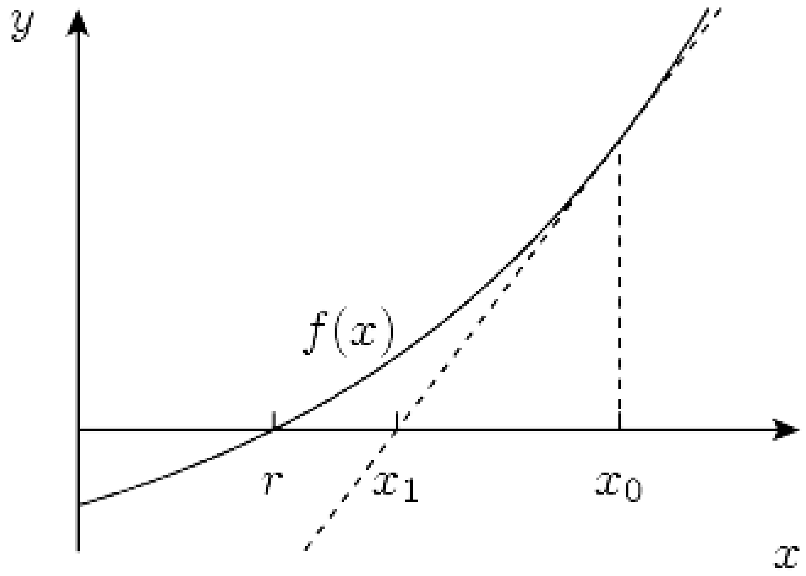
- Finding roots of functions
- Finding roots of numbers (ex: $\sqrt{77}$)
- Division
- Reciprocals (ex: the reciprocal of 77 is $\frac{1}{77}$)

Algorithm

1. Start with a given estimate, x_0.
2. Find the tangent line of the function at x_0.
3. The point at which this tangent line intersects with the x-axis becomes the next estimate, x_1.
4. Repeat the process until the estimate starts to converge.

This process can be described with the following equation:

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



(source)

Deriving the equation

- The slope of the tangent line at (x0, f(x0)) can be expressed as the following equation: f'(x0)=f(x0) – 0 / x0-x1, obtained by inserting the points (x0, f(x0)) and (x1, 0) into the slope formula m = y2-y1 / x2-x1.
- Solve the slope formula for x1:
- (x0-x1) f'(x0)=f(x0) - 0
- x0-x1=f(x0)f'(x1)
- x1=x0-f(x0)f'(x0)
- If we derive the equation for x2, we will end up with x2=x1-f(x1)f'(x1)and so on for x3,x4…. The sequence can be summarized with the Newton Iteration equation.

Example: Use Newton's Iteration to solve for $\sqrt[5]{1000}$. Start with $x_0 = 3.5$ and estimate root up to x_4

- $\sqrt[5]{1000} = 1000^{\frac{1}{5}} = x$
- $x - 1000^{\frac{1}{5}} = 0$
- $f(x) = x^5 - 1000$
- $f'(x) = 5x^4$
- x1=3.5-f(3.5)/f'(3.5)=4.132778009
- x2=4.132778009-f(4.132778009)/f'(4.132778009)=3.991807286
- x3=3.991807286-f(3.991807286)/f'(3.991807286)=3.9811209306
- x4=3.9811209306-f(3.9811209306)/f'(3.9811209306)=3.981071707
- Root calculated on a TI-84 calculator = 3.981071706

Division Using Newton's Iteration

Division (and the reciprocal of b) can be expressed as: $\frac{1}{b}$

We will express this as $\frac{1}{b}$ = x.

$$b - \frac{1}{x} = 0$$
$$f(x) = b - x^{-1}$$
$$f'(x) = x - 2$$

Substitute f(x) and f'(x) into the recurrence equation and simplify:

$$x_{n+1} = x_n - \frac{b - x_n^{-1}}{x_n^{-2}}$$
$$x_{n+1} = x_n(2 - bx_n)$$

Example: Calculate 1/115 using Newton's Iteration. Start with $x_0 = .001$

- x1=.001(2-115 • .001) =.001885
- x2=.001885(2-115 • .001885)=.0033613791
- x3=.0033613791(2-115 • .0033613791)=.0054233882
- x4=.0054233882(2-115 • .0054233882)=.0074642653
- x5=.0074642653(2-115 • .0074642653) =.0085212761
- Quotient calculated on a T1-84 calculator = .0086956522

Quadratic Convergence

- Convergence is the rate at which the series described by the recurrence equation reaches it’s limit (and the limit in this case is our “answer”)
- Newton’s Iteration has a quadratic rate of convergence. This means that for every iteration, its accuracy has essentially doubled.
- It can be expressed as:
  - $\frac{1}{a} - x_{n+1} = a(\frac{1}{a} - x_n)^2$

Newton's Iteration to Invert a Power Series - Divide and Conquer

- Truncated Power Series:
  - A(X)=a0 + a1X + ⋯ + an-1X^n-1 + O(X^n)
- Given the recurrence equation for reciprocals, as well as the quadratic convergence equation:
  - $x_n = a^{-1} + O\left(X^k\right)$
  - $x_{n+1} = a^{-1} + O(X^{2k})$

  - This indicates that there are twice as many terms correct terms in each iteration. Thus, we can treat each k as a sub-problem and we can set our base case to k=1, meaning there is one correct term.

Code

```
from scipy.misc import derivative

# basic newton iteration algorithm
def newton(func, estimate, min_error=.0001):
    diff = func(estimate)/derivative(func, estimate)
    while abs(diff) > min_error:
        diff = func(estimate)/derivative(func, estimate)
        estimate = estimate - diff
        diff = func(estimate)/derivative(func, estimate)
    return estimate
```

Code

```
# newton iteration for division
def newton_div(a, min_error=.0001):
    k = 0
    while 10**k < a:
        k += 1
    init_estimate = 1/(10**k)
    estimate = init_estimate * (2 - (a * init_estimate))
    diff = estimate - init_estimate
    while min_error < abs(diff):
        old_estimate = estimate
        estimate = estimate * (2 - (a * estimate))
        diff = estimate - old_estimate
    return estimate
```

Code

```
# based off of code from
#https://moodle.polytechnique.fr/pluginfile.php/116142/mod_resource/content/1/04-Newton.pdf
def invert_series(ma):
    n = len(ma)
    if n == 1:
        return [-1/ma[0]]
    k = int(ceil(n / 2))
    x = invert_series(ma[:k])+[0]*(n-k)
    # t = np.multiply(s, ma)
    t = np.dot(ma, x, n)  # -x*s
    t[0] += 1  # 1-a*s
    return np.add(x, np.dot(x, t, n))  # x+x(1-x*s)
```

Time Complexity

- Newton’s Iteration: O(lognF(n)) where F(n) is the time boundary on the operation f(x) / f’(x).
  - logn because of the quadratic convergence rate. Each iteration takes us halfway closer to the answer.
- Using divide and conquer: O(M(n)) where M(n) represents the time boundary on the operation of multiplying two polynomials together of degree n.

Thank you.