Approximating Roots with Computation

A mini-project

Rushil Ambati

Outline

In this mini-project I'll be using numerical methods and computation in order to approximate roots of an equation.

The specific method used here is called the *Newton-Raphson method* (alternatively known as just '*Newton's method*'), named after Isaac Newton and Joseph Raphson.

Contents

Theory

Newton-Raphson method

For a single-variable function f defined for a real variable x with derivative f' and an initial guess x_0 for a root of f. If the function satisfies assumptions, and the initial guess is close, then

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

where x_1 is a better approximation of the root than x_0. Geometrically, $(x_1, 0)$ is the intersection of the x-axis and the tangent of the graph of f at $(x_0, f(x0))$: that is, the improved guess is the unique root of the linear approximation at the initial point. The process is repeated as

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

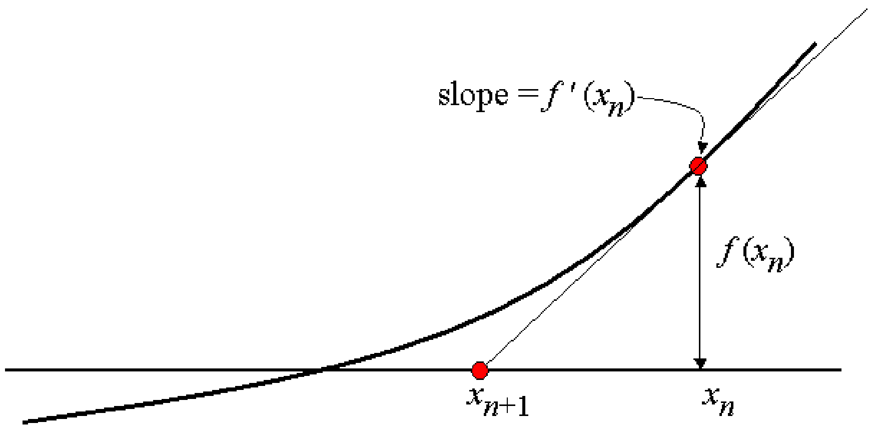
until a sufficiently precise value is reached. This algorithm is first in the class of Householder's methods, succeeded by Halley's method. The method can also be extended to complex functions and to systems of equations.

(*Source*)

Derivation of the equation

The iterative equation $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ is the form in the A-level specification too, and we'll use the same in our program here.
However, we weren't taught the derivation in class, so I'll attempt one here.

The geometric description above can be represented in a graph diagram, such as the one below:



(*Source*)

So we now have a tangent at x_n that has the following characteristics:

- It goes through the point $(x_n, f(x_n))$
- It has gradient $f'(x_n)$

So now to get the equation of the tangent, we can put it into point slope form $y - y_1 = m(x - x_1)$:

$$y - f(x_n) = f'(x_n)(x - x_n)$$

We want our value of x when $y = 0$ (or the root of the equation):

$$f(x_n) + f'(x_n)(x - x_n) = 0$$

Hence if we solve for x, which will be the x-intercept and closer to the root (our x_{n+1}):

$$f'(x_n)(x - x_n) = -f(x_n)$$
$$x - x_n = -\frac{f(x_n)}{f'(x_n)}$$
$$x = x_n - \frac{f(x_n)}{f'(x_n)}$$

we obtain our desired form.

Limitations and Practical Considerations

Difficulty in calculating the derivative of a function
Newton's method requires that the derivative can be calculated directly. An analytical expression for the derivative may not be easily obtainable or could be expensive to evaluate. In these situations, it may be appropriate to approximate the derivative by using the slope of a line through two nearby points on the function. Using this approximation would result in something like the *secant method* whose convergence is slower than that of Newton's method.

Overshoot
If the first derivative is not well behaved in the neighbourhood of a particular root, the method may overshoot, and diverge from that root. This includes points of inflection, local maxima or minima around x_0 or the root.

Features of graph around root
If a stationary point of the function is encountered, the derivative is zero and the method will terminate due to division by zero.

Poor initial estimate
A large error in the initial estimate can contribute to non-convergence of the algorithm.

Discontinuity
This will only work for functions defined for all real numbers. Asymptotes will also yield undefined values.

(*Source*)

Implementation

Newton-Raphson Method

Firstly, we'll define our function. Let's say for the sake of our example we're trying to find the roots of $x^2 - 4x - 7 = 0$.

```python
def f(x):
    return (x**2-4*x-7)
```

Next, we'll define the derivative of our function (so we can use it in place of the $f'(x_n)$ in the equation).

```python
def f_prime(x):
    return 2*x-4
```

Now we've got all our setup done, let's construct a function that will actually carry out the Newton-Raphson method.

```
def nrf(guess, n):
    print("x_0 = " + str(guess))
    for i in range(n+1):
        next_guess = guess - f(guess) / f_prime(guess)
        print("x_" + str(i+1) + " = " + str(next_guess))
        guess = next_guess
```

Now, let's call it. Say our starting value is 8, and we want it to iterate 10 times.

```
nrf(8, 10)
```

This provides the following output:

```
x_0 = 8
x_1 = 5.916666666666666
x_2 = 5.36258865248227
x_3 = 5.316938934730458
x_4 = 5.316624805231569
x_5 = 5.3166247903554
x_6 = 5.3166247903554
x_7 = 5.3166247903554
x_8 = 5.3166247903554
x_9 = 5.3166247903554
x_10 = 5.3166247903554
```

So our final value for the root closest to 8 is approximately 5.32 (to 3 significant figures.)
The true value of that positive root is $2 + \sqrt{11}$ and our approximation of 5.3166... is actually very close, and didn't take that many iterations to reach there. This is because our starting value of 8 is very close to the actual root.

We can also find the other root of this quadratic equation by starting at another value, say -4, iterating the same number of times.

```
nrf(-4, 10)
```

This provides the following output:

```
x_0 = -4
x_1 = -1.9166666666666665
x_2 = -1.3625886524822695
x_3 = -1.3169389347304572
x_4 = -1.3166248052315688
x_5 = -1.3166247903553998
x_6 = -1.3166247903553998
x_7 = -1.3166247903553998
x_8 = -1.3166247903553998
x_9 = -1.3166247903553998
x_10 = -1.3166247903553998
```

The other root is $2 - \sqrt{11}$ and our approximate of -1.3166... is also close. It reached a static value in just 5 iterations.

Secant Method - Tackling the derivative problem

Theory In the Limitations and Practical Considerations, we considered the possibility that we could be working with a function that may not have an easily obtainable derivative. In this case, I'll work around this problem by using an alternative method called the *Secant Method*. Although different in name, the only main difference between the *Newton-Raphson method* and the *Secant method* is the way that $f'(x_n)$ is calculated.

Instead of using an expression to programatically write in the derivative of the function, eg. `x**2-4*x-7`'s derivative being `2*x-4` we will use a finite difference method to calculate the derivative at a point of $f(x)$. The 'quotient difference', on which this method relies on, is nothing more than the fundamental idea that differentiation relies on.

We take differentiation from first principles to be as such:

$$f'(x) = \lim_{x \to 0} \frac{f(x+h) - f(x)}{h}$$

So, if we use a small value of h and just use our function for $f(x)$ we can get an approximation for the value of $f'(x)$ for any x. This means we don't actually have to evaluate the expression for the derivative, providing a workaround for the limitation discussed above.

Implementation All we have to do now, is to change our `f_prime` function to compute the quotient difference at some x with some difference value h.

```
def f_prime(x, h):
    return (f(x + h) - f(x)) / h
```

Of course, the smaller our h is, the better the approximation for the derivative, but the more expensive it is computationally.

Final Code

```
def f(x):
    # Define your function here
    return (x**2-4*x-7)

def f_prime(x):
    # Define your function's first derivative here
    return 2*x-4

def f_prime_sec(x, h):
    """Calculates the value of the derivative of a function

    Args:
        x (float): The value for which we are calculating the derivative value
        h (float): Quotient difference

    Returns:
        float: Approximate derivative of f(x) at x, with quotient difference h
    """
    return (f(x + h) - f(x)) / h

def approximate_root(guess, n, sec=False, h=1*10**-3):
    """Iteratively estimates the roots of a function by using the Newton-Raphson Method

    Args:
        guess (float): Initial value to iterate from
        n (int): Number of iterations
        sec (bool, optional): Whether or not to use the secant method. Defaults to False.
        h (float, optional): Quotient difference interval. Defaults to 1x10^-3.
    """
    print("x_0 = " + str(guess))
    for i in range(n):
        f_prime_x = f_prime_sec(guess, h) if (sec == True) else f_prime(guess)
        next_guess = guess - f(guess) / f_prime_x # Evaluating x_n+1
        print("x_" + str(i+1) + " = " + str(next_guess))
        guess = next_guess
```

Function can be invoked as follows:

```
approximate_root(8, 10) # Uses the derivative manually calculated
approximate_root(-4, 10, sec=True) # Uses secant method to approximate derivative
```

Points for discussion

- To see how the Secant method compares to the Newton-Raphson method depending on the type of graph considered or other factors.
- To look into other numerical methods that have other use-cases, as well as how they compare to the two considered above.
- I wonder if there's a more efficient way to write up my algorithm. Also, this is a simple enough algorithm, and doesn't use any libraries so isn't difficult to port to other languages. Maybe try porting it to something like C, where it'd actually be somewhat efficient - who knows, maybe even in Assembly?
- It'd be cool to see both of these methods applied to:
  - More than one dimension.
  - Complex functions
  - Systems of equations
- If it's possible (in some, or even all cases) to mathematically prove whether or not the iterative method will converge given a particular starting value and the equation of the graph.
- I've heard about the 'Laplace Transform' when talking about numerical methods, I wonder what that is.
- On the Wikipedia page for the Secant method, it discusses the "order of convergence" and the golden ratio. It'd be interesting to see where the golden ratio fits into it all.

  > The iterates x_n of the secant method converge to a root of f, if the initial values x_0 and x_1 are sufficiently close to the root. The order of convergence is φ, where
  >
  > $$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$
  >
  > is the golden ratio. In particular, the convergence is superlinear, but not quite quadratic.
  >
  > This result only holds under some technical conditions, namely that f be twice continuously differentiable and the root in question be simple (i.e., with multiplicity 1).
  >
  > (*Source*)