Machine Learning and Photonics

Math Foundations: Root Finding

Ergun Simsek

University of Maryland Baltimore County
simsek@umbc.edu

If $f(x) = a$, you can find that x numerically

February 7, 2023

Remember the Quadratic Formula?

When we were fifth graders, we learned that if $f(x) = ax^2 + bx + c$, then we can find the real solutions (roots) using

$$x_r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

But how can we find the solution (root) of a complex function? For example,

$$f(x) = 4 + x^3 + \sin(0.2\pi x)^2 + 0.3\mathrm{sech}(x) + 2^{-|x|}$$

Root Finding

The **root** or **zero** of a function, $f(x)$, is an x_r such that $f(x_r) = 0$.

- For functions such as $f(x) = x^2 - 9$, the roots are clearly 3 and -3.
- However, for other functions such as $f(x) = \cos(x) - x$, determining an **analytic**, or exact, solution for the roots of functions can be difficult.
- For these cases, it is useful to generate numerical approximations of the roots of f and understand the limitations in doing so.

Tolerance

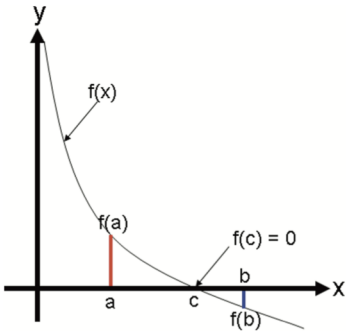
Error = True Value − Predicted (or Computed) Value

Tolerance = Level of error that is acceptable for a given application

- We say that a computer program has **converged** to a solution when it has found a solution with an error smaller than the tolerance.
- When computing roots numerically, or conducting any other kind of numerical analysis, it is important to establish both a metric for error and a tolerance that is suitable for a given engineering/science application.

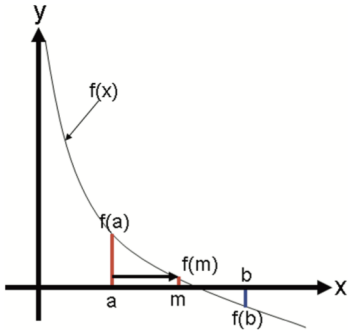
Intermediate Value Theorem

if $f(x)$ is a continuous function between a and b, and $\text{sign}(f(a)) \neq \text{sign}(f(b))$, then there must be a c, such that $a < c < b$ and $f(c) = 0$.



Bisection Method

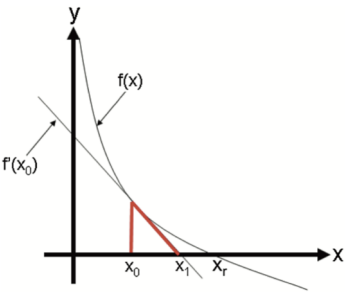
1. Assume, without loss of generality, that $f(a) > 0$ and $f(b) < 0$.
2. There must be a root on the open interval (a, b).
3. Now let $m = \frac{b+a}{2}$, the midpoint between and a and b. If $|f(m)| \leq \epsilon$, then m is a root.
4. If $f(m) > \epsilon$, then m is an improvement on the left bound, a, and there is guaranteed to be a root on the open interval (m, b).
5. If $-f(m) > \epsilon$, then m is an improvement on the right bound, b, and there is guaranteed to be a root on the open interval (a, m).



Exercise: In MATLAB, define a function that approximates a root of $f(x)$ bounded by a and b to within tolerance $|f(x)| <$ tol and calculate $\sqrt{2}$ by solving $f(x) = x^2 - 2$

Newton-Raphson Method

- Let $f(x)$ be a smooth and continuous function and x_r be an unknown root of $f(x)$.
- Assume that x_0 is a guess for x_r (but not the root that we are looking for)
- We want to find an x_1 that is an improvement on x_0 (i.e., closer to x_r than x_0).
- If we assume that x_0 is "close enough" to x_r, then we can improve upon it by taking the linear approximation of $f(x)$ around x_0, which is a line, and finding the intersection of this line with the x-axis.



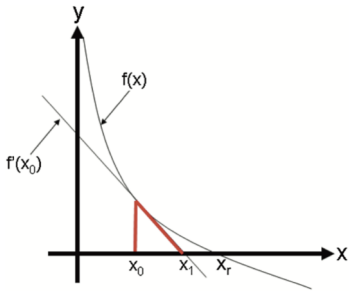
Newton-Raphson Method (Cont...)

- The linear approximation of $f(x)$ around x_0 is $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$.
- Using this approximation, we find x_1 such that $f(x_1) = 0$.

$$0 = f(x_0) + f'(x_0)(x_1 - x_0),$$

which when solved for x_1 is

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



Written generally, a **Newton step** computes an improved guess, x_i, using a previous guess x_{i-1}, and is given by the equation

$$x_i = x_{i-1} - \frac{g(x_{i-1})}{g'(x_{i-1})}.$$

The **Newton-Raphson Method** of finding roots iterates Newton steps from x_0 until the error is less than the tolerance.

Newton-Raphson Example

Let's determine $\sqrt{2}$ by solving $f(x) = x^2 - 2$ using $x_0 = 1.4$ as a starting point.

$$x = 1.4 - \frac{1.4^2 - 2}{2(1.4)} = 1.4142857142857144$$

```
import numpy as np

f = lambda x: x**2 - 2
f_prime = lambda x: 2*x
newton_raphson = 1.4 - (f(1.4))/(f_prime(1.4))

print("newton_raphson =", newton_raphson)
print("sqrt(2) =", np.sqrt(2))
```

```
newton_raphson = 1.4142857142857144
sqrt(2) = 1.4142135623730951
```

Newton-Raphson Example (Cont...)

Now let's write a function *my_newton*(f, *df*, *x*0, *tol*), where the output is an estimation of the root of f, f is a function object $f(x)$, *df* is a function object to $f'(x)$, *x0* is an initial guess, and *tol* is the error tolerance. The error measurement should be $|f(x)|$.

```
]: def my_newton(f, df, x0, tol):
       # output is an estimation of the root of f
       # using the Newton Raphson method
       # recursive implementation
       if abs(f(x0)) < tol:
           return x0
       else:
           return my_newton(f, df, x0 - f(x0)/df(x0), tol)
```

Newton-Raphson Example (Cont...)

Let's use *my_newton* to compute $\sqrt{2}$ to within tolerance of 1e-6 starting at *x0 = 1.5*.

```
estimate = my_newton(f, f_prime, 1.5, 1e-6)
print("estimate =", estimate)
print("sqrt(2) =", np.sqrt(2))
```

```
estimate = 1.4142135623746899
sqrt(2) = 1.4142135623730951
```

Bisection vs. Newton-Raphson

If x_0 is close to x_r, then it can be proven that, in general, the Newton-Raphson method converges to x_r much faster than the bisection method.

Root Finding in Python

Both MATLAB and Python have built-in root-finding functions. The function we will use Python to find the root is fsolve from the *scipy.optimize*.
The fsolve function takes in many arguments that you can find in the documentation, but the most important two is the function you want to find the root, and the initial guess.
Example: Compute the root of the function $f(x) = x^3 - 100x^2 - x + 100$ using fsolve.

```
from scipy.optimize import fsolve
```

```
f = lambda x: x**3-100*x**2-x+100

fsolve(f, [2, 80])
```

```
array([  1., 100.])
```

We know that this function has two roots $x = 1$ and $x = 100$, therefore, we can get the two roots out fairly simple using the fsolve function.

Root Finding in MATLAB

```
x = fsolve(@(x)x.^3-100*x.^2-x+100,10)
```

will give you 1

```
x = fsolve(@(x)x.^3-100*x.^2-x+100,80)
```

will give you 100

```
x = fsolve(@(x)x.^3-100*x.^2-x+100,[10, 80])
```

will give you 1 and 100

```
x = fsolve(@(x)x.^3-100*x.^2-x+100,[10, 20, 80])
```

will give you 1, 1, and 100