Lecture #1: Newton's Method and Other Functional Hijinks

Higher-Order Functions at Work: Iterative Update

- A general strategy for solving an equation:

```
Guess a solution
while your guess isn't good enough:
    update your guess
```

- The three underlined segments are parameters to the process.
- The last two segments clearly require functions for their representation—a *predicate* function (returning true/false values), and a function from values to values.
- In code,

```
def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result."""
```

Recursive Versions

```
def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result."""
    if done(guess):
        return guess
    else:
        return iter_solve(update(guess), done, update)
```

or

```
def iter_solve(guess, done, update):
    def solution(guess):
        if done(guess):
            return guess
        else:
            return solution(update(guess))
    return solution(guess)
```

Iterative Version

```
def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result."""
    while not done(guess):
        guess = update(guess)
    return guess
```

Adding a Safety Net

- In real life, we might want to make sure that the function doesn't just loop forever, getting no closer to a solution.

```
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result.  Causes error if more than
    ITERATION_LIMIT applications of UPDATE are necessary."""
```

Adding a Safety Net: Code

- In real life, we might want to make sure that the function doesn't just loop forever, getting no closer to a solution.

```
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result.  Causes error if more than
    ITERATION_LIMIT applications of UPDATE are necessary."""

    def solution(guess, iteration_limit):
        if done(guess):
            return guess
        elif iteration_limit <= 0
            raise ValueError("failed to converge")
        else:
            return solution(update(guess), iteration_limit-1)
    return solution(guess, iteration_limit)
```

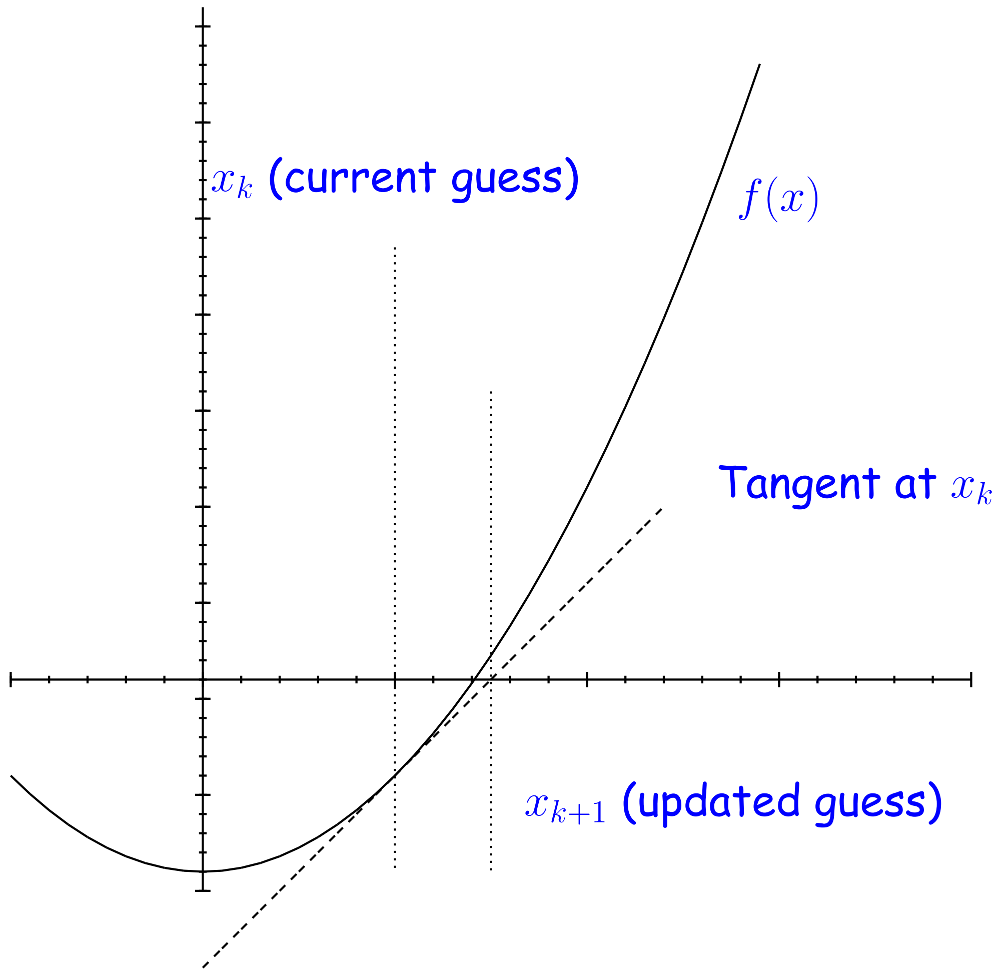
Iterative Version

```
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result.  Causes error if more than
    ITERATION_LIMIT applications of UPDATE are necessary."""

    while not done(guess):
        if iteration_limit <= 0:
            raise ValueError("failed to converge")
        guess, iteration_limit = update(guess), iteration_limit-1
    return guess
```

Newton's Method

- Newton's method uses the basic iterative scheme with a particular update strategy to solve $f(x) = 0$:



Using Iterative Solving For Newton's Method (I)

- Newton's method takes a function, its derivative, and an initial guess, and produces a result to some desired tolerance (that is, to some definition of "close enough").
- See http://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif
- Given a guess, x_k, compute the next guess, x_{k+1} by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

```
def newton_solve(func, deriv, start, tolerance):
    """Return x such that |FUNC(x)| < TOLERANCE, given initial
    estimate START and assuming DERIV is the derivatative of FUNC."""
    def close_enough(x):
        ________________?________________
    def newton_update(x):
        ________________?________________

    return iter_solve(start, close_enough, newton_update)
```

Using Iterative Solving for Newton's Method (II)

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

```
def newton_solve(func, deriv, start, tolerance):
    """Return x such that |FUNC(x)| < TOLERANCE, given initial
    estimate START and assuming DERIV is the derivatative of FUNC."""
    def close_enough(x):
        return abs(func(x)) < tolerance
    def newton_update(x):
        return x - func(x) / deriv(x)

    return iter_solve(start, close_enough, newton_update)
```

Using newton_solve for $\sqrt{\cdot}$ and $\lg\cdot$

```
def square_root(a):
    return newton_solve(lambda x: x*x - a, lambda x: 2 * x,
                        a/2, 1e-5)

def logarithm(a, base = 2):
    return newton_solve(lambda x: base**x - a,
                        lambda x: x * base**(x-1),
                        1, 1e-5)
```

Dispensing With Derivatives

- What if we just want to work with a function, without knowing its derivative?
- Book uses an approximation:

```
def find_root(func, start=1, tolerance=1e-5):
    def approx_deriv(f, delta = 1e-5):
        return lambda x: (func(x + delta) - func(x)) / delta
    return newton_solve(func, approx_deriv(func), start, tolerance)
```

- This is nice enough, but looks a little ad hoc (how did I pick delta?).
- Another alternative is the *secant method.*

The Secant Method

- Newton's method was

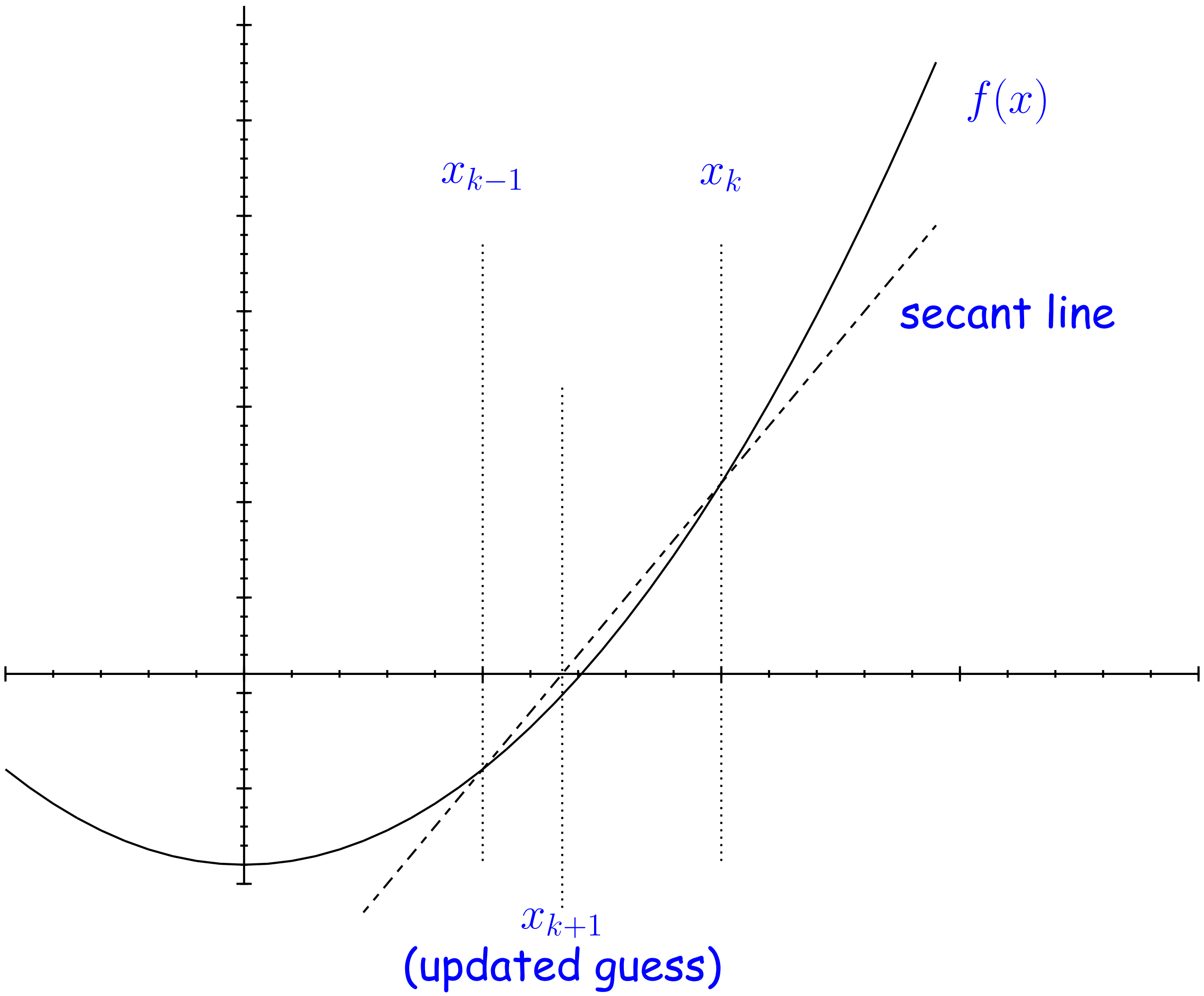
$$x_{k+1} = x_k - \frac{f(x)}{f'(x)}$$

- The secant method uses the last approximations values to get (in effect) a replacement for the derivative:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

- See `http://en.wikipedia.org/wiki/File:Secant_method.svg`
- But this is a problem for us: so far, we've only fed the update function the value of x_k each time. Here we also need x_{k-1}.
- How do we generalize to allow arbitrary extra data (not just x_{k-1})?

Secant Method Illustrated



Generalized iter_solve

```
def iter_solve2(guess, done, update, state=None):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS and STATE, until DONE yields a true value
    when applied to the result.  Besides a guess, UPDATE
    also takes and returns a state value, which is also passed to
    DONE."""
    while not done(guess, state):
        guess, state = update(guess, state)
    return guess
```

Using Generalized iter_solve2 for the Secant Method

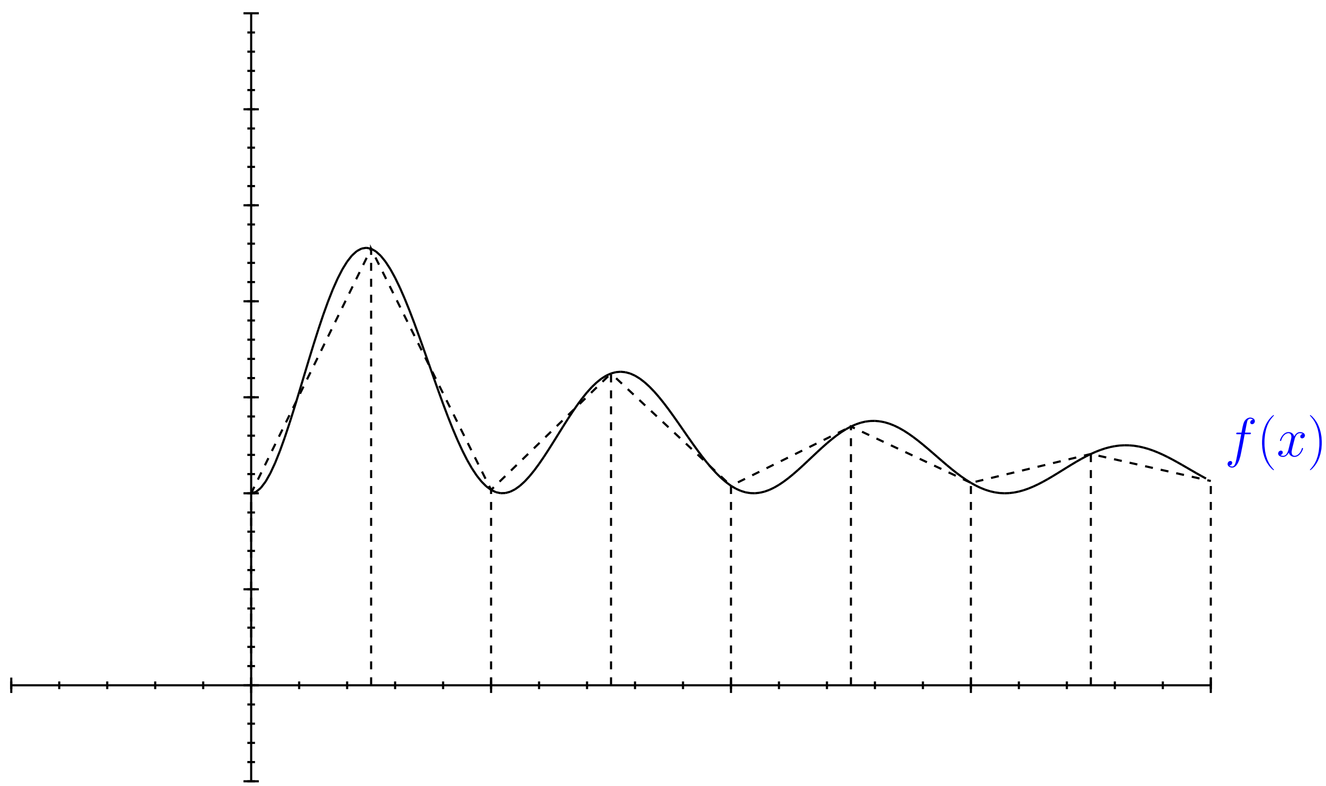
The secant method:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

```
def secant_solve(func, start0, start1, tolerance):
    def close_enough(x, state):
        return abs(func(x)) < tolerance
    def secant_update(xk, xk1):
        return (xk - func(xk) * (xk - xk1)
                                / (func(xk) - func(xk1)),
                xk)
    return iter_solve2(start1, close_enough, secant_update, start0)
```

Numerical Integration

You may have encountered a method of approximately integrating functions using the trapezoidal rule:



So $\int_0^4 f(x)dx$ is approximately the area under the dashed trapezoids.

Numerical Integration Method

```
def integrate_trapezoidal(f, low, high, step):
    """An approximation to the definite intregral of F from
    LOW to HIGH, computed by adding the areas of trapezoids of
    height STEP."""
```

Numerical Integration Method

```
def integrate_trapezoidal(f, low, high, step):
    """An approximation to the definite intregral of F from
    LOW to HIGH, computed by adding the areas of trapezoids of
    height STEP."""

    area = 0
    while low + step < high:
        area += (f(low) + f(low + step)) * step * 0.5
        low += step
    # Before returning, take care of the case where the final value
    # of low is less than high.
    return area + (f(low) + f(high)) * (high - low) * 0.5
```

- The file `e01.py` has a few interesting variations on this.