Root Finding

CS3220 Summer 2008

Jonathan Kaldor

What have we studied so far?

- So far, we have looked at problems where we can compute "exact" (modulo floating point issues) solutions in a fixed amount of time
- Now we move to problems where our solutions may have error tolerances, and our algorithms may have to iterate to compute them.

Issues We May Have To Deal With

- Algorithms that may not terminate for all problems
  - How fast does our algorithm converge to the correct answer (if it even converges)?
- Balancing error tolerances and running time
- Finding all of the solutions to the problem

Root Finding

- Problem statement: given a function f(x), find x such that f(x) = 0
- Common assumptions: f is continuous, differentiable (but typically dont assume much more - in particular, don't assume linearity)
- Can be in one variable, or a vector valued function $f(\underline{\mathbf{x}}) = \underline{\mathbf{0}}$ (we'll focus on the one variable case for the moment)

Root Finding

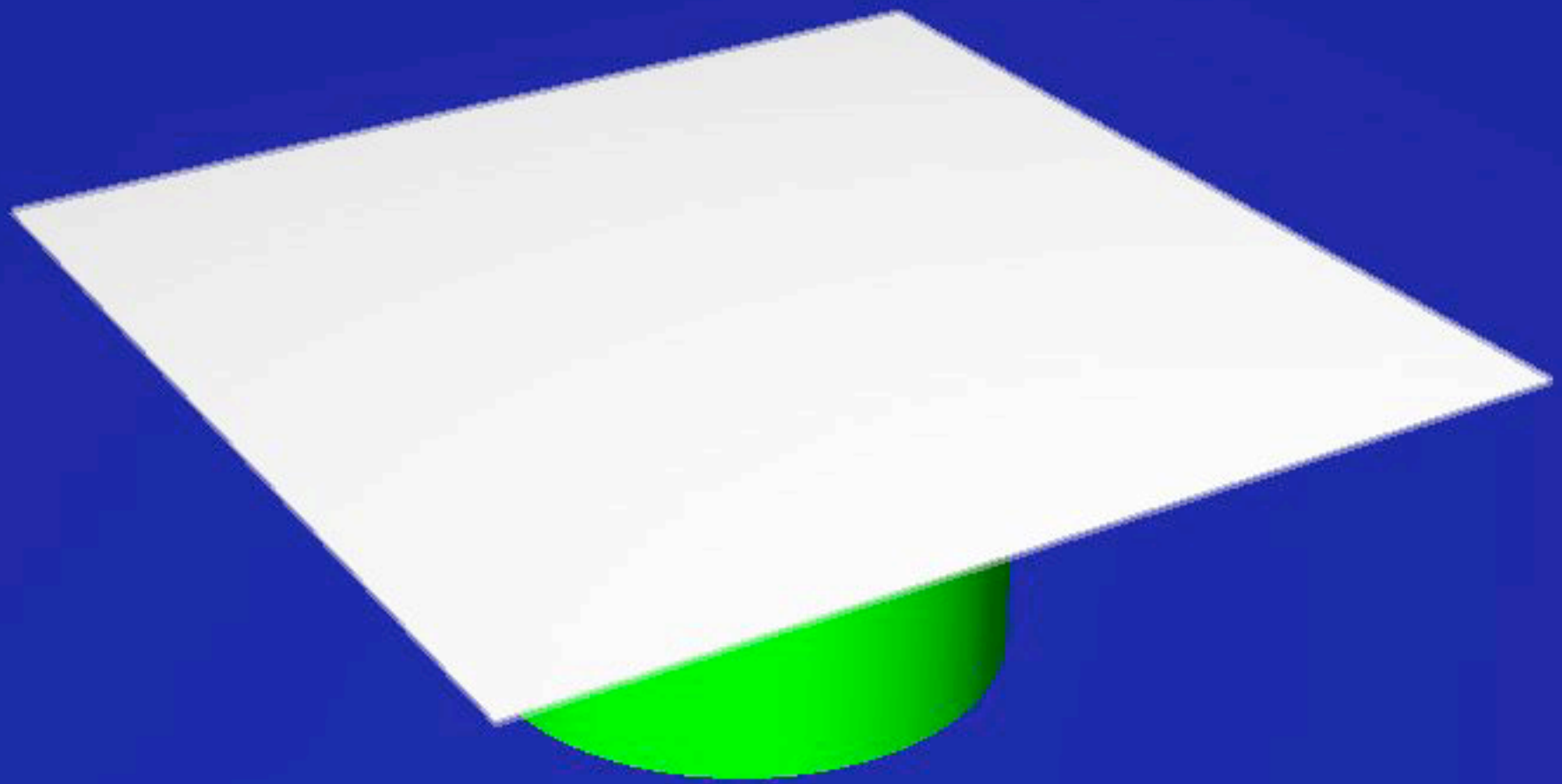
- Can of course be used to find x such that f(x) = c for any choice of c
  - Simply define g(x) = f(x) - c and find the roots of g(x)
- This is the nonlinear generalization of $\mathbf{A}\underline{\mathbf{x}}=\underline{\mathbf{b}}$
- ... but there may be no solutions, one, finitely many, or infinitely many solutions, even for well-defined problems

Applications

- Many problems can be expressed as finding the roots of some function, even in unexpected places
- Flashback: detecting collisions between triangles in 3D can be re-expressed as finding the roots of a particular cubic equation

Applications

- Cloth Simulation: we have the configuration of the cloth $\underline{\mathbf{x}}_n$ at some time n, and we want to find the configuration at time n+1
  - Baraff and Witkin: express it as a function
    $\underline{\mathbf{x}}_{n+1} = \underline{\mathbf{x}}_n + f(\underline{\mathbf{x}}_{n+1})$
    where $f(\underline{\mathbf{x}})$ is a function that describes how the cloth will react in a certain state
  - Need to find $\underline{\mathbf{x}}_{n+1}$ to satisfy above equation - use root finding



Root Finding

- For simple equations, we can find the roots analytically:
  linear equations
  quadratic equations
  certain trigonometric equations
- Requires knowing special facts about our problem - can't apply in general

Root Finding

- Our assumptions: f(x) is continuous, and differentiable (when we need it to be)
- We have some code that computes f(x) for given values of x - no other requirements on structure of f
- Our metric for speed: number of iterations, and number of function evaluations

Bisection Method

- Suppose we have [a,b] such that f(a) and f(b) have opposite signs (i.e. $f(a) > 0$ and $f(b) < 0$, or vice versa). Assume w/o loss of generality that $f(a) < 0$ and $f(b) > 0$
- Intermediate Value Theorem: if f(x) is continuous on [a,b] and $f(a) \leq y \leq f(b)$ (or vice versa), then there exists some c such that $a \leq c \leq b$ and $f(c) = y$

Bisection Method

- We have f(a) < 0 and f(b) > 0, so by Intermediate Value Thm, there is some c in [a,b] such that f(c) = 0
- Strategy: Evaluate f at (a+b)/2. Three cases can happen:
  f((a+b)/2) = 0 [we're done]
  f((a+b)/2) > 0
  f((a+b)/2) < 0

Bisection Method

- We have $f(a) < 0$, $f(b) > 0$
  If $f((a+b)/2) > 0$, then there must be a root in the interval $[a, (a+b)/2]$

  $f((a+b)/2) < 0$, then there must be a root in the interval $[(a+b)/2, b]$

  In either case, we now have a new interval to recurse on, of half the size. Update a and b appropriately and continue to next step

Bisection Method

- We terminate when we happen to get lucky and our root lies on the midpoint
- More likely, we terminate when we have achieved some bound on the error: when $|b-a| < \varepsilon$ for some chosen error ε
  - We know our root lies within the final interval, but not *quite* sure where

Example of Bisection

- Let $f(x) = (x - 2/3)^7$, choose a=0, b=1

Convergence of Bisection

- As long as a and b are chosen to bracket a root, bisection will *always* converge to a root
  - Need to choose a and b carefully if we want to converge to a *particular* root
- Finding a and b that bracket a root can be problematic
  - Consider $10*(x-1)^{10} - 0.0001$

Convergence of Bisection

- How fast do we converge to a root?
- Suppose we have a root r, and initial bounds $[a_0, b_0]$. Then if $c_0 = (a_0 + b_0)/2$ is our first midpoint, it must be the case that
  $|r - c_0| \leq (b_0 - a_0)/2$
- Similarly, if we then have $[a_1, b_1]$ as our new interval,
  $|r - c_1| \leq (b_1 - a_1)/2$

Convergence of Bisection

- In general, we have
  $|r - c_n| \leq (b_n - a_n)/2$
- Since the size of the interval is halved after each iteration, we can conclude
  $|r - c_n| \leq (b_0 - a_0)/2^{n+1}$
- After each iteration, we have added one more correct digit in base-2 floating point
  - Linear convergence

Convergence of Bisection

- Suppose we have a root bracketed in [32, 33]. How many iterations will it take to compute the root to a precision of epsilon in single (double) precision?

Other Details of Bisection

- Evaluating (a+b)/2 is problematic due to floating point issues
- Much better to evaluate a + (b-a)/2
- Also: save function evaluations at intervals - only need one additional evaluation per iteration instead of three

Bisection Method - Conclusions

- Guarantee of convergence is nice
- ... but very slow (relative) convergence
- We also need to bracket the root we're interested in
- Oftentimes used in combination with another method that converges faster but isn't guaranteed to converge

Other Details of Bisection

- Note: Bisection uses almost no information about the function - only needs to be able to evaluate it, and only uses the signs of the result.
- Can we use additional information about our function to speed convergence?

Newton's Method

- Also called Newton-Raphson iteration
- Extremely important tool for root finding, and can be directly extended for finding roots of vector functions (not just scalar functions we've been looking at)
- Added assumption: our function is both continuous and differentiable

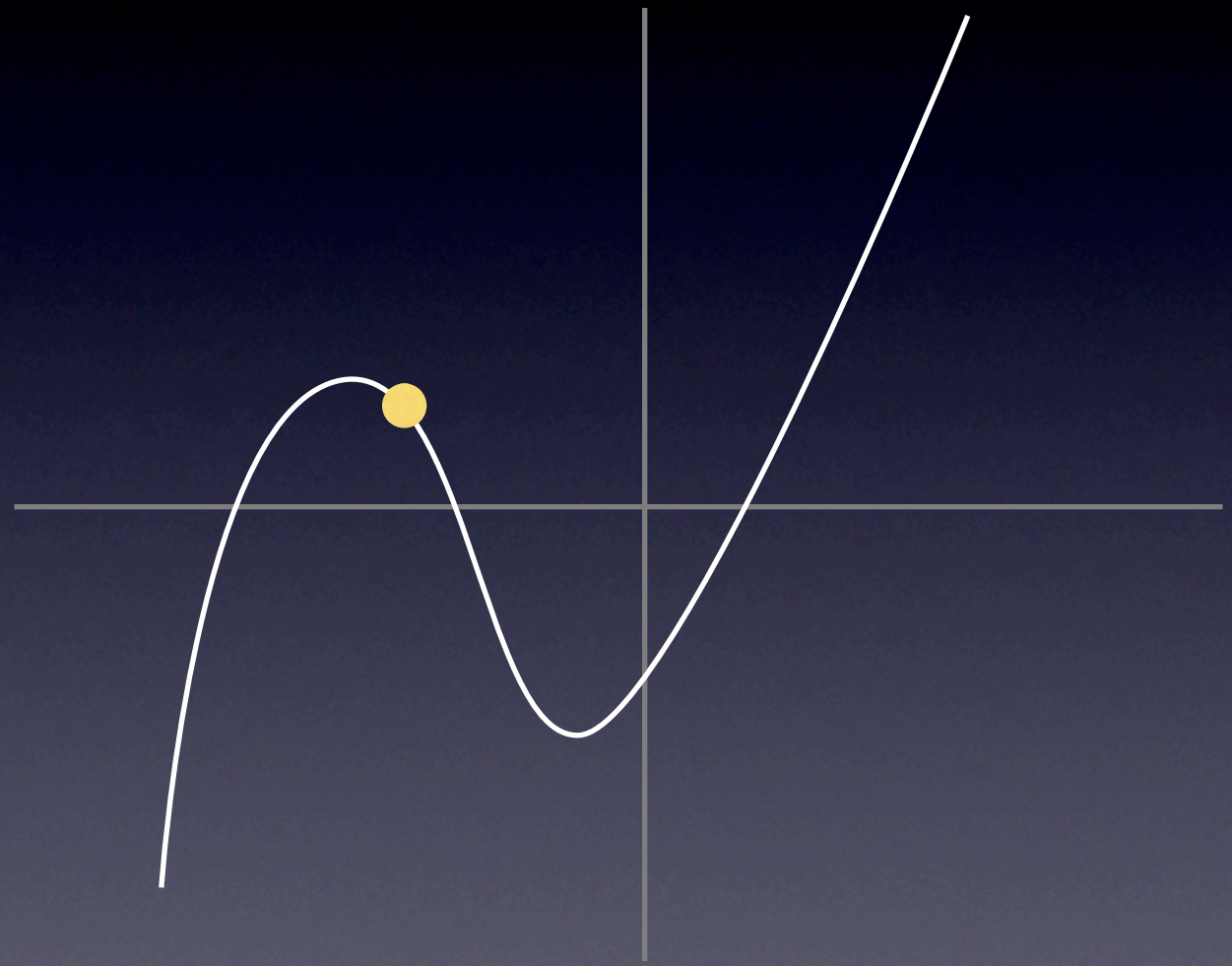
Newton's Method

- Can be derived in a variety of ways. Basic idea: if we are at a point x_i, approximate f(x) at x_i using a linear approximation, and take x_{i+1} to be root of linear approximation
- If f(x) is linear, solves for the root in one iteration

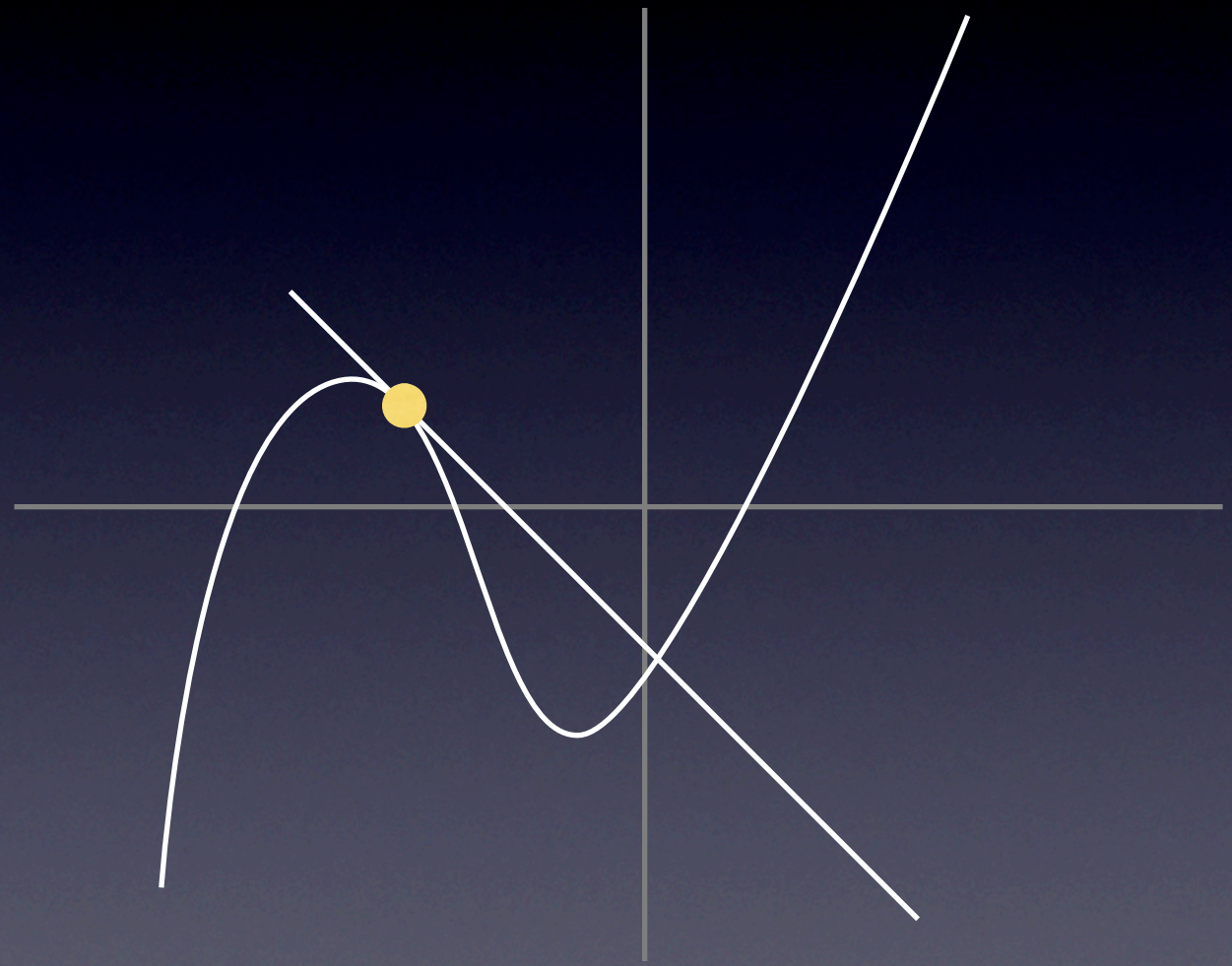
Newton's Method

- Follows from Taylor series of f(x) evaluated at x_i:
  $f(x_i + h) = f(x_i) + hf'(x_i) + h^2f''(x_i)/2 + ...$
- We truncate after the first two terms:
  $f(x_i + h) \approx f(x_i) + hf'(x_i)$
- The root of this function is then
  $0 = f(x_i) + hf'(x_i)$
  $h = -f(x_i) / f'(x_i)$

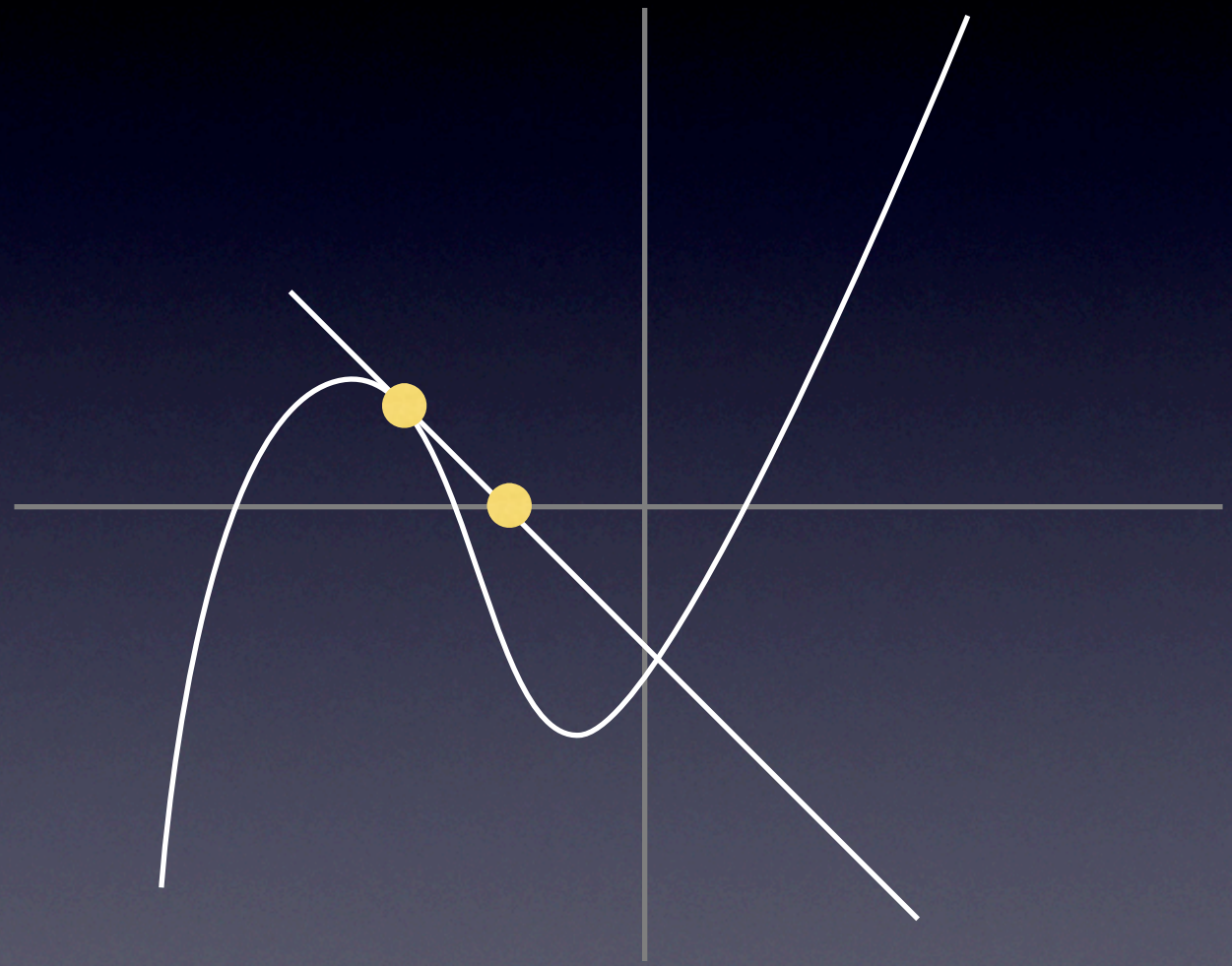
Newton's Method



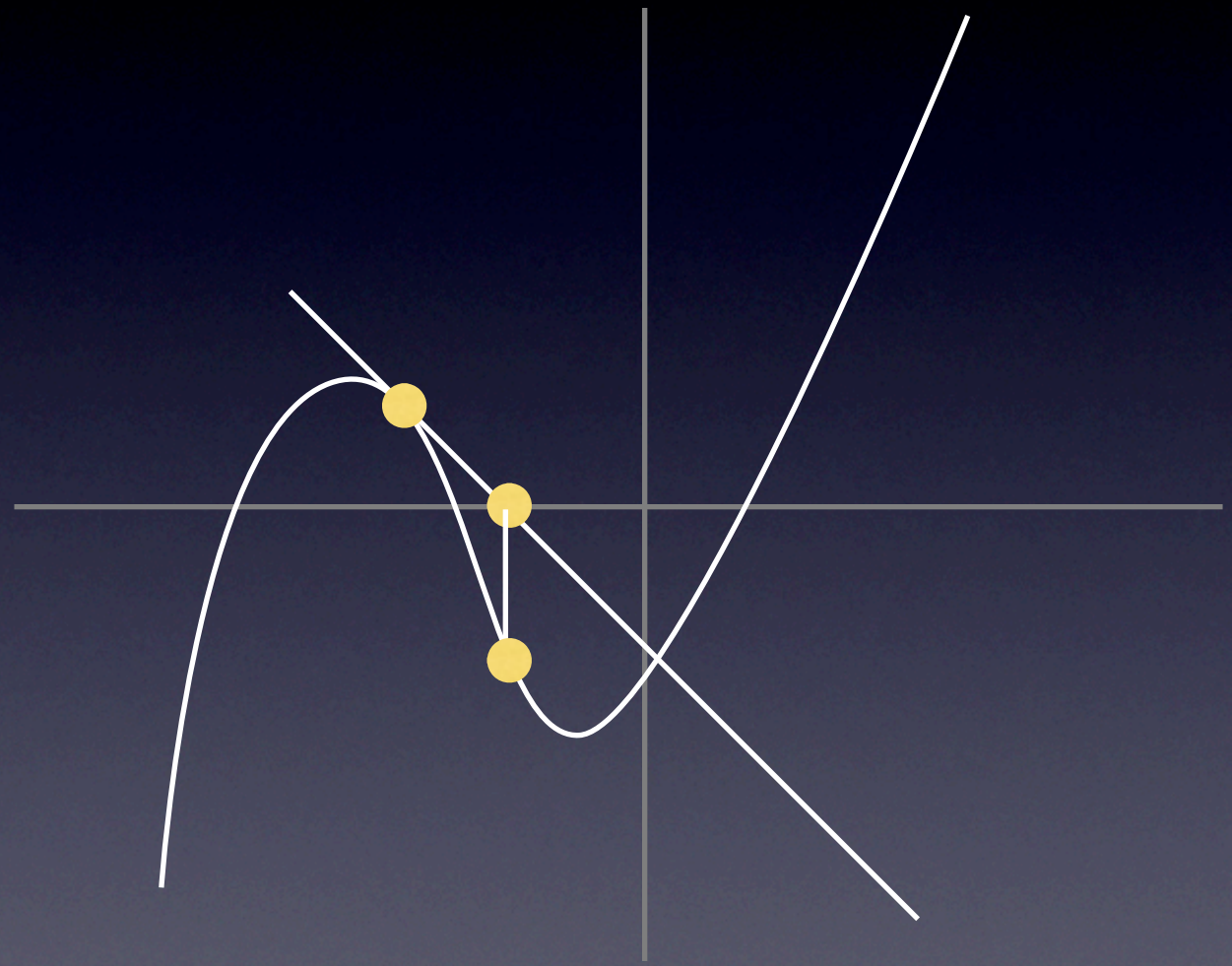
Newton's Method



Newton's Method



Newton's Method



Newton's Method

- If $x_{i+1} = x_i + h$, then, we get the full algorithm:

  x_1 = initial guess
  for i = 1, 2, ...
  $x_{i+1} = x_i - f(x_i)/f'(x_i)$
  end

Newton's Method

- Example: Finding the square root of a number c

  $f(x) = x^2 - c$

Newton's Method

- Properties of Newton's Method:
- Can converge quadratically (number of correct digits is appx. doubled per iteration) for simple roots. Can oftentimes get away with only 5 or 6 iterations of Newton and get full machine precision
- ... when it converges

Convergence of Newton's Method

- Local theory of convergence: suppose r is a simple root ($f'(r) \neq 0$), and f' and f'' are continuous. If x_0 is "close enough" to r, then Newton converges quadratically to the answer

  $$|r - x_{n+1}| \leq c\,|r - x_n|^2$$

  Compare to linear convergence of Bisection

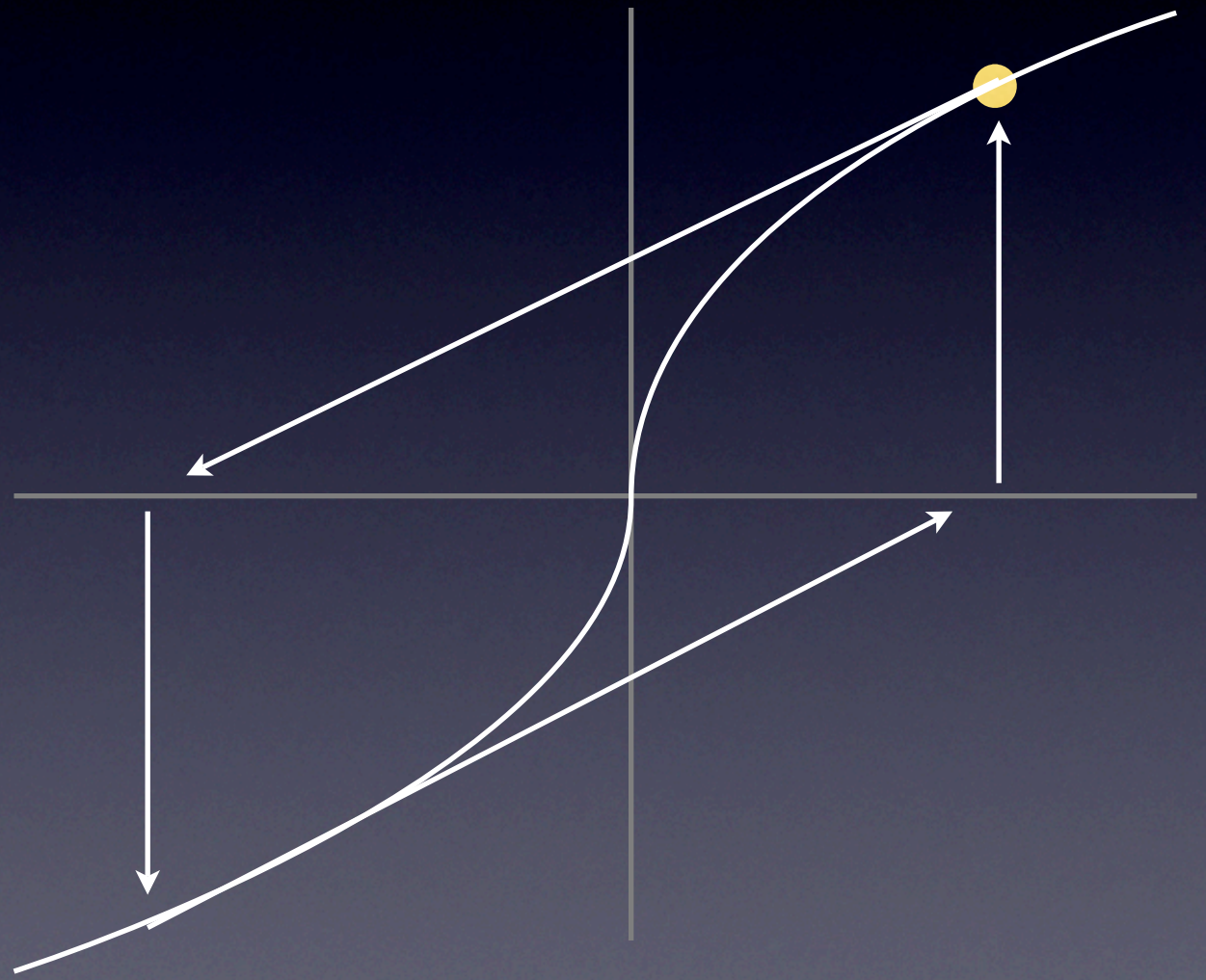
Newton's Method

- Potential problems:
  - Derivative can be zero or nearly zero - our next x_{i+1} will be very far away from x_i
  - Convergence properties are not quite as attractive as Bisection
    - We converge faster, but convergence is not guaranteed

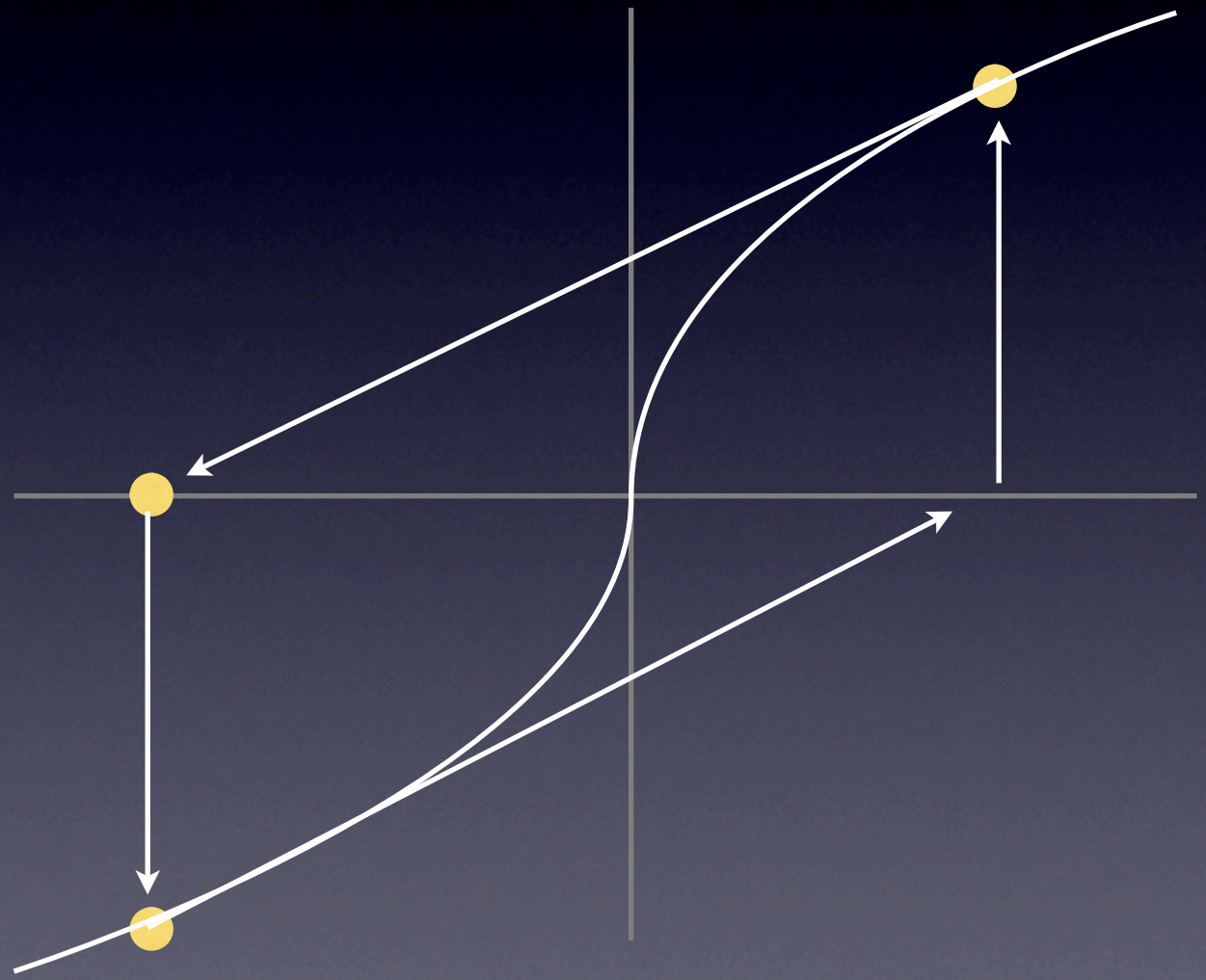
(Non)-Convergence of Newton's Method

- Can we get Newton's to iterate forever without converging OR diverging?
- Want it to cycle around a point a
- Then we want
  $x_{i+1} - a = -(x_i - a)$
- This is satisfied when
  $f(x) = \text{sign}(x-a)\sqrt{|x - a|}$

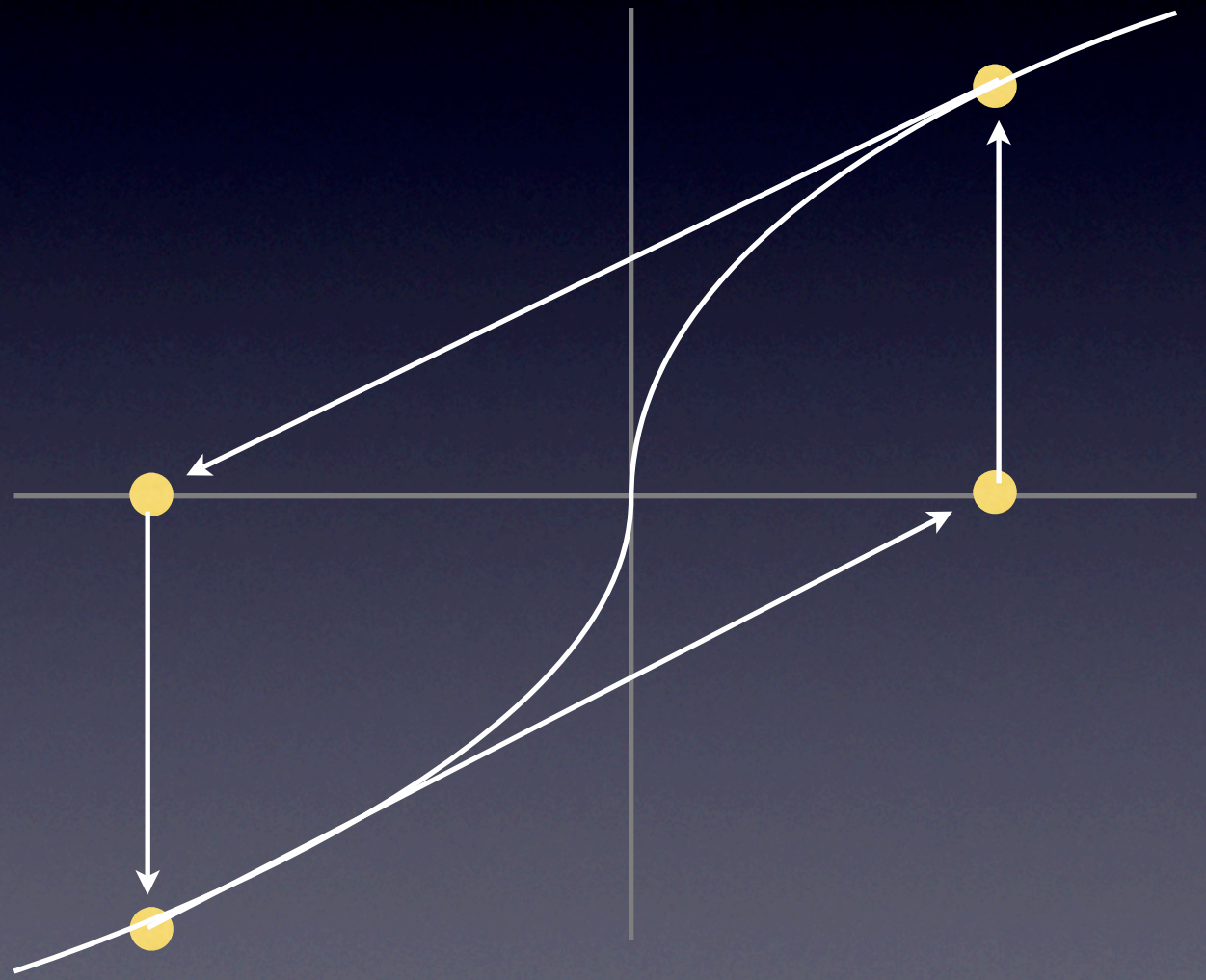
(Non)-Convergence of Newton's Method



(Non)-Convergence of Newton's Method



(Non)-Convergence of Newton's Method



(Non)-Convergence of Newton's Method

- We will continue to oscillate in this stable pattern, neither converging nor diverging
- What happens to cause this breakdown?
  - First derivative is unbounded at a
  - Can give other examples that cause oscillation for some iterates (but not all)

(Non)-Convergence of Newton's Method

- Other problems:
  - Divergence of iterates *away* from root (chalkboard)

Requirements of Newton's Method

- Note that we need to be able to evaluate both f(x) and f'(x)
  - Two function evaluations per iteration (one of each)
  - Derivative can occasionally be difficult to compute, or we may want to avoid the extra work

Derivative Refresher

- Well, what is the derivative anyways? Its simply the limit of the difference between two points on the function as those points get close together:

  lim (f(x+h) - f(x))/h

  So we can approximate the derivative using function evaluation at two points

Derivative Refresher

- What points would be good to evaluate at?
- How about x_n and x_{n-1}?
  - Saves a function evaluation
  - Don't need the derivative

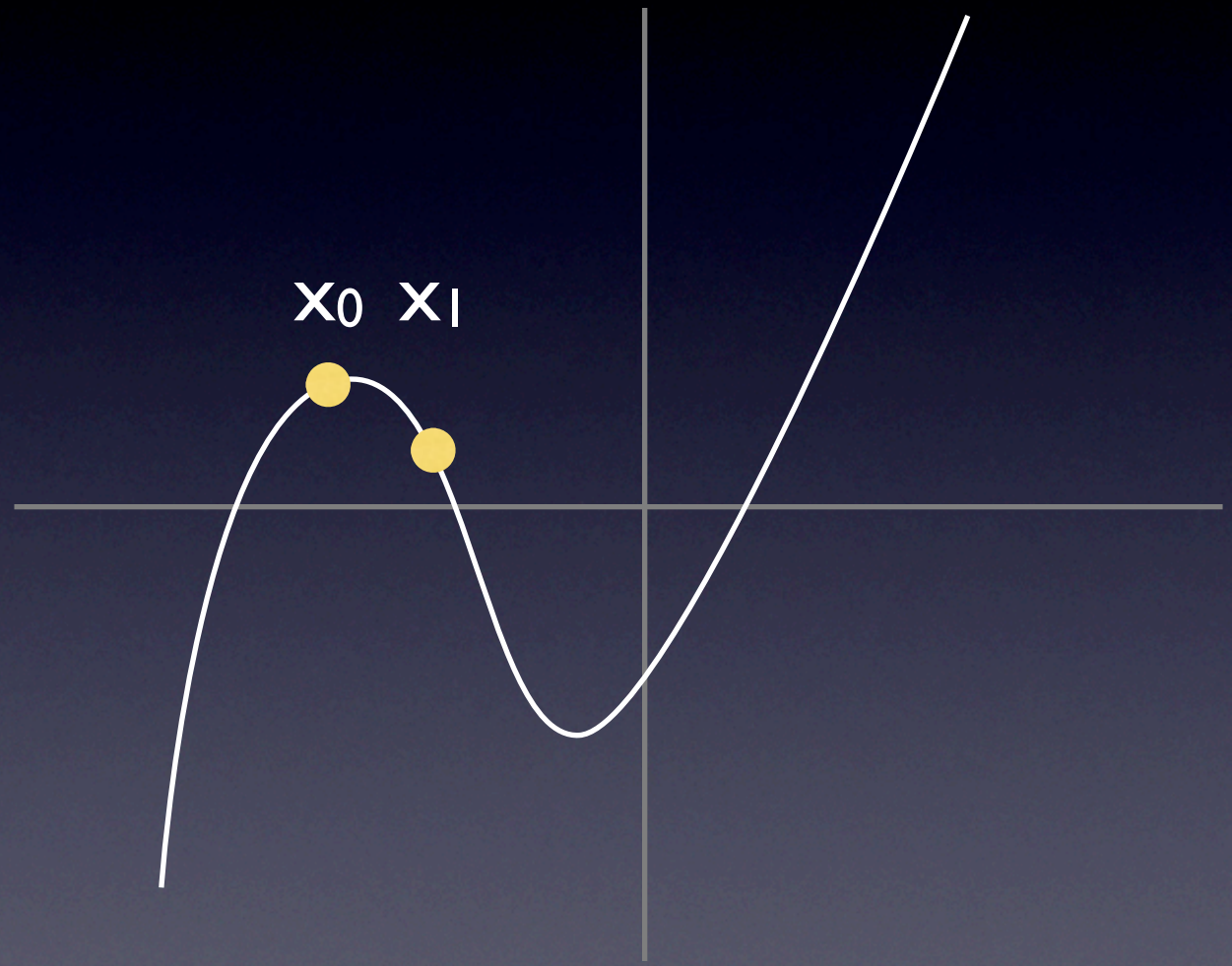
Secant Method

- Replace derivative evaluation in Newton's Method with the secant approximation of the derivative using the last two points
- Requires two values for initial guess
  - But no derivative evaluations

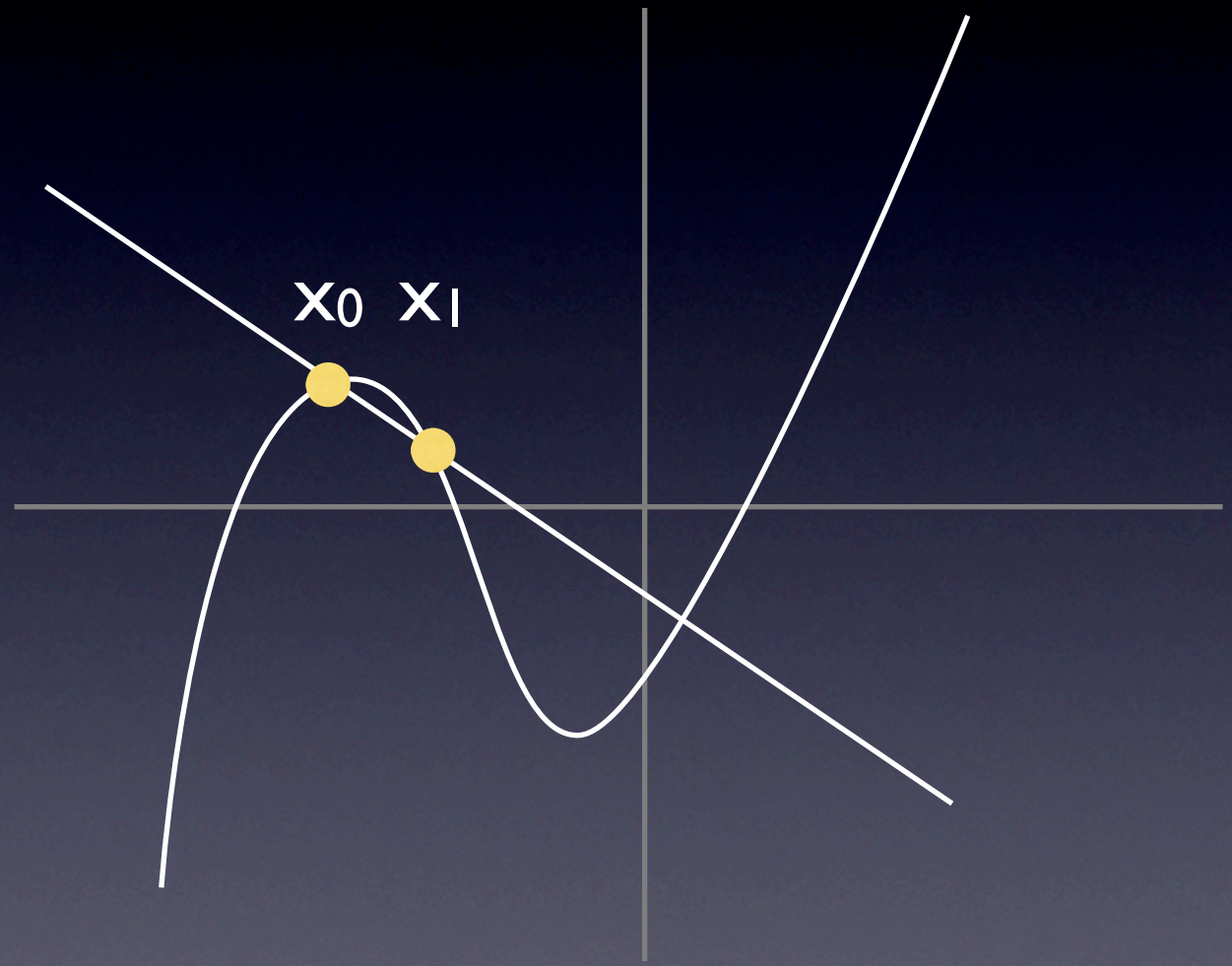
Secant Method

- x_0 = initial guess
  x_1 = initial guess
  for i = 1, 2, ...
    $s_{i+1} = (f(x_i) - f(x_{i-1}))/(x_i - x_{i-1})$
    $x_{i+1} = x_i - f(x_i)/s_{i+1}$
  end

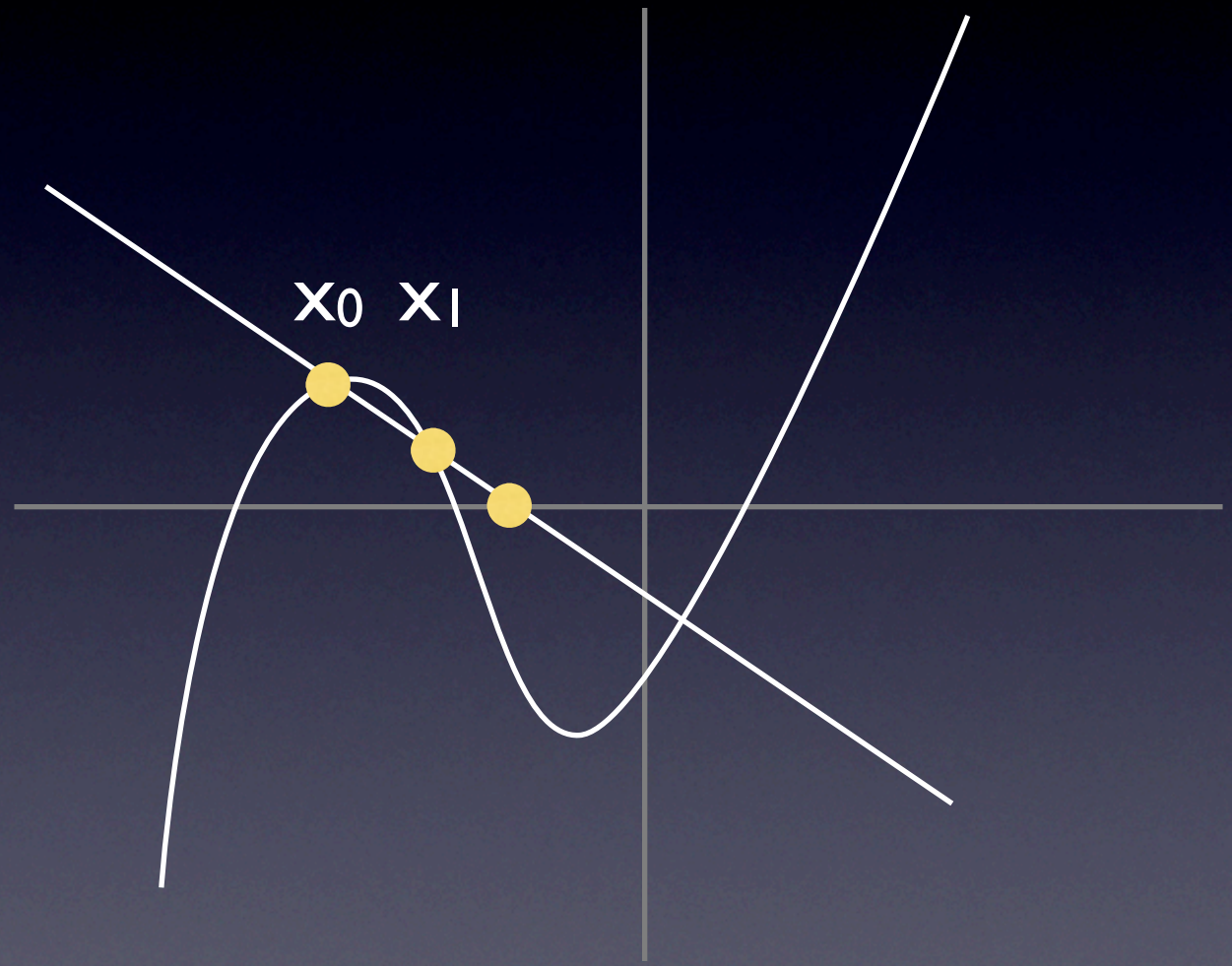
Secant Method



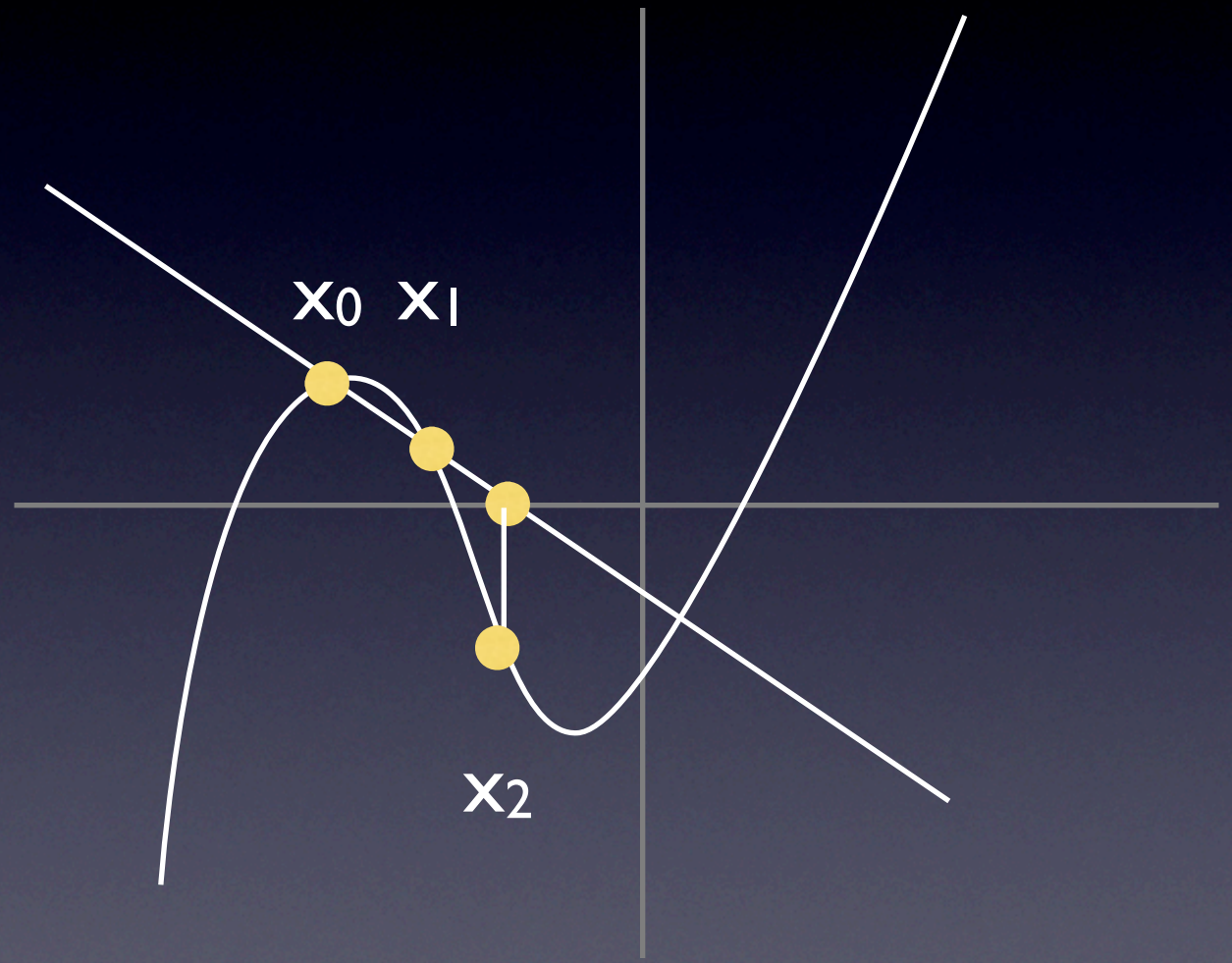
Secant Method



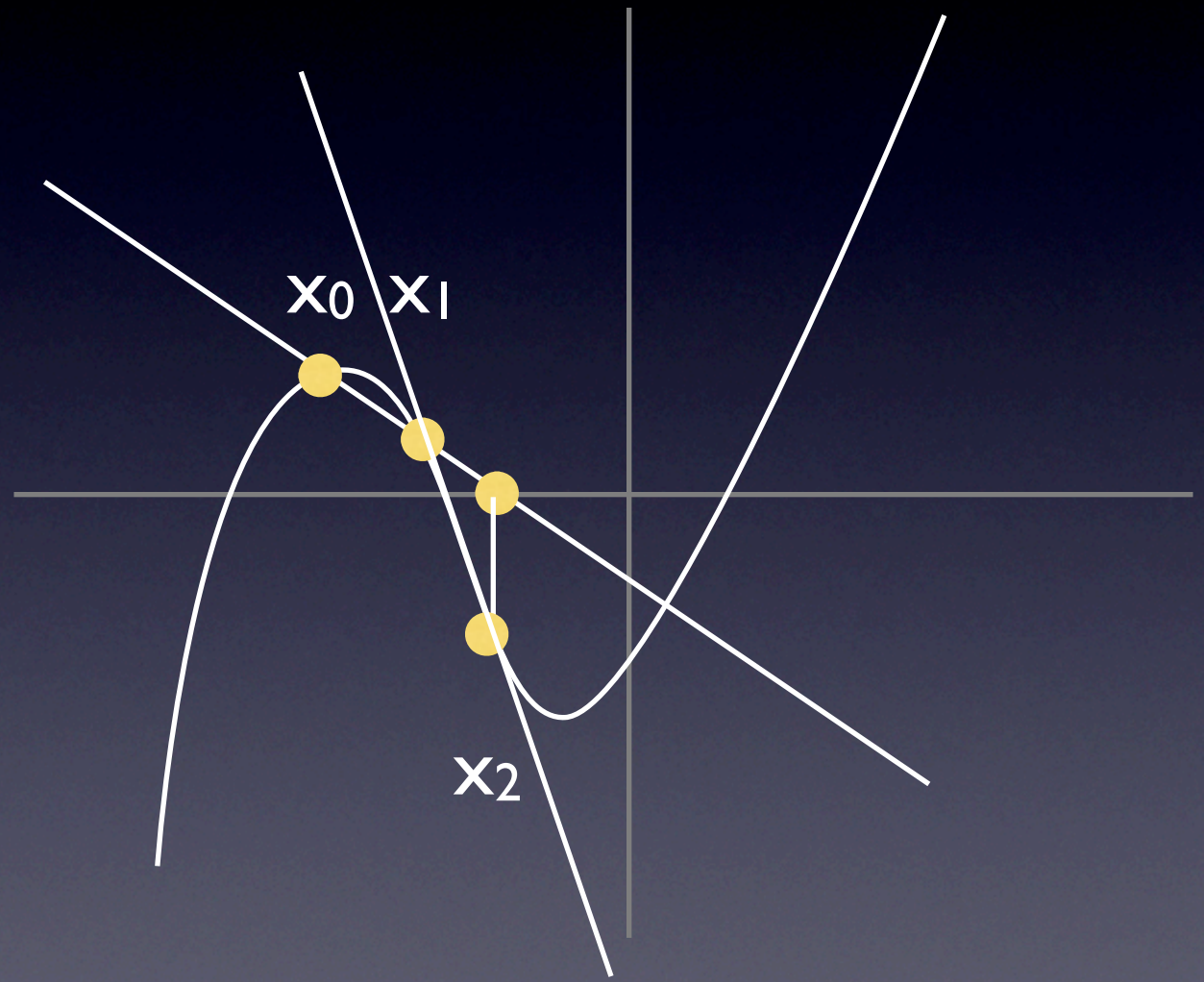
Secant Method



Secant Method



Secant Method



Secant Method

- Convergence of Secant Method is not as fast as Newton's Method, but faster than bisection
  - Order of convergence:
    $|r - x_{n+1}| \leq c\,|r - x_n|^{\phi}$ where
    $\phi = (\sqrt{5} + 1)/2 \approx 1.6$ is the golden ratio
  - Superlinear (but not quadratic) convergence

Secant Method

- As a bonus, we only have one function evaluation per iteration, instead of one + a derivative evaluation. This means that despite its slower convergence, secant can still be faster than Newton's in some cases

Drawbacks of Secant Method

- Computation of the secant involves division by a (hopefully) small value: dividing by $s_{i+1} = (f(x_i) - f(x_{i-1}))/(x_i - x_{i-1})$
- s_{i+1} is small when $f(x_i)$ is close to $f(x_{i+1})$, which causes $1/s_{i+1}$ to be very large. Ideally, this happens as we are already close to the root (when x_i is close to x_{i+1}) and so we are approximating the derivative in the limit

Drawbacks of Secant Method

- Of course, we have floating point issues to deal with
  - Can also have catastrophic cancellation
  - Not much we can do about it, other than be careful with our termination conditions

Hybrid Methods

- We have a slow but sure method (Bisection) and a fast but somewhat dangerous method (Newton / Secant). Can we combine the two of them and gain the benefits of both?

Hybrid Methods

- Start with a bracketing interval [a,b]. At each iteration:

  Take a (Newton / Secant / ...) step
  If x_{i+1} is inside [a,b], move there
  If xi+1 is outside [a,b], take a bisection step
  and update interval, try faster method
  again

Hybrid Methods

- We always make progress with this method, and we never lose sight of the root, but when we are close enough to the root we will enjoy the rapid convergence of our high speed method.

Other Details

- Suppose we want to find *all* the roots
  - Problematic in general
- Common restriction: find all the roots of a polynomial p
  - Find one root r_1, and then compute p / (x - r_1), find a root there, repeat

Roots in MATLAB

- For general root finding, use fzero(f, x0), which finds a root of the function f using x0 as an initial guess
- How do we specify the function f?

Sidestep: Fun with Functions

- We need some way to specify a function to be called by fzero()
- Suppose we have written an m-file called myFunction.m that takes a single argument and returns a single value. We need to tell fzero how to call this m-file
- Specify it using a function handle: fzero(@myFunction, x0)

Sidestep: Fun with Functions

- @ symbol: convert the specified function into a *function handle*
- This turns it into something like a variable, but you can call it as a function:

```
function c = myZero(f, x, a, b)
   fa = f(a);
   ...
```

Sidestep: Fun with Functions

- Still, it's annoying to have to create an m-file for every function we want to experiment with, particularly for simple functions
- Solution: anonymous functions
  f = @(x) x^2 - 1;
- This creates a function handle to a function that takes a single argument (x) and evaluates x^2 - 1

Sidestep: Fun with Functions

- We can create anonymous functions with more than one argument, and they can return scalars, vectors, matrices, etc
- However, the function body must be a single valid MATLAB expression - anything more complex than that requires an m-file

Roots in MATLAB

- So for ‘simple’ functions, we can use something like:
  X = fzero(@(x) x^2 - 1, 0.5)
- If we have an m-file called myFunc.m, then we can use:
  X = fzero(@myFunc, 0.5)

Roots in MATLAB

- For finding all of the roots of a polynomial p, we can use roots(pv), where pv is a vector consisting of the coefficients of the polynomial
- Returns all roots, using a method different than discussed here (but still iterative)