

# Useful websites

- MATLAB

- <http://www.mathworks.com/academia/>
- <http://www.mathworks.com/matlabcentral/>
- <http://blogs.mathworks.com/loren/>
- <http://blogs.mathworks.com/cleve/>

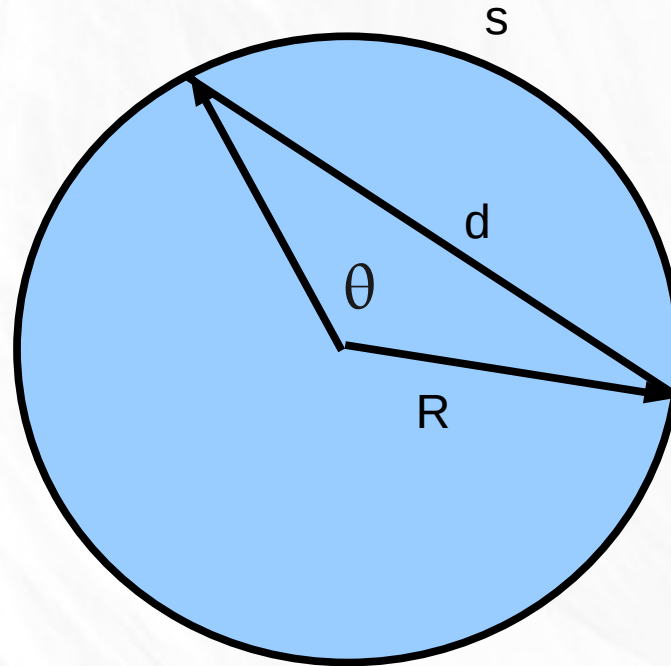
- Math

- <http://math.stackexchange.com/>

- Fun

- <https://news.ycombinator.com/news>
-

# Next: A “trivial” circle problem



$s$  = arc  
 $d$  = chord  
 $R$  = radius  
 $\theta$  = angle

Find  $\theta$  for which  $s = \frac{3}{2} d$

# Thinking about it...

- Consider limiting cases:

$$\theta = 0 \Rightarrow s = d = 0$$

$$\theta = \pi \Rightarrow s = \pi R, d = 2R$$

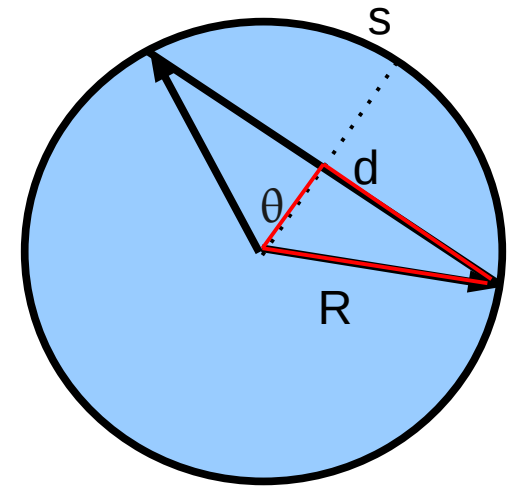
- Our desired  $s = \frac{3}{2} d$  lies somewhere in the middle.

- Now find solution.

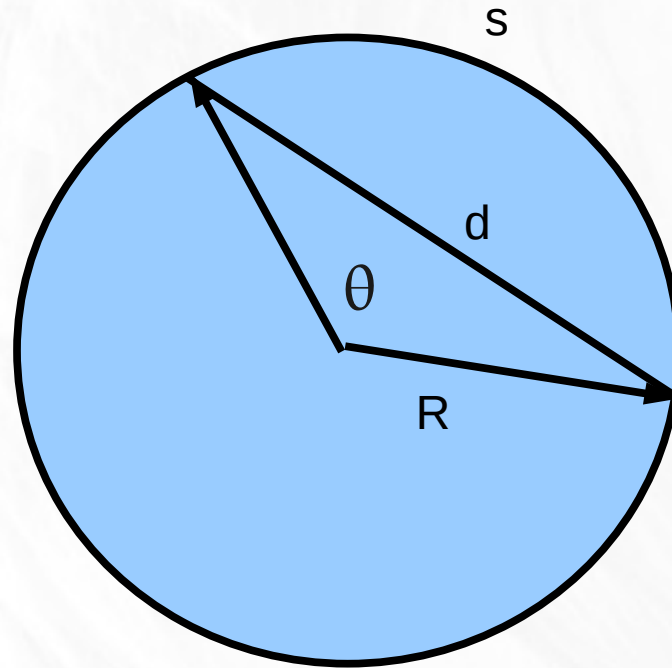
$$s = R \theta \quad \text{Arc length formula}$$

$$\sin\left(\frac{\theta}{2}\right) = \frac{d/2}{R} \Rightarrow d = 2R \sin\left(\frac{\theta}{2}\right) \quad \text{Trig relations for triangles}$$

$$s = \frac{3}{2} d \Rightarrow R \theta = \left(\frac{3}{2}\right) 2R \sin\left(\frac{\theta}{2}\right) \Rightarrow \theta = 3 \sin\left(\frac{\theta}{2}\right)$$



# Trivial problem



Find  $\theta$  for which  $s = 3/2 d$

$$\theta = 3 \sin(\theta/2) \quad \text{Now what?!?!?}$$

# The real world is complex

- There are many equations you can't solve analytically.
- They appear in engineering & science all the time.
- Numerical methods are your friend.

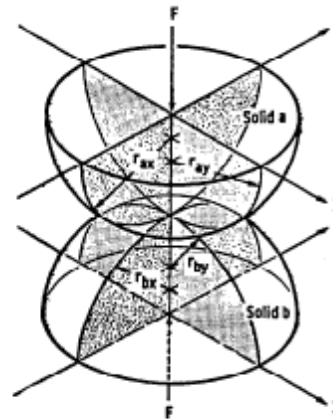


Figure 10. — Geometry of contacting elastic solids.

coordinate  $x$  then determines the direction of the semi-minor axis of the contact area when a load is applied and  $y$ , the direction of the semimajor axis. The direction of motion is always considered to be along the  $x$  axis.

A cross section of a ball bearing operating at a contact angle  $\beta$  is shown in figure 11. Equivalent radii of curvature for both inner- and outer-race contacts in, and normal to, the direction of rolling can be calculated from this figure. The radii of curvature for the *ball-inner-race* contact are

$$r_{ax} = r_{ay} = d/2 \quad (13)$$

$$r_{bx} = r_{by} = \frac{d_e - d \cos \beta}{2 \cos \beta} \quad (14)$$

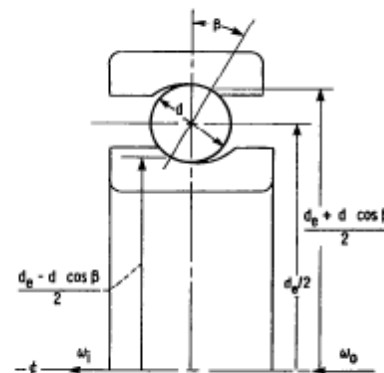


Figure 11. — Cross section of ball bearing.

$$r_{by} = -f_i d = -r_i \quad (15)$$

The radii of curvature for the *ball-outer-race* contact are

$$r_{ax} = r_{ay} = d/2 \quad (16)$$

$$r_{bx} = \frac{d_e + d \cos \beta}{2 \cos \beta} \quad (17)$$

$$r_{by} = -f_o d = -r_o \quad (18)$$

In equations (14) and (17),  $\beta$  is used instead of  $\beta_f$  since these equations are also valid when a load is applied to the contact. By setting  $\beta = 0^\circ$ , equations (13) to (18) are equally valid for radial ball bearings. For thrust ball bearings,  $r_{bx} = \infty$  and the other radii are defined as given in the preceding equations.

The curvature sum and difference, which are quantities of some importance in the analysis of contact stresses and deformations, are

$$\frac{1}{R} = \frac{1}{R_x} + \frac{1}{R_y} \quad (19)$$

$$\Gamma = R \left( \frac{1}{R_x} - \frac{1}{R_y} \right) \quad (20)$$

where

$$\frac{1}{R_x} = \frac{1}{r_{ax}} + \frac{1}{r_{bx}} \quad (21)$$

$$\frac{1}{R_y} = \frac{1}{r_{ay}} + \frac{1}{r_{by}} \quad (22)$$

$$\alpha = R_y / R_x$$

Equations (21) and (22) effectively redefine the problem of two ellipsoidal solids approaching one another in terms of an equivalent ellipsoidal solid of radii  $R_x$  and  $R_y$  approaching a plane. From the radius-of-curvature expressions, the radii  $R_x$  and  $R_y$  for the contact example discussed earlier can be written for the *ball-inner-race* contact as

$$R_x = \frac{d(d_e - d \cos \beta)}{2d_e} \quad (23)$$

$$R_y = \frac{f_i d}{2f_i - 1} \quad (24)$$

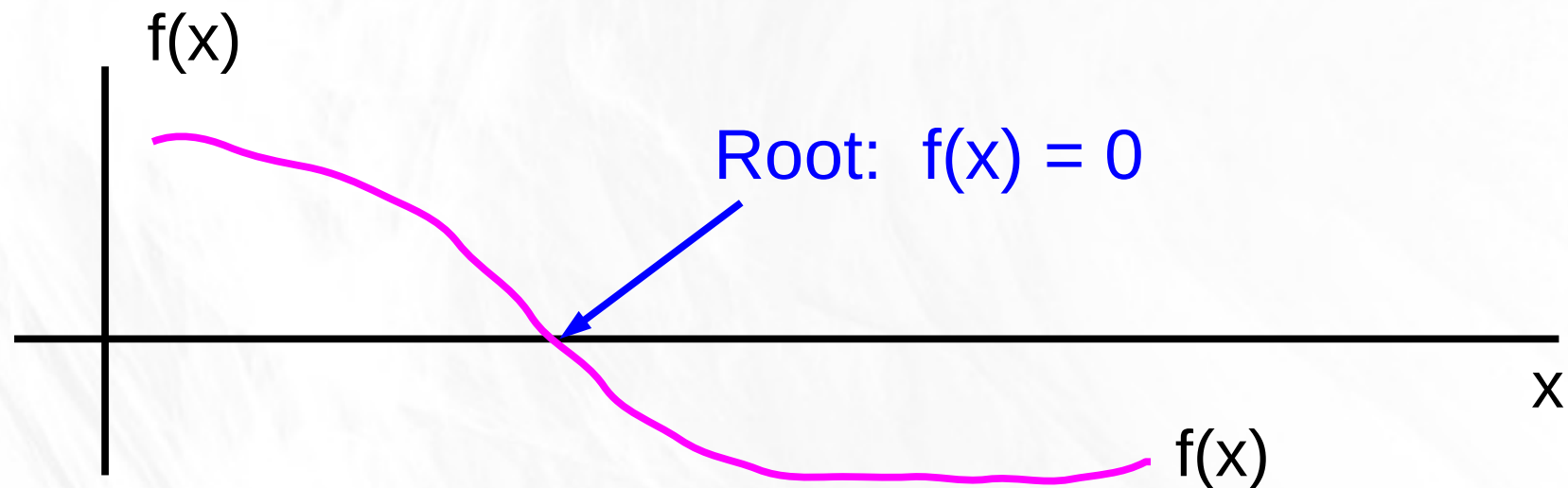
and for the *ball-outer-race* contact as

$$R_x = \frac{d(d_e + d \cos \beta)}{2d_e} \quad (25)$$

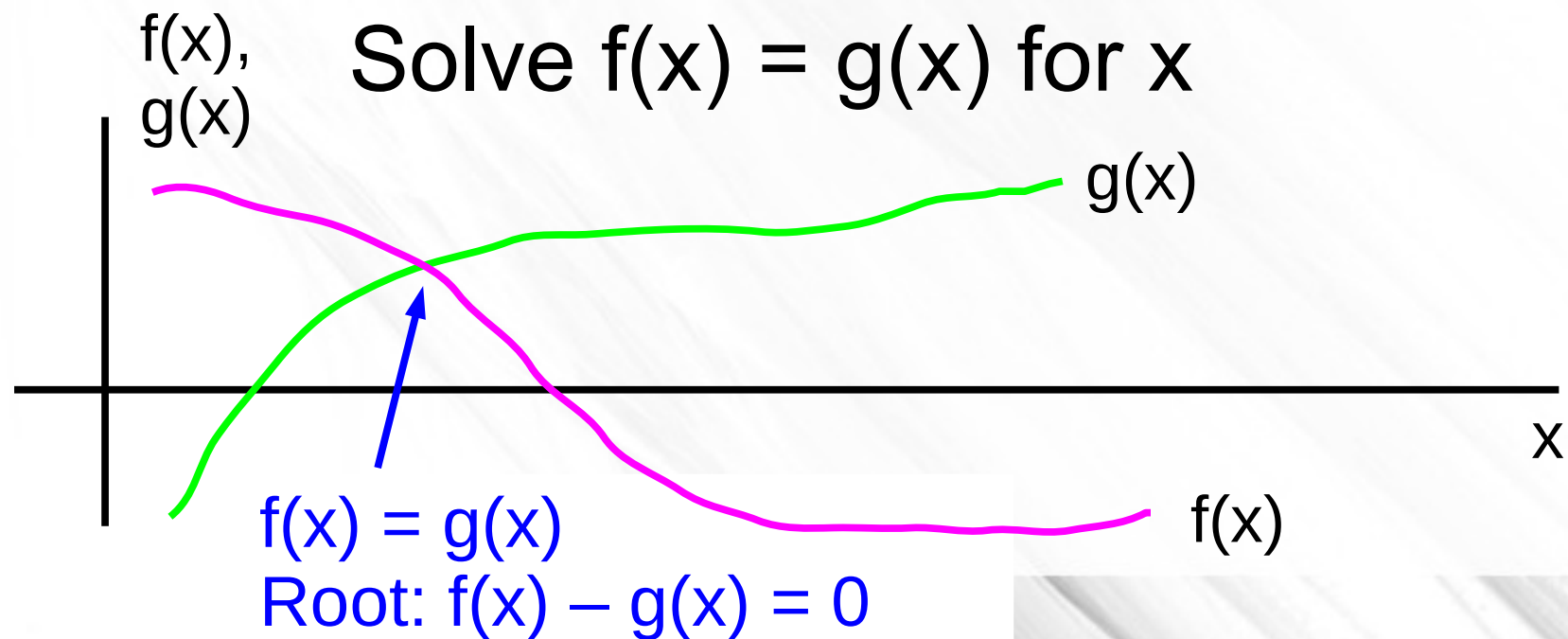
# Numerical root finding

- Find  $x$  which satisfies  $f(x) = 0$ .
- Variants:
  - Scalar  $x$  and scalar  $f$
  - Vector  $x$  and vector  $f$  – system of equations.
  - Optimization: Vector  $x$  and scalar  $f$  – multivariate  $f(x)$ .

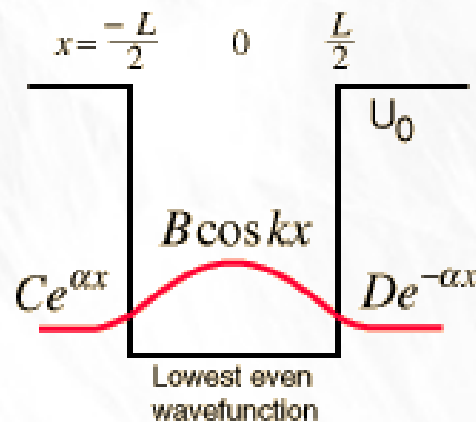
Root finding: Solve  $f(x) = 0$  for  $x$



Solve  $f(x) = g(x)$  for  $x$



# Example problem: 1D Quantum particle in a finite potential well



The condition of continuity for the wavefunction at the boundaries gives:

$$Ce^{-\alpha L/2} = B \cos(-kL/2) \quad \text{so } C = D$$

$$De^{-\alpha L/2} = B \cos(kL/2)$$

The condition of continuity for the derivative of the wavefunction gives:

$$\alpha Ce^{-\alpha L/2} = -kB \sin(-kL/2)$$

$$\alpha De^{-\alpha L/2} = -kB \sin(kL/2)$$

Dividing either of these two sets gives:

$$\alpha = k \tan \frac{kL}{2} = \sqrt{\beta^2 - k^2}$$

To get the wavefunction,  $\beta$  is known, and we must solve for  $k$ :

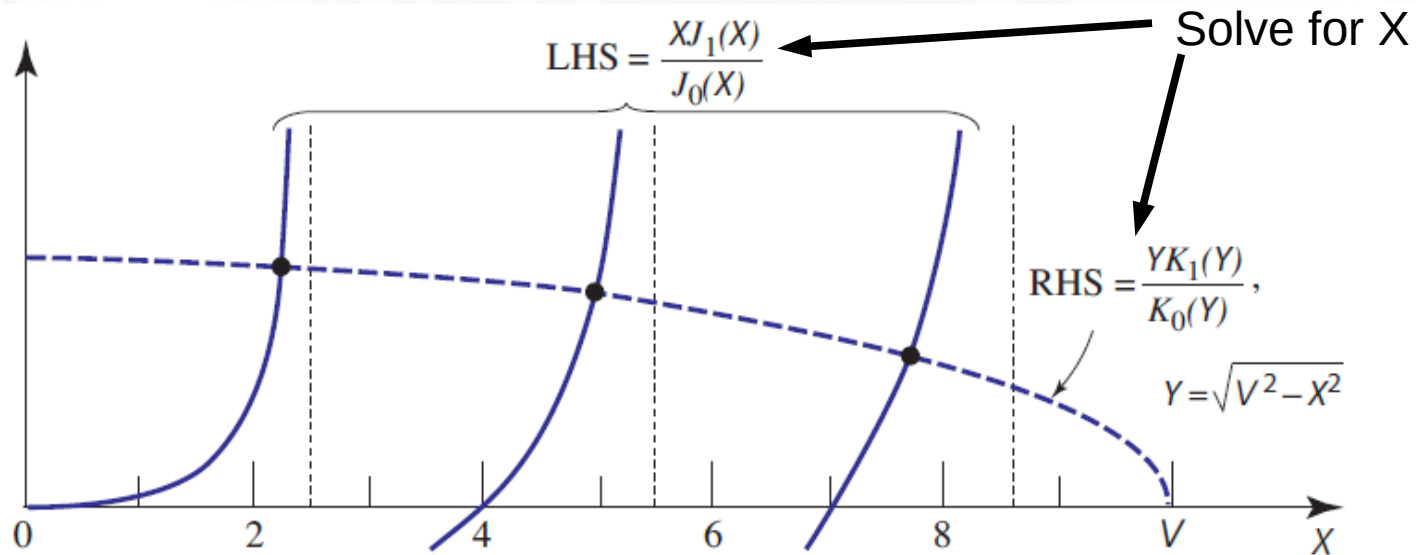
$$f(k) = k \tan \left( \frac{kL}{2} \right) - \sqrt{(\beta^2 - k^2)} = 0$$



# Example: Propagation of Light in Fibers

For weakly guiding fibers the characteristic equation obtained using the procedure outlined earlier turns out to be approximately equivalent to the conditions that the scalar function  $u(r)$  in (9.2-6) is continuous and has a continuous derivative at  $r = a$ . These two conditions are satisfied if

$$\frac{(k_T a) J_l'(k_T a)}{J_l(k_T a)} = \frac{(\gamma a) K_l'(\gamma a)}{K_l(\gamma a)}. \quad (9.2-11)$$



**Figure 9.2-3** Graphical construction for solving the characteristic equation (9.2-14). The left- and right-hand sides are plotted as functions of  $X$ . The intersection points are the solutions. The LHS has

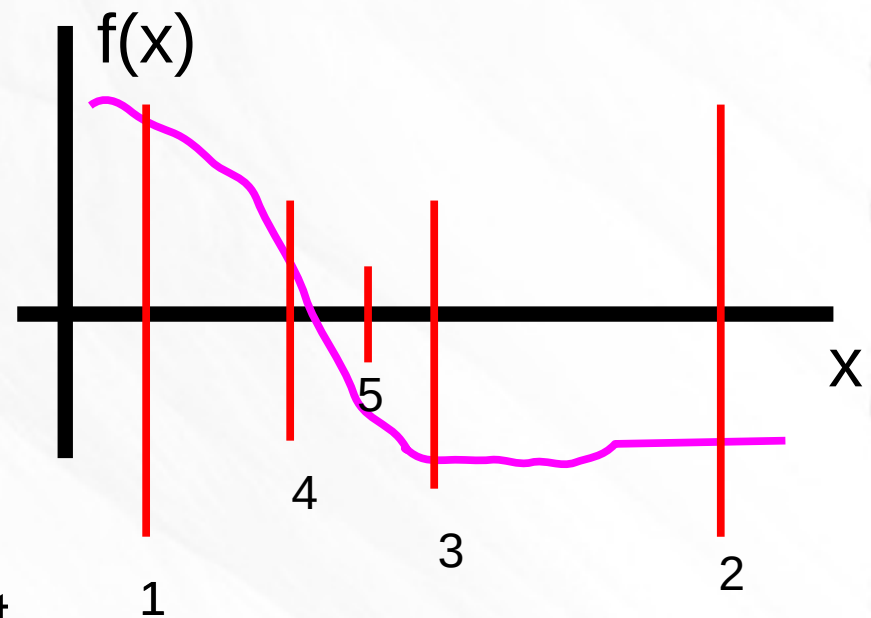
From "Fundamentals of Photonics" by Salah and Teich (Wiley, 1991)

# Preliminary points about root finding with a computer

- Requirement: The function whose roots you want is available – you can call it. (That is, it is not just discrete data.)
- Most methods assume continuity in both function and at least one of the derivatives.
- Useful for non-linear (as opposed to linear) equations.
- The methods are usually iterative – they iterate while finding ever closer approximations to the root.
- Therefore, you generally need to specify a tolerance which determines when you can quit because you're “close enough” to the actual root.

# Root Finding -- Bisection method

1. Make initial guess. Choose two end points which return opposite signs for  $f(x)$ .  
Example,  $f(\text{left}) = +$ ,  $f(\text{right}) = -$
2. Find mid-point  $x = \text{new}$ .
3. If  $\text{sign}(f(\text{left})) \neq \text{sign}(f(\text{new}))$ 
  - Then left & new bracket root. Right = new.
  - Else new & right bracket root. Left = new.
4. Loop until  $\text{right} - \text{left} < \text{tolerance}$ .
5. Return  $(\text{right} + \text{left})/2$



# Octave example: bisection.m

```
function x0 = bisection(f, a, b, tol)
% This fcn implements simple bisection
% method root finding.
% Call like this:
% f = @(x) x^3 - 2*x + 5*x - 7
% bisection(f, 1, 3, 1e-5)
```

Call with function handle,  
boundaries of domain, and  
stopping tol.

```
% Get signs of fcn at end points
sa = sign(f(a));
sb = sign(f(b));

% Validate input – check a > b
if (a >= b)
    error('Invalid input -- a >= b\n')
end

% Validate input – check signs are different
if (sa == sb)
    error('Invalid input -- sign(f(a)) == sign(f(b))\n')
end
```

```
% Do root finding in a loop to prevent infinite loops
for i = 1:100
```

```
    % Compute midpoint and sign of fcn at midpoint
    xm = (a+b)/2;          % midpoint
    sm = sign(f(xm));      % sign at midpoint
```

```
    % Now adjust end point depending upon sign(s)
    if (sa == sm)
        % sign(left) == sign(mid) => root on right side.
        a = xm;
        sa = sm;
    else
        % sign(right) == sign(mid) => root on left side.
        b = xm;
        sb = sm;
    end
```

```
    fprintf('a = %f,    b = %f\n', a, b);
```

```
    % Check if we're close enough to quit yet
    if (b-a) < tol
        printf('Terminating after %d iterations\n', i)
        x0 = (a+b)/2; % Return midpoint between a & b
        return
    end
```

```
end % end of for loop
```

```
printf('Terminated without convergence!\n')
x0 = nan;
```

```
end
```

# test\_bisection()

```
function test_bisection()
% This fcn tests my bisection method root finder.

% Define fcn whose roots I want to find. Define it using
% an anonymous function. Roots should be 3, 5, 7, 9.
f = @(x) (x-3).*(x-5).*(x-7).*(x-9);

% Initial endpoints of search -- should find root at 3.
a = 1;
b = 4;
r_exp = 3;

% Tolerance
tol = 1e-5;

r_comp = bisection(f, a, b, tol);
fprintf('====> Final answer = %f\n', r_exp)

if (abs(r_comp - r_exp) > tol)
    fprintf('Failure!\n')
else
    fprintf('Success!\n')
end

end
```

Function whose roots to find:

$$f(x) = (x-3)(x-5)(x-7)(x-9)$$

# Sample run

```
octave:13> test_bisection
```

a = 1.000000,	b = 4.000000,	diff = 3.000000
a = 2.500000,	b = 4.000000,	diff = 1.500000
a = 2.500000,	b = 3.250000,	diff = 0.750000
a = 2.875000,	b = 3.250000,	diff = 0.375000
a = 2.875000,	b = 3.062500,	diff = 0.187500
a = 2.968750,	b = 3.062500,	diff = 0.093750
a = 2.968750,	b = 3.015625,	diff = 0.046875
a = 2.992188,	b = 3.015625,	diff = 0.023438
a = 2.992188,	b = 3.003906,	diff = 0.011719
a = 2.998047,	b = 3.003906,	diff = 0.005859
a = 2.998047,	b = 3.000977,	diff = 0.002930
a = 2.999512,	b = 3.000977,	diff = 0.001465
a = 2.999512,	b = 3.000244,	diff = 0.000732
a = 2.999878,	b = 3.000244,	diff = 0.000366
a = 2.999878,	b = 3.000061,	diff = 0.000183
a = 2.999969,	b = 3.000061,	diff = 0.000092
a = 2.999969,	b = 3.000015,	diff = 0.000046
a = 2.999992,	b = 3.000015,	diff = 0.000023
a = 2.999992,	b = 3.000004,	diff = 0.000011
a = 2.999998,	b = 3.000004,	diff = 0.000006

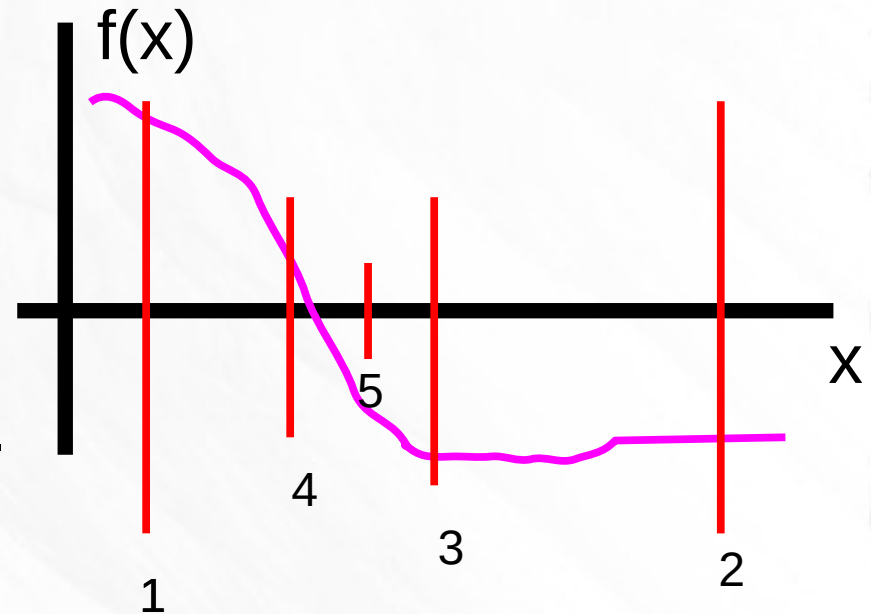
```
Terminating after 19 iterations
```

```
====> Final answer = 3.000001
```

```
Success!
```

# Remarks about the bisection method

- Requires two initial guesses which lie on opposite sides of root.
- Also needs stop criterion (tolerance).
- Guaranteed to converge. (Other methods can wander off to infinity.)
- Convergence is “linear”:  $err_{n+1} = K (err_n)^m$  with  $m=1$  and  $K=1/2$
- However, can converge to singularities as well as zeros.





# Linear convergence

- Wall separation decreases by  $\frac{1}{2}$  with each iteration.

a = 1.000000,	b = 4.000000,	diff = 3.000000
a = 2.500000,	b = 4.000000,	diff = 1.500000
a = 2.500000,	b = 3.250000,	diff = 0.750000
a = 2.875000,	b = 3.250000,	diff = 0.375000
a = 2.875000,	b = 3.062500,	diff = 0.187500

$$err_1 = err_0 / 2$$

$$err_2 = err_1 / 2$$

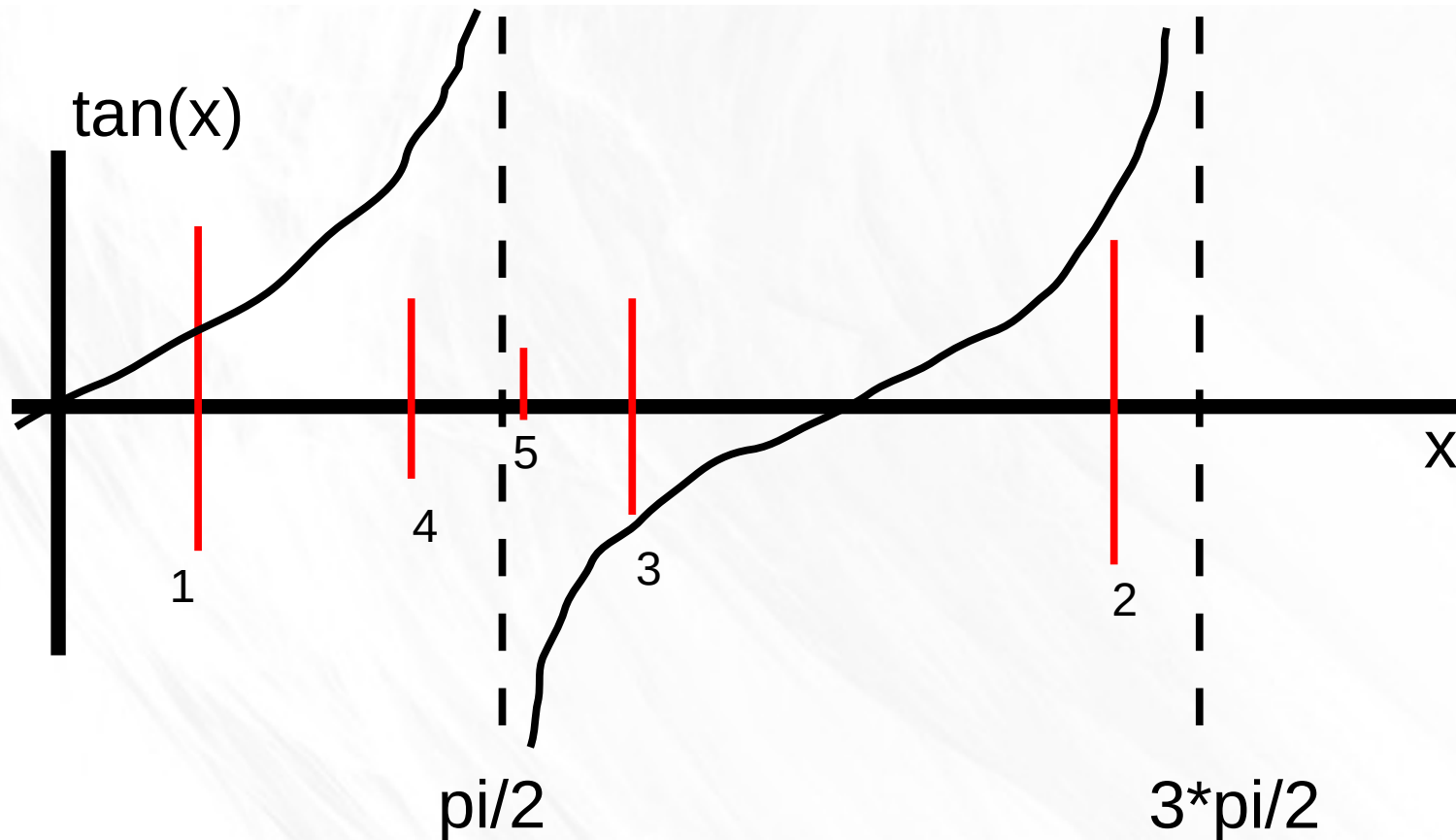
$$err_3 = err_2 / 2$$

$$err_{n+1} = K (err_n)^m$$

$$K = 1/2$$

$$m = 1 \quad \text{Linear convergence}$$

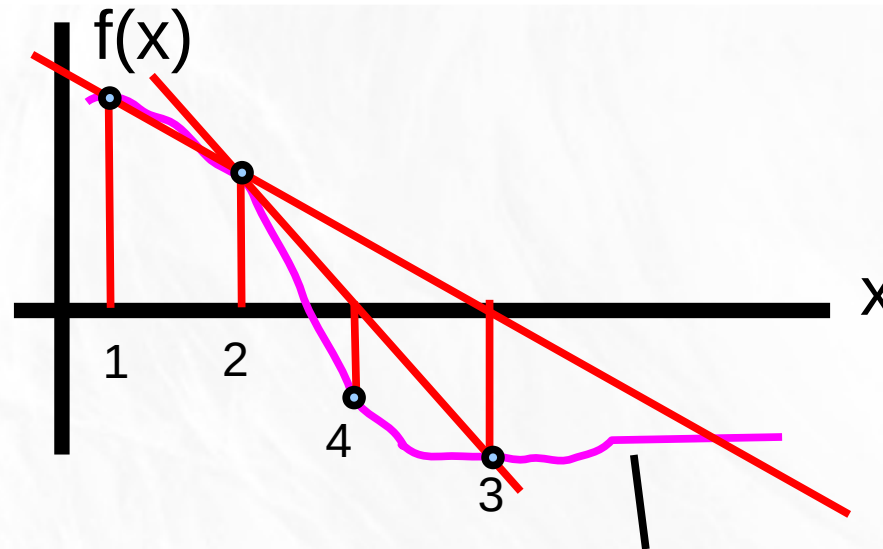
# Failure mode: consider $\tan(x)$



Depending upon your starting points, you can converge to the singularity, not the root.

**Be mindful of your problem!** It helps to plot your problem before running any code.

# Next topic: Secant method



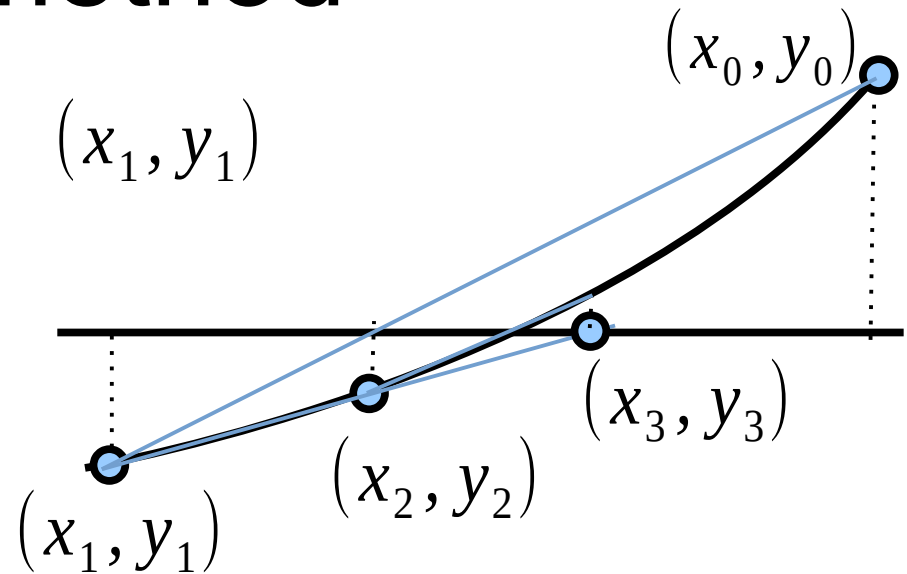
1. Choose 2 initial points on  $x$ . Close to root position is best.
2. Draw secant line (line connecting two points  $f(x_1)$ ,  $f(x_2)$ ).
3. Find intersection of secant line and  $x$  axis. This is the new end point.
4. Loop until  $\text{abs}(x_n - x_{n-1}) < \text{tolerance}$

# Secant method

- Start with points  $(x_0, y_0)$  &  $(x_1, y_1)$

- Draw connection line

$$(y - y_1) = \frac{y_1 - y_0}{x_1 - x_0} (x - x_1)$$



- Find point where it crosses zero.

$$\begin{array}{ccccc} \underset{\substack{\uparrow \\ \text{Zero crossing}}}{(0 - y_1)} = s \underset{\substack{\uparrow \\ \text{Solve for } x}}{(x - x_1)} & s = \frac{y_1 - y_0}{x_1 - x_0} & \Rightarrow & x_2 = x_1 - y_1 / s & \underset{\substack{\uparrow \\ \text{Zero crossing position}}}{x_2} \end{array}$$

- Draw new connection line, then find zero-crossing.

$$(y - y_2) = \frac{y_2 - y_1}{x_2 - x_1} (x - x_2) \quad \Rightarrow \quad s = \frac{y_2 - y_1}{x_2 - x_1} \quad x_3 = x_2 - y_2 / s$$

# Secant – general method

Seed method

$$y_0 = f(x_0)$$

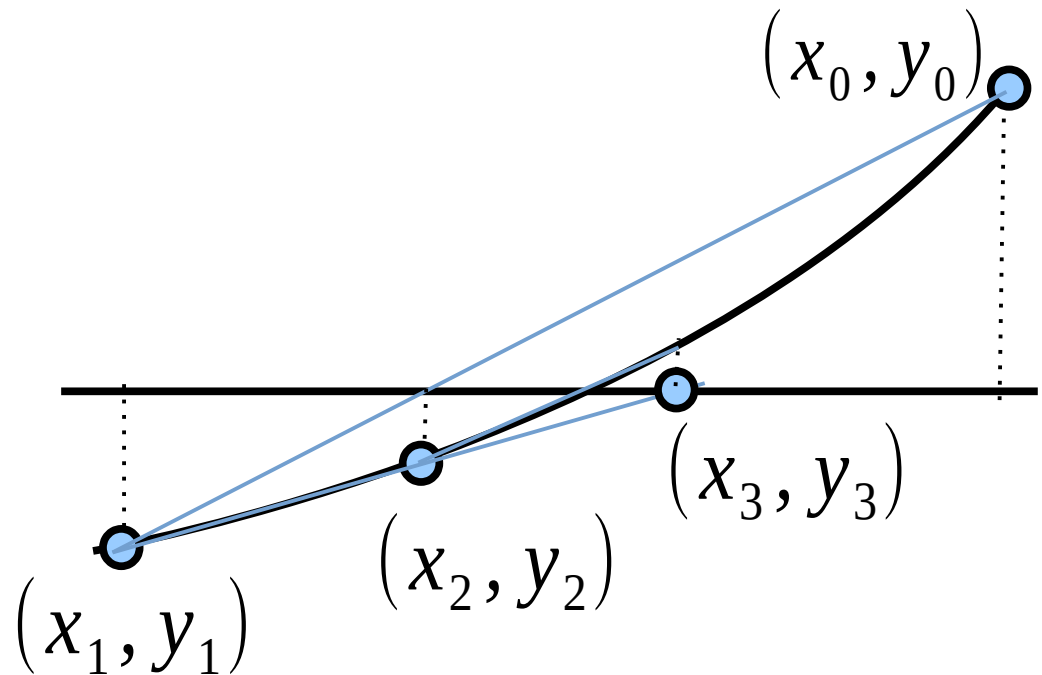
$$y_1 = f(x_1)$$

$$s = \frac{y_n - y_{n-1}}{x_n - x_{n-1}}$$

$$x_{n+1} = x_n - y_n / s$$


$$y_{n+1} = f(x_{n+1})$$




Move  $x_{n+1}, x_n, y_{n+1}$  and  $y_n$  back




```

function root = secant(f, x0, x1, tol)
    % This fcn implements root finding via secant method
    % Call like this:
    % f = @(x) x^3 - 2*x + 5*x -7
    % secant(f, 1, 3, 1e-5)

    % Initialize computation: compute values of f at x0, x1.
    f1 = f(x1); f0 = f(x0);  Seed the computation

    % Do root finding in a loop to prevent infinite loops from nonconvergence
    for i = 1:100
        s = (f1 - f0)/(x1 - x0);  Compute new slope
        x2 = x1 - (f1/s);  Compute new position
        f2 = f(x2);  Compute new y value
        printf('x0 = %f, x1 = %f, s = %f, x2 = %f\n', x0, x1, s, x2);

        % Now let's see if x1 and x2 are close enough to quit
        if abs(x2 - x1) < tol
            printf('Terminating after %d iterations because abs(x2-x1) < tol.\n', i)
            root = x2;
            return
        else
            x0 = x1; f0 = f1; x1 = x2; f1 = f2;  Move old values back
        end
    end % end of for loop

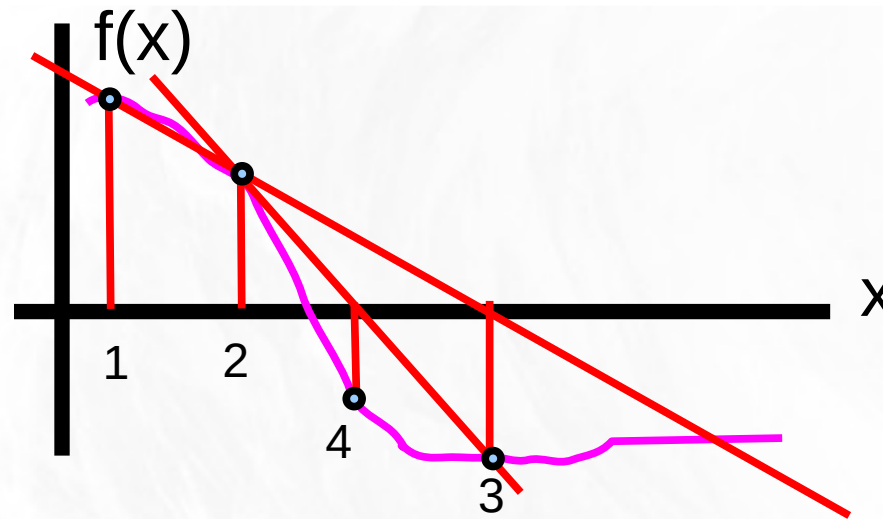
    error('Terminated without convergence!\n')
end

```

```
octave:19> test_secant
x0 = 1.000000, x1 = 4.000000, s = -133.000000, x2 = 3.887218
x0 = 4.000000, x1 = 3.887218, s = 6.317860, x2 = 6.374222
x0 = 3.887218, x1 = 6.374222, s = 9.381463, x2 = 5.562068
x0 = 6.374222, x1 = 5.562068, s = 0.615965, x2 = -5.995291
x0 = 5.562068, x1 = -5.995291, s = -1667.031729, x2 = 5.566338
x0 = -5.995291, x1 = 5.566338, s = -1666.412894, x2 = 5.570632
x0 = 5.566338, x1 = 5.570632, s = 8.308893, x2 = 4.705244
x0 = 5.570632, x1 = 4.705244, s = 14.033080, x2 = 5.058241
x0 = 4.705244, x1 = 5.058241, s = 16.632284, x2 = 5.003077
x0 = 5.058241, x1 = 5.003077, s = 15.740614, x2 = 4.999952
x0 = 5.003077, x1 = 4.999952, s = 15.987849, x2 = 5.000000
Terminating after 11 iterations because f(x2) < tol.
====> Final answer = 5.000000
```

- Converges in 11 iterations. (Bisection converged in 19.)
- But converges to different root!
- Observe how it wanders around, looking for a root.

# Secant method – remarks



- Requires two initial guesses.
- Converges, but only if initial guesses are “close” to root.
- Also need tolerance to know when to stop.
- Faster than linear convergence.  $err_{n+1} = K (err_n)^\alpha$ 
  - Golden ratio:  $\alpha = (\sqrt{5} + 1)/2 = 1.618$
- Convergence is not guaranteed. It might not find the root you want. It might cycle between values. It might wander forever.



# Next: Newton's Method (scalar)

- Also called Newton-Raphson method.
- Derivation on blackboard.
- Very common method.
- Convergence is quadratic – very fast!

$$err_{n+1} = K (err_n)^m \quad \text{with } m=2$$

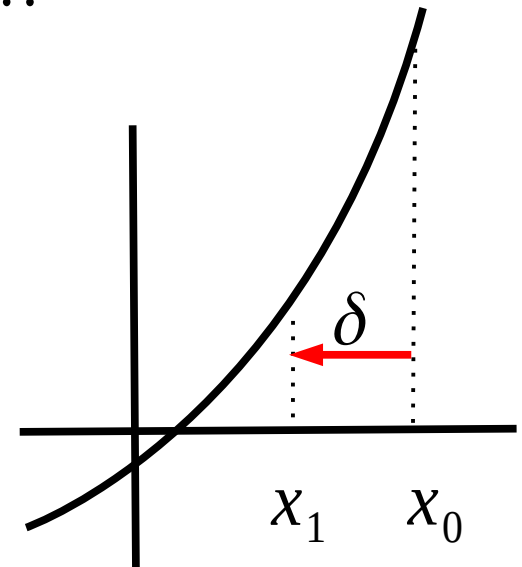
- However, works best when given an initial starting point **close** to the actual root.
  - Otherwise, it can wander off to infinity.
- Also, requires expressions for both the function *and* its **derivative**.

# First derivation

- Start with Taylor's series expansion for  $f(x_0 + \delta)$

$$f(x_0 + \delta) = f(x_0) + \delta \left. \frac{df}{dx} \right|_{x=x_0} + \frac{\delta^2}{2} \left. \frac{d^2 f}{dx^2} \right|_{x=x_0} + \dots$$

- Suppose I am standing at  $x_0$ , and I want to find the  $x$  where  $f(x)=0$ . How far should I step?



- Use Taylor's series and ignore all terms above linear to get

The point where  $f(x)$  should be zero.

$$f(x_0 + \delta) \approx 0 \approx f(x_0) + \delta \left. \frac{df}{dx} \right|_{x=x_0}$$

- From last page,

$$f(x_0 + \delta_0) \approx 0 \approx f(x_0) + \delta_0 \left. \frac{df}{dx} \right|_{x=x_0}$$

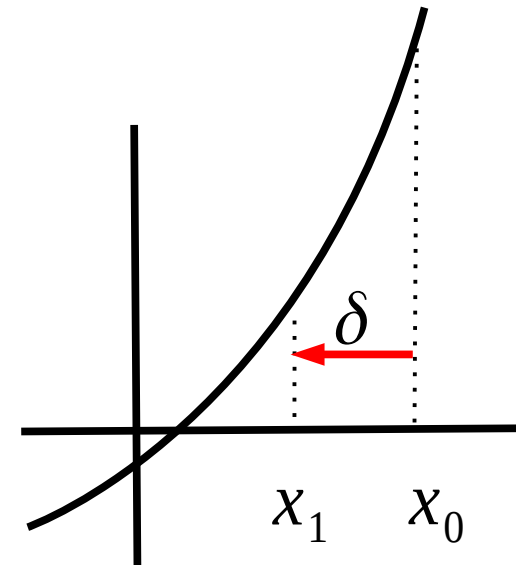
Note that I snuck a subscript into delta.

- Solve to get step

$$\delta_0 = -f(x_0) / \left. \frac{df}{dx} \right|_{x=x_0}$$

- Take step

$$x_1 = x_0 + \delta_0$$



- This won't take us immediately to the root, but what if we do it again? And again?

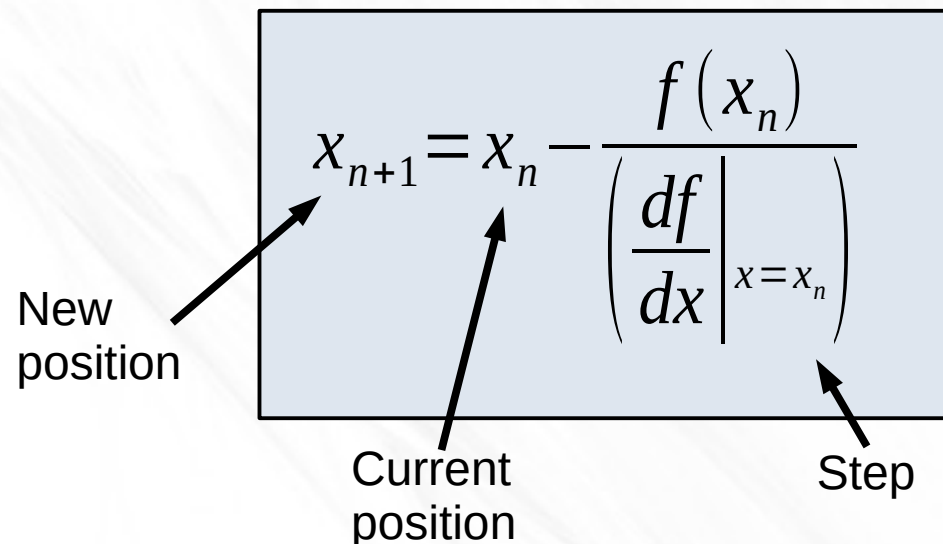
$$x_1 = x_0 + \delta_0 \quad \delta_1 = -f(x_1) / \left. \frac{df}{dx} \right|_{x=x_1} \quad x_2 = x_1 + \delta_1 \quad \text{etc.....}$$

# Newton's method in 1 dimension

- Compute step, then take it.

$$\delta_n = -f(x_n) / \left. \frac{df}{dx} \right|_{x=x_n} \quad x_{n+1} = x_n + \delta_n$$

- Combining the operations,



A light blue rectangular box contains the equation  $x_{n+1} = x_n - \frac{f(x_n)}{\left(\left.\frac{df}{dx}\right|_{x=x_n}\right)}$ . Three arrows point from text labels outside the box to parts of the equation: 'New position' points to  $x_{n+1}$ , 'Current position' points to  $x_n$ , and 'Step' points to the fraction term.

$$x_{n+1} = x_n - \frac{f(x_n)}{\left(\left.\frac{df}{dx}\right|_{x=x_n}\right)}$$

New position

Current position

Step

# Newton's Method -- algorithm

1. Select initial guess,  $x_0$ .
2. Compute next approximation,  $x_1$

$$x_1 = x_0 - \frac{f(x)}{f'(x)} \Big|_{x=x_0}$$

3. If  $\text{abs}(x_1 - x_0) < \text{tolerance}$ , return  $x_1$   
else loop again.

4. In general, iteration expression is

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

# Second derivation

- Want to shoot tangent lines from curve at  $(x_0, y_0)$ .

- Equation of line:  $(y_1 - y_0) = s(x_1 - x_0)$   
 $\swarrow \quad \searrow$   
 $= 0$   
 Slope

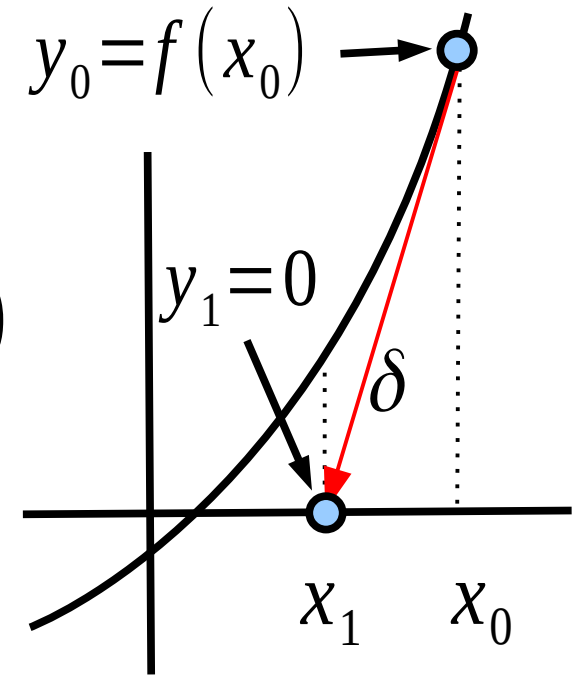
- Slope:  $s = \left( \frac{df}{dx} \right)_{x=x_0}$

- So,  $0 - f(x_0) = \left( \frac{df}{dx} \right)_{x=x_0} (x_1 - x_0)$

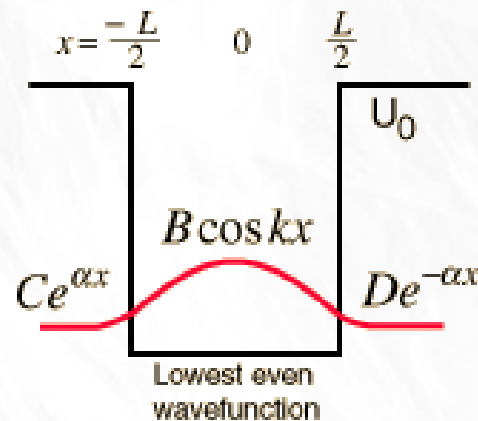
Looks like secant method except we use derivative to get slope.

- Or,  $x_1 = x_0 - \frac{f(x_0)}{\left( \frac{df}{dx} \right)_{x=x_0}}$

Same expression as before.



# Recall this example: 1D Quantum particle in a finite potential well



The condition of continuity for the wavefunction at the boundaries gives:

$$Ce^{-\alpha L/2} = B \cos(-kL/2) \quad \text{so } C=D$$

$$De^{-\alpha L/2} = B \cos(kL/2)$$

The condition of continuity for the derivative of the wavefunction gives:

$$\alpha Ce^{-\alpha L/2} = -kB \sin(-kL/2)$$

$$\alpha De^{-\alpha L/2} = -kB \sin(kL/2)$$

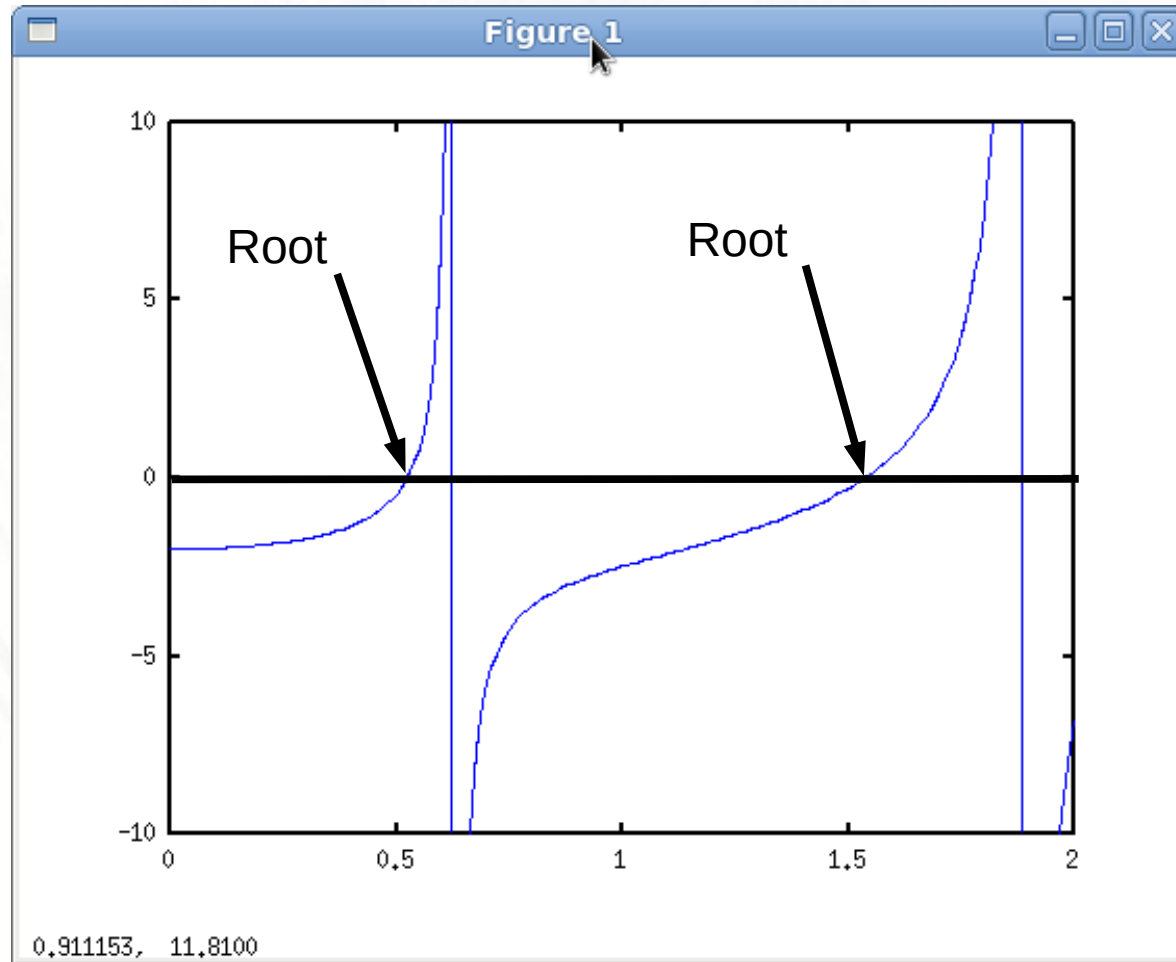
Dividing either of these two sets gives:

$$\alpha = k \tan \frac{kL}{2} = \sqrt{\beta^2 - k^2}$$

To get the wavefunction, we must solve for k:

$$f(k) = k \tan\left(\frac{kL}{2}\right) - \sqrt{\beta^2 - k^2} = 0$$

# Graphically

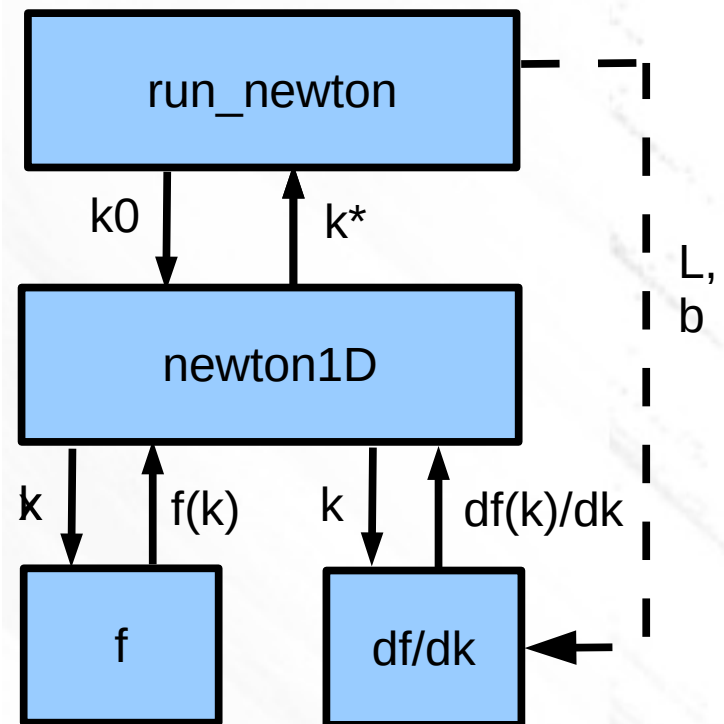


$$f(k) = k \tan\left(\frac{kL}{2}\right) - \sqrt{\beta^2 - k^2}$$



# Example written in Octave

- `~/Newton1D/  
run_newton1D.m`
- Run it using `run_newton(k)`,  
where  $k$  is starting point.
- Try for different start points:
  - 0.5
  - 1.5
  - 0.0001



# Implementing Newton's Method

1. If you can, graph  $f(k)$  to get some idea of where to look for roots. Helps to find initial guess!
2. We need both  $f(k)$  and  $f'(k)$ . Therefore, derive  $f'(k)$ . Then program both functions.
3. Write root finder in separate function.
4. Use bracketing to make sure your solution doesn't run away.
5. Always check that successive steps are decreasing

Look at `newton1D.m` and `run_newton1D.m`

# Convergence criteria

- Criterion on  $x$ :  $|x_{n+1} - x_n| < tol$
- Criterion on  $f$ :  $|f_{n+1} - f_n| < tol$
- Also want successive differences to be decreasing series:

$$|x_1 - x_0| > |x_2 - x_1| > |x_3 - x_2| \cdots$$

$$|f_1 - f_0| > |f_2 - f_1| > |f_3 - f_2| \cdots$$

- It's a good idea to check these difference at each step to make sure Newton's method is not wandering away.

# Backtracking

- Divergence of iteration usually due to taking steps which are too large.
- Therefore, implement mechanism which decreases step size to ensure convergence.
- Add parameter  $\alpha$  which controls step size.

$$x_{n+1} = x_n + \alpha p_n$$

$$p_n = - \left. \frac{f'(x)}{f''(x)} \right|_{x=x_n}$$

# Simple backtracking algorithm

Main loop

1. Set  $\alpha=1$

2. Compute Newton direction  $p_n = -\frac{f'(x)}{f''(x)} \Big|_{x=x_n}$

3. If  $|f(x_n + \alpha p_n)| < |f(x_n)|$  % Check for decreasing |f|

Then step:  $x_{n+1} = x_n + \alpha p_n$

Else, set:  $\alpha = \alpha/2$  , then go to 3

4. If  $|x_{n+1} - x_n| < tol$

Then converged: return  $x_{n+1}$

Else: loop again

Look at run\_newton\_backtracking1D.m

```
octave:44> run_newton1D(.1)
x = 0.10000000000000
x = 3.271548421552, delta = 3.171548421552
x = 3.395969811977, delta = 0.124421390425
x = 3.625554663918, delta = 0.229584851941
x = 3.722906393156, delta = 0.097351729239
x = 3.355716999951, delta = -0.367189393205
x = 0.195981160702, delta = -3.159735839249
x = 1.145170579750, delta = 0.949189419047
x = 1.111549154353, delta = -0.033621425397
x = 1.541883388794, delta = 0.430334234441
x = 1.260749161931, delta = -0.281134226863
x = 1.582606587359, delta = 0.321857425428
x = 1.540304245327, delta = -0.042302342032
x = 1.535373875671, delta = -0.004930369656
x = 1.535028645085, delta = -0.000345230586
x = 1.534991889236, delta = -0.000036755849
x = 1.534987461292, delta = -0.000004427944
x = 1.534986918594, delta = -0.000000542698
Terminating after 17 iterations because abs(delta) < tol.
root found at 1.534986918594
octave:45>
octave:45>
octave:45> run_newton_backtracking1D(.1)
x = 0.10000000000000
x = 0.35000000000000, delta = 0.25000000000000
x = 0.560460271405, delta = 0.210460271405
x = 0.536132040143, delta = -0.024328231262
x = 0.524390913752, delta = -0.011741126391
x = 0.522626542706, delta = -0.001764371046
x = 0.522576611565, delta = -0.000049931141
x = 0.522576010396, delta = -0.000000601170
Terminating after 7 iterations because abs(delta) < tol.
root found at 0.522576010396
```

# Next topic: Multivariate Newton's Method

- What if you have a ***system*** of nonlinear equations to solve?
- Generalization of 1D Newton's method.
- Derivation on blackboard.

# Derivation

- Assume 3 eqs, 3 unks, can generalize to any N if desired.

$$f_1(x_1, x_2, x_3) = 0$$

$$f_2(x_1, x_2, x_3) = 0$$

$$f_3(x_1, x_2, x_3) = 0$$

Superscripts count iterations.  
They are not powers.

- Assume we start at point  $\vec{x}^0 = [x_1^0, x_2^0, x_3^0]^T$
- As before, we'll use Taylor's expansion to get value of these three functions after taking a step  $\vec{\delta}^0 = [\delta_1^0, \delta_2^0, \delta_3^0]^T$



- Do Taylor's expansion about this point:

$$f_1(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) = f_1(x_1, x_2, x_3) + \delta_1 \frac{\partial f_1}{\partial x_1} + \delta_2 \frac{\partial f_1}{\partial x_2} + \delta_3 \frac{\partial f_1}{\partial x_3}$$

$$f_2(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) = f_2(x_1, x_2, x_3) + \delta_1 \frac{\partial f_2}{\partial x_1} + \delta_2 \frac{\partial f_2}{\partial x_2} + \delta_3 \frac{\partial f_2}{\partial x_3}$$

$$f_3(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) = f_3(x_1, x_2, x_3) + \delta_1 \frac{\partial f_3}{\partial x_1} + \delta_2 \frac{\partial f_3}{\partial x_2} + \delta_3 \frac{\partial f_3}{\partial x_3}$$

- Rearrange...

$$\begin{aligned} f_1(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_1(x_1, x_2, x_3) + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \\ f_2(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_2(x_1, x_2, x_3) + \begin{bmatrix} \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \\ f_3(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_3(x_1, x_2, x_3) + \begin{bmatrix} \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \end{aligned}$$

- From last slide:

$$\begin{aligned}
 f_1(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_1(x_1, x_2, x_3) + \left[ \begin{array}{ccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{array} \right] \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \\
 f_2(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_2(x_1, x_2, x_3) + \\
 f_3(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_3(x_1, x_2, x_3) +
 \end{aligned}$$

- Define vector quantities...

$$\vec{f} = [f_1, f_2, f_3]^T \quad \vec{x} = [x_1, x_2, x_3]^T \quad \vec{\delta} = [\delta_1, \delta_2, \delta_3]^T$$

- ...and recognize Jacobian matrix


$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix}$$

- Putting it together, this nasty expression

$$\begin{aligned} f_1(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_1(x_1, x_2, x_3) + \left[ \frac{\partial f_1}{\partial x_1} \quad \frac{\partial f_1}{\partial x_2} \quad \frac{\partial f_1}{\partial x_3} \right] \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \\ f_2(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_2(x_1, x_2, x_3) + \left[ \frac{\partial f_2}{\partial x_1} \quad \frac{\partial f_2}{\partial x_2} \quad \frac{\partial f_2}{\partial x_3} \right] \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \\ f_3(x_1 + \delta_1, x_2 + \delta_2, x_3 + \delta_3) &= f_3(x_1, x_2, x_3) + \left[ \frac{\partial f_3}{\partial x_1} \quad \frac{\partial f_3}{\partial x_2} \quad \frac{\partial f_3}{\partial x_3} \right] \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \end{aligned}$$

- becomes a nice vector expression,

$$\vec{f}(\vec{x} + \vec{\delta}) = f(\vec{x}) + J \vec{\delta}$$


 Jacobian matrix

- Then assume  $\vec{f}(\vec{x} + \vec{\delta}) = \vec{0}$  and solve for step

$$\vec{\delta} = -J^{-1} f(\vec{x})$$

- Write in terms of  $\vec{x}^n$  to get iteration rule

The diagram shows the iteration rule for Newton's method in N-dimensions for root finding, enclosed in a light blue rectangular box. The equation is:

$$\vec{x}^{n+1} = \vec{x}^n - \left( J^{-1}(\vec{x}) f(\vec{x}) \right) \Big|_{x=\vec{x}^n}$$

Three arrows point from text labels to parts of the equation:

- An arrow from "New position" points to  $\vec{x}^{n+1}$ .
- An arrow from "Current position" points to  $\vec{x}^n$ .
- An arrow from "Vector step direction and magnitude" points to the term  $\left( J^{-1}(\vec{x}) f(\vec{x}) \right) \Big|_{x=\vec{x}^n}$ .

- This is Newton's method in N-dimensions for **root finding** –  $f(x) = 0$ .
  - Similar method exists for optimization.
  - Optimization involves finding point where gradient (1<sup>st</sup> derivative) is zero –  $f'(x) = 0$ .

# Jacobian

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

- You need Jacobian matrix for Newton-Raphson.
- Don't compute  $\mathbf{J}^{-1}(\mathbf{x})$  in Matlab. Use “\” instead:

$$\vec{x}_{n+1} = \vec{x}_n - \mathbf{J}(\vec{x}) \backslash \vec{f}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

# Newton's Method in ND -- algorithm

1. Select initial guess,  $\vec{x}_0$
2. Compute next approximation,  $\vec{x}_1$

$$\vec{x}_1 = \vec{x}_0 - \mathbf{J}^{-1}(\vec{x}) \cdot \vec{f}(\vec{x}) \Big|_{\vec{x}=\vec{x}_0}$$

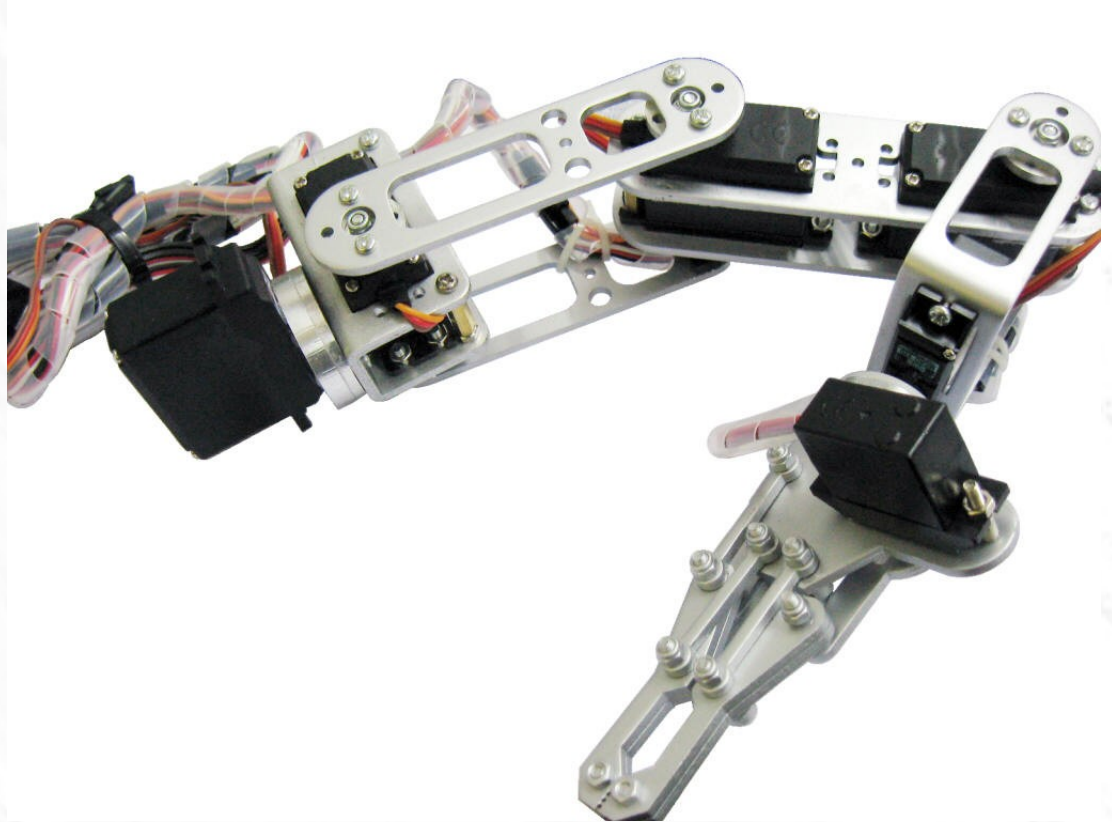
3. If  $\|\vec{x}_{n+1} - \vec{x}_n\| < tol$ , return  $\vec{x}_{n+1}$   
Else loop again.

4. In general,

$$\vec{x}_{n+1} = \vec{x}_n - \mathbf{J}^{-1}(\vec{x}) \cdot \vec{f}(\vec{x}) \Big|_{\vec{x}=\vec{x}_n}$$

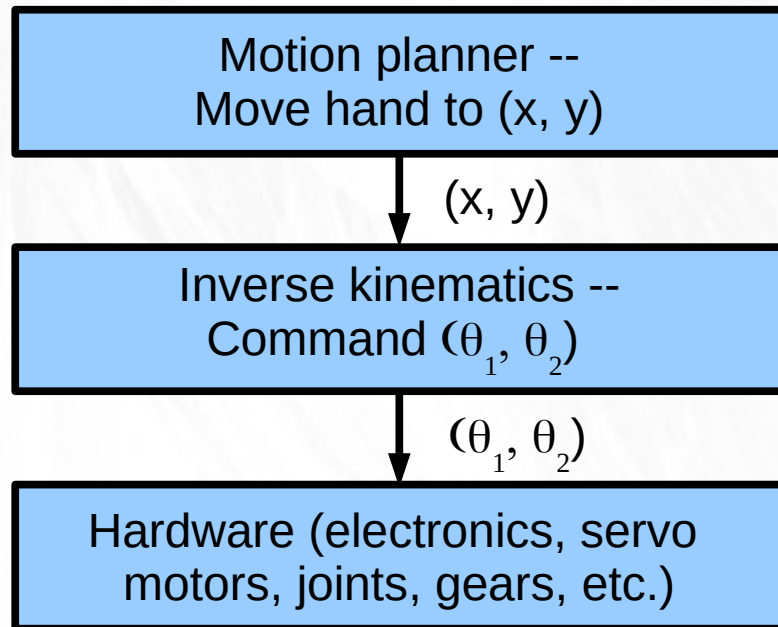
# Application – Robot Kinematics

- Kinematics – study of the geometry of motion.
- Robot arm
- Forward kinematics: Go from angles to position of hand
- Inverse kinematics: Go from desired hand position to the required angles.

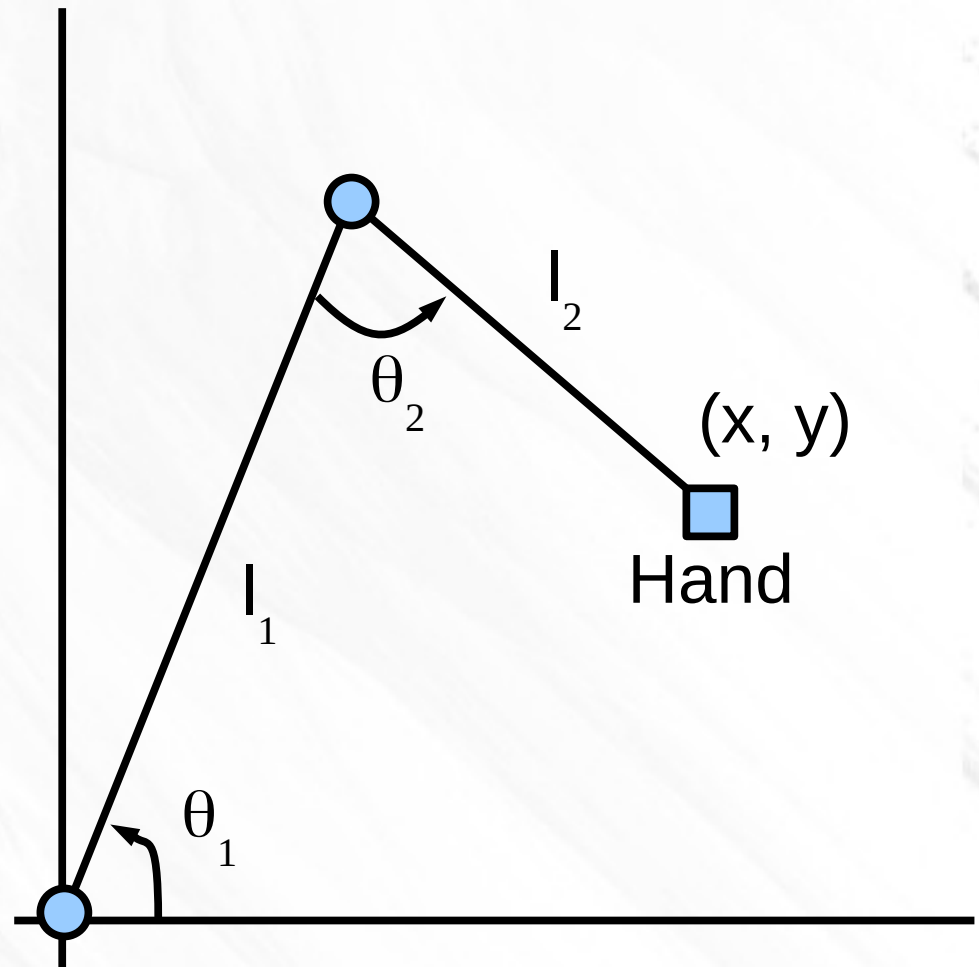


<http://www.youtube.com/watch?v=wE3fmFTtP9g>

# Consider arm with 2 DOF in 2D



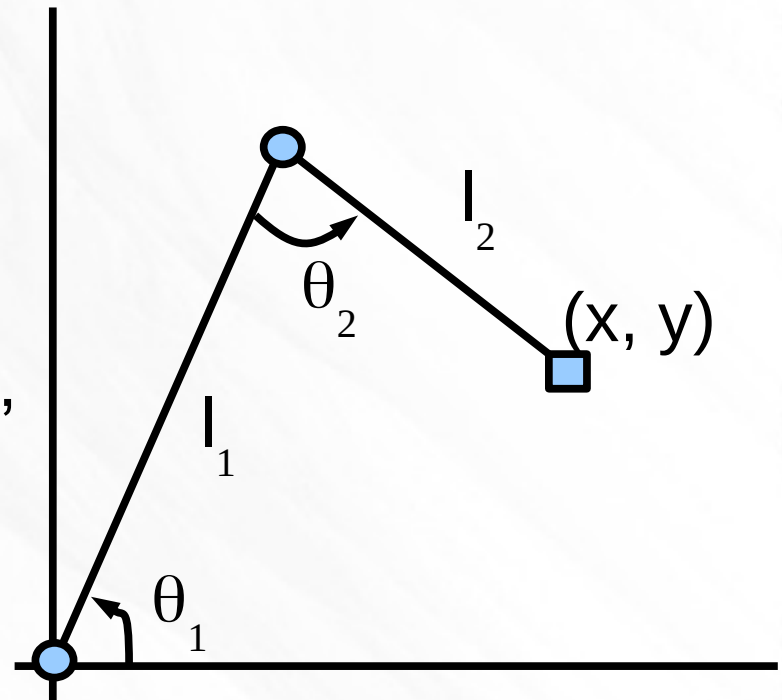
- Two equations, two unknowns
- Nonlinear equations, that is....





# The inverse kinematics problem:

- We know the  $\{x, y\}$  coordinates where we want to position the hand.
- We control the angles  $\{\theta_1, \theta_2\}$  using positioning motors.
- Therefore, we need to get  $\theta_1(x, y)$  and  $\theta_2(x, y)$
- The relationship is highly non-linear -> Newton's Method is required for solution!



Chalkboard derivation of equations

# Derivation

- First point

$$(x_1, y_1) = l_1 (\cos \theta_1, \sin \theta_1)$$

- Second point

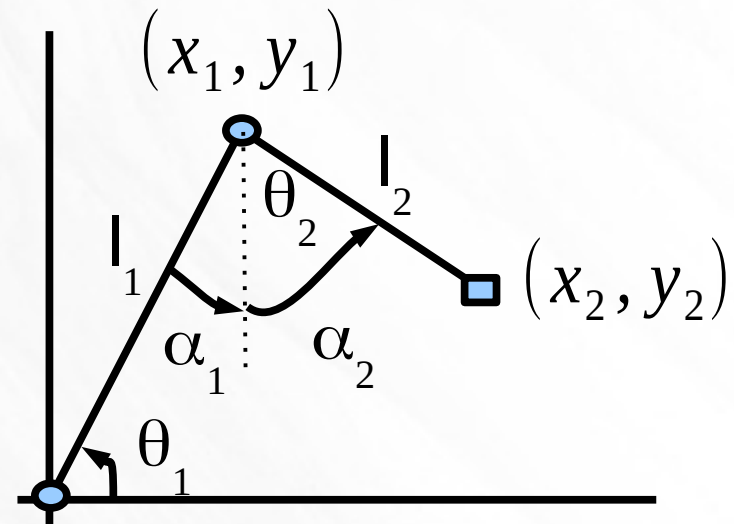
$$x_2 = x_1 + l_2 \sin \alpha_2$$

$$= x_1 + l_2 \sin (\theta_2 - \alpha_1)$$

$$\nearrow \pi/2 - \theta_1$$

$$= l_1 \cos \theta_1 + l_2 \sin (\theta_2 + \theta_1 - \pi/2)$$

$$= l_1 \cos \theta_1 - l_2 \cos (\theta_2 + \theta_1)$$



Computing  $y_2$   
proceeds in similar  
way.

# Forward kinematics

$$x_2 = l_1 \cos \theta_1 - l_2 \cos (\theta_2 + \theta_1)$$

$$y_2 = l_1 \sin \theta_1 - l_2 \sin (\theta_2 + \theta_1)$$

- Question: I know the  $(x, y)$  I want, but what are the required  $\theta_1, \theta_2$  ?

- Write as

$$f_1(\theta_1, \theta_2) = l_1 \cos \theta_1 - l_2 \cos (\theta_2 + \theta_1) - x_2 = 0$$

$$f_2(\theta_1, \theta_2) = l_1 \sin \theta_1 - l_2 \sin (\theta_2 + \theta_1) - y_2 = 0$$

- Now we have a 2D root finding problem.

# 2D root finding

- Newton's method in 2D for robot arm

$$\vec{\theta}^{n+1} = \vec{\theta}^n - \left( J^{-1}(\vec{\theta}) f(\vec{\theta}) \right) \Big|_{\theta = \vec{\theta}^n}$$

- With

$$f_1(\theta_1, \theta_2) = l_1 \cos \theta_1 - l_2 \cos(\theta_2 + \theta_1) - x_2$$

$$f_2(\theta_1, \theta_2) = l_1 \sin \theta_1 - l_2 \sin(\theta_2 + \theta_1) - y_2$$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial \theta_2} \end{bmatrix}$$

# Mathematically formulated robot example

Given  $(x, y)$ , find  $(\theta_1, \theta_2)$  so that

$$\vec{f}(\theta_1, \theta_2) = \begin{pmatrix} -l_2 \cos(\theta_2 + \theta_1) + l_1 \cos(\theta_1) - x \\ -l_2 \sin(\theta_2 + \theta_1) + l_1 \sin(\theta_1) - y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

With Jacobian matrix

$$J = \begin{pmatrix} l_2 \sin(\theta_2 + \theta_1) - l_1 \sin(\theta_1) & l_2 \sin(\theta_2 + \theta_1) \\ -l_2 \cos(\theta_2 + \theta_1) + l_1 \cos(\theta_1) & -l_2 \cos(\theta_2 + \theta_1) \end{pmatrix}$$

Iteration to implement

$$\begin{pmatrix} \theta_1^{n+1} \\ \theta_2^{n+1} \end{pmatrix} = \begin{pmatrix} \theta_1^n \\ \theta_2^n \end{pmatrix} - (J(\theta_1^n, \theta_2^n))^{-1} \cdot \vec{f}(\theta_1^n, \theta_2^n)$$

# Look at the code

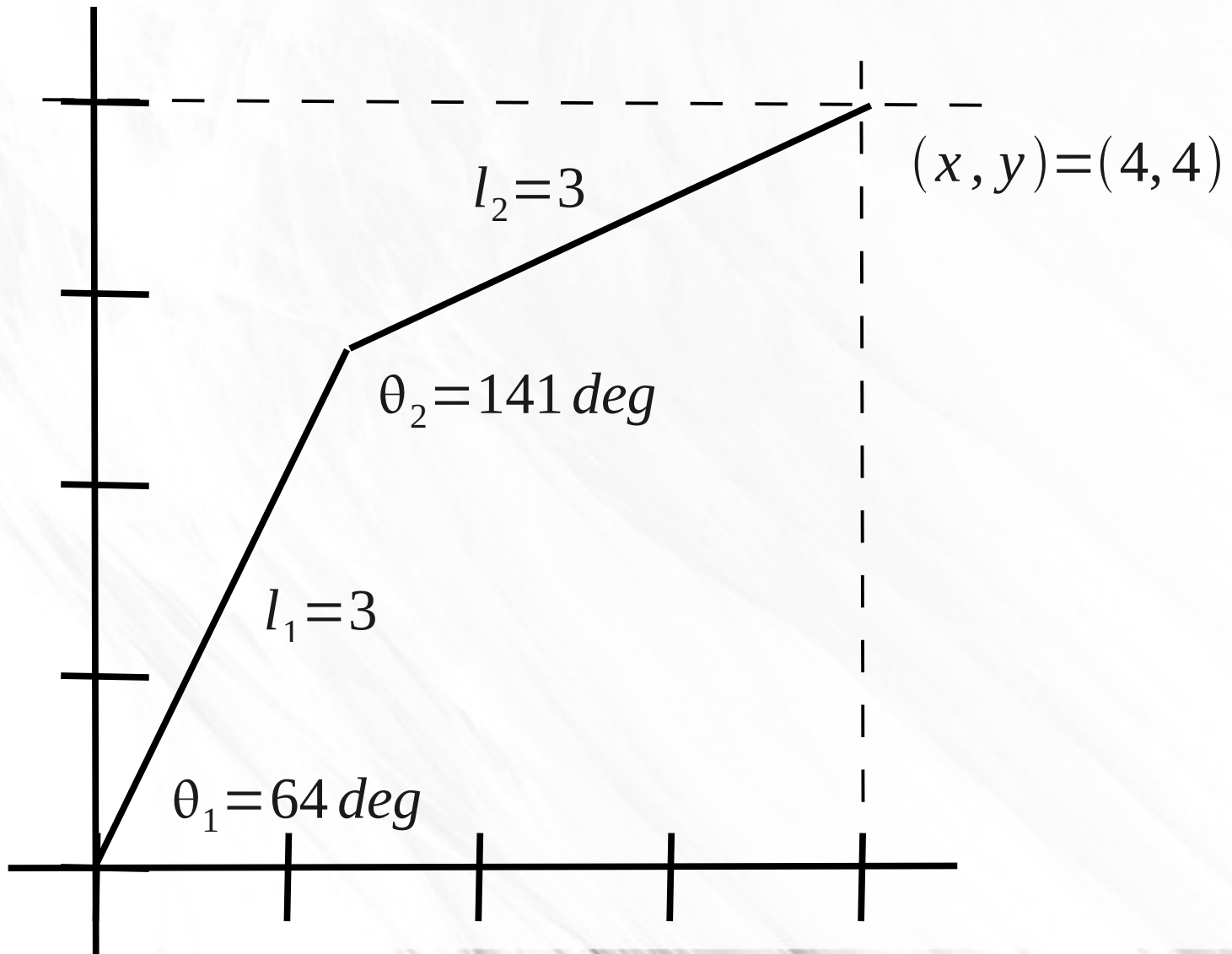
- I put  $f(x)$  and Jacobian into separate file so I can replace them without editing the newton2D for different equation systems.
- I had to use globals to communicate the coefficients  $l1$ ,  $l2$ ,  $x$ ,  $y$  to all sub-functions.
  - If you know OO, it's probably cleaner to pass the reference to an equation object to all sub-functions.

# Running the robot code

- Use `test_newton2D` as test harness.
- Set  $l_1 = l_2 = 3$ . Point to move to is  $[x, y] = [4, 4]$ .
- Good start point: near  $[\pi/2, \pi/2]$ . (Draw picture)
- Try starting near  $[0, 0]$  instead.
- Try starting near  $[\pi/2, 0]$ .
- Newton's method is very sensitive to initial starting point!

# Check the results

$\theta_1 = 64.471221 \text{ deg}$ ,  $\theta_2 = 141.057559 \text{ deg}$





# Testing Newton's method

- Testing is easy – just check that  $f(x) = 0$  within some tolerance.
- But what start point to use?
  - We'll talk about this later.....
  - If you know roughly where the solution is, please use that location as the start point.

# Main Ideas in this Session

- Root finding – nonlinear equations
- Bisection method – 1D
- Secant method – 1D
- Newton's method – 1D
  - Example from quantum mechanics
- Newton's method – ND
  - Example from robotics