PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Derivatives + Integrals

- One of the most obvious things in computational physics is to look at computation of derivatives and integrals
- You probably can guess how much of this is already known to you, since this is how you learned to do these things anyway!

- The “hard” part for you in calculus was probably getting your brain around taking the limits of the “simpler” things when the step size went to zero

- Well, that part is also hard for computers!
  - So, you have to think a little differently here, and go back to discrete derivatives and integrals

Derivatives + Integrals

- Conceptually this is probably the easiest chapter
- The devil is in the details, however

The devil in the details

- Short discussion in Chapter 1 of Garcia
- Also parts are addressed in Chapter 2 of Garcia

Derivatives

- We've now seen several differentials in the previous discussion

- We need to be able to compute the differential numerically, so as we mentioned, we take a step back :

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

- First we take the "forward difference" :

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \frac{h}{2} f''(x) + \mathcal{O}(h^2) .$$

- But we could equally have taken the "backward difference":

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{(-h)}{2} f''(x) + \mathcal{O}(h^2) .$$

Derivatives

- But! Here's the first devil :)

Improve
the accuracy!

- Combine the forward and backward differences to get a symmetric difference!

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \; .$$

Recall : “Big-Ohh” Notation

- The “big-ohh” notation stands for “order”
- $O(N^2)$ operations means “the leading coefficient in the number of operations scales like N^2”
- Remember, “operations” here really means “multiplications”... addition is cheap!
- In computing, we want to minimize this as much as possible since the computational time scales the same way

Derivatives

- Since "h" is small, the error that we make is smaller (h^2) :

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \; .$$

- We want to make the error that we make as small as possible!

- Simple thing : reduce h
  - But! This has a bit of a problem because it increases the computational time (ouch)

- Can we do better?

Derivatives

- Absolutely!

- Can try with a "five point stencil":
  - http://en.wikipedia.org/wiki/Five-point_stencil

- Consider the five points : $\{x-2h, x-h, x, x+h, x+2h\}.$

- Then the derivative looks like :

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \mathcal{O}(h^4)\ .$$

Derivatives

- Five-point derivative method is simple enough to just write it down

```
def diff_fivepoint( f, x, h) :
    ''' f      : name of function to be differentiated
        x      : the point at which df/dx is required
        h      : step size
    '''
    dfdx = ( f(x-2*h) - 8*f(x-h) + 8*f(x+h) - f(x+2*h)) / (12*h)
    return dfdx
```

Derivatives

- What if we don't know the functional form of the derivative?

- We have to use some approximate functional form of the data points to handle this

- One popular method is to use polynomial interpolation and extrapolation

$$f(x) \simeq \sum_{i=0}^{n} a_i x^n \ ,$$

$$f'(x) \simeq \sum_{i=1}^{n} a_i n x^{n-1}$$

Ridder's method

- Chapter 5, Section 7 of Numerical Recipes recommends Ridder's algorithm :
  - Advances in Engineering Software, 4 75-76 (1978)

- Uses Ridder's polynomial extrapolation.
- This relies on the so-called "Neville's algorithm" to compute the polynomial extrapolation, then computes the derivative

Neville's Algorithm

- Derived to compute polynomial interpolation
  - http://en.wikipedia.org/wiki/Neville's_algorithm

- Given n data points, you can construct the n-dimensional polynomial (which is unique) as follows :

  - Let $p_{i,j}$ denote the polynomial of degree j - i which goes through the points (x_k,y_k) for k=i..j.
  - The $p_{i,j}$ satisfy :

$$p_{i,i}(x) = y_i, \qquad 0 \le i \le n,$$

$$p_{i,j}(x) = \frac{(x_j - x)p_{i,j-1}(x) + (x - x_i)p_{i+1,j}(x)}{x_j - x_i}, \qquad 0 \le i < j \le n.$$

Neville's Algorithm

- So, we can fill a tableau to compute this from the left to the right :

$$
\begin{array}{ccccc}
p_{0,0}(x) = y_0 & & & & \\
 & p_{0,1}(x) & & & \\
p_{1,1}(x) = y_1 & & p_{0,2}(x) & & \\
 & p_{1,2}(x) & & p_{0,3}(x) & \\
p_{2,2}(x) = y_2 & & p_{1,3}(x) & & \boxed{p_{0,4}(x)} \\
 & p_{2,3}(x) & & p_{1,4}(x) & \\
p_{3,3}(x) = y_3 & & p_{2,4}(x) & & \\
 & p_{3,4}(x) & & & \\
p_{4,4}(x) = y_4 & & & &
\end{array}
$$

Ridder's method

- Start with the symmetric difference
- Compute polynomial extrapolations for n=10 polynomials
  - Reduce the step size for each n
  - Compute symmetric difference at smaller step size. Store the result.
  - Compute extrapolations for n-1 with Neville's algorithm
  - Compare each new extrapolation to one order lower at this step size, and the previous one
    - If error is smaller, keep the improvement
    - else, continue

If you have lots of derivatives, over the entire time this can save you a lot of CPU's

Ridder's Method

```
Input value
Initialize 10x10 array
Compute symmetric difference differential
for each polynomial extrapolation :
   reduce step size
   compute symmetric difference differential
   store results
   compute error to previous step size
   if error is better, keep it
   else, continue
```

Ridder's method

```
if h == 0.0 :
    print "diff_Ridders: h must be non-zero"
    exit

n = 10             # dimension of extrapolation table
a = array(  [[0.0] * n] *n )                # extrapolation table

a[0][0] = (f(x + h) - f(x - h)) / (2 * h)
answer = 0.0
error = nan_to_num( inf ) / 2.0  # get a large value for the error
for i in xrange(n) :
    h /= 1.4
    a[0][i] = (f(x + h) - f(x - h)) / (2 * h)
    fac = 1.4 * 1.4
    for j in range(1,i+1) :
        a[j][i]=(a[j-1][i] * fac - a[j-1][i-1]) / (fac - 1)
        fac *= 1.4 * 1.4
        err = max(abs(a[j][i] - a[j-1][i]),
                  abs(a[j][i] - a[j-1][i-1]))
        if err <= error :
            error = err
            answer = a[j][i]

    if abs(a[i][i] - a[i-1][i-1]) >= 2 * error :
        break
return answer,error
```

- docker pull srappoccio/compphys:latest

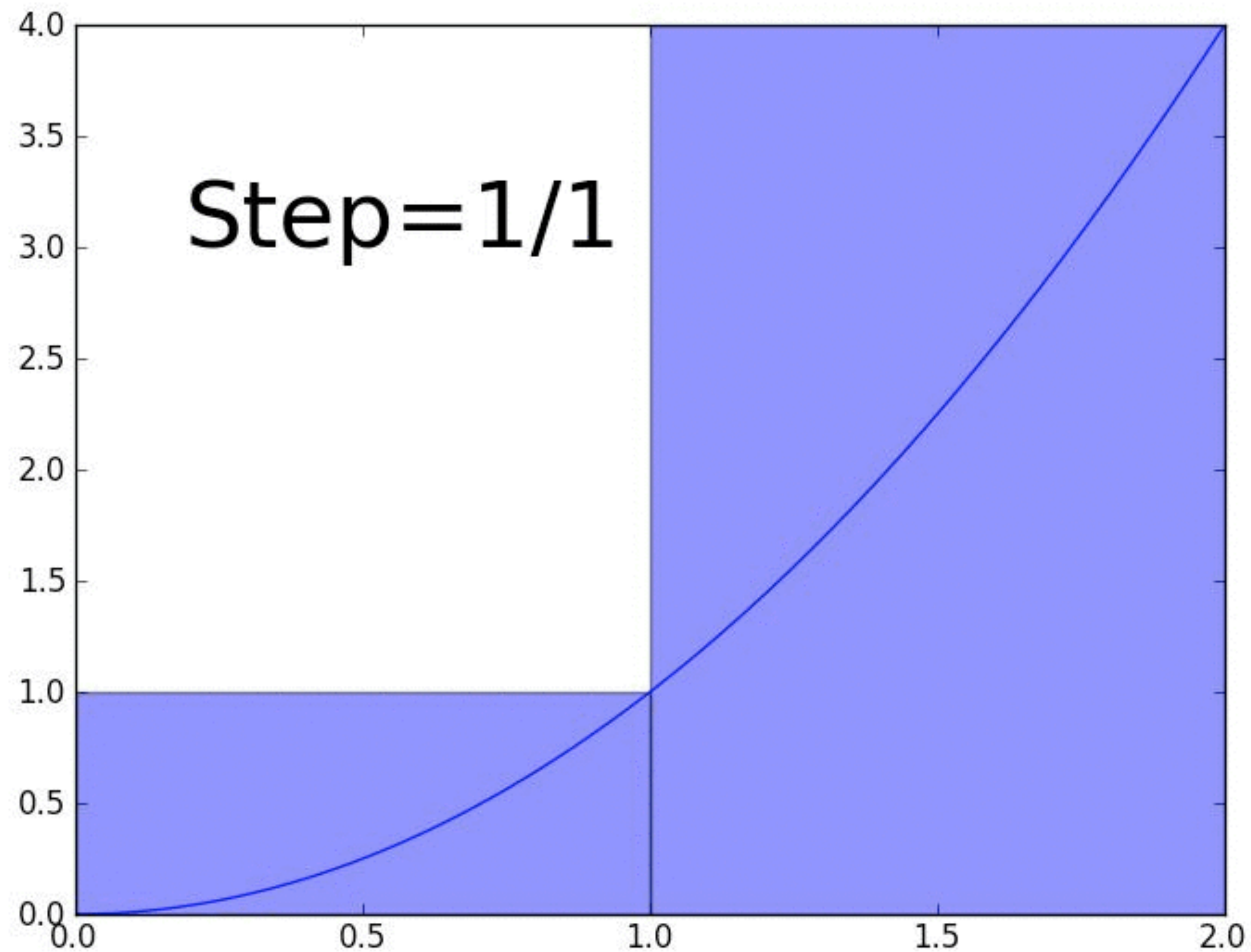
- cd results/CompPhys/
- git pull origin master

Integration

- Covered (cursorily) in Garcia Chapter 10.2
- Also covered in Numerical Recipes Chapter 4
  - (The C version is online for free :)
  - http://apps.nrbook.com/c/index.html

Integration

- We've done derivatives. Now on to integration.
- Recall your high-school (ish) calculus class :
  - http://en.wikipedia.org/wiki/Integral



Integration

- To first order, that's all we're going to do for computations of integrals
  - There are fancier, faster, better methods, but they are successive approximations of this kind of thing except one (Monte Carlo integration)
  - We'll first consider the class of problems called "quadrature" in numerical analysis
  - See Chapter 4 of "Numerical Recipes" (C version online for free from their website)

Integration

- Consider the integral :

$$I = \int_a^b dx\, f(x)\ .$$

- Then define :

$$y(x) \equiv \int_a^x dx'\, f(x')\ .$$

- Thus :

$$\frac{dy}{dx} = f(x)\ , \qquad y(a) = 0\ , \qquad y(b) = I\ .$$

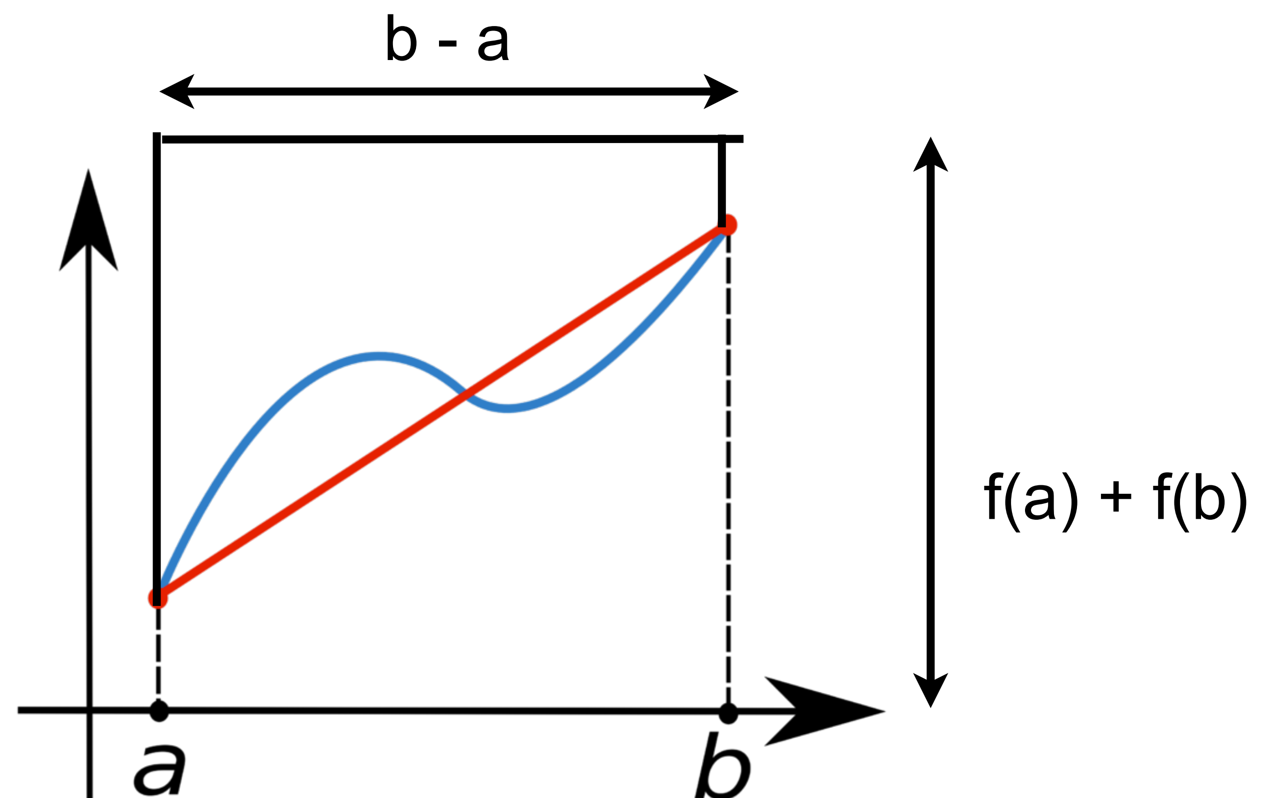
- We COULD solve this as a differential equation!
- But instead we'll start with the "bare bones" approximations that you learned in high school/freshman calculus

Integration : Trapezoidal Rule

- The rectangle sum is a slightly-less-than-wonderful approximation of the integral
- The trapezoid sum is actually much better
  - http://en.wikipedia.org/wiki/Trapezoidal_rule

- You compute the integral by approximating it as a trapezoid :

$$\int_a^b f(x)\,dx \approx (b-a)\left[\frac{f(a)+f(b)}{2}\right].$$



Integration : Trapezoidal Rule

- How accurate? Define h = (b-a) as our "small" parameter
- Approximate the integral by :

$$I = \int_a^b dx\, f(x) \simeq (b-a) f(x_0) = h f(x_0) \ ,$$

- We Taylor-expand f(x) :

$$f(x) - f(x_0) = (x - x_0) f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0) + \ldots \ ,$$

- The error in the estimate is :

$$\begin{aligned} I - (b-a) f(x_0) &= f'(x_0) \int_a^b dx\, (x - x_0) + \frac{1}{2} f''(x_0) \int_a^b dx\, (x - x_0)^2 + \ldots \\ &= f'(x_0) \left[\frac{b+a}{2} - x_0 \right] \frac{b-a}{2} \\ &\quad + \frac{1}{6} f''(x_0) \left[(b - x_0)^3 - (a - x_0)^3 \right] \\ &= \mathcal{O}(h^2) \ . \end{aligned}$$

Integration : Trapezoidal Rule

- Notice a trick we can play!
- If we choose $x_0 = (b+a) / 2$, then we get down to $O(h^3)$ instead of $O(h^2)$!
  - "Midpoint rule" :

$$I \simeq T = \frac{h}{2}\left[f(a) + f(b)\right] .$$

$$I = \int_a^b dx \left[f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \ldots\right]$$

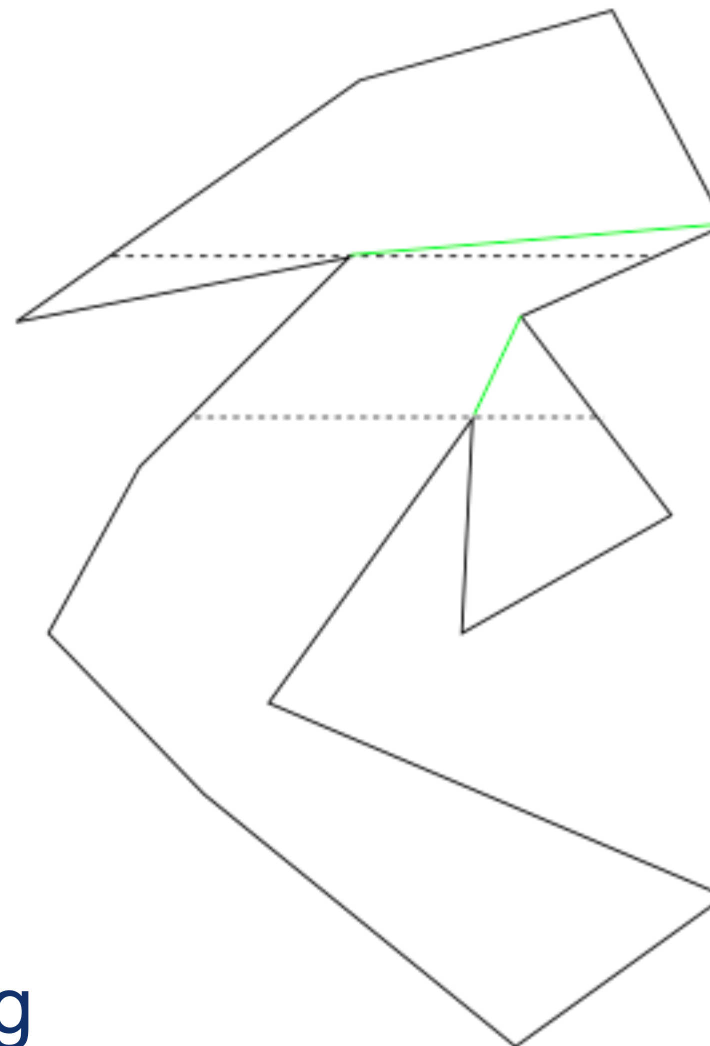
$$= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + \ldots$$

$$T = \frac{h}{2}\left[f(a) + f(a) + hf'(a) + \frac{h^2}{2}f''(a)\right] \ldots$$

$$I - T = -\frac{h^3}{12}f''(a) + \ldots = \mathcal{O}(h^3) .$$

Integration : Trapezoidal Rule

- Then can break the integral into a bunch of trapezoids
- This is also related to "polygon tessellation" in computer graphics, to compute (or display) the area in a 2-d image :
  - http://en.wikipedia.org/wiki/Polygon_triangulation
- Easier and faster than computing the area in a more complicated way!
- Almost all of your modern computer games will allow you to set the amount of tessellation to optimize performance or beauty depending on your taste



Integration : Trapezoidal Rule

Integration : Trapezoidal Rule

- Now you can guess what to do, we have successive approximations :

$$\int_a^b dx\, f(x) = \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} dx\, f(x)\ ,$$

- The uncertainty here is :

$$I \simeq I_T \equiv \sum_{i=1}^{N-1} \frac{x_{i+1} - x_i}{2} \left[f(x_{i+1}) + f(x_i) \right]\ ,$$

$$I - I_T \sim \mathcal{O}\left((N-1)h^3\right) \sim \mathcal{O}\left((b-a)h^2\right)\ ,$$

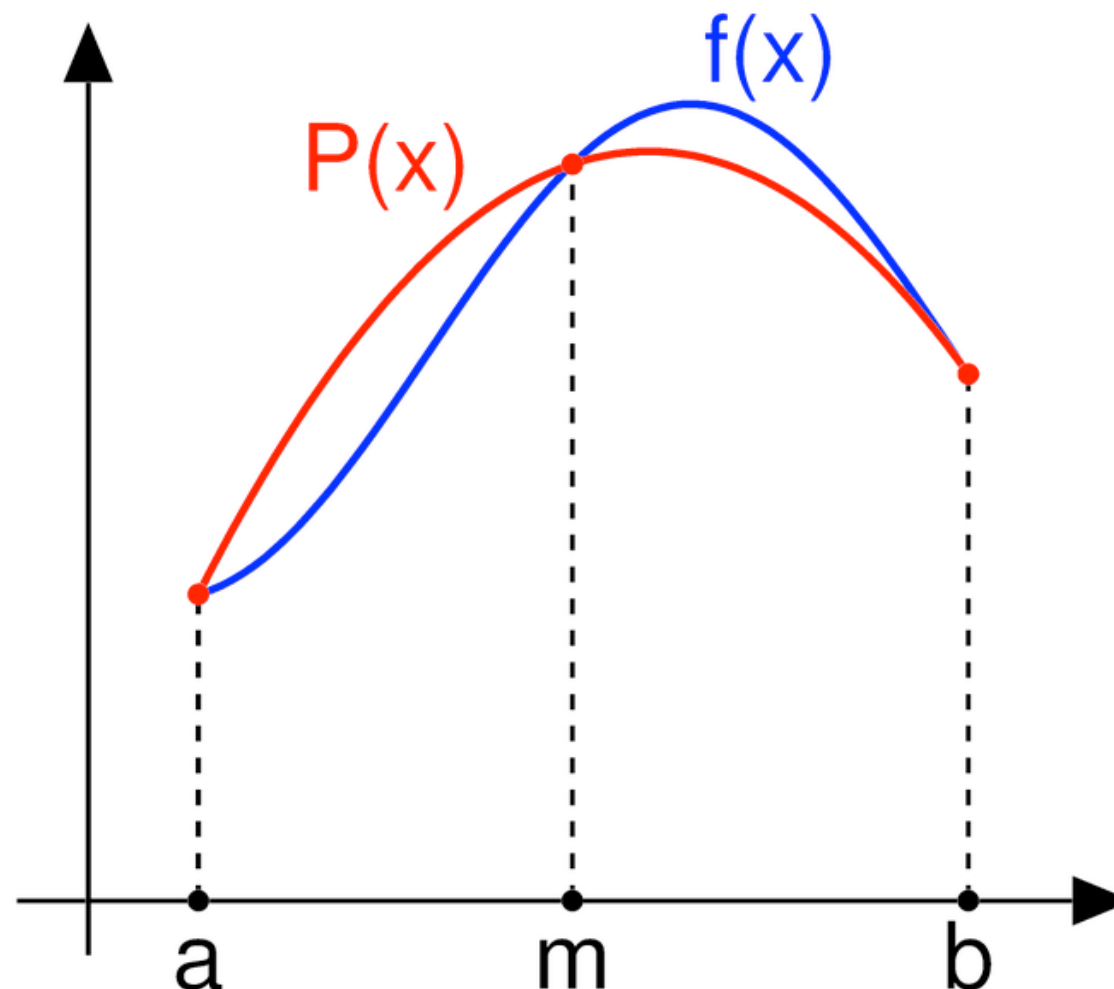
- Can therefore pick N,h to desired accuracy (same deal as in tessellation!)

Integration : Trapezoidal Rule

```
def trapezoidal_rule(f, a, b, n):
    """Approximates the definite integral of f from a to b by
    the composite trapezoidal rule, using n subintervals"""
    h = (b - a) / n
    s = f(a) + f(b)
    for i in xrange(1, n):
        s += 2 * f(a + i * h)
    return s * h / 2
```

Integration : Simpson's Rule

- This is more accurate than the Trapezoidal rule, and not really slower :
  - http://en.wikipedia.org/wiki/Simpson's_rule
- Instead of approximating by a trapezoid, use a parabola!
- This is a "three-point" rule, similar to that we saw last class for the derivatives with the "symmetric" derivative



Integration : Simpson's Rule

- The approximation is thus :

$$I = \int_a^{b=a+2h} dx\, f(x) = \frac{h}{3}\left[f(a) + 4f(a+h) + f(b)\right] + \mathcal{O}(h^5)\ .$$

- Similarly to above, we can divide into intervals of size 2h if we have a large area :

$$\int_a^b dx\, f(x) = \frac{h}{3}\Big[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a+4h) + \cdots + 2f(b-2h) + 4f(b-h) + f(b)\Big] + \mathcal{O}\left((b-a)h^4\right)\ .$$

- This particular implementation requires an EVEN number of intervals, and that the function is evaluated at an ODD number of points (need three points on each!)

Integration : Simpson's Rule

```
def simpson(f, a, b, n):
    """Approximates the definite integral of f from a to b by
    the composite Simpson's rule, using n subintervals"""
    h = (b - a) / n
    s = f(a) + f(b)

    for i in range(1, n, 2):
        s += 4 * f(a + i * h)
    for i in range(2, n-1, 2):
        s += 2 * f(a + i * h)

    return s * h / 3
```

Integration : For your homeworks!

- In your homeworks (assigned Monday) you will go through the same exercise of examining numerical precision of integration, like we did for derivatives.

Integration : Adaptive methods

- Can often adapt the algorithm to a desired precision by iterating
- This improves the accuracy dynamically, saving time when the function is fairly linear

Integration : Adaptive methods

- So, pseudocode is :

```
Choose N and compute h
Set h --> h/2
```

Compute $\Delta I \equiv |I_T(h) - I_T(2h)|$

If $\Delta I > \epsilon$ repeat

- Can also reuse the computations as we did in our previous example to speed up computational time :

Integration : Adaptive methods

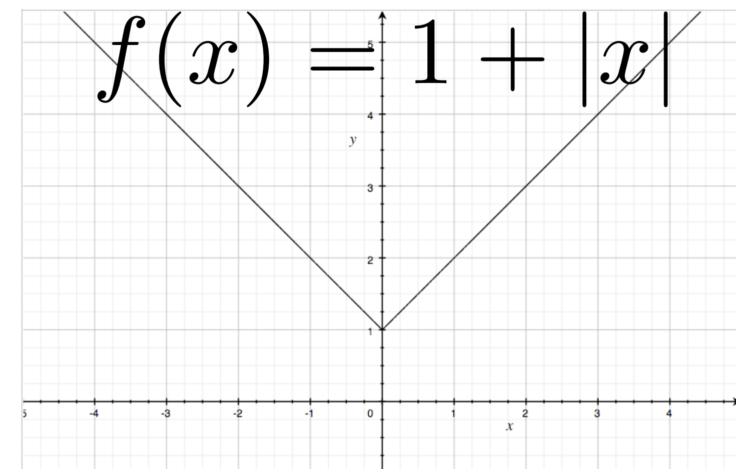
```python
def adaptive_trapezoid(f, a, b, acc, output=False):
    """
    Uses the adaptive trapezoidal method to compute the definite integral
    of f from a to b to desired accuracy acc.
    """

    old_s = -1e-30
    h = b - a
    n = 1
    s = (f(a) + f(b)) / 2
    if output == True :
        print "N = " + str(n+1) + ",  Integral = " + str( h*s )
    while abs(h * (old_s - s/2)) > acc :
        old_s = s
        for i in xrange(n) :
            s += f(a + (i + 0.5) * h)
        n *= 2
        h /= 2
        if output == True :
            print "N = " + str(n+1) + ",  Integral = " + str( h*s )
    return h * s
```

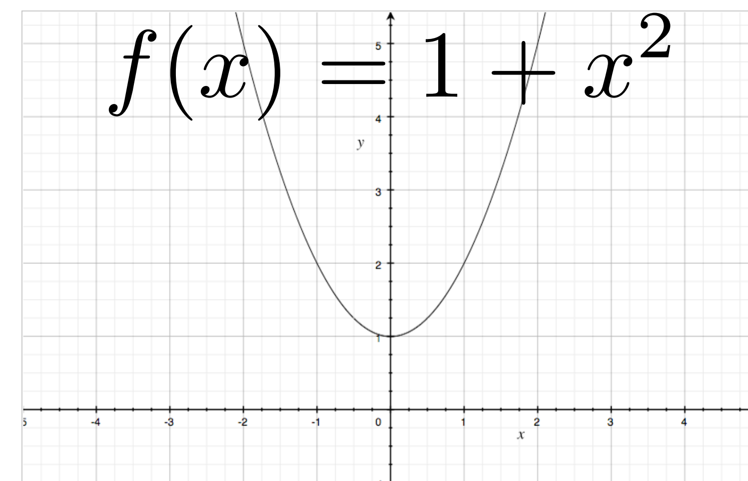
Next up : Root finding

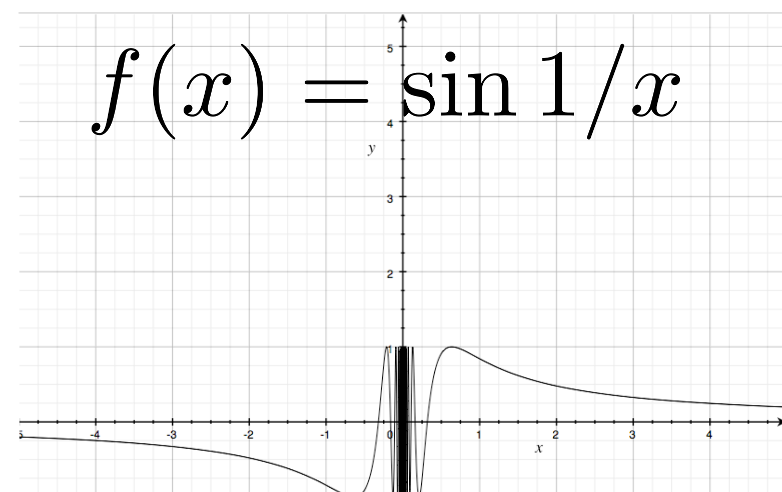
- The next issue is to find the roots of a function f(x)
- That is,

$$\{x | f(x) = 0\}$$

- Lots of issues, not only computational!
  - May not have a root
  - May have imaginary roots
  - May have a large number of roots
- Section 4.3 in Garcia, Chapter 9 in Numerical Recipes







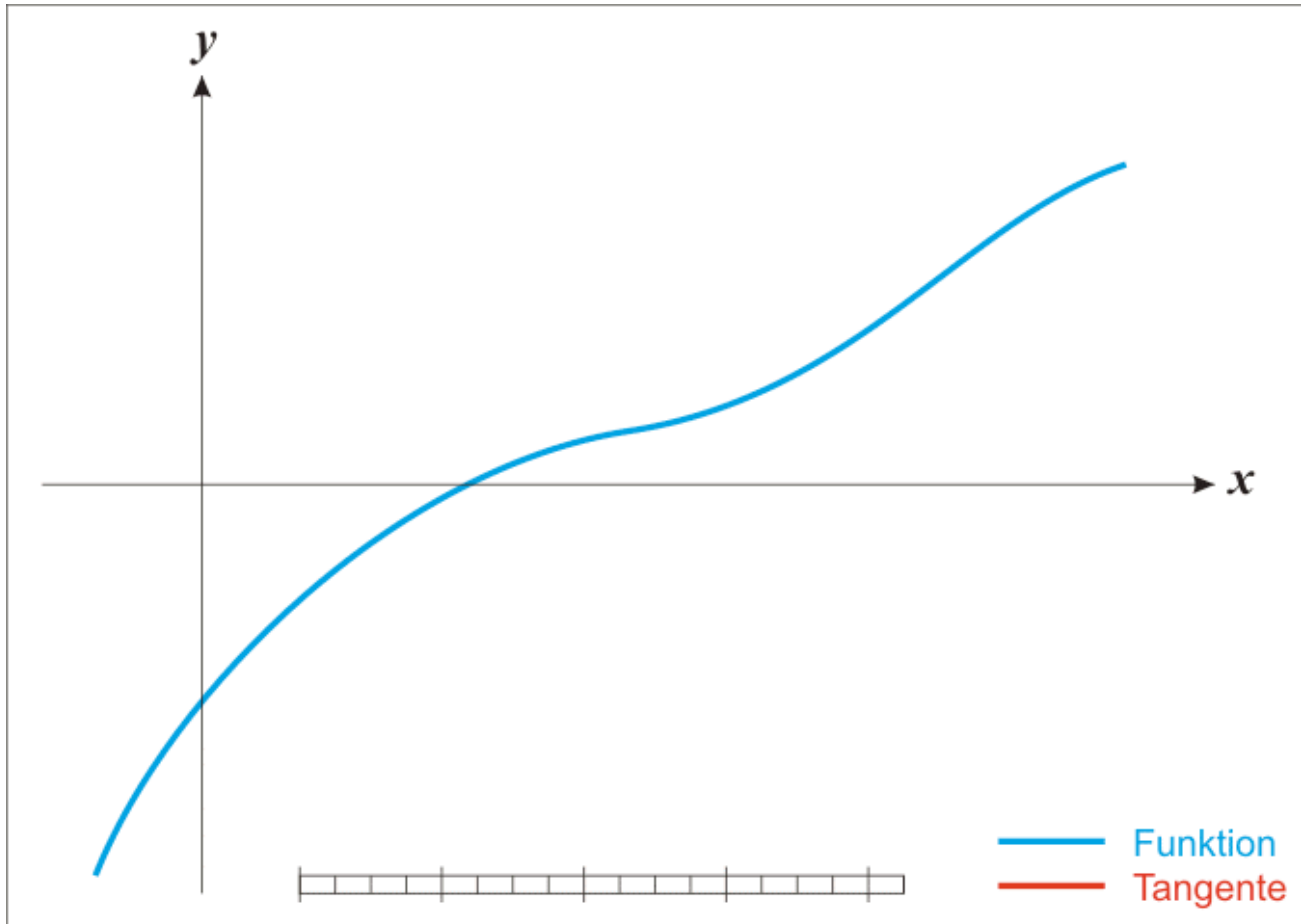
Root finding

- But, given those caveats, once again it is very straightforward logic here
- You’ve probably already seen Newton’s method in your mathematics classes
  - http://en.wikipedia.org/wiki/Newton's_method

- Guess at the answer
- Find derivative
- Use it to get successively better approximations

Root finding

Newton's Method



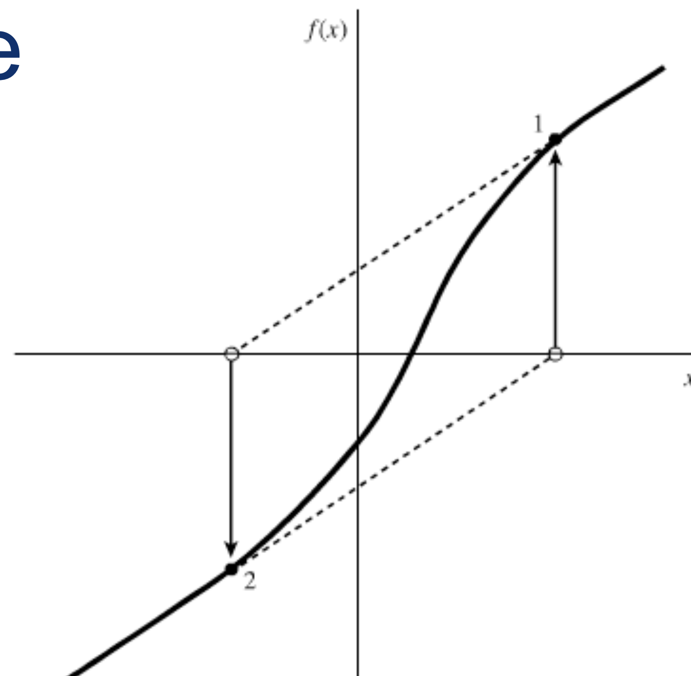
Root finding

- A very simple version (not yet Newton's version) :
- Choose accuracy you want : ϵ
- Guess x and dx, then $f_0 = f(x)$
- Step is : $x \to x + dx$
- Check to see if you've passed the root : $f_0 \times f(x)$
  - If negative, you changed sign so, reverse : $x \to x - dx$
    and reduce your step : $dx \to dx/2$
- If $|dx| < \epsilon$ or $f(x) = 0$, you're done
- Otherwise, iterate steps

Root finding

- The above assumes that the function f(x) is continuously differentiable with at least one real root
- Much of the complications arise when this is not the case :
  - Kinks
  - Discontinuities
  - No real roots

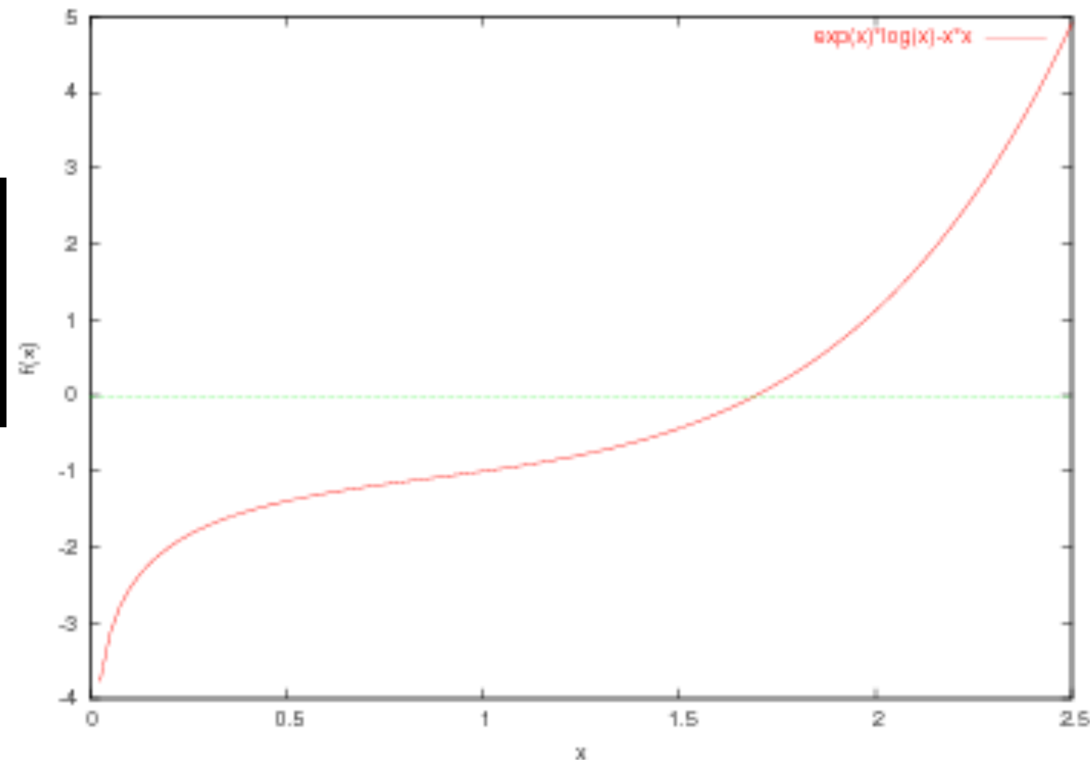
- So, we usually put in protections against this, and eventually the code will give up and print a failure message
- Even still, can have pathologies!



Root finding

- So, the code for our simple root finding is here :

```
def f (x) :
    return exp(x) * log(x) - x * x;
```



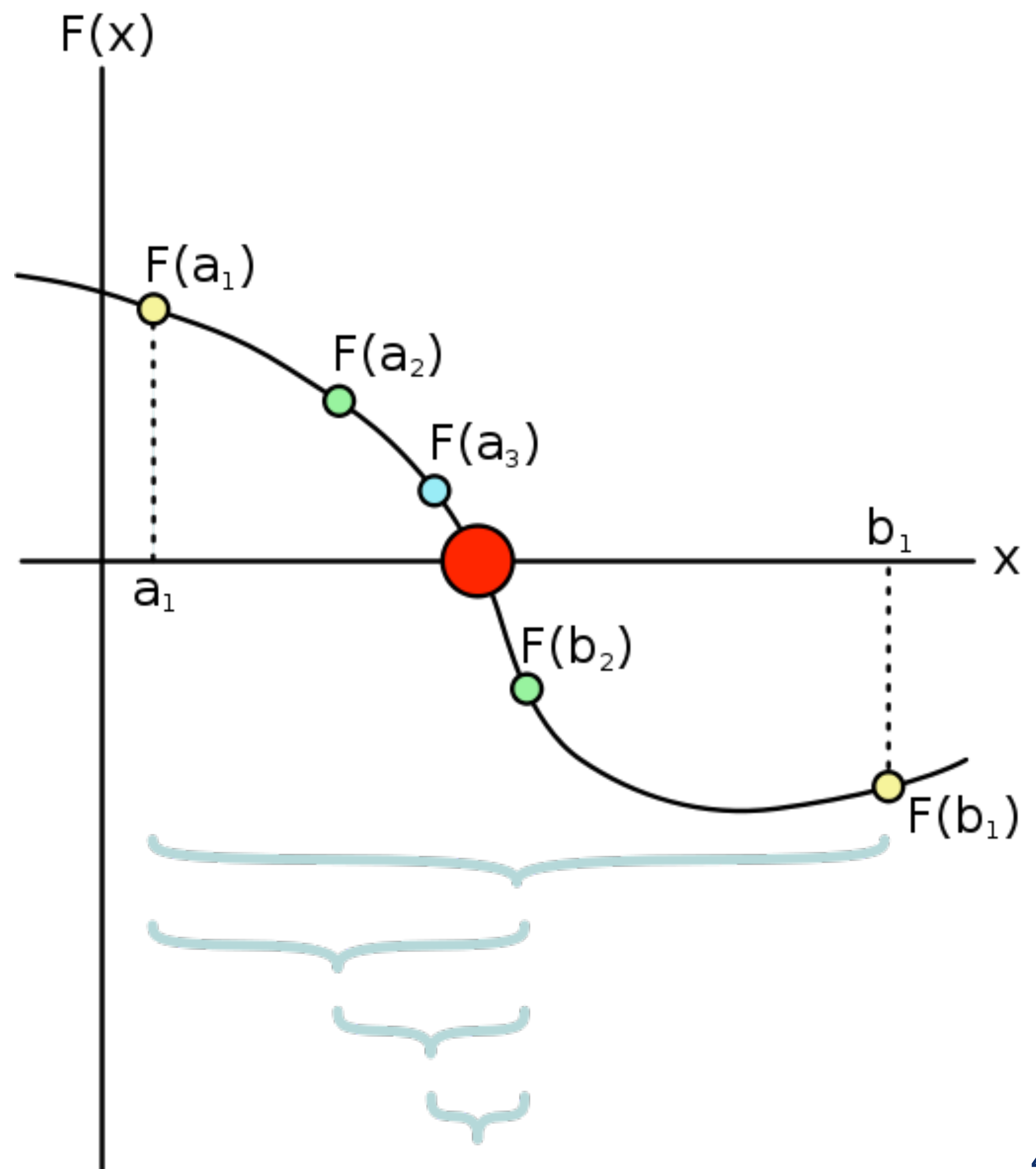

```
step = 0
print  ' {0:4.0f}    {1:20.15f} {2:20.15f}'.format(step, x, dx)
f_old = f(x)

while abs(dx) > abs(acc) :
    x += dx
    if f_old * f(x) < 0 :
        x -= dx
        dx /= 2

    ++step
    print  ' {0:4.0f}    {1:20.15f} {2:20.15f}'.format(step, x, dx)
```

Root finding

- Problem! We already need to know the structure pretty specifically of the function before we find the root
- So, the code will happily continue until infinity if we give it a guess in the wrong direction
- This is a bit of a pain, so we need something better

Root finding

- The next idea is to find a window within which the root will fall : bisection method
  - http://en.wikipedia.org/wiki/Bisection_method
- Utilizes the intermediate value theorem!
- Assumes that the function has exactly one root between x0,x1, at which point it changes sign



Root finding

- Repeatedly bisects the interval :

- Let $x_{\frac{1}{2}} = (x_0 + x_1)/2$ be the bisection point

- Compute : $f(x_0) \times f(x_{\frac{1}{2}})$
  - If positive, then x_0 and $x_{1/2}$ are on the same side of the root, and $x_{1/2}$ is closer, so replace $x_0 \rightarrow x_{\frac{1}{2}}$
  - Else, they're on opposite sides, so refine interval: $x_1 \rightarrow x_{\frac{1}{2}}$

- If $|x_1 - x_0| < \epsilon$ or if $f(x_{\frac{1}{2}}) = 0$ then we have the root with sufficient precision

Root finding

- So, the code looks like this :

Compute bisection

If they're both on the same sign, then refine to $[x_{1/2}, x_1]$

Otherwise refine to $[x_0, x_{1/2}]$

Iterate until the accuracy is achieved

```
def root_bisection(f, x1, x2, accuracy=1.0e-6, max_steps=1000, root_debug=False):
    """Return root of f(x) in bracketed by x1, x2 with specified accuracy.
    Assumes that f(x) changes sign once in the bracketed interval.
    Uses bisection root-finding algorithm.
    """
    f1 = f(x1)
    f2 = f(x2)
    if f1 * f2 > 0.0:
        raise Exception("f(x1) * f(x2) > 0.0")
    x_mid = (x1 + x2) / 2.0
    f_mid = f(x_mid)
    dx = x2 - x1
    step = 0
    if root_debug:
        root_print_header("Bisection Search", accuracy)
        root_print_step(step, x_mid, dx, f_mid)
    while abs(dx) > accuracy:
        if f_mid == 0.0:
            dx = 0.0
        else:
            if f1 * f_mid > 0:
                x1 = x_mid
                f1 = f_mid
            else:
                x2 = x_mid
                f2 = f_mid
            x_mid = (x1 + x2) / 2.0
            f_mid = f(x_mid)
            dx = x2 - x1
        step += 1
        if step > max_steps:
            warning = "Too many steps (" + repr(step) + ") in root_bisection"
            raise Exception(warning)
        if root_debug:
            root_print_step(step, x_mid, dx, f_mid)
    return x_mid
```

Root finding

- OK, much better, we just have to find a bounding interval
- Usually a lot easier than having to remember what the function actually looks like

- One problem : It's pretty darned slow.

Root finding

- Let's estimate the convergence rate :
  - Number of iterations needed before root is located with some desired accuracy
  - Either $dx < \epsilon$ or $f(x) < \alpha$
  - We usually do the former, not the latter
- Look at bisection.
  - After n bisection steps, then dx_n is given by :

$$|dx_n| = |x_1 - x_0| \qquad \text{after } n \text{ iterations}$$
$$= \frac{1}{2}|dx_{n-1}| = \frac{1}{2^2}|dx_{n-2}| = \cdots = \frac{1}{2^n}|dx_0| \ ,$$

  - So, $\frac{1}{2^n}|dx_0| \leq \epsilon$,
  - or : $n \geq \log_2\left[\frac{|dx_0|}{\epsilon}\right] = \frac{\log_{10}\left[\frac{|dx_0|}{\epsilon}\right]}{0.3010\ldots}$.

Root finding

- Can also represent as

$$|dx_n| \simeq C_F \, |dx_{n-1}|^{\alpha} \ ,$$

where C_F is a constant "convergence factor"

α is the "order of convergence"

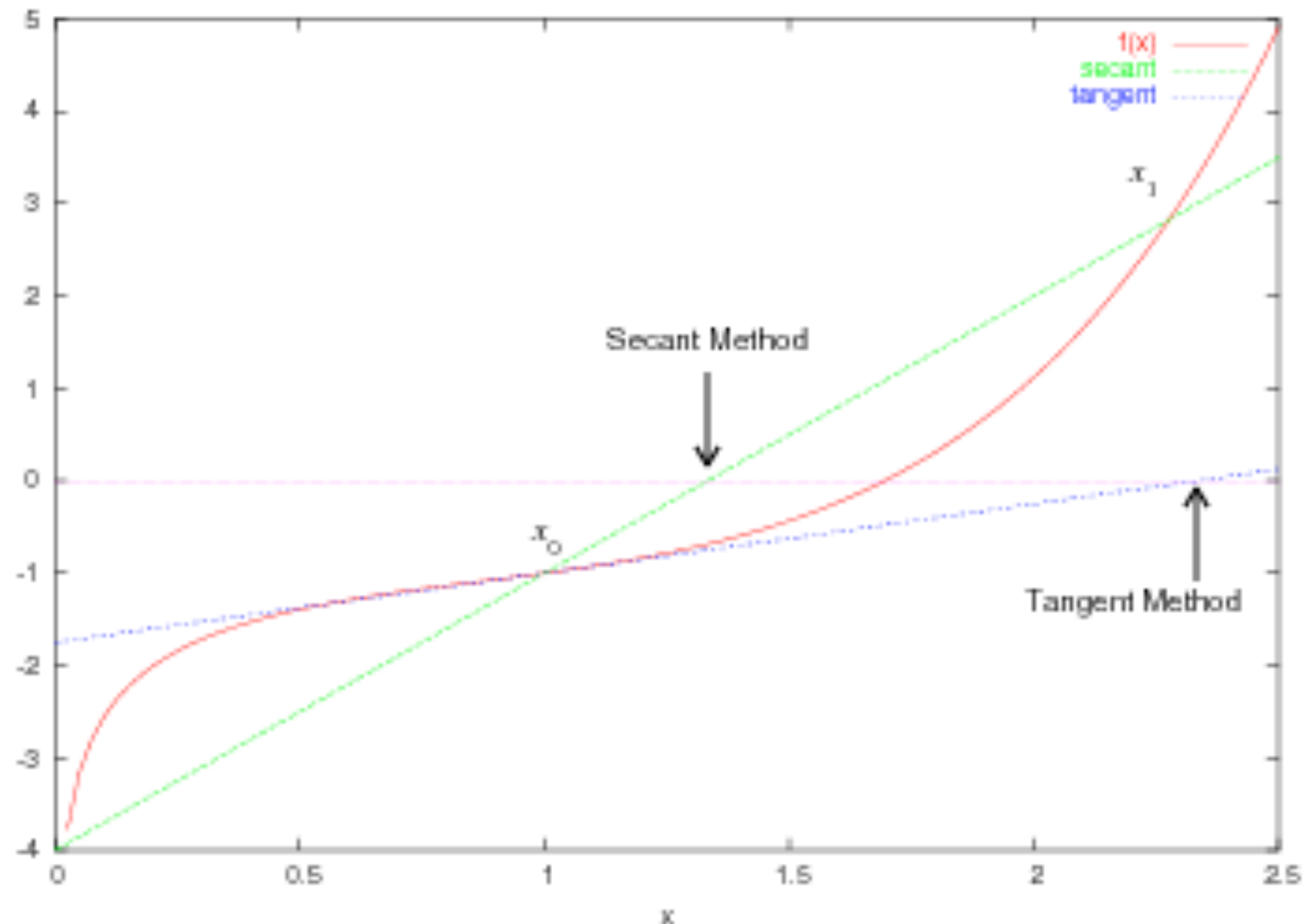
- For bisection : $$C_F = \frac{1}{2}, \alpha = 1$$

- For the simple step-halving : $$C_F \in [\frac{1}{2}, 1], \alpha = 1$$

- Both of these are pretty darned slow to converge
- Can we do better?

Root finding

- Two better options :
  - Secant method
    - http://en.wikipedia.org/wiki/Secant_method
  - Newton's method (or Newton-Raphson, or "tangent" method"
    - http://en.wikipedia.org/wiki/Newton's_method
- There are others, but we'll just use these



Root finding

- Secant method (secare : Latin, "to cut"... think "section")
- Choose the secant, the line between x0 and x1 that intersects f(x)
- Equation is : $$s(x) = f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) \ .$$
- Can utilize x0 as the initial guess, and then specify the initial window (dx = x1 - x0)
  - The next step is therefore chosen at where the secant intersects f(x) = 0:

$$s(x_{\text{new}}) = 0 \quad \Rightarrow \quad x_{\text{new}} = x_1 - (x_1 - x_0)\frac{f(x_1)}{f(x_1) - f(x_0)} \equiv x_1 + dx_{\text{new}} \ .$$

- Then iterate

Root finding

- So, pseudocode is :
  - choose x0 and x1 “near” the root, dx = x1 - x0
  - If either f(x0) = f(x1) then the method fails, so re-guess
  - Replace :

$$dx \to dx_{\text{new}} \, , x_0 \to x_1 \, , x_1 \to x_{\text{new}}$$

  - Check if: $|dx_{\text{new}}| < \epsilon$
    - If so, desired accuracy reached.
    - Otherwise, iterate

Root finding

- Here's the code for the secant method :

```
def root_secant(f, x0, x1, accuracy=1.0e-6, max_steps=20,root_debug=False):
    """Return root of f(x) given guesses x0 and x1 with specified accuracy.
    Uses secant root-finding algorithm.
    """
    f0 = f(x0)
    dx = x1 - x0
    step = 0
    if root_debug:
        root_print_header("Secant Search", accuracy)
        root_print_step(step, x0, dx, f0)
    if f0 == 0:
        return x0
    while abs(dx) > abs(accuracy):
        f1 = f(x1)
        if f1 == 0:
            return x1
        if f1 == f0:
            raise Exception("Secant horizontal f(x0) = f(x1) algorithm fails")
        dx *= - f1 / (f1 - f0)
        x0 = x1
        f0 = f1
        x1 += dx
        step += 1
        _root_number_of_steps = step
        if step > max_steps:
            root_max_steps("root_secant", max_steps)
        if root_debug:
            root_print_step(step, x1, dx, f1)
    return x1
```

Guess x0, x1

Check for anomaly

Make replacements

Iterate until accuracy reached

Root finding

- If a few conditions are met, then this is much faster than bisection :
  - If f(x) is smooth near the root
  - If x0 and x1 are close enough to the root
  - Given these two, a Taylor expansion should be a good approximation
- Assume that the root is at zero (for simplicity, but without loss of generality, you can always do a change of variables to make this at some other x)
- Then, in the expansion :

$$f(x) \simeq xf' + \frac{x^2}{2}f'' = xf'\left[1 + x\frac{f''}{2f'}\right] ,$$

  - we have written simply f'(0) = f', f''(0) = f''

Root finding

- Then we can plug this into the secant approximation to get :

$$x_{\text{new}} \simeq x_1 - \frac{(x_1 - x_0)x_1 f' \left[1 + x_1 \frac{f''}{2f'}\right]}{(x_1 - x_0)f' + \frac{x_1^2 - x_0^2}{2} f''}$$

$$= x_1 \left[1 - \frac{1 + x_1 \frac{f''}{2f'}}{1 + (x_1 + x_0)\frac{f''}{2f'}}\right]$$

$$\simeq x_1 x_0 \left(\frac{f''}{2f'}\right) ,$$

- To find the convergence, we rewrite x_new and x1 in terms of our convergence relation from above, and define C_F and alpha :

$$|x_{\text{new}}| = C_F |x_1|^\alpha , \qquad |x_1| = C_F |x_0|^\alpha .$$

Root finding

- So, we do a little algebraic massaging and get :

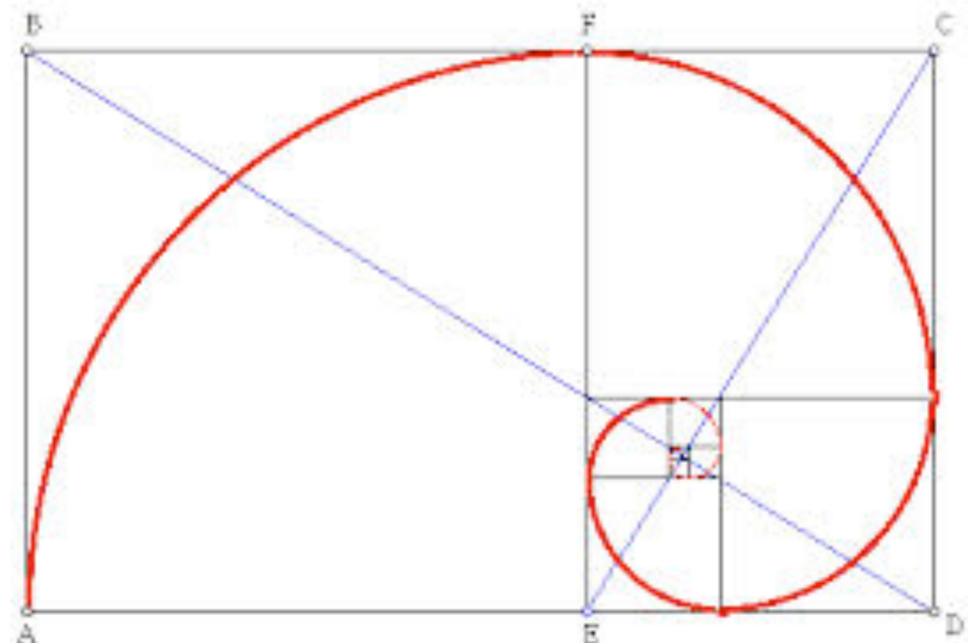
$$|x_1|^{\alpha-1-\frac{1}{\alpha}} = \frac{\left|\frac{f''}{2f'}\right|}{C_F^{1+\frac{1}{\alpha}}} \ ,$$

- The RHS is independent of x1, so we must have

$$\alpha - 1 - \frac{1}{\alpha} = 0 \qquad \Rightarrow \qquad \alpha = \frac{1+\sqrt{5}}{2} = 1.618033988\ldots = 1 + \frac{1}{\alpha} \ .$$

- I.e. the rate of convergence is equal to the golden mean!

- Faster than linear, but not quite quadratic
- But! Strong assumptions about behavior of f(x)



Root finding

- Finally, Newton-Raphson method (or “tangent” method) is the fastest we will consider that has the smallest number of assumptions
- But this time, instead of the secant, we utilize the derivative (“tangent”!)
- Tangent is : $t(x) = f(x_0) + f'(x_0)(x - x_0)$,
- Then we see where the tangent intersects the x axis:

$$x_{\mathrm{new}} = x_0 - \frac{f(x_0)}{f'(x_0)} \equiv x_0 + dx \; .$$

Root finding

- Similar to secant algorithm :

- Chose x0 near the root
- Check if f'(x0) = 0.
  - If = 0, fails
  - Else continue
- Compute dx, replace x0 by x_new
- Check if $|dx| < \epsilon$ or $f(x_{\text{new}}) = 0$
  - If so, accuracy reached
  - Else : iterate

Root finding

- Two cases here :
  - f’ is analytic : rate of convergence is ~quadratic
  - f’ must be computed numerically : rate of convergence is ~secant method

Root finding

- Tangent method is :

If f' is analytic, use this

Compute f, f', dx

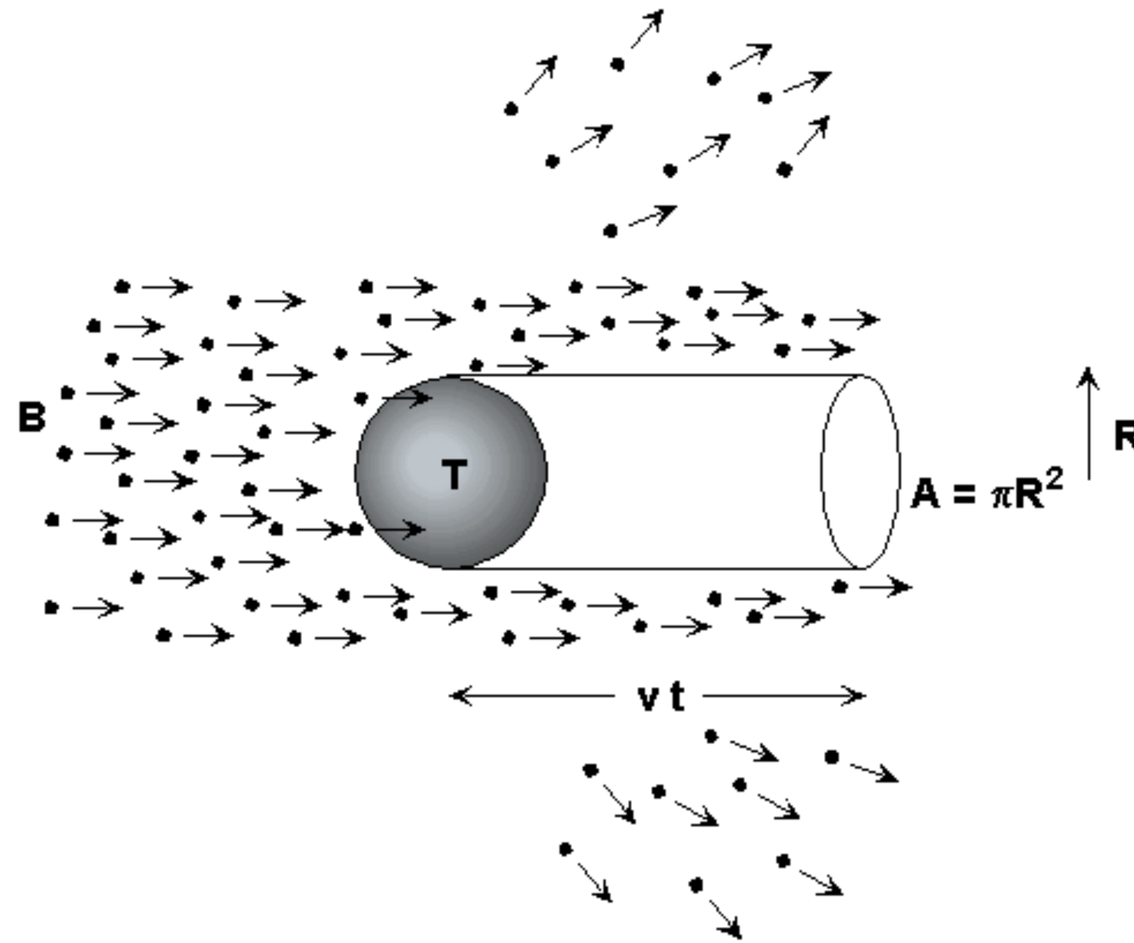
Make replacements

Iterate until convergence

```
def root_tangent(f, fp, x0, accuracy=1.0e-6, max_steps=20, root_debug=False):
    """Return root of f(x) with derivative fp = df(x)/dx
    given initial guess x0, with specified accuracy.
    Uses Newton-Raphson (tangent) root-finding algorithm.
    """
    f0 = f(x0)
    fp0 = fp(x0)
    if fp0 == 0.0:
        raise Exception(" root_tangent df/dx = 0 algorithm fails")
    dx = - f0 / fp0
    step = 0
    if root_debug:
        root_print_header("Tangent Search", accuracy)
        root_print_step(step, x0, dx, f0)
    if f0 == 0.0:
        return x0
    while True:
        fp0 = fp(x0)
        if fp0 == 0.0:
            raise Exception(" root_tangent df/dx = 0 algorithm fails")
        dx = - f0 / fp0
        x0 += dx
        f0 = f(x0)
        if abs(dx) <= accuracy or f0 == 0.0:
            return x0
        step += 1
        if step > max_steps:
            root_max_steps("root_tangent", max_steps)
        if root_debug:
            root_print_step(step, x0, dx, f0)
    return x0
```

Application : Cross sections

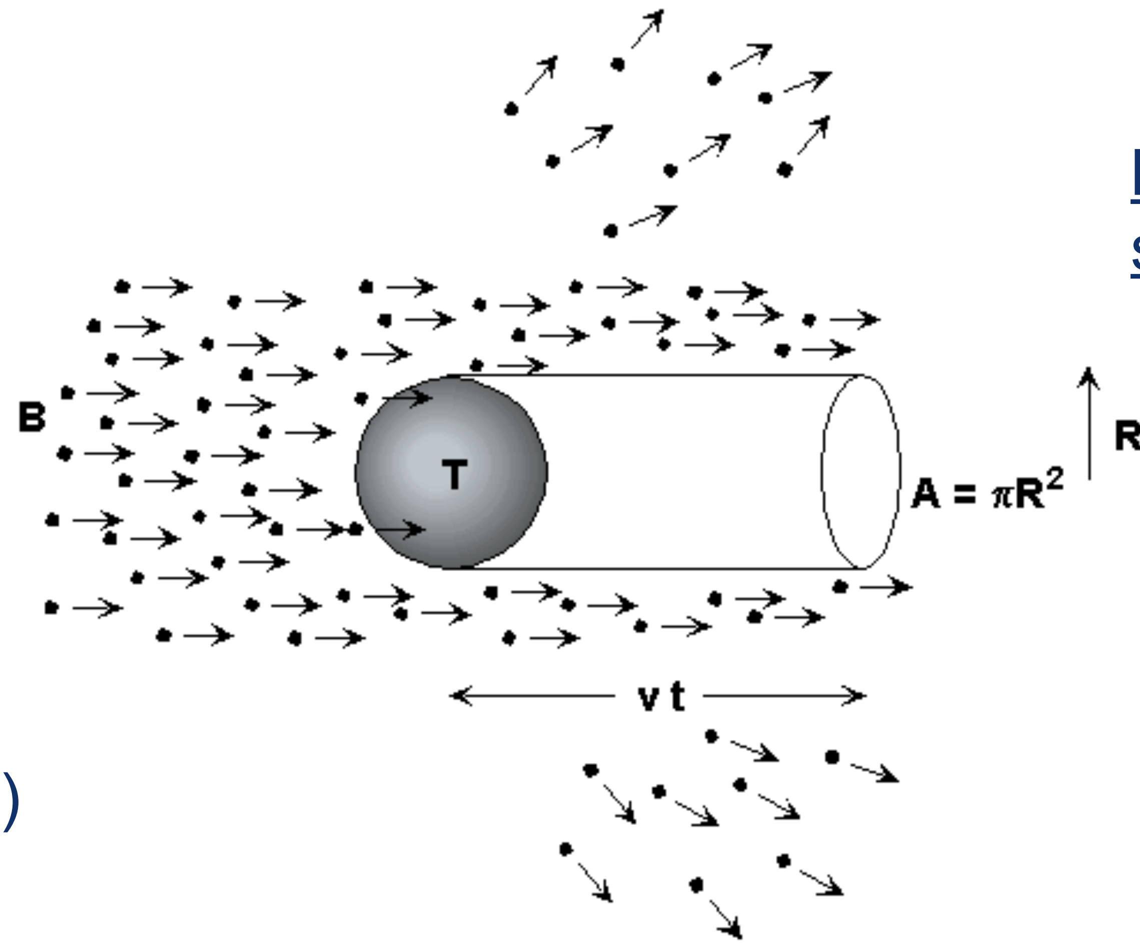
- You should have encountered cross sections in one of your classes :



$$\frac{\text{Number of interactions}}{\text{unit time}} = \frac{\text{Cross section}}{\text{Incident flux}}$$

Application : Cross sections

- You should have encountered cross sections in one of your classes :



http://www.jupiterscientific.org/sciinfo/crosssection.html

Number (dimensionless)

$$\frac{\text{Number of interactions}}{\text{unit time}} = \frac{\text{Cross section}}{\text{Incident flux}}$$

Area

Time

Number/time/area

Cross sections

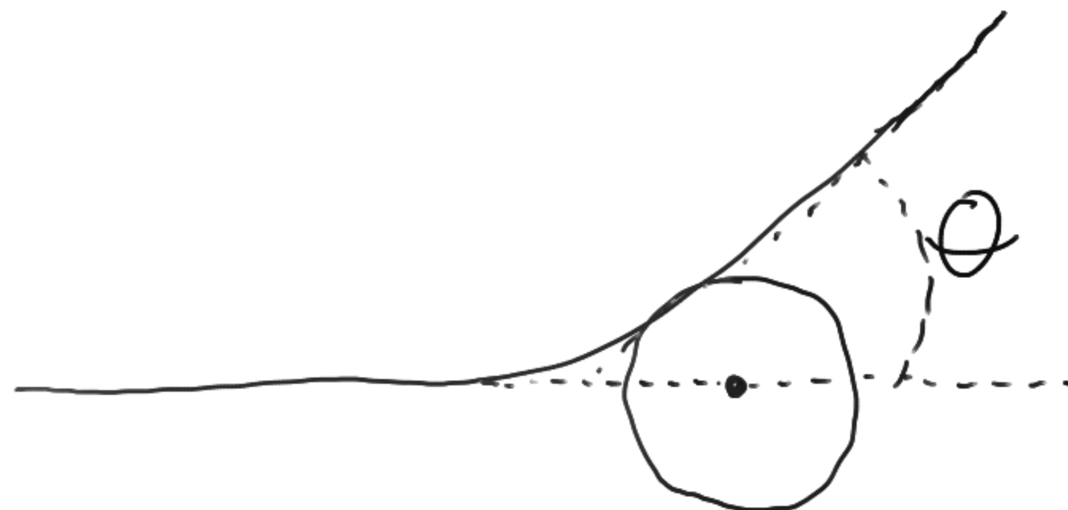
- Happens a lot in physics
  - Collision of galaxies
  - Particle physics (ubiquitous!)
  - Optical scattering
  - Etc

Cross sections

- Take a simple case :
  - Particle of mass “m” scattering from an isotropic central force field
    - Examples : billiard balls, Rutherford scattering

$$\mathbf{F} = f(r)\hat{\mathbf{r}} = -\frac{dV(r)}{dr}\hat{\mathbf{r}} \ .$$

- Use conservation of linear and angular momenta to solve the problem
  - This occurs in the plane of the scatter (2-d)



Cross sections

- Conservation of energy :

$$E = T + V = \frac{1}{2}m\left(\dot{r}^2 + r^2\dot{\theta}^2\right) + V(r)$$

- Conservation of angular momentum (normal to plane) is:

$$L_z = mr\dot{\theta} \equiv m\ell$$

- But we know : $\dot{r} = \frac{dr}{dt} = \frac{dr}{d\theta}\frac{d\theta}{dt}$

- So we can get rid of ALL of the time derivatives in the energy expression!

$$E = \frac{1}{2}m\ell^2\left[\left(\frac{du}{d\theta}\right)^2 + u^2\right] + V(1/u)\ , \qquad u \equiv \frac{1}{r}$$

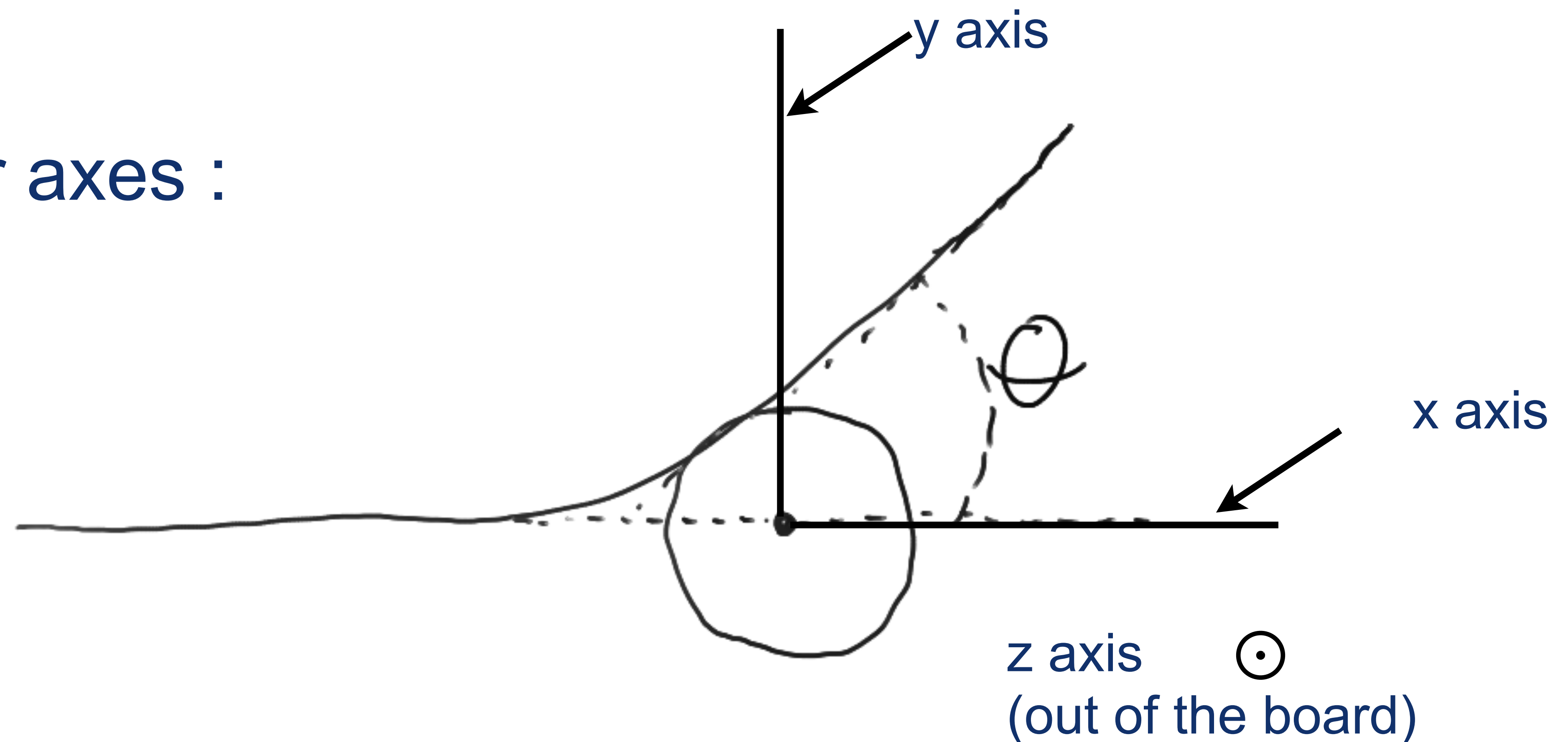
- Can then integrate this to get the trajectory in parametric form

Cross sections

- So we're looking for an equation of the form :

$$r = r(\theta)$$

- Define our axes :



Cross sections

- Define the "impact parameter" b

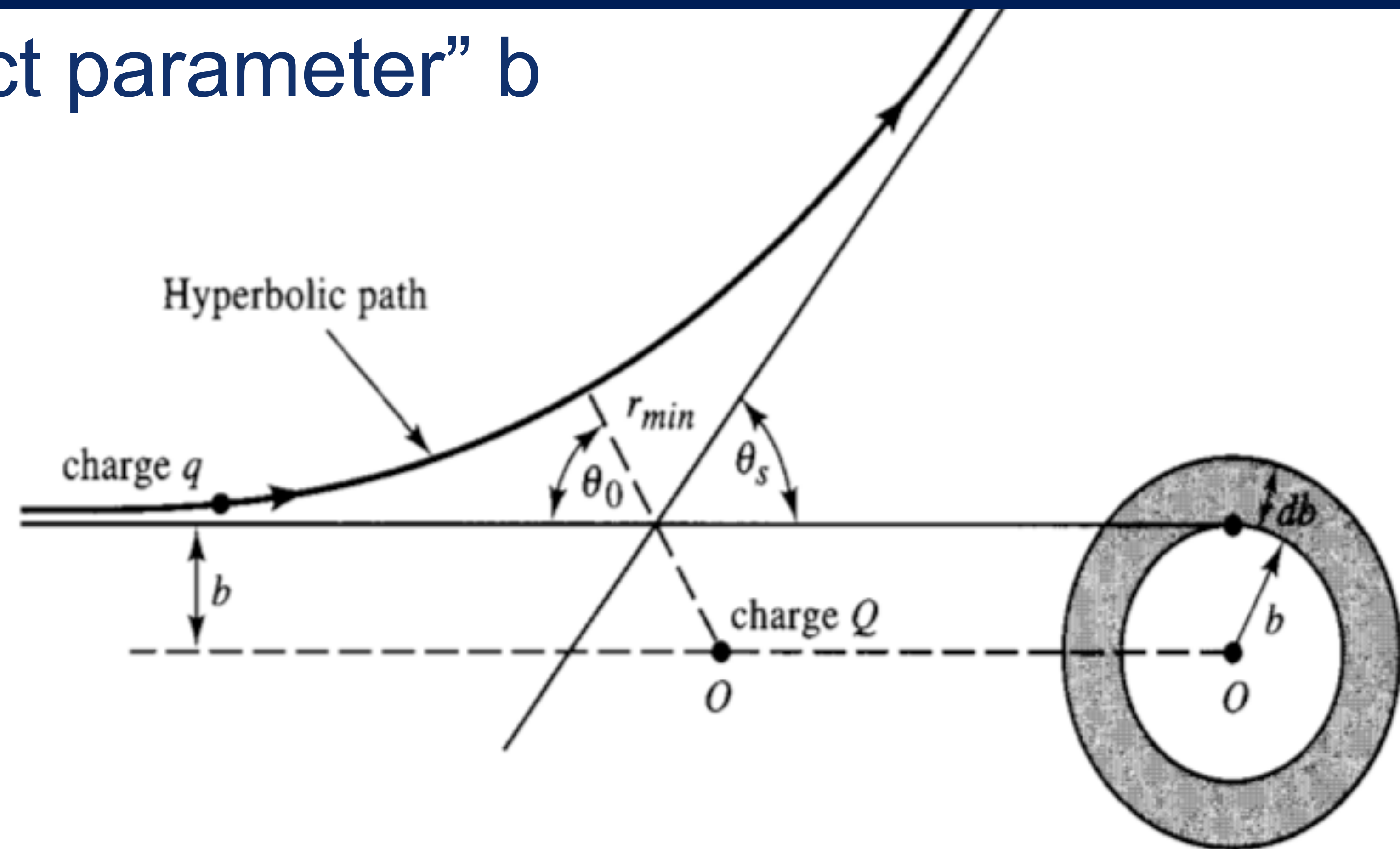


Figure 6.14.1 Hyperbolic path (orbit) of a charged particle moving in the inverse-square repulsive force field of another charged particle.

Fowles and Cassiday, Analytical Mechanics

- By conservation of angular + linear momenta and energy :

$$b = \frac{L}{mv_0} = \frac{L}{\sqrt{2mE}} = \frac{|\ell|}{\sqrt{2E/m}}$$

Cross sections

- Can solve the energy formula to get a parametric equation for r in terms of theta :

$$\frac{dr}{d\theta} = \pm\frac{r^2}{b}\sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}$$

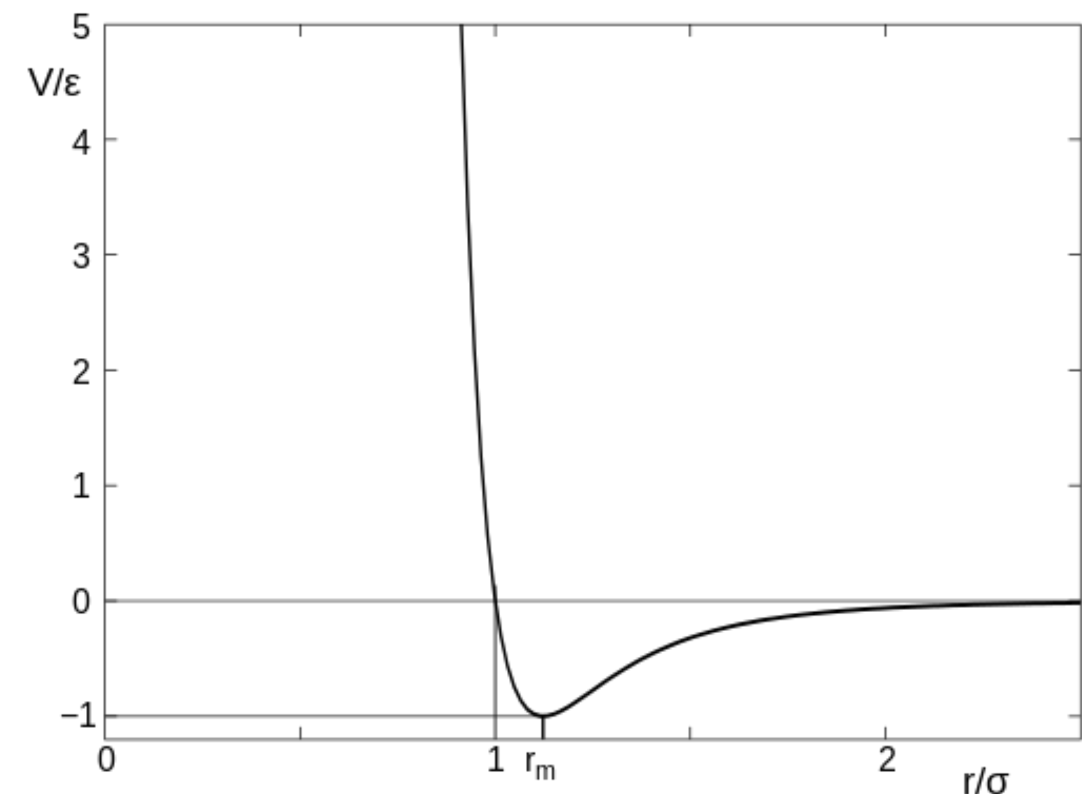
- At the point of closest approach (PCA) the derivative is zero, so define this as r_{min}.

Cross sections

- Typically we have experiments with many incident particles ("beam")
- Then we can consider a distribution of impact parameters with density

$$2\pi b\, db$$

- Classically, given E and b, you can get the unique scattering angle theta
- Example : Lennard-Jones potential for interactions between pairs of neutral atoms or molecules



$$V(r) = 4V_0\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right]$$

- Interesting bit is that more than one b can lead to the same theta!

Cross sections

- Consider a differential of the impact parameter. The scattering angles will therefore be in the range :

$$[\theta, \theta + d\theta] = \left[\theta(b, E), \theta(b, E) + \frac{d\theta(b, E)}{db} db\right]$$

- Typically detectors of particles are located "at infinity" (far away)
- They exist at some angle θ_s, and subtend some physical space (solid angle $d\Omega$)

- Thus we have :

$$(\text{Incident particles per unit area per unit time}) \times \text{Area} = 2\pi\, \mathcal{I}\, b\, db$$

Cross sections

- Now, consider the differential scattering cross section :

$$\sigma(\theta_s) = \frac{\text{Number detected per unit time}}{(\text{Incident Intensity}) \times d\Omega} = \frac{2\pi b\, db}{2\pi \sin\theta_s\, d\theta_s} = \frac{b}{\sin\theta_s}\left|\frac{d\theta_s}{db}\right|^{-1}$$

- Now, since many incident particles are detected in the same "slice" of the detector, define a **deflection angle** as the total number of radians that the position vector rotates along the trajectory :

$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt}\,dt = \pi - 2b\int_{r_{\min}}^{\infty} \frac{dr}{r^2\sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Cross sections

- The scattering angle is related to the deflection angle:

$$0 \leq \theta_s = \pm\Theta - 2n\pi \leq \pi$$

- And the differential cross section is :

$$\frac{d\sigma}{d\Omega} = \frac{\text{Number detected per unit time}}{(\text{Incident Intensity}) \times d\Omega} = \frac{2\pi b\, db}{2\pi \sin\theta_s\, d\theta_s} = \frac{b}{\sin\theta_s}\left|\frac{d\theta_s}{db}\right|^{-1}$$

Hey look! A discrete sum!

Scattering

- Recall definition of r_min :

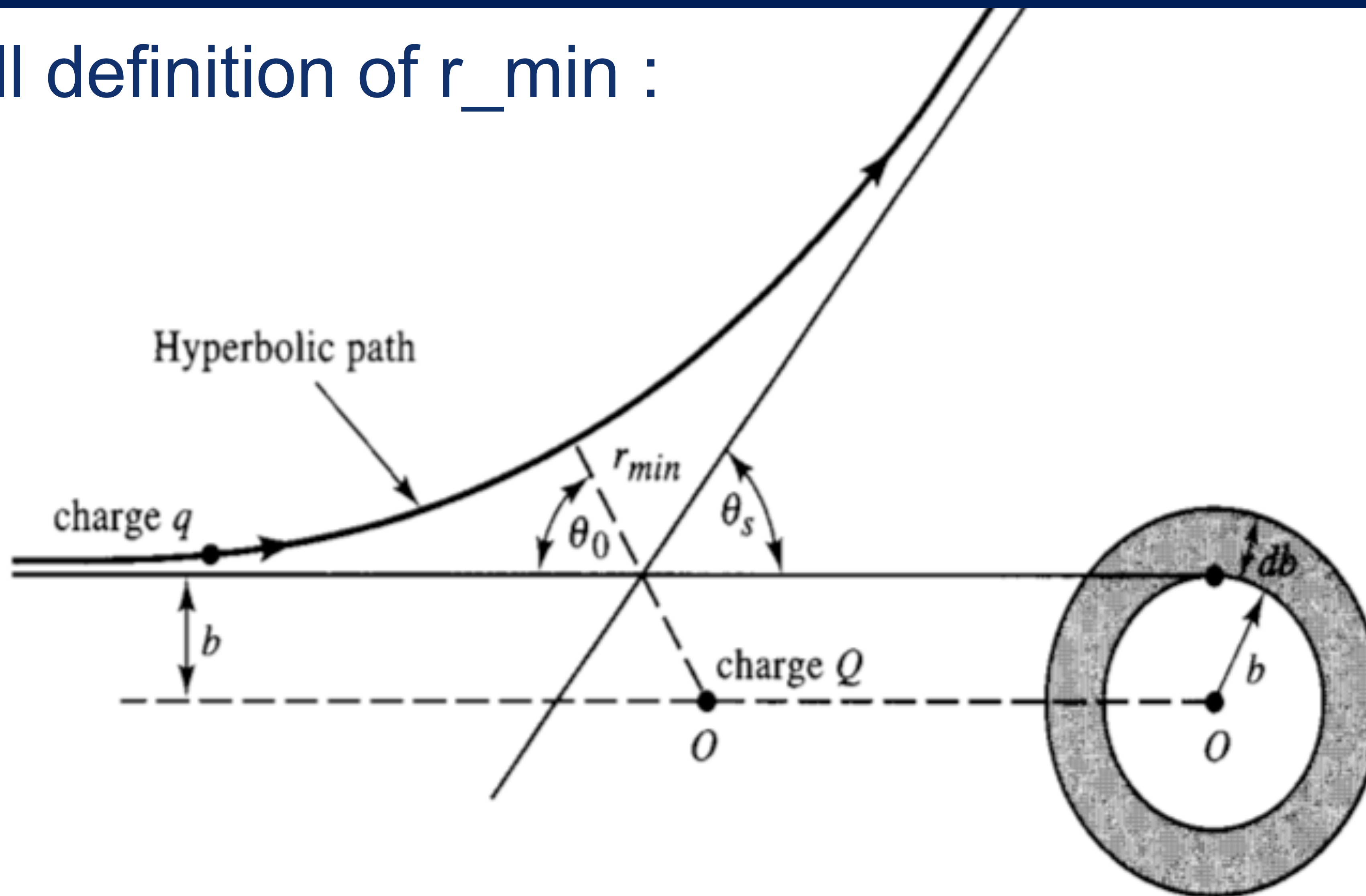


Figure 6.14.1 Hyperbolic path (orbit) of a charged particle moving in the inverse-square repulsive force field of another charged particle.

- We have shown : $$\frac{dr}{d\theta} = \pm\frac{r^2}{b}\sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}$$

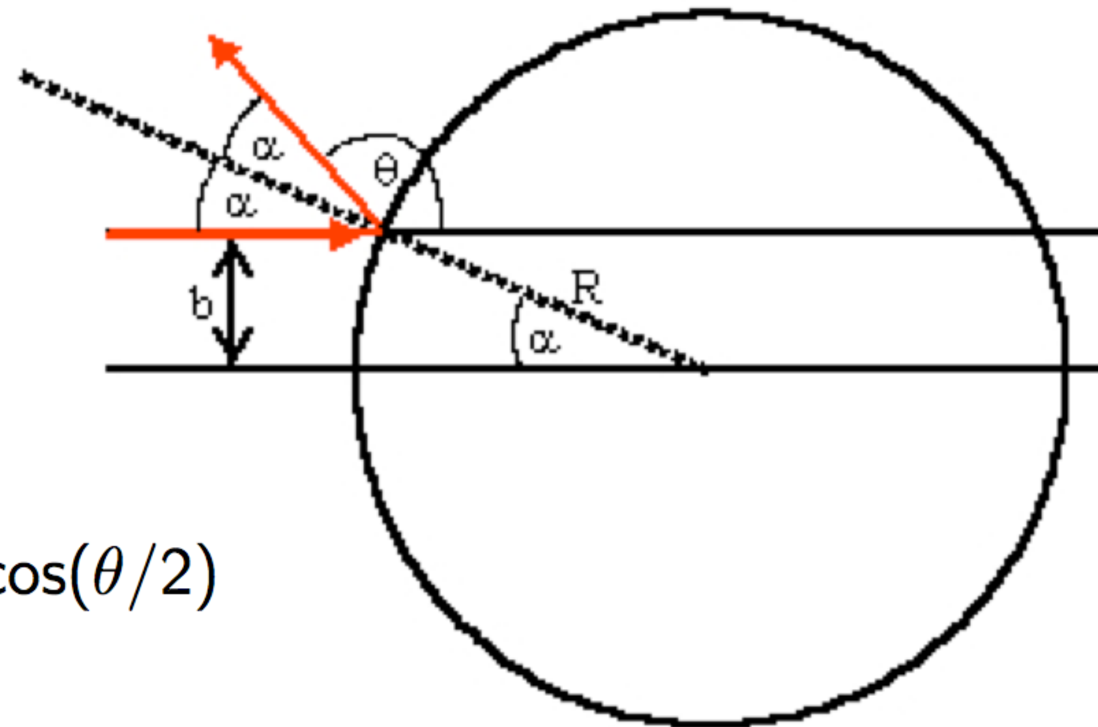
  – RHS is zero at r_min (Yay! It's a root!)

Scattering

- Also recall the differential cross section :

$$\frac{d\sigma}{d\Omega} = \frac{\text{Number detected per unit time}}{(\text{Incident Intensity}) \times d\Omega} = \frac{2\pi b\, db}{2\pi \sin\theta_s\, d\theta_s} = \frac{b}{\sin\theta_s} \left| \frac{d\theta_s}{db} \right|^{-1}$$

- If we can compute dtheta/db, we can get the scattering cross section
- Example : hard sphere



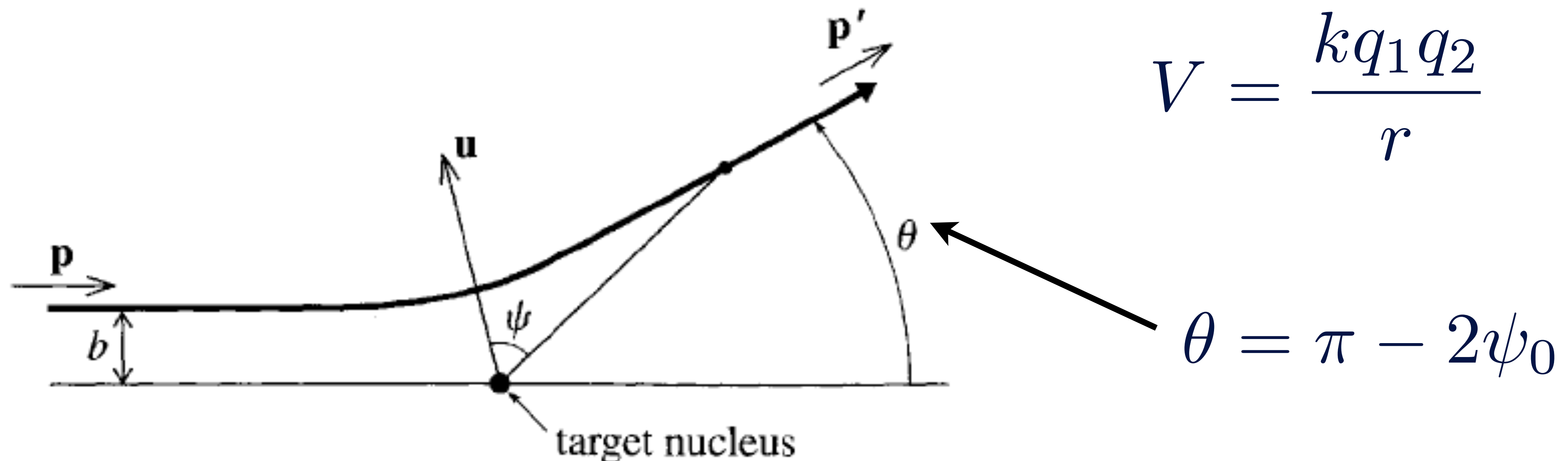
- So, we have

$$b(\theta) = R\sin\alpha = R\sin\left(\frac{\pi-\theta}{2}\right) = -R\cos(\theta/2)$$

- Thus : $$\frac{db}{d\theta_s} = \frac{R}{2}\sin\left(\frac{\theta_s}{2}\right)$$

Scattering

- Example : Rutherford Scattering : EM scattering of object with charge q1 off of an object with charge q2



- Look at the change in momentum : $\Delta\mathbf{p} = \mathbf{p}' - \mathbf{p}$

http://tberg.dk/books/Classical_Mechanics_(Taylor).pdf

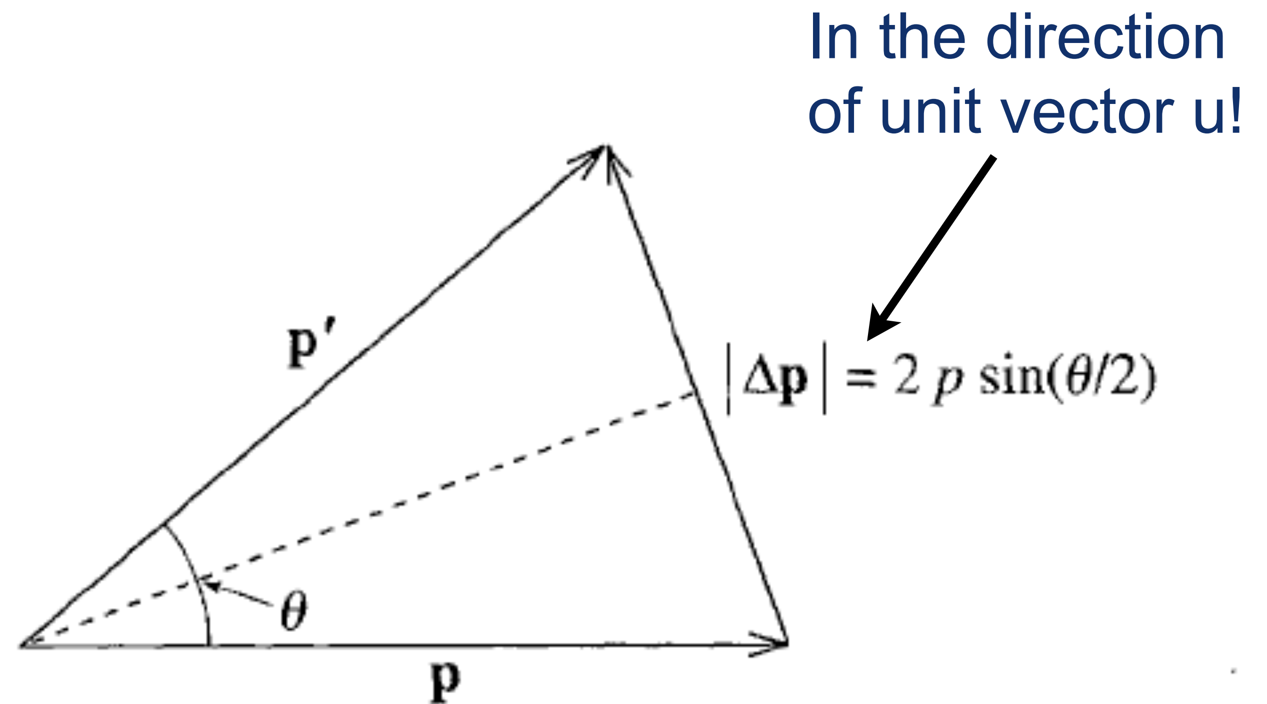
Scattering

- We know that $|\mathbf{p}'| = |\mathbf{p}|$

  so we can write

$$|\Delta \mathbf{p}| = 2p \sin \theta/2$$

- We get an isosceles triangle :



- But, we know from Newton's second law:

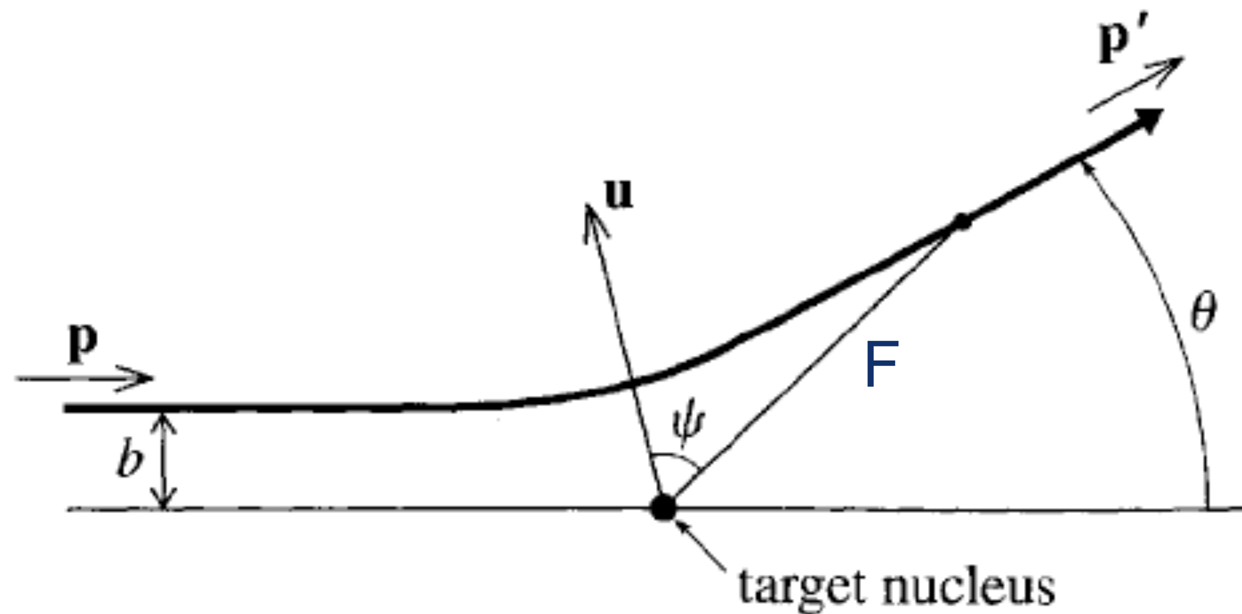
$$\Delta \mathbf{p} = \int \mathbf{F} \Delta t$$

- Since F is in the direction of u, we perform this in one dimension:

$$|\Delta \mathbf{p}| = \int_{-\infty}^{\infty} |\mathbf{F_u}| \Delta t$$

Scattering

- The components of the integral cancel except for the force in the u direction, so investigating this again:



$$|\mathbf{F_u}| = \mathbf{F} \cdot \hat{u} = \frac{kq_1q_2}{r^2}\cos\psi$$

- Thus :

$$|\Delta\mathbf{p}| = \int_{-\infty}^{\infty} \frac{kq_1q_2}{r^2}\cos\psi dt$$

Scattering

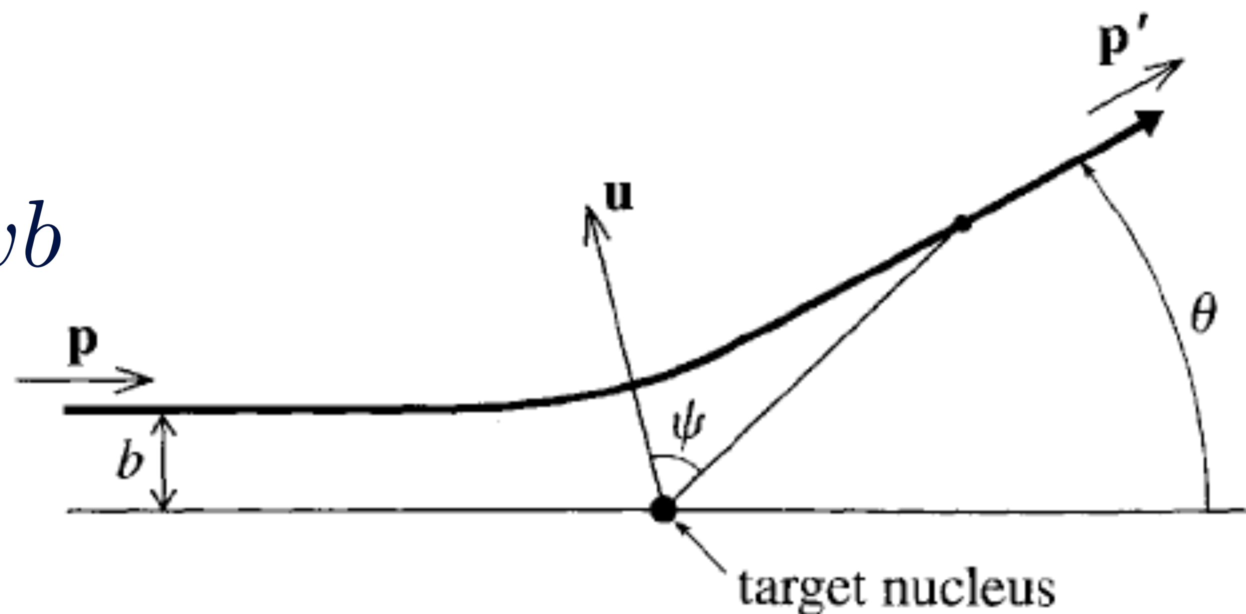
- Now use a trick :

$$\dot{\psi} = \frac{d\psi}{dt}$$

$$dt = \frac{d\psi}{\dot{\psi}}$$

- Can use conservation of angular momentum to solve for $\dot{\psi}$

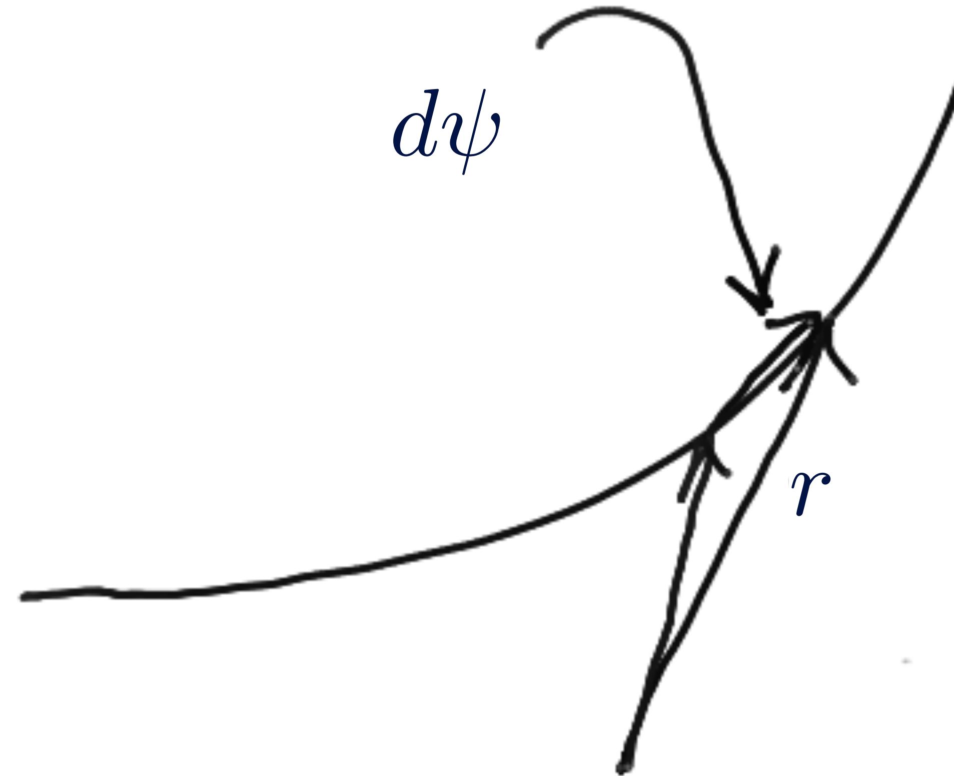
$$|\vec{L_1}| = mvb$$



$$|\vec{L_2}| = |\mathbf{r} \times \mathbf{p}'|$$

Scattering

- Solving for the magnitude of L_2 :



- Tangential velocity is:

$$v = r\frac{d\psi}{dt}$$

- So

$$|\mathbf{L_2}| = mr^2\frac{d\psi}{dt} = mr^2\dot{\psi}$$

- Finally can substitute this into the integral:

$$|\Delta\mathbf{p}| = \int_{-\infty}^{\infty} \frac{kq_1q_2}{r^2}\cos\psi dt \quad = \int \frac{kq_1q_2}{r^2}\cos\psi \frac{d\psi}{bp/mr^2}$$

Scattering

- Simplifying :

$$= \int_{-\psi_0}^{\psi_0} \frac{kq_1q_2m}{bp} \cos\psi d\psi$$

- And doing the integral, we get :

$$|\Delta \mathbf{p}| = \frac{2kq_1q_2m}{bp} \cos\theta/2$$ and $$|\Delta \mathbf{p}| = 2p\sin\theta/2$$

- We solve for b:

$$|\Delta \mathbf{p}| = \frac{2kq_1q_2m}{bp} \cos\theta/2 = 2p\sin\theta/2$$

$$b = \frac{kq_1q_2}{mv^2} \cot\theta/2$$

Scattering

- Can finally put it together and compute scattering cross section:

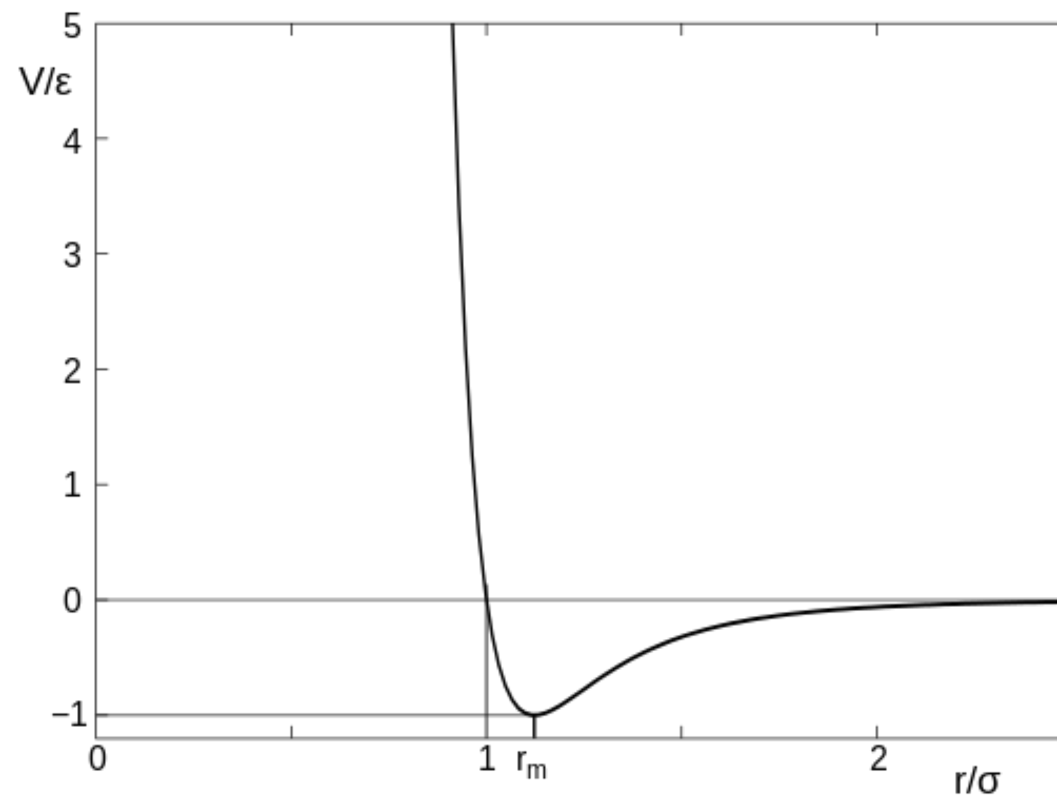
$$\frac{d\sigma}{d\Omega} = \frac{\text{Number detected per unit time}}{(\text{Incident Intensity}) \times d\Omega} = \frac{2\pi b\, db}{2\pi \sin\theta_s\, d\theta_s} = \frac{b}{\sin\theta_s}\left|\frac{d\theta_s}{db}\right|^{-1}$$

- in this case :

$$\frac{d\sigma}{d\Omega} = \left(\frac{kq_1k_2}{4E\sin^2\theta/2}\right)^2$$

Scattering

- Finally consider the Lennard-Jones potential:



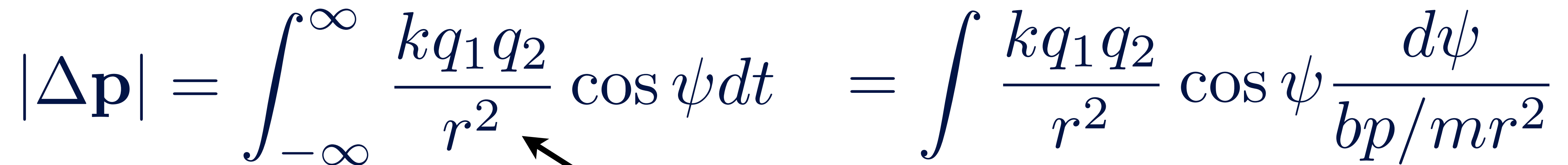
$$V(r) = 4V_0 \left[\left(\frac{r_0}{r} \right)^{12} - \left(\frac{r_0}{r} \right)^{6} \right]$$

Scattering

- How would we go about computing this?
- Of course, we need to do it numerically!
- Or rather : you’ll compute it numerically in your homework!
- Let’s sketch it out

Scattering

- Critical bit is here :



$$|\Delta \mathbf{p}| = \int_{-\infty}^{\infty} \frac{kq_1q_2}{r^2} \cos\psi dt \quad = \int \frac{kq_1q_2}{r^2} \cos\psi \frac{d\psi}{bp/mr^2}$$

- We had the force in the integrand, but the factors of r canceled fortuitously

- Can use another (less fortuitous) trick, though. Limits of integration were $\pm\psi_0$
- However, this is

$$\psi_0 = \int_0^{\infty} \dot{\psi} dt$$

- We can use the same trick:

$$\psi_0 = \int_{r_{\min}}^{\infty} \frac{\dot{\psi}}{\dot{r}} dr$$

Scattering

- Rewriting all of this in terms of E, v, and the potential, this is our total deflection angle:

$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2\sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

- In order to plot the differential cross section, we :
  - Compute this integral numerically for several b's
  - Compute the derivative $\frac{d\Theta}{db}$ numerically for those b's
  - We'd then have

$$\frac{d\sigma}{d\Omega} = \frac{\text{Number detected per unit time}}{(\text{Incident Intensity}) \times d\Omega} = \frac{2\pi b\, db}{2\pi \sin\theta_s\; d\theta_s} = \frac{b}{\sin\theta_s} \left| \frac{d\theta_s}{db} \right|^{-1}$$

Scattering

- To do this, we must compute the deflection angle :

$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

- Given r_min, we can compute the integral

- Therefore, this is a two-step problem :

  – Compute r_min numerically
  – Compute integral

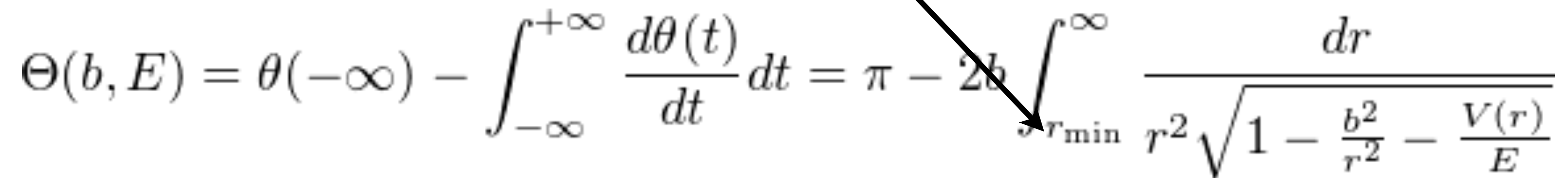
Scattering

- Our overall plan is thus :
  - Set up scattering problem (E, b)
  - Find r_min numerically
  - Integrate dtheta/dr(b,E) numerically, given r_min

$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Scattering

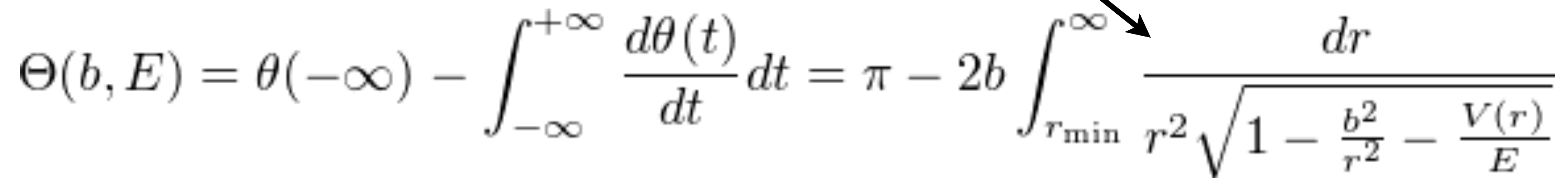
- Our overall plan is thus :
  - Set up scattering problem (E, b)
  - Find r_min numerically
  - Integrate dtheta/dr(b,E) numerically, given r_min



$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Scattering

- Our overall plan is thus :
  - Set up scattering problem (E, b)
  - Find r_min numerically
  - Integrate dtheta/dr(b,E) numerically, given r_min



Scattering

- Find r_min numerically :
  - Recall :
  - r_min is defined by dr/dtheta = 0
  - Function is:

$$\frac{dr}{d\theta} = \pm\frac{r^2}{b}\sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}$$

  - So, we find the root of this!

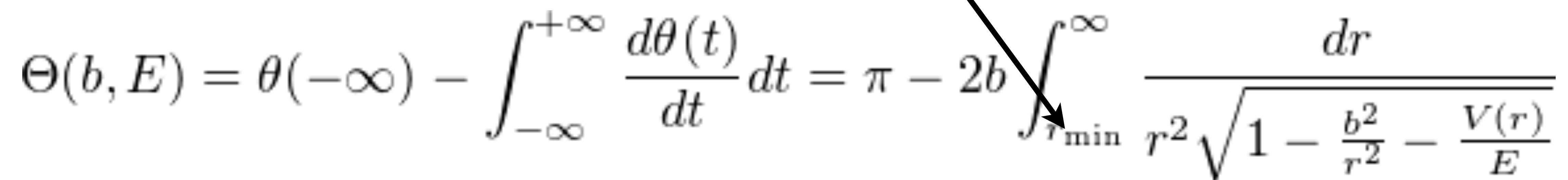
Scattering

- Integrate dtheta/dr numerically :
  - Find r_min (from previous)
  - Initialize to pi
  - Integrate over a small “dTheta” with Trapezoid rule
  - Add up the dTheta’s to get total

$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Scattering

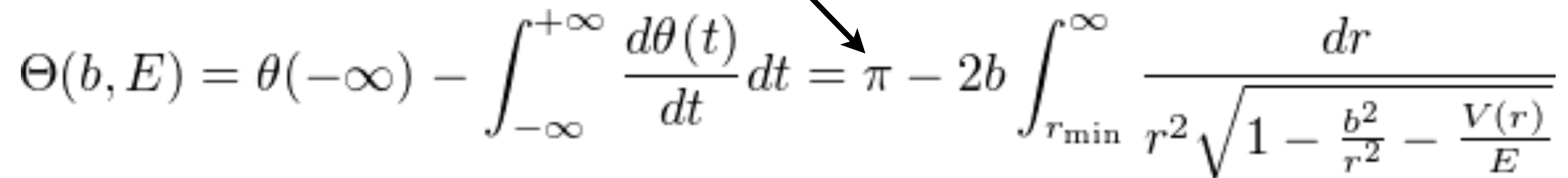
- Integrate dtheta/dr numerically :
  - Find r_min (from previous)
  - Initialize to pi
  - Integrate over a small “dTheta” with Trapezoid rule
  - Add up the dTheta’s to get total



$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Scattering

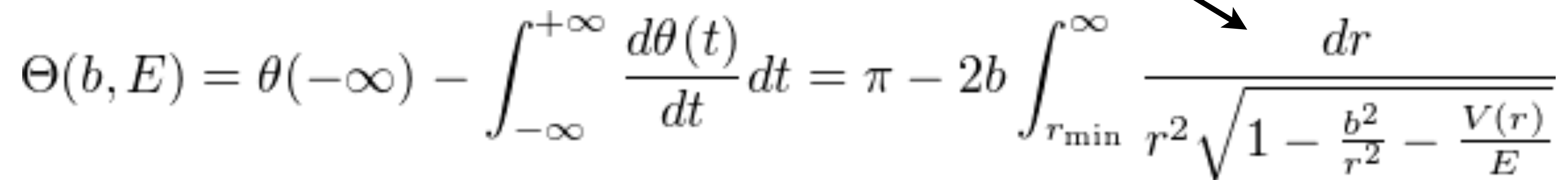
- Integrate dtheta/dr numerically :
  - Find r_min (from previous)
  - Initialize to pi
  - Integrate over a small "dTheta" with Trapezoid rule
  - Add up the dTheta's to get total



$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Scattering

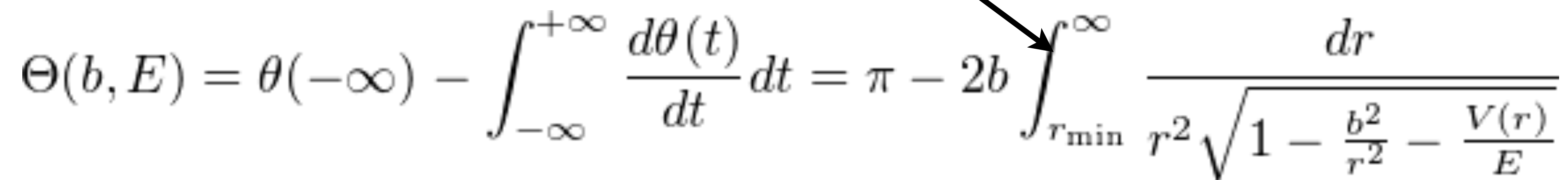
- Integrate dtheta/dr numerically :
  - Find r_min (from previous)
  - Initialize to pi
  - Integrate over a small "dTheta" with Trapezoid rule
  - Add up the dTheta's to get total



$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt}\,dt = \pi - 2b\int_{r_{\min}}^{\infty} \frac{dr}{r^2\sqrt{1-\frac{b^2}{r^2}-\frac{V(r)}{E}}}$$

Scattering

- Integrate dtheta/dr numerically :
  - Find r_min (from previous)
  - Initialize to pi
  - Integrate over a small "dTheta" with Trapezoid rule
  - Add up the dTheta's to get total



$$\Theta(b,E) = \theta(-\infty) - \int_{-\infty}^{+\infty} \frac{d\theta(t)}{dt} dt = \pi - 2b \int_{r_{\min}}^{\infty} \frac{dr}{r^2 \sqrt{1 - \frac{b^2}{r^2} - \frac{V(r)}{E}}}$$

Orbiting

- As you know, oftentimes in scattering, the potentials are attractive and the incoming particle can orbit the other
  - Gravitational capture
  - Electron capture
- We can also investigate orbiting in our example

- We're computing the deflection angle Theta, but if you're orbiting, this can go completely nuts (somewhat obviously)

Orbiting

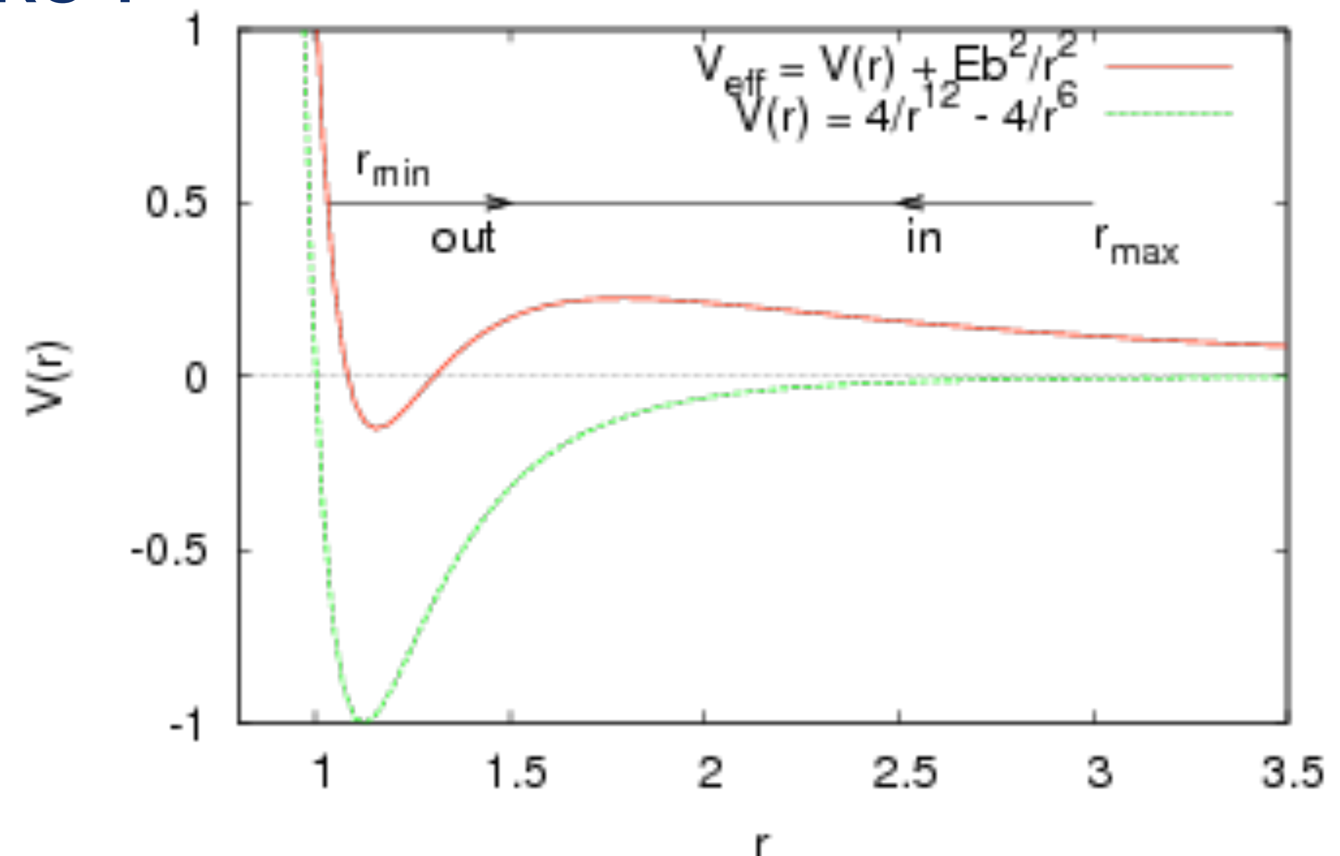
- Define the effective potential for scattering as the sum of the actual potential, and the centrifugal potential (from angular momentum of the incoming particle) :

$$V_{\text{eff}}(r) = V(r) + E\left(\frac{b}{r}\right)^2$$

- Then this looks something like :
- Orbiting occurs when E equals the max of the effective potential

$$\left.\frac{dV_{\text{eff}}(r)}{dr}\right|_{r=r_{\max}} = 0$$

$$V_{\text{eff}}(r_{\max}) = E$$



Scattering Pseudocode

Integrate dtheta/dr numerically :

- Initialize to pi
- Find r_min (from previous)
- Integrate over a small “dTheta” with Trapezoid rule
- Add up the dTheta’s to get total

```
def trajectory( self ) :
    # Define theta step :
    dtheta = -1.0 * asin( self.b / self.r_max )
    # To return : list of trajectories
    rtheta = [self.r_max, pi + dtheta]
    traj = [ array( rtheta ) ]
    # To return : Total deflection
    deflection = pi - 2*dtheta


    # Find the distance of closest approach with the "root_simple" method
    dr = -1.0 * self.r_max / 100
    r_max = self.r_max
    r_min = root_simple( self.f_r_min, r_max, dr)

    # Integrate to find successive changes in theta :
    dr = (r_max - r_min) / self.steps
    accuracy = 1e-6
    for i in xrange(self.steps) :
        r_upper = traj[i][0]
        r_lower = r_upper - dr
        itheta = traj[i][1]
        dtheta = -self.b * adaptive_trapezoid( self.dTheta_dr, r_lower, r_upper, accuracy )
        rtheta[0] -= dr
        rtheta[1] += dtheta
        traj.append( array( rtheta ) )
        deflection += 2 * dtheta


    # Use symmetry to get the outgoing trajectory points
    for i in range( self.steps-1, 0, -1) :
        rtheta[0] += dr
        dtheta = traj[i][1] - traj[i-1][1]
        rtheta[1] += dtheta
        traj.append( array( rtheta ) )

    return [deflection, traj]
```

Scattering Pseudocode

- For each value of b:
  - Calculate deflection angle
  - Plot x vs y of scatter
- Be careful about rmax!
  – Make sure it makes sense!

```
int main()
{

  using namespace std;
  cout << " Classical Scattering from Lennard-Jones potential" << endl;
  double E = 0.705;       // set global value of E
  cout << " Energy E = " << E << endl;
  double b_min = 0.6, db = 0.3;
  int n_b = 6;
  double b = 0.0;
  double V0 = 1.0;
  cout << " b       " << '\t' << "Theta(b)\n"
       << " -------" << '\t' << "---------" << endl;
  lennard_jones lj( V0 );
  for (int i = 0; i < n_b; i++) {

    stringstream sstream;
    sstream << "trajfile_cpp_" << i << ".data";
    ofstream file(sstream.str().c_str());

    b = b_min + i * db;
    std::vector< std::pair<double,double> > trajectory;
    double deflection = 0.0;
    Theta<lennard_jones> theta( lj, E, b, 3.5, 100 );
    theta.trajectory(deflection, trajectory);
    std::cout << " " << b << "\t\t" << deflection << std::endl;
    for (int i = 0; i < trajectory.size(); i++) {
      double r = trajectory[i].first;
      double theta = trajectory[i].second;
      char buff[1000];
      sprintf(buff, "%8.4f %8.4f", r*cos(theta), r*sin(theta));
      file << buff << std::endl;
    }
    file << std::endl;

    file.close();
  }

}
```