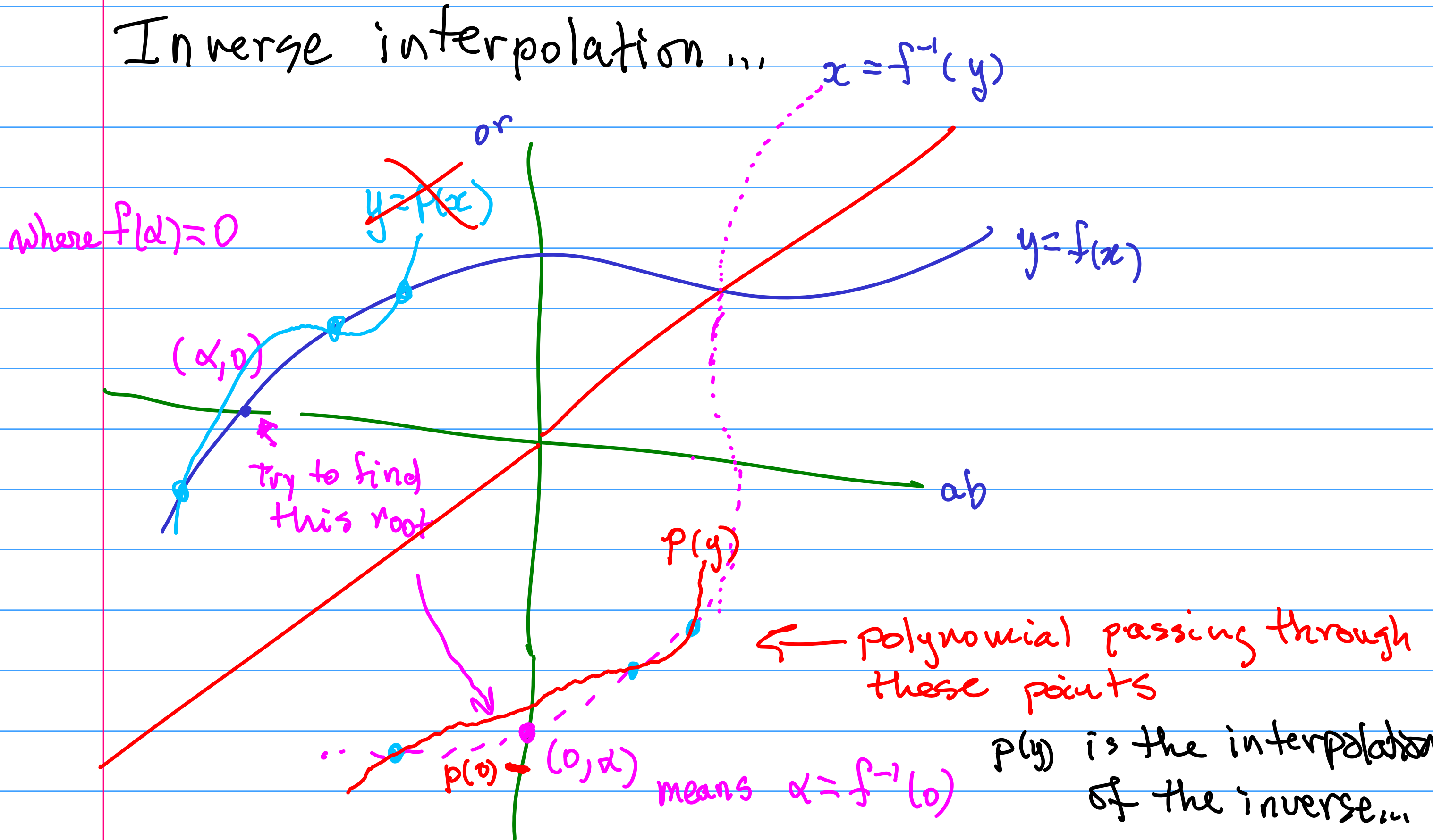
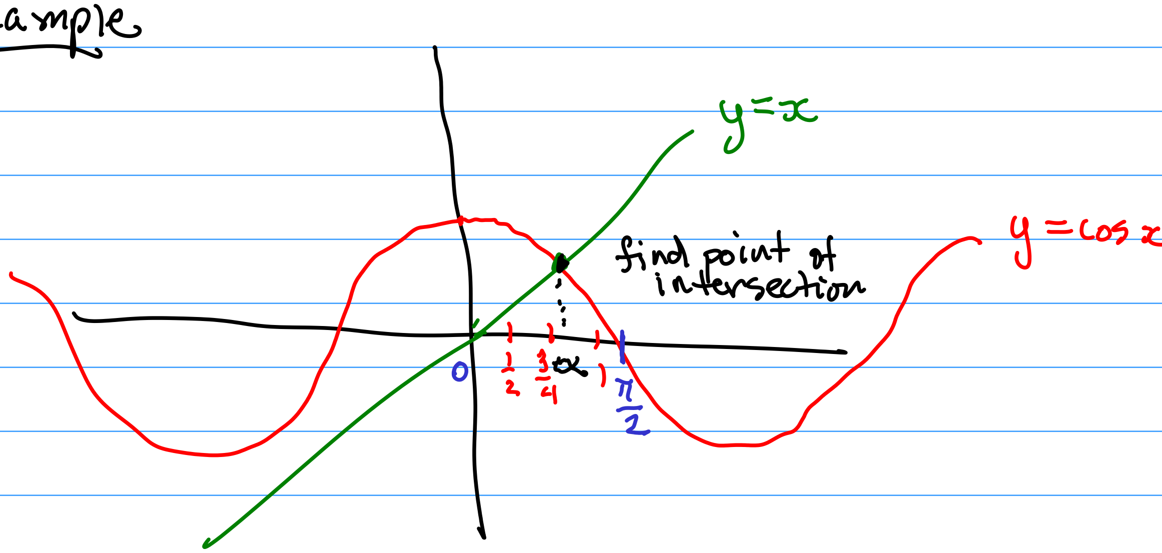
Inverse interpolation...



approximation $f^{-1}(y) \approx p(y)$

Approximation of $\alpha = f^{-1}(0) \approx p(0)$

Example



$f(x) = x - \cos x$ then $f(\alpha) = 0$

$x_1 = \frac{1}{2}$ $x_2 = \frac{3}{4}$ $x_3 = 1$

$f(x_1) = \frac{1}{2} - \cos\frac{1}{2}$ $f(x_2) = \frac{3}{4} - \cos\frac{3}{4}$ $f(x_3) = 1 - \cos 1$

Inverse interpolation

$$(f(x_1), x_1), \quad (f(x_2), x_2), \quad (f(x_3), x_3)$$

Notation

$$(y_1, x_1), \quad (y_2, x_2), \quad (y_3, x_3)$$

$$l_k(t) = \prod_{j \neq k} \frac{t - y_j}{y_k - y_j}$$

$$p(t) = \sum_{k=1}^{3} x_k \, l_k(t)$$

There are two things left in the material for this course

1. Numerically test the idea of inverse interpolation for finding roots
2. Develop the theory, similar to the quadratic convergence of Newton's method.

This one first

So we finish the example...

```
julia> x=[1/2,3/4,1]
3-element Vector{Float64}:
 0.5
 0.75
 1.0

julia> f(x)=x-cos(x)
f (generic function with 1 method)

julia> y=f.(x)
3-element Vector{Float64}:
 -0.37758256189037276
  0.018311131126179103
  0.45969769413186023
```

set up the points we'll use to construct the polynomial p that approximates the inverse...

Use Lagrange interpolating formula from last time, but switch the x's with the y's so $p(t)$ passes through the points

$$(y_1, x_1), \quad (y_2, x_2), \quad (y_3, x_3)$$

```
julia> function l(k,t)
           r=1
           for j=1:3
               if j!=k
                   r*=(t-y[j])/(y[k]-y[j])
               end
           end
           return r
       end
l (generic function with 1 method)
```

specify 3 points

These are now y rather than x.

```
julia> function p(t)
           s=0
           for k=1:3
               s+=x[k]*l(k,t)
           end
           return s
       end
p (generic function with 1 method)
```

This is now x instead of y

Now to approximate the root we only need evaluate

$$\alpha = p(0) \approx f^{-1}(0)$$

```
julia> alpha=p(0)
0.7389742930663052

julia> f(alpha)
-0.00018549886625851553
```

← approximation of the root

← residual error

recall that the backwards error analysis can be used to estimate the error in α from the value of $f(\alpha)$.

Now we iterate this by replacing one of the x's used in the interpolation with α.

```
julia> x=[x[2], x[3], alpha]
3-element Vector{Float64}:
 0.75
 1.0
 0.7389742930663052

julia> y=f.(x)
3-element Vector{Float64}:
  0.018311131126179103
  0.4596976941318602
 -0.00018549886625851553
```

add α delete x_1 and rotate x_2 and x_3 to the start of the vector

all these values we already computed. It's easier to recompute them here for the demonstration... but not efficient.

Now we iterate the same sequence of commands to find a new polynomial and new approximation of the root.

```
julia> alpha=p(0)
0.7390850869275516
```

better approximation

```
julia> f(alpha)
-7.746749852710622e-8
```

error is smaller

One can keep iterating, but we have to stop it at any point the new α is the same as the old one.
If that happens the polynomial p is no longer well defined, but that's okay because we've also found the root.

```
julia> x=[x[2], x[3], alpha]
3-element Vector{Float64}:
 1.0
 0.7389742930663052
 0.7390850869275516

julia> y=f.(x)
3-element Vector{Float64}:
  0.4596976941318602
 -0.00018549886625851553
 -7.746749852710622e-8

julia> alpha=p(0)
0.7390851332153578

julia> f(alpha)
3.3006930522105904e-13
```

another iteration

The residual error is now much smaller

We'll keep iterating until the method blows up...

```
julia> x=[x[2], x[3], alpha]
3-element Vector{Float64}:
 0.7389742930663052
 0.7390850869275516
 0.7390851332153578

julia> y=f.(x)
3-element Vector{Float64}:
 -0.00018549886625851553
 -7.746749852710622e-8
  3.3006930522105904e-13

julia> alpha=p(0)
0.7390851332151607

julia> f(alpha)
0.0
```

another iteration...

since the residual error is zero this is as accurate value for α as we can detect...

```
julia> x=[x[2], x[3], alpha]
3-element Vector{Float64}:
 0.7390850869275516
 0.7390851332153578
 0.7390851332151607

julia> y=f.(x)
3-element Vector{Float64}:
 -7.746749852710622e-8
  3.3006930522105904e-13
  0.0

julia> alpha=p(0)
0.7390851332151607

julia> f(alpha)
0.0
```

It's possible to iterate one more time because the y-values are all still different

There will be a problem, however, with the next iteration.

Trying to iterate one more time leads to repeated points

```
julia> x=[x[2], x[3], alpha]
3-element Vector{Float64}:
 0.7390851332153578
 0.7390851332151607
 0.7390851332151607
```
} same

```
julia> y=f.(x)
3-element Vector{Float64}:
 3.3006930522105904e-13
 0.0
 0.0
```
} same

```
julia> alpha=p(0)
NaN
```

The polynomial is not defined since there are only 2 unique points left

It blew up... not as exciting as when things blow up in a chemistry lab, but never mind... the root is

$$\boxed{\alpha = 0.7390851332151607}$$

The method seemed to converge fast, but how fast was it supposed to converge? We'll do an analysis similar to how we showed Newton's method was quadratically convergent after Thanksgiving... Note: one can use higher degree interpolating polynomials to obtain cubic, quartic or even faster rates of convergence.

At the same time, since quadratic convergence already implies the number of significant digits double at each iteration, a method where they triple at each iteration may not be needed — especially when only 15 significant digits are used for the calculation...

Since there was time, we tried the bignum support in Julia to get a better idea of the rate of convergence. I've polished that here as a loop...

```
julia> x=big.([1/2,3/4,1])
       y=f.(x)
       for k=1:6
           alpha=p(0)
           display(f(alpha))
           x=[x[2], x[3], alpha]
           y=f.(x)
       end
-0.0001854988662584421546202114791336074038840164570545697132045817579188357714550073
-7.746749851498057538649584925351760148732105829492674866709536450795429044872386e-08
3.300330057215580493116713805462457769535449707353119466080901740126757706331495e-13
2.787074436492344604278538618224473141878818061882155563254422435873116419300736e-25
-4.187201437563249195776071928598620031379743599085965363331499016925873630696337e-46
8.636168555094444625386351862800399571116000364436281385023703470168591803162427e-78
```

from here we see that the number of significant digits appears to be doubling with each iteration — just like Newton's method but with no derivatives needed to perform the iteration.

Have a Happy Thanksgiving!

next week we'll use the interpolation theorem to analyse the rate of convergence...