Interpolating polynomial Theorem

Suppose g is a function with $n+1$ derivatives and p is the interpolating polynomial of degree n through $(y_i, g(y_i))$ for $i = 0, 1, \ldots, n$. Then

$$g(y) = p(y) + (y-y_0)(y-y_1)\cdots(y-y_n)\frac{g^{(n+1)}(c)}{(n+1)!}$$

where c is some value between $\min(y, y_0, y_1, \ldots, y_n)$ and $\max(y, y_0, y_1, \ldots, y_n)$.

Finding roots $f(x) = 0$ using inverse interpolation.

let $g(y) = f^{-1}(y)$ in a neighborhood of the root

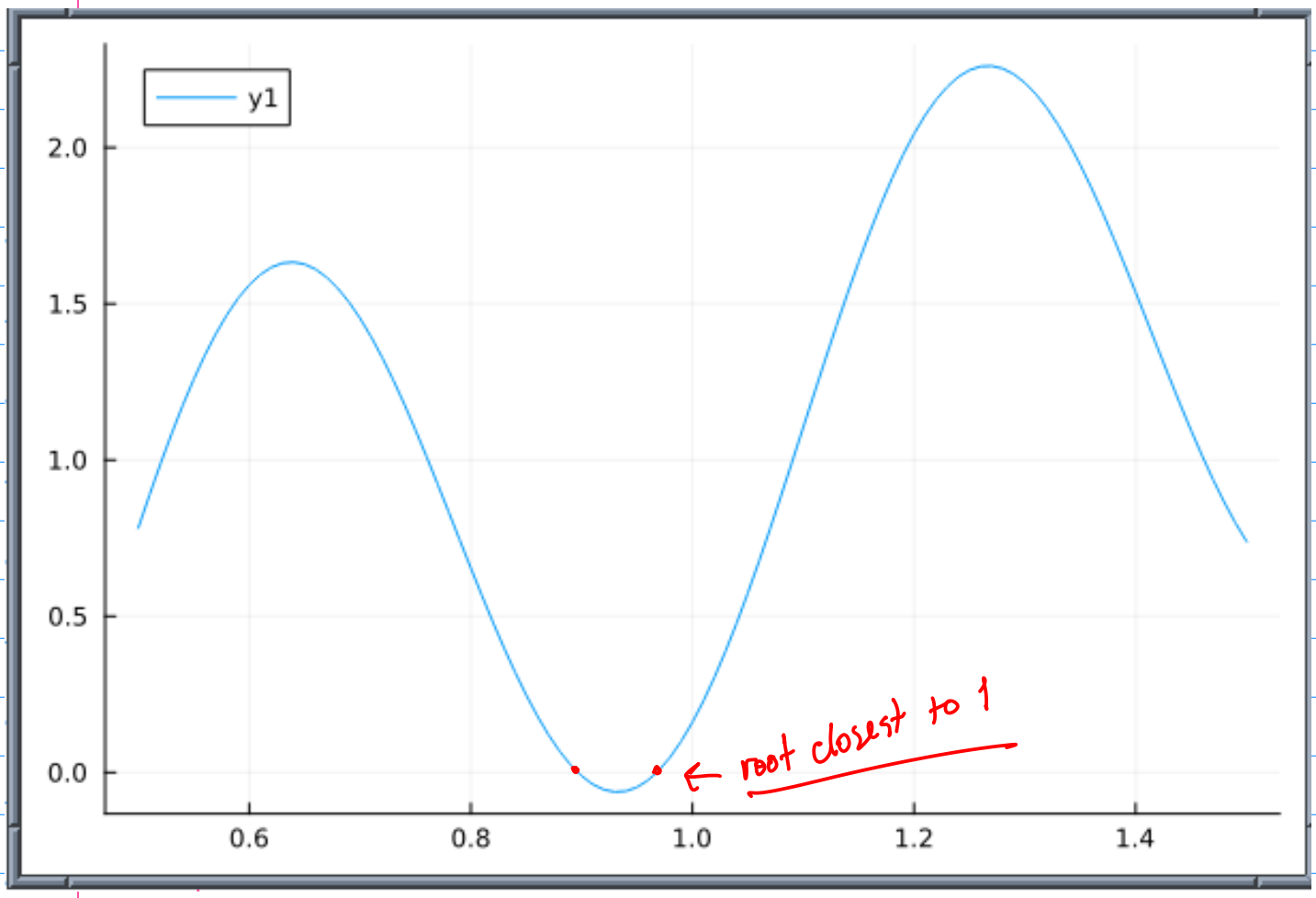
notation...

- If I knew f^{-1} in a neighborhood of the root then $r = f^{-1}(0)$ is the root

Since I don't know f^{-1} instead approximate it using an interpolating polynomial.

Example from book

```
julia> f(x)=x+cos(10*x)
f (generic function with 1 method)

julia> using Plots

julia> plot(f,0.5:0.01:1.5)
```



I could use Newton method

$f(x) = x + \cos 10x$, $f'(x) = 1 - 10\sin 10x$

Newton iteration

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

$$= x_n - \frac{x_n + \cos 10x_n}{1 - 10\sin 10x_n}$$

initial approx

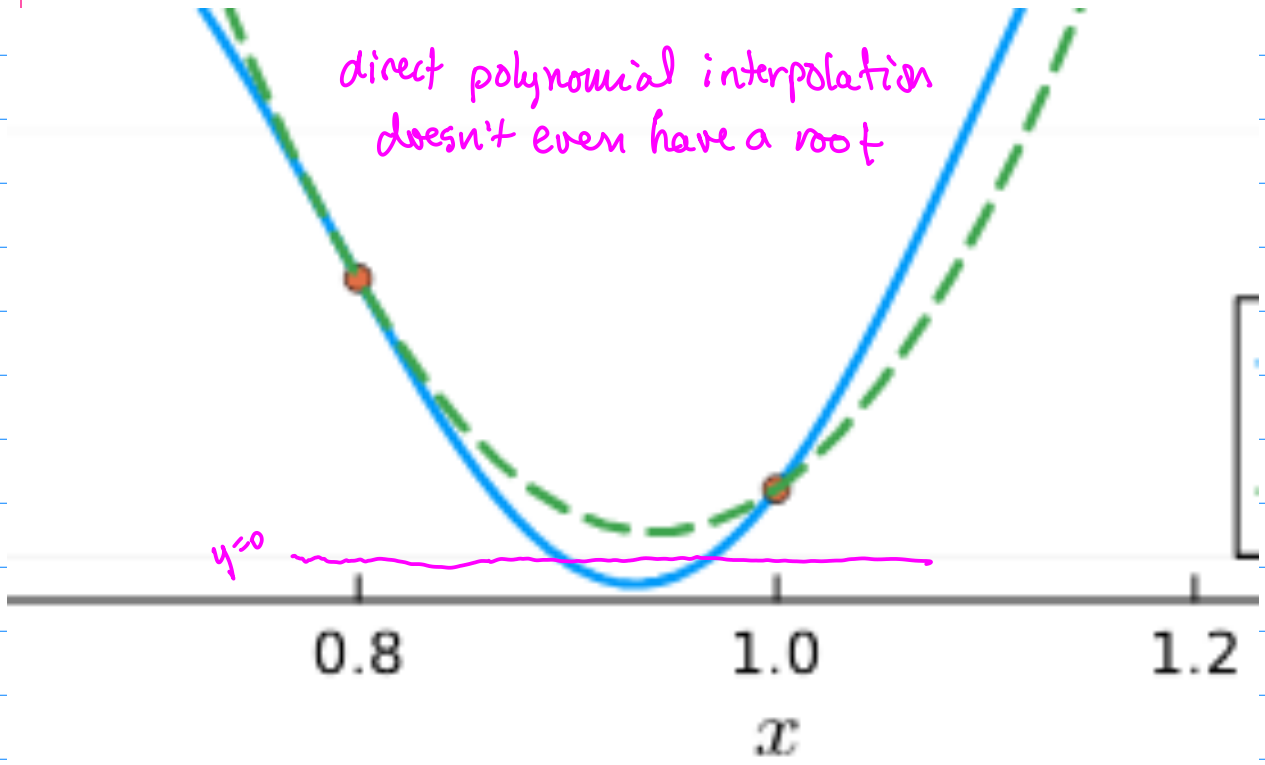
$x_0 = 1$

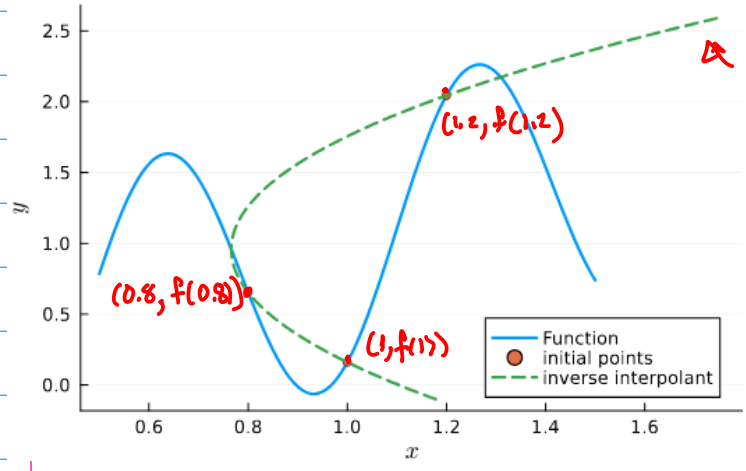
```
julia> g(x)=x-f(x)/df(x)
g (generic function with 1 method)

julia> df(x)=1-10*sin(10*x)
df (generic function with 1 method)

julia> x0=1
1

julia> xn=x0
       for n=1:5
           xn=g(xn)
           println("xn=",xn)
       end
xn=0.975011926130603
xn=0.9684656613050743
xn=0.9678929149590452
xn=0.9678884021293123
xn=0.9678884018488256
```

need derivative.

compute f and f' each iteration.

only this digit changes next iteration.





↑ polynomial through the points

```
julia> xs=[0.8,1.2,1]
3-element Vector{Float64}:
 0.8
 1.2
 1.0

julia> f.(xs)
3-element Vector{Float64}:
 0.6544999661913865
 2.043853958732492
 0.16092847092354756
```

$(f(0.8), 0.8)$, $(f(1.2), 1.2)$ $(f(1), 1)$

Create polynomial passing through these points

Vandermonde Matrix : $p(y) = ay^2 + by + c \cdot 1$

$$\begin{bmatrix} 1 & y_1 & y_1^2 \\ 1 & y_2 & y_2^2 \\ 1 & y_3 & y_3^2 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

```
julia> V=[(y->y^j)(f(xs[i])) for i=1:3, j=0:2]
3×3 Matrix{Float64}:
 1.0  0.6545    0.42837
 1.0  2.04385   4.17734
 1.0  0.160928  0.025898
```

values of y going into the matrix

```
julia> coef=V\xs
3-element Vector{Float64}:
  1.1039813854404716
 -0.7053726758383064
  0.3681045155182172
```

← values of x on other side

← c ← b ← a

```
julia> c,b,a=coef
3-element Vector{Float64}:
  1.1039813854404716
 -0.7053726758383064
  0.3681045155182172
```

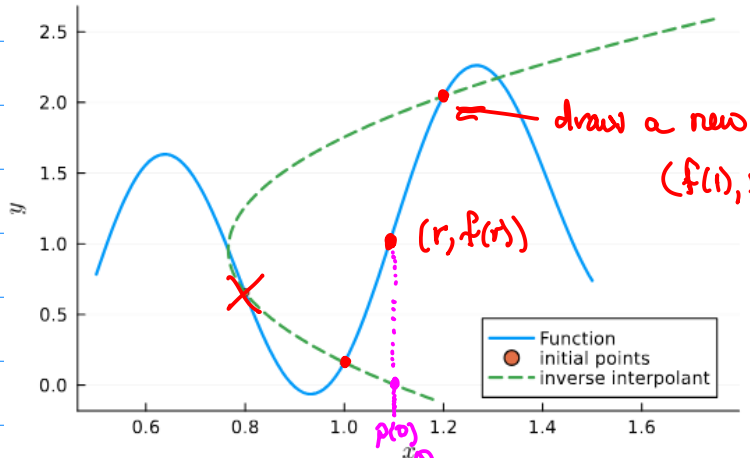
```
julia> p(y)=a*y^2+b*y+c
p (generic function with 1 method)
```

← factor nested

```
julia> p(y)=(a*y+b)*y+c
p (generic function with 1 method)
```

approximation of root is $r = p(0)$

```
julia> p(0)
1.1039813854404716
```



draw a new polynomial through (f(1), 1), (f(1.2), 1.2), (f(1.104...), 1.104...)

(r, f(r))

p(0)

new approx of root...

```
julia> xs=[1,1.2,p(0)]
3-element Vector{Float64}:
 1.0
 1.2
 1.1039813854404716

julia> V=[(y->y^j)(f(xs[i])) for i=1:3, j=0:2]
3×3 Matrix{Float64}:
 1.0  0.160928  0.025898
 1.0  2.04385   4.17734
 1.0  1.14821   1.31838
```

```
julia> coef=V\xs
3-element Vector{Float64}:
 0.9832357453813031
 0.10401102026654052
 0.0010008570265645396
```

```
julia> c,b,a=coef
3-element Vector{Float64}:
 0.9832357453813031
 0.10401102026654052
 0.0010008570265645396
```

```
julia> p(y)=(a*y+b)*y+c
p (generic function with 1 method)

julia> p(0)
0.9832357453813031
```

new approx of root.

first digit correct...

```
xn=0.9678929149590452
xn=0.9678884021293123
xn=0.9678884018488256
```

- This is after 2 iterations... this inverse interpolation method using parabolas actually converges faster than Newton's method... we will see this theoretically using the polynomial interpolation theorem after the theoretical midterm.