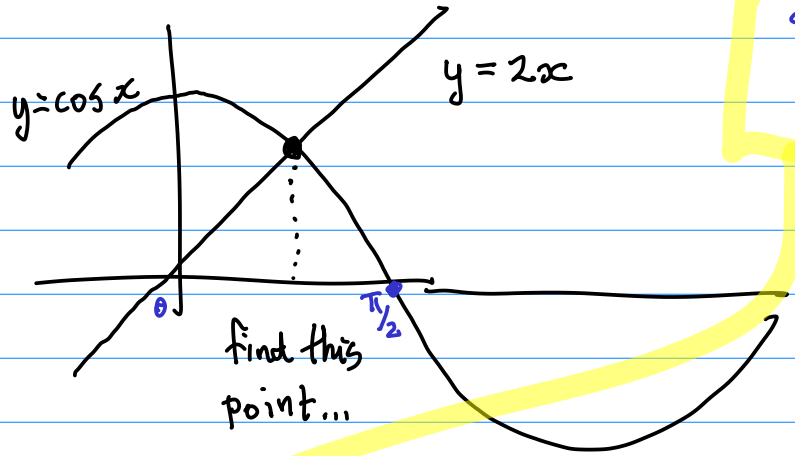
Username: guest
Password: math466

Today we'll learn about Julia...



$$\begin{cases} y = 2x \\ y = \cos x \end{cases}$$

initial guess

$x_0 = \frac{\pi}{4}$

$2x = \cos x$

Newton's Method: Solve $f(x) = 0$

$f'(x) = 2 + \sin x$ $\quad f(x) = 2x - \cos x$

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

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _    |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |   |
  | | |_| | | | (_| |   |  Version 1.6.1 (2021-04-23)
 _/ |\__'_|_|_|\__'_|   |
|__/                    |

julia> 1+1
2

julia> typeof(1)
Int64

julia> typeof(1.0)
Float64
```

Float64 is the 64-bit floating point we discussed last time...

Define functions simple way in Julia...

```
julia> f(x)=2*x-cos(x)
f (generic function with 1 method)

julia> df(x)=2+sin(x)
df (generic function with 1 method)
```

Define functions the complicated way...

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

Edit by pressing arrow up and then arrow back. Insert an additional line in the REPL by pressing alt+return...

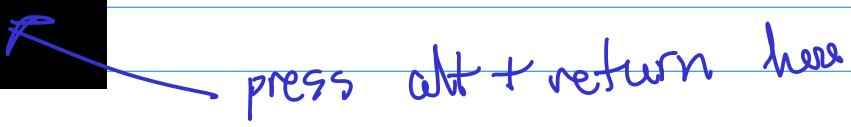
```
julia> function g(x)
           println(x)
           return x-f(x)/df(x)
       end
```

The initial condition for Newton's method...

```
julia> x0=pi/4
0.7853981633974483
```

Create a block of code by adding a line using alt+return

```
julia> x=x0
```



```
julia> x=x0
       for n=1:10
           x=g(x)
           println("x=$x")
       end
```

then enter the rest of the loop...

The REPL is interactive so it will run when you press return after entering the final end.

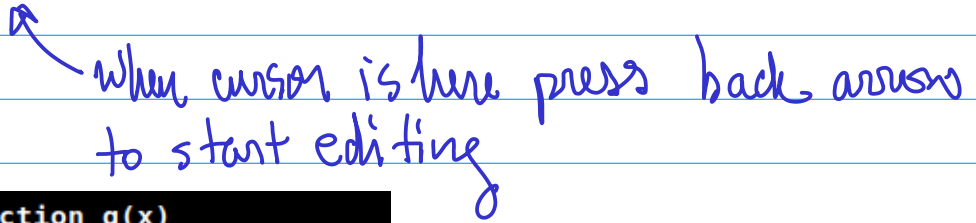
```
0.7853981633974483
x=0.46635291863256073
0.46635291863256073
x=0.45023140545751744
0.45023140545751744
x=0.45018361171715576
0.45018361171715576
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
0.45018361129487355
x=0.45018361129487355
```

It's double printing each number because there is also a println in the function g(x)

Press up arrow until you get back to the definition of g(x) in the command history...

```
julia> function g(x)
           println(x)
           return x-f(x)/df(x)
       end
```

Now press back arrow to move the cursor into the block of code




```
julia> function g(x)
           println(x)
           return x-f(x)/df(x)
       end
```

cursor positioned ready to delete the extra println...

```
julia> function g(x)
           return x-f(x)/df(x)
       end
```

After deleting the extra println using the backspace key press return to evaluate the new definition of g(x)

Now press up arrow to get back to the loop and press return again to execute it...

```
julia> x=x0
       for n=1:10
           x=g(x)
           println("x=$x")
       end
x=0.46635291863256073
x=0.45023140545751744
x=0.45018361171715576
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
```

note: actually converged to full precision in only 4 iterations!!

Julia also has the ability to work with extended precision arithmetic, similar to Python. Simply change the definition of x0 to be a big number. Then the change is dynamically propagated throughout the program...

```
julia> x0=big(pi)/4
0.78539816339744830961566084581987572104929234984377645524373614807695410157154 95

julia> typeof(x0)
BigFloat
```

The type is no longer Float64, but instead a number with lots of digits.

It is possible to adjust how many digits the big numbers used...up to hundreds, thousands and hundreds of thousands...one won't set any world records for computing the number of digits in pi, but one can definitely obtain far more digits than anyone would ever want...

Press up arrow to run the loop again...

```
julia> x=x0
       for n=1:10
           x=g(x)
           println("x=$x")
       end
x=0.46635291863256070164223672823283818362819460651399497180747444734038975027177 11
x=0.45023140545751743334099168023586906365565231778848834812450621107834382362028 45
x=0.45018361171715572723509343405604262651883494147061001922745186068554813870241 82
x=0.45018361129487357306950519613642146551953111318608243601136748723118723795045 19
x=0.45018361129487357303653869676268182732033741703795518835505368888035722388983 67
x=0.45018361129487357303653869676268182732013650172305543401505849136364156698522 17
x=0.45018361129487357303653869676268182732013650172305543401505849136364156698522 17
x=0.45018361129487357303653869676268182732013650172305543401505849136364156698522 17
x=0.45018361129487357303653869676268182732013650172305543401505849136364156698522 17
x=0.45018361129487357303653869676268182732013650172305543401505849136364156698522 17
```

The fact that only two more iterations for a total of six are needed to obtain all the digits illustrates how fast Newton's method converges...

The number os significant digits doubles at each iteration...

If you exit Julia, the workspace is cleared...Let's copy everything with the mouse into an editor for safe keeping...

Note, we actually did this little bit at a time during class, but you can always scroll back through the terminal session or use the up arrow key to review all the commands you've typed...

When copying text from the terminal window use ctrl+shift+c to put things in the mouse...

When pasting text into the editor using ctrl+v

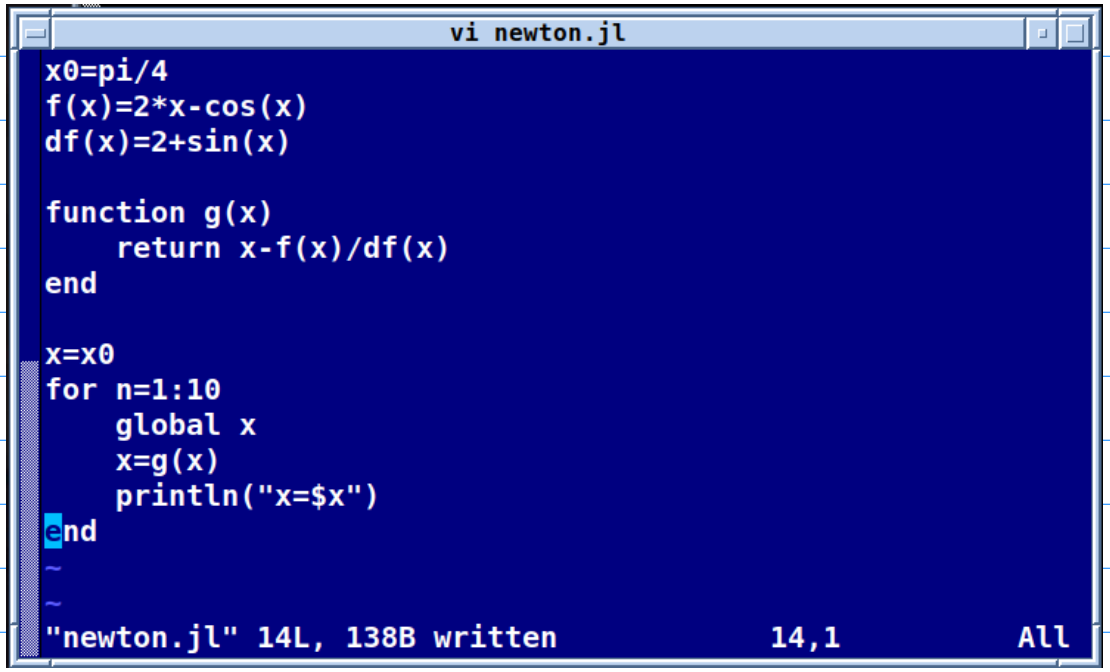
note the extra shift needed for the terminal

It's often easier to work the other way...typing everything into the editor to begin with and then pasting things into Julia as needed.

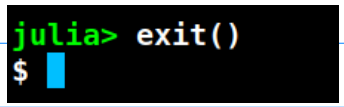
When copying text from the editor use ctrl+c

When pasting text into the Julia terminal use ctrl+shift+v

Here is the file from the editor...this file is also available by clicking on the link from the class webpage...

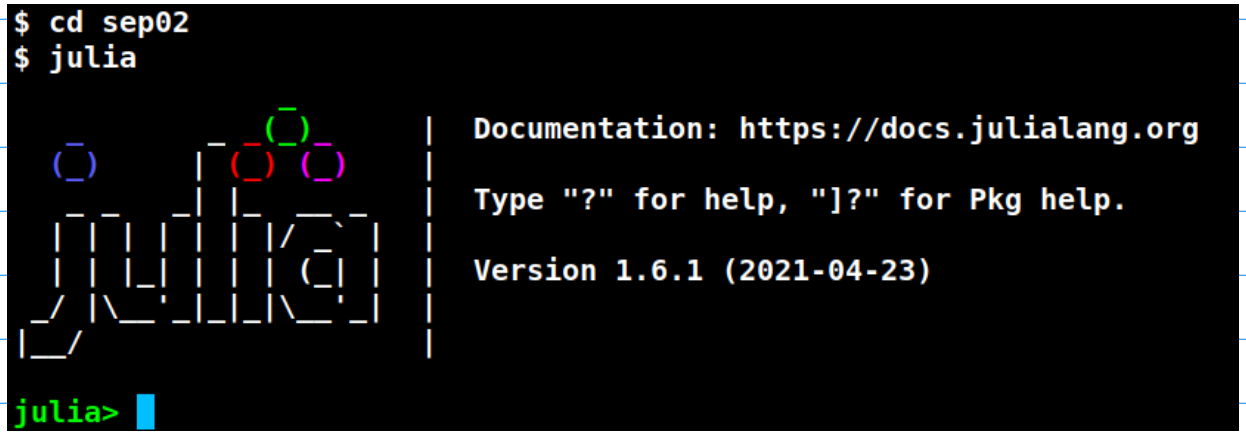


Save the file newton.jl in a folder called sep02 to keep our work for the class organized...note that since we are using the guest account in the computer lab today, all our work will be lost when the computer reboots, but never mind...I'll post it on the web...



Use exit() to quit Julia... you can also type ctrl-d to exit...

Now, lets change to the sep02 folder and run Julia again...



To load the file newton.jl type include...

```
julia> include("newton.jl")
```

press return to run the script...

```
julia> include("newton.jl")
┌ Warning: Assignment to `x` in soft scope is ambiguous because a global variable by the
same name exists: `x` will be treated as a new local. Disambiguate by using `local x` to
suppress this warning or `global x` to assign to the existing global variable.
└ @ ~/teach/466/2021/sep02/newton.jl:11
ERROR: LoadError: UndefVarError: x not defined
Stacktrace:
 [1] top-level scope
   @ ~/teach/466/2021/sep02/newton.jl:11
 [2] include(fname::String)
   @ Base.MainInclude ./client.jl:444
 [3] top-level scope
   @ REPL[1]:1
in expression starting at /x/libb/ejolson/teach/466/2021/sep02/newton.jl:10

julia>
```

Oh no! There is an error... And it fills the screen...

Remember, that Julia includes a just-in-time compiler running behind the scenes to make things run fast... if all those turning wheels and gears get stuck and this is what happens...the verbosity of the error message and stack trace is irritating... still, it runs fast when it works, and Julia is generally easier than writing Fortan...

In this case the error is related to differences between running a script from the REPL compared to from a file... All that stuff about soft scope is related to x being used inside a for loop that is not inside a function...

These problems go away if one uses Julia like a programming language and writes a main function such as one might in Java or C...

Never mind that... right now it is more convenient to work with scripts...

The problem can be fixed by adding global x inside the loop to resolve the ambiguity that happens with the soft scope...whatever that is...

The new version of newton.jl look like

vi newton.jl

```
x0=pi/4
f(x)=2*x-cos(x)
df(x)=2+sin(x)

function g(x)
    return x-f(x)/df(x)
end

x=x0
for n=1:10
    global x
    x=g(x)
    println("x=$x")
end
~
~
-- INSERT --                                  11,13         All
```

global added to resolve the soft scope ambiguity in the script...

Now it's possible to run the script from Julia...

```
julia> include("newton.jl")
x=0.46635291863256073
x=0.45023140545751744
x=0.45018361171715576
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
x=0.45018361129487355
```

☑ Newton finished

Rounding error

Example: 3-sig. digit rounding in decimal
(53-sig. bit rounding in binary)

$$\begin{array}{r} 6.12 \\ +\ 5.32 \\ \hline 11.44 \end{array} \longrightarrow \text{round to 3-sig digits} \longrightarrow \begin{array}{r} 11.4 \\ +\ 1.34 \\ \hline 12.74 \end{array} \longrightarrow \text{round} \longrightarrow 12.7$$

$$\begin{array}{r} 1.34 \\ +\ 5.32 \\ \hline 6.66 \end{array} \longrightarrow \text{nothing has} \qquad \begin{array}{r} 6.66 \\ 6.12 \\ \hline 12.78 \end{array} \longrightarrow \text{round} \longrightarrow 12.8$$

Depending on the order in which the number are added one gets a different answer...thus addition no longer satisfies the usual rules of algebra once rounding is taken into account...

General rule: Add the smaller numbers first to get the more accurate answer.

Let's try to make an example on the computer...

```
julia> a=1/3
0.3333333333333333

julia> b=1/4
0.25

julia> c=1/5
0.2

julia> (a+b)+c==a+(b+c)
false
```

test whether addition is associative on the computer.

means the two expressions were not equal.

That was easy...maybe it never works...

Maybe a=1/3 was the problem... try changing it...

```
julia> a=1/2
0.5

julia> (a+b)+c==a+(b+c)
true
```

So sometimes addition is associative...sometimes not...