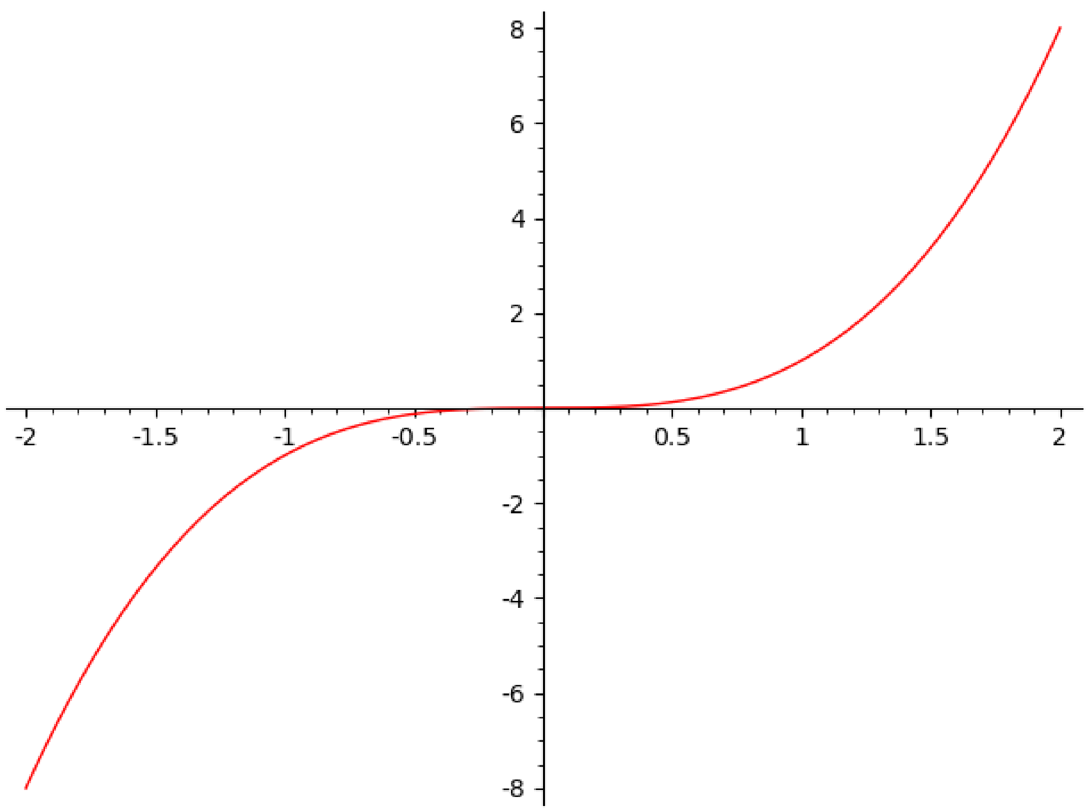
MATH1110H-B-lab-F02-2023-10-03

October 3, 2023

[1]:
```
# MATH 1110H-B F02 Lab 2023-10-23
#
# Wherein we add a couple of tricks to our knowledge of the plot command,
# learn how to solve equations,  and get started on solving differential
# equations too.
#
# First new plotting trick:  chage the colour of the graph to something
# other than the default blue:
#
plot(x^3,-2,2,color='red')
```

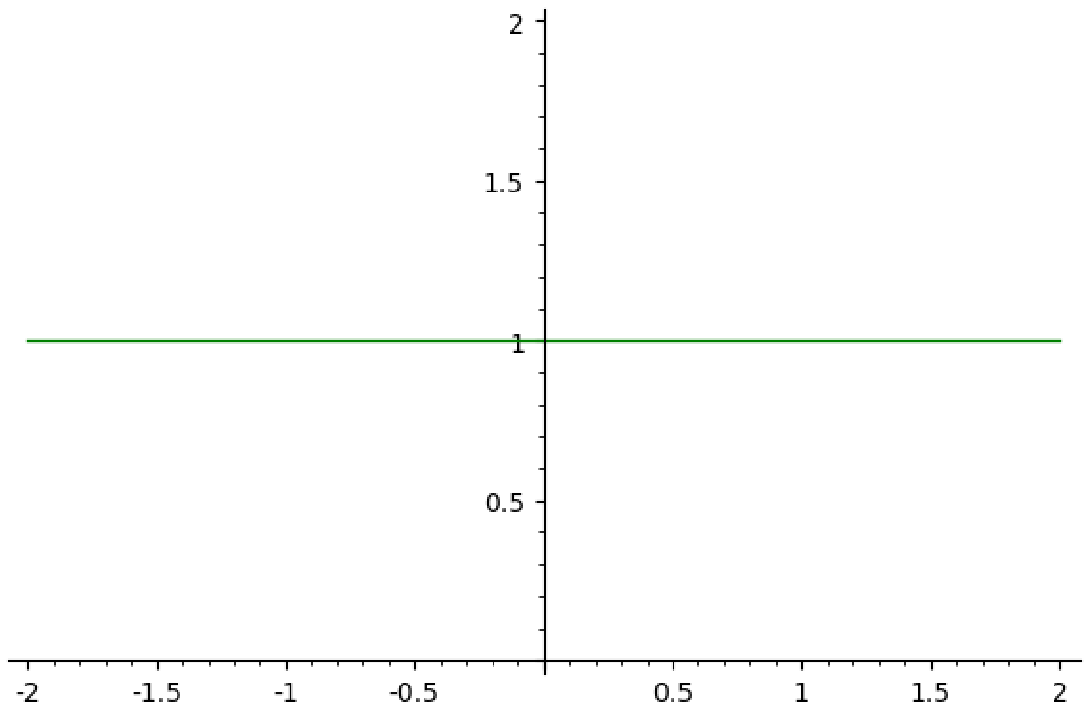
[1]:



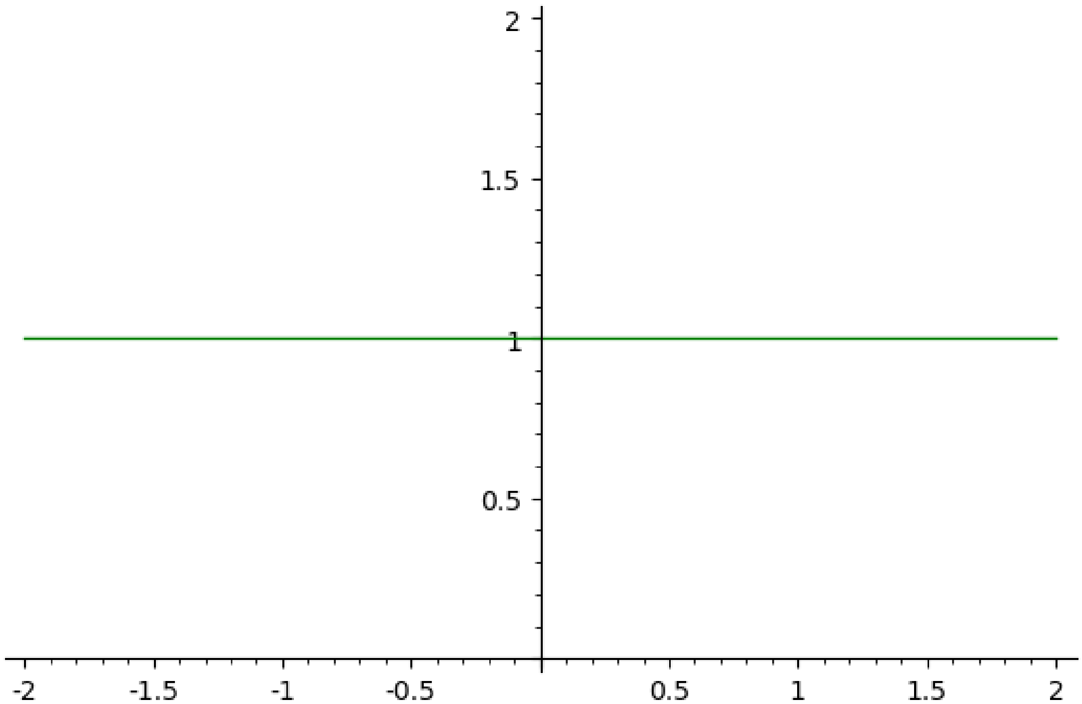
[2]:
```
p1 = plot(x^0,-2,2,color='green')  # Second trick:  give a plot a name,
```

[3]:
```
p1  # and then display it directly
```

[3]:



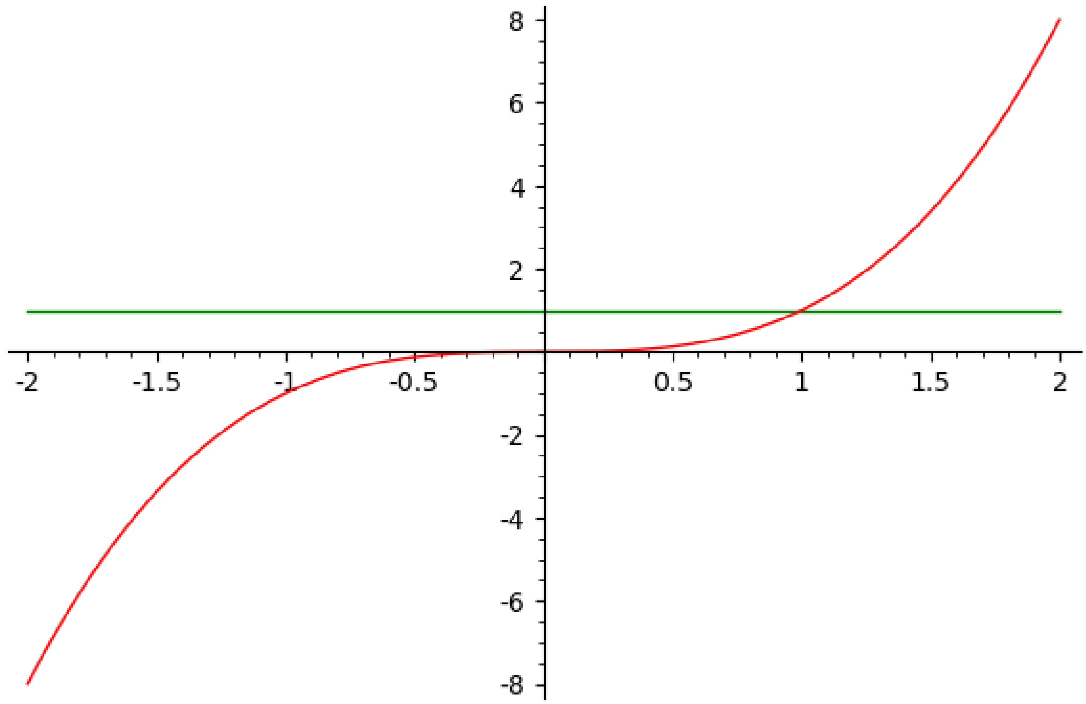
[4]:
```
show(p1)  # or by using the show command.
```



[5]:
```
p2 = plot(x^3,-2,2,color='red')  # We give another plot a name.
```

[6]:
```
p1 + p2  # Third trick:  add the plots to superimpose them:
```

[6]:



[7]:
```
solve( x^2 == 2, x )  # The solve command lets us find solutions
                      # to equations,  but you must specify the
                      # variable to be solved for,  even if it is
                      # the only one in the equation.
```

[7]: [x == -sqrt(2), x == sqrt(2)]

[8]:
```
solve( x^2 == -2, x )  # The solve command will find complex-valued
                       # solutions,  too.  Note that I is used to
                       # represent the square root of -1.
```

[8]: [x == -I*sqrt(2), x == I*sqrt(2)]

[9]:
```
solve( sqrt(x) == x, x )  # One weakness of the solve command is that
                          # will give you a lazy and useless solution
                          # if it can find an x by itself in an
                          # equation which isn't polynomial.
```

[9]: [x == sqrt(x)]

[10]:
```
solve ( x == x^(1/2), x )  # Writing a square root as a fractional
                           # power doesn't help...
```

[10]: [x == sqrt(x)]

[11]:
```
solve( x == x^2, x )  # ... but putting in a bit of effort yourself
                      # to rewrite the eqution to eliminate that
                      # fractional power lets SageMath take it the
                      # rest of the way.
```

[11]: [x == 0, x == 1]

[12]:
```
solve( cos(x) == 5, x )  # You acn use the solve command to try to
                         # where a function takes on certain values,
                         # but the symbolic answers don't always make
                         # sense.  In this example 5 is not in the
                         # domain of arccos.  (Its domain is [-1.1].)
```

[12]: [x == arccos(5)]

[13]:
```
N(arccos(5))  # Using the N command,  which tries to find a decimal
              # approximation,  makes the problem above apparent:
              # the result NaN means "Not a Number".
```

[13]: NaN

[14]:
```
N(sin(1/2))  # N can be used to get decimal answers like a calculator.
```

[14]: 0.479425538604203

[15]:
```
var("y")  # We need to declare y to be variable before we can use it.
solve(sinh(y) == x, y )  # Here we try to invert sinh using the solve
                         # command.
```

[15]: [y == arcsinh(x)]

[16]:
```
solve( x == (e^y - e^(-y))/2, y )  # To actually get an expression
                                   # for arcsinh,  we need to start
                                   # with the definition of sinh.
# Note that the first expression given for arcsinh makes no sense
# for any real number x because x - sqrt(x^2 - 1) < 0 for all real x
# and logatrithms are only defined for positive real numbers.
```

[16]: [y == log(x - sqrt(x^2 + 1)), y == log(x + sqrt(x^2 + 1))]

[17]:
```
y = function('y')(x)  # This is how to declare y to be an unspecified
                      # function of x so it can be differentiated.
desolve( diff(y,x) == 7*x, y )  # We can now write diff(y,x) for the
                                # derivative of y with respect to x,
# and use this to set up a differential equation that the desolve
# command will try to solve for y.  The desolve command is optimized
# for dealing with differential equations, which the basic solve
```

```
# command is not,  but also needs to be told which "variable" is to
# be solved for.  Note that the answer is given up to a generic
# constant _C since there is not enough information to pin it down
# any further.
```

[17]: 7/2*x^2 + _C

[18]:
```
desolve( diff(y,x) == 7*x, y, ics=[-7,18] )  # Such additional
                                             # information is often
# supplied by specifying "initial conditions" that the solution to
# the given initial condition is to satisfy.  In this case,  the
# clause ics=[0.1] specifies that when x = 0,  we should have y = 1.
# This pins down the generic constant to a particular value.
```

[18]: 7/2*x^2 - 307/2

[19]:
```
# One thing I forgot to do in this lab is show how to use SageMath
# to compute limits.  The limit of sin(x^2) as x approaches 13,  for
# example,  can be computed as follows using the lim command:
#
lim( sin(x^2), x=13 )
```

[19]: sin(169)

[20]:
```
N(sin(169))  # The N command gives us a probably more useful number.
```

[20]: -0.601999867677605

[21]:
```
lim( 1/sinh(x), x=oo ) # The lim command can also be used to compute
                       # limits as x goes to infinity or -infinity.
                       # Note the use of a double lower-case  o,
                       # that is oo,  to represent infinity.
```

[21]: 0

[22]:
```
# That's all for now!
```