Introduction

This is the Mathematica Tutorial that we have peer leaders (undergraduate students who took the course the previous semester) present during the first three weeks to the students as an introduction to programming in Mathematica and examples of tools commonly used within Wolfram demonstrations. At the beginning of the semester, the students receive the blank tutorial called "Mathematica_Tutorial.nb" and complete the examples by following the peer leaders and receiving assistance from peer leaders or graduate student instructors when necessary. Once the tutorial has been completed, we give the students the "Mathematica_Tutorial-completed.nb" file (this file), which has all the examples completed along with the answers to the challenges. The "Mathematica_Tutorial-completed&annotated.pdf" file is the file the peer leaders use when presenting the tutorial, as it includes instructions on how to present certain topics and examples.

More information about Compute-to-Learn can be found at our website: http://umich.edu/~pchem/about.html.

License

Usually GSIs (TAs) go over the Curriculum, Grading, and Tutorial Guidelines sections

Curriculum

- **Week 1**
  - Get to know your peers, peer leaders, and GSIs.
  - Learn about the expectations of the course.
  - Read through the prompt options, and choose a project for the semester.
  - Begin *Mathematica* tutorial.
  - HW: Begin collecting information and studying your prompt.
  - HW: Type one page of notes about your topic
  - HW: Get your Mathematica trial!
- **Week 2**
  - Continue *Mathematica* tutorial.
  - Have a conversation with a peer leader and GSI regarding your prompt.
  - HW: Take home exercise (complete manipulation of beaker).
  - HW: Continue collecting information about your prompt.
- **Week 3**
  - Complete *Mathematica* tutorial.
  - HW: Continue studying your prompt.
  - HW: "Storyboard" what your Demo will look like. Draw pictures and describe in words how you think you could organize your code. See the expectations worksheet to ensure you present enough material.
- **Week 4**
  - Informal presentation of storyboards.
  - **Peer Review #1:** Complete critique form 1 for each peer. Have a conversation with the peer leader and GSI regarding your ideas and how your project can be implemented.
  - **Pizza party!!!!** while requesting *Mathematica* Licenses if not done already.
  - HW: Continue gathering resources regarding your topic of interest, and begin exploring and understanding the material. Attempt to code some graphs or graphics associated with your prompt.
- **Week 5-8**
  - Coding.

- Meet with GSI and peer leader to discuss your progress.
- HW: Continue coding. Prepare to informally present your progress to your peers in 5 minutes or less.

- **Week 9-10**
  - **Peer Review #2/#3:** Complete critique form 2 for each peer. Have a conversation with the peer leader and GSI regarding your progress, questions, and future direction.
  - Complete Coding.
  - Make sure your demo fits the requirements of the Mathematica Demonstrations Project**.**
  - Begin writing a one page explanation of your demo as well as a ~5 sentence abstract detailing your demo. The ~5 sentence abstract will be published on your demonstration page.
  - Submit your demonstration with the one-page explanation AND ~5 sentence abstract to your peer leaders.
  - HW: Complete coding and prepare demonstration page.
  - Extra Credit: If you submit your Mathematica Demonstration to the demonstrations page by week 10, you will receive a free “upgrade” to any previous grade.
- **Week 11**
  - Submit your Demonstration to the Mathematica Demonstrations Project website.
  - HW: Write a one-page homework assignment with multiple questions. These questions must require use of your demonstration as a *tool* to answer the questions.
- **Week 12**
  - Make edits based on review given by the Wolfram Demonstrations team.
  - HW: Get ready for formal presentation session.
- **Week 13**
  - Formal presentation session and pizza party to celebrate everyone’s accomplishments!

You can go over the Grading section fairly quickly. The main things to hit are the three assignments at the beginning (the page of notes, Take-Home assignment, and Storyboard) and the emphasis on attendance and getting approval at least 24 hours in advance. This can also be a good spot to mention that the expectation is that you will have a maximum of 2-3 hours of outside work each week, with the possibility of much less if the studio time is used efficiently in weeks 5-11.

Grading

The Grading for CHEM 230/260H largely depends on your commitment to the course, active attendance, and completion of assignments to your best ability. (For example, if you stop showing up to the studio or give minimal to no effort on multiple assignments, you will not receive an H credit). You must receive an overall average of ✓ or ✓+ to receive the H at the end of the semester.

- Assignments that will be graded on the ✓+, ✓, or ✓- scale:
    - One typed page of notes about chosen prompt (due Week 2) (worth 1 check)
    - Completion of Take-Home Assignment (Week 3) (worth 1 check)
    - Storyboard (Week 4) (worth 2 checks)
    - Attendance
        - ***You cannot miss more than one studio session without approval.*** (ಠ_ಠ)
        - If an important event comes up where you **must** miss one more session, you may get approval by the GSI at least 24 hours in advance of the studio session to be missed.
    - Overall engagement with *Mathematica* software and involvement with the peer review process. (worth ಠ_ಠand 1 check per peer review session)
        - Involvement in the peer review process includes completing the critique form for your peers on weeks 4, 9, and 10, reading the critique form given to you by your peers, and giving a written description of the changes you made to your demo based on your peers advice (this can be written at the end or beginning of your one page explanation of your demo).
    - One-page explanation of demo (Week 9/10) (worth 1 check)
    - One-page homework assignment (Week 11) (worth 1 check)
    - Overall Final *Mathematica* Demonstration Page (worth 3 checks)

We want every individual involved in CHEM 230/260H to enjoy this course and leave with a published demo. At the end of the semester, we hope that each of you will complete a demo that can actually be used by other students to help them understand concepts they are struggling with in CHEM 230 or 260.

Main things in this section are that there are lots of resources for help with Mathematica, particularly the documentation which you can find online or in the notebook itself.

Tutorial Guidelines

- This is an **interactive** tutorial, make sure to **save** your work after adding notes to each slide so you can always come back and see your progress.
  - You can press “command+s” for Mac or “control+s” for Windows to save.
  - After each session, **remember to email your saved tutorial to yourself**. The school computers occasionally delete files so there is no guarantee the file will be there for you next week.
  - After each of the tutorial sessions, a completed tutorial through that day will be posted. This is meant to allow you to see some of the other options to completing certain examples and to check your in-studio exercises for any errors, if they cannot be found during studio. You still must be following along with the peer leaders throughout the tutorial.

- *Mathematica* has a HUGE library of options and functions to evaluate different types of problems.
  - Our intent is to give you an introduction to those options you will need to make a demo.

- Definitely continue to explore *Mathematica*’s huge library beyond what we show you.

- When you need to solve a new problem, there are many ways to access documentation and get examples of code.
  - You can always go to the Mathematica website (or even a search engine such as Google) for an answer to your problem.
    - e.g. Search “how to make a scatter plot in Mathematica?”
  - or
  - 1. From within a *Mathematica* notebook: Search for a particular built-in function name, e.g., **ListPlot**, by typing the following and evaluating the cell:

```
?ListPlot
```

ListPlot[{y_1, y_2, …}] **plots points corresponding to a list of values, assumed to correspond to x coordinates 1, 2, … .**

ListPlot[{{x_1, y_1}, {x_2, y_2}, …}] **plots a list of points with specified x and y coordinates.**

ListPlot[{$list_1$, $list_2$, …}] **plots several lists of points.** >>

Note that *Mathematica* is case sensitive! If you try the following search, the error appears because the function name is not recognized with lower case letters:

```
?listplot
```

```
Information::notfound : Symbol listplot not found. >>
```

If you want further examples, click the “>>” link, and a separate *Mathematica* documentation page will pop up.

- 2. From the web: Visit the Mathematica Documentation Center and enter a search term. You can either search for a specific built-in function name or a topic. E.g., searching for “scatter plot” will return a list of related topics (there is no built-in function called ScatterPlot).
- 3. From the web: Visit Mathematica StackExchange and enter a search term or click one of the “Explore our Questions” categories. This is a wonderful site for getting ideas and code samples for problems.
- 4. *Mathematica* has many powerful visualization tools. In addition to the Mathematica Demonstrations Project page, the Plot Themes page is a great place to get ideas and see what kinds of displays are possible.

- Also, although we may show you one way to solve one type of problem, know that there are often **multiple ways** to solve that same problem.
  - While you create your own demonstration this semester, try to think creatively about what you can display with *Mathematica*.
  - In addition, after you finish writing your code, always try to go back and see if there is a way you could have written it even better!

- The tutorial will last 3 weeks. After each week (except for today), you will have a small take home project that you can do collaborating with a group of your choice. YOU MUST DO YOUR OWN WORK AND SUBMIT YOUR OWN COPY. Your peer leaders will give you a ✓+, ✓, or ✓- based on your work.

We usually have students open the Hangman demonstration, so
they can have a fun intro to what an interactive demo looks like

Mathematica For Physical Chemistry Introductory Tutorial:

Chem 230/260 Honors Studio

★ Click on the following links to see cool examples of *Mathematica* demonstrations. At the end of the course, you will have developed a physical-chemistry-related *Mathematica* demonstration.

Mathematica Demonstration: Hangman

Mathematica Demonstration: Electron in a Nanocrystal Modeled by a Quantum Particle in a Sphere

Mathematica Demonstration: Simulating Gas Exchange in a Model of Pulmonary Fibrosis

Undergraduate peer leaders start after this section. A couple things to emphasize for the presentation of the tutorial:
- Make sure students know that they should be following along and typing the same thing as the peer leader leading the tutorial
- Go slow. Many of the students are just starting coding, so even though things may seem obvious to you now, they are going to move slower.
- That being said, you can move to the next example when it seems as if most students have gotten it. Anyone who is behind can be helped by a floater (i.e., a GSI or peer leader not giving the tutorial at the moment). You also do not need to wait for the floaters to be done talking to students to move on. This will be especially important in Week 2 where there is a lot of material to be covered.
- If you are giving the tutorial, don't worry about going over and helping students, the floaters will take care of it. It is better to have you at the front.
- Make sure to speak clearly, loudly, and with enthusiasm. This is especially important for classes where attendance is voluntary.
- IMPORTANT: Say everything as you are typing it. This is very helpful for people who are trying to follow what you are doing, particularly as the examples become more complicated. For example, if the input is "Sin [theta]", you should say "capital S sin, square bracket, theta, close square bracket".

How to use *Mathematica* input and output to solve simple arithmetic and algebra

- **Basic Math**: Addition, subtraction, multiplication, division, exponents
  - **Example 1**:
    - Place your cursor directly below the gray box at the end of these bullet points. Once your cursor turns horizontal, left click when your mouse is horizontal and use the keyboard shortcut “Alt+9” for Windows and “Cmd+9” for Mac.
    - An alternative option is once your cursor turns horizontal, right click (on a Mac laptop or with a Mac mouse, right click is “Ctrl+Click”). Hover your mouse over "Insert New Cell" and left click from the drop down menu to choose "Input".
    - Now, use the keyboard signs + , - , * , / , and ^ to show examples of addition, subtraction, multiplication, division, and exponents, respectively. Press *Shift+Enter* to compile your input.

Emphasize the keyboard shortcuts. You can also bring it up throughout the tutorial, people always forget.

As a first example, look at the text below with the white background. After typing "2+1" and then pressing Shift+Enter, the outputted gray box appeared with the number "3".

This example is already in the tutorial, so just describe

```
2 + 1
3
```

```
2 / 3
```

$\frac{2}{3}$

Shows how to get decimal values

```
2. / 3
N[2 / 3]
0.666667
0.666667
```

```
2 ^ 3
8
```

- **Variable**: Text can be set equal to values or expressions known as variables. Variables are normally assumed to be "global", meaning that each time a variable is used, *Mathematica* assumes it is the same object each time. Local variables are defined for specific programs or modules.

  Example 2: Add Z to F and then evaluate. Then repeat the previous exercise, but first assign a value to each variable.

```
Z + F
```

```
F + Z
```

```
Z = 1.5
F = 2
Z + F
```

```
1.5
```

```
2
```

```
3.5
```

Good to say this
Note out loud

Note: Think of variables as storage locations. In the examples above, the value "1.5" was stored in the location named "Z".

Example 3: Use a semicolon (**;**) at the end of the lines defining Z and F to suppress the output.

```
Z = 1.5;
F = 2;
Z + F
```

```
3.5
```

Example 4: Assignments are global and effective throughout the notebook unless cleared. Assign a value to U then add U to Z and F.

```
U = 24;
U + Z + F
```

```
27.5
```

Good to say this
Note out loud

Note: Be aware that some variable names are protected, since they correspond to built-in Mathematica functions. Assigning a value to these will produce an error:

Also, protected variables always
start with a capital letter

```
D = 7
N = 9
```

Set::wrsym : Symbol D is Protected. >>

```
7
```

Set::wrsym : Symbol N is Protected. >>

```
9
```

Also remember that Mathematica is case-sensitive, so using ***d*** *and* ***n*** *instead will work fine:*

Good to say this out loud

```
d = 7
n = 9
```

```
7
```

```
9
```

A string is really just text that you want to be text, rather than a variable

- **String**: A series of characters that form text using quotation marks.
  - **Example 1**: Type a phrase in quotation marks; then evaluate.

```
"Hello"
```

```
Hello
```

- **Function**: *Mathematica* has many built-in functions that will evaluate your mathematical problem or display your data in certain ways.
  - Functions are called by typing in the function name, followed by a pair of square brackets “[ ]”. In the square brackets are the arguments of the function, separated by commas.
  - Some examples are: **Sin[ ]**, **Cos[ ]**, or **Integrate[ ]**.
  - All built-in names in *Mathematica* (function names and constants) begin with capital letters.
  - They all require slight variations in the input syntax.
  - **Example 1**: Check that the value of **Sin[Pi]** is equal to zero.

Built-in functions always start with a capital letter and use square brackets

```
Sin[Pi]
```

```
0
```

- **Argument**: If you have a function, $F(x_1,x_2, \ldots., x_n)$, the argument is one of the n parameters or variables upon which the function depends.
  - In other words, for the single variable function $f(y)$, the variable y is the argument. Similarly, for the multivariable function $F(x_1,x_2, \ldots., x_n)$, the arguments are the n variables $x_1, x_2, \ldots, x_n$.

Sometimes functions have more than one value, i.e., argument, in the square brackets

Example 2: Determine the value of $\log_2(1024)$.

```
Log[2, 1024](* More than one argument *)
```

```
10
```

- **List**
  - A list is an ordered set of elements.
  - To specify a list, put a pair of curly braces "{ }" around the elements, separated by commas.

Lists always use curly braces

Example 1: Define a variable equal to a list of the numbers 4, 9, and 8.

```
listNum = {4, 9, 8}
```

```
{4, 9, 8}
```

- The elements of a list can be numbers, symbolic expressions, other lists, or a mixture of those.

Example 2: Define a variable equal to a trigonometric functions $\sin(\theta)$, $\cos(\theta)$, and $\tan(\theta)$.

```
listExpr = {Sin[theta], Cos[theta], Tan[theta]}
```

```
{Sin[theta], Cos[theta], Tan[theta]}
```

- A matrix is created by making a list of lists.

Example 3: Create the 3x2 matrix below by making a list of three lists.

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

```
matrix = {{a, b},
          {c, d}, (* Mention that assignments are global *)
          {e, f}}
```

```
{{a, b}, {c, 7}, {e, f}}
```

- Operations thread over lists, meaning that they operate on each value in the list successively.

Example 4: Subtract 2 from the list of numbers you created in **Example1**.

```
listNum (* to remind ourselves what listNum is *)
```

```
{4, 9, 8}
```

A single operation will apply to all elements of a list

```
listNum - 2
```

```
{2, 7, 6}
```

Example 5: Multiply the list of numbers you created in **Example1** by a list of 0, 1/3, and 1/10.

An operation with a same-sized list will perform the operation on the matching elements

```
listNum * {0, 1/3, 1/10}
```

$\left\{0, 3, \frac{4}{5}\right\}$

Example 6: Take the natural log of the list of numbers you created in **Example1**.

Functions will apply to each element of the list

```
Log[listNum]
```

```
{Log[4], Log[9], Log[8]}
```

```
N[Log[listNum]]
```

```
{1.38629, 2.19722, 2.07944}
```

- A number of functions use lists within their arguments.

Example 7: Use the built-in function **NIntegrate[ ]** to evaluate the area under the mathematical line, $y = x$, between $x = 0$ and $x = 1$. Using the same range, integrate x^2.

```
NIntegrate[x, {x, 0, 1}]
```

```
0.5
```

```
NIntegrate[x^2, {x, 0, 1}]
```

```
0.333333
```

- A number of functions return lists.
  - **Range[ ]** returns a list of numbers with specific starting and ending points, and the step size can also be specified.

Example 8: Evaluate the **Range[ ]** function three times with arguments of *10*; *5, 10*; and *5, 10, 0.5*.

```
Range[10](* Default start from 1 *)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
Range[5, 10](* Default step size 1 *)
```

```
{5, 6, 7, 8, 9, 10}
```

```
Range[5, 10, 0.5]
```

```
{5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10.}
```

- **Table[ ]** returns a list, where each element is generated using the same expression, but different values are plugged in.
- **Table[ ]** takes in an expression and a list as its arguments.

Example 9: Evaluate the **Table[ ]** function with the expression 2^i with *i* going from 0 to 10 in steps of 1.

The generic form of Table given here in the comment is also in the tutorial, so students will have it to reference

```
(* Table[expression,
        {variableName, min, max, step}] *)
Table[2^i,
      {i, 0, 10, 1}]
```

```
{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}
```

Example 10: Repeat Example9 but with j^i instead of 2^i.

```
(* Table symbolic expressions *)
Table[j^i,
      {i, 0, 10, 1}]
```

$\{1, j, j^2, j^3, j^4, j^5, j^6, j^7, j^8, j^9, j^{10}\}$

Example 11: Define a variable equal to a table with a list of expressions *i* and i^2 with *i* going from 0 to 10 in steps of 1..

When you evaluate Example 11 it will usually give a menu beneath the output that includes "display as" with the option "matrix" which, if you click it, gives the same thing as Example 12. It can be good for them to know how to do MatrixForm manually also (and I've seen some versions of Mathematica not give the menu, so they have to do it manually)

```
(* Table list *)
mat = Table[{i, i^2},
       {i, 0, 10, 1}]
```

```
{{0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16},
 {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100}}
```

Example 12: Build a pretty matrix out of the nested list we just generated.

```
MatrixForm[mat]
```

$$\begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ 5 & 25 \\ 6 & 36 \\ 7 & 49 \\ 8 & 64 \\ 9 & 81 \\ 10 & 100 \end{pmatrix}$$

- **Comment**: Used for placing notes in *Mathematica*, the text inside parentheses' and asterisks *only helps describe the input.*
  - Syntax: **(* *comment* *)**
    - A keyboard shortcut exists to quickly insert the comment syntax. Press “command” + “/” to instantly create the following syntax, “(*cursor*)” , where your cursor will immediately be set between the asterisks. This is a preferred way to insert a comment to avoid placing unwanted input into the commented section.

  Example 1: Evaluate the following input and observe the output. Is there a difference between the two functions?

The input is already there, so just need to evaluate it

```
1 + 2
1(*hello i am one *) + 2
```

```
3
```

```
3
```

Note: Comments are for the user only. The machine will completely ignore comments.

- ***Mathematica* Error Messages**

When working with *Mathematica*, there are different types of elements that you must use to execute your program. In addition, we are all prone to writing programs with mistakes (bugs).

When we get an error message from a bug, sometimes confusing jargon will show up. See the following input.

```
Table[{numbers, , numbers, 5, 15}]
```

```
{numbers, Null, numbers, 5, 15}
```

Hover your mouse over the red caret in the input to see the error message.

The error message for this example is "Too few arguments given for Table" - this is because Table requires 2 arguments and, because all the elements currently in Table are in a list, it thinks you have only given 1 argument. The correct form for this example is "Table[numbers, {numbers, 5, 15}]" which you can correct if you want or skip (since the focus is the error message)

In Studio Exercise I

The following page has four exercises in basic math. Complete them with your group.

You can give students ~5-7 min to do Exercise I and then go over the answers. You can ask students for how they did the exercises or just give the answers, though it is usually good to get some student participation at this point

Exercise I

1. Use a function to create a list of the first 20 integers cubed. (you should see in your output, “{1, 8, 27...}”)

```
Table[int^3, {int, 1, 20}]
```

```
{1, 8, 27, 64, 125, 216, 343, 512, 729, 1000,
 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000}
```

```
Range[1, 20]^3
```

```
{1, 8, 27, 64, 125, 216, 343, 512, 729, 1000,
 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000}
```

```
(* Not done in class, but this can be done with a user-
 defined function. User-defined functions will be taught during Day 2 *)
getCube[int_] := int^3
getCube /@ Range[20]
```

```
{1, 8, 27, 64, 125, 216, 343, 512, 729, 1000,
 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000}
```

2. Use basic arithmetic operations to determine if the order of operations is conserved in Mathematica.

```
5 + 2/3
```

$\frac{17}{3}$

```
(5 + 2) / 3
```

$\frac{7}{3}$

```
5 + (2/3)
```

$\frac{17}{3}$

3. Evaluate the following input and decide which formation correctly gives the output value of

sin(60°)

```
Sin[60]
```

```
Sin[60]
```

```
Sin[60 Degree]
```

$$\frac{\sqrt{3}}{2}$$

This is just to convert 60° into radians

```
Pi * 60/180
```

$$\frac{\pi}{3}$$

```
Sin[Pi/3]
```

$$\frac{\sqrt{3}}{2}$$

4. What output/value do you get when you add together “1 + Pi”? How about “1+N[Pi]”?

```
1 + Pi
```

$$1 + \pi$$

```
1 + N[Pi]
```

```
4.14159
```

Note: Similar to the variable **D**, variables **N** and **Pi** are also protected.

End Day 1 (Remember, save your work and email it to yourself!)

Recap: How is *Mathematica* Structured?

- **Input**: The text written by a human author, which is written under the “Input” format. (the In[ ] and Out[ ] will appear automatically when you evaluate a cell in *Mathematica*. The number inside the brackets is the number of evaluations that have happened in the current *Mathematica* session)
- **Output**: The computer-generated evaluation of the “Input” text.
- **Brackets** [ ]
  - Brackets are used to enclose function arguments.
  - **Example 1**: Use the *built-in Mathematica function* **Sin[ ]** to evaluate the sine of Pi.

```
Sin[π]
```

```
0
```

- **Braces** { }
  - Braces are used to enclose lists.
  - **Example 2**: Evaluate the following **List[ ]** and **Range[ ]** functions, and view the braced format of the output.

```
List[1, 2, 3, 4, 5, 6]
```

```
{1, 2, 3, 4, 5, 6}
```

```
Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

- **Parentheses** ( )
  - These are used to organize expressions and determine the order of operations. For example, view the following:

Input and Output are already there, they just need to view it

```
(3 + 2) * (4 + 9)
```

```
65
```

```
3 + (2 * 4) + 9
```

```
20
```

- **Cells**: *Mathematica* notebooks are organized using "cells", which are shown by the open-ended boxes on the far right side of the *Mathematica* window.
  - **Example 3**: Click to the right of your work and note how some cells are embedded within others.

- **Notebook**: The type of file where most *Mathematica* codes are developed or where mathematical problems can be solved.
  - The file is saved with the extension ".nb"

- **Demonstration**: A *Mathematica* file that dynamically demonstrates a concept that can be shown using *Mathematica* language, which can be published online under the Wolfram Demonstrations Project**.** A demonstration is presentable, and may include interactive elements to convey information.

User-defined functions

The Wolfram Language allows you to define functions of your own. These function definitions are essentially just assignments that provide transformation rules for specified patterns. To define functions you will use *delayed assignments*, so as to perform the defined calculation any time you call a function. In other words, delayed assignments indicate that *Mathematica* will not evaluate the function until you provide the input for the function.

Before we continue any further, we should clear the memory of the variable x to avoid any confusion with the machine.

Input is there but they need to evaluate it by putting their cursor in the box and doing Shift + Enter

```
Clear[x]
Clear["Global`*"]
(* Use Clear["Global`*"] to clear all variables and functions *)
```

- To define a function, you must first give it a name. These names are just symbols, so you should avoid starting them with capital letters to prevent confusion with built-in functions.

Good to say this part that's highlighted

- The assignment of a function involves the use of the operators **Blank** (_) and **SetDelayed** (**:=**), which will be further explained later in the tutorial.

Example 1: Define a simple function that squares an argument.

```
f[x_] := x^2
```

Example 2: Use the function to square 3.

```
f[3]
```

```
9
```

Example 3: Apply the function to an expression *x* + *1*.

Note that though 3 was put in in place of x in Example 2, that does not assign x=3. You have to actually put x=3 to assign 3 to x.

```
f[x + 1]
```

$(1+x)^2$

Example 4: Apply the function to a list of expressions: *x*, *x* + *1*, and *x* -*1*.

```
f[{x, x + 1, x - 1}]
```

$\{x^2, (1+x)^2, (-1+x)^2\}$

- What happens if you do not use the operators _ and **:=** ?

- The **Blank** (_) operator tells *Mathematica* that x is not a variable, but instead an argument of the function to be defined.

Example 5: Try defining and then evaluating a new function without _ operator. What happens?

```
g[x] := x^2
```

```
g[5]
```

```
g[5]
```

```
g[x]
```

x^2

- The **SetDelayed** (**:=**) operator is more subtle. It evaluates the RHS expression each time the function is called.
- This is for most of the time what you would want, however on very rare cases you want the RHS to be evaluated only once when the function is defined.
- Rule of thumb: Do not use “=” while defining functions unless you know exactly what you are doing.

Example 6: Evaluate the following input to see how the **SetDelayed** operator works.

Main point really is that they should always use the SetDelayed operator because you would only not use it in some very specific cases that they are unlikely to need

Input is there but they will need to evaluate each box (including the 1st)

```
randomOneTime[x_] = x * RandomReal[];(* RandomReal is called only once. *)
randomManyTimes[x_] := x * RandomReal[];(* RandomReal is called every time. *)
```

RandomReal[] returns a random number between 0 and 1

```
Table[randomOneTime[100], {i, 1, 10}](* Returns 10 identical numbers *)
```

```
{22.9856, 22.9856, 22.9856, 22.9856,
 22.9856, 22.9856, 22.9856, 22.9856, 22.9856, 22.9856}
```

```
Table[randomManyTimes[100], {i, 1, 10}](* Returns 10 different numbers *)
```

```
{17.0222, 33.2078, 40.6668, 87.9137,
 97.1809, 26.2522, 24.0268, 4.77177, 81.013, 89.5447}
```

- You can define a function to accept multiple arguments.

Example 7: Define a function to solve the Arrhenius equation, $rate = A \bullet e^{-Ea/RT}$, that accepts A, Ea, and T as arguments.

GasConstantR is already there but not the arrhenius function

```
GasConstantR = 0.008314 (* kJ / mol•K *);
arrhenius[A_, Ea_, T_] := A * Exp[-Ea / (GasConstantR * T)]
```

Example 8: Determine the rate constant for $A = 1.03*10^{\wedge}12\ s^{-1}$, Ea = 196.6 kJ/mol, T = 393 K.

```
arrhenius[1.03 * 10 ^ (12), 196.6, 393]
```

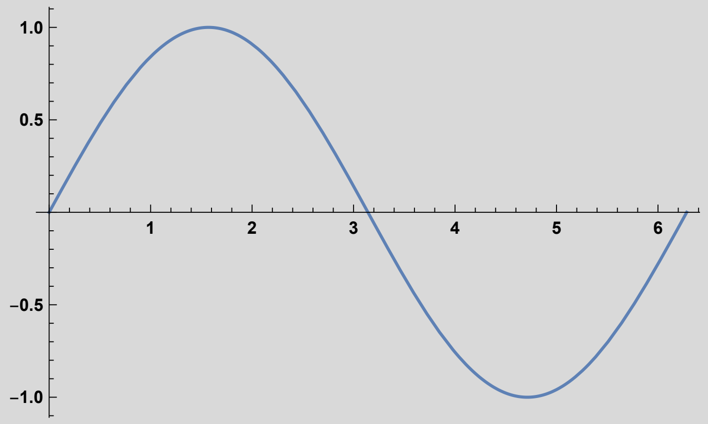
```
7.6082 × 10^-15
```

For the plots, graphics, and grid sections, you should type it as shown here to encourage good programming practices. The comments showing the structures of the input are already there so that can help guide the students. Also, you don't need to type the additional comments with the input, they are there for it the students look at the completed tutorial provided after the tutorial session.

Plots

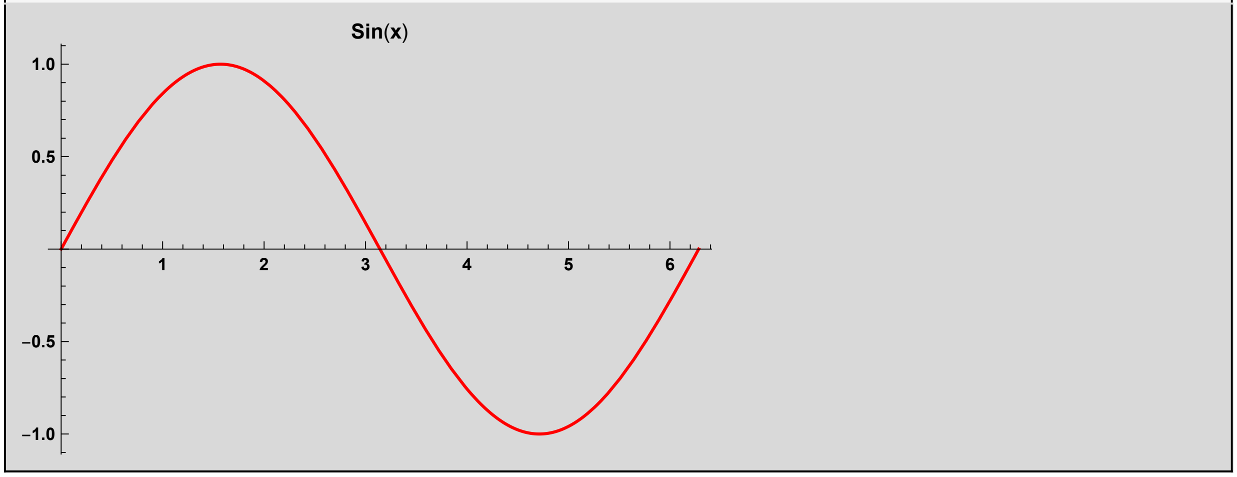
- **Plot[ ]** returns a plot object.
  - **Example 1**: Create a sine plot over the range 0 to 2π.

```
(* Plot[ expression,
        { varName, min, max}] *)
Plot[Sin[x],
     {x, 0, 2 * Pi}]
```



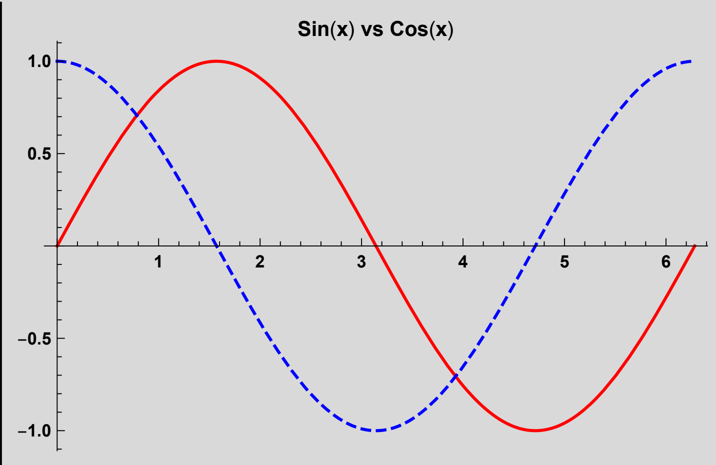
- **Options** can be added to **Plot[ ]** as additional arguments to change the behavior of the plot.
  - Options are special expressions with an arrows in them. To type an arrow, type in ->.
  - As functions require more and more elaborate arguments, it is useful to split single functions into multiple lines.
  - **Example 2**: Again **Plot Sin[x]** from zero to 2pi, but use options to set the **PlotStyle** to **Red** and add a **PlotLabel**.

```
(* Plot[ expression,
        {varName ,min ,max },
        option1,
        option2] *)
Plot[Sin[x],
    {x, 0, 2 * Pi},

    (* options *)
    PlotStyle → Red,
    PlotLabel → "Sin(x)"]
```



- To plot multiple expressions on the same plot, pass the multiple expressions in a list. Keep in mind that functions like **Plot[ ]** thread through lists.
  - The same rule goes for some options such as **PlotStyle**.
  - **Example 3**: Plot Sin[x] and Cos[x] from zero to 2*Pi. Specify a solid red line for sine and a dashed blue line for cosine, and include a label.

```
(* Plot[ {
            expression1,
            expression2,
            expression3 },
        {varName ,min ,max },
        option1,
        option2] *)

Plot[{
  (* expressions *)
  Sin[x],
  Cos[x]},

 (* range *)
 {x, 0, 2 * Pi},

 (* options *)
 PlotStyle → {Red,
             {Dashed, Blue}},
 PlotLabel → "Sin(x) vs Cos(x)"]
```



- Remember, if you are unsure of the syntax for a function, you can either evaluate the function with a preceding “?” or simply hover your cursor over the function name and click the “i” icon.

```
?Plot
```

Plot[f, {x, x_{min}, x_{max}}] generates a plot of f as a function of x from x_{min} to x_{max}.

Plot[{f_1, f_2, …}, {x, x_{min}, x_{max}}] plots several functions f_i.

Plot[{…, $w[f_i]$, …}, …] plots f_i with features defined by the symbolic wrapper w.

Plot[…, {x} ∈ reg] takes the variable x to be in the geometric region reg. >>

Graphics

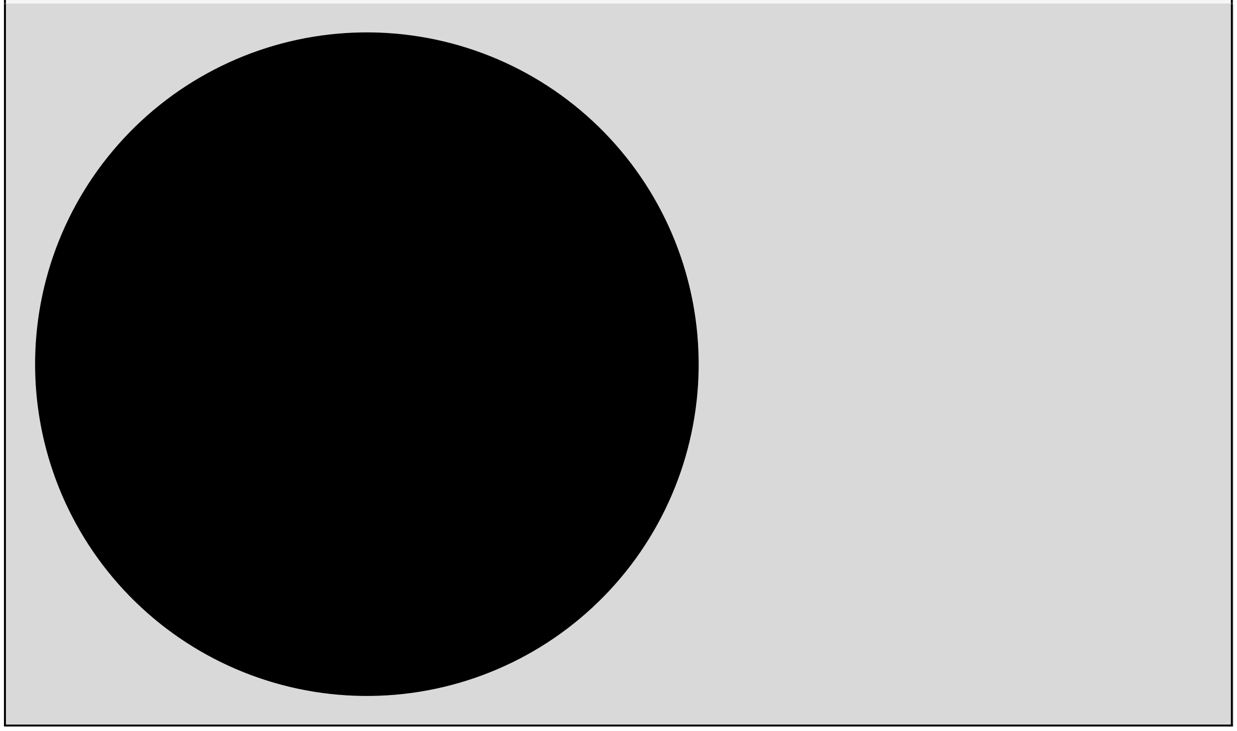
Mathematica Graphics can either be imported from external files or created via built-in *Mathematica* functions. Most *Mathematica* graphics come in 2D or 3D.

- **Graphics[ ]** Is the function that returns a 2D graphic. To tell it what to draw a shape is provided as its argument.
  - Some 2D shapes in *Mathematica*.
    - **Graphics[Disk[{*x* , *y* }, *r*]]**
    - **Graphics[Circle[{*x* , *y* }, *r*]]**
    - **Graphics[Triangle[{p_1, p_2, p_3}]]**

Example 1: Draw a filled-in **Disk** at the origin.

{0, 0} is the origin and 1 is the radius

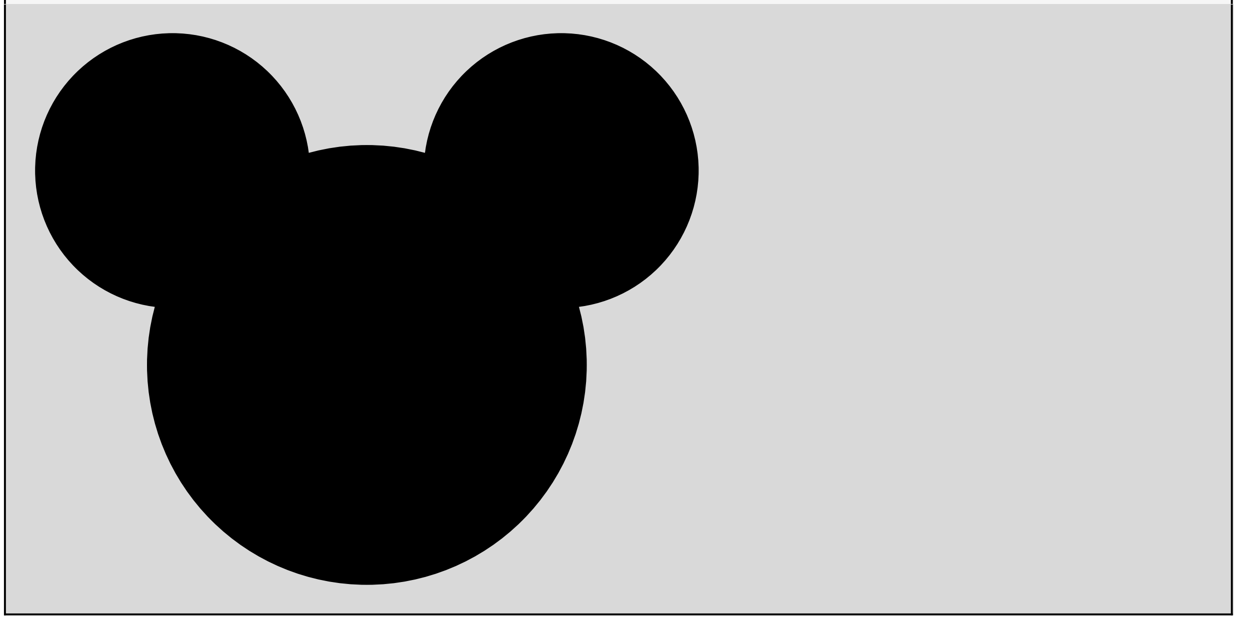
```
Graphics[Disk[{0, 0}, 1]]
```



- To draw multiple shapes at the same time, include all the shapes in a list, akin to how multiple expressions are included in a plot.

Example 2: Draw a Mickey Mouse.

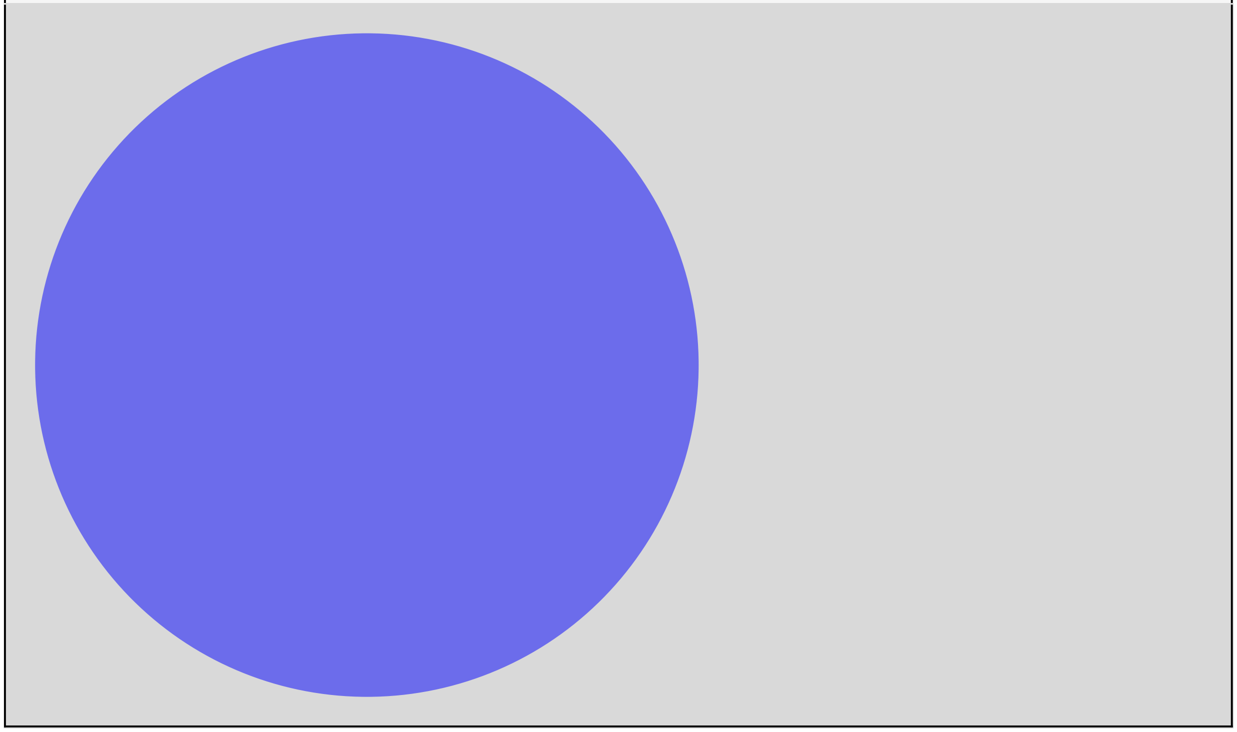
```
(*
Graphics[{
  Disk1,
  Disk2,
  Disk3
 }]
*)
Graphics[{
  Disk[{0, 0}, 0.8],
  Disk[{Sqrt[0.5], Sqrt[0.5]}, 0.5],
  Disk[{-Sqrt[0.5], Sqrt[0.5]}, 0.5]
 }]
```



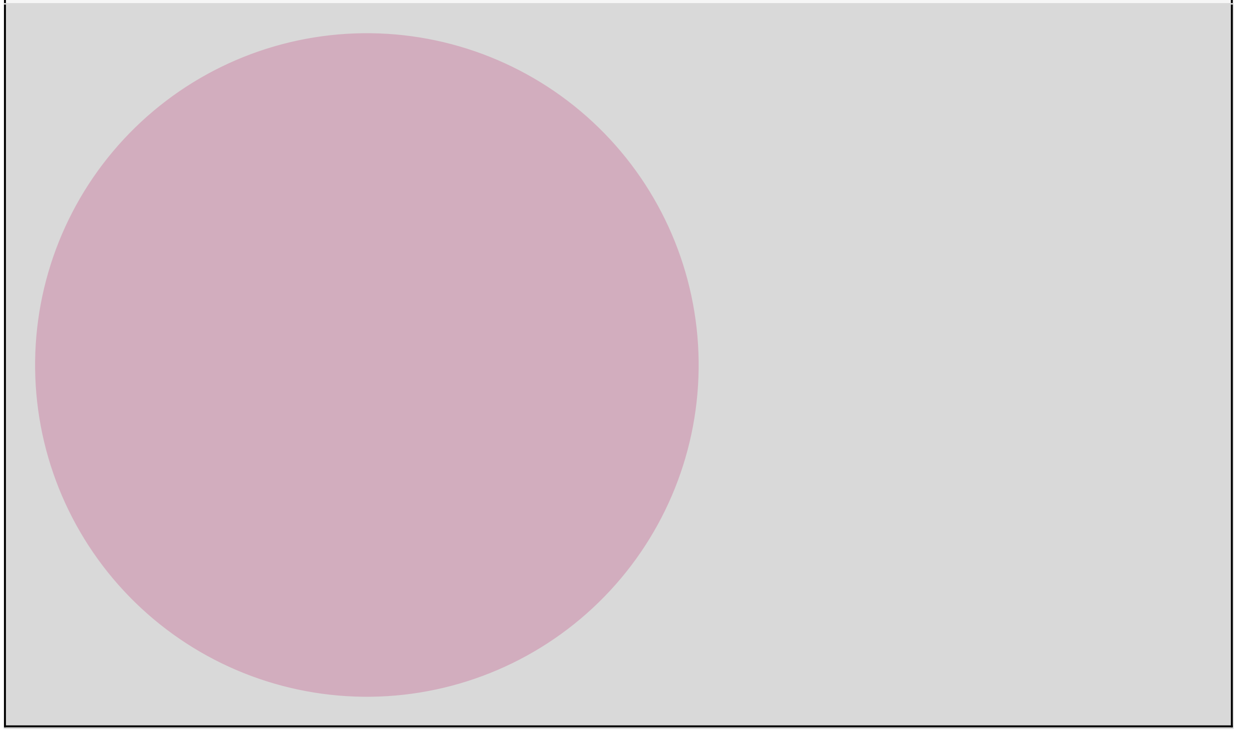
- To change the shape, transparency, border, etc. of the shapes, directives can be added as additional elements of the list.
  - Directives are specific expressions such as Red or Opacity[0.5].
  - *Directives have to be specified before the shape! This is unlike options, which are specified AFTER the main variables of the function.*

Example 3: We will change the color and opacity of the disk graphic, simply by typing a color before the shape function. You can adjust the colors and opacities to see how the graphics change.

```
Graphics[{Blue, Opacity[0.5], Disk[{0, 0}]}]
```



```
Graphics[{Opacity[0.3], RGBColor[0.77, 0.28, 0.5], Disk[{0, 0}]}]
```

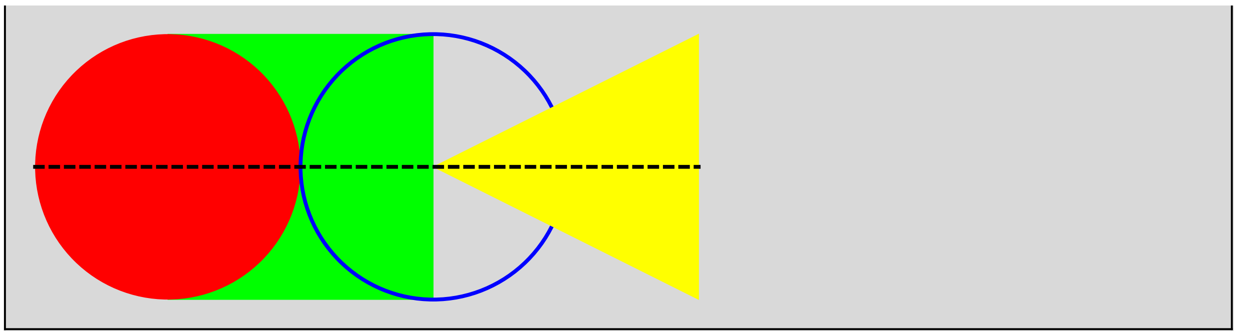


- The graphics can become very elaborate when you combine various shapes with various directives. Try to structure your code to make it easier to read.

Good to let the students know that this is a good general structure to put your graphics code in, as seen in the example directly below

```
(*
Graphics[{
  Directive,
  Directive,
  Shape1,

  Directive,
  Directive,
  Shape2,

  Directive,
  Directive,
  Shape3,

  ...
 }]
*)
Graphics[{
  Thick,
  Green,
  Rectangle[{0, -1}, {2, 1}],

  Red,
  Disk[],

  Blue,
  Circle[{2, 0}],

  Yellow,
  Polygon[{{2, 0}, {4, 1}, {4, -1}}],

  Black,
  Dashed,
  Line[{{-1, 0}, {4, 0}}]
 }]
```

We can also see the ordering of graphics in this example, as the shapes listed later are layered on top of the ones listed earlier

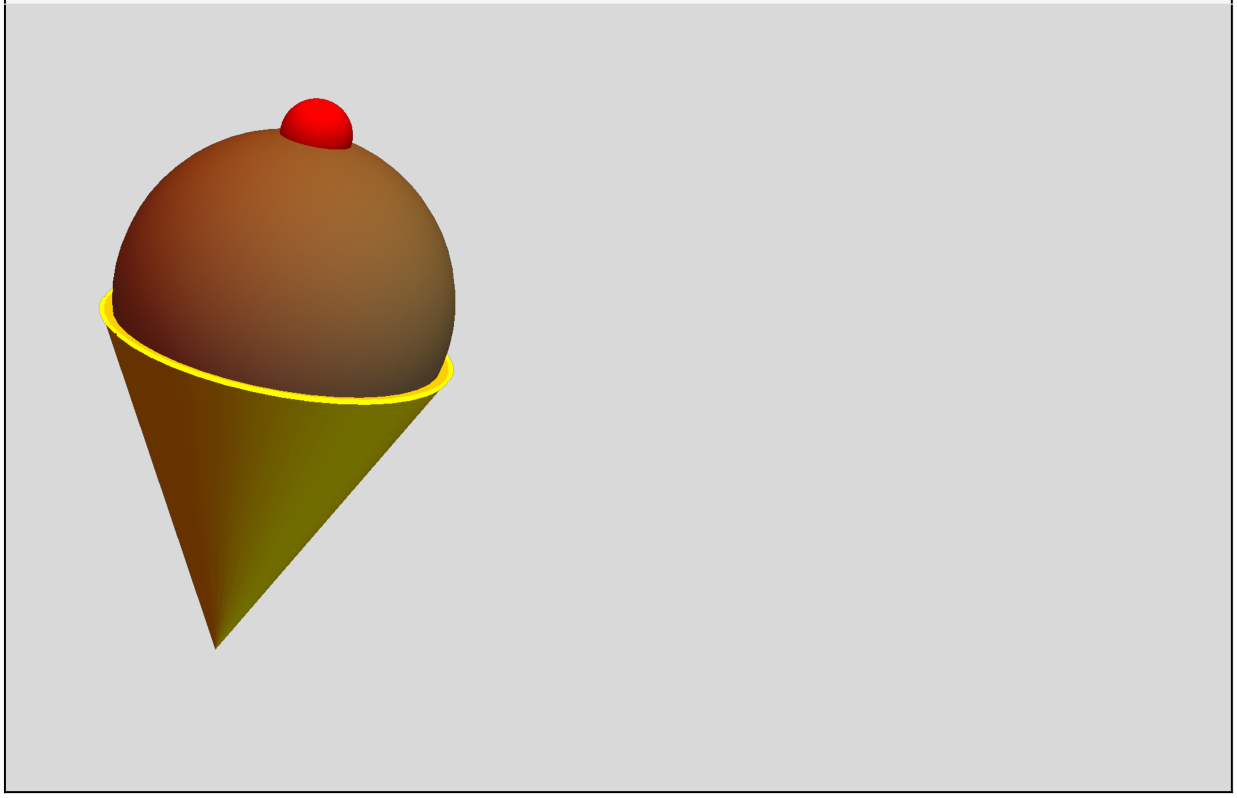


- **Graphics3D[ ]** → (*x* vs *y* vs *z* grid)
  - In a similar way to 2D graphics, we can draw 3D graphics.
  - Syntax: Same as **Graphics[ ]**, but 3D shapes such as **Sphere[ ]** and **Cone[ ]** are used.
    - Each shape has a way to specify coordinates which you should read in the documentation.
    - Many of the same directives for 2D graphics can be applied to 3D graphics.
    - You can interact with 3D graphics and rotate them with your mouse.

For Graphics[] and Graphics3D[], the students will definitely need to look at the documentation for the shape functions

Example 4: Draw a chocolate flavored ice cream in a cone from a sphere and a cone. Begin by drawing a yellow cone. Then add a brown sphere. If you want, add a cherry on top!

```
Graphics3D[{
   Yellow,
   EdgeForm[{Thick, Yellow}],
   Cone[{{0, 0, 1}, {0, 0, 0}}, 0.5],

   Brown,
   Sphere[{0, 0, 1.1}, 0.48],

   Red,
   Sphere[{0, 0, 1.58}, 0.1]
  }, Boxed → False]
```



Give them about 5-7 minutes for them to try it out, reminding them that they should look at the documentation for sphere and cone. Then when presenting the solution, tell them to stop where they are, open a new Input cell, and follow along these steps:

1. Make the Graphics3D[{
           }]
2. Make the Cone, evaluate
3. Turn the Cone Yellow, evaluate, point out the black lines on the edges, then do EdgeForm and evaluate
4. Make the ice cream Sphere, evaluate. Point out that the values in the Sphere were not random, but were chosen So that the origin was just above the cone and the radius was just smaller than the cone's. While making graphics, you'll probably have to do a fair amount of guess-and-check with the values but it is useful to start with intelligent guesses
5. Turn the Sphere Brown, evaluate.
6. Make the cherry Sphere, evaluate.
7. Turn the cherry Red, evaluate.
8. Note the box around the graphics, show how to turn off with Boxed -> False, making sure it is after the curly brace (since it is an Option)

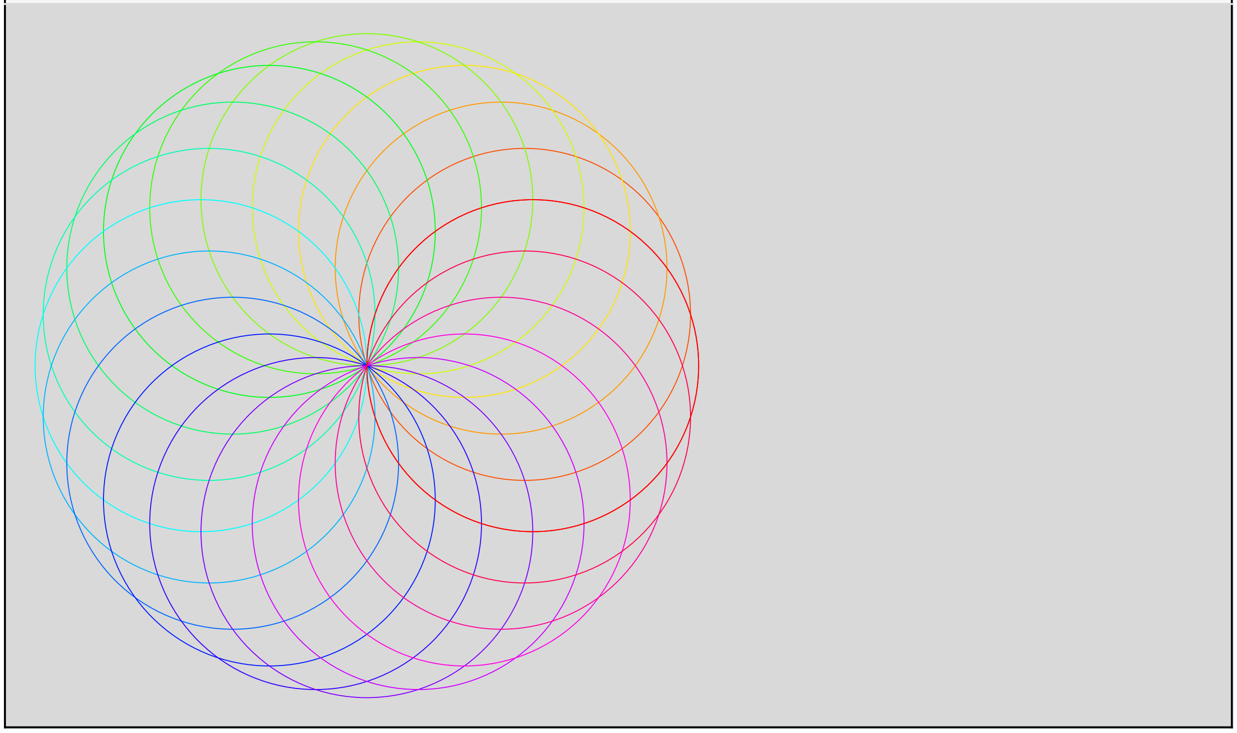
This part is just for fun or if they finish ice cream quickly, you don't have to go over it beyond saying it's something they can try out later

To look at for fun...

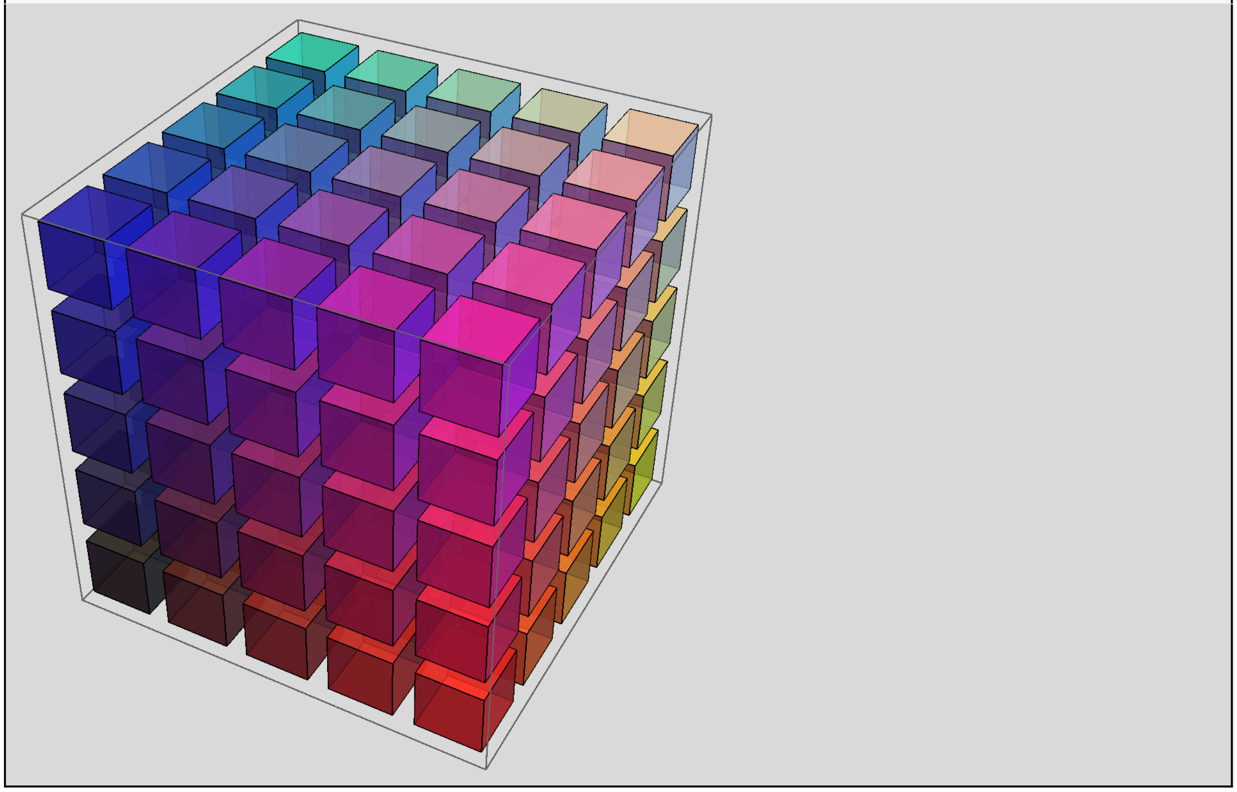
Neat Examples from the Wolfram website.

Graphics[ ] and Graphics3D[ ] takes in a list as their argument. The list can be generated by functions such as Table[ ] in stead of being typed out. This allows you to do some arcane black magic.

```
Graphics[Table[{Hue[t / 20],
   Circle[{Cos[2 Pi t / 20], Sin[2 Pi t / 20]}]}, {t, 0, 20}]]
```

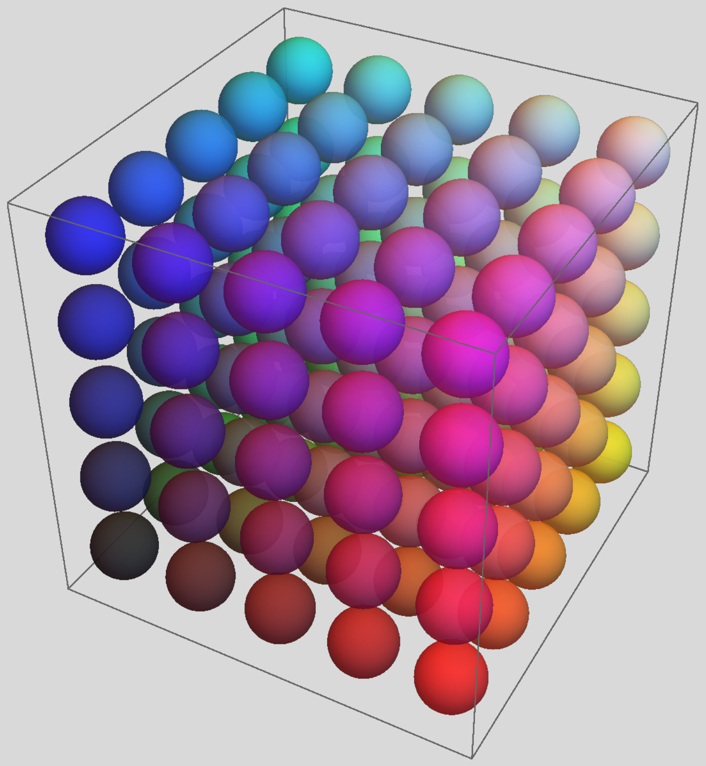


```
Graphics3D[
 Table[With[{p = {i, j, k} / 5}, {RGBColor[p], Opacity[.75],
    Cuboid[p, p + .15]}], {i, 5}, {j, 5}, {k, 5}]]
```



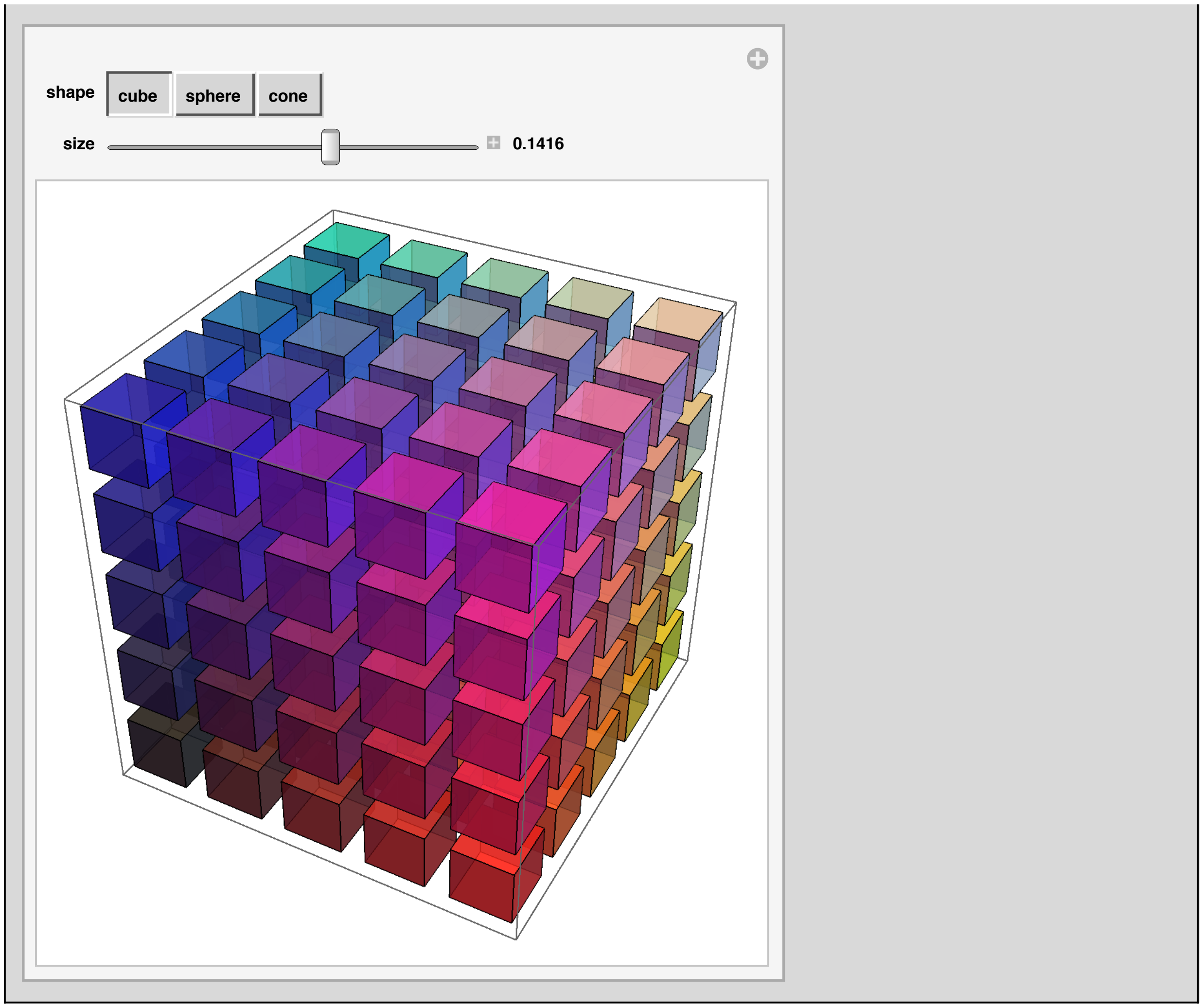
Challenge: *Can you change the picture of the “cube of rainbow cubes” into a “cube of rainbow spheres”?*
Double Challenge: *Can you change this graphic into a Mathematica Demonstration? For example, use the manipulate function as well as a setter type controller in order to show three different versions of this rainbow cube. One with a cube of rainbow cubes, another with a cube of rainbow spheres, and finally one with a cube of rainbow 3-D objects of your choice. Then add a slider to change the size of the cubes, spheres, and cones.*

```
(* Answers to Challenges below *)

Graphics3D[
 Table[With[{p = {i, j, k} / 5}, {RGBColor[p], Opacity[.75],
    Sphere[p, .15 / 2]}], {i, 5}, {j, 5}, {k, 5}]]
```



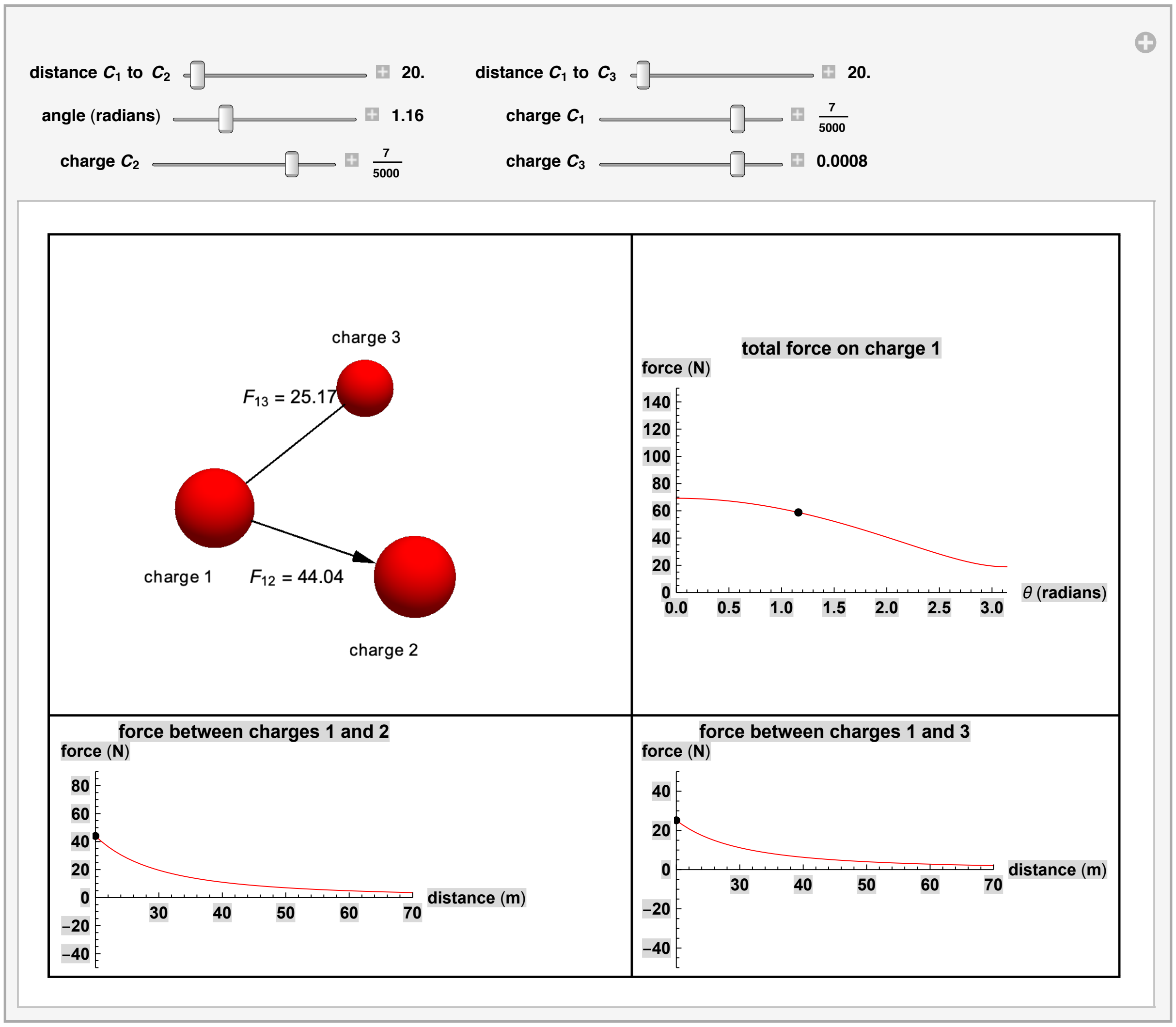
In[5]:=
```
Manipulate[Which[button == 1, Graphics3D[
    Table[With[{p = {i, j, k} / 5}, {RGBColor[p], Opacity[.75],
        Cuboid[p, p + size]}], {i, 5}, {j, 5}, {k, 5}]],
   button == 2, Graphics3D[Table[With[{p = {i, j, k} / 5},
      {RGBColor[p], Opacity[.75], Sphere[p, size / 2]}],
     {i, 5}, {j, 5}, {k, 5}]],
   button == 3, Graphics3D[Table[With[{p = {i, j, k} / 5},
      {RGBColor[p], Opacity[.75], Cone[{p, p + Sqrt[size^2 / 2]},
        size / 2]}], {i, 5}, {j, 5}, {k, 5}]]]



 ,
 Control[{{button, 1, "shape"}, {1 → "cube",
    2 → "sphere", 3 → "cone"}, ControlType → SetterBar}],
 Control[{{size, 0.1, "size"}, 0.05, 0.2, Appearance → "Labeled"}]
]
```

Out[5]=



Using Grid[ ] to organize the layout of your output

Grid is a function that formats expressions into a two dimensional grid. **Grid** can be used to organize numerical data, graphics, plots, etc.



- **Grid[ ]**
  - Its argument is a list of rows, where the rows are lists themselves.
    - Syntax: **Grid[{{a_{11}, a_{12}, a_{13}}, {a_{21}, a_{22}, a_{23}}}]**

```
(* General form of a grid, this one is 2x2 *)

(* Grid[{
  {first row first element, first row second element},
  {second row first element, second row second element}
 },
 options
] *)
```

Example 1: Use **Grid** to make a table of the following two columns and four rows of information.

```
gas pressure (atm)  gas volume (L)
        2                 5
        4                 10
        3                 7.5
```

Remind them about strings and that they will need to use them here to get the text

```
Grid[{
  {"gas pressure (atm)", "gas volume (L)"},
  {2, 5}, {4, 10}, {3, 7.5}
 }
]
```

```
gas pressure (atm)  gas volume (L)
        2                 5
        4                 10
        3                 7.5
```

Example 2: Now, make the elements to be right-aligned on the right column and left-aligned on the left column. (You can also choose some alignment/style options by left-clicking below your output and choosing from the option bar that shows up.)

Alignment is an Option so it must be after the curly brace containing the elements of the grid

```
Grid[{
  {"gas pressure (atm)", "gas volume (L)"},
  {2, 5}, {4, 10}, {3, 7.5}
 },
     Alignment → {{Left, Right}}
]
```

```
gas pressure (atm)  gas volume (L)
2                                5
4                               10
3                              7.5
```

- **Options**: Like any other *Mathematica* function, there are multiple options you can add to this function.

- **Frame→All** : Puts a frame around every element.
- **Frame→True** : Frames the entire grid.

💡 **Example 3**: Change the **Frame** to **True**, and observe what happens. Then change the **FrameStyle** to **Red**. Observe the output.

```
Grid[{
  {"gas pressure (atm)", "gas volume (L)"},
  {2, 5}, {4, 10}, {3, 7.5}
 },
     Frame → True, FrameStyle → Red
]
```

```
gas pressure (atm)  gas volume (L)
        2                 5
        4                 10
        3                 7.5
```

- **Dividers→{ *input* }** : Enables drawing lines in a variety of positions based on the parameter you choose and the ***input***.

💡 **Example 4**: For the table from **Example3**, draw a vertical solid black line between two columns and a horizontal dashed gray line between the first and second rows. Also, change the frame color back to black.

```
Grid[{
  {"gas pressure (atm)", "gas volume (L)"},
  {2, 5},
  {4, 10},
  {3, 7.5}
 },

 Frame → True,
 Dividers → {{2 → True},
             {2 → Blue, 2 → Dashed}}
]
```

```
gas pressure (atm) | gas volume (L)
- - - - - - - - - - - - - - - - - -
        2          |      5
        4          |      10
        3          |      7.5
```

In Dividers, the first argument is about the vertical lines and the second is about the horizontal lines. The vertical and horizontal lines are numbered from left to right and top to bottom, respectively, including the lines defining the edge of the grid (that's why it's 2 and 2 within Dividers)

- Nested Grids

💡 **Example 5**: Try reproducing the following.

Despite the wording, you should do Example 5 directly (not have them try it) as they will have the chance to try in Exercise V and time is tight

```
                         first row
 second row, first column | second row, second column
                         third row
```

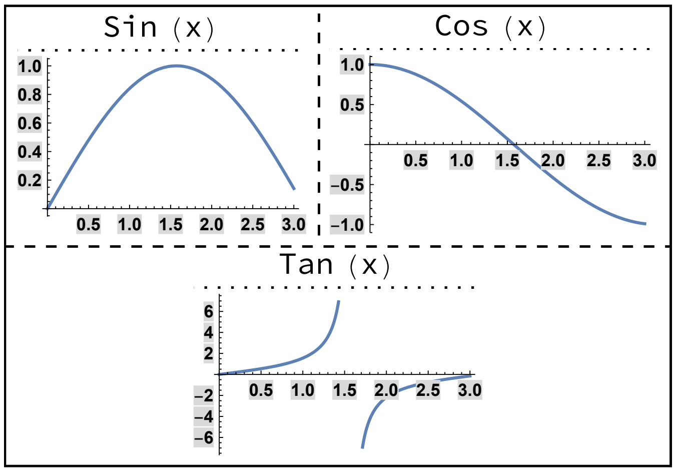
For Grids inside of Grids, it can be easier to define a variable equal to the internal Grid and place just the variable within the overall Grid.

```
secondRow = Grid[{
    {"second row, first column", "second row, second column"}
   },
   Frame → All,
   FrameStyle → Red
  ];

Grid[{
  {"first row"},
  {secondRow},
  {"third row"}
 },
 Frame → All
]
```

```
                         first row
 second row, first column | second row, second column
                         third row
```

In Studio Exercise V

Make a grid of plotted sine, cosine and tangent functions. Arrange them as seen in the example output below. Use the plot variables defined below. This will require nested grids, which may be easier to handle by defining the internal grids as variables and creating the overall grid with the variables.



This is given

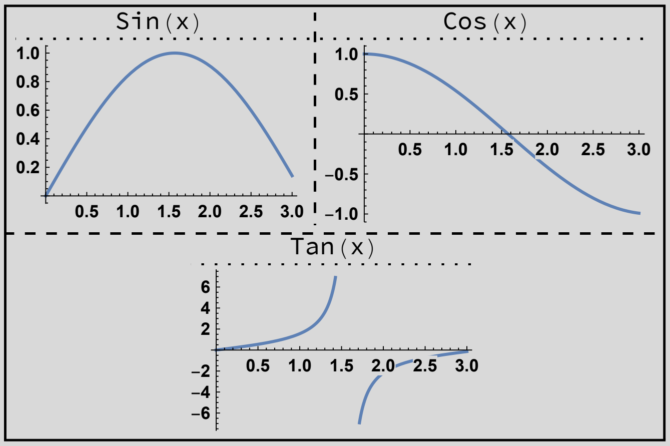
```
sinPlot = Plot[Sin[x], {x, 0, 3}, ImageSize → 150];
cosPlot = Plot[Cos[x], {x, 0, 3}, ImageSize → 170];
tanPlot = Plot[Tan[x], {x, 0, 3}, ImageSize → 150];
```

```
topGrid = Grid[
   {{"Sin(x)", "Cos(x)"},
    {sinPlot, cosPlot}},
   Dividers → {2 → Dashed(* second vertical divider *) ,
     2 → Dotted(* second horizontal divider *)}
  ];
```

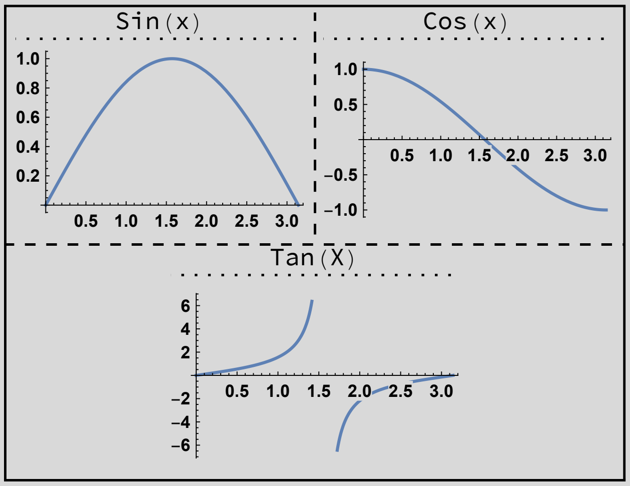
```
bottomGrid = Grid[
   {{"Tan(x)"},
    {tanPlot}},
   Dividers → {False (* no vertical dividers *),
     2 → Dotted (* second horizontal divider *)}
  ];
```

Give them about 5-7 min to try this then go over it (after reminding students to stop where they are and open a new Input box) by doing:
1. Build topGrid without the Options and evaluate
2. Add Options to topGrid and evaluate.
3. Open a new Input box, build bottomGrid without the Options, and evaluate.
4. Add Options to bottomGrid and evaluate.
5. Build the overall grid without the Options and evaluate
6. Add Options to overall grid and evaluate

```
Grid[
 {{topGrid},
  {bottomGrid}},
 Dividers →
  {True (* both vertical dividers, the right and left side of the frame*),
   2 → Dashed(* second horizontal divider *)},
 Frame → True (* complete the frame with top and bottom *)
]
```



```
(* Not done in class but is another way of tackling the problem *)
(* An additional function is defined to avoid repetitive code *)
gridPlot[title_, expr_] := Grid[
   {{title},
    {Plot[
      expr,
      {x, 0, Pi},

      ImageSize → {150, 100}]
    }},

   Dividers → {{False},
              {2 → Dotted}}
  ];

(*To make life easier,
It would be better to type in the solution in a Layer-by-layer manner*)
(* Layer 1
 Grid[
  {{top},
   {Bottom}},
   Frame→True,
   Dividers→{{False},
             {2→Dashed}}
  ]
 *)

 (* Layer 2
  Grid[
   {{Grid[
      {{gridPlot["Sin(x)",Sin[x]],gridPlot["Cos(x)",Cos[x]]}},

      Dividers→{{2→Dashed},
                {False}}
     ]},
    {gridPlot["Tan(X)",Tan[x]]}},

   Frame→True,
   Dividers→{{False},
             {2→Dashed}}
  ]
 *)

 Grid[
  {{Grid[
     {{gridPlot["Sin(x)", Sin[x]], gridPlot["Cos(x)", Cos[x]]}},

     Dividers → {{2 → Dashed},
                {False}}
    ]},
   {gridPlot["Tan(X)", Tan[x]]}},
  Dividers → {{True},
             {2 → Dashed}},
  Frame → True
 ]
```

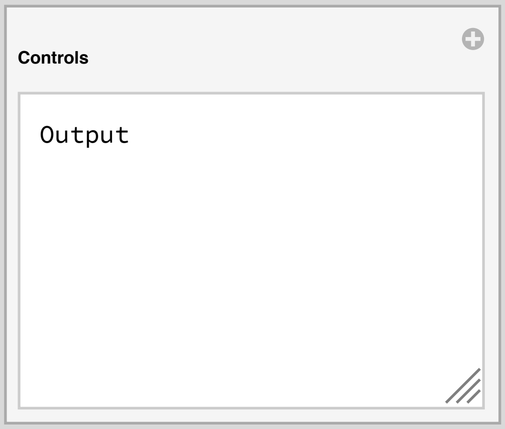

Sin(x)
Cos(x)
Tan(X)

How to make a demonstration with the **Manipulate** function

- **Manipulate**: The function **Manipulate[ ]** allows you to make the output of your code into an *interactive environment* for the user. The user can use sliders and click buttons and observe how your output changes as different variables are manually changed.
  - The Manipulate[ ] function is structured like the following:

They have the Input and Output of this; you should go over each part and show its corresponding part in the Output

```
Manipulate[
 "Line one for calculation";
 "Another line of calculation";

 "Output"
 (*Number, plot, graphics, etc. you would like to show*)
 ,

 "Controls"
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {240, 160}
 (* Add any other options here *)
]
```

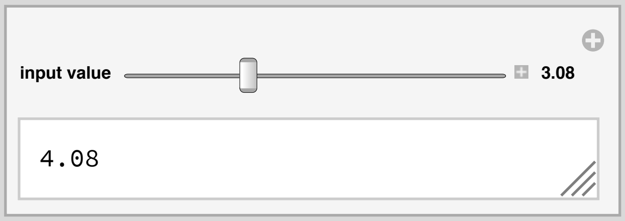


Controls

```
Output
```

  - Here is a bare-minimum of an interactive demonstration. It takes a number as input, and shows a number that is 1 larger as the output.

```
Manipulate[
 (*Calculations*)
 y = x + 1;

 (*Output*)
 y,

 (*Define variables and controls*)
 {{x, 0, "input value"}, 0, 10, Appearance → "Labeled"},
 {{y, 1}, ControlType → None},

 (*Options*)
 ContentSize → {300, 40}
]
```



```
4.08
```

Again, they have the Input and Output of this; you should go over each part
and show its corresponding part in the Outputand then move the slider

In Studio Exercise III: The making of a demo

For this exercise you will make a demonstration for the ideal gas model of O_2.
The ideal gas equation goes as follows:

$$PV = nRT$$

Where:

$$R = 0.082057338 \text{ L atm mol}^{-1} \text{K}^{-1}$$

$$n = 1 \text{ mol}$$

We can rearrange this equation to give the ideal gas pressure as a function of volume and temperature:

$$P(V, T) = \frac{nRT}{V}$$

The specs of the demo is as follows:

Input: The input would be the temperature of H_2, within the range of 100 K to 500 K.

Output: The output would be a pressure - volume plot containing the curve at the input temperature.

Range for plotted pressure:	0.1 to 75 atm
Range for plotted volume:	0.1 to 1 L

Exercise: The making of a demo

1. We start by defining the constants and parameters:

Evaluate this box

```
(*Physics constants*)
nO2 = 1.0;(*mol*)
R = 0.082057338;(*L atm mol^-1 K^-1 *)

(*Control/plot parameters*)
minV = 0.1; (* L *)
maxV = 1.0;  (* L *)
minP = 0.1;  (* atm *)
maxP = 75.0;  (* atm *)
minT = 50.0;  (* K *)
maxT = 500.0;  (* K *)

(*define functions*)
pressure[V_, T_] := nO2 * R * T / V;
```

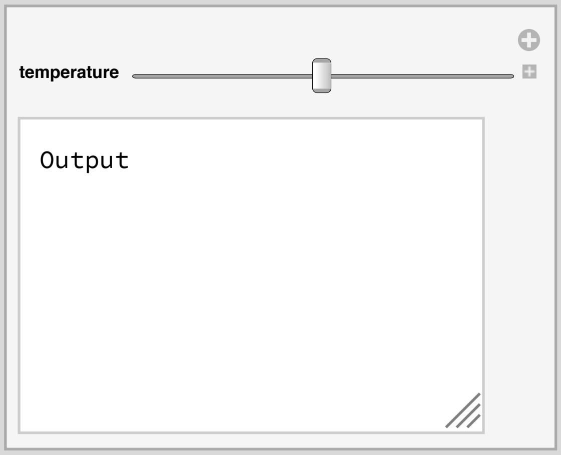
2. For the actual demonstration, we start from the skeleton:

```
Manipulate[
 "Line one for calculation";
 "Another line of calculation";

 "Output"
 (*Number, plot, graphics, etc. you would like to show*)
 ,

 "Controls"
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {240, 160}
 (* Add any other options here *)
]
```

3. Fill in the controls. For this demonstration, we only need controls for temperature:

Make the descriptive slider here (everything else is there already)

```
Manipulate[
 "Line one for calculation";
 "Another line of calculation";

 "Output"
 (*Number, plot, graphics, etc. you would like to show*)
 ,
 (*{T,minT,maxT}*) (* Basic slider *)
 Control[{{T, 298, "temperature"}, minT, maxT}](* Descriptive slider *)
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {240, 160}
 (* Add any other options here *)
]
```

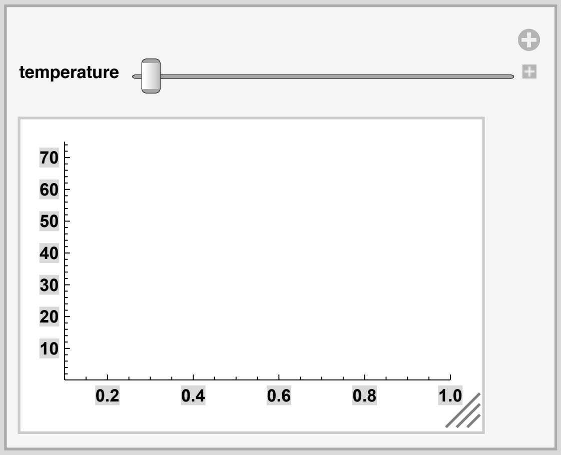


```
Output
```

4. Fill in the calculations and output. For this demonstration, our output is a plot and we do not need additional calculations.

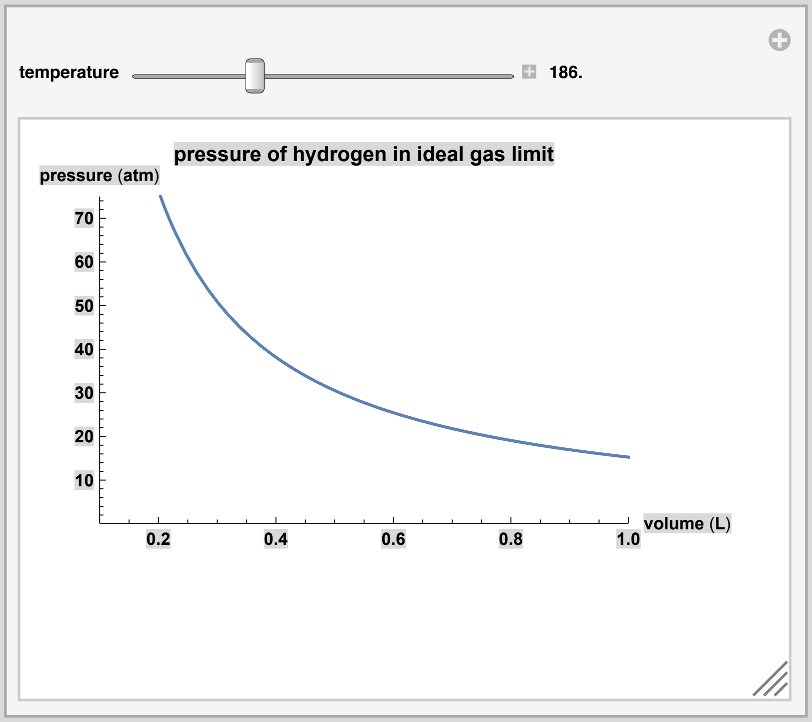
Make the Plot in this one

```
Manipulate[
 (*No calculation is needed*)

 Plot[
  pressure[V, T],
  {V, minV, maxV},

  PlotRange → {{minV, maxV}, {minP, maxP}}
 ]
 (*Number, plot, graphics, etc. you would like to show*)
 ,

 Control[{{T, 298, "temperature"}, minT, maxT}]
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {240, 160}
 (* Add any other options here *)
]
```



5. After testing the code, we should tweak the appearance of the controls and outputs to make the demonstration more user-friendly.

```
Manipulate[
 (*No calculation is needed*)

 Plot[
  pressure[V, T],
  {V, minV, maxV},

  PlotRange → {{minV, maxV}, {minP, maxP}},
  PlotLabel → "pressure of hydrogen in ideal gas limit",
  AxesLabel → {"volume (L)", "pressure (atm)"}
 ]
 (*Number, plot, graphics, etc. you would like to show*)
 ,

 Control[{{T, 298, "temperature"}, minT, maxT, Appearance → "Labeled"}]
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {400, 300},
 SaveDefinitions → True (*Important!*)
 (* Add any other options here *)
]
```



6. Science check: We've created a nice looking demo showing the ideal gas limit with oxy-

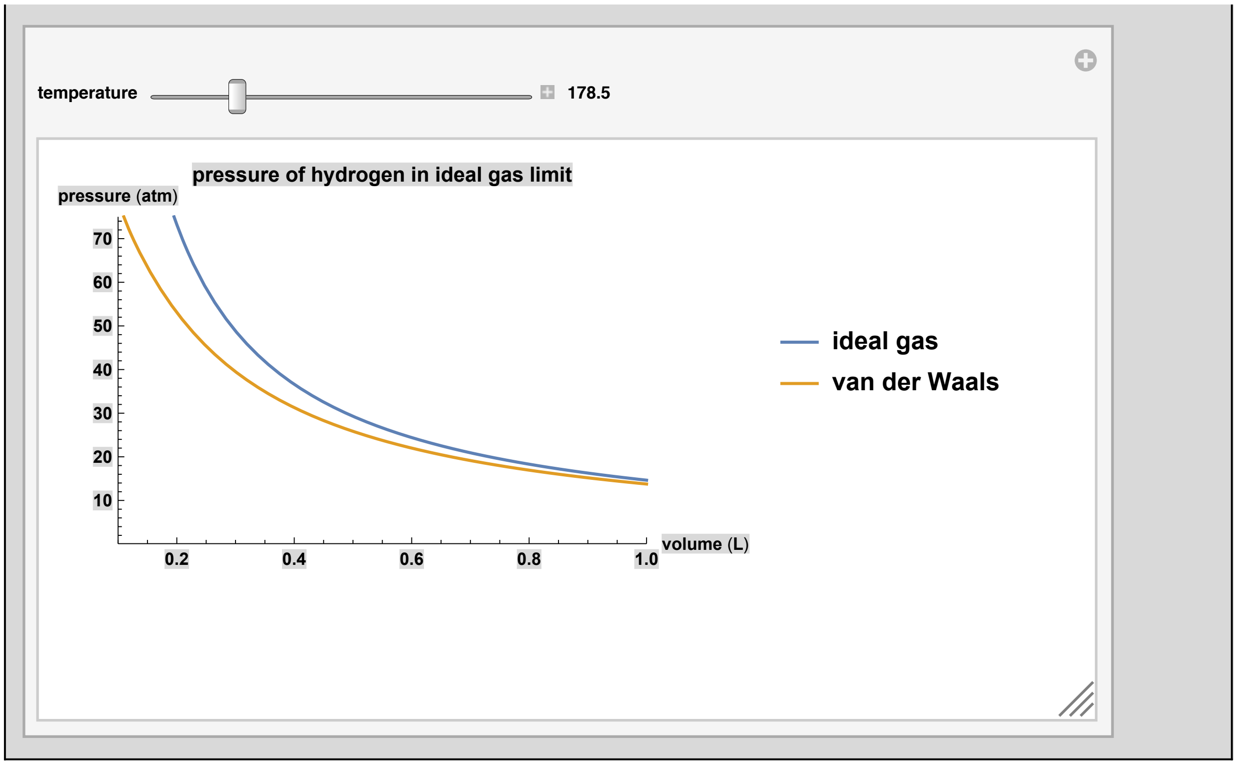
Science bugs are the most tricky as they won't cause the program to fail but are still errors that need to be caught!

gen. However, we have some science bugs in our code. The boiling temperature of oxygen is ~90 K so we shouldn't be looking at any temperatures below that. Also, we're looking at some fairly high pressures and low temperatures, so the ideal gas limit could be failing. Below, we've added in the van der Waals equation to attempt to better describe the behavior of the oxygen gas.

Remember, bugs in your code can be programming or science problems and the science ones are often harder to notice, as the code will not give an error. Make sure to be checking the validity of your equations and limits as you are creating your demo.

May need to evaluate this box to get the slider working

```
Clear[minT];
minT = 100; (* K *)
aO2 = 1.36; (* L^2 atm mol^-2 *)
bO2 = 0.0318; (* L mol^-1 *)
pVanderWaals[V_, T_] := nO2 * R * T / (V - nO2 * bO2) - nO2^2 * aO2 / V^2;

Manipulate[
 (*No calculation is needed*)

 Plot[
  {pressure[V, T], pVanderWaals[V, T]},
  {V, minV, maxV},

  PlotRange → {{minV, maxV}, {minP, maxP}},
  PlotLabel → "pressure of hydrogen in ideal gas limit",
  AxesLabel → {"volume (L)", "pressure (atm)"},
  PlotLegends → {"ideal gas", "van der Waals"}
 ]
 (*Number, plot, graphics, etc. you would like to show*)
 ,

 Control[{{T, 298, "temperature"}, minT, maxT, Appearance → "Labeled"}]
 (*The variables defined here can change value interactively*)
 ,

 ContentSize → {550, 300},
 SaveDefinitions → True (*Important!*)
 (* Add any other options here *)
]
```



Plot[ ] and **Manipulate[ ]** have a lot of additional options and the same goes for other functions that you may use in your project.
While it is impossible to cover them all in this tutorial, you can always go to the Mathematica website for information.

Additional (and advanced) resource that you may find useful:
Intro to Manipulate[ ]
Control objects and Dynamic[ ]
Advanced Manipulate[ ]
Pure functions

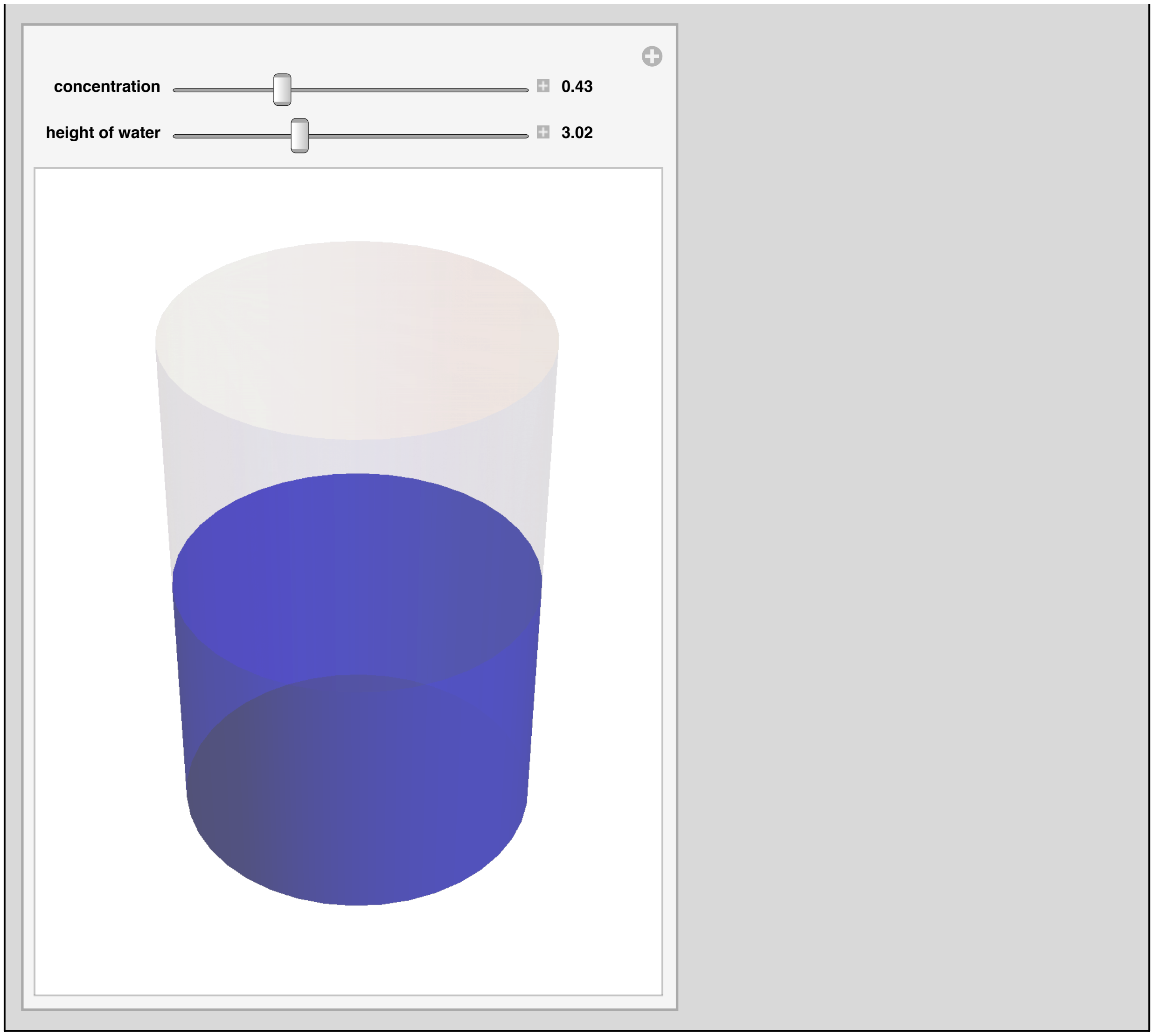
Mathematica Demonstrations Project

- All Mathematica Demonstrations Project demos use the function, **Manipulate**, once in the code.
- At the end of this semester, you will have also created one Demo for the Mathematica Demonstrations Project, as well as for other Chem 230/260 students, that uses the **Manipulate** function.
- In the following take home exercise, we will help you make your first Demo.

Take-Home Exercise

(GSI or peer leader): Let them know this is their homework for next week and then show them where the InitializationCode.nb is on Canvas. Open it so they can see what a Mathematica notebook looks like normally and mention that the notebook functions the same way as the tutorial, with Shift + Enter giving the Output. Also, mention that if they are working on the same coding problem for more than 30 min, they should contact a GSI or peer leader for help

For the take-home exercise, you will create a beaker filled with blue liquid that can be manipulated in both its height in the beaker and in concentration of the color of the liquid. Your end demonstration should look like this, with minor modifications allowed for its appearance:



Some starting steps:

1. Open up the Initialization Code for the take-home assignment on Canvas. You can code input and do Shift + Enter to get the output in a notebook just like you do in the tutorial.

2. Look back at the **Graphics3D** section and **Example3** (the ice cream cone) on slide 11 to remember how to make 3D graphics. You should start by building a static beaker with liquid. You will need to look at the Mathematica reference to understand how to build cylinders.

3. Build the **Manipulate[ ]** function around the static beaker following the **Manipulate** section on slide 15 and **In-Studio Exercise III**.

This should take you around an hour to complete. If it is taking longer than that, *that is definitely ok*, just make sure to ask a peer leader or GSI for help on what you are stuck on! It is usually useful when asking for help via email to include your code as an attachment as well.

End Day 2 (Remember, save your work!)

Conditionals

- Conditional expressions have values of either **True** or **False.**
  - The operators **==** , **!=** , **<=** , **>=** , **<** and **>** compares the value of two numbers. Pay attention to the difference between set (or assignment) operator **=** and equal operator **==** .

💡 **Example 1**: Write your prediction of the following conditional expressions as a comment next to each line, then evaluate to see if you are right.

The Input is already there; you can walk through each line, having someone guess the correct answer and putting the guess as a comment, then evaluating at the end

```
one = 1; two = 2; three = 3;

one == 1
two != 2
three ≥ 3
three ≤ 2
two > three
two < three
```

Emphasize == is the Equal operator (= is the assignment operator)

Will need to explain that != is the Not Equal operator

```
True
```

```
False
```

```
True
```

```
False
```

```
False
```

```
True
```

- The operators **&&** , **||** and **!** are the logical **And**, **Or** and **Not** operators, respectively. They take conditional expressions and returns a value of **True** or **False.**

💡 **Example 2**: Predict the result of the following code, then evaluate to see if you are right.

Again, the input is already there; you can give them ~30 sec to write the comments and then evaluate

```
(one == 1) && (two ≠ 2)
(three ≥ 3) || (three ≤ 2)
! (two > three)
```

```
False
```

```
True
```

```
True
```

Functions that utilize conditional expressions in *Mathematica* include **If[ ]** and **While[ ]**.

- **If[ ]**
    - Syntax: **If[*condition*, *true_expr*, *false_expr*]**.
    - Evaluates ***true_expr*** if the condition is **True**, or evaluates ***false_expr*** if the condition is **False**.
    - **Example 3**: Define a variable *testVal* as a number. Write an **If** statement that evaluates to "*In bounds*" if *testVal* is in between 10 and 20, or "*Out of bounds*" if otherwise.

```
testVal = 15;
(* Give testVal some value *)

If[testVal ≥ 10 && testVal ≤ 20,
 "In bounds",
 "Out of bounds"]
```

```
In bounds
```

- **While[ ]**
    - Syntax: **While[*condition*, *body*]**
    - Evaluates ***body*** repeatedly until the condition becomes **False**.
    - Watch out for infinite loops! Use Evaluation > Abort Evaluation to break an infinite loop.
    - **Example 4**: Calculate the sum of all the prime numbers that are smaller than 100. Use **Prime[n]** for the n^{th} prime number.

```
sum = 0; n = 1;
While[Prime[n] ≤ 100,
 sum = sum + Prime[n]; (*May also introduce += *)
 (* n = n + 1 *) (* same thing as n += 1 *)
 n += 1;
]
sum
```

Good idea to first do this with n = n + 1 then replace that with n += 1 to show how they do the same thing

```
1060
```

- **Which[ ]**
  - Syntax: **Which[*test*$_1$, *value*$_1$, *test*$_2$, *value*$_2$, ...]**
  - Evaluates each ***test*** until one is true.
  - It will return the value from the *first* true test, even if multiple tests are true.

Before doing the Which statement example, we are first reminding ourselves what testVal is and in the process, showing the weird setting of Mathematica where it changes the order of strings & variables. One way to fix this is using the Row function, which we're showing them here

Example 5: Assign a variable to a Which statement using *testVal* and two true tests then evaluate.

```
(* want to remind ourselves what testVal is *)
"testVal is " testVal (* Mathematica will print variables before strings *)

(* this will put items in a row in the order listed *)
Row[{"testVal is ", testVal}]
```

Evaluate this first to show that it is not what we want, then do the Row function

```
15 testVal is
```

```
testVal is 15
```

```
testWhich = Which[
   testVal == 10, "testVal equal to 10",
   testVal < 50, "testVal less than 50",
   testVal < 20, "testVal less than 20"
  ];

testWhich
```

```
testVal less than 50
```

Point out that while testVal is less than both 50 and 20, it is going to print out just the statement corresponding to testVal < 50, since that was the first True test

Rules and Substitutions

- Here is how to plug in a specific value to an expression.

Both Input & Output are already there

```
x^2 + 10 x + 20 /. x → 5
```

```
95
```

- The general syntax for substitution: *expr* /. *rule*
  - A *rule* is a special expression with an arrow in it.
  - Technically, the options in **Plot[ ]** and **Grid[ ]** are also rules. However, those are used in another way and has nothing to do with our discussion here.
  - The **ReplaceAll** operator **/.** threads through lists, but some special syntax apply.

Example 1: Use replace all with a list of x, x^2 , and $x + 1$.

```
{x, x^2, x + 1} /. x → 5
```

```
{5, 25, 6}
```

Example 2: Evaluate the following input to see how the replacement works with lists.

```
x /. {x → 14, x → 19, x → 10} (* Applies the replacements one by one *)
x1 + x2 + x3 /. {x1 → 14, x2 → 19, x3 → 10} (* How this is meant to work *)
```

```
14
```

```
43
```

Example 3: Use ReplaceAll to create a list of 14, 19, and 10.

```
x /. {{x → 14}, {x → 19}, {x → 10}} (* Thread through nested lists *)
```

```
{14, 19, 10}
```

You can say that this section was showing the ReplaceAll operator on its own, but we will most commonly be using it with the Solve and NSolve functions, shown next

How to Solve *Mathematica* Equations

- **Solve[ ]** and **NSolve[ ]**
  - Solves basic (e.g. polynomial) equations and systems of equations analytically or numerically, respectively.
  - Syntax: **Solve[*exp, var*].** The first argument being a conditional expression that stands for the equation.
  - **Example 1**: First define a function *poly3(x)* equal to $x^3 + 3x^2 - 6x - 8 = 0$. Use **Solve[ ]** to find the three roots of this third-order polynomial.

```
Clear[x]; Make sure to Clear x
poly3[x_] := x^3 + 3 x^2 - 6 x - 8 == 0
Solve[poly3[x], x]
```

```
{{x → -4}, {x → -1}, {x → 2}}
```

- The solution is given in the form of a nested list of rules, which can be used for replacement.
  - **Example 2**: Extract the numbers from the solution of the polynomial in **Example1** using **x /. {...}**

Returns a list for roots

```
roots = x /. Solve[poly3[x], x]
```

```
{-4, -1, 2}
```

  - To get a specific element from the list, use **listName[[index]]**
  - The **index** starts from 1 (not 0).
  - **Example 3**: Retrieve the first root value.

```
roots[[1]]
```

```
-4
```

- Solving simultaneous equations.
  - **Example 4**: Solve $x^2 + y^2 = 1$ and $x = 2y$.

```
Clear[x, y];
Solve[x^2 + y^2 == 1 && x == 2 y, {x, y}]
```

$$\left\{\left\{x \to -\frac{2}{\sqrt{5}}, y \to -\frac{1}{\sqrt{5}}\right\}, \left\{x \to \frac{2}{\sqrt{5}}, y \to \frac{1}{\sqrt{5}}\right\}\right\}$$

- **FindRoot[ ]**
  - Finds *ONE* root of a function numerically from a given starting point. It will automatically choose the root closest to the starting value.

  - Syntax: **FindRoot[*expr*, {*var*, *startpoint* }]**

Example 5: Use FindRoot[ ] on the third-order polynomial from earlier.

```
FindRoot[poly3[x], {x, 0}]
```

```
{x → -1.}
```

Programming Style

Organized code is very important! They will be working on their code on-and-off for several weeks & it is much easier to come back to and work on with other people if it is organized and commented. Also, if you want other people to look at and use your demo after it is published, this is more likely if it is organized

- Following a consistent, easy to read style makes your program easier to read and understand.
  - Use proper indentation
    - In most cases, you have nested functions with multiple brackets. Using proper indentation helps you to easily locate different functions and pieces of code. Reading the second example is far easier than the first one:

☹ Unorganized style

```
Grid[
 {{Show[Plot[Sqrt["value 1"], {k, 0, Pi}, PlotRange → {{0, Pi},
       {0, 150}}, AxesLabel → {"θ (radians)", "force (N)"},
     PlotStyle → {Red, Thickness[0.003]}, ImageSize → 250,
     PlotLabel → Style["total force on charge 1", FontSize → 8]],
    Graphics[{PointSize[0.025],
      Point[Dynamic[ {k, Sqrt["value 2"]}]]}]]]}}]
```

☺ Better, Organized Style

```
Grid[{{
   Show[
    Plot[
     Sqrt["value 1"],
     {k, 0, Pi}, PlotRange→{{0, Pi}, {0, 150}},
     AxesLabel→{"θ (radians)", "force (N)"},
     PlotStyle→{Red, Thickness[0.003]}, ImageSize→250,
                 PlotLabel→Style[
       "total force on charge 1", FontSize→8]
    ],
    Graphics[
     {PointSize[0.025], Point[Dynamic[ {k, Sqrt["value 2"]}]]}]
    ]}
 }]
```

- (* Use notes so that a person new to your code can follow your work *)
- Use consistent style. No matter which one you choose, stick to your choice.
- Don’t make style an afterthought.
- Use proper and meaningful names for defining your variables.
  - Don’t name your variables something incomprehensible such as x, y, tt. Instead, use meaningful names like “initialTemp” or “inputVal”.

In Studio Exercise IV

Look at slide 24 and do not evaluate the cells until you skim through the code. With your group, spend 10 minutes and describe what each piece of the code is doing. After ten minutes, look at slide 25 and continue this exercise. As a group discuss the benefits of having notes in your code.

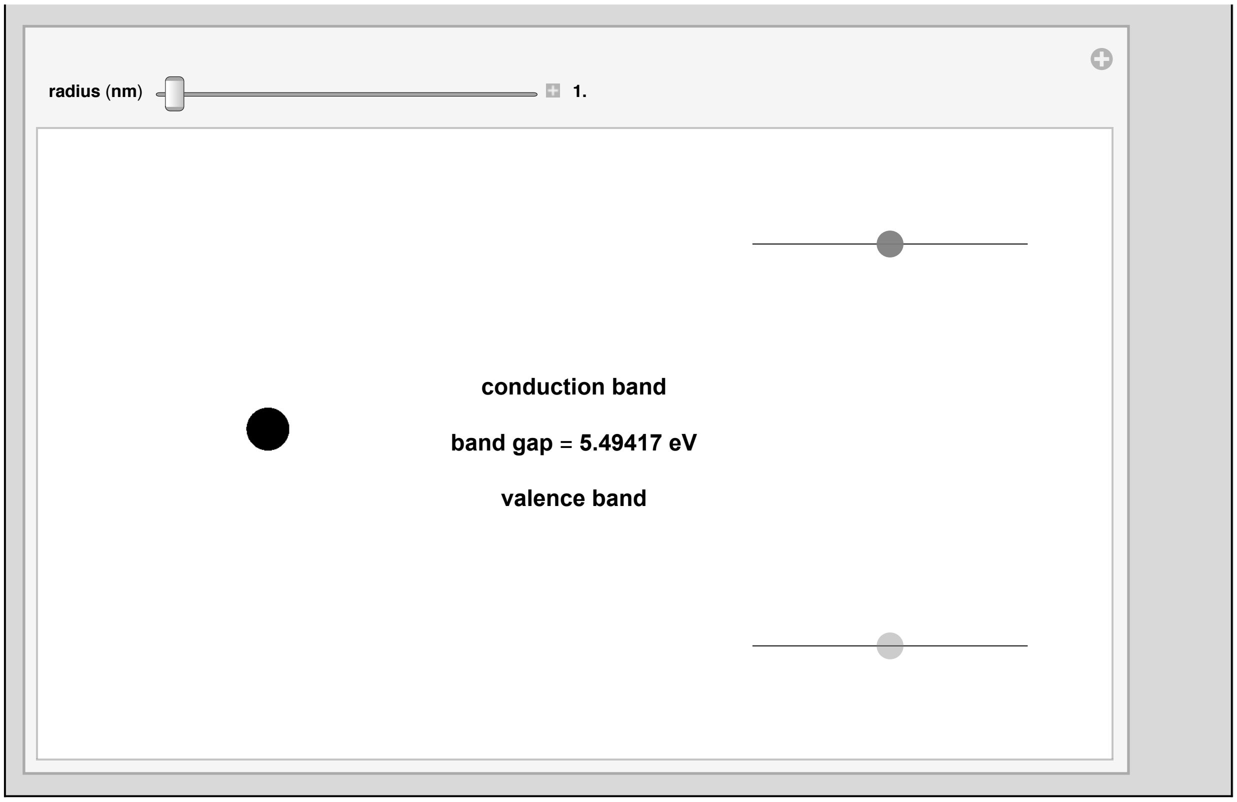
10 minutes is pretty long, you can say "Try to predict what the unorganized code will output, then look at the organized code and output"

Exercise IV

Code with no notes, poor variable names, and few indents:

Also note that the demo changes size sometimes when moving the slider, which is not allowable for published demos. This is fixed in the
better code shown on the next slide by introducing the function **NumberForm[]**, which allows you to control the number of decimal
places shown.

```
Eg = 2.81 * 10 ^ - 19;
h = 6.626 * 10 ^ - 34;
me = 1.18 * 10 ^ - 31;
mh = 4.09 * 10 ^ - 31;
c = 3 * 10 ^ 8;

Manipulate[
 e = (Eg + (h ^ 2 / (8 * (r * 10 ^ - 9) ^ 2)) * ((1 / me) + (1 / mh)));
 w = (h * c) / e;
 color = w * 10 ^ 9;
 bandgap = 6.2415 * 10 ^ 18 * e;
 Row[{Graphics3D[{ColorData["VisibleSpectrum"][color], Sphere[{0, 0, 0}, r]},
    Axes → False, Boxed → False, PlotRange → 6, ImageSize → {200, 300}], Text[
    Style[Grid[{{"conduction band"}, {}, {Row[{"band gap = ", bandgap, " eV"}]},
       {}, {"valence band"}}], FontSize → 12]],
   Graphics[{Line[{{1.5, .4 * bandgap}, {-1.5, .4 * bandgap}}],
     Line[{{1.5, - .4 * bandgap}, {-1.5, - .4 * bandgap}}],
     Opacity[.95, Gray], PointSize[0.07], Point[{{0, .4 * bandgap}}],
     Opacity[.4, Gray], PointSize[0.07], Point[{{0, - .4 * bandgap}}],
     Opacity[0], EdgeForm[], Rectangle[{1.6, 2.2}, {-1.6, -2.2}] },
    PlotRangePadding → 0.5, ImageSize → {200, 275}]}] ,
 Control[{{r, 1, "radius (nm)"}, 1, 5, Appearance → "Labeled"}],
 SaveDefinitions → True
]
```



conduction band

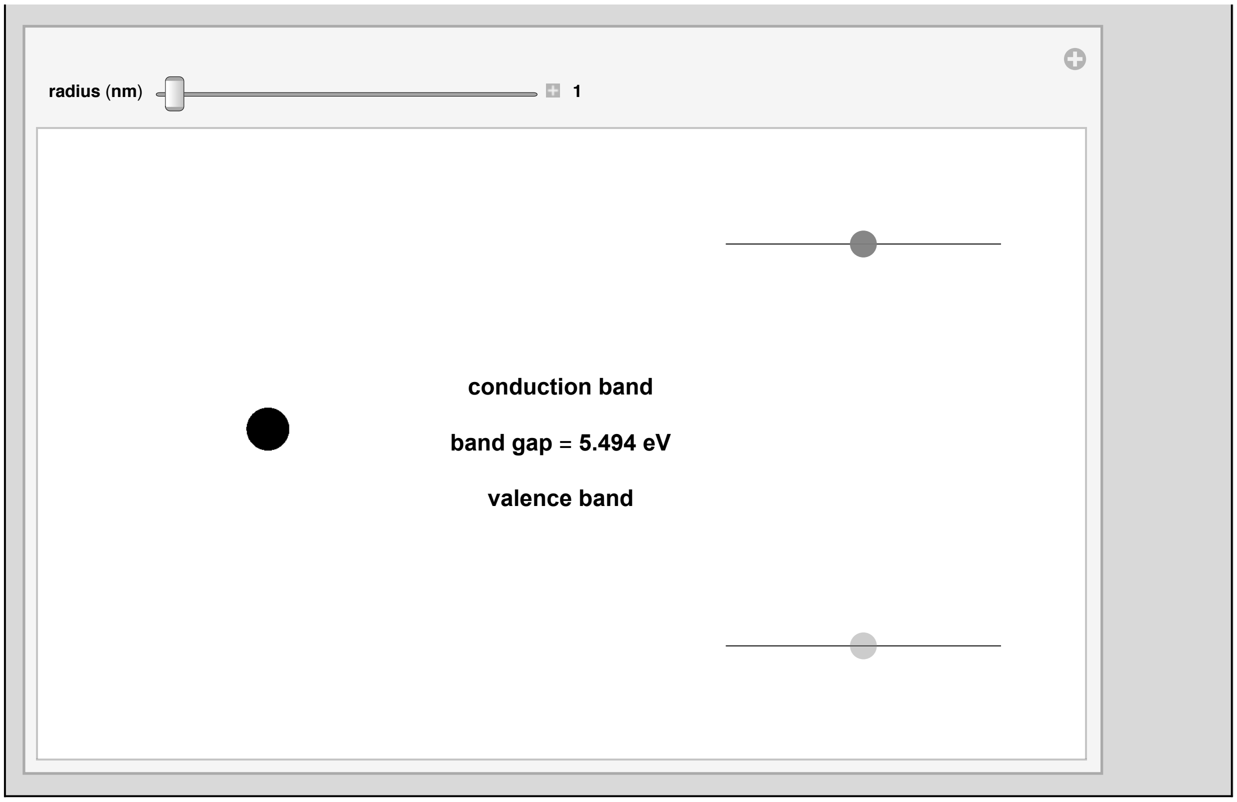
band gap = 5.49417 eV

valence band

Code with notes, descriptive variable names, and indents and line breaks:

This demo does not change size when moving the slider because of the function **NumberForm[]** around the outputted bandgap energy,
which allows you to control the number of decimal places shown.

```
energyGap = 2.81 * 10 ^ - 19; (* bulk band gap energy J *)
h = 6.626 * 10 ^ - 34; (* Planck's constant Js *)
massElectron = 1.18 * 10 ^ - 31;(* mass of electron kg *)
massHole = 4.09 * 10 ^ - 31;  (* mass of hole kg *)
c = 3 * 10 ^ 8; (* speed of light m/s *)

Manipulate[
 (*equation that uses given radius to provide energy in J *)
 energy = (energyGap +
    (h ^ 2 / (8 * (radius * 10 ^ - 9) ^ 2)) * ((1 / massElectron) + (1 / massHole)));
 (*equation that uses energy to provide wavelength in m *)
 wavelength = (h * c) / energy;
 (*equation that converts wavelength in m to nm *)
 color = wavelength * 10 ^ 9;
 (* equation that converts the band gap energy from J to eV *)
 bandgap = 6.2415 * 10 ^ 18 * energy;

 Row[{
   (* Nanoparticle with color given by equation *)
   Graphics3D[{
     ColorData["VisibleSpectrum"][color],
     Sphere[{0, 0, 0}, radius]
    },
    Axes → False, Boxed → False, PlotRange → 6, ImageSize → {200, 300}
   ],

   (* Output of band gap value
    and labeling of conduction band and valence band *)
   Text[
    Style[
     Grid[{
       {"conduction band"},
       {},
       {Row[{"band gap = ", NumberForm[bandgap, {4, 3}], " eV"}]},
       {},
        {"valence band"}
       }],
      FontSize → 12]],

    Graphics[{
      (* Conduction band *)
      Line[{{1.5, .4 * bandgap}, {-1.5, .4 * bandgap}}],
      (* Valence Band *)
      Line[{{1.5, -.4 * bandgap}, {-1.5, -.4 * bandgap}}],
      (* Dot on conduction band *)
      Opacity[.95, Gray], PointSize[0.07], Point[{{0, .4 * bandgap}}],
      (* Dot on valence band *)
      Opacity[.4, Gray], PointSize[0.07], Point[{{0, -.4 * bandgap}}],
      (* Invisible rectangle to maintain proportions with moving bands *)
      Opacity[0], EdgeForm[], Rectangle[{1.6, 2.2}, {-1.6, -2.2}]
     },
     PlotRangePadding → 0.5,
     ImageSize → {200, 275}
    ]
   }]
  ,
  (* radius of the nanoparticle in nm *)
  Control[{{radius, 1, "radius (nm)"}, 1, 5, Appearance → "Labeled"}],
  SaveDefinitions → True
]
```



Note: The input from Exercise IV is adapted from a demonstration describing the color tuning in CdSe Nanocrystals. View the demonstration and demonstration code here:
Color Tuning of CdSe Semiconductor Nanocrystals

While publication is a ways off, it is good to keep these requirement in mind, as it is much easier to follow them from the beginning than to reach the end and realize you have to make a bunch of changes

How to get your demonstration published

As alluded to previously, The Wolfram Demonstrations Project has some requirements to keep all of its published tutorials uniform.

- Only capitalize proper nouns in the user-interface of your demonstration.

- Make sure your demonstration does not change sizes when you play with sliders or change the size of your window.

- Only use ONE **Manipulate** function (no nested **Manipulate**).

- Keep in mind that there are size limitations. Try to keep your demo pretty small in size, but still easy to read.

- Make your variable very specific. Some variables, such as **D**, **N**, **E**, are actually stored by *Mathematica* and you cannot use them in a demonstration. If you encounter this, the solution to this problem is easy. Simply rename your variables.

- Read the following link. When you are ready to publish your demonstration, check this link again to make sure you are following these guidelines.
  - In the meantime, don’t be afraid to be creative with your demo and push the boundaries.

Authoring Demonstrations

Extra Practice: Debugging a File

In each of the following four codes, “SuperpositionofWaves1.nb”, “SuperpositionofWaves2.nb”, and “SuperpositionofWaves3.nb”, “SuperpositionofWaves4.nb” there is one bug (Each of these files can be found in the CTools folder). If you would like extra practice with debugging, you can work on correcting the bugs and checking the solutions to see if you found them.

If they want practice debugging outside of class, there are files they can practice debugging with (along with the solutions) in the "Debugging Practice" folder with the Tutorial

End

Fun Demo

Fun demo of a Potter's Wheel to do at the
end to show possible complexity of a demo

End Day 3 (Remember, save your work!)