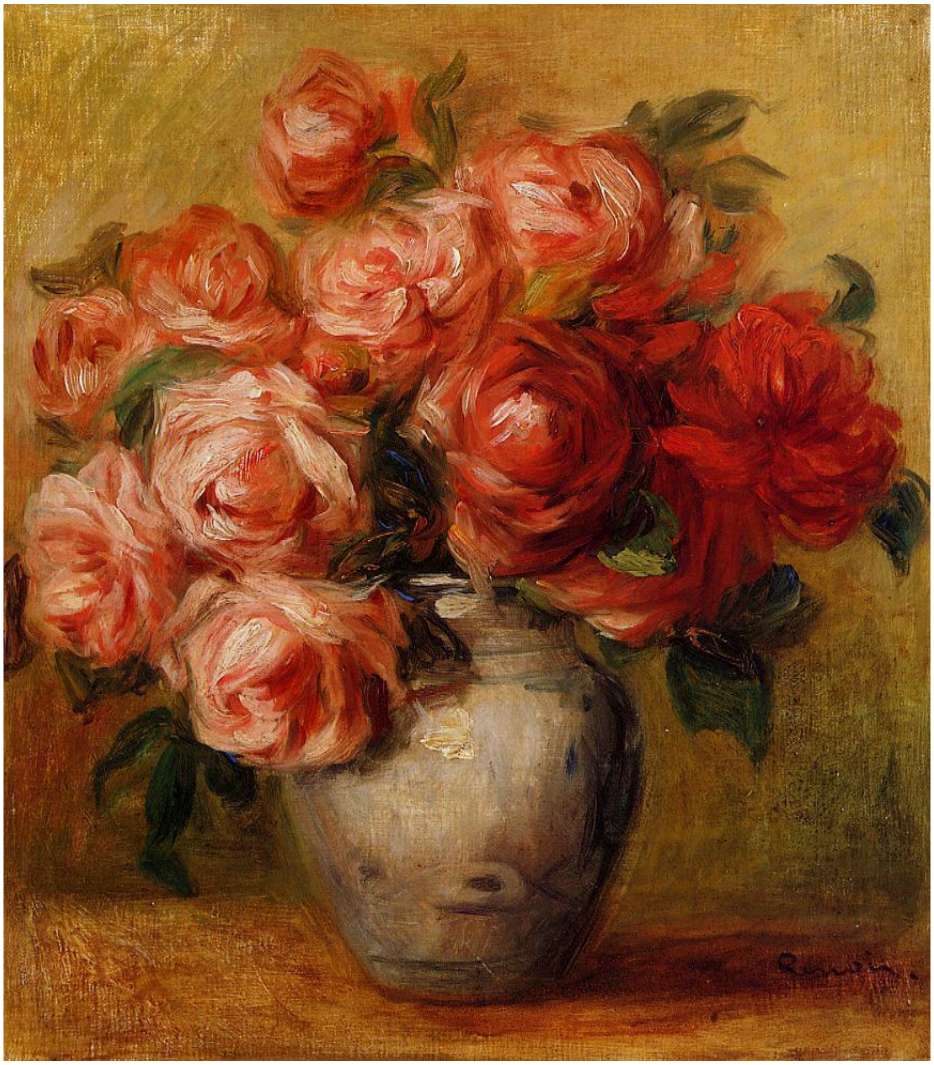


https://cdn2.oceansbridge.com/2017/07/25000538/Still-Life-with-Roses-Pierre-Auguste-Renoir-Oil-Painting-768x875.jpg

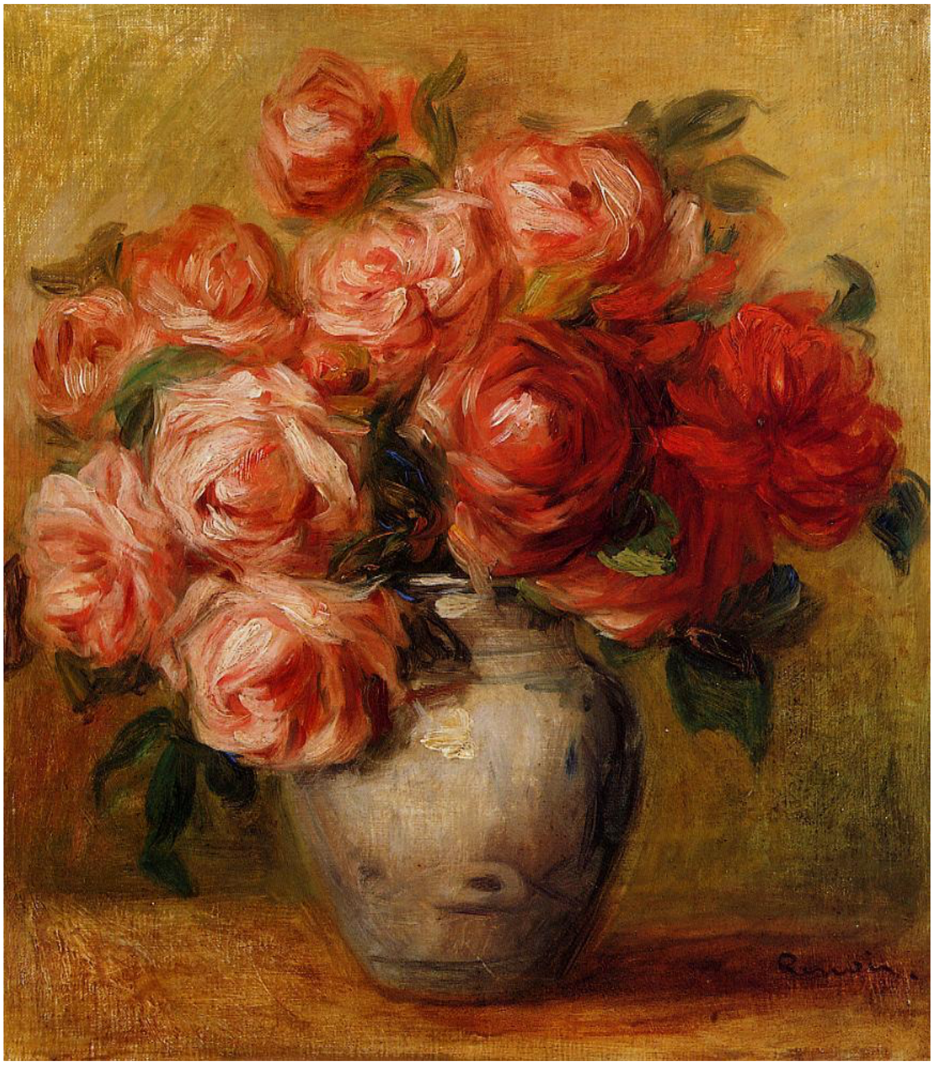
# Physics 371
# Design Like a Physicist
# Fall 2022

George Gollin
University of Illinois at Urbana-Champaign

Physics 371
George Gollin
University of Illinois at Urbana-Champaign
Spring 2022
Fridays, 1 pm – 5 pm; 3 credit hours.

Table of Contents

Introduction and Syllabus....5
Week 1: Organizations, Distributions, and Installations....37
Week 2: Breadboarding, TinkerCad....61
Week 3: DAQ, TinkerCad, Schematic Capture....69
Weeks 4 - 14: Away we go! ....89
C++ and Python Primer/Refresher ....91



https://cdn2.oceansbridge.com/2017/07/25000538/Still-Life-with-Roses-Pierre-Auguste-Renoir-Oil-Painting-768x875.jpg

# Physics 371
# Design Like a Physicist
# Fall 2022

# Introduction and Syllabus

George Gollin
University of Illinois at Urbana-Champaign

Physics 371
George Gollin
University of Illinois at Urbana-Champaign
Fall 2022
Fridays, 1 pm – 5 pm; 3 credit hours.

## Introduction and Syllabus

Introduction: project physics ........ 5
Prerequisites........ 7
We are not building robots ........ 7
Some possible projects ........ 8
The style in which we will work ........ 8
The tools we will use........ 10
- Arduino coding IDE ........ 10
- Schematic capture........ 11
- 3D printing........ 12
- Offline data analysis ........ 15

Sensors and other hardware........ 15
Some of what you'll learn ........ 17
The rhythm of things ........ 17
Homework, exams, grading, milestones, and your obligations........ 18
Course staff........ 22
Course calendar ........ 22
Detailed syllabus; see the list a few pages back for milestones. ........ 22
- Week 1, in class: see the Week 1 supplemental material writeup. ........ 22
- Post-class assignment (try to finish all of this before you meet with us next week)........ 23
- Week 2, conference with the professor + TAs ........ 23
- Week 2, in class........ 23
- Post-class assignment ........ 24
- Week 3, conference with the professor + TAs ........ 24
- Week 3, in class........ 24
- Post-class assignment ........ 24
- Week 4, conference with the professor + TAs ........ 24
- Week 4, in class........ 24
- Post-class assignment ........ 25
- Week 5, conference with the professor + TAs ........ 25
- Week 5, in class........ 25

Post-class assignment ....25
Week 6, conference with the professor + TAs ....25
Week 6, in class ....25
Post-class assignment ....25
Week 7, conference with the professor + TAs ....26
Week 7, in class ....26
Post-class assignment ....26
Week 8, conference with the professor + TAs ....26
Week 8, in class ....26
Post-class assignment ....26
Week 9, conference with the professor + TAs ....26
Week 9, in class ....26
Post-class assignment ....27
Week 10, conference with the professor + TAs ....27
Week 10, in class ....27
Post-class assignment ....27
Week 11, conference with the professor + TAs ....27
Week 11, in class ....27
Post-class assignment ....27
Week 12, conference with the professor + TAs ....27
Week 12, in class ....27
Post-class assignment ....27
Week 13, conference with the professor + TAs ....27
Week 13, in class ....27
Post-class assignment ....27
Week 14, conference with the professor + TAs ....28
Week 14, in class ....28
A fully loaded data logger ....28

### *Introduction: project physics*

Some years ago Carl Wieman won the Nobel Prize for creating a Bose-Einstein condensate in a dilute cloud of 2,000 atoms. At the time he was a professor at the University of Colorado, and had noticed that his physics students appeared to undergo a dramatic transition during the first year of graduate school. As undergraduates they would attend lecture-based classes and master course content by listening to their professors and slogging through weekly problem sets. (You all know what this is like!) By the end of the semester, most of the class would understand most of the material, but would find it difficult to integrate it into a coherent picture of, say, classical electrodynamics. And a semester after a course had ended, most students would not have retained their mastery of the topic. They would find it difficult to apply the material in, say, a lab course. But after a year of graduate school—during which students would work on difficult material without the distracting edge effects of 50-minute class periods—their competence at navigating confusing subjects and difficult problems would increase enormously.

Wieman thought that teaching physics to undergraduates in a manner that more closely resembled graduate education might be beneficial. He began to explore project-based courses, in which students would learn physics by mastering what they needed to complete tasks that were more like research projects than was usually true in undergraduate instruction. The results were dramatic.

You've already had some experience with this instructional mode if you've taken Physics 298owl from me. It's different from fighting to stay awake for an hour in lecture, then sifting through the wreckage to extract what you need to do the homework assignment!

You will be performing the one-semester analog of a PhD research thesis: defining a measurement to be performed, designing and building an instrument that might be capable of recording data necessary for the measurement, testing your device, doing the field work to record valid data, then analyzing the data to form supportable, reproducible conclusions. If all goes well, you'll find this so captivating that it will be hard to put your work aside to attend to your other academic obligations. I suspect it is this strong engagement with a project that drives the transition from an undergraduate level of skill to the expert mastery typical of graduate researchers.

Your device will comprise an embedded processor—many of you will use a Microchip Technology Inc. ATmega2560 microcontroller on an Arduino Mega 2560 board—interfaced to a suite of sensors built onto small "breakout" printed circuit boards. The Arduino's USB interface will allow you to download your (compiled) programs to the microcontroller and communicate with the processor through a serial interface.

After selecting your research partners and choosing a project, you'll begin assembling your instrument on a breadboard. You will develop the programs necessary to drive the various sensors, integrate them into a data acquisition system of your own design, build a more robust version of your device on a printed circuit board, 3D-print a case for it, and venture out into the world to do field work and record data. You will analyze your data, draw (and justify) conclusions, and document (and present) your findings. In the interest of efficiency, I encourage you to use in your design as much public-domain material as you are able to find. There is, in this course, no reason to reinvent the wheel.

### ***Covid-19***

You will need to do this in a Covid-19 compliant fashion, communicating with other members of your project group and with course staff while properly wearing (non-vented, preferably N95) masks and maintaining CDC-recommended distances.

The university's policies concerning covid-19 are clear. See
https://covid19.illinois.edu/guides/students/ and
https://www.youtube.com/watch?v=ugA2zoaLfpg
for more details. Briefly:

1. Unless you are unable to be vaccinated because of valid medical or religious reasons, you must "be fully vaccinated (defined as 14 days after the final dose) with a university-accepted vaccine before beginning the fall semester" if you plan to be on campus this semester. No other grounds for remaining unvaccinated (for example, refusing vaccination because of a personal philosophical code) are acceptable.

2. If you are unable to be vaccinated, you must test for infection every other day. The SaferIllinois app (and the website https://covid19.illinois.edu/health-and-support/on-campus-covid-19-testing-locations/) list testing sites.

3. Regardless of vaccination status, you should wear a (non-vented) face mask at all times while you are inside the Physics 371 classroom.

The penalties for violating the university's covid-19 policies are substantial, and I will be unyielding in their enforcement.

The following are not university policies, but are based on my understanding of the CDC guidelines for classroom instruction, and the current state of what is known about the delta variant of the SARS-CoV-2 virus.

1. We should try to maintain a three-foot distance from each other at all times.

2. Vaccinated people are easily infected by the omicron variant, and can carry viral loads that are similar to those of unvaccinated infected individuals. Even though these "breakthrough infections" are almost always mild, or even asymptomatic, vaccinated-but-infected individuals can infect other people. Always keep this in mind.

3. If you plan to travel (for example, to visit your relatives at Thanksgiving), you should test for infection the day before leaving town to reduce the chance that you will carry an infection with you, and endanger your relatives.

### ***Prerequisites***

You must already know how to program. If you've learned to code in Python or C/C++, or Java, or some other language, you'll do fine. A grade of B- or better in CS 101, CS 125, or Physics 298owl is a suitable prerequisite. It's also fine if you've learned on your own. If you've never programmed before, consider delaying enrollment in Physics 371 until after you've done some coding.

You must have a basic working knowledge of introductory physics at the level of Physics 211 and Physics 212. More is better, though not necessary.

### ***We are not building robots***

Physics 371 is not a course in robot building. That would be an engineer thing, and we are physicists, not engineers. We are going to tackle measurements that—if they prove feasible—might make our corner of the world a little bit better. If we ***did*** build a robot, it would be to accomplish a significant end, for example recognizing the onset of a potentially catastrophic fall by an elderly person.

In Physics 371 you'll construct a hand-held device loaded with inexpensive sensors that are interrogated by a microcontroller—a small computer larded with additional features such as timers and analog-to-digital converters—and write the data acquisition software necessary to perform the measurements associated with your project. You'll assemble a prototype on a breadboard, construct a final (electrically equivalent) version on a printed circuit board, use a 3D printer to build a case for it, do field work, then write analysis code to understand what conclusions can be drawn from your data. You'll write a report presenting your results and justifying your conclusions, then publish it to the course web site.

We will loan you the parts and tools necessary to construct the prototype, and will expect you to return these at the end of the course. But we will give you what you need to build the PCB version, and let you keep some of it at the end of the term. (If you withdraw from the course we'll want you to return everything we've given you.)

The intellectual tradition in physics is for researchers to build their own instruments (buying off-the-shelf parts when available), ultimately creating sophisticated devices to perform the measurements that will tell us about the physics we are researching. It is not like this in all fields; my wife's background is in bio-inorganic chemistry, and she would assemble reactors from stock components, then run reaction products through spectrometers built by vendors like Varian and Hitachi.

So you'll be following the physics tradition, and you will be working in (intellectually) close collaboration with two or three other students.

### *Some possible projects*

Some of the projects are probably best imagined as feasibility studies that might inform the design of a more definitive future measurement. We will see how it goes!

See the course website for some that I have in mind. You are free to suggest other possibilities, though I reserve the right to veto anything that I feel is too difficult or too expensive.

### *The style in which we will work*

"DLP" stands for "Design Like a Physicist." That's a reasonably descriptive term for how we will go about things, though it wasn't my first choice for the three-character course identifier. Here is what I mean. If you took Physics 298owl from me you'll remember that I had you hand-code a lot of algorithms—integrators, Fourier transforms—that could also be found in professionally produced libraries. For pedagogical purposes, I had you reinventing a lot of wheels.

That's not how I've gone about my own research. If there's a pre-coded numerical algorithm that I can use, I'll appropriate it, generally putting proper attribution to its source in comments in my own code. If there's a circuit I need that's described in an engineering web site, I'll use it. Proper attribution can be placed on my schematic diagram. Sometimes it might be difficult to publish the source attribution—the 3D STL files you'll create for TinkerCad projects—but do keep in your own notes information about where you have found useful material.

You will keep track of your efforts in a physical paper notebook in which you describe your work, useful revelations, and calculations. I do not want this to be anything fancy, but your notes should be cumulative, rather than something you discard at the end of each class meeting. You should have your notebook open while working on your project. We won't be monitoring your compliance with this, but you really will find it useful. Be sure to date each of your entries. You should put notes about techniques you find (or invent) into your notebook so you can find them later.

You are to refrain from using your phones and laptops to visit social media sites, chat with friends, read news publications, and so forth. You are not to be looking at your tiny screens while any of us are presenting to the class. I will become VERY CROSS is I catch you doing this.

We will be using several different IDEs—Integrated Development Environments—during the semester. I expect you to install and use these in your work. If there are other tools that you'd prefer to use, keep in mind that it will be hard for you to share material with other members of your group. If you insist on staying with these, I am not going to be happy to find you wasting time translating your work into a mutually acceptable format.

You will be working with things for which your understanding will often be a little blurry. That's OK, and in fact that's the usual state of things in research. Taking the time to understand every last detail about an IDE is a waste of your time: it is better to focus your efforts on getting by, on muddling through. You will get more done per week this way than you would if you spent the time to understand everything completely. There is too much to do, and far more interesting things to consider than the arcane details of SPI and I2C interfaces. You want to understand them well enough to work with them, but not to write—without reference to external sources—the definitive *Handbuch der Was Auch Immer* document.

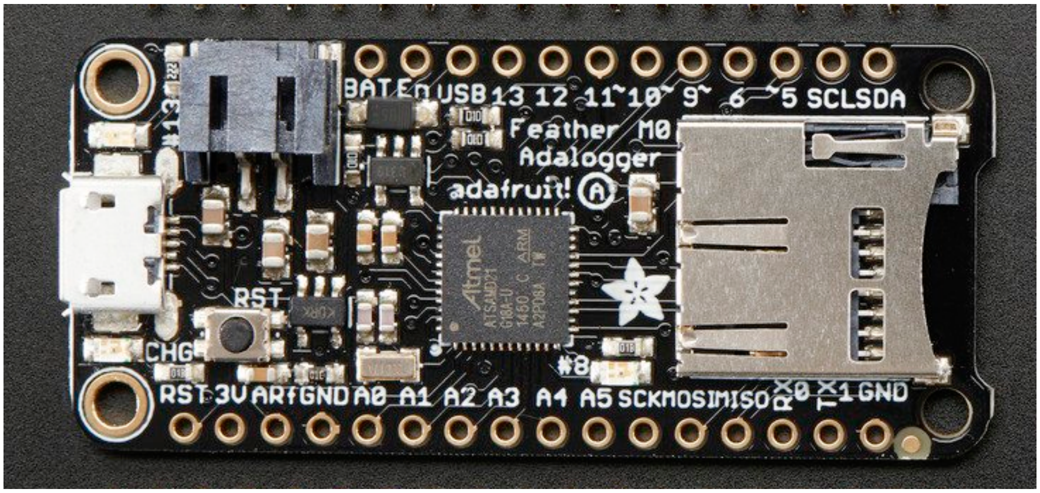
I will expect you to have these windows open on your laptop in class at all times: (1) the IDE for whatever you're doing; (2) a browser window with which you can search for (and download) useful things.

You will need to use a laptop that is able to read (micro)SD memory cards and to accept a standard USB-A connector. If your laptop can't do this, please acquire an adapter that will let you read a microSD memory card and another that will let you attach a USB-A cable, through which you will talk to the Arduino. I will have a few of these I can loan to you, but please get your own. Note that a "charger cable" WILL NOT work for you: it only carries ground and 5V lines, and omits the D+ and D- data lines. It's hard to tell the difference from external inspection between a full-blown USB cable and a charger cable. Figure below: USB-A on the left, USB-C on the right.

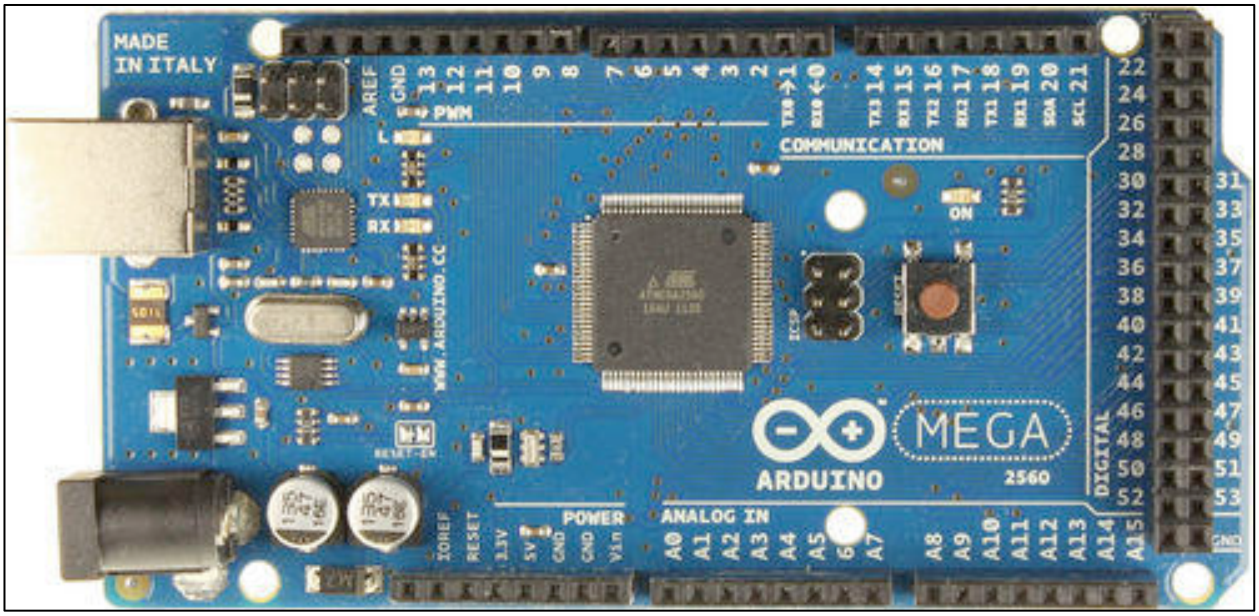
***The tools we will use***

*Arduino coding IDE*

The heart of your data logger will probably be either an Adalogger Feather M0 or an Arduino Mega 2560 microcontroller board. Pictures follow:



Adalogger Feather M0



Arduino Mega 2560

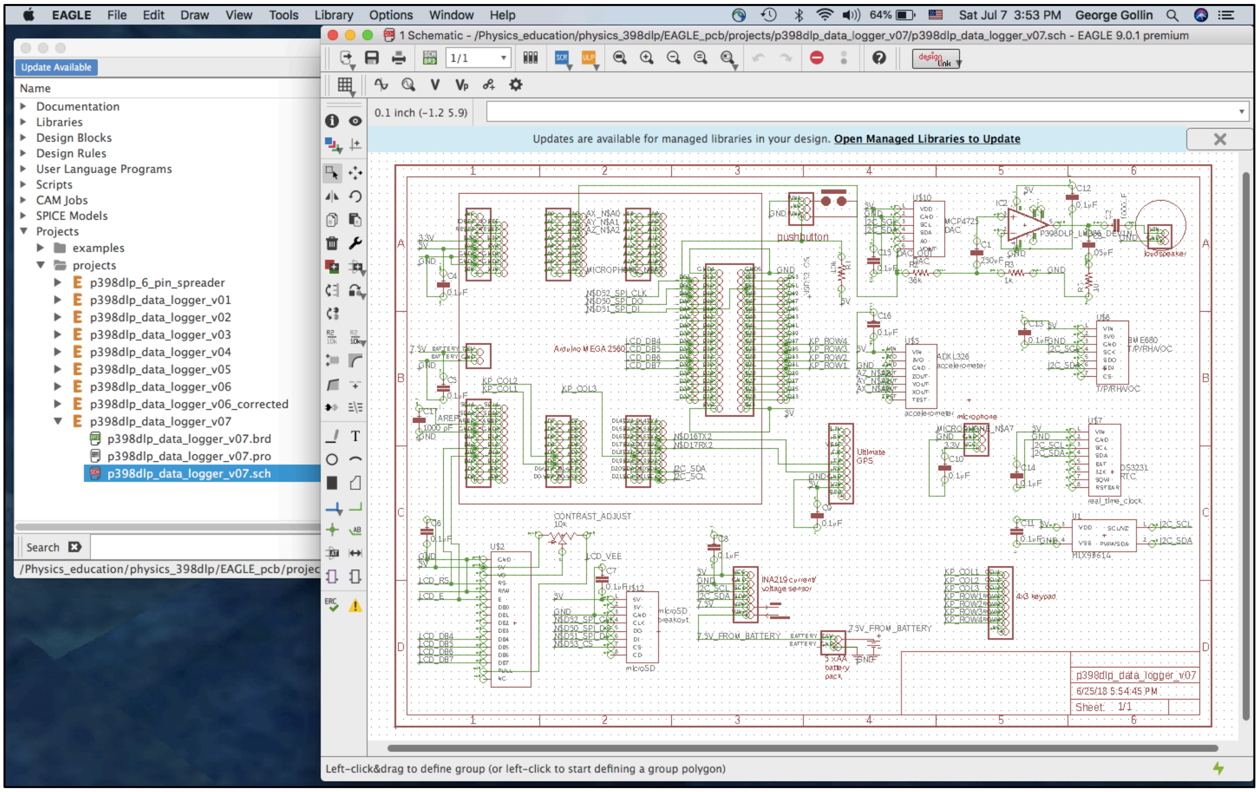
The Arduino is a remarkable little gizmo, featuring an Atmel Atmega2560 microcontroller running at 16MHz. The Atmega2560 has 256kB of flash memory in which your program will reside, along with 8kB of SRAM (static random access memory) in which will live the variables your program modifies as it executes. There are 16 analog inputs that feed an internal multiplexer whose output drives a successive approximation analog to digital converter.

The Arduino’s IDE (Integrated Development Environment) is quite a bit simpler than Anaconda’s iPython IDE. Most of what you will see on your screen is an editor window in which you will create/modify C++ programs that you will compile and upload to the Arduino.

*Schematic capture*

As you assemble your prototype on a breadboard, you’ll want to keep track of the wiring in your ever-more complex circuit. To do this, you’ll register an account with Autodesk and use their free-for-three-years EAGLE schematic capture tool. You will start with my version of the schematic and throw away the parts of it that you don’t plan to use.

Here’s a screen shot of the schematic capture tool. It might take you an hour to become reasonably proficient with it.



Updates are available for managed libraries in your design. Open Managed Libraries to Update

Left-click&drag to define group (or left-click to start defining a group polygon)

EAGLE schematic capture tool

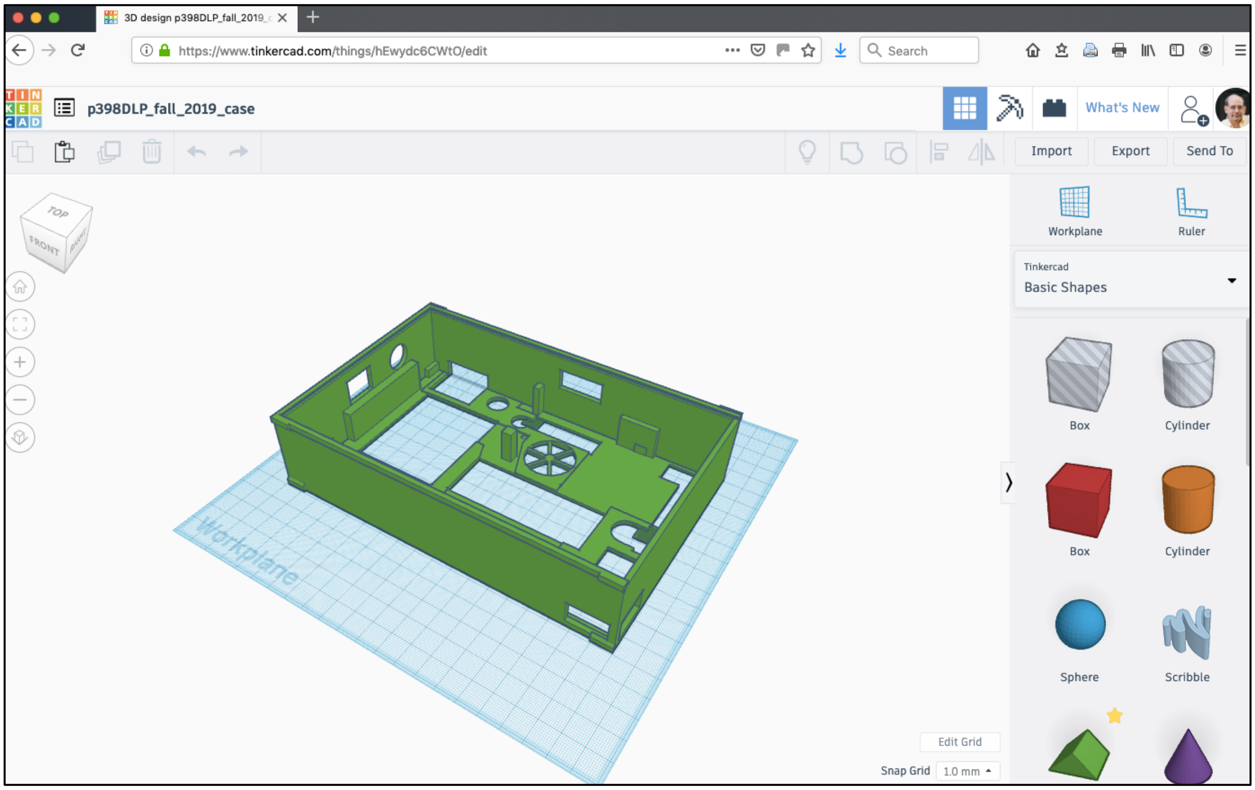
I've designed a PCB that is sufficiently general that everyone can use it. You will install the sensors that you plan to use, and omit those that you don't. I've had a few dozen copies of this fabricated by a commercial vendor so that we'll have plated-through holes and "silkscreen layers" at our disposal.

*3D printing*

You will use TinkerCad—another Autodesk product—to design a case for your device. TinkerCad will produce an STL (stereolithography) file that we will feed to Cura, also an Autodesk application, to convert the STL file into a gcode file of instructions to be executed by a 3D printer. (You are welcome to base your design on my version.)

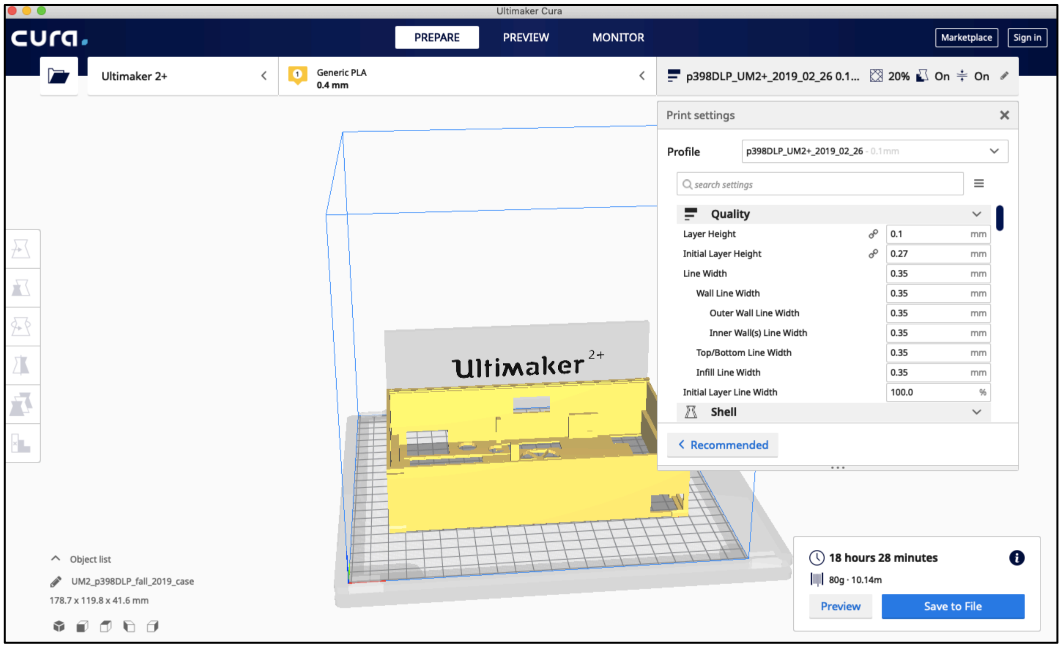
For practice you might consider designing a small box to hold a few of your sensor "breakout boards"; I can print these for you on the Ultimaker 2+ printers in my lab. When it comes time to fabricate the case for your data logger, I'll run the print jobs for you rather than negotiating with the Business School's MakerLab management.

Here's a TinkerCad screen shot of my Fall 2018 case design. TinkerCad is a web-based tool, easy to learn, and great fun to use.



TinkerCad screen shot

Here’s the Cura window. The latest version (as of January 2022) is 4.12.1; you’ll need to make sure you have the correct nozzle diameter (0.4 mm for the machines in my lab). You should also make sure that the “Preferences → Printers → Machine Settings → Printer → Origin at center” box is unchecked.



p398DLP_UM2+_2019_02_26 0.1... 20% On On

Print settings

Profile: p398DLP_UM2+_2019_02_26 - 0.1mm

search settings

Quality

| | | |
|---|---|---|
| Layer Height | 0.1 | mm |
| Initial Layer Height | 0.27 | mm |
| Line Width | 0.35 | mm |
| Wall Line Width | 0.35 | mm |
| Outer Wall Line Width | 0.35 | mm |
| Inner Wall(s) Line Width | 0.35 | mm |
| Top/Bottom Line Width | 0.35 | mm |
| Infill Line Width | 0.35 | mm |
| Initial Layer Line Width | 100.0 | % |

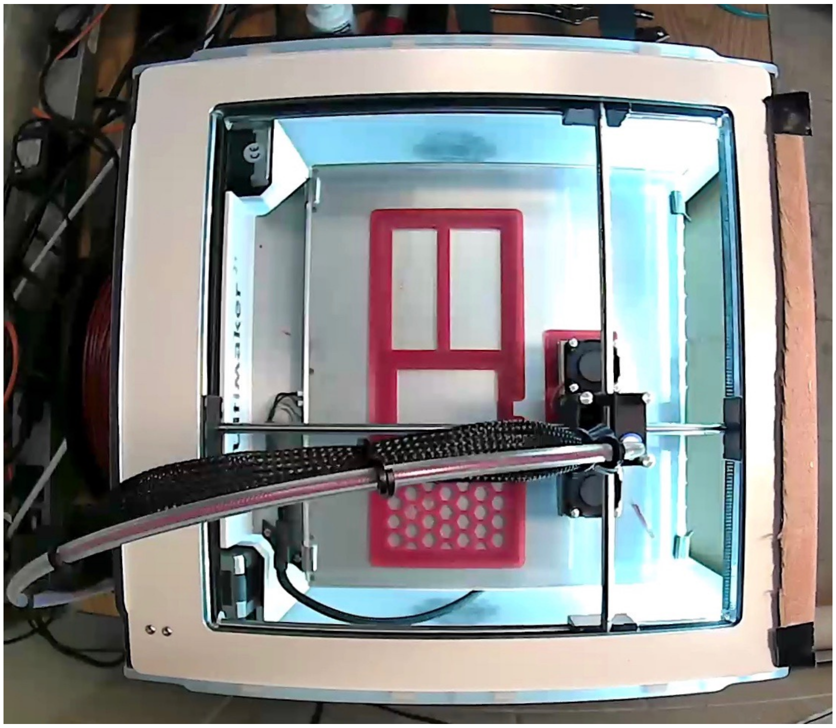
Shell

18 hours 28 minutes
80g · 10.14m

Object list
UM2_p398DLP_fall_2019_case
178.7 x 119.8 x 41.6 mm

Cura screen shot

Here’s an Ultimaker 2+ printing the Fall 2018 version of the data logger case. The camera uses a fish eye lens, which is responsible for the distortion of the image.

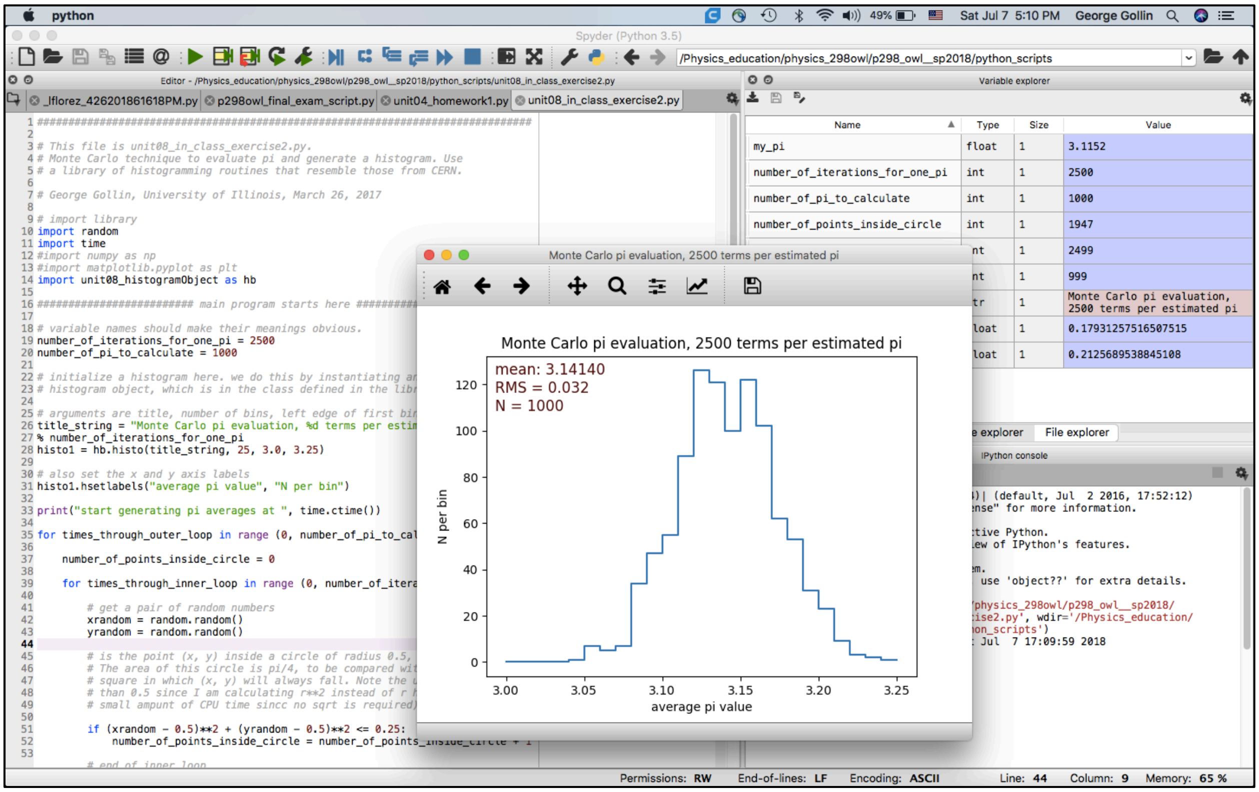


Ultimaker 2+ 3D printer

*Offline data analysis*

You should install Anaconda's Python spyder IDE for your data analysis work. Many of you are already familiar with Python, and with spyder. I will distribute copies of some of my Physics 298owl Python material as an introduction/refresher to the language.

A screenshot of the Python IDE is below.


python
Sat Jul 7 5:10 PM George Gollin
49%
Spyder (Python 3.5)
/Physics_education/physics_298owl/p298_owl__sp2018/python_scripts
Editor - /Physics_education/physics_298owl/p298_owl__sp2018/python_scripts/unit08_in_class_exercise2.py
_lflorez_42620186l618PM.py
p298owl_final_exam_script.py
unit04_homework1.py
unit08_in_class_exercise2.py
Variable explorer
Monte Carlo pi evaluation, 2500 terms per estimated pi
Monte Carlo pi evaluation, 2500 terms per estimated pi
mean: 3.14140
RMS = 0.032
N = 1000
N per bin
0
20
40
60
80
100
120
3.00
3.05
3.10
3.15
3.20
3.25
average pi value
e explorer
File explorer
IPython console
Permissions: RW
End-of-lines: LF
Encoding: ASCII
Line: 44
Column: 9
Memory: 65 %

```
################################################################################

# This file is unit08_in_class_exercise2.py.
# Monte Carlo technique to evaluate pi and generate a histogram. Use
# a library of histogramming routines that resemble those from CERN.

# George Gollin, University of Illinois, March 26, 2017

# import library
import random
import time
#import numpy as np
#import matplotlib.pyplot as plt
import unit08_histogramObject as hb

######################### main program starts here ##########

# variable names should make their meanings obvious.
number_of_iterations_for_one_pi = 2500
number_of_pi_to_calculate = 1000

# initialize a histogram here. we do this by instantiating an
# histogram object, which is in the class defined in the libr

# arguments are title, number of bins, left edge of first bin
title_string = "Monte Carlo pi evaluation, %d terms per estim
% number_of_iterations_for_one_pi
histo1 = hb.histo(title_string, 25, 3.0, 3.25)

# also set the x and y axis labels
histo1.hsetlabels("average pi value", "N per bin")

print("start generating pi averages at ", time.ctime())

for times_through_outer_loop in range (0, number_of_pi_to_cal

    number_of_points_inside_circle = 0

    for times_through_inner_loop in range (0, number_of_itera

        # get a pair of random numbers
        xrandom = random.random()
        yrandom = random.random()

        # is the point (x, y) inside a circle of radius 0.5,
        # The area of this circle is pi/4, to be compared wit
        # square in which (x, y) will always fall. Note the u
        # than 0.5 since I am calculating r**2 instead of r h
        # small ampunt of CPU time sincc no sqrt is required)

        if (xrandom - 0.5)**2 + (yrandom - 0.5)**2 <= 0.25:
            number_of_points_inside_circle = number_of_points_inside_circle + 1

        # end of inner loop
```

| Name | Type | Size | Value |
|---|---|---|---|
| my_pi | float | 1 | 3.1152 |
| number_of_iterations_for_one_pi | int | 1 | 2500 |
| number_of_pi_to_calculate | int | 1 | 1000 |
| number_of_points_inside_circle | int | 1 | 1947 |
|  | nt | 1 | 2499 |
|  | nt | 1 | 999 |
|  | tr | 1 | Monte Carlo pi evaluation, 2500 terms per estimated pi |
|  | loat | 1 | 0.17931257516507515 |
|  | loat | 1 | 0.2125689538845108 |

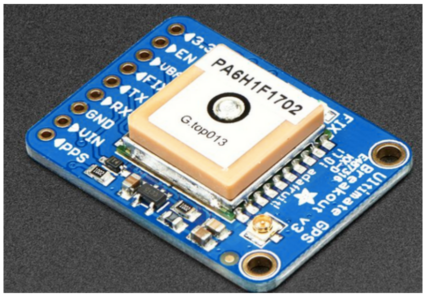
```
4)| (default, Jul  2 2016, 17:52:12)
ense" for more information.

ctive Python.
iew of IPython's features.

em.
  use 'object??' for extra details.

'physics_298owl/p298_owl__sp2018/
cise2.py', wdir='/Physics_education/
hon_scripts')
: Jul  7 17:09:59 2018
```

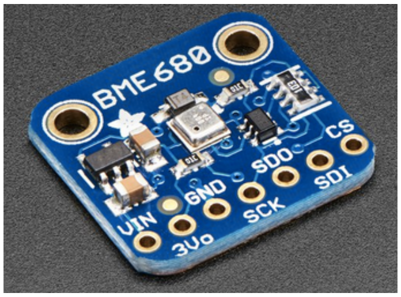
iPython IDE

***Sensors and other hardware***

Most of the sensors we'll use have been assembled onto small "breakout boards" by Adafruit Industries. I'll solder "pin headers" onto them so you can plug them into the breadboard you'll use when developing your prototype.

Here are photos of a few of the sensors, taken from the Adafruit web site. In order, left to right: GPS, amplified microphone, temperature-pressure-humidity-atmospheric volatile organic compound level. They are tiny, typically less than a square inch in size.





I have a partial list of sensors and other goodies that we have on hand in the Week 1 supplemental material document. Most of them are from Adafruit. I have enough of most of them for all of your breadboard and PCB devices, though depending on which project you do, you might use an Adalogger instead of Arduino processor.

If you'd like to fly an instrument package over, say, a farm, you can use the Physics 371 DJI Mavic 2 Pro drone. Note that you'll need to get a drone pilot's license before you can fly it.

DJI Mavic 2 Pro drone

Tools/materials we'll loan to each of you:

- wire stripper
- needle-nosed pliers
- tweezers
- eye protection
- breadboard
- digital multimeter
- spools of brightly colored 22 gauge solid-core wire
- lots of breakout boards

So it’s a pretty good collection of stuff.

### *Some of what you’ll learn*

You will learn to identify a measurement that you could make, in hopes of understanding more about, and perhaps improving, the State of Things.

You will learn to construct and build a device that might allow you to make those measurements.

You will learn to test and calibrate your device so that your data bear an understandable relationship to the physical parameters you are studying.

You will learn to do fieldwork, so that you come back from the world with valid data that can be analyzed and interpreted.

You will learn to report your results and speak of them to an audience of skeptics, supporting your conclusions with irrefutable facts.

You will also have a blast building your gizmo.

### *The rhythm of things*

Time-on-task is an important part of mastering the tools you will use this semester. Rather than staging the various tasks for completing your project sequentially, you’ll work with many of them in parallel. This will give you more time to digest the fine points of working with the tools we’ll employ.

I expect that you will work closely (in an intellectual sense) with other members of your group, both in and out of class. You should spend at least five or six hours each week, outside of class, working on your project.

Speaking loosely, these are the things you’ll do:

- select a project in collaboration with the other members of your group
- discuss measurements (and sensors) necessary for your project
- register a student, 3-years-free account with AutoCAD
- install IDEs for Arduino C++, Anaconda Python, EAGLE schematic capture, and Cura
- wire up a breadboard version of your data logger
- write (or adapt) code to talk to all the sensors you are using

- write a data acquisition program
- update a schematic diagram so that it represents your device
- figure out how to take data and verify its validity
- load and test a PCB
- 3D-print a case
- do your field work
- write code to analyze your data and (cross) calibrate your sensors
- report your results.

The computer-based tool set comprises the following:

- Arduino programing IDE
- Anaconda iPython programming IDE.
- EAGLE schematic capture
- TinkerCad 3D printing design tool
- Cura 3D printing renderer, though I will probably run this for you.

Each week you'll advance the design of your data logger, write Arduino code to communicate with your sensors, further develop your plans for field work (including the structure of data acquisition code), and address physical infrastructure matters like case construction and PCB fabrication.

After the first week we'll have brief reports to the class from one of the teams. Topics (which I will assign) might include how a particular sensor works, what an interrupt does, how the I2C data transfer protocol works, and so forth. A report should last at most ten minutes, be carried in at most ten PowerPoint slides, be presented to the class by all the team members, and be suitable for upload to the course web site.

Every week (times to be scheduled), each team will meet with me for 30 minutes to discuss progress, problems, clever ideas, and any other issues that might arise.

### ***Homework, exams, grading, milestones, and your obligations***

The homework will consist of moving your design forward as far (and as fast) as you can. I expect you will spend about six hours at this outside of class every week. You should work with the other members of your team as much as possible, sharing code and design tips as convenient. You should document your progress, your plans, your brilliant realizations, your frustrations, and your concerns in your notebook.

There will also be weekly homework assignments, to be submitted by email to the TA.

You and your team will meet with me once per week in Loomis 437 for 30 minutes. All members of your team must attend, and must arrive promptly at the scheduled time, without exception. You must always have on hand your laptop, breadboard circuit, notebook, and (once it is under construction) PCB circuit.

You and your team members will give several reports to the class over the course of the semester. The reports are to be clearly written PowerPoint presentations (with proper attribution of sources) aimed at your audience of fellow students, who will not necessarily know what you mean by, for example, "I2C interface."

You must come to class on time, arriving with your laptop and power adapter, goodie box of parts and tools, and breadboard. When you have a PCB version of your device you should bring both the breadboarded and PCB versions of your device.

There will be no midterm exams. But there are milestones that I ask you to meet, and will consider these when evaluating your work for a grade. Here are the milestones and deadlines:

1a. Modify the Arduino's blink program so that it blinks the initials (of your English/American name) in Morse code. (Week 1, by end of Friday class)

1b. Install and test a BME680. (Week 1, by end of Friday class)

1c. On your breadboard, install the following devices (in addition to the BME680 and Arduino): LCD (including 10kΩ trimpot), keypad, and microSD breakout. (Week 2, by beginning of Friday class)

1d. Formulate a project plan and division of project responsibilities. (Week 2, by midweek group conference with course staff)

2a. Install, set, and read back a DS3231 real time clock. (Week 2, by end of Friday class)

2b. Install and read back a GPS module. Use it to set the DS3231 real time clock. (Week 2, by end of Friday class)

2c. Write a short text file to your SD card. Copy the file to your laptop, then write a short Python program to read it and display its contents. (Week 2, by end of Friday class)

2d. Finish installing all the parts on your breadboard required for your project's data logger. (Week 3, by beginning of Friday class)

2e. Register an Autodesk user account, then visit the TinkerCad website. (Week 3, by beginning of Friday class)

3a. Write a single bare-bones program that read all your project circuit's sensors and writes data to a microSD file. (Week 3, by end of Friday class)

3b. Write a single bare-bones Python data analysis program that generates histograms and plots of environmental data read by your BME680. Calculate means and RMS widths for these quantities. (Week 3, by end of Friday class)

3c. Log in to Autodesk and download EAGLE. (Week 4, by midweek group conference with course staff)

4a. Finish writing a reasonably sophisticated DAQ and use it for a quick field test of your devices. (Week 4, by end of Friday class)

4b. Analyze your field test data, generating the plots and calculations that you expect to appear in your ultimate report. (Week 4, by end of Friday class)

4c. Install breakout boards on your PCB and test it. (Week 5, by midweek group conference with course staff)

5a. Perform a longer set of field tests and run them through your analysis. (Week 5, by beginning of Friday class)

5b. In consultation with course staff, refine your offline analysis. (Week 5, by end of Friday class)

5c. Finish PCB and transition to using it for more field test data; verify that PCBs function as expected. (Week 5, by end of Friday class)

5d. Use TinkerCad to design personalized covers for your PCB cases. (Week 5, by end of Friday class)

5d. Use TinkerCad to design personalized covers for your PCB cases. (Week 5, by end of Friday class)

6a. Take all the data that you think you'll need for your project. (Week 6, by end of Friday class)

6b. Verify that your data are valid: analyze them. (Week 6, by end of Friday class)

7a. Analyze production data and discuss your conclusions with course staff. (Week 7, by end of Friday class)

7b. Draft a modified run plan if appropriate, take more production data. (Week 8, by midweek group conference with course staff)

8a. Develop a detailed data analysis including cross calibration techniques, and run all your data through it. (Week 8, by end of Friday class)

8b. Write brief outline of a possible project report, discuss with course staff. (Week 8, by end of Friday class)

9-10. Write and submit "nearly final" draft of project report. (Week 10, by start of Friday class)

11-12. Rewrite and submit "final" project report. (Week 12, by start of Friday class)

13-14a. Prepare PowerPoint project presentation. (Week 14, by start of Friday class)

13-14b. Prepare and submit final project report. (Week 14, by start of Friday class)

In place of a final exam, I will require you and members of your team to generate two documents, with all team members as coauthors:

- A 30-minute PowerPoint presentation describing your project: the measurement(s) you've made, and the reasons for so doing; the hardware and software you've built to perform these measurements; your fieldwork and calibrations; your analysis and conclusions. Your intended audience comprises your Physics 371 classmates. Presentations are due at the start of class, week 14; you will present during the 14th week of the term.
- A written report of approximately 20 pages (single spaced), describing your measurements and conclusions; this is essentially the same information that you will write into your PowerPoint presentation. However, your report should be aimed at an audience that is outside the university community. For example, if you've measured the anomalous transverse accelerations of Amtrak trains, your audience might be the Illinois Department of Transportation. Your report is due in stages; see the course calendar for details.

The grading will be similar to Physics 298owl: if you work hard, are clever, come to all class activities, and do a good job on your hardware, software, and reports you will receive at least an A. But if you miss class, miss conferences with me, are late uploading your diary files, miss milestones, or do not do a good job on your project or reports, I will hammer you.

If you miss class for a legitimate reason, you must submit documentation to Kate Shunk in the Undergraduate Physics Office or upload it through the excused absence portal linked to the "Course policies" page on the p371 web site. Take note of the deadlines for submission of your documentation! I will expect you to make up the work you didn't accomplish.

Your obligations include working in a safe manner: ***always*** wearing eye protection when soldering and ***always*** washing your hands soon after handling metallic objects such as header pins or solder.

Course components used to calculate your course grade:

- Short in-class group presentations on technical issues pertaining to your project;
- Quality and efficiency of your group's collaborative interaction;
- Quality of your project hardware and data acquisition work;
- Quality of your offline analysis software work;
- Compliance with schedule milestones and deadlines;
- Project PowerPoint presentation;
- Project report.

Your grade will be based almost entirely on the semester's work, including your electronic diaries, and the final project reports and associated material.

***Course staff***

Instructor: George Gollin, Loomis 437d, (217) 333-4451, g-gollin@illinois.edu.
Teaching Assistant: Tony Mirasola, aem8@illinois.edu

Office hours: On demand, via Zoom, by appointment. Each team will also have a 30-minute face-to-face meeting with course staff every week so we can monitor your progress.

***Course calendar***

| what | day | date | comments |
|---|---|---|---|
| semester begins | Monday | August 22 | |
| week 1 | Friday | August 26 | |
| week 2 | Friday | September 2 | see Celia Eliot's presentation on communicating clearly |
| week 3 | Friday | September 9 | |
| week 4 | Friday | September 16 | |
| week 5 | Friday | September 23 | |
| week 6 | Friday | September 30 | |
| week 7 | Friday | October 7 | |
| week 8 | Friday | October 14 | |
| week 9 | Friday | October 21 | Celia Eliot: “Writing clearly” |
| week 10 | Friday | October 28 | Nearly-final draft of report due |
| week 11 | Friday | November 4 | |
| week 12 | Friday | November 11 | Final draft of report due |
| week 13 | Friday | November 18 | |
| Fall break, November 19 – 27 | | | |
| week 14 | Friday | December 2 | Rewritten report and PPT presentation due |
| end of term conferences | Monday – Wednesday | December 5 – December 7 | |
| reading day | Thursday | December 8 | |

***Detailed syllabus; see the list a few pages back for milestones.***

*Week 1, in class: see the Week 1 supplemental material writeup.*

- Form up into research groups of three or four people and begin discussing which project you’ll pursue. Decide which sensors and other devices you’ll use.
- Schedule a time for the team’s weekly meeting with professor and TA.
- Install the Arduino programming IDE.

- Plug the Arduino into a USB port on your laptop, download the blinking LED sample program. Confirm that the LED really does blink. (You'll want to create a sensibly named folder to hold your Arduino programs.)
- Modify the blinking LED program to flash your first and last initials in Morse code.
- Duct-tape your Arduino to your breadboard and install a BME680, along with whatever libraries are necessary. Play with the BME680, taking note of its sensitivity to changes in atmospheric pressure.
- Install other components, as described in the Week 1 supplemental writeup.
- Install the latest version of Anaconda Python.
- Select/assign a group to report next week about the I2C communication protocol.

*Post-class assignment (try to finish all of this before you meet with us next week)*

- Formulate a plan of action with the members of your team. I want all of you to be involved with all flavors of activity: writing Arduino code, generating schematics to represent your devices, and so forth. But it is fine for one person to take the lead on, say, managing the code that interrogates the GPS package. Discuss with your group members your tentative plans for executing your project: who will focus on what; which sensors you will need; who you might need to contact for permission to enter their space for your measurements.
- Finish any unfinished tasks on the week's "in-class" list.
- Read the entire (printed) "Introduction and Syllabus" and other Physics 371 documents I've given out in class.

*Week 2, conference with the professor + TAs*

- Discuss your project plan, including who you might contact for permission to, for example, install atmospheric methane detectors in the UIUC barns.
- Show us what your breadboard circuits can do.
- Describe sharing of project responsibilities among group members.

*Week 2, in class*

- Hear a group's "informative report."
- Install a DS3231 RTC and set its time.
- Install a GPS module and use it to set the RTC. Depending on where you are working, you may need to step outside when it's time to test your GPS board.
- Install everything you'll be using for your measurements onto the breadboard, but do not yet install any of the wiring required for any of the new breakout boards. Place a 0.1μF capacitor between +5V and ground close to each breakout board's power pins.
- One breakout board at a time, attach power and ground wires, and any other connections that are necessary to drive the board. (Note that you can "daisy chain" the I2C connections from board to board.) Find some demo software and make the Arduino talk to the board you've just wired. After it works, go on to the next breakout board.

- Select/assign a group to report next week about the BME680 T/P/RH sensors.

*Post-class assignment*

- Finish as many unfinished tasks on the week's "in-class" list as possible. (You should try to finish loading and wiring your breadboard, and talking to the individual sensors.)
- Register an account with Autodesk.
- Log in to TinkerCad and make sure it recognizes your Autodesk account. Find a simple design for something amusing on the TinkerCad site and export it to an STL file after you place your initials on the object.

*Week 3, conference with the professor + TAs*

- Keep me apprised of your progress, and how you have decided to share the responsibilities for the various tasks needed to advance your project.
- Show us what you breadboards can do.

*Week 3, in class*

- Hear a group's "informative report."
- Informal (oral) progress reports from each group: describe your project to the class, and tell us your thoughts on how you'll approach it.
- Do initial DAQ code development, and start on offline Python work.
- Download EAGLE from the Autodesk site.
- Select/assign a group to report next week about how a successive approximation ADC works.

*Post-class assignment*

- Finish as many unfinished tasks on the week's "in-class" list as possible.
- Log into your Autodesk account, then install the EAGLE schematic capture/PCB IDE.
- Download and open the schematic for my version of the data logger, and delete everything except the components you will be using.
- Work on your DAQ code, and make sure it can write to the microSD card memory.

*Week 4, conference with the professor + TAs*

- Keep us apprised of your progress. The first version of your DAQ should be fairly far along by the time we meet.

*Week 4, in class*

- Hear a group's "informative report."
- Do a quick field test of all your breadboards.
- Analyze the field test data and make plots and histograms.
- Select/assign a group to report next week about electret microphones.
- Plus components into the PCB and begin testing it.
- Last ten minutes: whole class group discussion. How is it going? What's too hard/too easy? What are your thoughts about the field work you'll be doing? What kind of

technical support might I provide to make your work go more smoothly? Have you made contact yet with anyone to learn the rules concerning entering into their environment to make measurements?

*Post-class assignment*

- Finish as many unfinished tasks on the week's "in-class" list as possible.
- Log into TinkerCad, find my sample case for the fall 2019 data logger, and copy it to your own account, then modify it to suit your tastes and preferences.
- Try to finish your DAQ.

*Week 5, conference with the professor + TAs*

- Progress reports from you, including how far you got with TinkerCad.
- Show us that your breadboard works, and show us how far along you've gotten your PCB.

*Week 5, in class*

- Hear a group's "informative report."
- Finish your DAQ.
- Finish PCB checking. It should be electrically equivalent to your breadboard.
- Field test the PCB; analyze the data.
- Refine your offline analysis.
- Use TinkerCad to finish any design work needed for your project.
- Select/assign a group to report next week about GPS.

*Post-class assignment*

- Run (and analyze) a longer field test.
- Generate plots of your test data; discuss possible modifications to your run plan based on what you see.
- Finish your TinkerCad work so I can generate a case for you.

*Week 6, conference with the professor + TAs*

- Progress reports from you, especially what you found in your field tests.
- Show us that your PCB works, and show us your case design.
- Tell us about your run plan and take production data.

*Week 6, in class*

- Five-minute (PowerPoint) progress reports from each group: describe your field tests.
- Hear a group's "informative report."
- Finish your DAQ and offline analysis code.
- Select/assign a group to report next week about GPS.

*Post-class assignment*

- Take data: all the data you think you'll need for your project.
- Generate analysis plots from your production run data.

*Week 7, conference with the professor + TAs*

- Progress reports from you, especially what you found in your data.
- Show us your offline analysis results and discuss possible conclusions to be drawn.
- Discuss your plans to determine calibrations for your sensors.

*Week 7, in class*

- Hear a group's "informative report."
- Refine offline analysis and consider modifying your run plan.
- Take more production data if appropriate.
- Select/assign a group to report next week about the Atmel ATmega2560 microcontroller.

*Post-class assignment*

- Perform whatever auxiliary measurements are necessary to calibrate your devices.
- Prepare a ten minute status report to present to the class about your field tests.

*Week 8, conference with the professor + TAs*

- Discuss your status report, calibrations, tentative conclusions, and online/offline code progress.
- Discuss possible modifications to your run plan, if it is warranted by what you found.

*Week 8, in class*

- Hear a group's "informative report."
- Ten minute status reports presented to the class, along with group discussions of calibrations, analysis techniques, and systematic uncertainties.
- Continue refining your offline analysis and conclusions.
- Begin outlining your project report.
- Select/assign a group to report next week about how an LSM9DS1 "9-axis" accelerometer works.

*Post-class assignment*

- Try to finish your analysis and draw your preliminary conclusions.
- Outline your project report paper. Your intended audience will be readers who are not familiar with Physics 371.

*Week 9, conference with the professor + TAs*

- Discuss your analysis, conclusions, and project report outline with us.

*Week 9, in class*

- Hear a group's "informative report."
- Write the first draft of your paper, discussing the details inside your group as you go: who writes which sections, and so forth.
- Further refine your offline analysis and conclusions if appropriate.
- Select/assign a group to report next week about "Pulse Width Modulation."

*Post-class assignment*

- Finish a substantial, nearly final draft of your project report.

*Week 10, conference with the professor + TAs*

- Give us a progress report on your paper.

*Week 10, in class*

- Hear a group's "informative report."
- Peer-reading/evaluating/commenting/correcting of your drafts.
- More work on paper and analysis as necessary.
- Select/assign a group to report next week about the MLX90614 infrared sensor.

*Post-class assignment*

- Write your "final draft" of the paper. You should think of this as the ultimate, polished, final version.

*Week 11, conference with the professor + TAs*

- Paper status

*Week 11, in class*

- Hear a group's "informative report."
- More analysis refinement and project report work.

*Post-class assignment*

- Finish writing the "final draft" of the paper.

*Week 12, conference with the professor + TAs*

- Paper status

*Week 12, in class*

- Peer-reading/evaluating/commenting/correcting of your project reports.
- Discuss in your group changes to be made to your paper.

*Post-class assignment*

- Begin rewriting your very final project report.

*Week 13, conference with the professor + TAs*

- We'll discuss corrections/improvements/rewrites you should make to your paper.

*Week 13, in class*

- Analysis tweaks and paper rewrites.
- Work on your project PowerPoint presentation.

*Post-class assignment*

- Finish rewriting your project report.
- Finish your PowerPoint presentation, ~30 minutes in length, describing your results to your classmates.

*Week 14, conference with the professor + TAs*

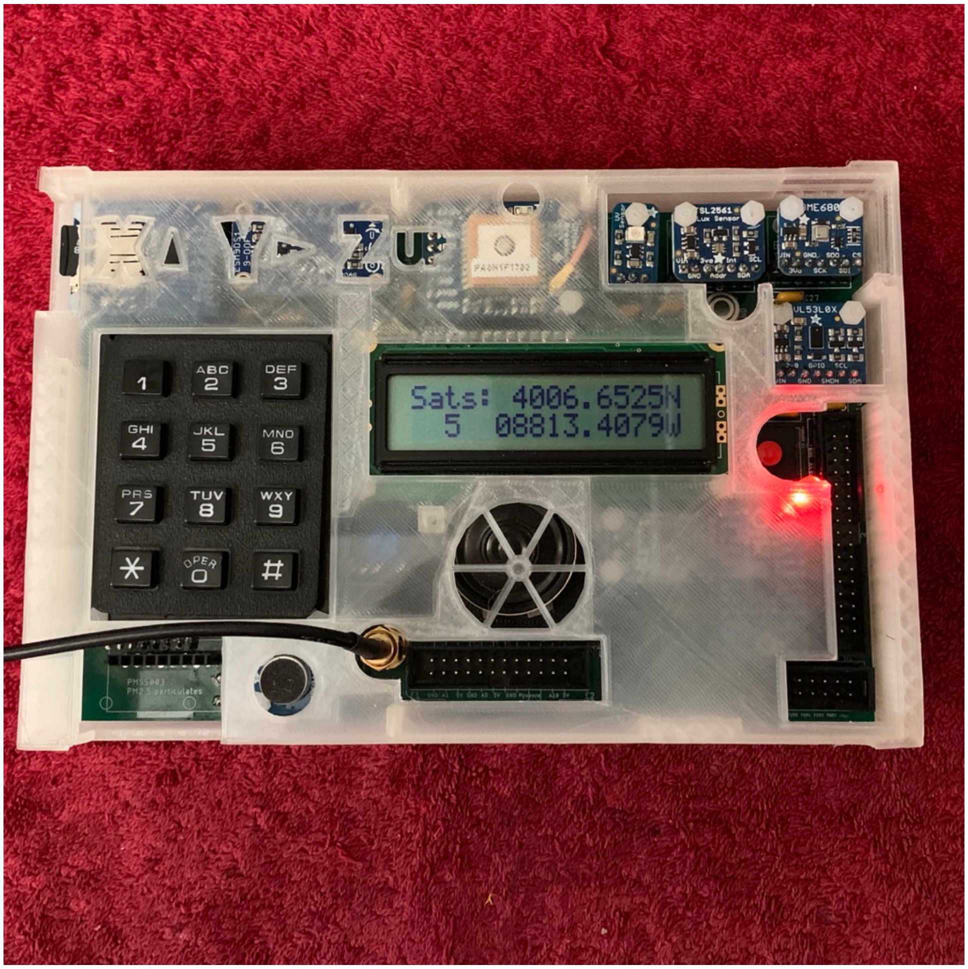
- Paper and PPT discussions.

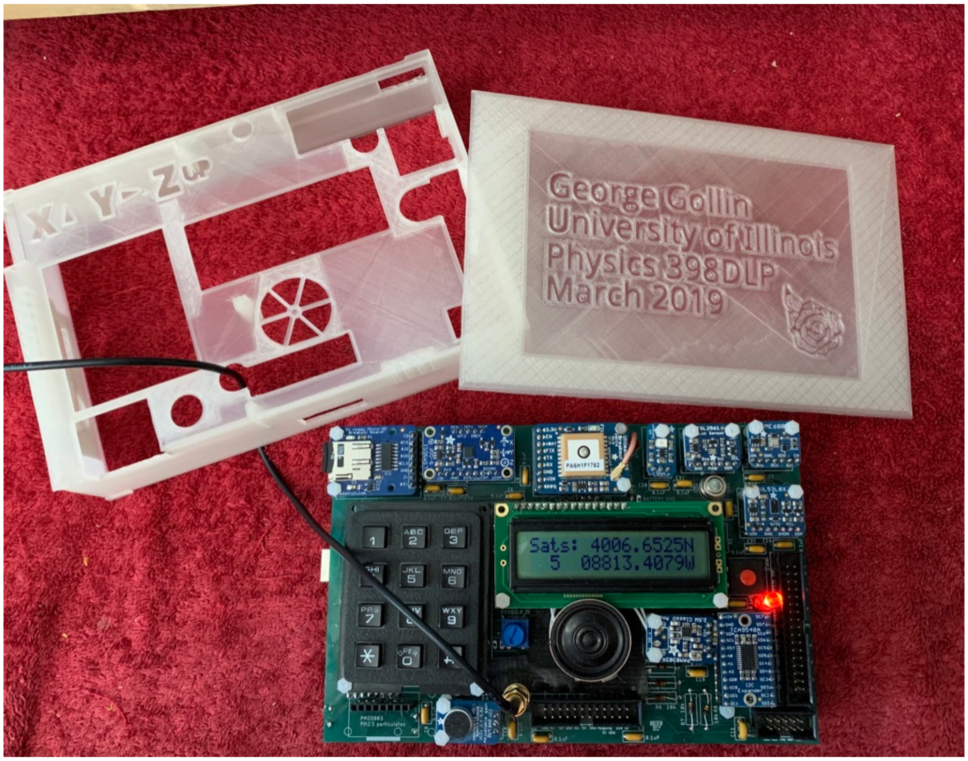
*Week 14, in class*

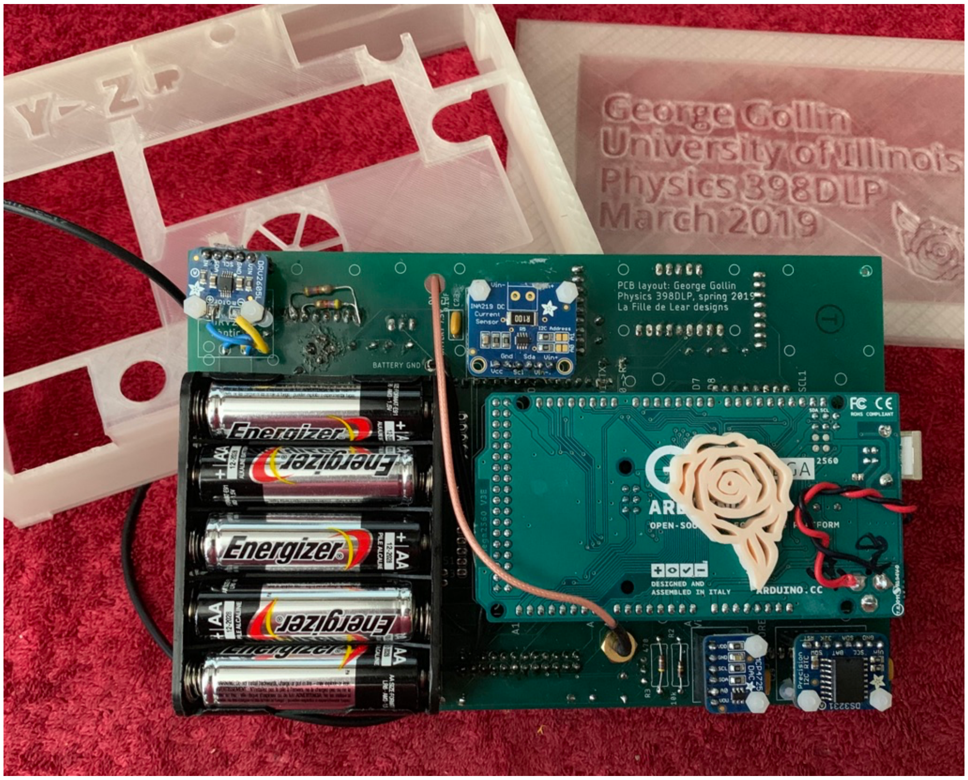
- PPT presentations by each group

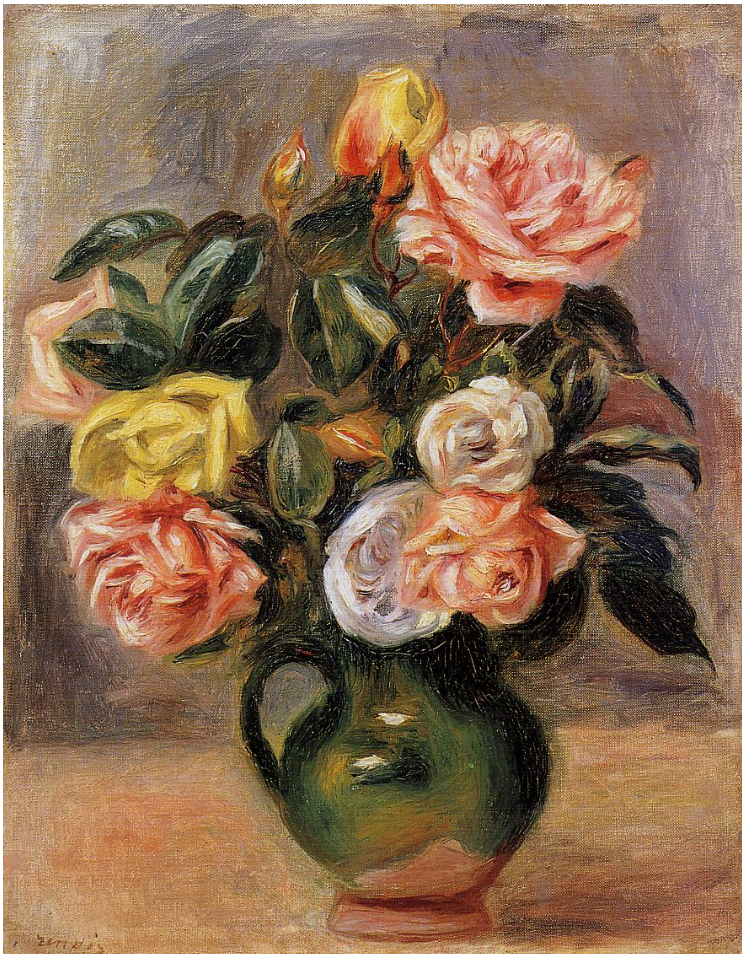
***A fully loaded data logger***

Here are photos of my version of the data logger, onto which I've installed pretty much everything we have.




X Y Z UP
George Gollin
University of Illinois
Physics 398DLP
March 2019
Sats: 4006.6525N
5 08813.4079W


Y Z UP
George Gollin
University of Illinois
Physics 398DLP
March 2019
PCB layout: George Gollin
Physics 398DLP, spring 2019
La Fille de Lear designs
Energizer
ARDUINO.CC



https://1.bp.blogspot.com/-XhQVLp4bA34/TzAnlNdwZHI/AAAAAAAABCbI/NCj49D2uP8w/s1600/Pierre+Auguste+Renoir+Flowers+-+Tutt%2527Art%2540+%252817%2529.jpg

# Physics 371

Design Like a Physicist

Fall 2022

George Gollin

University of Illinois at Urbana-Champaign

## Week 1: Organizations, Distributions, and Installations

☛ supplemental material ☚

# Week 1: Organizations, Distributions, and Installations

Goals for this week ....3
Groups, projects, electronic diaries ....3
Distribution of stuff! ....4
Installing the Arduino programming IDE; running code ....7
In-class exercise/milestone 1a: blink code ....13
Breadboarding! ....13
In-class exercise/milestone 1b: BME680 ....15
In-class exercise and milestone 1c: version zero (due at the start of class 2) ....18
How to read a schematic ....18
The second-to-last thing this week: install the Anaconda Python IDE ....20
In-class exercise and milestone 1d: formulate a plan! ....20
Wrap up: informative presentation for next week (need a group to volunteer!) ....21
Appendix: soldering ....21
Safety first! ....21
The Proper Solder Joint ....21
Types of Solder Joints ....22
(NOT) The Proper Solder Joint ....23
A word to the wise ....24

### *Goals for this week*

- You will organize your group and choose a project.
- You will assemble a starter circuit on a breadboard and use the Arduino Integrated Developers Environment (IDE) to program and test it.
- You will install the Anaconda Python IDE.
- You will begin developing a project plan, and signing responsibilities among your group members.

### *Groups, projects, notebooks*

Please form up into research/project groups, discuss which project you might prefer to undertake, and find a time that all members of your group can meet with the course staff for 30 minutes each week. My preference is sometime Monday (10 am – 4 pm), Tuesday (10 am – 4 pm), Wednesday (10 am – 4 pm), or Thursday (10 am – 4 pm).

Each of you will assemble a breadboarded version of a device, then construct a printed circuit board-based version that is electrically equivalent.

Open up your notebook, and add to your account of the afternoon's activities as you work. Brainstorm about which sensors and other hardware you'll need; see the following list. This semester the notebook will be an informal tool, and you are to keep it as a handwritten physical lab notebook. FOR SURE you'll want to keep track of your work, and your revelations, so you can return to them at a later date. We won't be looking at your notes, but if I ask you something like “where did you go to find that cool piece of demo software?” you had better be able to give me an answer based on your notes.

***Distribution of stuff!***

I have an alarmingly large amount of stuff available for you. Some of the following I’ve already packed in the parts/tools kits I’ll distribute in class.

| **part** |
|---|
| 16 x 2 LCD CFAH1602B-NGG-JTV |
| 2 x AAA & 1 x AAA battery holders |
| 3 x 4 keypad (#3845) |
| 5 x AA battery holder, Adafruit 3456 |
| 74HC137 3-8 decoder/demultiplexer (TTL) |
| 74HC157N quad 2 input (TTL) multiplexer |
| Adalogger Feather M0, Adafruit 2796 |
| ADXL326 accelerometer (#1018) |
| ALLPOWERS 2.5W, 5V photovoltaic (solar) cell |
| Arduino Mega 2560 |
| BME 680 T/RH/P/VOC (#3660) |
| capacitive touch sensors, Adafruit 1374 |
| Diodes: 1N5817 Schottky |
| DRV2605L haptic vibrator driver |
| DS3231 real time clock (#3013) |
| electret microphone with amplifier (#1063) |
| EMG sensors, electrodes, USB isolators, Adafruit 2699, 2773, 2107 |
| Feather M0 with 900 MHz radio (#3178) |
| Featherwing boards: RTC and SD (#2922) |
| Featherwing RTC + SDA, Adafruit 2922 |
| Force-sensitive resistors, Adafruit 166 |
| Gas sensors: methane and others |
| GPS Antenna - External Active Antenna (#960) |
| GUVA Analog UV sensor (#1918) |
| Haptic feedback motorized vibrator |
| high temperature thermocouples (#3245) |
| INA219 battery voltage/current sensor (#904) |
| IR receiver, TSOP38238, 38 kHz, Adafruit 157 |
| K thermocouple; therm. amp, Adafruit 3245 & 269 |
| L3GD20H Triple-Axis Gyro Breakout (#1032) |

| |
|---|
| low dropout voltage regulators, eg LD1117-3.3 TO-220 (#2165) |
| LSM9DS1 accelerometer/magnetometer/gyroscope 0x1E (#3387) |
| MAX 31855 thermocouple amplifiers (#269) |
| MCP23008 I2C port expander, Adafruit 593 |
| MCP4725 DAC, 12-bit, I2C (#935) |
| methane sensor: MQ4, on breakout board |
| microphones: Adafruit 1713, SparkFun BOB-12758 |
| MicroSD card breakout (#254) |
| MicroSD memory card 8 GB SDHC (#1294) |
| MiCS5524 CO, alcohol, VOC gas sensor, Adafruit 3199 |
| Mini Metal Speaker w/ Wires - 8 ohm 0.5W (#1890) |
| Mini TTL serial JPEG camera, Adafruit 1386 |
| Mini-USB spy camera connections, Adafruit 3202 |
| MLX90614 IR sensor (#1748) |
| MLX90640 24 x 32 IR 55° x 35° camera, Adafruit 4407 |
| MMA8451 ±2 g accelerometer |
| momentary capacitive switch |
| MyoWare EMG muscle sensor |
| P164 GP2Y0A21YK0F IR distance sensor |
| PAM8302 Audio Amplifier (#2130) |
| PAM8302 audio amplifier, Adafruit 2130 |
| pH sensor, DFRobot |
| photo interrupter, T-slot, Adafruit 3985 |
| PM2.5 PMS5003 airborne particulate monitor (#3686) |
| QS-FS01 anemometers |
| Radio Feather M0 LoRa, Adafruit 3178 |
| red LEDs (#297) |
| TCA9548A mux I2C multiplexer |
| thermal camera (#4407) |
| Tiny camera Adafruit “mini spy camera” part 3202 |
| TMP36 analog medium temperature thermometers (#165) |
| TMP36 analog temp. sensor, Adafruit 165 |
| TPS7133 3.3V regulator |
| Tricolor flashing LEDs (#680) |
| TSL2561 Digital Luminosity/Lux/Light sensor (#439) |
| TSL2591 Digital Luminosity/Lux/Light sensor (#1980) |
| TTL serial camera (#1386) |
| ultimate GPS breakout board (#746) |
| USB DIY connectors |

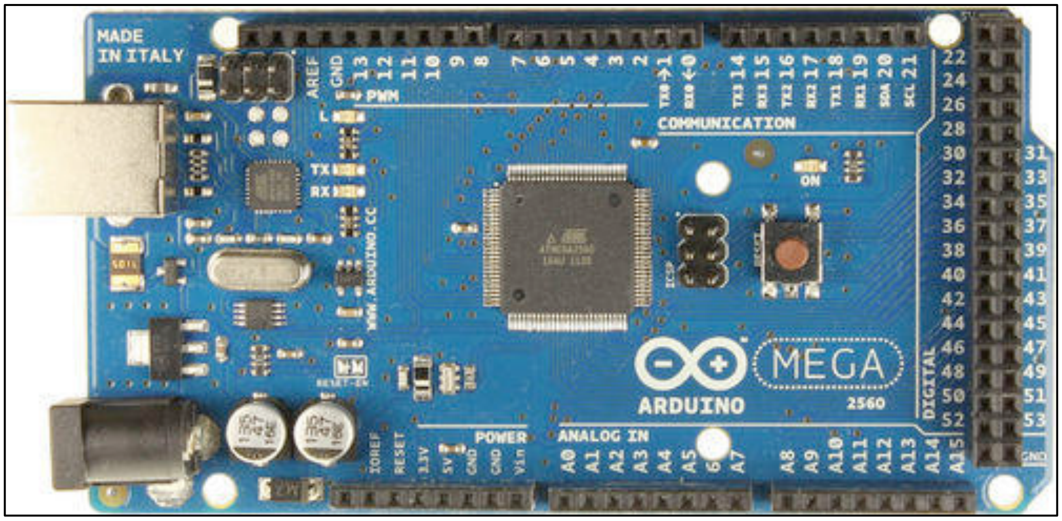
| |
|---|
| USB isolator |
| VL53L0X Time of Flight Distance Sensor (#3317) |
| voltage regulator: 3.3V, 800mA, and heat sinks, Adafruit 2165 & 977 |

Because of the potential issues surrounding covid-19 I’ve transferred the Physics 371 parts/tools library from Loomis to my basement! Here it is:

### *Installing the Arduino programming IDE; running code*

Go to the Arduino website https://www.arduino.cc/ and navigate to https://www.arduino.cc/en/Guide/ArduinoMega2560. Download and install the Arduino Desktop IDE (Integrated Development Environment) on your laptop. See information in the Physics 371 "C++ and Python Primer" for more details about how to do this.

The heart of your data logger will probably be an Arduino Mega 2560 microcontroller board, shown here.



Arduino Mega 2560

It’s a remarkable little gizmo, featuring an Atmel Atmega2560 microcontroller (built by Microchip Technology, Inc.) running at 16MHz. The Atmega2560 has 256kB of flash memory in which your program will reside, along with 8kB of SRAM (static random access memory) in which will live the variables your program modifies as it executes. There are 16 analog inputs that feed an internal multiplexer whose output drives a 10-bit successive approximation analog to digital converter (ADC). See https://store.arduino.cc/arduino-mega-2560-rev3 for more details.

Some of the projects might involve a different processor: I have been developing systems using a pair of Adafruit devices, both of which employ Microchip ATSAMD21G18 microcontrollers. The SAMD21 runs at 48 MHz, features a 12-bit ADC, and has 32 kB of SRAM, four times more than the Atmega2560. It also has a 10-bit digital to analog converter (DAC), which the 2560 does not.

The Adalogger M0 includes a built-in microSD memory holder, while the Feather M0 with RFM95 LoRa radio has a built-in 915 MHz long range radio transceiver. Both are very cool.

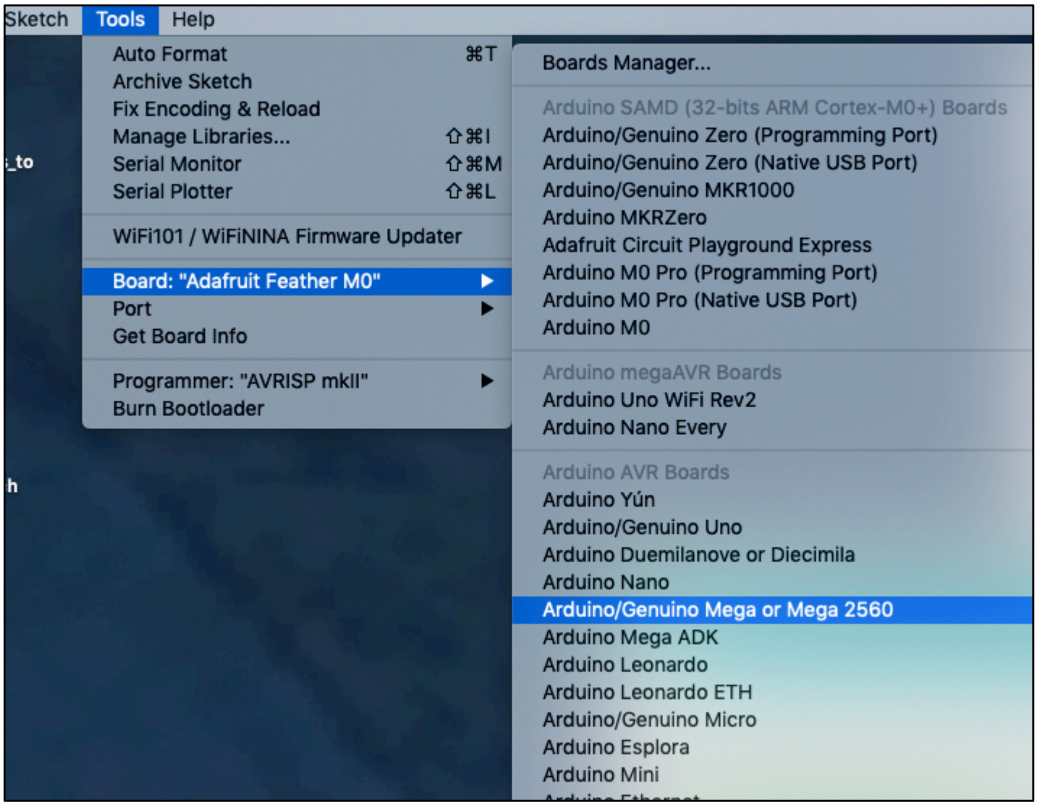
The Arduino IDE is quite a bit simpler than Anaconda’s iPython IDE. Most of what you will see on your screen is an editor window in which you will create/modify C++ programs that you will compile and upload to the Arduino. See the screen shot, below. You’ll write and compile programs using the IDE, then upload them to the Arduino through a USB cable.

There isn’t a debugger, so you’ll be forced to print things to a “serial monitor” screen to keep track of what’s going on (and going wrong) in the code executed by the Arduino.

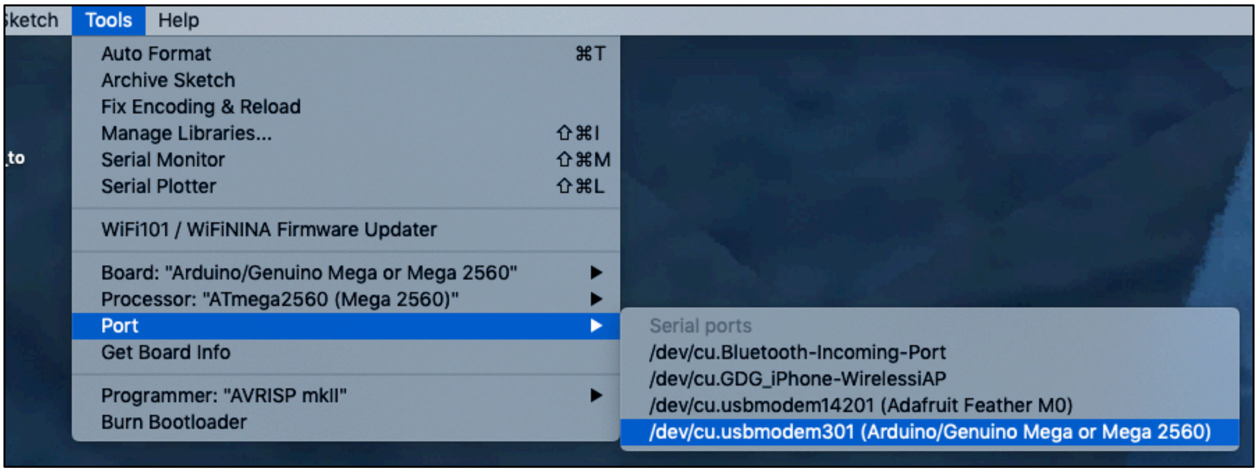
Something to keep in mind: the Arduino runs code that is very much like C++, with some minor differences. But the structure of a program is constrained: there are always two specific functions that must be included in a program. The first is called “setup”; it takes no arguments and does not return a value. It is the first routine in a program that executes. The next is called “loop.” It too takes no arguments and does not return a value. Upon completion of setup, the

Arduino goes straight into the loop function. It executes loop over and over, jumping back into it each time the function completes.

Do this: plug the Arduino into a USB port on your laptop, then fire up the IDE. Open the Blink example, compile and upload it, and see if your processor will talk to you. You'll need to make sure the IDE knows what kind of processor you are using: a Mega 2560.

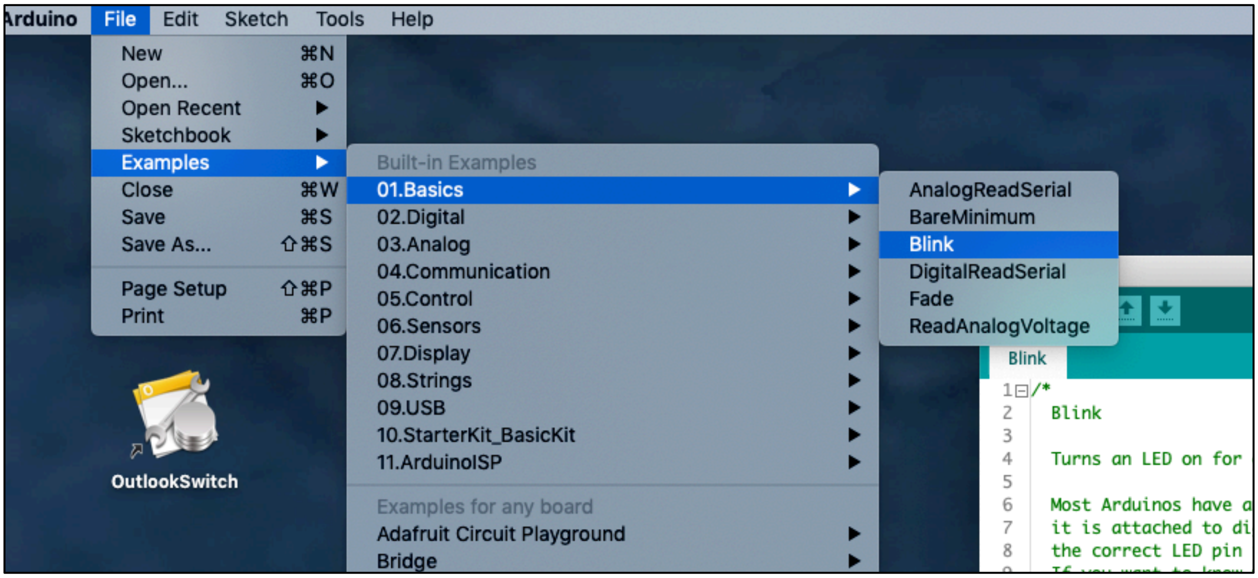


You might also need to tell the IDE which "port" to use:


ketch
Tools
Help
Auto Format ⌘T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... ⇧⌘I
Serial Monitor ⇧⌘M
Serial Plotter ⇧⌘L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino/Genuino Mega or Mega 2560"
Processor: "ATmega2560 (Mega 2560)"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
to
Serial ports
/dev/cu.Bluetooth-Incoming-Port
/dev/cu.GDG_iPhone-WirelessiAP
/dev/cu.usbmodem14201 (Adafruit Feather M0)
/dev/cu.usbmodem301 (Arduino/Genuino Mega or Mega 2560)

Since I had a second processor plugged into a USB port, the IDE saw both of them.

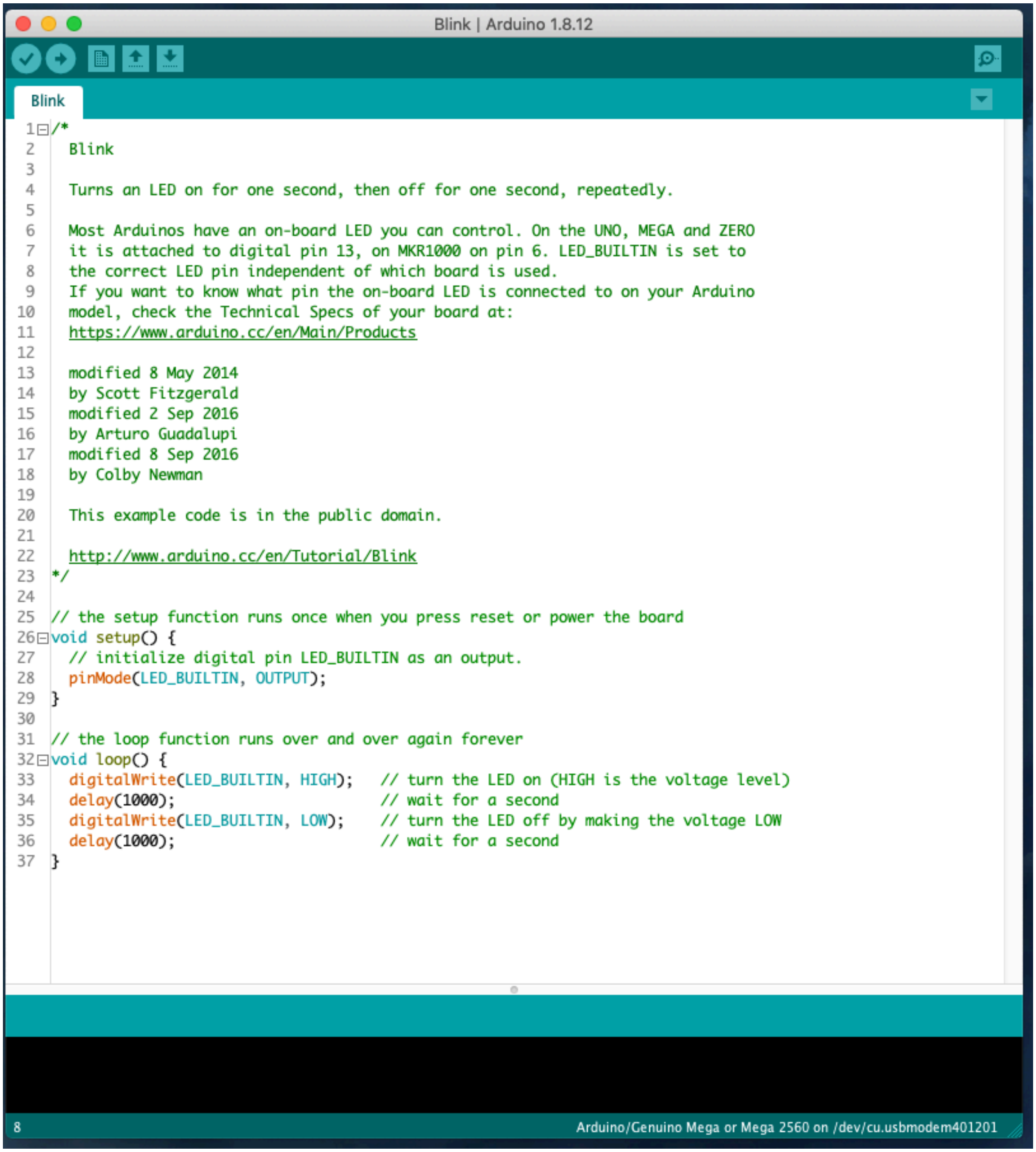
Now go to the examples menu and open the Blink program. See the following screen shot.



```
/*
  Blink

  Turns an LED on for

  Most Arduinos have a
  it is attached to di
  the correct LED pin
```

What the program does is pretty obvious. See the following screen shot to see how it looks in the IDE.

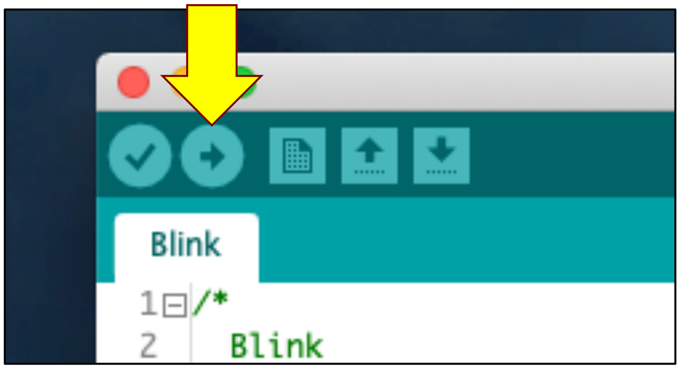
Good coding references are on the Arduino site: see https://www.arduino.cc/reference/en/ and https://www.arduino.cc/en/reference/libraries.



```
/*
  Blink

  Turns an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
  the correct LED pin independent of which board is used.
  If you want to know what pin the on-board LED is connected to on your Arduino
  model, check the Technical Specs of your board at:
  https://www.arduino.cc/en/Main/Products

  modified 8 May 2014
  by Scott Fitzgerald
  modified 2 Sep 2016
  by Arturo Guadalupi
  modified 8 Sep 2016
  by Colby Newman

  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Notice that some of the variables display as green text: LED_BUILTIN, HIGH, LOW, OUTPUT. These are quantities known to the compiler, with values substituted into the executable upon compilation. Some of them depend on the particular microcontroller the program will use. For example, the Arduino's red LED is connected to pin 13, so LED_BUILTIN takes the value 13.

You can compile the program  upload it to your Arduino by clicking the right-arrow button near the top of the window:

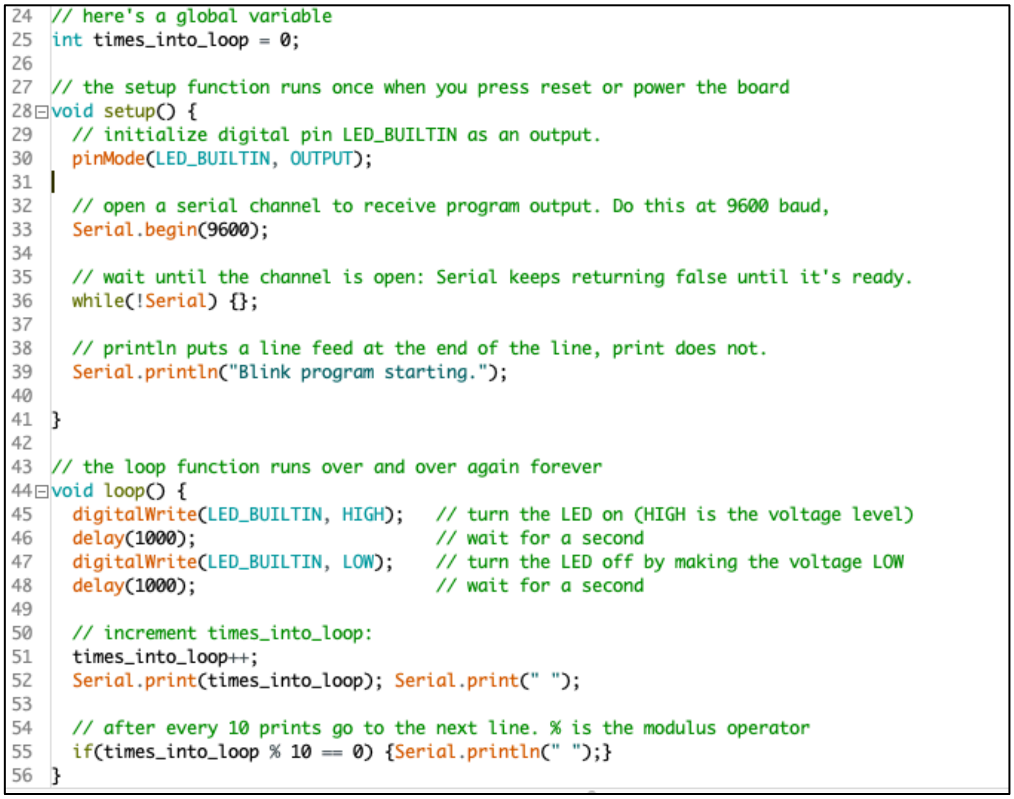


Note the presence of the setup and loop functions.

Some more technical commentary: many of the Arduino's "pins" are configurable: they can be defined to be digital inputs, or outputs, or analog inputs. The pinMode instruction defines the pin driving the red LED to be a (digital) output.

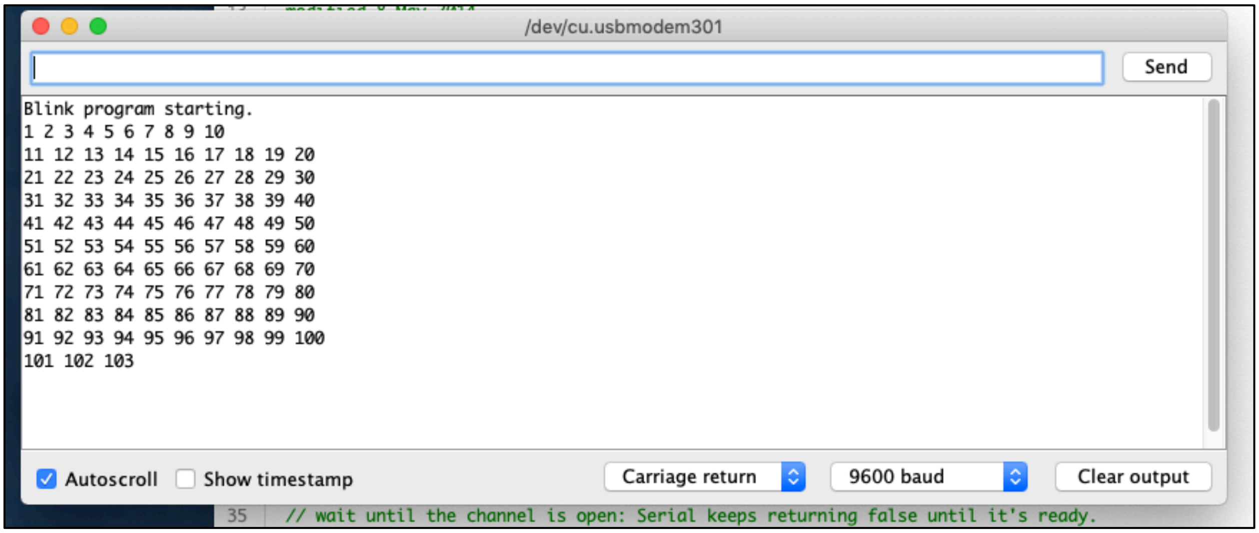
I have a schematic for an Arduino Mega 2560 linked to the course's "Code & design resources repository" web page.

Here's what you can do to view program output in a "serial monitor" window. First add some code to write to it:



```
// here's a global variable
int times_into_loop = 0;

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);

  // open a serial channel to receive program output. Do this at 9600 baud,
  Serial.begin(9600);

  // wait until the channel is open: Serial keeps returning false until it's ready.
  while(!Serial) {};

  // println puts a line feed at the end of the line, print does not.
  Serial.println("Blink program starting.");

}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second

  // increment times_into_loop:
  times_into_loop++;
  Serial.print(times_into_loop); Serial.print(" ");

  // after every 10 prints go to the next line. % is the modulus operator
  if(times_into_loop % 10 == 0) {Serial.println(" ");}
}
```

Then go to the IDE's Tools menu and select Serial Monitor. Make sure you've set the baud rate to 9600. Upload, and you are all set.



```
Blink program starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103
```

```
// wait until the channel is open: Serial keeps returning false until it's ready.
```

### *In-class exercise/milestone 1a: blink code*

Modify the blink program so that it blinks the initials (of your English/American name) in Morse code.

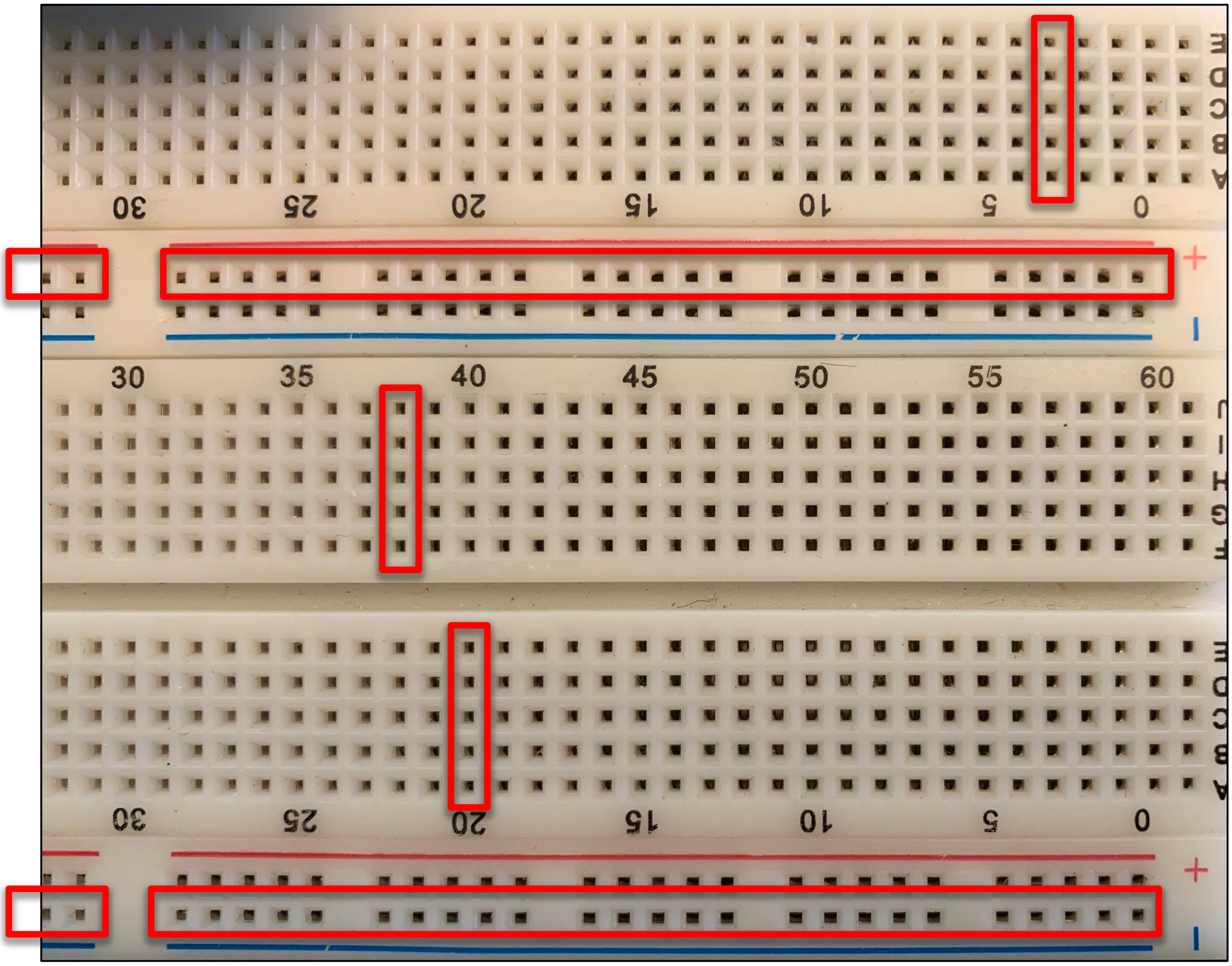
### *Breadboarding!*

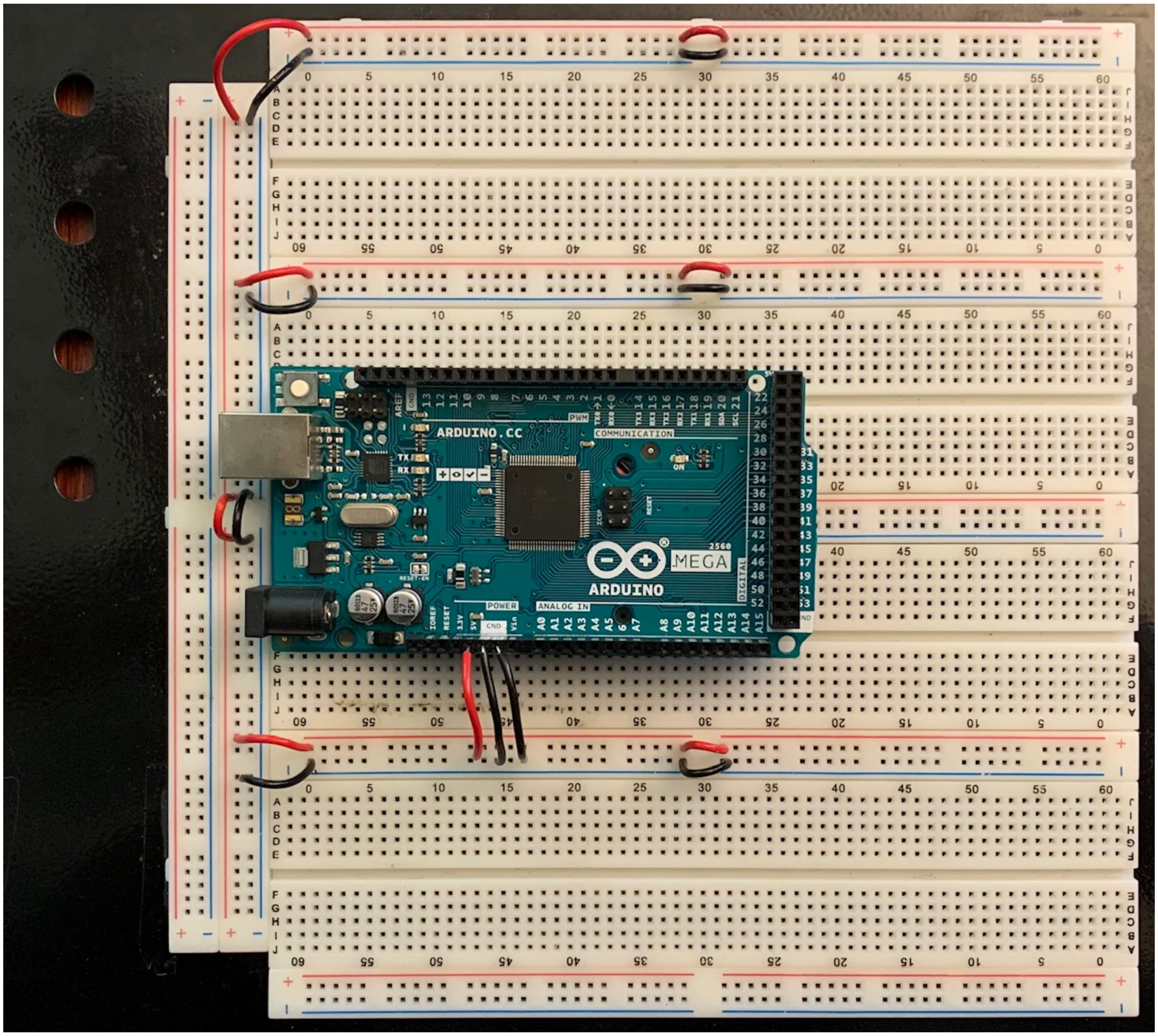
Let’s prep our breadboards. Please fasten an Arduino to your breadboard. I recommend duct tape (the baby sitter’s friend!); position the device to that it doesn’t cover any of the breadboard’s plastic structures that are used to hold components. If you don’t have any duct tape, improvise something from packing tape or Band-Aids.

Here’s how the holes are interconnected, underneath the plastic surface. The five holes in a column are connected, as shown in a few spots in the picture below. The 25 holes in a horizontal row are also connected.

In the photo after the close up I show a breadboard with an Arduino and connections between the Arduino’s 5V and grounds to the breadboard. The doubled connection of the Arduino grounds is not electrically necessary, but it does provide redundancy in case one of the ground wires falls out.

Be sensible in your choices of wire colors: always use red for 5V and black for ground. You’ll want to strip about 5 mm of insulation from each end of a wire when establishing your connections.

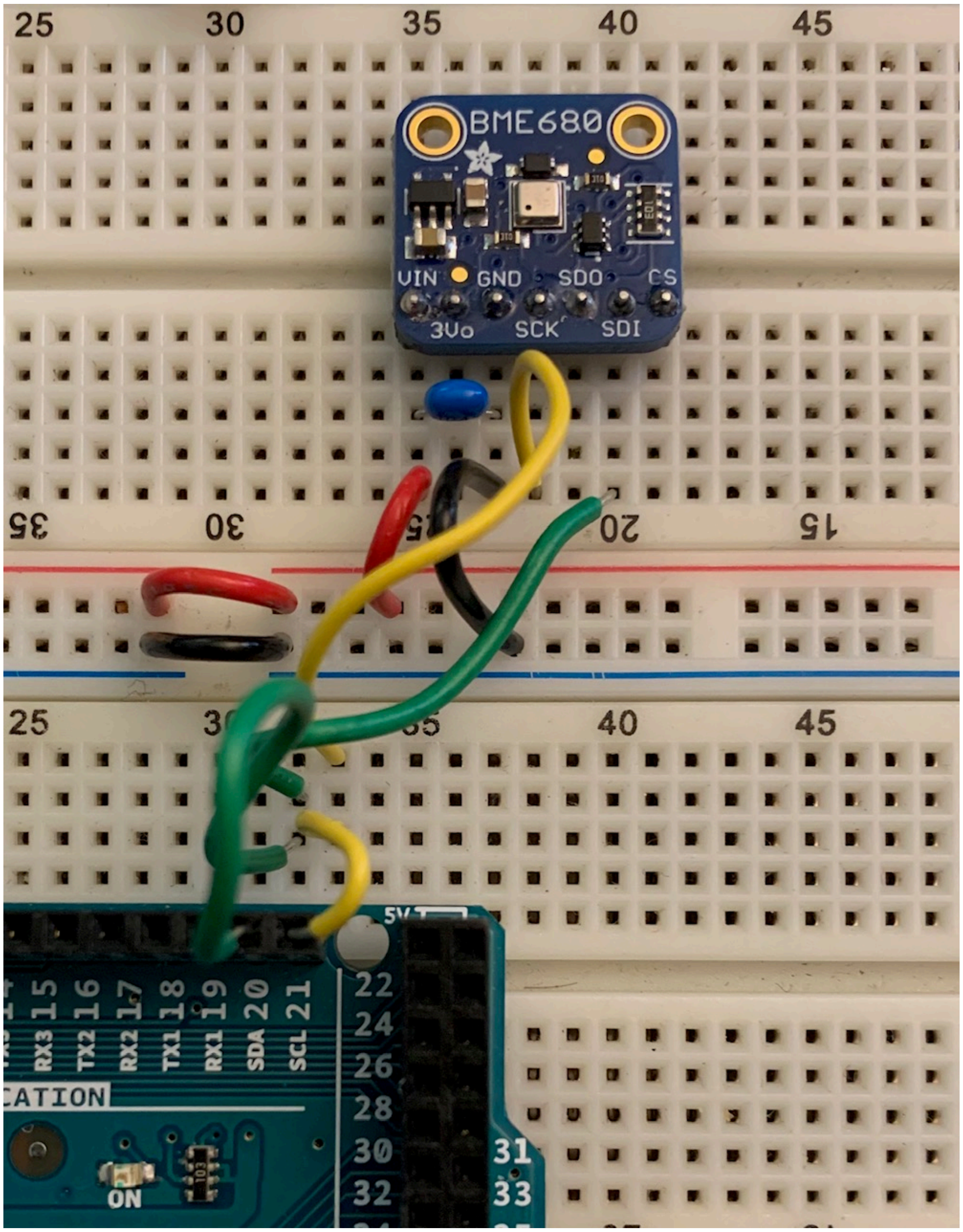

30
25
20
15
10
5
0
E
D
C
B
A
+
-
30
35
40
45
50
55
60
J
I
H
G
F
E
D
C
B
A
30
25
20
15
10
5
0
+
-



***In-class exercise/milestone 1b: BME680***

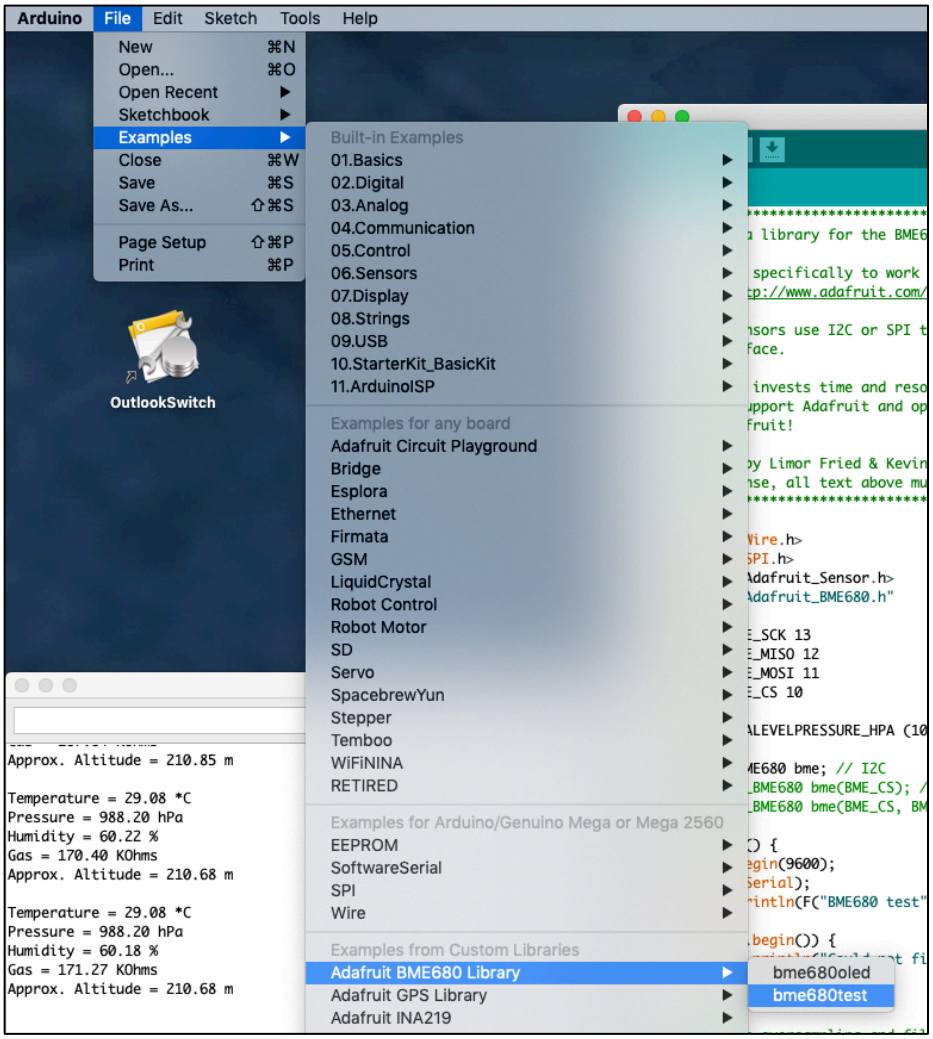
Most of our breakout boards were built by Adafruit Industries, a wonderful provider of small electronic packages intended largely for the hobbyist market. Go to the Adafruit site https://www.adafruit.com/ and find the BME680 page that mentions some of the supporting infrastructure available for you.

Install the BME680 onto your breadboard. (See https://www.adafruit.com/product/3660 and links therein.) You should power it using the Arduino's +5V and GND lines. Using sensible colors (red for +5V, black for ground, other colors for signal lines), connect GND to one of the Arduino's GND lines and VIN to one of the Arduino's 5V lines. Also connect the leads of a 0.1μF capacitor to the BME680's power and ground inputs.



We’ll let the device and the Arduino communicate using an I2C (“I Two C”: Inter Integrated Circuit) interface; set this up by connecting the BME680’s SCK (serial clock) pin to the Arduino’s SCL output (pin 21). Also connect the BME680’s SDI (serial data) to the Arduino’s SDA input (pin 20). You should leave unconnected the BME680’s 3Vo, SDO, and CS pins.

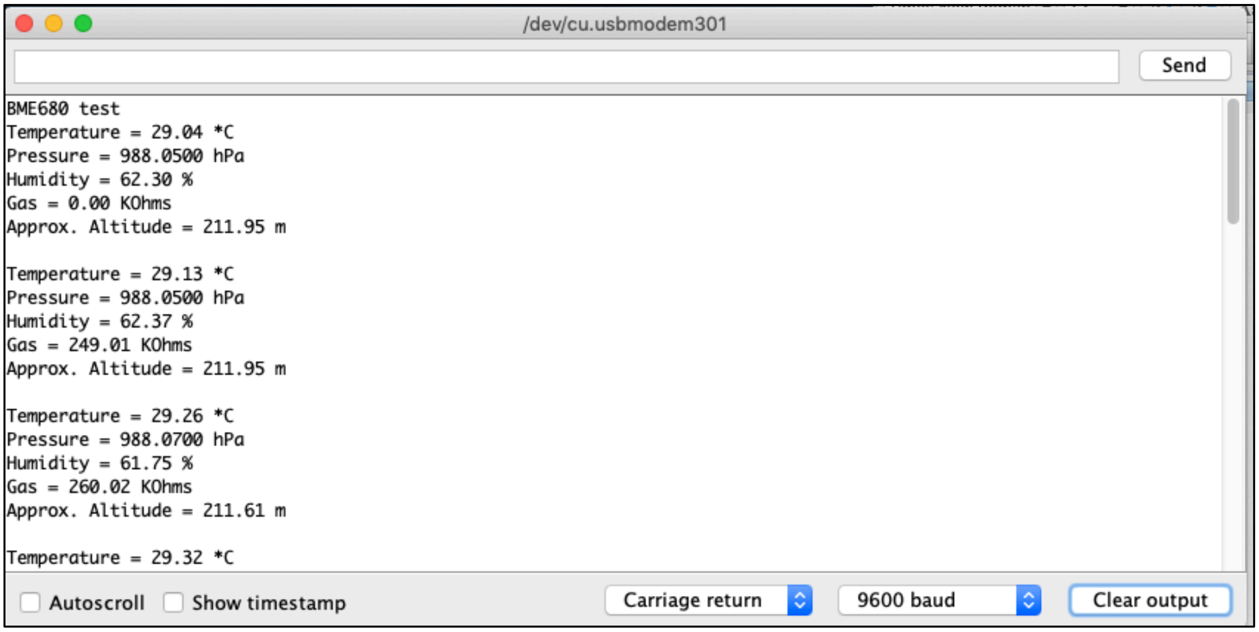
You’ll need to install one of Adafruit’s libraries to drive the BME680. See https://learn.adafruit.com/adafruit-bme680-humidity-temperature-barometic-pressure-voc-gas/arduino-wiring-test and scroll down to the section titled “Install Adafruit_BME680 library.” Follow the directions to install the library and upload to the Arduino the demonstration software. Then open, compile, and run the example program BME680test.



```
Approx. Altitude = 210.85 m

Temperature = 29.08 *C
Pressure = 988.20 hPa
Humidity = 60.22 %
Gas = 170.40 KOhms
Approx. Altitude = 210.68 m

Temperature = 29.08 *C
Pressure = 988.20 hPa
Humidity = 60.18 %
Gas = 171.27 KOhms
Approx. Altitude = 210.68 m
```

```
*********************
a library for the BME6
 specifically to work
tp://www.adafruit.com/

nsors use I2C or SPI t
face.

 invests time and reso
upport Adafruit and op
fruit!

by Limor Fried & Kevin
nse, all text above mu
*********************

Wire.h>
SPI.h>
Adafruit_Sensor.h>
Adafruit_BME680.h"

E_SCK 13
E_MISO 12
E_MOSI 11
E_CS 10

ALEVELPRESSURE_HPA (10

ME680 bme; // I2C
_BME680 bme(BME_CS); /
_BME680 bme(BME_CS, BM

() {
egin(9600);
Serial);
rintln(F("BME680 test"

.begin()) {
```

You should find that the pressure transducer is so sensitive that it can tell that you've lifted the board up from your worktable by a couple of feet just from the change in atmospheric pressure.



```
BME680 test
Temperature = 29.04 *C
Pressure = 988.0500 hPa
Humidity = 62.30 %
Gas = 0.00 KOhms
Approx. Altitude = 211.95 m

Temperature = 29.13 *C
Pressure = 988.0500 hPa
Humidity = 62.37 %
Gas = 249.01 KOhms
Approx. Altitude = 211.95 m

Temperature = 29.26 *C
Pressure = 988.0700 hPa
Humidity = 61.75 %
Gas = 260.02 KOhms
Approx. Altitude = 211.61 m

Temperature = 29.32 *C
```

### *In-class exercise and milestone 1c: version zero (due at the start of class 2)*

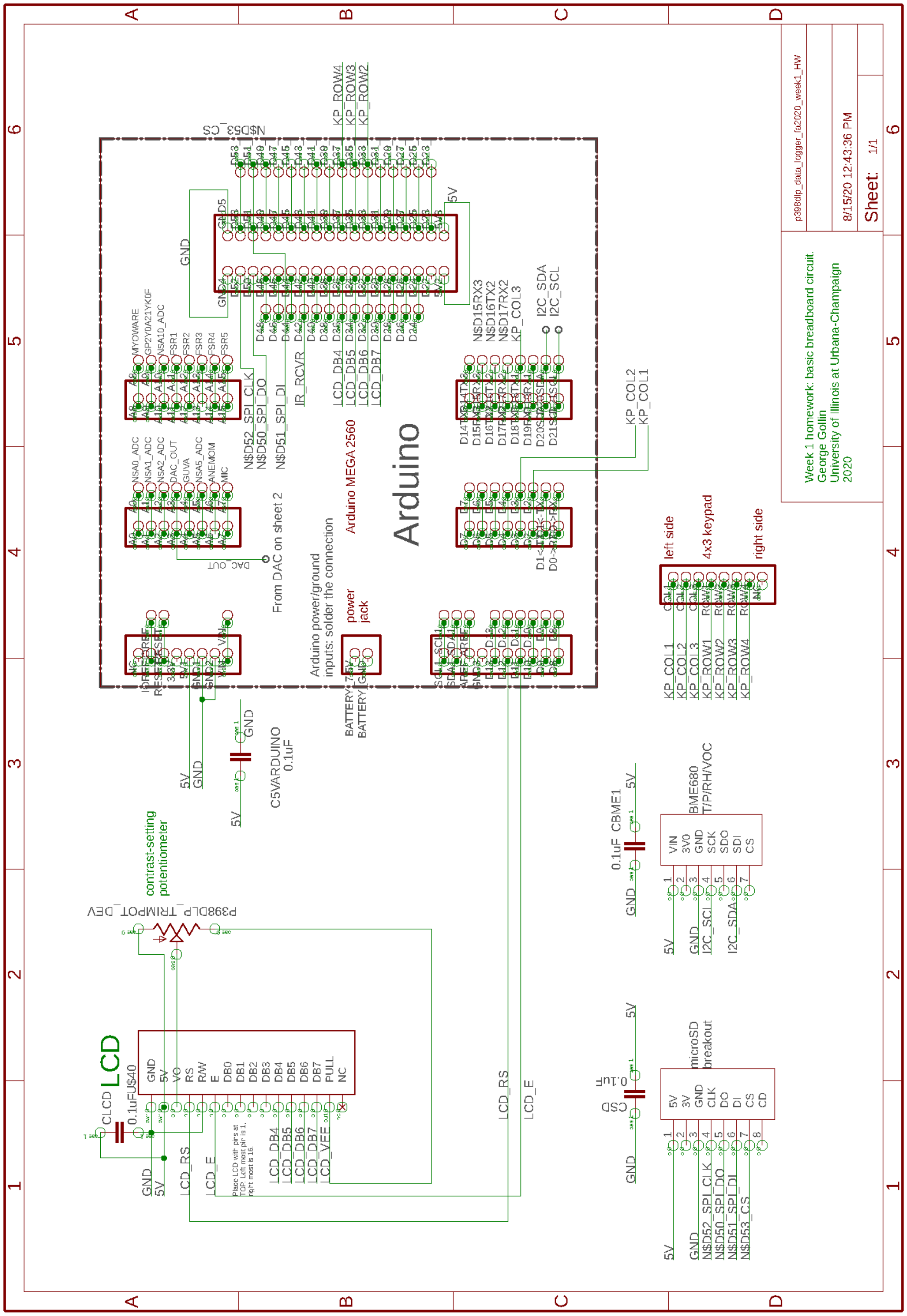
You'll spend most of the rest of the first class meeting on this task, and finish it up at home before the start of class next week. We'll want to see your progress when we meet with your group midweek.

On your breadboard, install the following devices (in addition to the BME680 and Arduino): LCD (including 10kΩ trimpot), keypad, and microSD breakout. See the schematic, below.

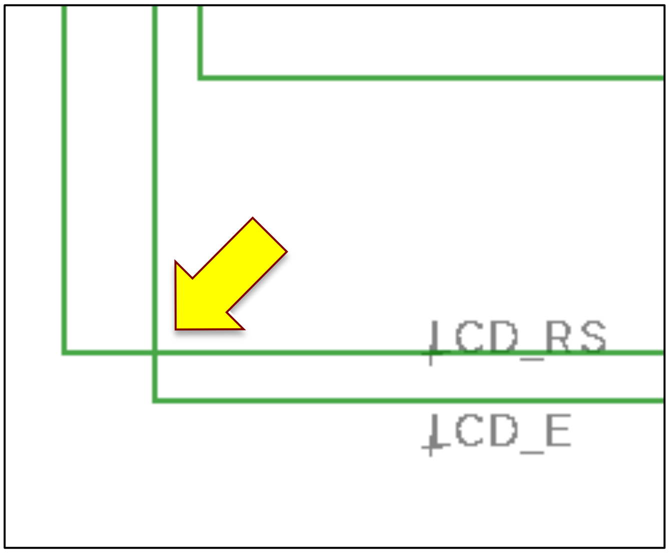
For each device find a demo program (perhaps on the Adafruit site, or from a library that you might install, or from the course's "Code & design resources repository") and confirm that it functions properly. You'll need to fool around with the 10k trimpot to adjust the LCD contrast properly.

### *How to read a schematic*

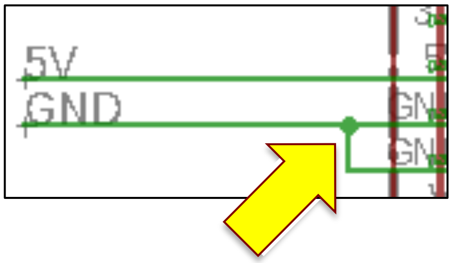
A schematic holds a topological representation of an electronic circuit. The two most important things on it are symbols for the various components—resistors, capacitors, integrated circuits and so forth—and (named) nets, which define the electrical connections between components. For example, the BME680 symbol on the schematic shows seven pins, with pins 1, 3, 4, and 6 connected to the nets named 5V, GND, I2C_SCL, and I2C_SDA, respectively. (Pins 2, 5, and 7 are not connected to anything.) Anything tied to the 5V net is electrically connected to everything else on the 5V net.


Arduino
Arduino MEGA 2560
From DAC on sheet 2
Arduino power/ground inputs: solder the connection
power jack
LCD
contrast-setting potentiometer
P398DLP_TRIMPOT_DEV
C5VARDUINO 0.1uF
CLCD 0.1uF
CBME1 0.1uF
CSD 0.1uF
BME680 T/P/RH/VOC
microSD breakout
4x3 keypad
left side
right side
Week 1 homework: basic breadboard circuit.
George Gollin
University of Illinois at Urbana-Champaign
2020
p398dlp_data_logger_fa2020_week1_HW
8/15/20 12:43:36 PM
Sheet: 1/1

When lines representing nets cross each other the point of crossing does ***not*** represent a connection between the nets! For example, the LCD_RS and LCD_E nets are not connected to each other here:



We indicate a point at which two lines (which are on the ***same*** net) are electrically connected with a dot as follows:



***The second-to-last thing this week: install the Anaconda Python IDE***

Install the Anaconda Python IDE on your laptop before the end of class and make sure it runs properly. We can help if there are problems.

See the "C++ and Python Primer" for more details about how to do this. I STRONGLY encourage you to use Anaconda's Python IDE for your "offline" code development. It's free, and you can download it from here: https://www.anaconda.com/products/individual.

***In-class exercise and milestone 1d: formulate a plan!***

Begin discussions in your group of how you will proceed. What kinds of measurements will you need to make? What devices will you need, perhaps for me to purchase on your behalf?

How will you calibrate your stuff? How much data will you need to record to be able to draw conclusions?

One way to organize your efforts is to divide up responsibility for the following:

- DAQ (data acquisition program, pronounced "DACK")
- Offline (Python) data analysis and calibration program
- 3D printing tasks and other mechanical design issues
- (Cross) calibration and hardware verification
- Data taking and run plan
- Data analysis: extracting conclusions from your measurements
- Estimate of statistical and systematic uncertainties

Begin discussing how you will divide up the tasks associated with your project. We will expect you to have a fairly detailed sense of this when we meet with you in the middle of the second week.

### ***Wrap up: informative presentation for next week***

For next week: I'd like one of the groups to give the class a ten-minute PowerPoint presentation on… (I'll sort out the topic list once the semester begins.)

### ***Appendix: soldering***

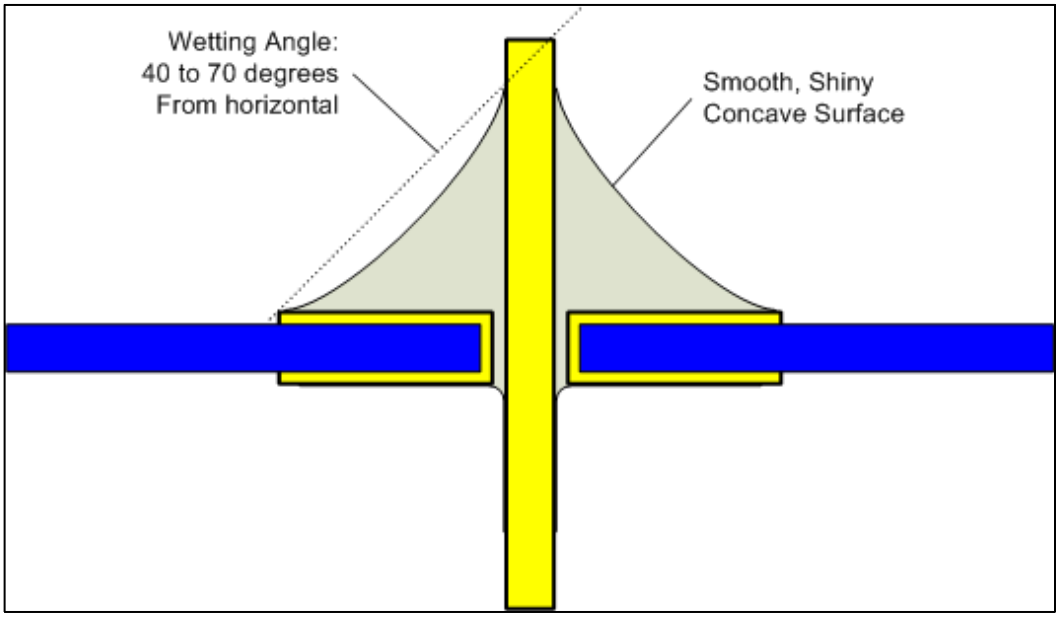
Todd Moore, an electrical engineer who used to build complex devices for the High Energy Physics Group but now staffs the undergraduate physics program, will run a soldering clinic in class. He'll probably have you attach pin headers to a few of the sensor breakout boards in your collection of goodies. There are safety considerations we'll need to observe about working with distance between all of us.

Here's an "inline" version of Todd Moore's PowerPoint soldering tutorial.

*Safety first!*

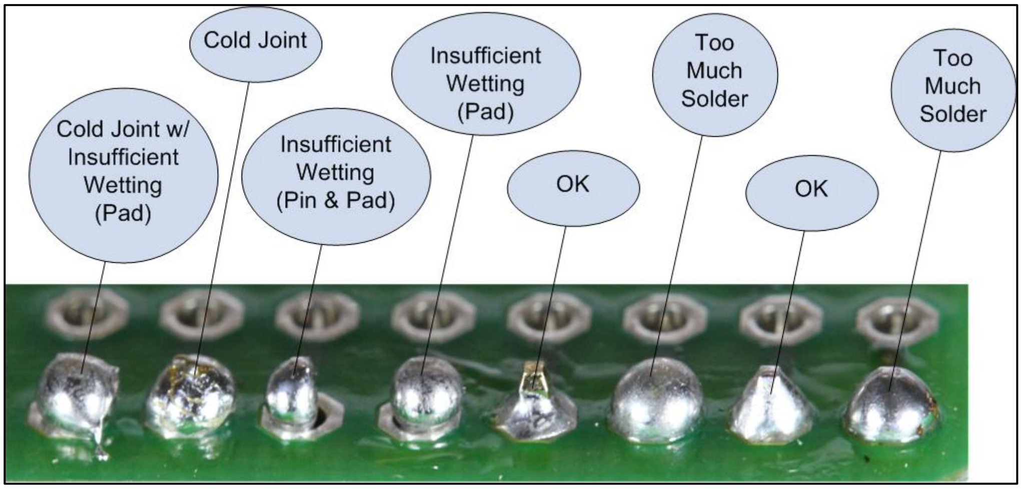
- Wear safety goggles.
- Be careful to not get burned using the soldering iron & solder wick.
- Be careful to not drop molten solder onto your clothing, shoes, etc.
- Turn off all equipment after every use.
- Solder has lead in it so:
    - Wash your hands after any soldering activity.
    - Don't put solder in your mouth to hold it.

*The Proper Solder Joint*

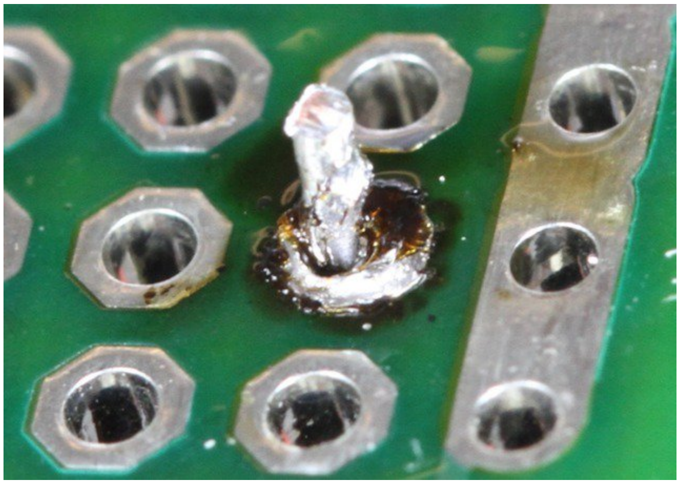


Heat the pin and pad with your soldering iron, not the solder itself. As it melts, the solder will flow onto the conductors you've heated.

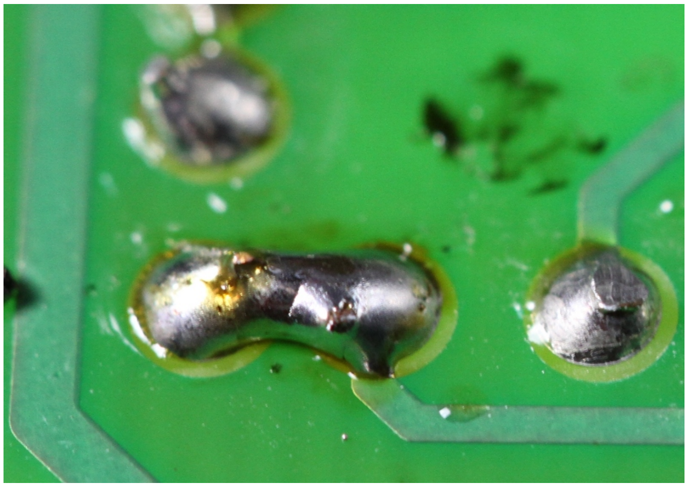
*Types of Solder Joints*


Cold Joint w/ Insufficient Wetting (Pad)
Cold Joint
Insufficient Wetting (Pin & Pad)
Insufficient Wetting (Pad)
OK
Too Much Solder
OK
Too Much Solder

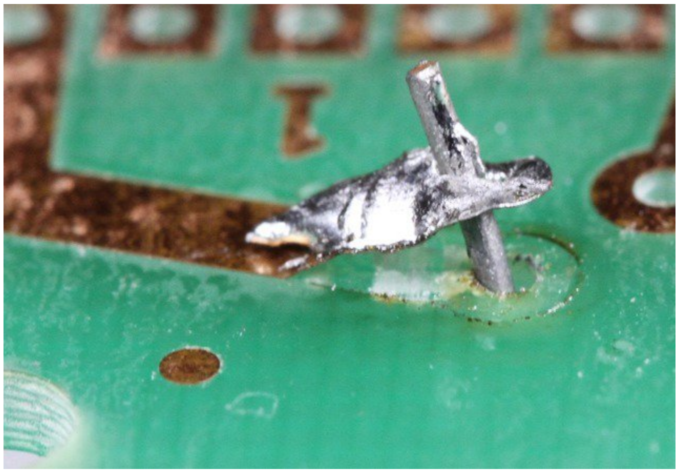
*(NOT) The Proper Solder Joint*



Overheated joint



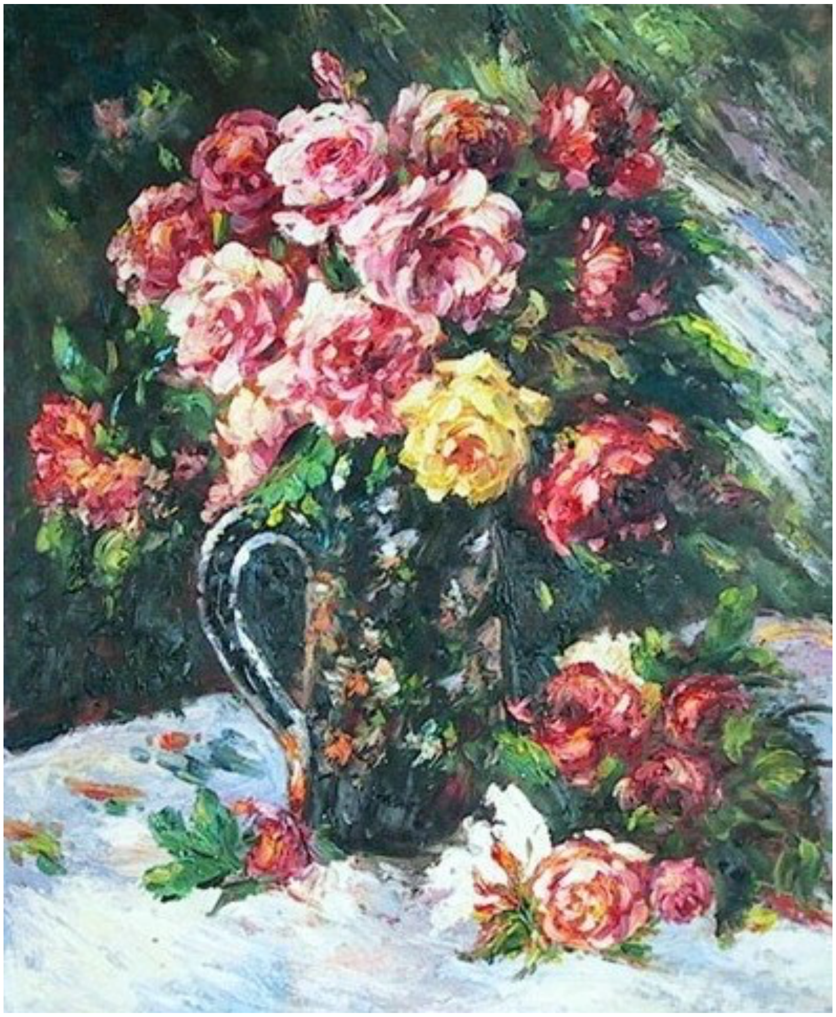
Solder bridge



Lifted pad from overheating or desoldering

***A word to the wise***

Get started on the assigned post-class work the day following our first class meeting. Don’t let it slip until the last minute, and don’t be intimidated by what I’m asking you to do! See me, or the TA for assistance in making sense of how to approach all this technology.



https://www.judaica-art.com/881-custom_default/roses-1879-by-pierre-auguste-renoir-art-gallery-oil-painting-reproductions.jpg

# Physics 371
Design Like a Physicist
Fall 2022

George Gollin
University of Illinois at Urbana-Champaign

## Week 2: Breadboarding, TinkerCad
☛ supplemental material ☚

# Week 2: Breadboarding, Coding, TinkerCad

Goals for this week ........ 3
I2C vs. SPI vs.… ........ 3
About the Real Time Clock ........ 4
About the GPS module ........ 4
In-class exercise/milestone 2a: RTC ........ 4
In-class exercise/milestone 2b: GPS ........ 4
In-class exercise/milestone 2c: Python code ........ 4
In-class exercise/milestone 2d: Finish installing parts on your breadboard (due at the start of class 3) ........ 4
At-home exercise/milestone 2e: Register an Autodesk user account, then visit the TinkerCad website ........ 5
3D printers ........ 6
Wrap up: informative presentation for next week (need a group to volunteer!) ........ 7

### *Goals for this week*

- You will install and test the rest of the breakout boards and other hardware required by your project's data logger.
- You will write Arduino and Python code to write (plain text) data to a microSD card, then read it from the card using a program running on your laptop.
- You will register an Autodesk account and take a quick look at TinkerCad.
- The group that volunteered last week will give a ten-minute presentation.

### *I2C vs. SPI vs.…*

The I2C communication protocol allows multiple devices to share common clock and common data lines. One device serves permanently as the controller (most people still use the soon-to-be-abandoned term "master") while the others act as the controller's obedient peripherals (slaves, but this is going to disappear, I bet), following instructions from the controller.

The controller sends out an address at the start of an I2C message; each of the other I2C devices knows its own built-in address and only pays attention to I2C transactions with the correct address. For example, the default I2C address for a BME680 is hexadecimal 0x77 ($7 \times 16 + 7 = 119$ in base 10). When you add another I2C device to your breadboard you'll connect its data and clock lines to the I2C data and clock lines shared by all the other I2C devices.

The microSD card breakout board uses the SPI (Serial Peripheral) interface communications protocol, rather than the slower I2C protocol used by the BME680. There's demonstration software at the Adafruit site that you probably used last week to test it.

You’ll have noticed that the number of communication lines is three, in contrast to I2C which uses two. Each SPI device also receives a “chip select” signal from the microcontroller, and it is the assertion of a device’s CS input that tells it to pay attention to its SPI data and clock lines.

### *About the Real Time Clock*

When you assemble the DS3231 real time clock module you’ll need to slide in a CR1220 battery. Be sure to get the orientation correct.

Oh—CR1220 means this: CR = $LiMnO_2$; 1220 = 12 mm (actually 12.5) diameter, 2.0 mm thick.

### *About the GPS module*

The GPS module also takes a CR1220 (backup) battery. Even if you don’t need the GPS for capturing the location of your device, you’ll want to use it to synchronize the RTCs of the different group members’ data loggers. Even though the GPS reports data about once per second, the module generates a PPS (pulse per second) signal just as its satellite-based clock advances to the next second. With this, you can synchronize multiple RTCs with a precision of a few hundred nanoseconds.

There’s red LED on the GPS module that blinks about once per second while it is looking for satellites. Once it has acquired signals from several satellites the blink rate will decrease to once per fifteen seconds.

### *In-class exercise/milestone 2a: RTC*

If you haven't already done so, install the DS3231 real time clock breakout board. Find some demo software and set it to the current time and date.

### *In-class exercise/milestone 2b: GPS*

Find some demo software and do something with the GPS, if you’re using one. Also use the GPS to set the real time clock to UTC (“Coordinated Universal Time,” which is almost the same thing as Greenwich Mean Time). I have some Arduino code on the web site that you’re welcome to use.

### *In-class exercise/milestone 2c: Python code*

Have your Arduino write a short text file to your SD card. Copy the file to your laptop, then write a short Python program to read it and display its contents.

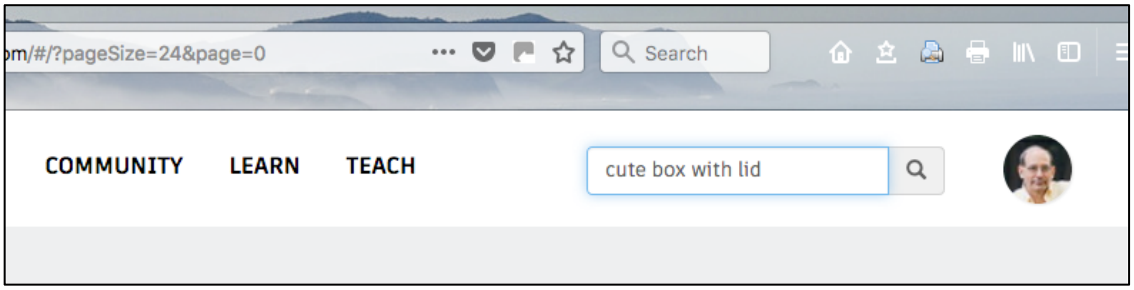
### *In-class exercise/milestone 2d: Finish installing parts on your breadboard (due at the start of class 3)*

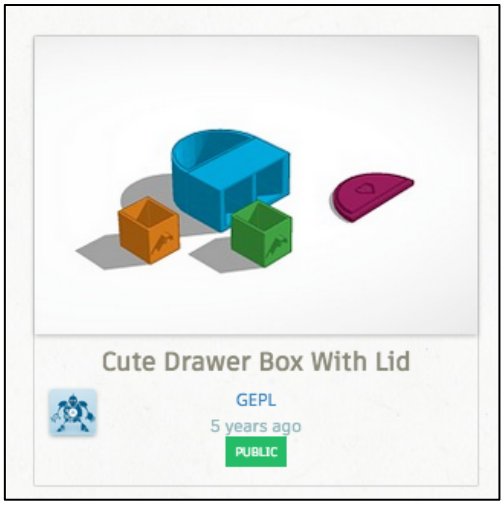
Finish installing all the parts on your breadboard your project will need. Since your breadboard circuit is to be electrically equivalent to your PCB version, you’ll need to install the five-AA-battery holder, INA219, LED, pushbutton, and so forth. Try to get as much of this done before your group meets with me in the middle of the week. Find demo software to test each device as you install it.

***At-home exercise/milestone 2e: Register an Autodesk user account, then visit the TinkerCad website***

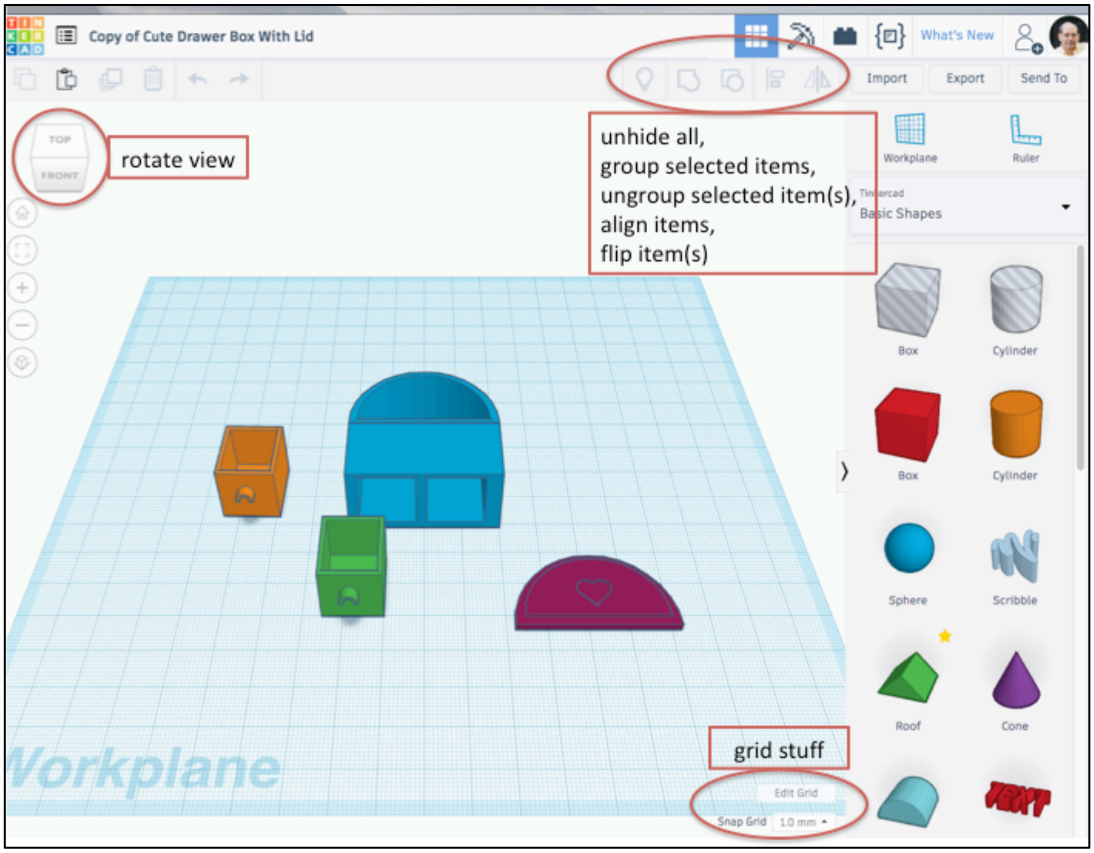
You will start using TinkerCad fairly soon to do a little bit of 3D printing. You'll need to register a (free) student account with Autodesk: go to https://knowledge.autodesk.com/customer-service/account-management/education-program/create-education-account/create-account-students-educators. You'll have much better (free!) access to Autodesk's tools this way than you would if you registered a (paid) non-student account.

TinkerCad is a web-based tool, so there's nothing to download. Go to TinkerCad.com and log in using your Autodesk username and password. Search for something interesting but small: "cute box with lid," for example.





Click on the image, then select "Copy and Tinker." Here's what the working area looks like:



If you are inclined, engage with a basic TinkerCad tutorial to see how to do simple things. But you'll probably find that there are plenty of good YouTube videos that will show you how to do most anything you want. Make very brief notes on any tricks that you find useful, for later recall and reference. I can give you a hands-on TinkerCad demo next week, if you'd like.

After you've done as much nastiness as you want to the thing you've been editing, make an STL (stereo lithography) file by going to the export tab near the upper right side of the screen. Later, you'll use TinkerCad to design the case for your device.

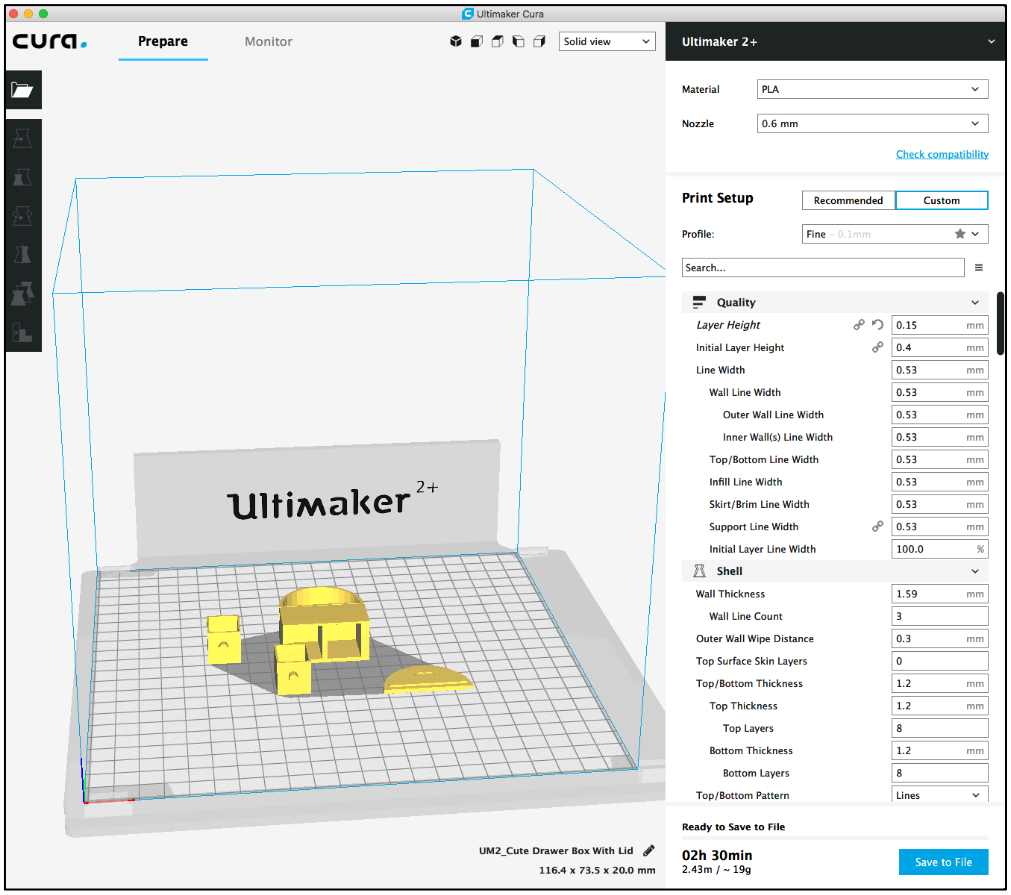
It's fine if you don't have the time or the energy to do anything with TinkerCad this week. We'll probably begin working with it next week.

### *3D printers*

The trio of 3D printers in my lab are built by Ultimaker B.V., a Dutch company. The printers do not work directly with STL files, instead reading a long list of instructions from a "gcode" file. Cura is their software product that generates gcode files (the inputs to a 3D printer) from STL files. It will probably work best if you send me the STL files you'd like printed; from these I can run Cura to generate the print files. This way I can correct any errors I notice in your STL files.

But if you'd like to take a peek at Cura, download and install the latest version from Ultimaker's web site: go to https://ultimaker.com/en/products/ultimaker-cura-software. Open

Cura and drop onto it the STL file you've created with TinkerCad. Have it generate a "gcode" file for an Ultimaker 2+ 3D printer that is equipped with a 0.4 mm nozzle. Note that the program gives an estimate of how long your object will take to print. Don't worry of this all seems too obscure.



| | |
|---|---|
| Material | PLA |
| Nozzle | 0.6 mm |
| Print Setup | Custom |
| Profile: | Fine – 0.1mm |
| **Quality** | |
| *Layer Height* | 0.15 mm |
| Initial Layer Height | 0.4 mm |
| Line Width | 0.53 mm |
| Wall Line Width | 0.53 mm |
| Outer Wall Line Width | 0.53 mm |
| Inner Wall(s) Line Width | 0.53 mm |
| Top/Bottom Line Width | 0.53 mm |
| Infill Line Width | 0.53 mm |
| Skirt/Brim Line Width | 0.53 mm |
| Support Line Width | 0.53 mm |
| Initial Layer Line Width | 100.0 % |
| **Shell** | |
| Wall Thickness | 1.59 mm |
| Wall Line Count | 3 |
| Outer Wall Wipe Distance | 0.3 mm |
| Top Surface Skin Layers | 0 |
| Top/Bottom Thickness | 1.2 mm |
| Top Thickness | 1.2 mm |
| Top Layers | 8 |
| Bottom Thickness | 1.2 mm |
| Bottom Layers | 8 |
| Top/Bottom Pattern | Lines |

Ready to Save to File

02h 30min
2.43m / ~ 19g

If you'd like, I can talk you through the Cura "slicing" process and discuss a couple of settings you'll want to impose if you generate the gcode files yourself. A couple of technical issues to consider: (1) adhesion to the build surface; (2) support for bridged features. I'll explain what I mean if we discuss Cura.

### *Wrap up: informative presentation for next week (need a group to volunteer!)*

For next week: I'd like one of the groups to give the class a ten-minute PowerPoint presentation on…

http://paintingandframe.com/prints/pierre_auguste_renoir_vase_of_roses-797.html

# Physics 371

Design Like a Physicist

Fall 2022

George Gollin

University of Illinois at Urbana-Champaign

## Week 3: DAQ, TinkerCad, Schematic Capture

☛ supplemental material ☚

# Week 3: DAQ, TinkerCad, Schematic Capture

Goals for this week ........................................3
Register-oriented microcontroller architecture: GG comments ........................................3
TinkerCad hands-on demo........................................5
Whole-class discussion........................................6
In-class exercise/milestone 3a: bare bones DAQ ........................................6
In-class exercise/milestone 3b: bare bones Python offline analysis (due at the start of class 4).....6
Post-class exercise/milestone 3c: log in to Autodesk and download EAGLE ........................................6
Circuit schematics and PCB layouts........................................7
For future reference: making a schematic ........................................9
Wrap up: informative presentation for next week (need a group to volunteer!) ........................................20

## *Goals for this week*

- We will discuss register-oriented microcontrollers, after which we will mess around with TinkerCad.
- You will begin work on your project's DAQ (data acquisition program) and offline Python analysis program.
- The group that volunteered last week will give a ten-minute presentation.

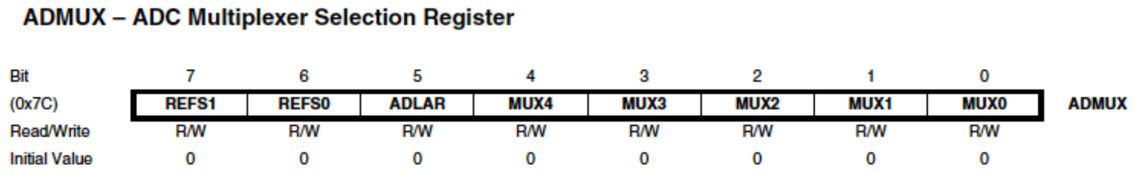
## *Register-oriented microcontroller architecture: GG comments*

The Arduino's microcontroller includes a CPU, several timers, an ADC (analog to digital converter, of course), and other quasi-independent functional blocks. These functional blocks communicate in a manner that reminds me of something from an espionage novel: the microcontroller blocks monitor special locations in memory called registers, reading from (and writing to) the registers various control signals and data. They do not communicate directly with each other. In a novel spies use dead drops to exchange information, avoiding face to face meetings as much as possible. One spy will hide a document in a prearranged location then make a chalk mark at another location to signal to a colleague that new information is available.

Let's look briefly at the three registers that control the ADC module. Note that many of the predefined functions in the Arduino programming environment take care of all the register manipulations for you, so that you'll only need to work directly with registers if you are forced by constraints of execution speed to do so.

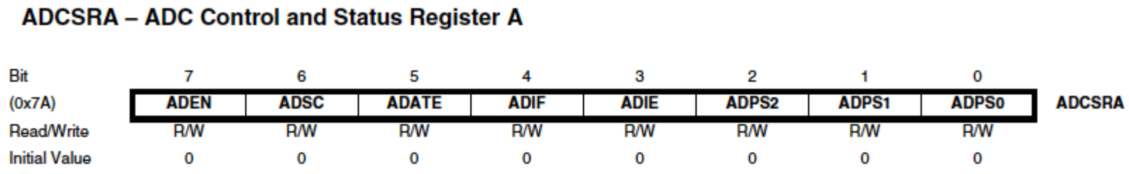
The microcontroller contains a single ADC that receives a signal from the output of an analog multiplexer (a 16-input, 1-output switch) whose inputs are the microcontroller's sixteen analog pins A0 – A15. Before requesting a digitization, the CPU sets the low-order five bits in the ADMUX register. These bits are referred to as MUX0 through MUX4.

An Atmel 2560's ADC will produce a 10-bit digitization of its input by comparing the input voltage with a reference voltage. The CPU specifies which reference voltage is to be used by setting the two higher order ADMUX bits. These are called REFS0 and REFS1.



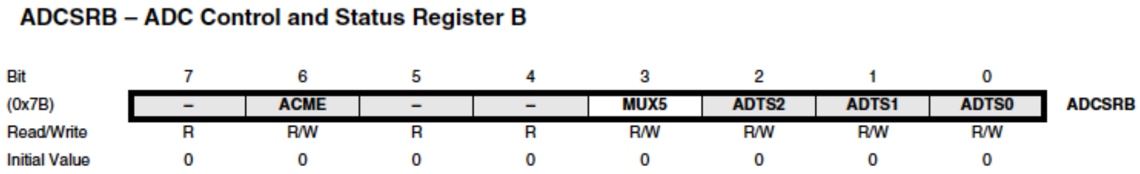
**ADMUX – ADC Multiplexer Selection Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7C) | REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |



**ADCSRA – ADC Control and Status Register A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7A) | ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 | ADCSRA |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |



**ADCSRB – ADC Control and Status Register B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7B) | – | ACME | – | – | MUX5 | ADTS2 | ADTS1 | ADTS0 | ADCSRB |
| Read/Write | R | R/W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

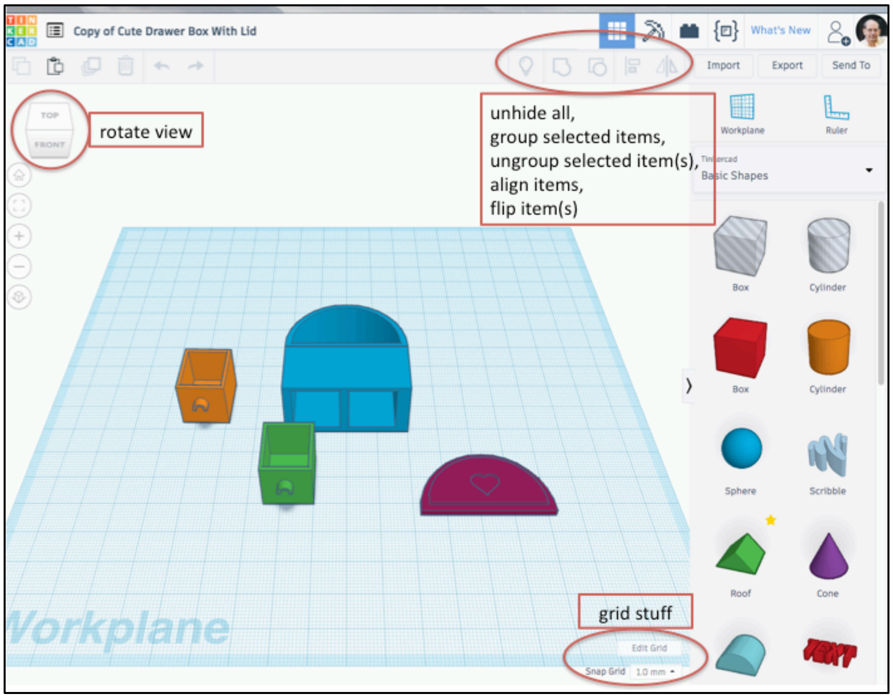
From Atmel’s data sheet (linked to the course web site):

> The ADC generates a 10-bit result which is presented in the ADC Data Registers, ADCH and ADCL. By default, the result is presented right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX.

The reference voltage choices are the Arduino’s 5V supply, or built-in reference voltages of 1.1V and 2.56V.

Please take a look, if you’re interested, at chapter 26 in the Atmel 2560 manual on the course web site for more ADC register information: https://courses.physics.illinois.edu/phys371/fa2021/code/Atmel_2560_datasheet.pdf.

***TinkerCad hands-on demo***



rotate view

unhide all,
group selected items,
ungroup selected item(s),
align items,
flip item(s)

grid stuff

You should find something amusing to work on, then follow along. Take notes. We'll try some of these actions:

- rotating point of view; zooming in and out
- grouping and ungrouping objects
- moving objects using the mouse and arrow keys
- rotating objects
- hiding/unhiding objects
- changing sizes while preserving (or not) aspect ratios
- making holes
- using (and changing) the grid
- aligning items
- quick example: how to make a lidless box
- adding text

After you've done as much nastiness to the thing you've been editing, make an STL (stereo lithography) file by going to the export tab near the upper right side of the screen.

In past semesters I've asked students to install Cura, the software that translates STL into gcode, the list of instructions that the 3D printer will execute. But it's often been the case that I need to correct small mistakes—parts floating above the printer's build plate, rather than lying on it, for example—so I think it'll be easier if you just send me STL files to process, rather than gcode from running Cura yourself.

### *Whole-class discussion*

- How is it going? What's too hard/too easy?
- What are your thoughts about the field work you'll be doing in a few weeks? Have you made contact yet with anyone to learn the rules concerning entering into their environment to make measurements?
- What might your data acquisition code look like?
- What kind of technical support might I provide to make your work go more smoothly?
- How will you go about analyzing your data? How will you address calibration issues?

### *In-class exercise/milestone 3a: bare bones DAQ*

"DAQ" is professional shorthand for "Data Acquisition Software." Working with the other members of your team, please write a program that opens a microSD file, then executes a loop 100 times in which you record the time (from the DS3231, of course), the temperature, pressure, and humidity. Have your loop execute approximately once per second; after 100 passes have it close the microSD file. It'll be easiest to inspect the file if you write it as human-readable plain text rather than binary.

Write (or find) another program that reads your data file from the microSD card and writes its contents to the serial monitor window.

### *In-class exercise/milestone 3b: bare bones Python offline analysis (due at the start of class 4)*

Remove the microSD card from the breakout board and try to open it with the SD card input on your laptop, if you have one, or else on someone else's laptop. Working together, discuss a general outline of an offline analysis code that will read the file you've transferred from your SD card and generate histograms of the BME680 quantities, both as functions of time and as plots that show how often particular values were received.

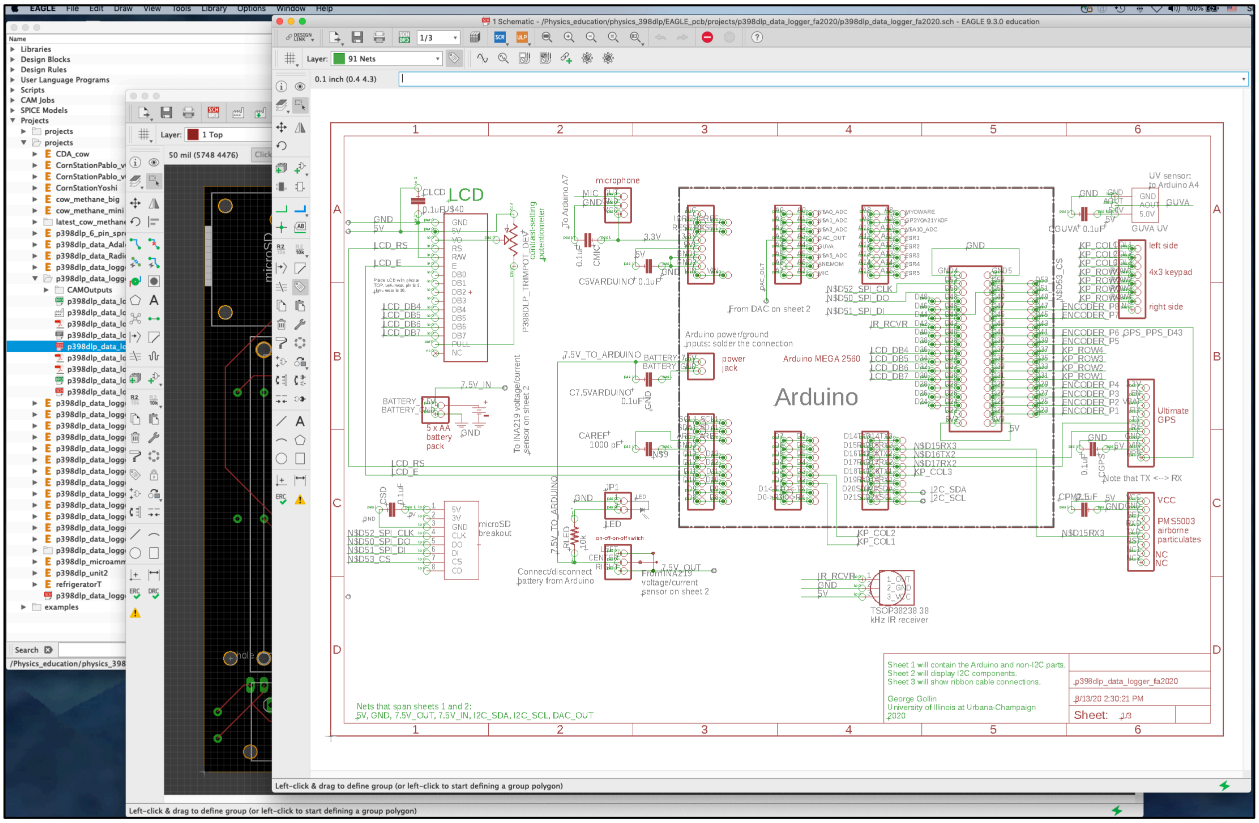
Generate another run of your DAQ, but with faster sampling, and record a few thousand environmental measurements. Calculate the means and RMS deviations of the BME680 quantities.

### *Post-class exercise/milestone 3c: log in to Autodesk and download EAGLE*

Visit https://www.autodesk.com/education/free-software/eagle and sign in. Follow the steps to download EAGLE, Autodesk's schematic capture and printed circuit board layout tool. In a week or two I'll want one member of your group to become competent with the schematic capture tool so that you can edit down my version, which has a lot of superfluous stuff.

There are good tutorials on how to install, then use EAGLE on the sparkfun.com web site. I've linked them to the course's "Code and design resource repository" web page under the heading "EAGLE schematic capture and PCB tools." You should start with these.

Here’s a quick look, in case you’d like start with something other than the SparkFun material: this is the first page of my schematic for an all-the-bells-and-whistles version of what you’ll be working with. It is shown in the EAGLE IDE.



1 Schematic - /Physics_education/physics_398dlp/EAGLE_pcb/projects/p398dlp_data_logger_fa2020/p398dlp_data_logger_fa2020.sch - EAGLE 9.3.0 education

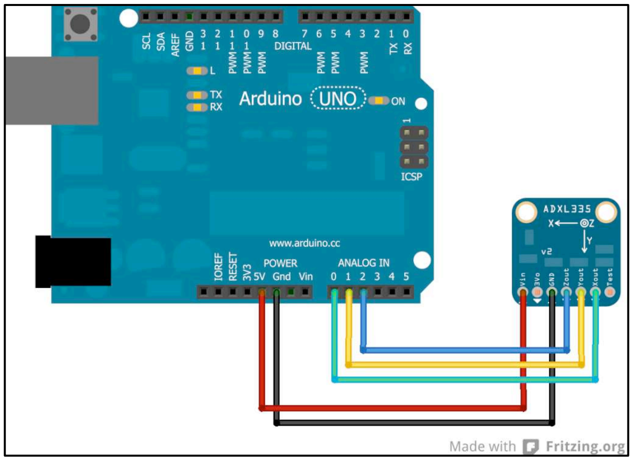
Left-click & drag to define group (or left-click to start defining a group polygon)

EAGLE
File
Edit
Draw
View
Tools
Library
Options
Window
Help
Libraries
Design Blocks
Design Rules
User Language Programs
Scripts
CAM Jobs
SPICE Models
Projects
examples
Search
Layer: 91 Nets
Layer: 1 Top
0.1 inch (0.4 4.3)
50 mil (5748 4476)
LCD
microphone
Arduino
Arduino MEGA 2560
Arduino power/ground inputs: solder the connection
power jack
From DAC on sheet 2
5 x AA battery pack
microSD breakout
Connect/disconnect battery from Arduino
To INA219 voltage/current sensor on sheet 2
From INA219 voltage/current sensor on sheet 2
UV sensor: to Arduino A4
left side
4x3 keypad
right side
Ultimate GPS
PMS5003 airborne particulates
Note that TX <--> RX
TSOP38238 38 kHz IR receiver
Nets that span sheets 1 and 2:
5V, GND, 7.5V_OUT, 7.5V_IN, I2C_SDA, I2C_SCL, DAC_OUT
Sheet 1 will contain the Arduino and non-I2C parts.
Sheet 2 will display I2C components.
Sheet 3 will show ribbon cable connections.
George Gollin
University of Illinois at Urbana-Champaign
2020
p398dlp_data_logger_fa2020
8/13/20 2:30:21 PM
Sheet: 1/3

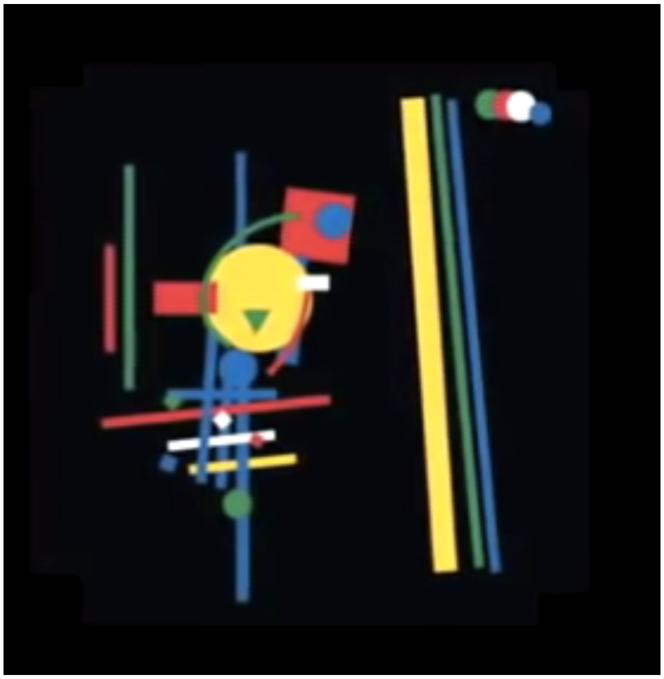
### *Circuit schematics and PCB layouts*

One of the very first steps in designing a circuit is the creation of a schematic diagram that indicates which parts will be used in the circuit, and how they are wired to each other.

If you go to the Adafruit site to see how to wire up an ADXL326 accelerometer, you will probably see a representation of the circuit that looks like the following. That’s not a schematic: though the illustration clearly represents the components and interconnections, it is inefficient in its use of space.



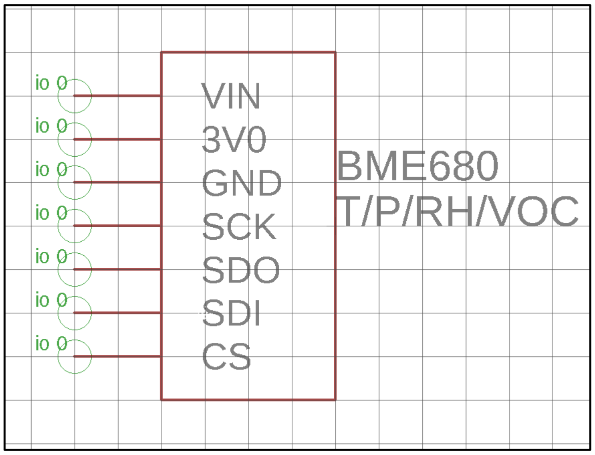
Did any of you play with Colorforms when you were little? It’s a creative toy which includes a variety of thin vinyl shapes—circles, rectangles, and the like—that will adhere to the smooth, black surface that a child uses as a drafting environment.

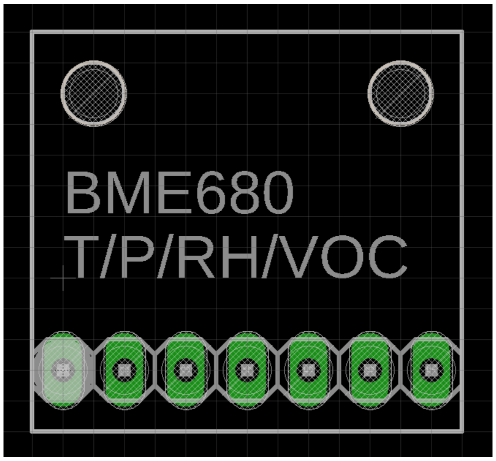


A schematic capture tool works in similar fashion: the designer selects parts from a library of components, places them on the design surface, and draws the interconnections between component pins.

The data characterizing each part in the library include two distinct representations: one specifying just the pins to which connections are made, and another that incudes geometric information. We use the first to define the topology of the circuit under construction, while we employ the other to decide where we can actually place components on a circuit board.

An example: here are the symbol and package (or footprint) for a BME680.

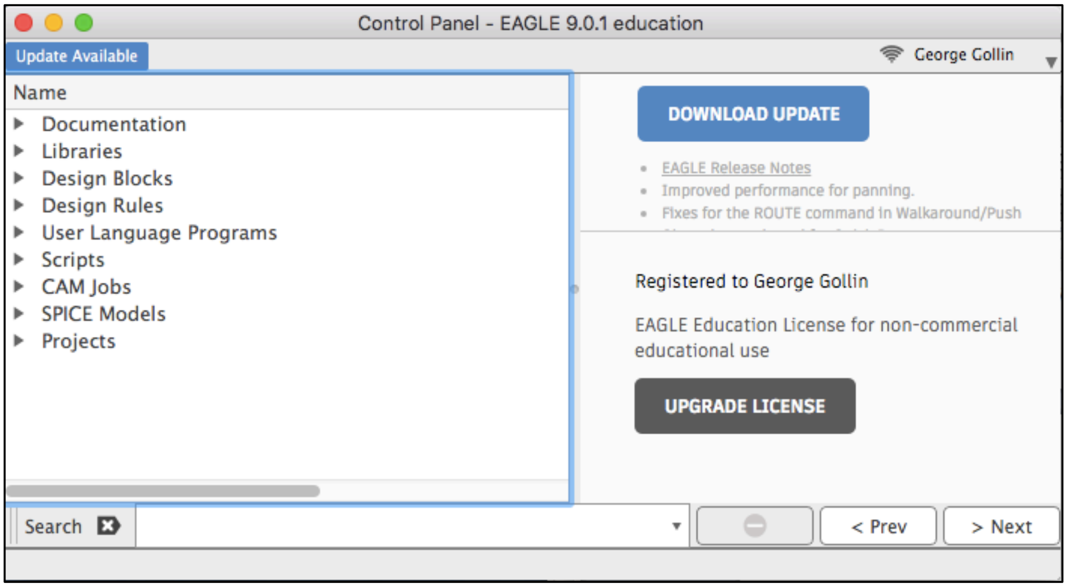




### *For future reference: making a schematic*

Let’s say you’d like to create the schematic for your breadboard circuit. To do so, please download and unzip the files under the link shown as “Some EAGLE parts libraries” on the code repository page in the course web site. Put these in a folder that you find (or create) named external_lbrs.

Open EAGLE. You should find that it starts by displaying the control panel.

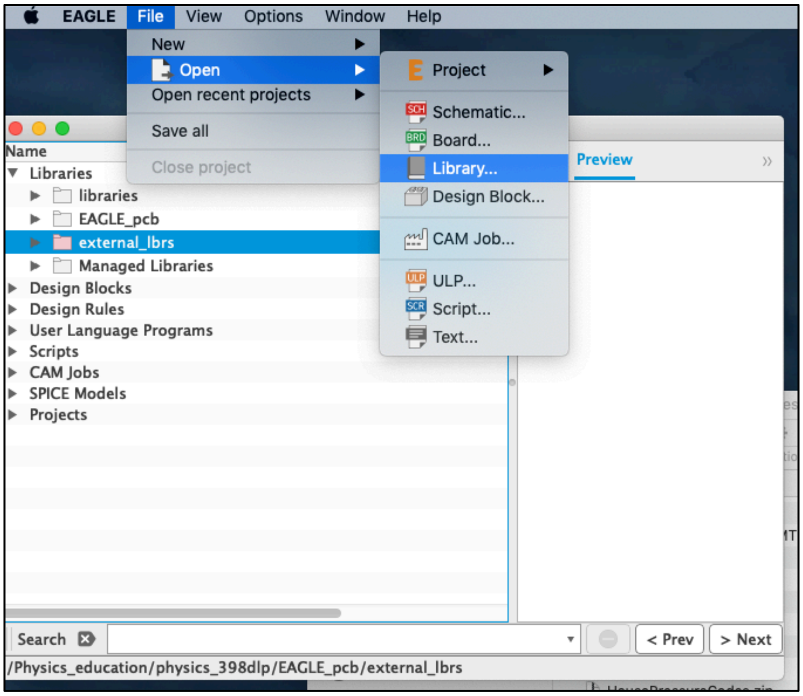


- Improved performance for panning.
- Fixes for the ROUTE command in Walkaround/Push

Registered to George Gollin

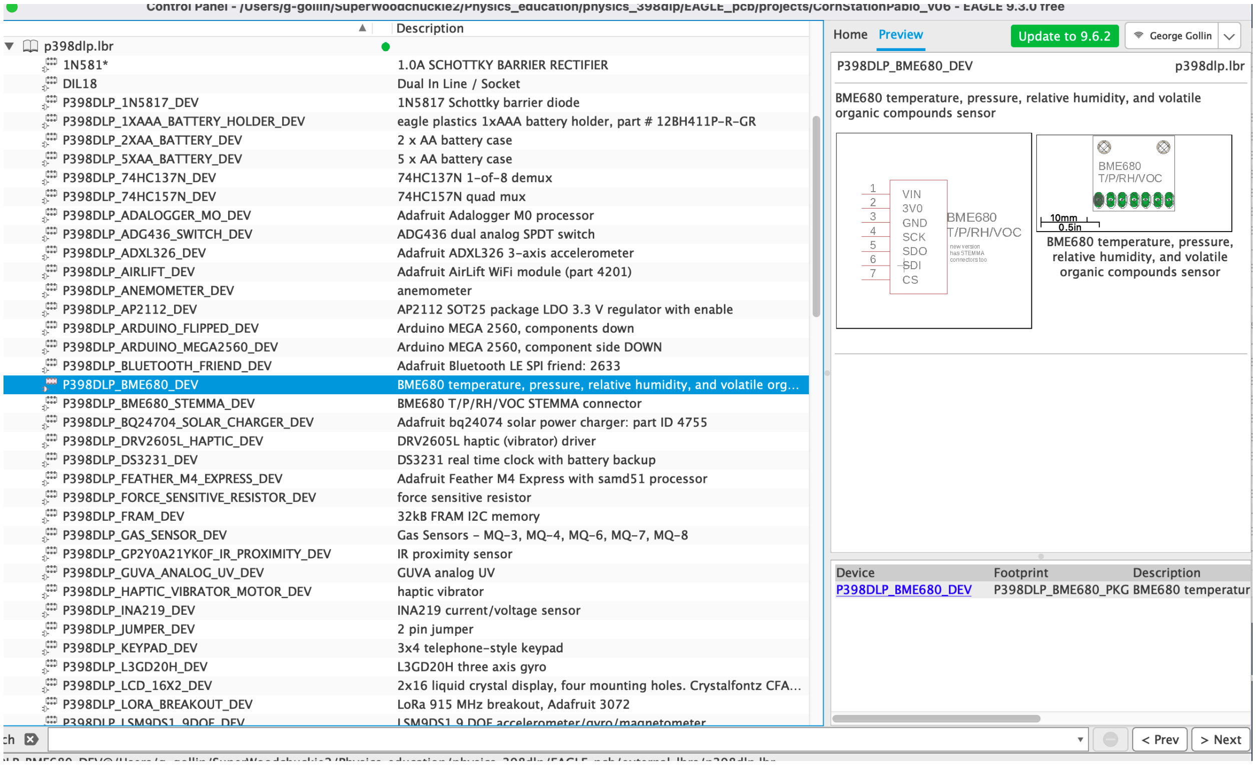
EAGLE Education License for non-commercial educational use

If the external_lbrs folder is not visible inside the Libraries category, find it yourself:



/Physics_education/physics_398dlp/EAGLE_pcb/external_lbrs

Some of the parts you'll be using are found in the p371.lbr library file. For example, here's the BME680:


Control Panel - /Users/g-gollin/SuperWoodchuckie2/Physics_education/physics_398dlp/EAGLE_pcb/projects/CornStationPablo_V06 - EAGLE 9.3.0 free

| ▲ | Description |
|---|---|
| p398dlp.lbr | |
| 1N581* | 1.0A SCHOTTKY BARRIER RECTIFIER |
| DIL18 | Dual In Line / Socket |
| P398DLP_1N5817_DEV | 1N5817 Schottky barrier diode |
| P398DLP_1XAAA_BATTERY_HOLDER_DEV | eagle plastics 1xAAA battery holder, part # 12BH411P-R-GR |
| P398DLP_2XAA_BATTERY_DEV | 2 x AA battery case |
| P398DLP_5XAA_BATTERY_DEV | 5 x AA battery case |
| P398DLP_74HC137N_DEV | 74HC137N 1-of-8 demux |
| P398DLP_74HC157N_DEV | 74HC157N quad mux |
| P398DLP_ADALOGGER_MO_DEV | Adafruit Adalogger M0 processor |
| P398DLP_ADG436_SWITCH_DEV | ADG436 dual analog SPDT switch |
| P398DLP_ADXL326_DEV | Adafruit ADXL326 3-axis accelerometer |
| P398DLP_AIRLIFT_DEV | Adafruit AirLift WiFi module (part 4201) |
| P398DLP_ANEMOMETER_DEV | anemometer |
| P398DLP_AP2112_DEV | AP2112 SOT25 package LDO 3.3 V regulator with enable |
| P398DLP_ARDUINO_FLIPPED_DEV | Arduino MEGA 2560, components down |
| P398DLP_ARDUINO_MEGA2560_DEV | Arduino MEGA 2560, component side DOWN |
| P398DLP_BLUETOOTH_FRIEND_DEV | Adafruit Bluetooth LE SPI friend: 2633 |
| P398DLP_BME680_DEV | BME680 temperature, pressure, relative humidity, and volatile org... |
| P398DLP_BME680_STEMMA_DEV | BME680 T/P/RH/VOC STEMMA connector |
| P398DLP_BQ24704_SOLAR_CHARGER_DEV | Adafruit bq24074 solar power charger: part ID 4755 |
| P398DLP_DRV2605L_HAPTIC_DEV | DRV2605L haptic (vibrator) driver |
| P398DLP_DS3231_DEV | DS3231 real time clock with battery backup |
| P398DLP_FEATHER_M4_EXPRESS_DEV | Adafruit Feather M4 Express with samd51 processor |
| P398DLP_FORCE_SENSITIVE_RESISTOR_DEV | force sensitive resistor |
| P398DLP_FRAM_DEV | 32kB FRAM I2C memory |
| P398DLP_GAS_SENSOR_DEV | Gas Sensors - MQ-3, MQ-4, MQ-6, MQ-7, MQ-8 |
| P398DLP_GP2Y0A21YK0F_IR_PROXIMITY_DEV | IR proximity sensor |
| P398DLP_GUVA_ANALOG_UV_DEV | GUVA analog UV |
| P398DLP_HAPTIC_VIBRATOR_MOTOR_DEV | haptic vibrator |
| P398DLP_INA219_DEV | INA219 current/voltage sensor |
| P398DLP_JUMPER_DEV | 2 pin jumper |
| P398DLP_KEYPAD_DEV | 3x4 telephone-style keypad |
| P398DLP_L3GD20H_DEV | L3GD20H three axis gyro |
| P398DLP_LCD_16X2_DEV | 2x16 liquid crystal display, four mounting holes. Crystalfontz CFA... |
| P398DLP_LORA_BREAKOUT_DEV | LoRa 915 MHz breakout, Adafruit 3072 |
| P398DLP_LSM9DS1_9DOF_DEV | LSM9DS1 9 DOF accelerometer/gyro/magnetometer |

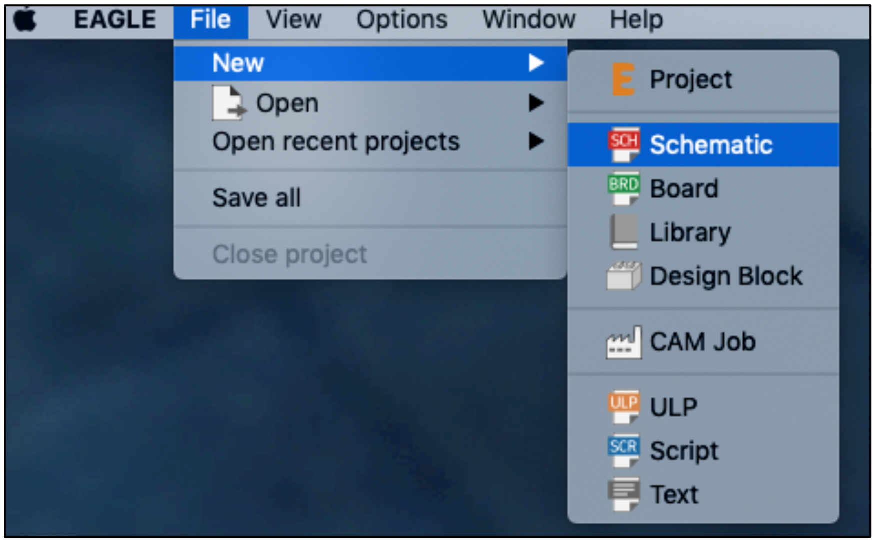
P398DLP_BME680_DEV p398dlp.lbr

BME680 temperature, pressure, relative humidity, and volatile organic compounds sensor

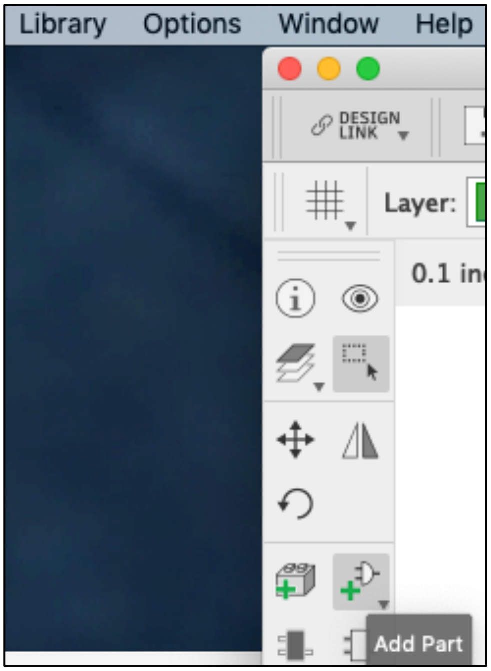
BME680 temperature, pressure, relative humidity, and volatile organic compounds sensor

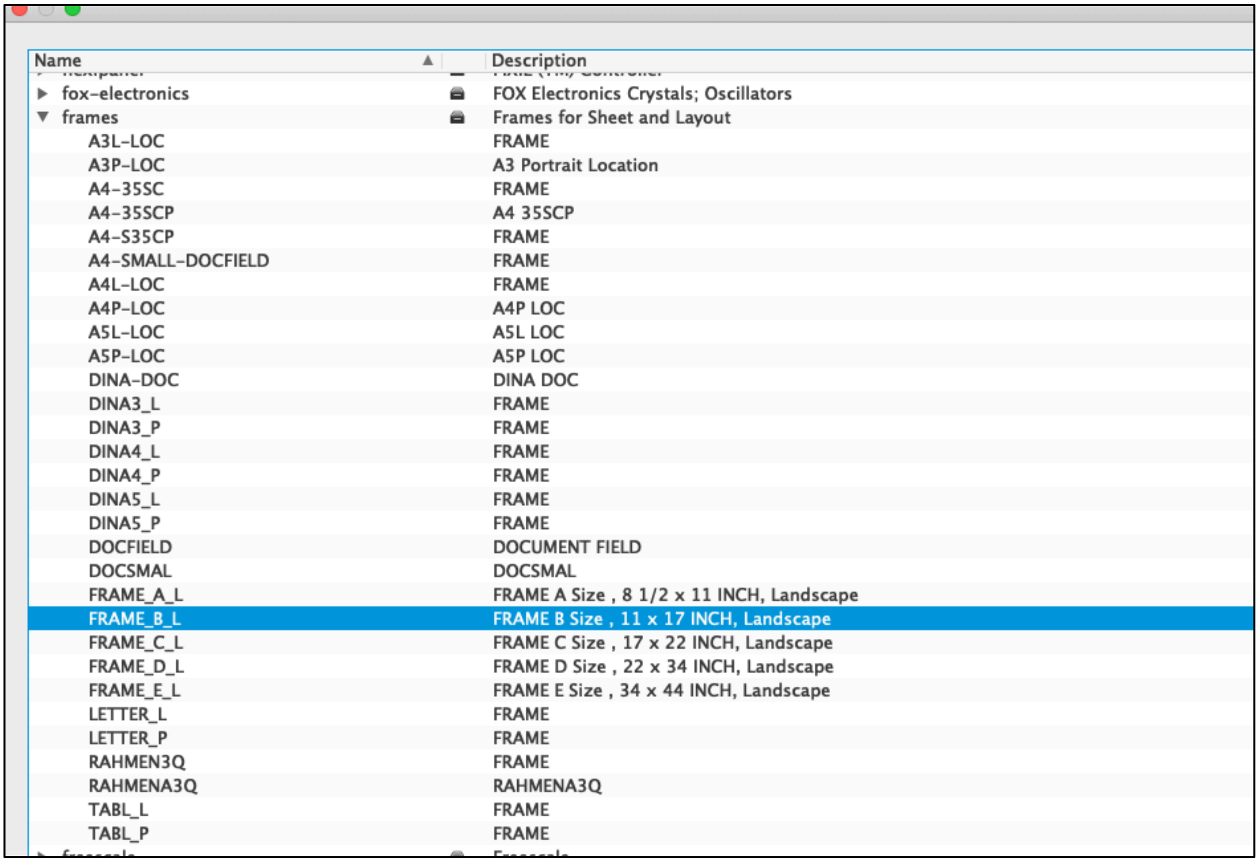
| Device | Footprint | Description |
|---|---|---|
| P398DLP_BME680_DEV | P398DLP_BME680_PKG | BME680 temperatur |

Here's how to open a new schematic:



then click on the "Add Part" tile. Scroll down to the library named "frames" and pick a frame that works well for you.

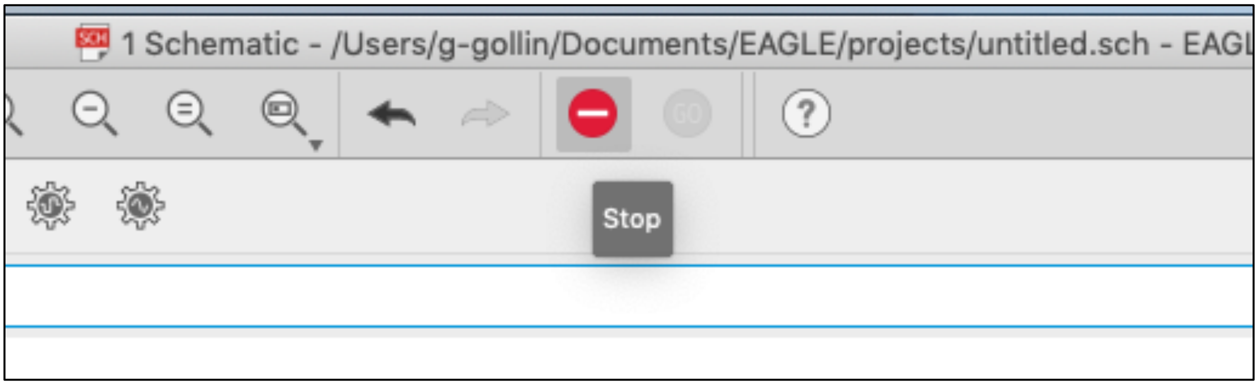




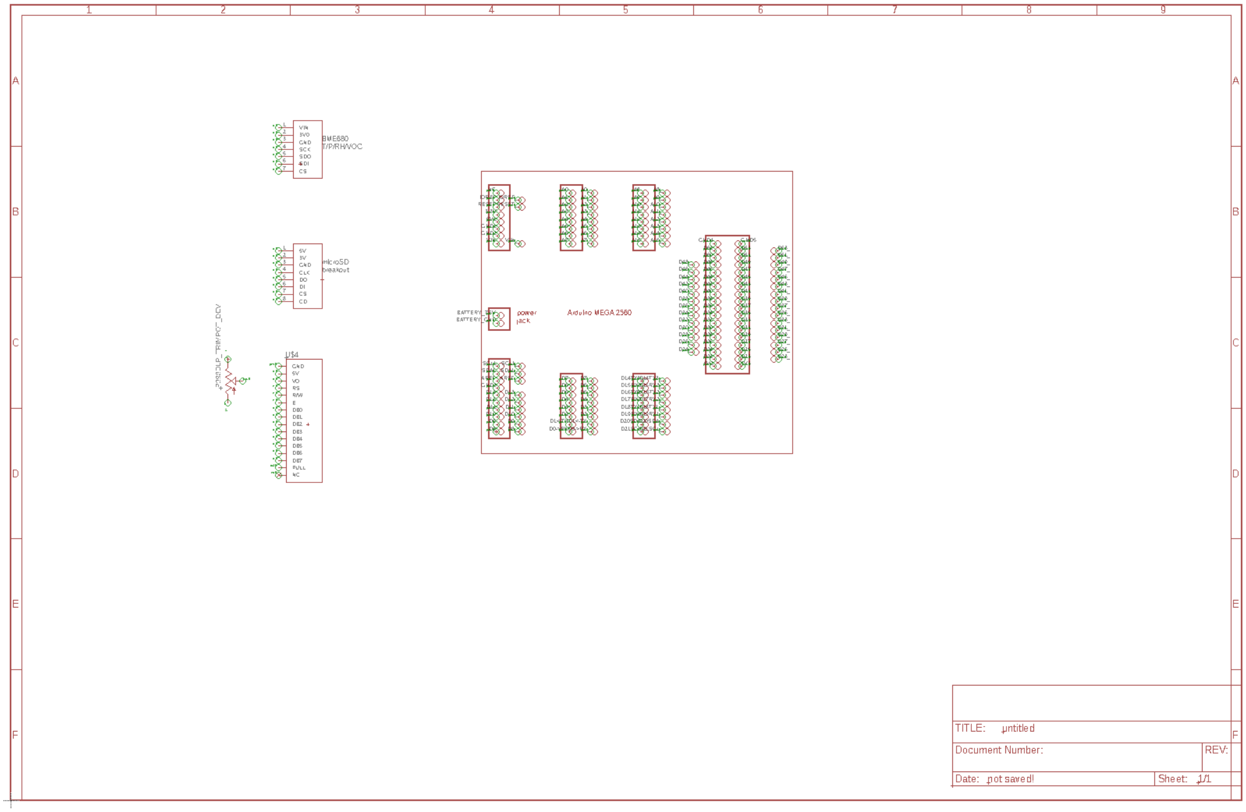
| Name | Description |
|---|---|
| ▶ fox-electronics | FOX Electronics Crystals; Oscillators |
| ▼ frames | Frames for Sheet and Layout |
| A3L-LOC | FRAME |
| A3P-LOC | A3 Portrait Location |
| A4-35SC | FRAME |
| A4-35SCP | A4 35SCP |
| A4-S35CP | FRAME |
| A4-SMALL-DOCFIELD | FRAME |
| A4L-LOC | FRAME |
| A4P-LOC | A4P LOC |
| A5L-LOC | A5L LOC |
| A5P-LOC | A5P LOC |
| DINA-DOC | DINA DOC |
| DINA3_L | FRAME |
| DINA3_P | FRAME |
| DINA4_L | FRAME |
| DINA4_P | FRAME |
| DINA5_L | FRAME |
| DINA5_P | FRAME |
| DOCFIELD | DOCUMENT FIELD |
| DOCSMAL | DOCSMAL |
| FRAME_A_L | FRAME A Size , 8 1/2 x 11 INCH, Landscape |
| FRAME_B_L | FRAME B Size , 11 x 17 INCH, Landscape |
| FRAME_C_L | FRAME C Size , 17 x 22 INCH, Landscape |
| FRAME_D_L | FRAME D Size , 22 x 34 INCH, Landscape |
| FRAME_E_L | FRAME E Size , 34 x 44 INCH, Landscape |
| LETTER_L | FRAME |
| LETTER_P | FRAME |
| RAHMEN3Q | FRAME |
| RAHMENA3Q | RAHMENA3Q |
| TABL_L | FRAME |
| TABL_P | FRAME |

Place the frame so that its lower left corner is at the origin, indicated by the small cross at $(x, y) = (0, 0)$.

Now add parts to the schematic. Click the Add Part tile again and find the p371_fa2021 library. One at a time, select parts and drop them onto the schematic. You'll need to click the stop button near the top of the window when you've added all the particular parts of one kind that you plan to use:

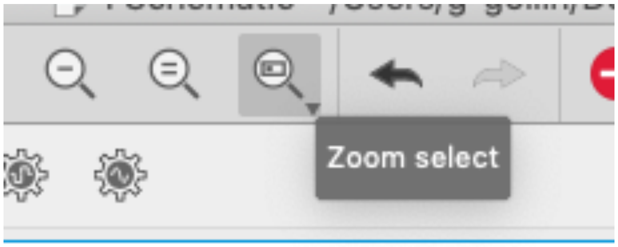

1 Schematic - /Users/g-gollin/Documents/EAGLE/projects/untitled.sch - EAGL
GO
?
Stop

In the schematic below I've inserted an Arduino, a BME680, a microSD card carrier, an LCD, and a contrast-setting trimpot (a variable resistor) for the LCD.

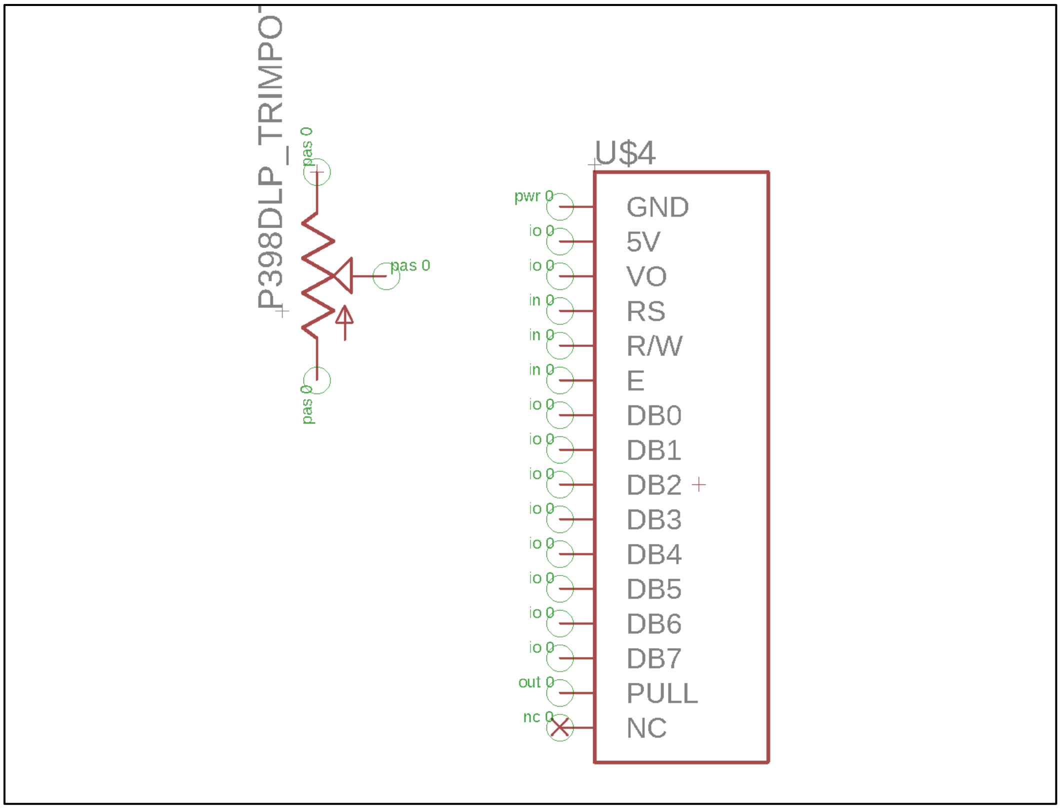


TITLE: untitled
Document Number:
REV:
Date: not saved!
Sheet: 1/1

Here’s a useful button:

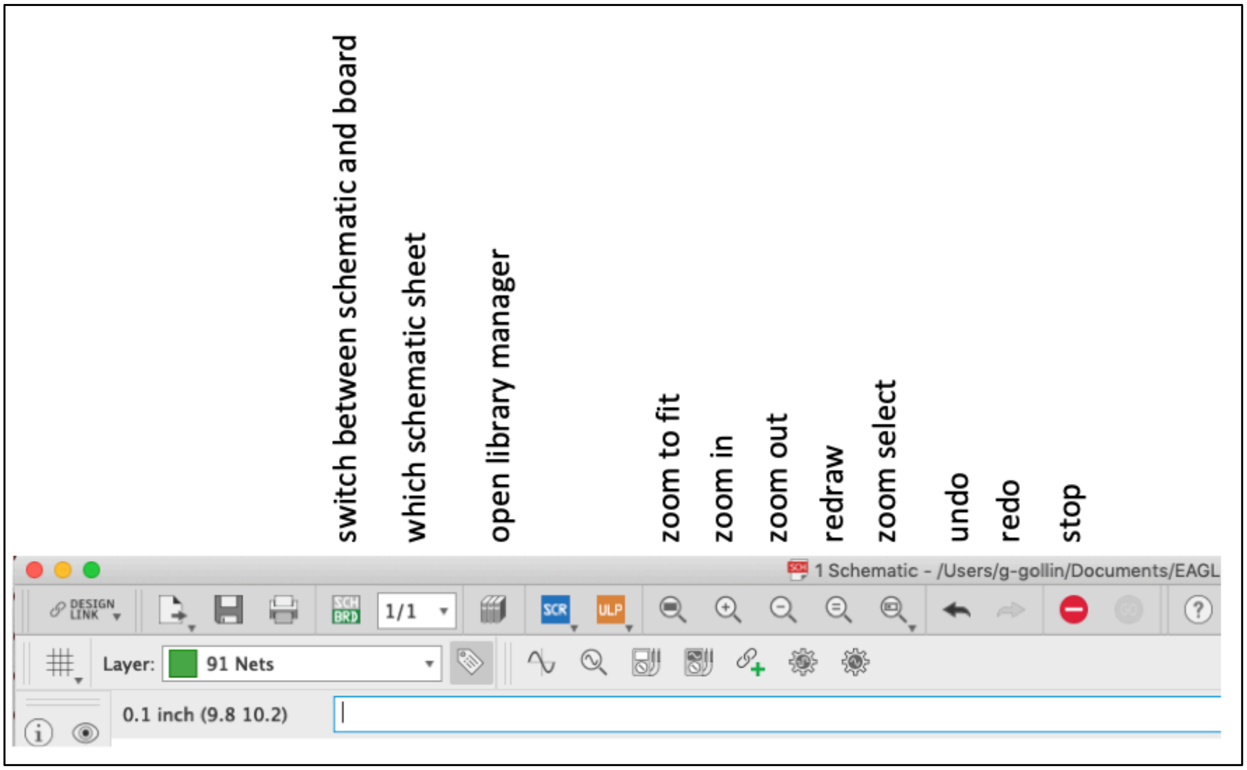


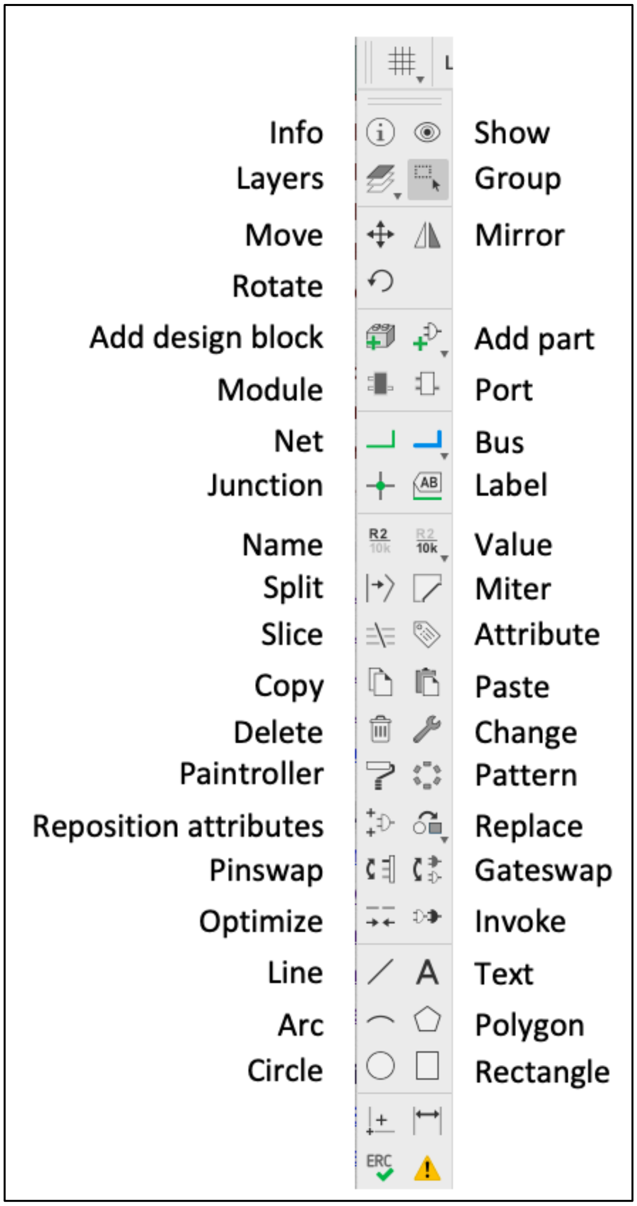
Use it to zoom in on a selected region of the schematic. For example:



Take note of the small cross just to the right of "DB2": that's the "origin" for this part, and is where you'll click to drag the part to a different position. (Click on the "Move" tile on the left side of the window.)

Here are the arrays of function tiles on the top and left sides of the window. I've labeled most of them for you.


switch between schematic and board
which schematic sheet
open library manager
zoom to fit
zoom in
zoom out
redraw
zoom select
undo
redo
stop
1 Schematic - /Users/g-gollin/Documents/EAGL
DESIGN LINK
SCH BRD
1/1
SCR
ULP
Layer:
91 Nets
0.1 inch (9.8 10.2)



The stop/interrupt button is especially useful: if EAGLE decides to ignore your commands and keeps beeping at you, click on it. You'll also probably find that you make a lot of use of the Group, Move, Copy, Paste, Net, Delete, and Change tiles.

Keep in mind that each part has an "origin" indicated with a small cross. You need to find this in order to be able to move the part. When you select several parts with the Group command, you can right-click on any selected part's origin and choose "Move group" from the popup menu.

The Change button opens another menu, offering a choice of what exactly you'd like to change.

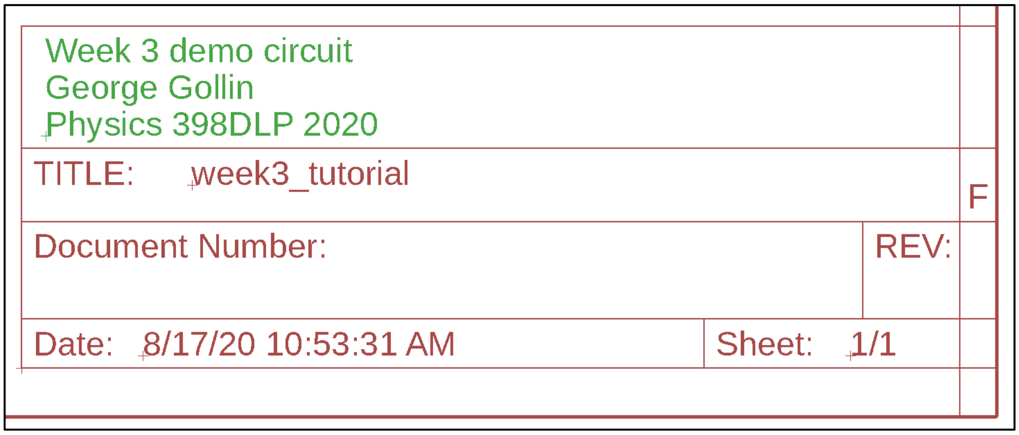
You can type commands (such as "name") into the text-accepting field below the zoom, etc. tiles.

Let’s say you’d like to add some text to the description box at the lower right of the week3_tutorial schematic. Do this: click on the “zoom selection” button, then drag a rectangle over the information box on the schematic. at the lower right.

Click the “T” tile in the left toolbar. Enter what you want, then click once to drop the text into place. Take note of the small + sign in the lower left corner of the text field. That indicates the location of the handle you can use to reposition or modify the text field. Click the red stop tile at the top of the window to avoid making a second copy of the text.

To change the size of the text, click the Change tile on the left of the page. To reposition the text click the Move tile, then click on the text field’s origin. If you are not happy with the positioning grid, go to the Grid tile in the upper left of the window and change the size away from 0.1 inches.

Now save your schematic, then go to View → Redraw. You’ll see that the filename and date appear in the box.



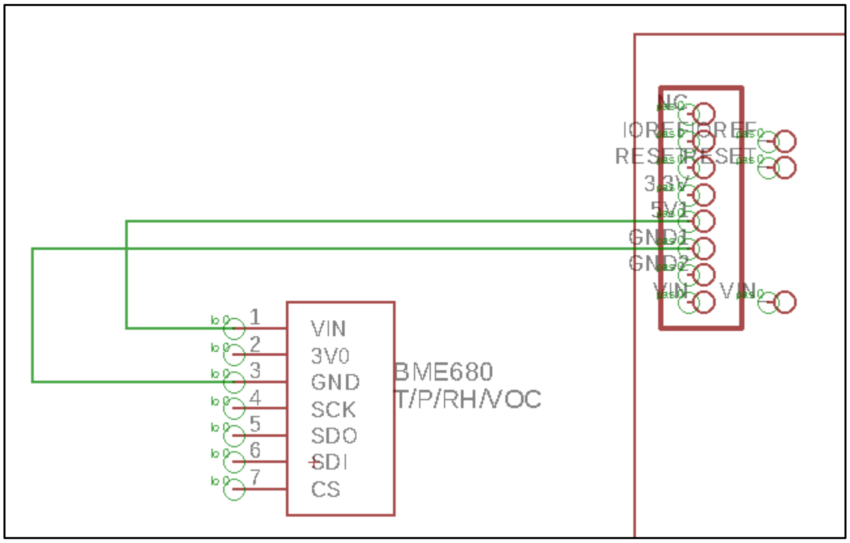
Week 3 demo circuit
George Gollin
Physics 398DLP 2020

TITLE: week3_tutorial

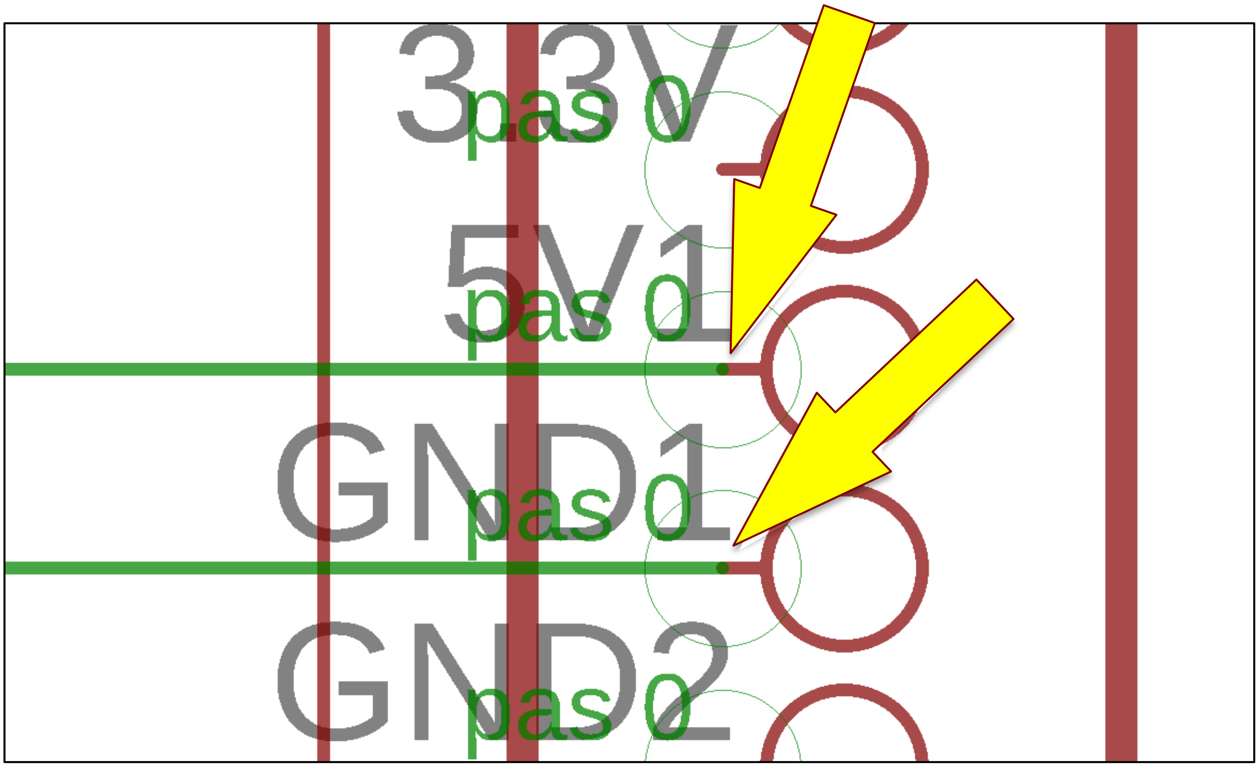
Document Number: REV:

Date: 8/17/20 10:53:31 AM Sheet: 1/1

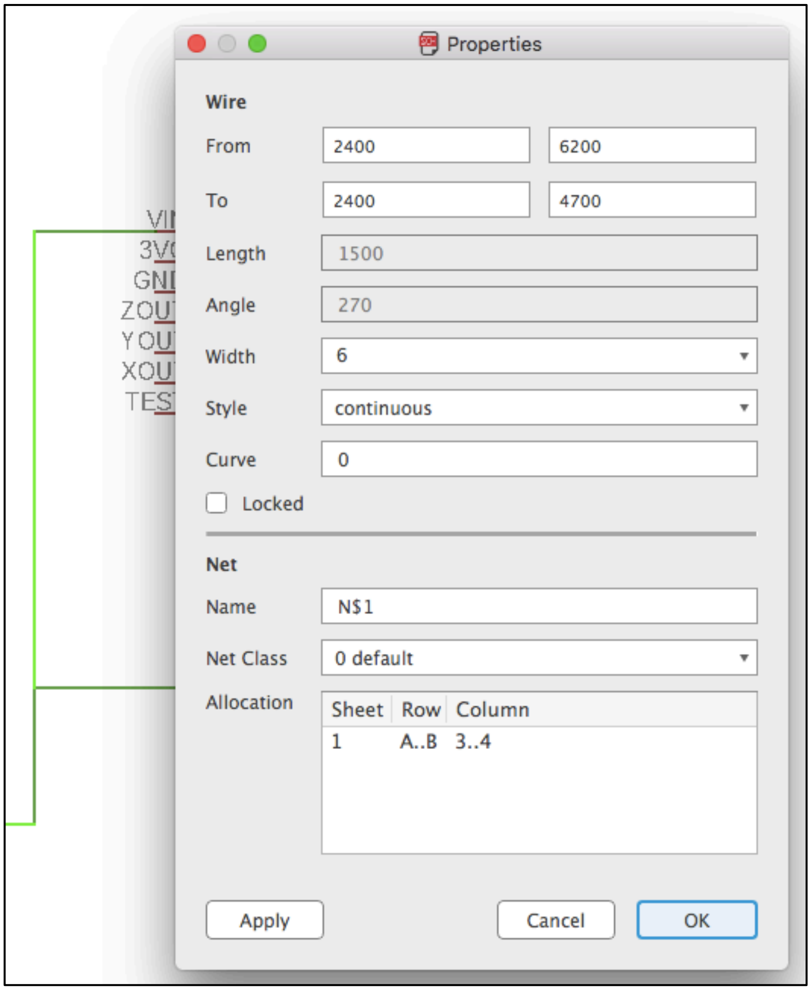
Let’s connect the VIN and GND pins on the BME680 to the Arduino’s 5V1 and GND1 pins. Position the parts where you would like them, then click once on the “net” button on the left toolbar. Run the net from the VIN to 5V1 pins; you can turn corners by clicking. Do the same for the GND connection. If you decide you don't like the placement of nets that you’ve established, click once on the “move” tile, then click once on the part of the net you’d like to move.



Here’s a close-up showing exactly where the connection between a net and a pin is made. Be careful to end the net on a pin, and not some other feature in the symbol for the part.



Now right click on one of the nets you’ve just drawn and select “properties.” You’ll see something like the following screen shot. Many of the fields can be edited. I’d suggest you change the net name from the uninformative “N$1” to something more descriptive like “5V.” Right click on the net a second time and select Name, then click in the “Place label” box.



Properties

**Wire**

| | | |
|---|---|---|
| From | 2400 | 6200 |
| To | 2400 | 4700 |
| Length | 1500 | |
| Angle | 270 | |
| Width | 6 | |
| Style | continuous | |
| Curve | 0 | |

□ Locked

**Net**

| | |
|---|---|
| Name | N$1 |
| Net Class | 0 default |

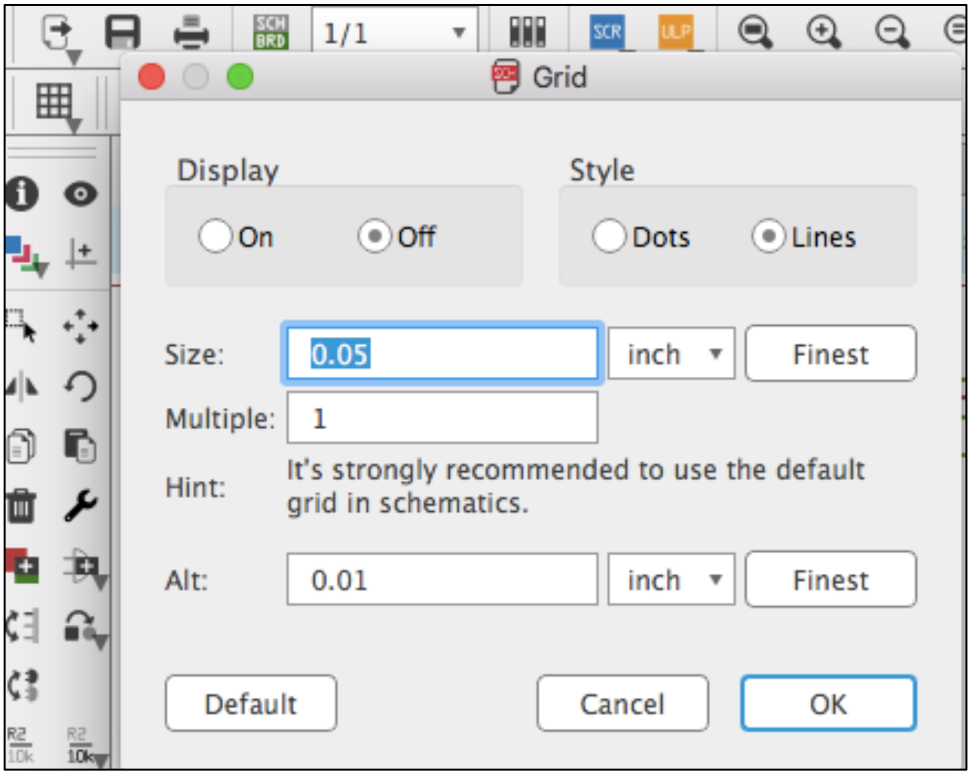
Allocation

| Sheet | Row | Column |
|---|---|---|
| 1 | A..B | 3..4 |

Now connect the rest of the pins as you've done on your breadboard. Naturally, the pins that you've left unconnected on your breadboard circuit will not be attached to any nets on your schematic. Label the nets.

It's generally a very good idea to put a capacitor between power and ground close to the inputs of each integrated circuit or, in our case, breakout board. So please put 0.1μF capacitors onto your schematic. You can find them in the "capacitor-wima" library among other places; consider using the C5/2.5 part, but you'll need to set the capacitance values by hand. You can connect them but attaching nets to their pins, then labeling the nets with the same names you had used for your 5V and ground nets. The nets don't actually need to be drawn so that they are attached to the VIN and GND pins of the sensors, as long as the net names are the same. To set the capacitances, right click on the devices, select "Value," and enter "0.1uF."

You've probably noticed that EAGLE forces objects to be placed on a grid. You can control the grid size using the control button that's immediately above the left side toolbar. I've reduced the grid size from 0.1 inches to 0.05 inches in my schematic.



Display: ○ On ● Off

Style: ○ Dots ● Lines

Size: 0.05 inch

Multiple: 1

Hint: It's strongly recommended to use the default grid in schematics.

Alt: 0.01 inch

***Wrap up: informative presentation for next week (need a group to volunteer!)***

For next week: I'd like one of the groups to give the class a ten-minute PowerPoint presentation on…

https://www.judaica-art.com/3728-custom_default/roses-and-jasmine-in-a-delft-vase-by-pierre-auguste-renoir-oil-painting-art-gallery.jpg

# Physics 371

Design Like a Physicist
Fall 2022

George Gollin
University of Illinois at Urbana-Champaign

## Weeks 4 - 14: Away we go!

## ☛ supplemental material ☚

### *Goals for weeks 4 – 14*

Goals are listed in the course syllabus. Briefly:

| Week | Goals |
|---|---|
| 1 | Organize; start project; work with Arduino |
| 2 | Intensive breadboarding; Python; TinkerCad |
| 3 | TinkerCad; DAQ work; EAGLE PCB |
| 4 | DAQ work; Python work; field tests; PCB assembly |
| 5 | 3D printing; analyze field test data; field test PCB |
| 6 | Production data taking; Python data verification |
| 7 | First conclusions from data; modify run plan as necessary |
| 8 | Detailed data analysis |
| 9 | Write “nearly final” draft of project report |
| 10 | Finish and submit “nearly final” draft of project report |
| 11 | Correct/modify project report |
| 12 | Finish “final report” |
| 13 | Rewrite final report, prepare PowerPoint presentation |
| 14 | Submit PowerPoint and ultimate report draft; give presentation |

### *Informative reports*

Most weeks one group will give a ten-minute (PowerPoint) presentation meant to inform us about some of the devices and conventions used by Physics 371 project groups. Possible topics (I may change these once you’ve chosen projects):

| Week | Report topic |
|---|---|
| 2 | BME680 T/P/RH sensors |
| 3 | I2C |
| 4 | Electret microphones |
| 5 | How a successive approximation ADC works |
| 6 | GPS |
| 7 | The Atmel ATmega2560 microcontroller |
| 8 | How an LSM9DS1 “9-axis” accelerometer works |
| 9 | PM2.5 particulates monitor |
| 10 | JPEG file format and compression |
| 11 | MLX90614 infrared sensor |

https://i.ebayimg.com/images/g/vGAAAOSweW5VbLXE/s-l1600.jpg

# Physics 371

Design Like a Physicist
Fall 2022

George Gollin
University of Illinois at Urbana-Champaign

# C++ and Python Primer/Refresher

# C++ and Python Primer/Refresher

Introduction: C++ and Python ....4
Install the Arduino programming IDE....4
Install and configure Anaconda Python....6
Representing structure in C++ and Python....9
Comments in C++ and Python ....10
Continuing a statement onto the next line in C++ and Python....11
Scope of variables....11
Some Pythonic surprises....12
  Exponentiation....12
  Indentation....13
  Array assignment....13
Libraries....14
The order in which things appear in your C++ and Python programs can differ....14
Structure of an Arduino program....15
A Python Refresher, from Physics 298owl ....17
Useful Python stuff....18
Basic concepts, mostly for Python ....19
  Variables and assignment statements ....19
  Kinds of variables....20
  Mathematical operations....21
  Logical operations ....23
  Scripts....23
  Lists and arrays....25
  Loops ....27
  A loop to calculate the sum of a few squares ....27
  Other loop matters ....28
  Functions and modules ....29
An exercise: an infinite series for π....31
Libraries....33
  Numpy ....33
  Matplotlib ....34
Drawing curves in two dimensions ....35
  How to draw a circle....35
  Drawing figures in new windows....36
Another exercise: graphing the magnification of a weird optical system ....37
Graphical representations of three dimensions....38
  Drawing a helix ....38

How to draw a surface whose height above the x-y plane depends on a function ....................40
Drawing a surface using polar coordinates....................................................................44
A final exercise: graphing a surface you'll see in General Relativity .........................................44

### *Introduction: C++ and Python*

I assume you already know how to program. If you've learned to code in python or C/C++, or Java, or some other language, you'll be fine. A B- or better in CS 101, CS 125, or Physics 298owl is a suitable prerequisite. It's also fine if you've learned on your own. But if you've never programmed before, or did poorly in an intro CS course, you should delay enrollment in Physics 371 until after you've done some coding.

We'll be working with two different languages: a slightly stripped-down version of C++ for programs to be executed by your Arduino and Anaconda's Python for developing the software to analyze your data.

There are minor differences in the two languages that you'll need to keep in mind. If you've worked with Java, you are already familiar with a language that requires you to declare the type of each variable you plan to use. C++ also requires this, but Python does not. Another difference is the use of a semicolon to end a line in C++ but not in Python. In addition, C++ uses curly brackets to define structure—which blocks of code are inside which other blocks of code—while Python uses indentation for this.

### *Install the Arduino programming IDE*

Go to the Arduino website https://www.arduino.cc/ and navigate to https://www.arduino.cc/en/Guide/ArduinoMega2560. Download and install the Arduino Desktop IDE (Integrated Development Environment) on your laptop.

Connect an Arduino to a USB port in your laptop. The Arduino probably comes with a blink-an-LED program preloaded, so a yellow LED near the USB connector might start blinking as soon as the board is powered.

You'll need to go to the Tools → Port menu to select which communication channel your laptop will use to talk to the Arduino. While you're at it, open a serial monitor window by following Tools → Serial Monitor. The Arduino will write information to this window as instructed by the program it is running.

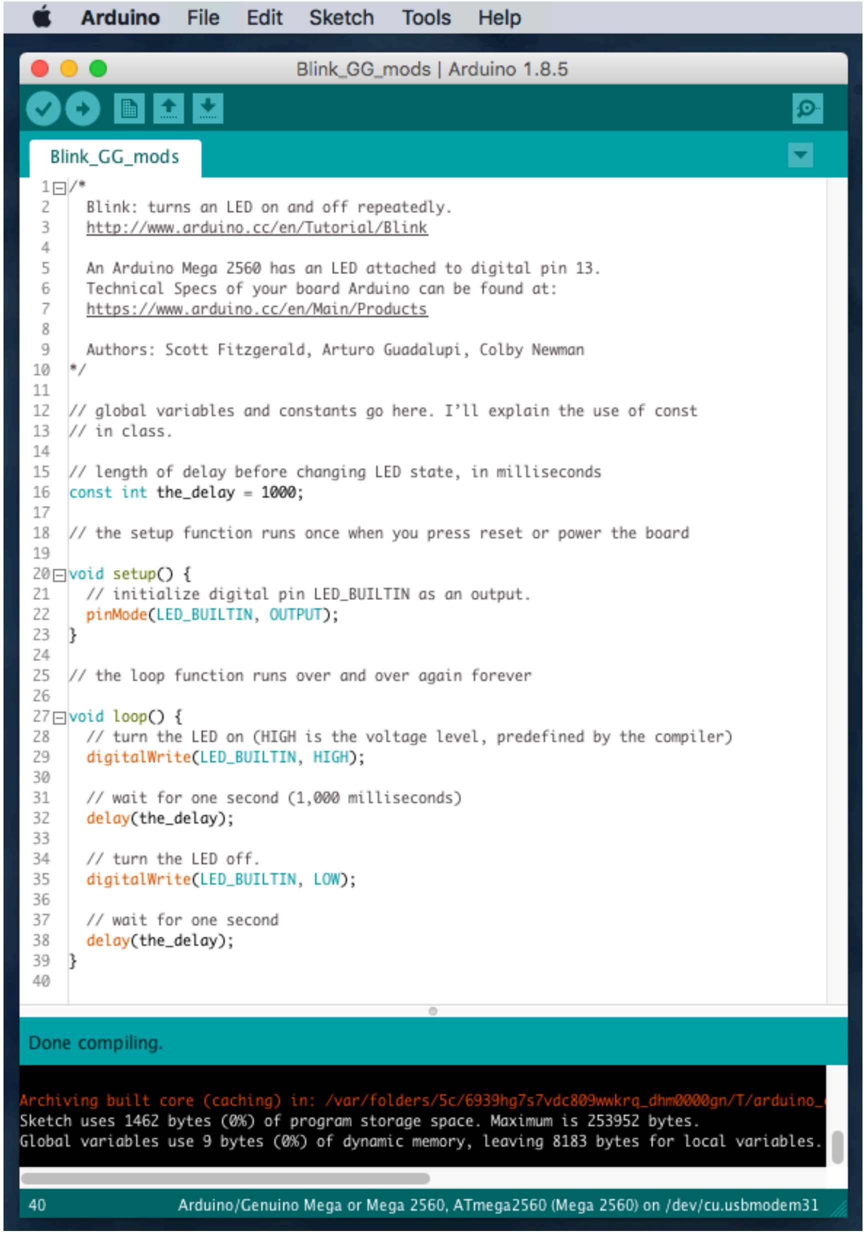
Please create a folder in which you will store your various Arduino programs. (For some reason people call an Arduino program a sketch. I think that sounds silly.)

You can find Arduino tutorials and sample programs at https://www.arduino.cc/en/Tutorial/BuiltInExamples.

See also File → Examples → 01.Basics → Blink for a ready-to-run program, which (after a few modifications) looks like this:

```
/*
  Blink: turns an LED on and off repeatedly.
  http://www.arduino.cc/en/Tutorial/Blink

  An Arduino Mega 2560 has an LED attached to digital pin 13.
  Technical Specs of your board Arduino can be found at:
  https://www.arduino.cc/en/Main/Products

  Authors: Scott Fitzgerald, Arturo Guadalupi, Colby Newman
*/

// global variables and constants go here. I'll explain the use of const
// in class.

// length of delay before changing LED state, in milliseconds
const int the_delay = 1000;

// the setup function runs once when you press reset or power the board

void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever

void loop() {
  // turn the LED on (HIGH is the voltage level, predefined by the compiler)
  digitalWrite(LED_BUILTIN, HIGH);

  // wait for one second (1,000 milliseconds)
  delay(the_delay);

  // turn the LED off.
  digitalWrite(LED_BUILTIN, LOW);

  // wait for one second
  delay(the_delay);
}
```

Here's how it looks in the IDE's editor:



```
/*
  Blink: turns an LED on and off repeatedly.
  http://www.arduino.cc/en/Tutorial/Blink

  An Arduino Mega 2560 has an LED attached to digital pin 13.
  Technical Specs of your board Arduino can be found at:
  https://www.arduino.cc/en/Main/Products

  Authors: Scott Fitzgerald, Arturo Guadalupi, Colby Newman
*/

// global variables and constants go here. I'll explain the use of const
// in class.

// length of delay before changing LED state, in milliseconds
const int the_delay = 1000;

// the setup function runs once when you press reset or power the board

void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever

void loop() {
  // turn the LED on (HIGH is the voltage level, predefined by the compiler)
  digitalWrite(LED_BUILTIN, HIGH);

  // wait for one second (1,000 milliseconds)
  delay(the_delay);

  // turn the LED off.
  digitalWrite(LED_BUILTIN, LOW);

  // wait for one second
  delay(the_delay);
}

```

Done compiling.

```
Archiving built core (caching) in: /var/folders/5c/6939hg7s7vdc809wwkrq_dhm0000gn/T/arduino_
Sketch uses 1462 bytes (0%) of program storage space. Maximum is 253952 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 8183 bytes for local variables.
```

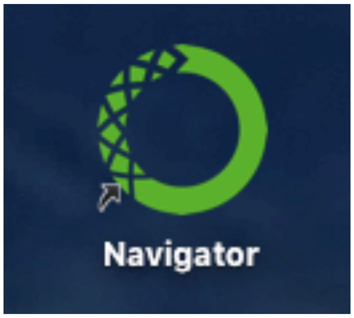
40
Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/cu.usbmodem31

Click on the check mark to compile the program; click on the right arrow button to compile it and download the executable to the Arduino.

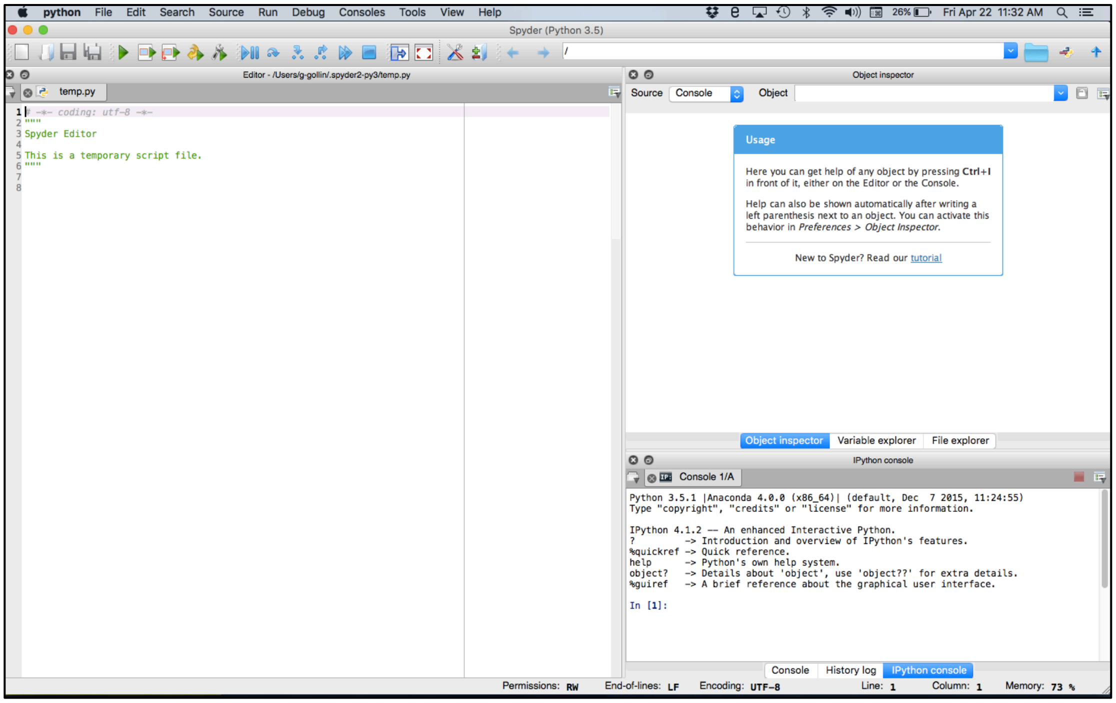
Please open the Blink example, then save it to the folder you've created to hold your programs. Compile and download it to confirm that it works. We'll come back to this later.

### *Install and configure Anaconda Python*

Please download the installation file for the most recent version of Anaconda Python, available here: https://www.anaconda.com/download/. On a Mac the installer will create an icon/shortcut to "Anaconda Navigator" that will allow you to launch applications. On a Windows machine you might need to access Navigator here: Start → All Programs → Anaconda3 (64-bit) → Anaconda Navigator.



The Anaconda software contains a number of different programs. We will be working with spyder, the "Scientific Python Development Environment." This is an integrated development environment (IDE), which incudes an editor, a control console, a debugger, a table of program variables, and other tools.
Click on the spyder launch button in the navigator window. The development environment workspace will open.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

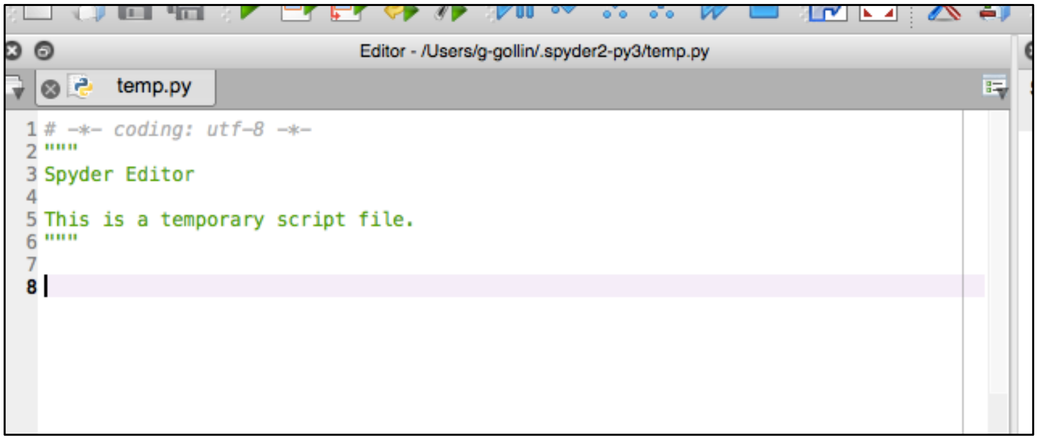
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

```
Python 3.5.1 |Anaconda 4.0.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

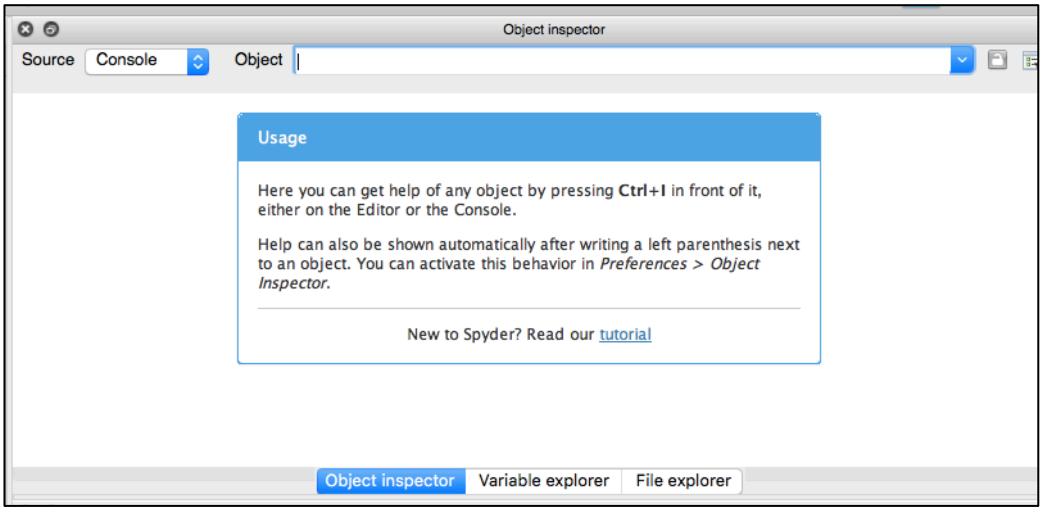
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 1 Column: 1 Memory: 73 %

The window on the left is an editor, which you will use to create script files (program files containing executable instructions. The paired triple quotes enclose comments). Here's a screen shot of part of it.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

The upper right window allows you to look at the contents of "objects," variables, and file directories. Note the tabs at the bottom of the window for selecting what is shown. You will probably find the file and variable explorer tabs most useful. Sometimes the file modification dates shown by file explorer do not update when I save changes to a file! That is surely a bug.



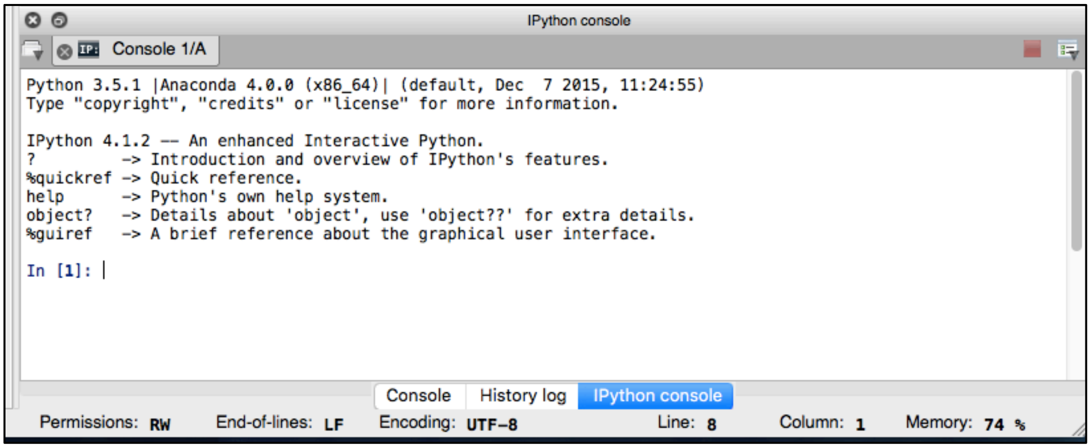
Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector.*

New to Spyder? Read our tutorial

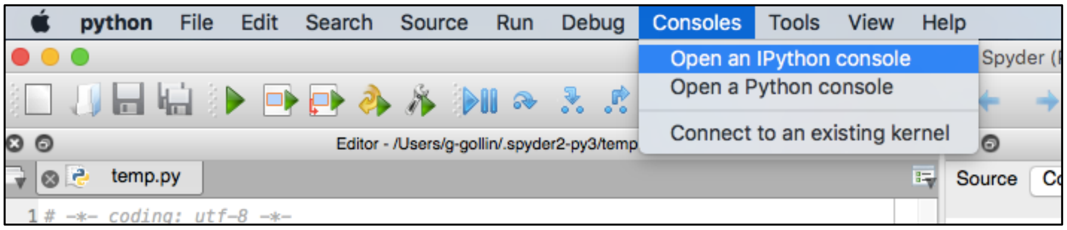
The lower right window shows an iPython console, a sort of operator's station from which you can issue commands to Python. It also displays program output.



```
Python 3.5.1 |Anaconda 4.0.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: |
```

Permissions: RW    End-of-lines: LF    Encoding: UTF-8    Line: 8    Column: 1    Memory: 74 %

If you trash the iPython console by mistake you can open a new one through the "Consoles" menu at the top of the workspace window.



There are a few parameters that you should set. Go to the Python preferences menu and do this:
preferences : run : default working directory
set to a sensibly-named folder that will hold all your scripts

preferences : current working directory : console directory
set to the same folder as that which will hold your scripts

preferences : iPython console : graphics : Backend
set to "Automatic"

preferences : History log : Settings
set "History depth" to 2000 entries

Quit spyder, then restart.

### *Representing structure in C++ and Python*

The two languages use different conventions for defining when a block of code lives inside another block of code. C++ uses curly braces and semicolons ("{", "}", and ";") while Python uses indentation. The following three code snippets (the first two are C++ for the Arduino, the third is Python) are functionally equivalent.

```
// C++
bool debug_it = true;
Serial.begin(9600);
if (debug_it) {
Serial.print("Debug flag is set ");
Serial.println("for some reason.");
}
Serial.print("This line always prints. ");

// C++
bool debug_it = true;
Serial.begin(9600);
if (debug_it) {
      Serial.print("Debug flag is set ");
      Serial.println("for some reason.");
      }
Serial.print("This line always prints. ");

# Python
debug_it = true
if (debug_it):
      print("Debug flag is set ", "for some reason.")
print("This line always prints.")
```

This snippet isn't, however:

```
# Python
debug_it = true
if (debug_it):
      print("Debug flag is set ", "for some reason.")
      print("This line only prints when debug_it is true."
```

### *Comments in C++ and Python*

C++ single-line comments begin with a pair of forward slashes, as shown above. You can also define a C++ multiline comment this way:

```
/*
This is a multi-
line comment
in C++.
*/
```

In Python single line comments begin with an octothorpe ("#") as shown above. Multiline comments are delimited with starting and ending triples of double quotes:

```
"""
This is a multi-
line comment
in Python.
"""
```

### *Continuing a statement onto the next line in C++ and Python*

For improved readability, you'll want to break overly long lines of code, continuing them onto the next line. This is especially easy in C++ since the compiler doesn't consider a statement to end until the compiler encounters a semicolon. In Python you'll need to use a backslash to break the line. For example:

```
// C++ continuation example
Serial.print(
"They call me Bond. James Bond. "
);

# Python continuation example
print( \
"They call me Bond. James Bond. " \
)
```

### *Scope of variables*

This is the stuff of headaches. Some variables are global, and knowable by all functions in a code file. Others are local, and known only inside the function in which they are defined.

Here's a Python example, in which the variable "text" is used as a local variable in functions f1 and f2, and a global variable in f3, as well as the main program.

```
# define "text" here, which makes it a global variable.
text = "text is a global variable, defined as this string."

#define functions f1, f2, and f3... "\n" is the newline character.
def f1():
     print("now printing 'text' from inside f1.")
     text = "This is 'text' as defined locally inside function f1."
     print(text, "\n")
def f2():
     print("now printing 'text' from inside f2.")
     text = "This is 'text' as defined locally inside function f2."
     print(text, "\n")
def f3():
     print("now printing 'text' (which is a global variable) from inside f3.")
     print(text, "\n")

print("about to call f1.")
f1()
print("about to call f2.")
f2()
print("about to call f3.")
f3()
print("now print 'text' from top level program.")
print(text)
```

The program's output follows:

```
about to call f1.
now printing 'text' from inside f1.
This is 'text' as defined locally inside function f1.

about to call f2.
now printing 'text' from inside f2.
This is 'text' as defined locally inside function f2.

about to call f3.
now printing 'text' (which is a global variable) from inside f3.
text is a global variable, defined as this string.

now print 'text' from top level program.
text is a global variable, defined as this string.
```

Scoping works similarly in C++. When you encounter mysterious behavior in your code, consider looking at it with an eye towards a problem with the scope of a variable.

### *Some Pythonic surprises*

Sometimes Python will surprise you. Here are a few of the things that I have tripped over.

#### *Exponentiation*

Python's exponentiation operator is **. You might be tempted to use ^ for exponentiation, but that's not correct: the symbol "^" performs a bit-by-bit exclusive-OR between the two variables.

An example:

```
In [13]: 3**3
Out[13]: 27
In [14]: 3^3
Out[14]: 0
```

*Indentation*

Watch out for this! The following code

```
sum = 0
for i in range(1,100000):
     sum += i
print(sum)
```

will print a single line of output, while

```
sum = 0
for i in range(1,100000):
     sum += i
     print(sum)
```

will print 100,000 lines.

*Array assignment*

In most languages with which I am familiar, assigning a new variable to equal an existing variable creates a copy of the original variable in a different storage location. For example, in C++ the code snippet

```
char a[ ] = {'u', 'v', 'w'}; char b[3];
// copy a into b using the memcpy function (b = a isn't allowed for C++ arrays)
memcpy(b, a, 3);
Serial.println(b[0]);
a[0] = 'x';
Serial.println(a[0]);
Serial.println(b[0]);
```

produces the following output.

```
u
x
u
```

Note that modifying a[0] does not affect b[0].

It is different in Python, in which assigning a "new" array just provides an alternate name for the same locations in memory. The following Python code…

```
import numpy as np
a = np.array([10, 20, 30])
print("a = ", a)
b = a
b[0] = 5
print("a = ", a)
```

…yields the following output.

```
a =  [10 20 30]
a =  [ 5 20 30]
```

In Python, changing b also changes a.

### *Libraries*

In Python you load libraries of useful stuff using import commands, generally placed near the top of your program file. For example:

```
import numpy as np
…
a =  np.array([10, 20, 30])
```

In C++ you load libraries with the include compiler directive:

```
#include <Adafruit_MCP4725.h>
…
// instantiate a DAC object named "dac":
Adafruit_MCP4725 dac;
```

You can manage your Arduino C++ libraries through the IDE (Integrated Development Environment) menu path Sketch → Include Library.

### *The order in which things appear in your C++ and Python programs can differ*

It would be handy if Python were able to read through your code in a first pass, picking up the identities of the various functions you define, before beginning to execute your code. That way, you'd be able to put your main program near the top of the file, and append new functions as you write them at the end of your file. But Python is an interpreted, not a compiled language, and the Python interpreter doesn't jump around in your file to figure out what is where. In the scoping example we just discussed, putting the functions f1, f2, and f3 at the end of the file will throw all sorts of annoying error messages.

C++ is a compiled language: the compiler produces a machine language executable program, rather than reading (and parsing) a “script” of instructions. The compiler is able to identify the components of your program, so it would be perfectly fine to place the functions f1, f2, and f3 at the end of your file.

### *Structure of an Arduino program*

The Arduino IDE's compiler expects to find functions named "setup" and "loop" somewhere in your program file. Setup is executed only once, immediately after the program starts running. After exiting setup, the compiler will execute loop; each time the program leaves loop it will immediately reenter the routine.

I suggest you organize the content of a program file as follows:

1. a block of explanatory comments
2. include directives for libraries
3. definitions of global variables and instantiations of objects representing hardware devices
4. setup function
5. loop function
6. other functions

Here's an example that will blink the yellow LED on the Arduino circuit board that is connected to the Arduino's pin 13. (You can find the code on the course's "Code & design repository" web page.) You'll want to open a 9600 baud serial monitor window after connecting to the Arduino: follow the menu path Tools → Port, and then Tools → Serial Monitor.

```
/*
  Blink the yellow LED that is driven by an opamp attached to the
  Arduino Mega 2560's pin 13.

  George Gollin, University of Illinois, January 7, 2019.
*/

///////////////////////// includes //////////////////////////////

// include a library for one device you'll eventually be using, namely
// an INA219 current/voltage monitor breakout board. I don't actually
// do anything with it in this program.

#include <Adafruit_INA219.h>

///////////////////////// instantiations //////////////////////////////

// instantiate a current sensor object named "ina219":
Adafruit_INA219 ina219;
```

```
///////////////////////// globals //////////////////////////////

// approximate on and off time, neglecting time to enter/exit loop
// I do not expect to change these, so declare them as constants.
const int on_milliseconds = 250;
const int off_milliseconds = 750;

// pin number for the LED
const int LED_pin = 13;

// flashes so far... note that this is a two-byte signed integer and
// will get weird after 32,767.
int flashes_so_far;

///////////////////////// setup //////////////////////////////

// The setup function runs once when you press reset, or power the board.
// Since it doesn't return a value, declare it as type "void"

void setup() {

  // fire up the serial monitor, set to 9600 baud.
  Serial.begin(9600);

  // initialize digital pin LED_pin as an output.
  pinMode(LED_pin, OUTPUT);

  // Initialize the INA219 current/voltage monitor (You probably
  // don’t have one of these yet.)
  ina219.begin();

  // initialize a (global) variable...
  flashes_so_far = 0;

  // print a message. println puts a return at the end of the line.
  Serial.println("All done with setup.");
}

///////////////////////// loop //////////////////////////////

// the loop function runs over and over again, forever

void loop() {

  // turn the LED off. "HIGH" and "LOW" are system-defined.
  digitalWrite(LED_pin, LOW);

  // now wait.
  delay(off_milliseconds);

  // turn the LED on.
  digitalWrite(LED_pin, HIGH);

  // now wait.
  delay(on_milliseconds);
```

```
    // increment a counter, just for fun.
    flashes_so_far++;

    // every fifth flash call a function. % is the modulus function.
    if (flashes_so_far % 5 == 0) {
      print_something(flashes_so_far);
    }
  }

//////////////////////// print_something ////////////////////////////

// the print_something function just prints a line when called.

void print_something(int the_number) {

  Serial.print("Number of LED flashes so far: ");
  Serial.println(the_number);
}
```

Output to the serial monitor looks like this:

```
All done with setup.
Number of LED flashes so far: 5
Number of LED flashes so far: 10
Number of LED flashes so far: 15
Number of LED flashes so far: 20
Number of LED flashes so far: 25
```

etc.

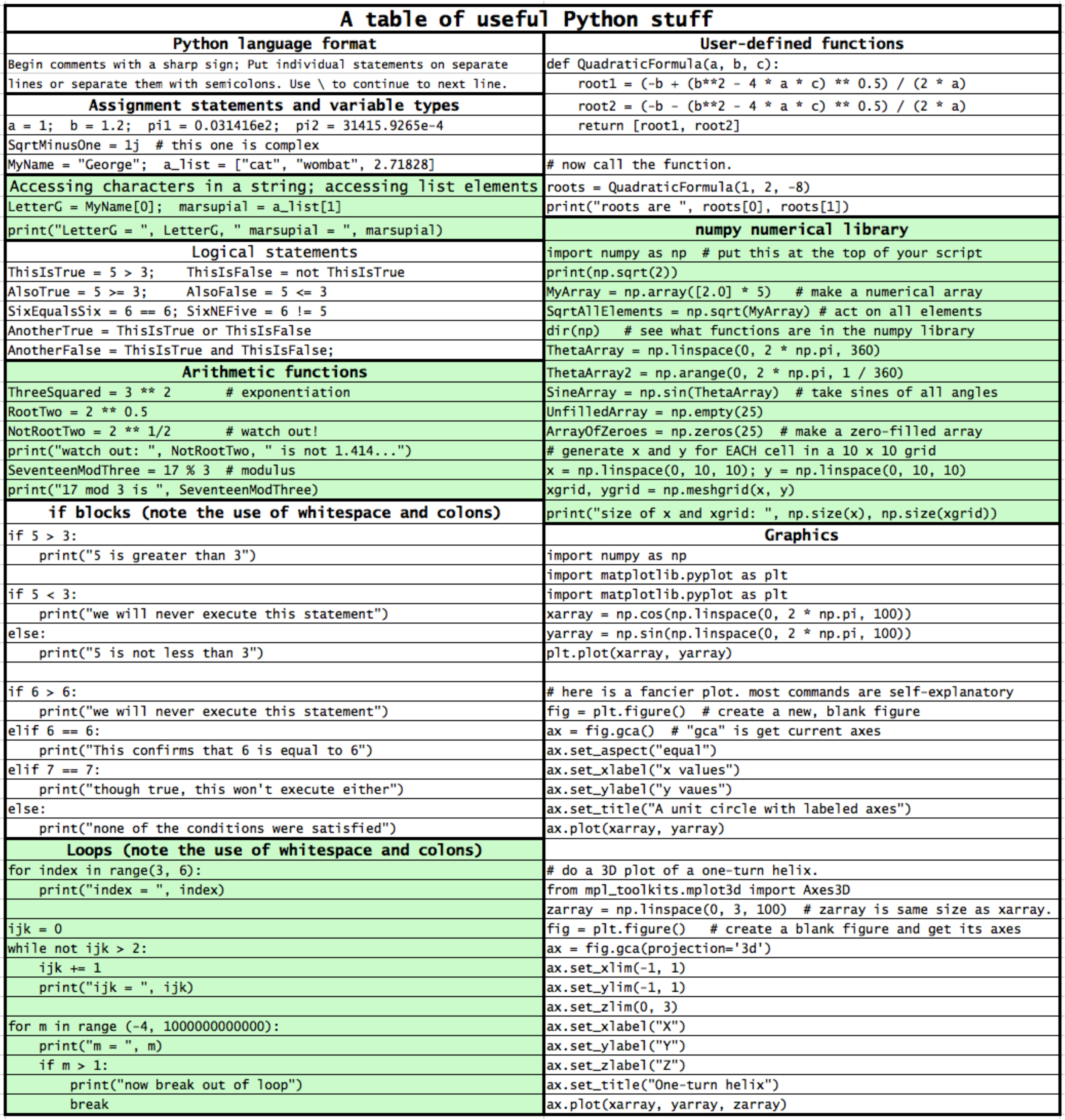
# *A Python Refresher, from Physics 298owl*

Our goals are to :

- Install Anaconda's spyder Python developer's environment on your laptop;
- Experiment with Python by typing commands directly into the iPython console;
- Learn about some of the basic tools in programing, including loops, conditional statements, and mathematical operations;
- Write and execute a program that sums (part of) an infinite series for $\pi$;
- Use the numpy and matplotlib libraries to generate graphical representations of arrays of points, and continuous functions of one and two variables;
- Graph a couple of peculiar functions, one of which (as you will learn during the next unit) relates to the spacetime curvature induced by general relativistic effects in the vicinity of a massive object.

### *Useful Python stuff*

The following table is taken from my Physics 298owl material.



| **A table of useful Python stuff** | |
|---|---|
| **Python language format** | **User-defined functions** |
| Begin comments with a sharp sign; Put individual statements on separate | def QuadraticFormula(a, b, c): |
| lines or separate them with semicolons. Use \ to continue to next line. | root1 = (-b + (b**2 - 4 * a * c) ** 0.5) / (2 * a) |
| **Assignment statements and variable types** | root2 = (-b - (b**2 - 4 * a * c) ** 0.5) / (2 * a) |
| a = 1;  b = 1.2;  pi1 = 0.031416e2;  pi2 = 31415.9265e-4 | return [root1, root2] |
| SqrtMinusOne = 1j  # this one is complex | |
| MyName = "George";  a_list = ["cat", "wombat", 2.71828] | # now call the function. |
| Accessing characters in a string; accessing list elements | roots = QuadraticFormula(1, 2, -8) |
| LetterG = MyName[0];  marsupial = a_list[1] | print("roots are ", roots[0], roots[1]) |
| print("LetterG = ", LetterG, " marsupial = ", marsupial) | **numpy numerical library** |
| Logical statements | import numpy as np  # put this at the top of your script |
| ThisIsTrue = 5 > 3;    ThisIsFalse = not ThisIsTrue | print(np.sqrt(2)) |
| AlsoTrue = 5 >= 3;     AlsoFalse = 5 <= 3 | MyArray = np.array([2.0] * 5)   # make a numerical array |
| SixEqualsSix = 6 == 6; SixNEFive = 6 != 5 | SqrtAllElements = np.sqrt(MyArray) # act on all elements |
| AnotherTrue = ThisIsTrue or ThisIsFalse | dir(np)   # see what functions are in the numpy library |
| AnotherFalse = ThisIsTrue and ThisIsFalse; | ThetaArray = np.linspace(0, 2 * np.pi, 360) |
| **Arithmetic functions** | ThetaArray2 = np.arange(0, 2 * np.pi, 1 / 360) |
| ThreeSquared = 3 ** 2       # exponentiation | SineArray = np.sin(ThetaArray)  # take sines of all angles |
| RootTwo = 2 ** 0.5 | UnfilledArray = np.empty(25) |
| NotRootTwo = 2 ** 1/2       # watch out! | ArrayOfZeroes = np.zeros(25)  # make a zero-filled array |
| print("watch out: ", NotRootTwo, " is not 1.414...") | # generate x and y for EACH cell in a 10 x 10 grid |
| SeventeenModThree = 17 % 3  # modulus | x = np.linspace(0, 10, 10); y = np.linspace(0, 10, 10) |
| print("17 mod 3 is ", SeventeenModThree) | xgrid, ygrid = np.meshgrid(x, y) |
| **if blocks (note the use of whitespace and colons)** | print("size of x and xgrid: ", np.size(x), np.size(xgrid)) |
| if 5 > 3: | **Graphics** |
|     print("5 is greater than 3") | import numpy as np |
| | import matplotlib.pyplot as plt |
| if 5 < 3: | import matplotlib.pyplot as plt |
|     print("we will never execute this statement") | xarray = np.cos(np.linspace(0, 2 * np.pi, 100)) |
| else: | yarray = np.sin(np.linspace(0, 2 * np.pi, 100)) |
|     print("5 is not less than 3") | plt.plot(xarray, yarray) |
| | |
| if 6 > 6: | # here is a fancier plot. most commands are self-explanatory |
|     print("we will never execute this statement") | fig = plt.figure()  # create a new, blank figure |
| elif 6 == 6: | ax = fig.gca()  # "gca" is get current axes |
|     print("This confirms that 6 is equal to 6") | ax.set_aspect("equal") |
| elif 7 == 7: | ax.set_xlabel("x values") |
|     print("though true, this won't execute either") | ax.set_ylabel("y vaues") |
| else: | ax.set_title("A unit circle with labeled axes") |
|     print("none of the conditions were satisfied") | ax.plot(xarray, yarray) |
| **Loops (note the use of whitespace and colons)** | |
| for index in range(3, 6): | # do a 3D plot of a one-turn helix. |
|     print("index = ", index) | from mpl_toolkits.mplot3d import Axes3D |
| | zarray = np.linspace(0, 3, 100)  # zarray is same size as xarray. |
| ijk = 0 | fig = plt.figure()   # create a blank figure and get its axes |
| while not ijk > 2: | ax = fig.gca(projection='3d') |
|     ijk += 1 | ax.set_xlim(-1, 1) |
|     print("ijk = ", ijk) | ax.set_ylim(-1, 1) |
| | ax.set_zlim(0, 3) |
| for m in range (-4, 1000000000000): | ax.set_xlabel("X") |
|     print("m = ", m) | ax.set_ylabel("Y") |
|     if m > 1: | ax.set_zlabel("Z") |
|         print("now break out of loop") | ax.set_title("One-turn helix") |
|         break | ax.plot(xarray, yarray, zarray) |

You should add to this table as you come upon other useful bits of Python wisdom.

### *Basic concepts, mostly for Python*

Take a look at the table of useful Python stuff on the inside front cover of the course packet. I am going to go through most of the information presented there, but quickly so we can being writing code.

*Variables and assignment statements*

A variable is a name assigned to one location in memory. You manipulate the contents of that memory location by referring to it by the name of the variable. For example, to ***associate*** the name "A" with a location in memory, then ***assign it*** the value 12, you would type the following into the iPython console window.

```
A=12
```

The computer does something analogous to the "copy a1, a2" machine instruction we discussed earlier, with a1 holding the address of a word in memory that contains the integer 12, and a2 holding the memory address that has been assigned to the variable A.

To define a new variable as the sum of **A** and the number 4 you would type:

```
B=A+4
```

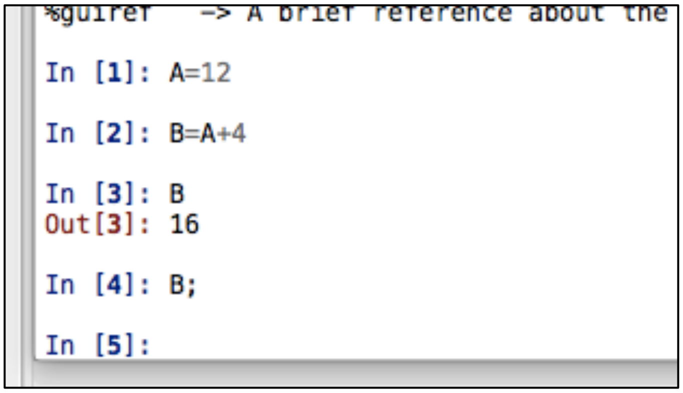
To inspect the value of **B** you would just type its name into the console:

```
B
```

Note that a semicolon at the end of a line suppresses the normal output produced in response to that line:

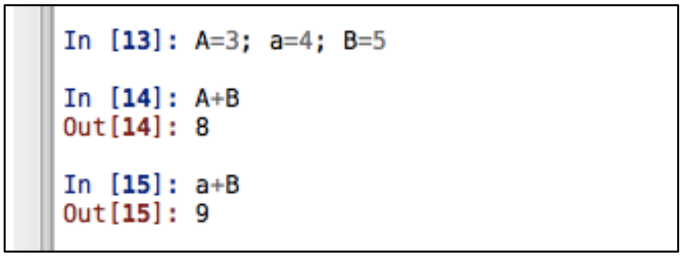
```
B;
```

yields no output. Here is a screen shot of the console with the above commands.



```
%guiref   -> A brief reference about the

In [1]: A=12

In [2]: B=A+4

In [3]: B
Out[3]: 16

In [4]: B;

In [5]:
```

You can place multiple assignments on a single line by separating them with semicolons. Note that variable names are case sensitive. Take a look:



```
In [13]: A=3; a=4; B=5

In [14]: A+B
Out[14]: 8

In [15]: a+B
Out[15]: 9
```

Keep in mind that an equal sign in Python is actually an assignment of value, and not the same thing as an equation expressing the equivalence of the left and right sides. For example, to increment the value of **`A`** by **`1`** we'd do this:

```
A=A+1
```

*Kinds of variables*

There are many different kinds of variables that are defined in Python. For example, the statement

```
A=12         # inline comments begin with an octothorpe
```

defines an *integer* variable. The statement

```
C=2.71828    # C is a floating point variable
```

defines a *floating point* variable, a numerical variable which is allowed to take on non-integer values. The statement

```
D=(1+2j)     # D is complex
```

defines a *complex* variable with the value 1 + 2*i*. ( $i = \sqrt{-1}.$ ) Note the use of **`j`** instead of **`i`**. It is fine to mix together integer, floating point, and complex numbers in arithmetic statements:

```
In [1]: A = 12;
In [2]: B = 2.5;
In [3]: C = (5 + 7j);
In [4]: A + B + C
Out[4]: (19.5+7j)
```

The statement

```
MyName=”George”     # a string!
```

defines a *string*. You may use single quotes if that is your preference. It is fine to enclose whitespace and single quotes inside double-quoted strings:

```
In [1]: AnotherString = "George's car"
In [2]: print(AnotherString)
George's car
```

A string is really a list of individual characters; you can access the $n^{th}$ character in a string this way (note that position 0 yields the first character):

```
In[11]: AnotherString[2]
Out[11]: 'o'
In[12]: AnotherString[0]
Out[12]: 'G'
```

*Boolean* (logical) variables can only take the values True and False.

```
In [1]: ObviouslyTrue = 3 > 2; print(ObviouslyTrue)
True
```

Python is able to convert most variables from one type to another as necessary.

*Mathematical operations*

Here are examples of some of the mathematical operations that Python supports. Many are self explanatory.

```
In [1]: a=8+9; print(a)          # addition, with two statements on one line!
17

In [2]: a=8/9; print(a)
0.8888888888888888

In [3]: A=3**2; print(A)         # ** means exponentiation. NB: ^ is NOT!!
9

In [4]: print(25**0.5)           # one way to take a square root
5.0

In [4]: print(pow(25,0.5))       # another way: “pow” is power
5.0
```

The % sign is used to determine the modulus of one number with respect to another. What I mean is this: the value of *a* % *b* is the remainder when *a* is divided by *b*. Some examples:

```
In[1]: 7 % 4
Out[1]: 3
In[2]: 14 % 7
Out[2]: 0
```

```
In[3]: 13 % 7
Out[3]: 6
```

You may need to import a *module* of routines that aren't already known to Python. Your Python installation includes lots of these, and Python knows how to find them if you use the import command. You will eventually find it convenient to define some of your own modules. (That's for later!) Here's how this works.

```
In [1]: print(sqrt(25))          # this won't work yet
NameError: name 'sqrt' is not defined
In [2]: import numpy as np
In [3]: print(np.sqrt(25))       # now it will work.
5.0
```

Keep in mind that your computer's internal workings use binary, not decimal, so sometimes there can be surprises. For example, the internal representation of 0.1 is inexact, as you can see in the following:

```
In[1]: 0.1 + 0.2
Out[1]: 0.30000000000000004
```

There are ways to improve the precision used by Python in its calculations, but the language isn't nearly as versatile as some others in its options for greater accuracy. For now, keep in mind that sometimes zero isn't quite zero:

```
In[1]: 0.3 - 0.1 - 0.2
Out[1]: -2.7755575615628914e-17
In[2]: abs(0.3 - 0.1 - 0.2) == 0
Out[2]: False
In[3]: abs(0.3 - 0.1 - 0.2) < 1.e-16
Out[3]: True
```

Here is something you can do to learn the level of precision offered by your computer's Python. (A "floating point" number is a real number with a decimal point. A "long double precision" number is a floating point number with a few extra digits of precision on Macs and some (but not all) windows machines. First import "numpy," a built-in numerical python module.

```
In[1]: import numpy as np  # import the numpy module, refer to it as “np”

In[2]: np.finfo(np.float)        # ask for information about floats
Out[2]: finfo(resolution=1e-15, min=-1.7976931348623157e+308,
max=1.7976931348623157e+308, dtype=float64)

In[3]: np.finfo(np.longdouble)   # ask for information about long doubles
Out[3]: finfo(resolution=1e-18, min=-1.18973149536e+4932,
max=1.18973149536e+4932, dtype=float128)
```

*Logical operations*

It is easy to perform logical test of the values of variables and constants. Note the use of the double equal sign.

```
In [1]: 1==2               # values are equal
Out[1]: False
In [2]: 2==2
Out[2]: True               # note that True and False begin with upper case
In [3]: 1<2                # first less than second
Out[3]: True
In [4]: 2<=2               # first less than or equal to second
Out[4]: True
In [5]: 1>=2               # first greater than or equal to second
Out[5]: False
In [6]: 6!=9               # first is not equal to second
Out[6]: True
In [7]: 6!=9 and 6==9      # logical AND
Out[7]: False
In [8]: 6!=9 or 6==9       # logical OR
Out[8]: True
```

To execute a block of instructions only when a particular condition is true, indent the block of instructions following an "if" statement. Note that the if statement must end with a colon.

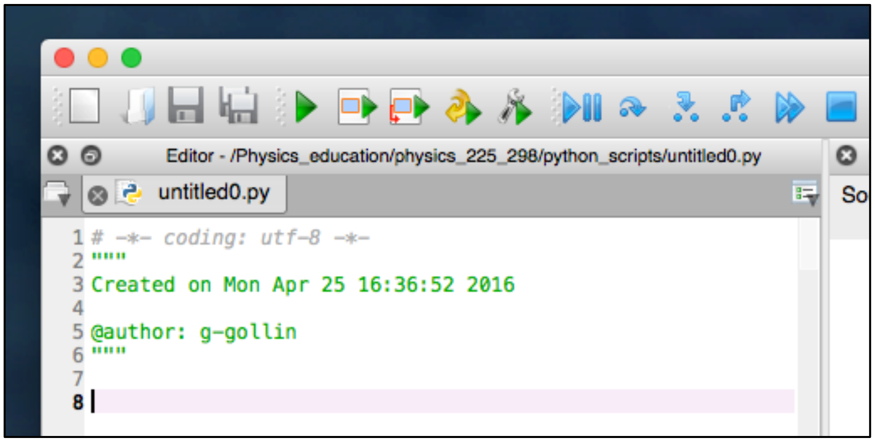
```
In[1]: LogicalValue = 4
In[2]: if LogicalValue < 5:
  ...:     print("LogicalValue is less than 5")
  ...:
LogicalValue is less than 5
```

It is very clumsy to execute if-blocks this way! A better way is to put a string of executable instructions into a script file, then execute the script.

*Scripts*

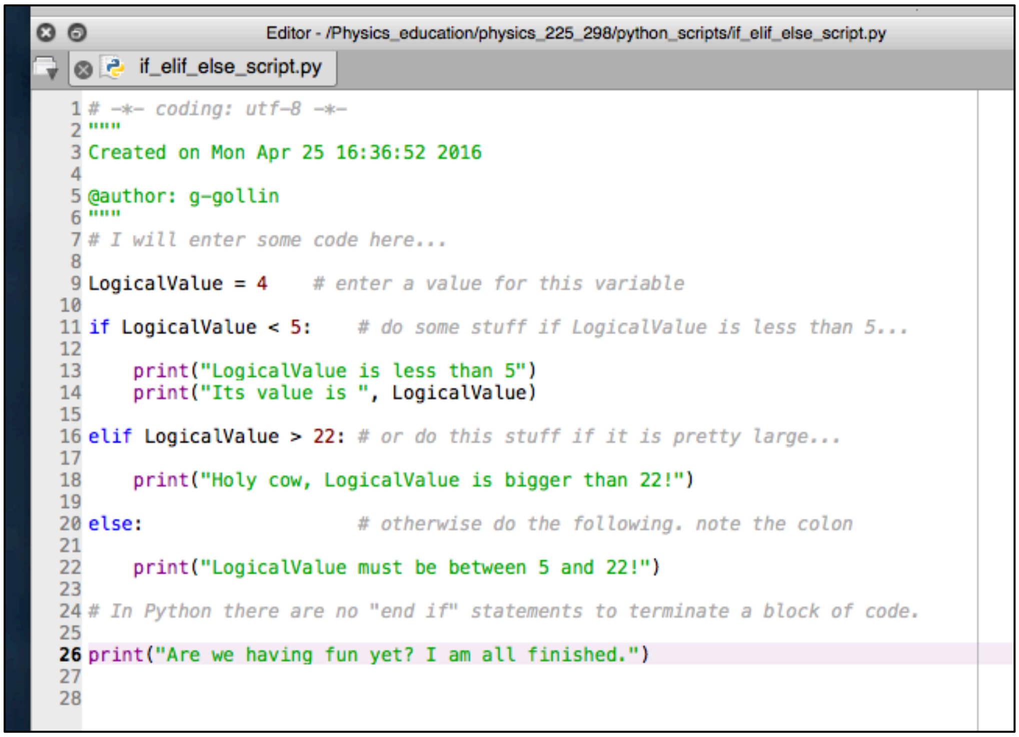
To work with scripts, you will first need to tell spyder where to find them. Begin by creating a folder in which you will store your scripts. (I've named mine "python_scripts.") Go to the "Global working directory" window in spyder's preferences to set the startup directory.
The editor opens with an untitled default script that begins with a (three-quotation mark delimited) comment.

.


Editor - /Physics_education/physics_225_298/python_scripts/untitled0.py
untitled0.py
So

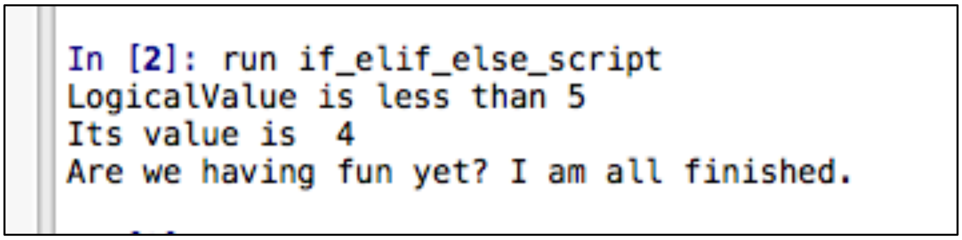
```python
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 16:36:52 2016

@author: g-gollin
"""


```

Enter some well-commented code into the editor window, then save the file. In the following screen shot I have an if-then-else-if block, followed by an example of running it from the console. The pattern of indentations is important: take careful note of it. This is how Python defines what code is inside an if block (or a loop) and what is outside. Also note the presence of the colon after the logical expression to be evaluated.


Editor - /Physics_education/physics_225_298/python_scripts/if_elif_else_script.py
if_elif_else_script.py

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 16:36:52 2016

@author: g-gollin
"""
# I will enter some code here...

LogicalValue = 4     # enter a value for this variable

if LogicalValue < 5:    # do some stuff if LogicalValue is less than 5...

    print("LogicalValue is less than 5")
    print("Its value is ", LogicalValue)

elif LogicalValue > 22: # or do this stuff if it is pretty large...

    print("Holy cow, LogicalValue is bigger than 22!")

else:                   # otherwise do the following. note the colon

    print("LogicalValue must be between 5 and 22!")

# In Python there are no "end if" statements to terminate a block of code.

print("Are we having fun yet? I am all finished.")


```

Run the program from the IPython console by typing “run” followed by the file name (leave off the “.py” filename extension.). It is possible that you will first need to tell the console to load the file: do this by typing “import” then the filename, omitting the .py extension. It is unclear to me when you actually need to do this!



```
In [2]: run if_elif_else_script
LogicalValue is less than 5
Its value is  4
Are we having fun yet? I am all finished.
```

*Lists and arrays*

Lists and arrays are rather like subscripted variables: $a_0$, $a_1$, $a_2$, … But there is a fundamental difference between the two: Python, before the import of a library like numpy, only knows about lists. A list can comprise elements of different types; if you try to "add" two lists you'll produce a concatenation of the two lists, rather than an element-by-element sum. For example,

```
In[1]: a = [1, 2, "cat"]
In[2]: b = [3, 4, "dog"]
In[3]: print(a + b)
[1, 2, 'cat', 3, 4, 'dog']
In[4]: type(a)
Out[4]: list
```

Note the use of the "type" function to ask Python what type of object is the variable **a**.
Here's another way to define a list with 8 elements, all of which are set to 3.

```
In [1]: a=[3]*8; a
Out[1]: [3, 3, 3, 3, 3, 3, 3, 3]
```

Recall that the first list element has index value 0, not 1. For example,

```
In [1]: a=[1, 2, 3, 8]
In [2]: print(a[0],a[3]) # print the first and last
1 8
```

You will certainly do more with arrays than with lists. Numpy can create them and do various operations on them. Copy/paste this script into a file and run it:

```
####################################################################

# This file is unit01_ArrayOperations.py. It contains a few examples
# of operations on lists and arrays

# George Gollin, University of Illinois, May 20, 2016

####################################################################

# use numpy to create arrays, which can be used for arithmetic operations.
```

```
aa = np.array([2, 3, 5])
bb = np.array([7, 9, 11])

# do an element-by-element sum:
print("aa = ", aa)
print("bb = ", bb)
print("aa + bb = ", aa + bb)

# calculate an element-by-element product:
print("aa * bb = ", aa * bb)

# add a scalar to every element of an array. Note the "newline" \n.
print("\naa + 100 = ", aa + 100)

# multiply every element of an array by a scalar
print("\naa * 6 = ", aa * 6)

# take the sqrt of every element of an array
print("\nnp.sqrt(aa) = ", np.sqrt(aa))

# take the square of every element of an array
print("\naa**2 = ", aa**2)

# take the sine of every element of an array
cc = np.array([0., np.pi/6, np.pi/4, np.pi/2])
print("\ncc (radians) = ", cc)
print("np.sin(cc) = ", np.sin(cc))

# convert radians to degrees
print("\nnp.degrees(cc) = ", np.degrees(cc))

# sum the elements in an array
print("\nnp.sum(aa) = ", np.sum(aa))

##################################################################
```

A very common mistake—I trip over this all the time—is to create a list instead of an array, then try to use it in a mathematical expression. I would suggest that you ALWAYS use numpy to make arrays: do this

```
cc = np.array([0., 3.2, 9., np.pi/6])
```

instead of this:

```
cc = [0., 3.2, 9., np.pi/6].
```

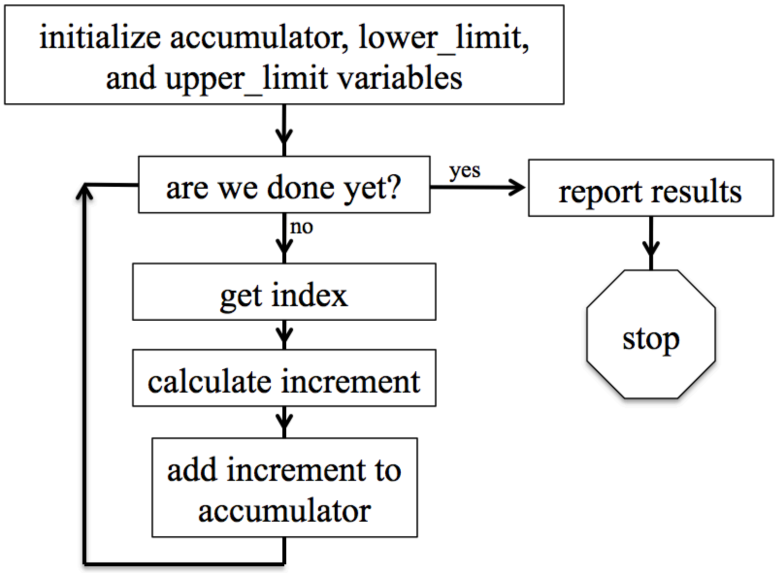
Note the placement of brackets and parentheses. What's happening here is that the np.array takes a Python list as input and produces a numpy array as output.

*Loops*

Most of your programs will include one or more loops. A loop is just what you'd expect it to be: a procedure that you execute many times, updating some of the variables each time you execute the loop.

When you write code you will want to be very clear about exactly what each line of your program is meant to accomplish. Unless you are already an experienced coder, you should consider drawing a diagram that illustrates what you think your software is going to do before you type a single line of code. Once you are clear about this you can begin writing code. I'll include flowcharts for some of the in-class exercises during the first several units to help you get the hang of this.

Here is a flow diagram for a typical loop. Note the names I've given to some variables: "accumulator," "lower_limit," "upper_limit," "increment," and "index."


initialize accumulator, lower_limit, and upper_limit variables
are we done yet?
yes
report results
no
get index
stop
calculate increment
add increment to accumulator

*A loop to calculate the sum of a few squares*

Here's the text of it; pay careful attention to the variable names in the loop. A common mistake new programmers make is to confuse the increment and accumulator variables.

```
"""
# This file is unit01_loop_structure.py. It contains a sample loop that
# calculates the sum of the squares of the numbers 1 through 10.

# George Gollin, University of Illinois, January 15, 2017
"""

# initialize variables here. take note of the names.

# the "accumulator variable" is where we sum the effects of whatever we
# calculated during successive passes through the loop. we initialize it to
# zero.  I am using the decimal point to make it a floating point variable,
# which isn't really necessary.

accumulator = 0.0

# the "increment variable" is something we'll generally need to calculate each
# pass through the loop. after calculating it we will add it to the accumulator
# variable. Since it will vary each time we go through the loop we don't need
# to initialize it here.

# specify the lower and upper limits for the loop now. Use the range function,
# which takes two integers as arguments, and creates a sequence of unity-spaced
# numbers. Note that the upper limit is not included in the sequence:
# range(1,5) gives the numbers 1, 2, 3, 4. Note that I will add 1 to the upper
# limit in my range function since range will stop short of this by 1.

lower_limit = 1
upper_limit = 10

# here's the loop. note the "whitespace" that is required, as well as the end-
# of line colon.

for index in range(lower_limit, upper_limit + 1):

    # in python we square things using a double asterisk followed by the
    # desired power. Note that a carat will not work: 3^2 is NOT 9.
    increment = index ** 2

    # now add into the accumulator.
    accumulator = accumulator + increment

    # I could have written all of this much more compactly using the +=
    # operator, but that'd be confusing, and you might find that it makes for
    # buggy, unclear code.

# we end the loop by having a line of unindented code.

print("all done! sum of squares is ", accumulator)

###################################################################

"""
Note that I could have written the code more compactly in a single line, but it
would have been harder to decipher:

   >> print(sum(np.array(range(1,11))**2))
      385
"""
```

*Other loop matters*

There is at least one other way to execute loops in Python, using “while” statements. For example, in the above code replace

```
for index in range(lower_limit, upper_limit + 1):
    increment = index ** 2
    accumulator = accumulator + increment
```

with

```
index = lower_limit
while index <= upper_limit:
    increment = index ** 2
    accumulator = accumulator + increment
    index = index + 1
```

It is possible to exit early from a loop by using the “break” command. Inserting the (properly indented) line

```
       if index > 5: break
```

into the loop will prematurely terminate it.

*Functions and modules*

As your programs get longer and more complicated, it might become convenient to break them up into multiple files, each containing one or more functions which are referenced by the main program, and/or by each other.
Here is an example, in which I have placed the functions SampleFunction1 and SampleFunction2 inside the file SampleFunctions.py.

```
###################################################################

# This file is SampleFunctions.py. It contains a few sample functions
# written in Python, included for pedagogical purposes.

# George Gollin, University of Illinois, April 29, 2016

###################################################################

def SampleFunction1(x, y, z):

    """
    This function returns (x * y) + z.

    Created on Thu Apr 28 16:34:11 2016

    Note that the multi-line string literal (all the stuff between the triple
quotes)
    serves as a "docstring": it is printed in response to a help query about
this
    function.

    Use SampleFunction1 this way:
```

```
    import SampleFunctions                                # load the module
    help(SampleFunctions.SampleFunction1)                 # ask for help
    TheAnswer = SampleFunctions.SampleFunction1(3,4,5)    # call the function

    author: g-gollin
    """

    WorkingVariable = x * y
    WorkingVariable = WorkingVariable + z
    return WorkingVariable

    # end of SampleFunction1

###################################################################

# Now define a second function

###################################################################

def SampleFunction2(x, y):

    """

    This function returns sqrt(x^2 + y^2). Use this way after importing the
    module:
    print(SampleFunctions.SampleFunction2(3,4))

    """
    # sum the squares of the two arguments
    WorkingVariable = x**2 + y**2

    # now take the square root.
    WorkingVariable = WorkingVariable ** 0.5

    # all done.
    return WorkingVariable

    # end of SampleFunction2

###################################################################
```

For the sake of clarity I have made no attempt to write efficient code! For example, I could have shortened the executable parts of SampleFunction1 into a single line:

```
    return x * y + z
```

Things to note:

- Each function begins with a few lines of text set off by triple quotes. Python treats these as a "docstring" and will spit them out in response to a help query about the function.

- There are a lot of explanatory comments. You should not be parsimonious in your inclusion of comments in your own programs!
- You refer to the functions inside a "module" using notation that is very common in object oriented languages: <module name>.<function name>. The module name is just the name of the file, with the ".py" filename extension omitted. For example,

```
Hypotenuse = SampleFunctions.SampleFunction2(5,12)
```

***An exercise: an infinite series for π***

Recall that we can generally find infinite series representations of transcendental functions like sin($x$). In particular,

$$\tan^{-1}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \qquad -1 < x \le 1.$$

Since tan$^{-1}$(1) = $\pi$/4, we can write the following (slowly converging) infinite series□

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots.$$

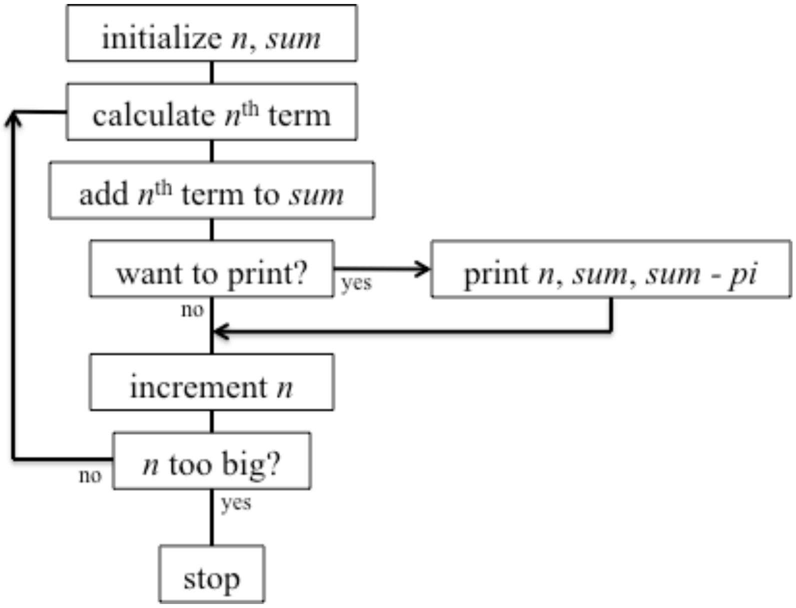
If we group adjacent terms in the series we can rewrite this as

$$\begin{aligned} \frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} &= \left(1 - \frac{1}{3}\right) + \left(\frac{1}{5} - \frac{1}{7}\right) + \left(\frac{1}{9} - \frac{1}{11}\right) + \cdots \\ &= \frac{3-1}{3 \cdot 1} + \frac{7-5}{7 \cdot 5} + \frac{11-9}{11 \cdot 9} + \cdots \\ &= \frac{2}{3} + \frac{2}{35} + \frac{2}{99} + \cdots \\ &= 2 \cdot \sum_{n=0}^{\infty} \left[ \frac{1}{(4n+3)(4n+1)} \right]. \end{aligned}$$

The value of $\pi$□□□ 3.14159265358979323846264338327950288419716939937510582…, though the precision with which your computer can calculate it is probably limited to fewer digits than this.

Please write a Python script that calculates an approximation to $\pi$ using the arctan series, and compare its accuracy after the $n$ = 10 term, 100 term, 10,000 term, and 1,000,000 term. (Use a conditional statement to print something after the appropriate terms.)

You should approach this by initializing a few things, then executing a loop that calculates the $n^{\text{th}}$ term, with $n$ running from 0 to 999,999, summing the terms as you go. Here's a flowchart for one way to structure your program…



…and here's a listing of a template you could start with.

```
"""
Goal/purpose: This file is a template. You will build
your arctan(1) series (as well as subsequent in-class and homework assignments)
from it.
The code here actually calculates the sum of the square roots of the integers
0, 1, 2, 3, 4.

Assignment: unit 1 in-class machine exercise 2

Author(s): Monica and George

Collaborators: Monica, George, and Neal (CS professor)

Date: January 18, 2017


Reference(s):
Stack overflow web site (see
http://stackoverflow.com/documentation/python/193/getting-started-with-python-
language#t=201701181706539874984)
Physics 246 course notes

"""

###################################
# Import libraries
###################################
```

```
import numpy as np

###################################
# Define and initialize variables
###################################

# Accumulator variable
accumulator = 0

# Index and upper limit variables for the loop
lower_limit = 0
upper_limit = 4

###############################################################################
# Loop to sum the square roots of a bunch of integers
###############################################################################

for index in range(lower_limit, upper_limit + 1):

    # calculate increment, then add it to accumulator.
    increment = np.sqrt(index)
    accumulator = accumulator + increment

    # I could have just added np.sqrt(index) to accumulator, without defining
    # increment.

###############################################################################
# End of loop. Print the results.
###############################################################################

print("all done. Sum of square roots is ", accumulator)

###############################################################################
#
```

## *Libraries*

### *Numpy*

Numpy is one of the libraries that is included with the Anaconda Python release. Wikipedia describes it this way:[1] “NumPy… is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.”
You’ll need to make Python aware of its existence by importing it:

```
In [1]: import numpy
```

You can see what lives inside numpy by issuing this command:

```
In [2]: dir(numpy)
```

[1] https://en.wikipedia.org/wiki/NumPy

```
Out[2]:
['ALLOW_THREADS',
'BUFSIZE',
'CLIP',
'ComplexWarning',
…
```

There's quite a lot there, including a sqrt routine. After importing numpy you can call its routines like this:

```
In [3]: numpy.sqrt(2)
Out[3]: 1.4142135623730951
```

If you would prefer to use a shorter name for numpy (perhaps to save some typing), you could have imported it this way:

```
In [4]: import numpy as np
In [5]: np.sqrt(2)
Out[5]: 1.4142135623730951
```

Numpy also knows the value for $\pi$:

```
In [6]: np.pi
Out[6]: 3.141592653589793
```

### *Matplotlib*

Matplotlib "is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits… There is also a procedural "pylab" interface… designed to closely resemble that of MATLAB."[2]

You will want to import both matplotlib and pyplot. Do the following:

```
In [6]: import matplotlib
In [7]: import matplotlib.pyplot as plt
```

You will probably want to include the import statements into scripts/programs you write so that you aren't required to import them from the iPython console each time. (But there's nothing wrong with importing something multiple times.)

There is good documentation (including examples) here: http://matplotlib.org/.

[2] https://en.wikipedia.org/wiki/Matplotlib

### ***Drawing curves in two dimensions***

*How to draw a circle*

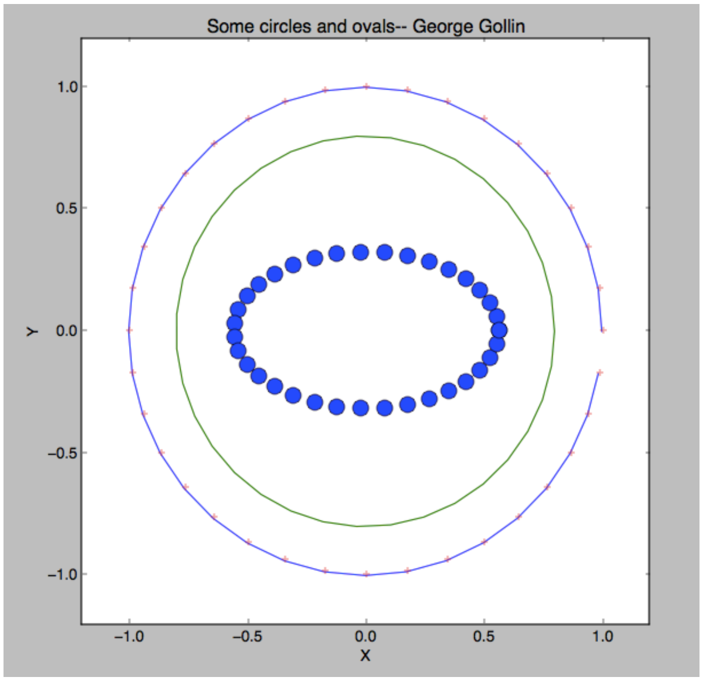
Let's load an array with a reasonably large number of *x*, *y* points that lie on a circle, then plot them and save the plot to a file. Here's a script that does this, making a number of figures in the same window.

Let's talk through what's in the file. When you make other kinds of plots, consider copying what's in this script into your own, then changing a few things to make it do what you want, rather than writing something from scratch. That'll save you time, and also the effort of understanding the minutiae of the graphics code.

```
# load arrays with coordinates of points on a unit circle, then plot them
# this file is unit02_draw_circles.py

# import the numpy and matplotlib.pyplot libraries
import numpy as np
import matplotlib.pyplot as plt

# create the arrays. first is angle, running from [0,2pi). Note the use of
# endpoint = False in linspace to make this interval open on the right. The
# first two arguments in linspace are the beginning and end of the interval
# of uniformly spaced points. The third is the total number of points.
ThetaArray = np.linspace(0, 2*np.pi, 36, endpoint=False)

# cos and sin in numpy can act on all elements in an array. Note that the
# output is also an array.
x = np.cos(ThetaArray)
y = np.sin(ThetaArray)

# set the size for figures so that they are square. (Units: inches???)
plt.figure(figsize=(8.0, 8.0))

# also set the x and y axis limits
plt.xlim(-1.2, 1.2)
plt.ylim(-1.2, 1.2)

# plot the x,y points, connecting successive points with lines
plt.plot(x,y)

# now plot the points again, on the same axes, but using red + signs:
plt.plot(x,y, 'r+')

# now redo the array of angles (and x,y points) to include the 2pi endpoint.
# make a smaller circle this time...
ThetaArray = np.linspace(0, 2*np.pi, 36, endpoint=True)
x = 0.8*np.cos(ThetaArray)
y = 0.8*np.sin(ThetaArray)

#plot it. note how the line color changes...
plt.plot(x,y)
```

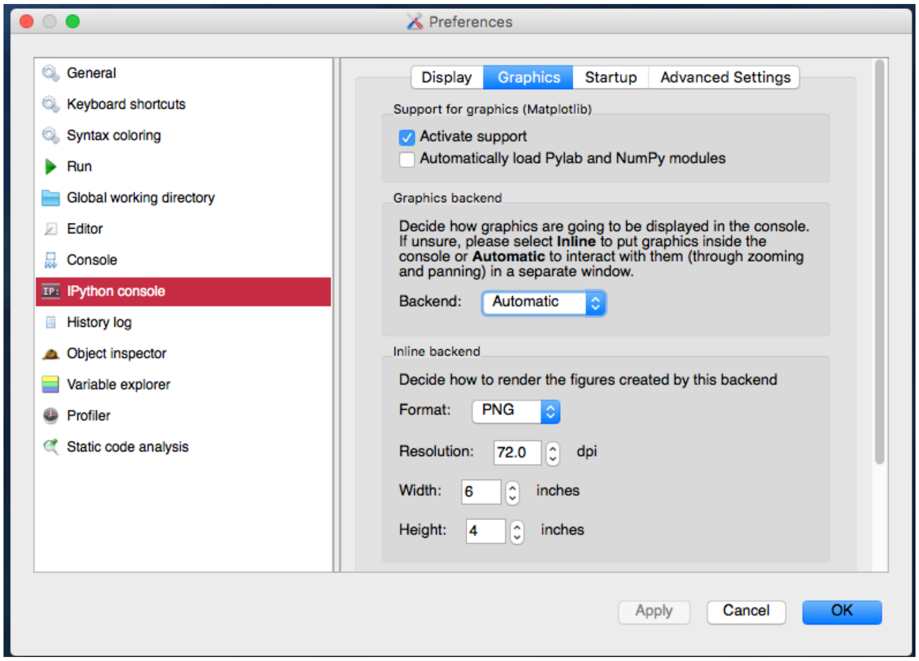
```
# now make an oval by down-scaling the x, y arrays, then plot
# it using rather large, filled blue circles.
x = 0.7 * x
y = 0.4 * y
plt.plot(x,y, 'bo', markersize=12)

# now put a title onto the plot, then label the axes
plt.title("Some circles and ovals-- George Gollin")
plt.xlabel("X")
plt.ylabel("Y")

# now save plot to a png (portable network graphics) file
plt.savefig("CirclePlotOne.png")
```

Note that ***some*** functions (such as np.sin) will act on all the elements in the array to which it is applied. This is very convenient! Here's what we get:



*Drawing figures in new windows*

If you forgot to attend to setting Python's parameters when you installed it last week, your version of Spyder probably puts your graphs into the same iPython console that you use to enter commands. To make plots open in new windows, go to the python preferences menu (on a Mac it lives in the "Python…" menu at the top of the screen), then select "iPython console" and "Graphics." Set the "Backend" field to Automatic.



Support for graphics (Matplotlib)

☑ Activate support
□ Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with them (through zooming and panning) in a separate window.

Backend: Automatic

Inline backend

Decide how to render the figures created by this backend

Format: PNG

Resolution: 72.0 dpi

Width: 6 inches

Height: 4 inches

Once you've done this, each figure should open in a new window.

### *Another exercise: graphing the magnification of a weird optical system*

Imagine that you stumble upon a strange optical device that produces magnified images of objects placed downfield of a critical point $x_c = 10$ meters. The magnification $M$—the ratio of image height to object height—is guaranteed by the manufacturer to satisfy the equation

$$M(x) = \frac{1}{\sqrt{1 - \frac{x_c}{x}}} \quad .$$

(The manufacturer's literature warns that the device will self-destruct if it is exposed to objects closer than $x_c$.)

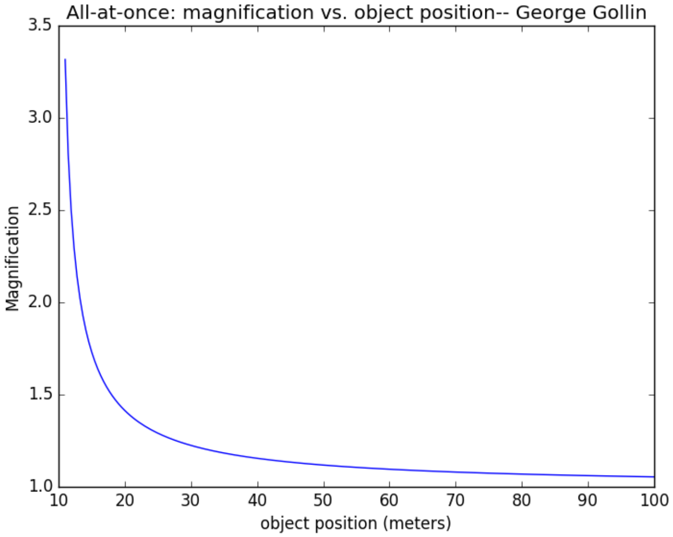
Please generate a graph of $M(x)$ vs. $x$ for the range $1.1\ x_c < x < 10\ x_c$ . Use a step size that is small enough to allow your graph to look smooth and continuous. Do it one of two ways: by coding up a loop that loads appropriate arrays or by using Python's all-at-once capabilities built into numpy for performing array calculations.

If you're not sure how to get started, make a copy of unit02_draw_circles.py and throw away what you don't need, then have it generate an array of $x$ values, and then the corresponding magnifications.

You can put the following line of code into your program before you make a plot to close all already-open graphics windows, if you want: `plt.close("all").`

Note that this strange function is going to appear in discussions of space-time curvature in General Relativity. In that context, $x_c$ is replaced with the Schwarzschild radius of a compact

massive object. And the “magnification” is instead a measure of the discrepancy between $2\pi$ and the ratio of the circumference and radius of a circle with the massive object at its center.
Your result should look something like this:



***Graphical representations of three dimensions***

If we want to draw a curve in three dimensions, we’ll need to import more libraries. You’ll want something like this in your programs:

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
```

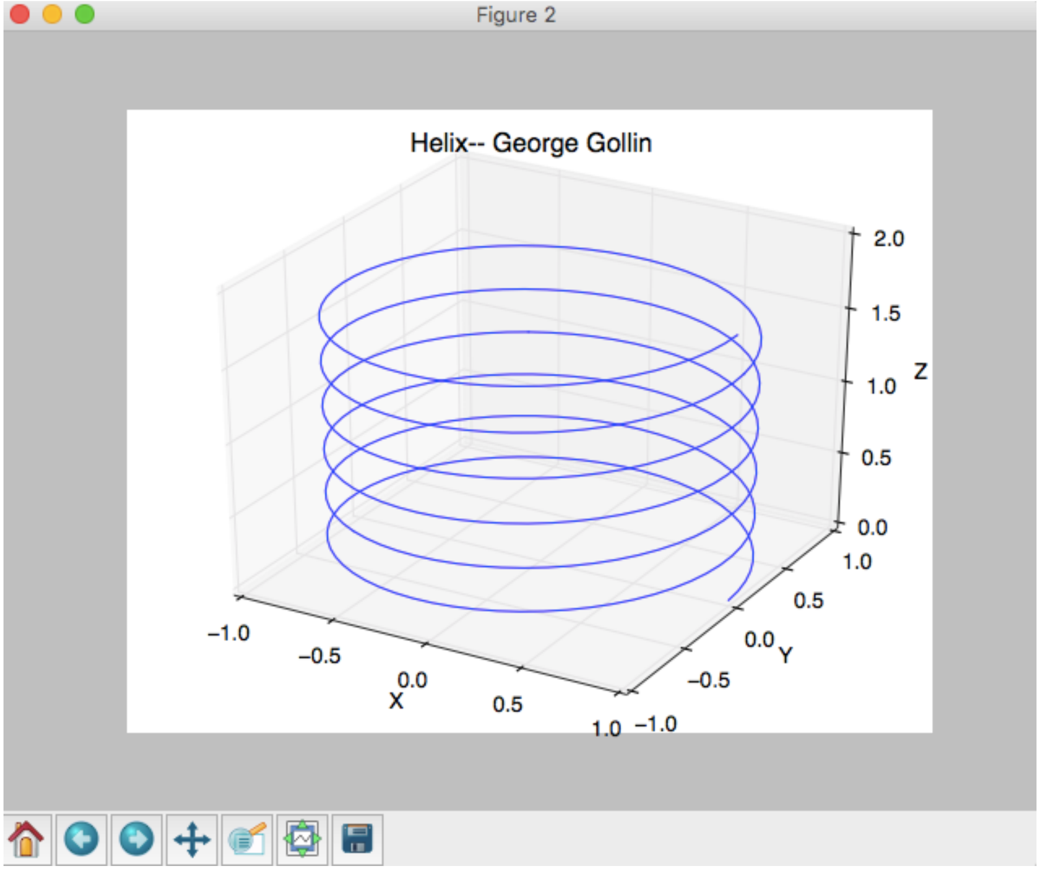
*Drawing a helix*

Let’s say we want to draw three turns of a helix whose projection on the *x-y* plane is a circle of unit radius, and that advances along the positive *z* axis by 0.3 meters per turn. I will assume the helix begins at (*x, y, z*) = (1, 0, 0), and that it winds in a counterclockwise direction when seen from above.
Here is a script that generates the drawing.

```
# load arrays with coordinates of points on a helix, then plot them
# this file is unit02_draw_helix.py

# import libraries. Python may complain about the first one, but you really do
# need it.
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
```

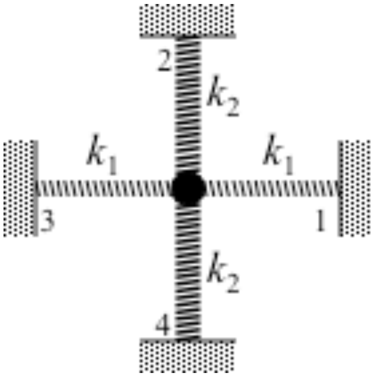
```
# total number of turns the helix will make
NumberTurns = 6

# pitch (meters per turn)
pitch = 0.3

# radius
Radius = 1.0

# points to plot per turn
PointsPerTurn = 60

# create the array of angles for successive points. the arguments to linspace
# are (1) first value; (2) last value; (3) number of equally-spaced values
# to put into the array.
ThetaArray = np.linspace(0, NumberTurns*2*np.pi, NumberTurns*PointsPerTurn)

# Now get x,y,z for each point to plot.
x = np.cos(ThetaArray)
y = np.sin(ThetaArray)
z = np.linspace(0, NumberTurns*pitch, NumberTurns*PointsPerTurn)

# now create a (blank) figure so we can set some of its attributes.
fig = plt.figure()

# "gca" is "get current axes." set the projection attribute to 3D.
ax = fig.gca(projection='3d')

# set the x, y, and z axis limits of the plot axes
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
ax.set_zlim(0, 2)

# label the axes and give the plot a title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.set_title("Helix-- George Gollin")

# now plot the helix.
ax.plot(x, y, z)

# now save the plot to a png (portable network graphics) file
plt.savefig("HelixPlot.png")
```

The result follows. Take note of the pan/scroll/rotate button (the cross made of double-headed arrows): it allows you to rotate a 3D figure to view it from different angles. Try this out.



*How to draw a surface whose height above the x-y plane depends on a function*

Imagine that a mass is constrained to move in the $x$-$y$ plane, and is held in place by four springs of length $L$ as shown in the following illustration. Springs 1 and 3 have identical spring constants $k_1$, while springs 2 and 4 have spring constants $k_2$, with $k_2 > k_1$.



If the mass is displaced from equilibrium to the point $(x, y)$ (with displacement that is very small compared to the springs' lengths $L$), the system's potential energy will increase by

$$\Delta U = k_1 x^2 + k_2 y^2 .$$

How might we draw the surface that represents $U(x, y)$?

I'll assume that $U(0, 0) = 0$, $k_1 = 2$, $k_2 = 5$, and that our graph is to span the range $-3 < x < 3$, $-3 < y < 3$, with a grid employing cell size 0.1 × 0.1. This means that our graph will show the height above the $x$, $y$ plane at 61 × 61 = 3,721 points.

We could do something like this to begin defining the grid.

```
x = np.linspace(-3, 3, 61, endpoint=True)
y = np.linspace(-3, 3, 61, endpoint=True)
```

This will give us a pair of arrays, each of 61 elements, with successive entries spaced by 0.1. But that's not really what we want: we need a list of the $x$, $y$ coordinates of all 3,721 points on our grid. We can do this using the **`numpy`** function **`meshgrid`** after defining the **`x`** and **`y`** arrays (as I did a few lines ago):

```
xgrid, ygrid = np.meshgrid(x,y)
```

I appreciate that this is confusing at first. Let's consider smaller arrays so I can print them out for you. Here's what I get, after importing **`numpy as np`**:

```
# make 3-element arrays which give the x values of "columns"
# and y values of "rows."
In [1]: x = np.linspace(-1, 1, 3, endpoint=True)
In [2]: y = np.linspace(-1, 1, 3, endpoint=True)

# now list the values for the x and y arrays.
In [3]: x
Out[3]: array([-1.,  0.,  1.])
In [4]: y
Out[4]: array([-1.,  0.,  1.])

# our 3 x 3 array will have nine cells, of course, so we will need to
# load one 9-element array with the x values for each cell and another
# with the y values for each cell.
In [5]: xgrid, ygrid = np.meshgrid(x, y)

# now list the x values for each of our nine cells
In [6]: xgrid
Out[6]:
array([[-1.,  0.,  1.],
       [-1.,  0.,  1.],
       [-1.,  0.,  1.]])
# now list the y values for each of our nine cells
In [7]: ygrid
Out[7]:
array([[-1., -1., -1.],
       [ 0.,  0.,  0.],
       [ 1.,  1.,  1.]])
```

Take note of the order in which the cells appear in our arrays: the first cell is at $x = -1$ and $y = -1$. The second is at $x = 0$, $y = -1$; the third at $x = +1$, $y = -1$, and so forth.

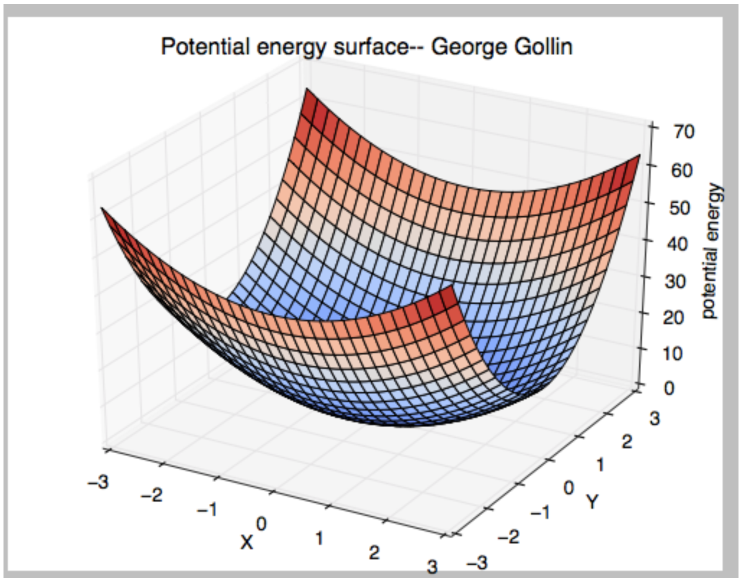
| | $x = -1$ | $x = 0$ | $x = +1$ |
|---|---|---|---|
| $y = +1$ | 7 | 8 | 9 |
| $y = 0$ | 4 | 5 | 6 |
| $y = -1$ | 1 | 2 | 3 |

Now we're ready to create another array that holds the potential at the *x,y* position of each cell. A listing of a program that actually generates a graph of *U* (along with the program's output) follows.

```
# load arrays with coordinates of points on a surface, then plot them
# this file is unit02_draw_surface.py

# import libraries
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
# let's also import a color map so we can make the picture prettier
from matplotlib import cm

# define parameters for our plot
xmin = -3
xmax = -xmin
ymin = xmin
ymax = -ymin

# number of rows and columns in our grid
nrows = 61
ncolumns = 61

# number of rows and columns per grid line to be drawn
rowsPerGrid = 2
columnsPerGrid = 2

# define the coefficients in the potential
xcoeff = 2
ycoeff = 5

# create the arrays.

# first get the x values of the "columns" in the grid.
x = np.linspace(xmin, xmax, ncolumns)

# now get the y values of the "rows" in the grid.
y = np.linspace(ymin, ymax, nrows)

# now generate the x and y coordinates of all nrows * ncolumns points
```

```
xgrid, ygrid = np.meshgrid(x,y)

# now generate the potential for all points on our grid.
zsurface = xcoeff * xgrid**2 + ycoeff * ygrid**2

# now create a (blank) figure so we can set some of its attributes.
fig = plt.figure()

# "gca" is "get current axes." set the projection attribute to 3D.
ax = fig.gca(projection='3d')

# set labels and title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("potential energy")
ax.set_title("Potential energy surface-- George Gollin")

# now put the graph into the blank figure. Note the line-continuation character.
# colormap (cmap) argument is optional.
surf = ax.plot_surface(xgrid, ygrid, zsurface, rstride=rowsPerGrid, \
cstride=columnsPerGrid, cmap=cm.coolwarm )

# now save the plot to a png (portable network graphics) file
plt.savefig("SurfacePlotCartesian.png")
```

The “rstride” and “cstride” arguments in the ax.plot_surface function tell the graphics routines how many rows and columns in the *x*-*y* mesh to use as the spacing between grid lines drawn on the surface. You can replace the color map specification `cmap=cm.coolwarm` with an RGB hexadecimal code for the color to be used on the surface, for example `color="#FF0000"` to use red.



*Drawing a surface using polar coordinates*

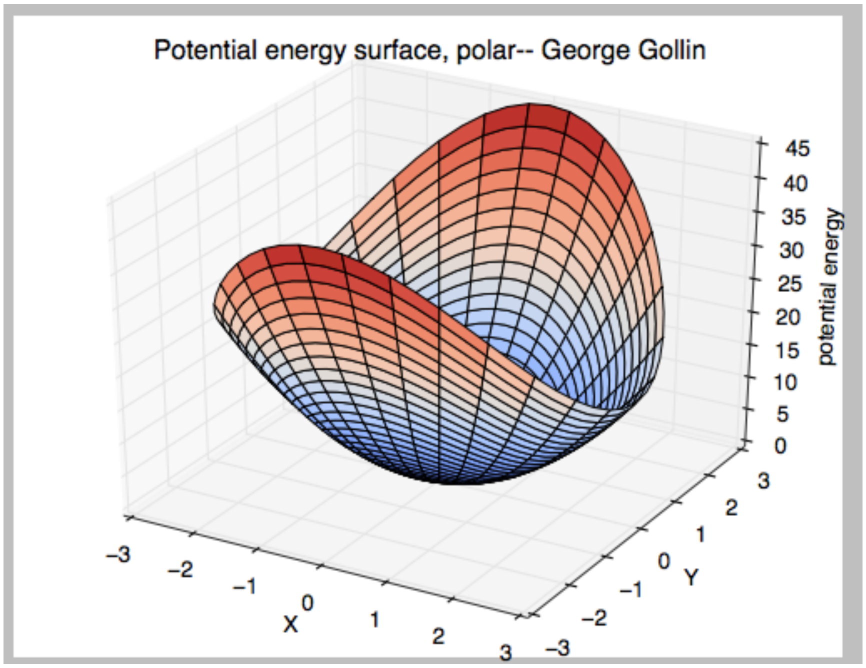
It is easy to draw a surface using cylindrical, instead of Cartesian coordinates. To do this in the previous example, replace the lines

```
x = np.linspace(ymin, ymax, nrows)
y = np.linspace(ymin, ymax, nrows)
xgrid, ygrid = np.meshgrid(x,y)
```

with something like this (after setting rmax and thetamax):

```
r = np.linspace(0, rmax, 61)
theta = np.linspace(0, thetamax, 61)
rgrid, thetagrid = np.meshgrid(r, theta)
xgrid, ygrid = rgrid*np.cos(thetagrid), rgrid*np.sin(thetagrid)
```

The result is shown below.



***A final exercise: graphing a surface you'll see in General Relativity***

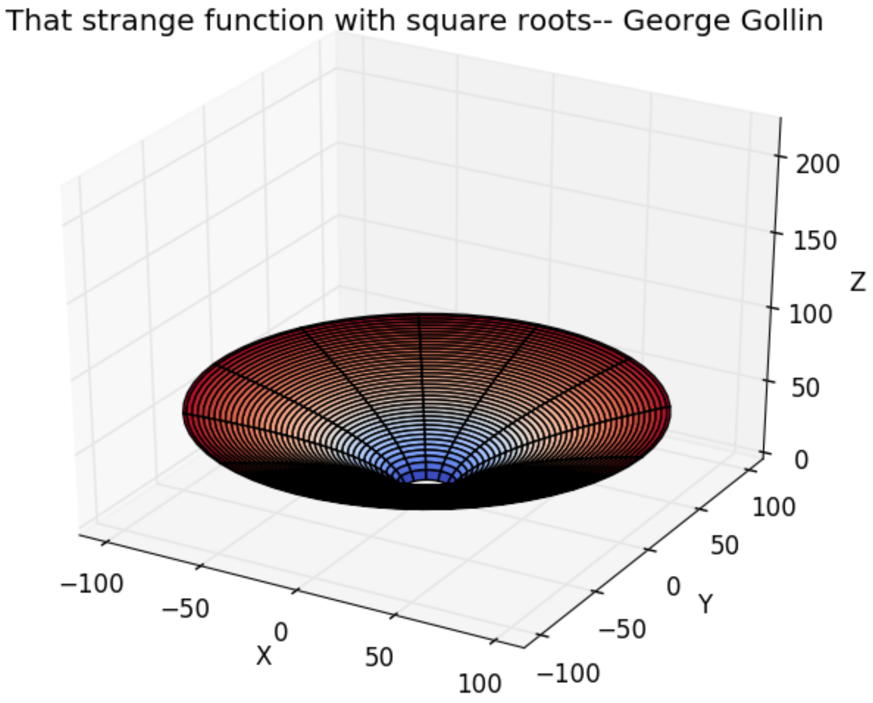
Here's another function you'll see when discussing curved spacetime, when you might use an approximation to it to generate what is called an embedding diagram. Consider the following function $z(r,\theta)$, where $r$, $\theta$ are the familiar polar coordinates. (Note that $z$ is actually independent of $\theta$.)

$$z(r,\theta) = 2r_S\left\{\sqrt{\frac{r}{r_S}-1}-\sqrt{\frac{r_0}{r_S}-1}\right\}.$$

Assume that the scale parameter $r_S$ has the value $r_S = 10$ meters and that the constant $r_0$ has the value 11 meters.

Please plot the surface defined by $z$ for $r_0 < r < 10r_0$ and $0 < \theta < 2\pi$. Your plot ought to look something like mine, below. Note that I've forced the aspect ratio to be 1:1:1 by doing this:

```
        ax.set_xlim(-rmax, rmax)
        ax.set_ylim(-rmax, rmax)
        ax.set_zlim(0, 2*rmax)
```


That strange function with square roots-- George Gollin
X
Y
Z