Physics 398DLP
Design Like a Physicist
Fall 2019

George Gollin
University of Illinois at Urbana-Champaign

Physics 398DLP
George Gollin
University of Illinois at Urbana-Champaign
Fall 2019
Fridays, 1 pm – 5 pm; 3 credit hours.

Table of Contents

Introduction and Syllabus   4

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

Physics 398DLP
Design Like a Physicist
Fall 2019

Introduction and Syllabus

George Gollin
University of Illinois at Urbana-Champaign

Physics 398DLP
George Gollin
University of Illinois at Urbana-Champaign
Fall 2019
Fridays, 1 pm – 5 pm; 3 credit hours.

Introduction and Syllabus

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—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.

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 398DLP 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 398DLP 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 398DLP 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—at least in this first year of the course—we will give you what you need to build the PCB version, and let you keep most 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 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!

Here are 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.

- Noise mitigation in public spaces (e.g. Farren's restaurant)
- Solar farm power conditioner noise mitigation
- Greenhouse gas production by livestock
- The acoustic properties of the Krannert Center's Caldwell Playhouse as a function of source (and audience) position
- Mitigation of heat leakage through exterior walls by installation of spot insulation
- Noise and pressure profiles in the vicinity of wind turbines
- Psychoacoustics: pitch recall and reproduction
- Characterizing vibrato in stringed instruments and the human voice
- Airborne particulate concentrations in home and institutional kitchens (I will teach you to make fresh ramen noodles!)
- Measurement of a runner's tendencies towards pronation or supination
- Gel pack vest cooling of hazmat-wearing workers
- Precisely cross-timed distributed measurements of atmospheric conditions in an outdoor performance venue
- How harmful to the ear are campus hand driers?
- High frequency, large amplitude vibrations experienced by MTD passengers
- Inexpensive electronic veterinary stethoscope feasibility study
- Airborne particulate concentrations in agricultural settings: outside/inside tractor and/or combine cabins

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 an electronic diary in which you describe your work, useful revelations, and calculations. Put into it screen shots of useful stuff. I do not want this to be anything fancy, but the diary should be cumulative, rather than something you close off at the end of each class meeting. When your file becomes unwieldy, close it and start the next "volume." You should have your diary open while working on your project.

My preference is for you to use Microsoft Word, though if there's another word processing system you'd prefer to use, that's fine. You will be uploading a PDF version of the file to the course directory right before the beginning of each class. You should put notes about techniques you find (or invent) into your diary so you can find them later.

Weekly upload of your diary is mandatory. I will reduce your course grade if you do not upload an account of your work every week.

You may be tempted to use LaTeX for your diary. But unless you are exceptionally facile with LaTeX (and can already upload screen shots, for example), this will be a mistake. I am going to be fairly hardnosed about this: making a lovely version of your diary at the cost of ten minutes of extra time in class is unacceptable

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.
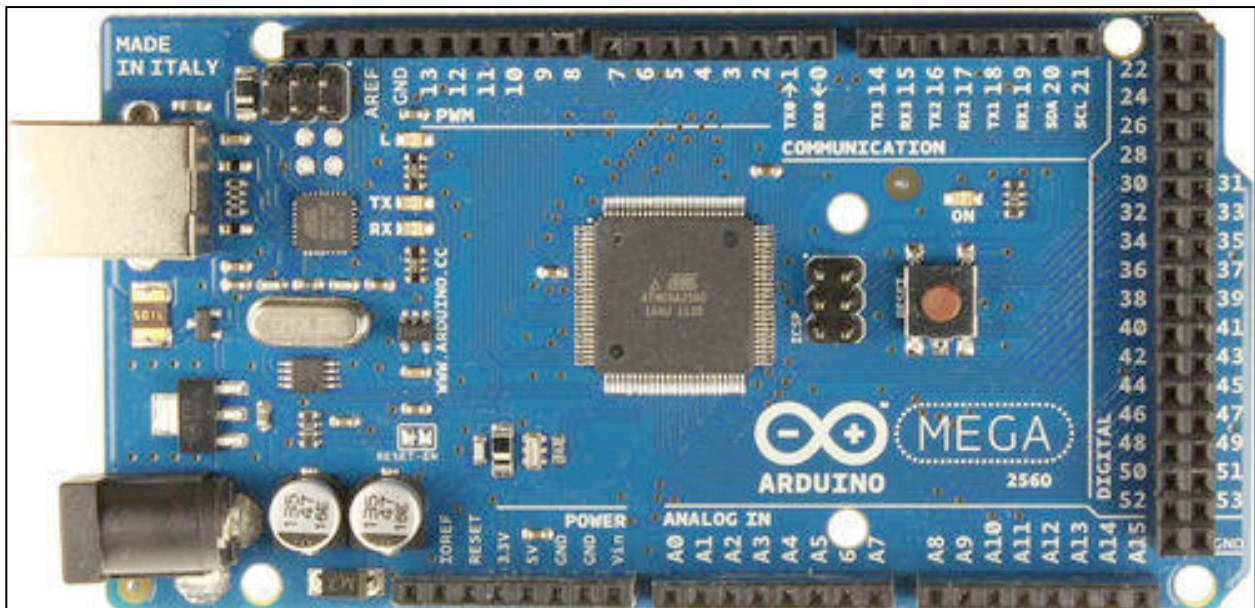
You will be working with things for which your understanding will often be a little blurry. That's OK, and in fact that's the usual state of things in research. Taking the time to understand every last detail about an IDE is a waste of your time: it is better to focus your efforts on getting by, on muddling 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; (3) a word processor window in which you are updating your diary.

The tools we will use

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

Arduino Mega 2560

It's 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.
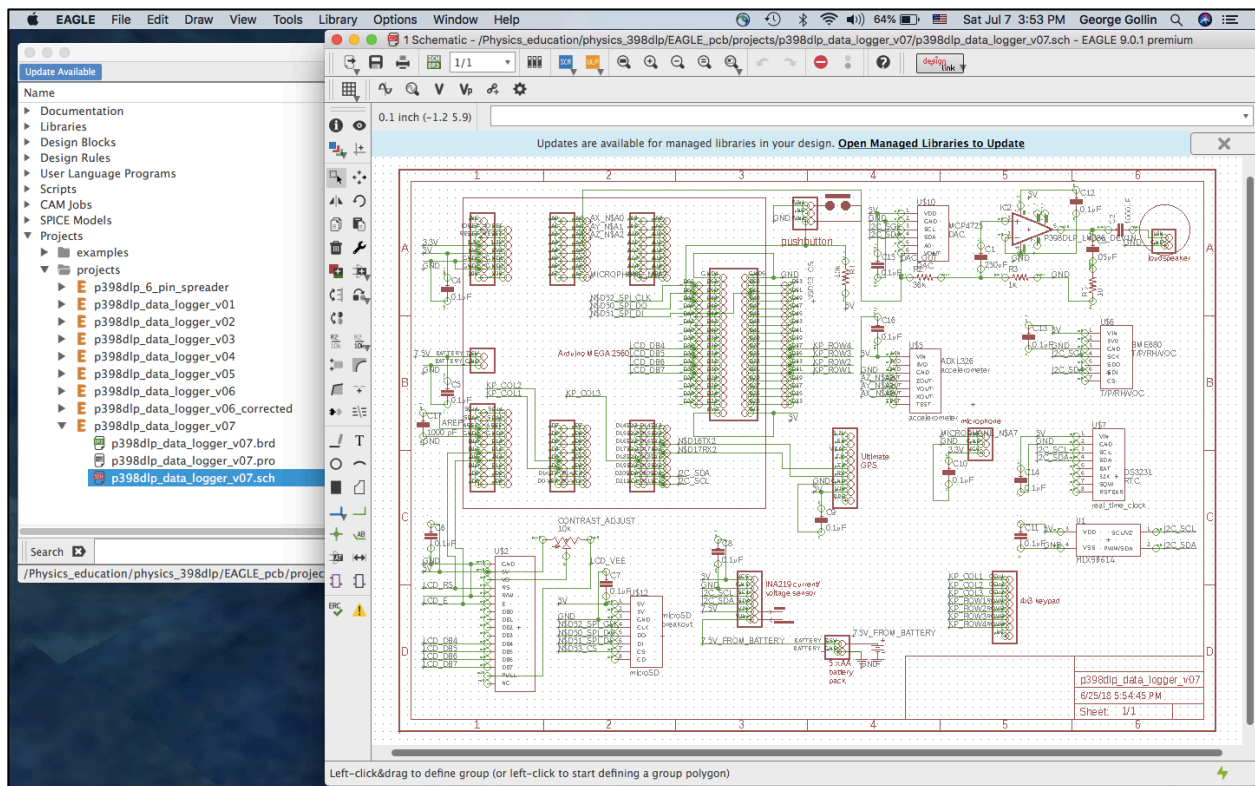

EAGLE schematic capture tool

I will lay out 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'll have 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.

Introduction and Syllabus   11

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.

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, we'll run the print jobs ourselves 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.
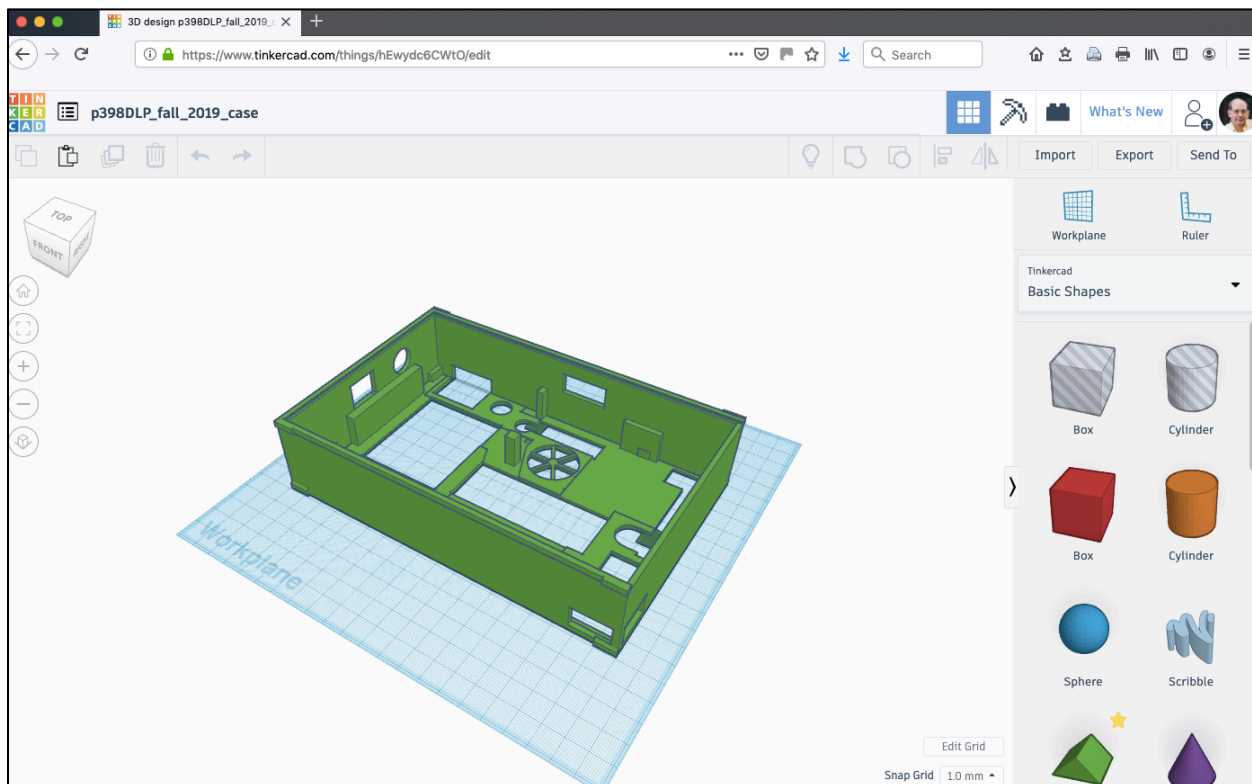
TinkerCad screen shot

Here's the Cura window. The latest version is 4.2.1; 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..

Physics 398DLP, University of Illinois   ©George Gollin, 2019

Introduction and Syllabus   12
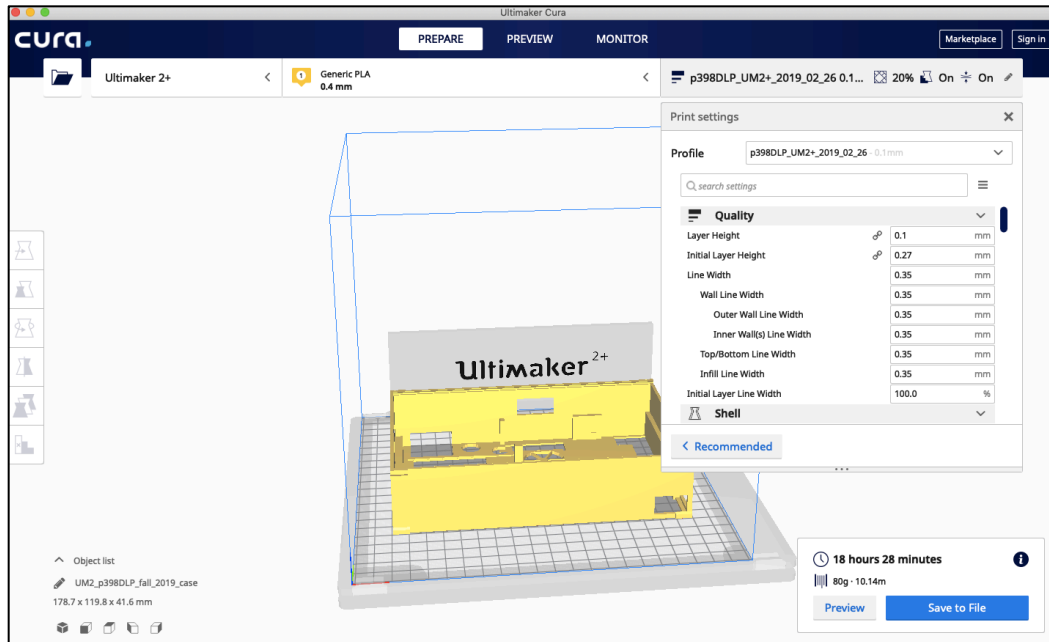
Cura screen shot

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 distortions in the image.

Ultimaker 2+ 3D printer

Introduction and Syllabus   13
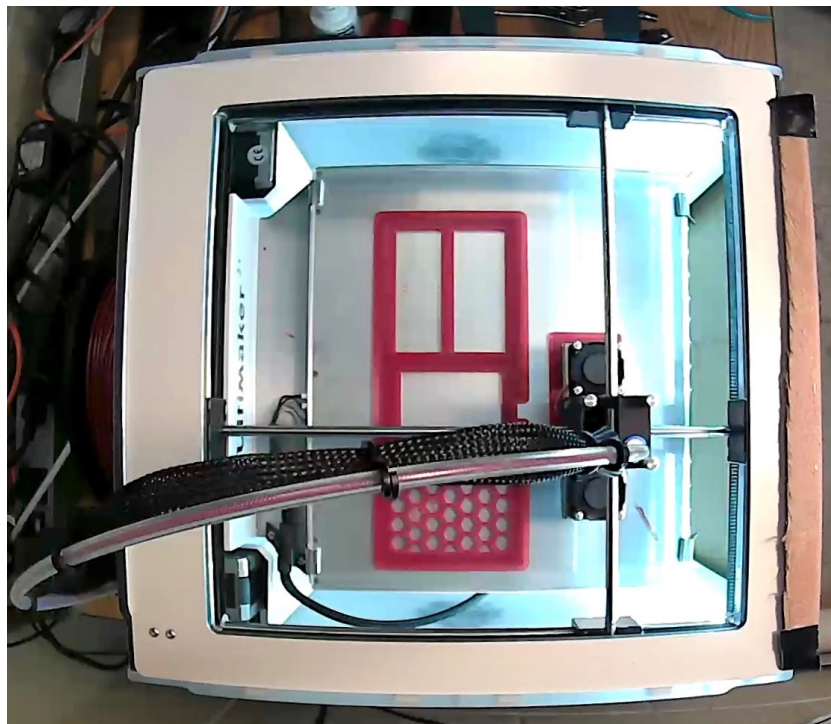
Offline data analysis

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.
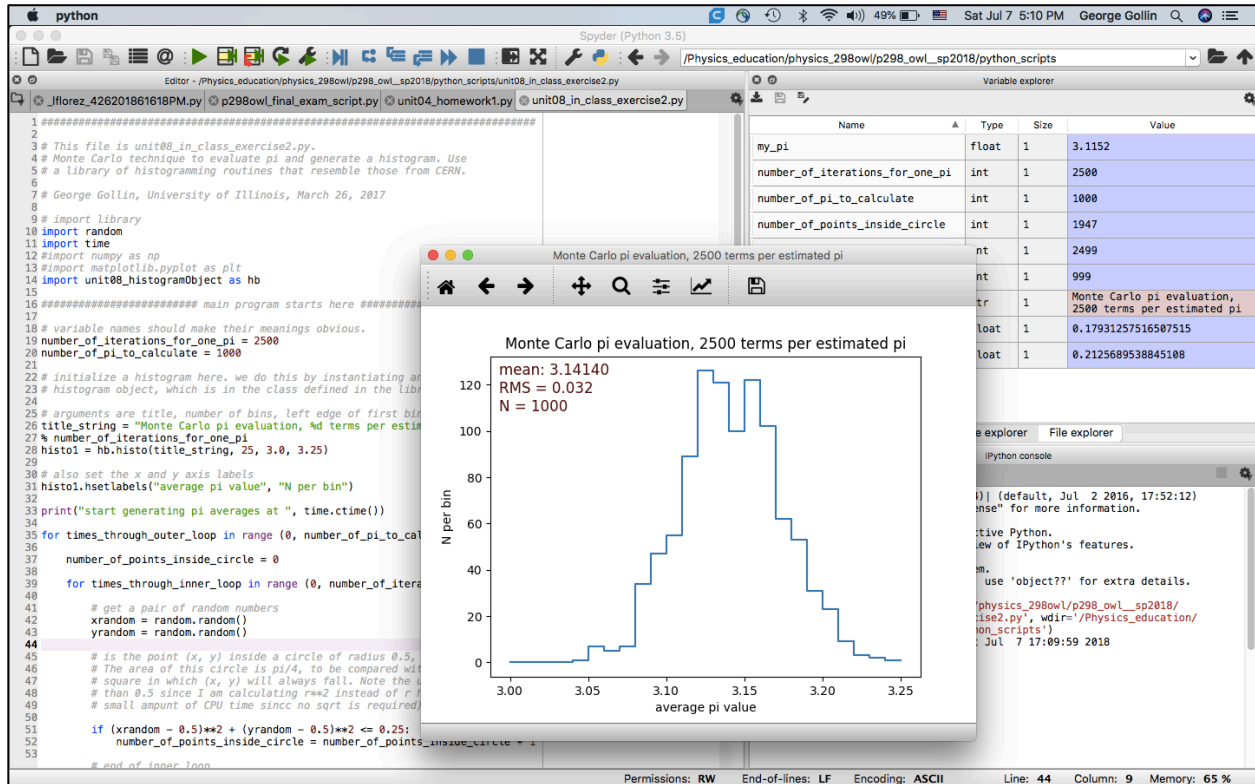
iPython IDE

Sensors and other hardware

Most of the sensors we'll use have been assembled onto small "breakout boards" by Adafruit Industries. We'll solder "pin headers" onto them and plug these 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.
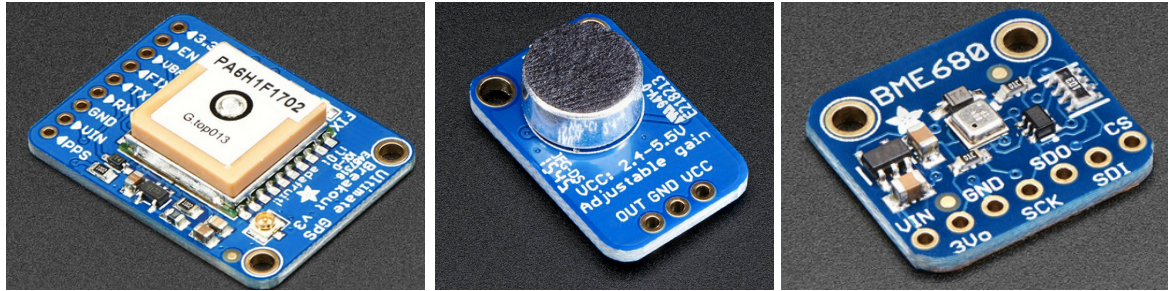
Here's a (nearly complete) list of sensors and other goodies that we have on hand. Most of them are from Adafruit. I have enough of most of them for all of your breadboard and PCB devices.

- Liquid crystal display: 16 x 2 LCD CFAH1602B-NGG-JTV
- 3 x 4 keypad (#1824)
- AA batteries and battery cases; CR1220 backup bateries
- ADXL326 accelerometer (#1018)
- Analog UV sensor (#1918)
- Arduino Mega 2560 microcontroller
- BME 680 T/RH/P/VOC (#3660)
- DRV2605L haptic vibrator driver
- DS3231 real time clock (#3013)
- electret microphone with amplifier (#1063)
- EMG electrodes
- Force-sensitive resistors
- GPS Antenna - External Active Antenna (#960)
- GUVA Analog UV sensor (#1918)
- Haptic feedback motorized vibrator
- INA219 battery voltage/current sensor (#904)
- L3GD20H Triple-Axis Gyro Breakout (#1032)
- LSM9DS1 accelerometer/magnetometer/gyroscope 0x1E (#3387)
- MCP4725 DAC, 12-bit, I2C (#935)
- MicroSD card breakout (#254)
- MicroSD memory card 8 GB SDHC
- Mini Metal Speaker w/ Wires - 8 ohm 0.5W (#1890)
- MLX90614 IR sensor (#1748)
- MMA8451 ±2 g accelerometer
- MyoWare EMG muscle sensor
- On-Off-On-Off Alternating Pushbutton (#1684)
- P164 GP2Y0A21YK0F IR distance sensor (#164)
- PAM8302 Audio Amplifier (#2130)

- PM2.5 PMS5003 airborne particulate monitor
- QS-FS01 anemometers
- red LEDs
- SMA to uFL/u.FL/IPX/IPEX RF Adapter Cable (#851)
- T-slot photointerrupter
- TCA9548A mux I2C multiplexer (#2717)
- Adafruit "mini spy camera" (#3202)
- TSL2561 Digital Luminosity/Lux/Light sensor (#439)
- TSL2591 Digital Luminosity/Lux/Light sensor (#1980)
- tweezers (#422)
- ultimate GPS breakout board (#746)
- USB A / microB cables
- USB Cable - Standard A-B - 3 ft/1m (#62)
- USB isolator
- VL53L0X Time of Flight Distance Sensor (#3317)

If you'd like to fly an instrument package over, say, a farm, you can use the Physics 398DLP 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 we'll loan to each of you:
- wire stripper
- needle-nosed pliers
- tweezers
- eye protection

- breadboard
- digital multimeter (and spare battery, if necessary)

Tools/devices/material we'll share:
- soldering stations, solder, solder wick, "solder suckers"
- spools of brightly colored 22 gauge wire
- basic toolset (pliers, screwdrivers, etc.)
- Ultimaker 2+ printers
- PLA plastic filament, in various colors

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 with other members of your group, both in and out of class. Your 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
- learn to solder, then install pin headers on the breakout boards you'll use
- wire up a breadboard version of your data logger
- write (or adapt) code to talk to all the sensors you are using
- update a schematic diagram so that it represents your device
- load and test a PCB
- 3D-print a case
- write a data acquisition program
- figure out how to take data and verify its validity
- do your field work
- write code to analyze your data
- 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

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 a few 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 (Tuesday, Wednesday, or Thursday), each team will meet with me for 30 minutes to discuss progress, problems, clever ideas, and any other issues that might arise.

From time to time we may have experts speak to us about subjects relevant to your efforts.

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 electronic diary.

You are to upload a PDF version of your entire cumulative electronic diary to the course directory **every week**. Please put this into a single file; when you close one volume and open another, please upload a file that includes **all previous volumes** as well as the present one. Uploads are due at noon on Fridays, beginning with the second week. You should indicate the time/date for the material you enter, as well as with whom you've been collaborating.

You and your team will meet with me once per week for 30 minutes. All members of your team must attend, and must arrive promptly at the scheduled time, without exception. You must always bring your laptop, breadboard circuit, 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:

1. Reasonably detailed set of measurements and run plan are defined: week 3, conference with professor + TAs.
2. Breadboard fully loaded, all sensors can communicate with the Arduino (using demo software you've found or written): week 3, end of class.
3. Rudimentary DAQ (data acquisition software) can interrogate all sensors and write files to microSD memory: week 4, end of class.
4. Test run performed, and has generated valid data, using all of a group's breadboards: week 6, conference with professor + TAs.
5. PCB version of data logger finished and debugged: week 6, end of class.

6. Off line analysis generates meaningful plots of test run data: week 7, end of class.
7. 3D printed case design finished and fabricated: week 7, end of class.
8. Main data taking run has produced valid data, and has been plotted and analyzed: week 9, conference with professor + TAs.
9. Nearly final draft of paper is completely finished: week 10, ***start of class.***
10. Final version of paper is completely finished: week 12, ***start of class.***
11. Rewritten paper and to-the-class PowerPoint presentation finished: week 14, ***start of 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 20 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 398DLP 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 at least ten 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, 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 p398DLP 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 mile markers 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.
Office hours: On demand, by appointment. Each team will also have a 30-minute meeting with course staff every week so we can monitor your progress.

Teaching Assistant: Christian Williams, Loomis 458 (south door), cdw4@illinois.edu
Office hours: On demand, by appointment.

Teaching Assistant: Justin Languido, Loomis 458 (south door), jal2@illinois.edu
Office hours: On demand, by appointment.

Introduction and Syllabus 21

Course calendar

Note: the list of speakers and special topics is speculative, and will evolve.

what	day	date	comments
semester begins	Monday	August 26	
week 1	Friday	August 30	
week 2	Friday	September 6	
week 3	Friday	September 13	Communicating clearly
week 4	Friday	September 20	
week 5	Friday	September 27	
week 6	Friday	October 4	
week 7	Friday	October 11	
week 8	Friday	October 18	
week 9	Friday	October 25	Writing clearly
week 10	Friday	November 1	Nearly-final draft of report due
week 11	Friday	November 8	
week 12	Friday	November 15	Final draft of report due
week 13	Friday	November 22	
Thanksgiving break			
week 14	Friday	December 6	Rewritten report and PPT presentation due
end of term conferences	Monday – Wednesday	December 9 – 11	
reading day	Thursday	December 12	

Detailed syllabus

Week 1, in class

- 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 GG and TAs.
- Sign out the tools and hardware needed to build your devices.
- Install the Arduino programming IDE.
- Install the latest version of Anaconda Python
- Plug the Arduino into a USB port on your laptop, download the blinking LED sample program described in GG's p398dlp_code_primer.pdf file. (See the course's Code & design resources repository web page.) Confirm that the LED really does blink. (You'll want to create a sensibly named folder to hold your Arduino programs.)
- Lecture/tutorial: we'll go through GG's "C++ and Python Primer/Refresher." Follow along on your laptop.
- Modify the blinking LED program to flash your first and last initials in Morse code.

- Get a soldering lesson from Todd Moore, an electrical engineer who is staffing the undergraduate physics program. Solder (male) pin headers onto the bottom of a BME680 temperature sensor. Also solder a 16-pin (male) header onto the bottom of an LCD.
- Install the power terminals and plastic feet onto your breadboard and duct-tape your Arduino to the surface of your breadboard, but not on top of any of the interconnect holes.
- Install a BME680 on your breadboard, following the instructions on the Adafruit site, and download the demonstration software to communicate with it. Make it report what it sees in a serial monitor window. (Hands-on demo by GG if desired.)
- Install the LCD onto your breadboard, find some demonstration software, and display some text on the LCD. (Note that you'll need to install a 10k potentiometer too.)
- Create a new Arduino program that will read information from the BME680 and display it on the LCD.
- Select/assign groups for the following reports:
  ☆ How an electret microphone works
  ☆ What's inside an LSM9DS1 9 DOF sensor, and how its 3-axis accelerometer works
  ☆ How a BME680 measures *atmospheric pressure*

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 398DLP documents I've given out in class.
- Install a keypad on your breadboard, find some demo software that recognizes (and responds to) keypad activity.
- Modify your BME680 + LCD program so that you can use the keypad to select whether to display only the the temperature, only the pressure, or only the humidity on the LCD.
- 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.

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
- Spread over the class period, hear groups' reports on those three topics.
- If you didn't do this last week, get a soldering lesson from Todd Moore. Solder pin headers as necessary onto the remaining breakout boards you will need for your breadboard circuit.
- 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. You should already have soldered a 16-pin (male) header onto the bottom of an LCD.
- 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. You'll probably need to step outside when it's time to test your GPS board.
- Select/assign groups for the following reports:
  ☆ An introduction to the Arduino Mega 2560
  ☆ How a successive approximation ADC works
  ☆ How a BME680 measures ***temperature***

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.
- Download Cura 3.6. Have Cura generate a gcode file from your STL file.

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.
- Milestone 1: "Reasonably detailed set of measurements and run plan are defined."

Week 3, in class
- Five-minute (PowerPoint) progress reports from each group: describe your project to the class, and tell us your thoughts on how you'll approach it.
- Spread over the class period, hear groups' reports on those three topics.
- Finish Milestone 2 by the end of class: "Breadboard fully loaded, all sensors can communicate with the Arduino (using demo software you've found or written)."
- Tentative class visitor: Ms. Celia Elliott (Physics). *Communicating clearly*.
- Do further DAQ code development, and start on offline Python work.

Introduction and Syllabus   24

- Hands-on TinkerCad demo by GG. You will take notes as you follow along.
- Select volunteers for the following reports:
  - ☆ I2C communication protocol
  - ☆ How a BME680 measures *humidity*
  - ☆ How the MCP4725 DAC 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 me apprised of your progress. The first version of your DAQ should be fairly far along by the time we meet.

Week 4, in class
- Spread over the class period, hear groups' reports on those three topics.
- Select/assign groups for the following reports:
  - ☆ Interrupts
  - ☆ Adafruit's "Ultimate GPS" breakout board
  - ☆ How a BME680 measures atmospheric *volatile organic compound concentration*
- Finish Milestone 3 by the end of class: "Rudimentary DAQ (data acquisition software) can interrogate all sensors and write files to microSD memory."
- Solder headers, capacitors, resistors, etc. onto your PCB.
- 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 in a few weeks? 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.
- I will have a few soldering irons in my lab (458 Loomis). Stop by to do more soldering on your PCB as your time permits.
- 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.

Physics 398DLP, University of Illinois   ©George Gollin, 2019

Week 5, in class
- Spread over the class period, hear groups' reports on those three topics.
- Select/assign groups for the following reports:
  - ✮ Atmel 2560 microcontroller timer modules
  - ✮ PWM (pulse width modulation) and the PAM8302 amplifier.
  - ✮ How the PM2.5 particulates monitor works
- Finish PCB soldering and checking/debugging the board. It should be electrically equivalent to your breadboard.
- Finish your DAQ.
- Tentative class visitor: *Sustainability*.

Post-class assignment
- Run a quick field test. For example, if you plan to study Amtrak trains, ride an MTD bus for a half hour.
- 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, if we don't have the MakerLab facility reserved.

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.
- Milestone 4: "Test run performed, and has generated valid data, using all of a group's breadboards."

Week 6, in class
- Five-minute (PowerPoint) progress reports from each group: describe your field tests.
- Spread over the class period, hear groups' reports on those three topics.
- Select/assign groups for the following reports:
  - ✮ How the LSM9DS1's 3-axis gyroscope works
  - ✮ VL53L0X Time of Flight Distance Sensor
  - ✮ High speed microSD data transfer issues
- Finish Milestone 5: "PCB version of data logger finished and debugged."
- Finish your DAQ and offline analysis code.

Post-class assignment
- Do a much more extensive field test, this time using your PCB, installed in its case if it is ready (and using your breadboards if your project requires more than one sensor package per person).
- Generate analysis plots from your test run data.

Week 7, conference with the professor + TAs
- Progress reports from you, especially what you found in your field tests.

Physics 398DLP, University of Illinois   ©George Gollin, 2019

- 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
- Spread over the class period, hear groups' reports on those three topics.
- Finish Milestone 6: "Off line analysis generates meaningful plots of test run data."
- Finish Milestone 7: "3D printed case design finished and fabricated."

Post-class assignment
- Perform full-scale data taking runs, check that your data are valid and readable.
- Perform whatever auxiliary measurements are necessary to calibrate your devices.
- Run your data through your offline analysis.
- 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
- 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.

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 398DLP.

Week 9, conference with the professor + TAs
- Milestone 8: "Main data taking run has produced valid data, and has been plotted and analyzed."
- Discuss your analysis, conclusions, and project report outline with us.

Week 9, in class
- 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.

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
- Milestone 9: "Nearly final draft of paper is completely finished." This is due ***at the start of class***.
- Peer-reading/evaluating/commenting/correcting of your drafts.
- More work on paper and analysis as necessary.

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
- 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
- Milestone 10: "Final version of paper is completely finished." This is due ***at the start of class***.
- Class discussion: How'd it go?
- Peer-reading/evaluating/commenting/correcting of your project reports.

Post-class assignment
- Begin rewriting your 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.

Post-class assignment
- Finish rewriting your project report.
- Create a PowerPoint presentation, ~20 minutes in length, describing your results to your classmates.

Week 14, conference with the professor + TAs
- Paper and PPT discussions.

Week 14, in class
- Milestone 11: "Rewritten paper and to-the-class PowerPoint presentation finished." This is due *at the start of class*.
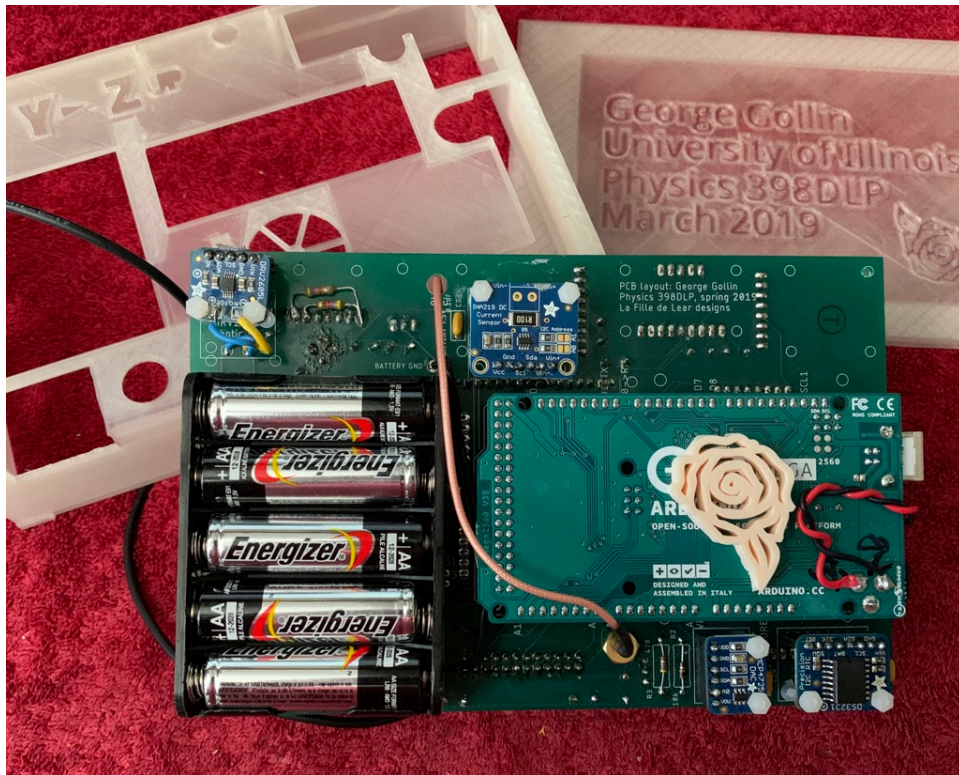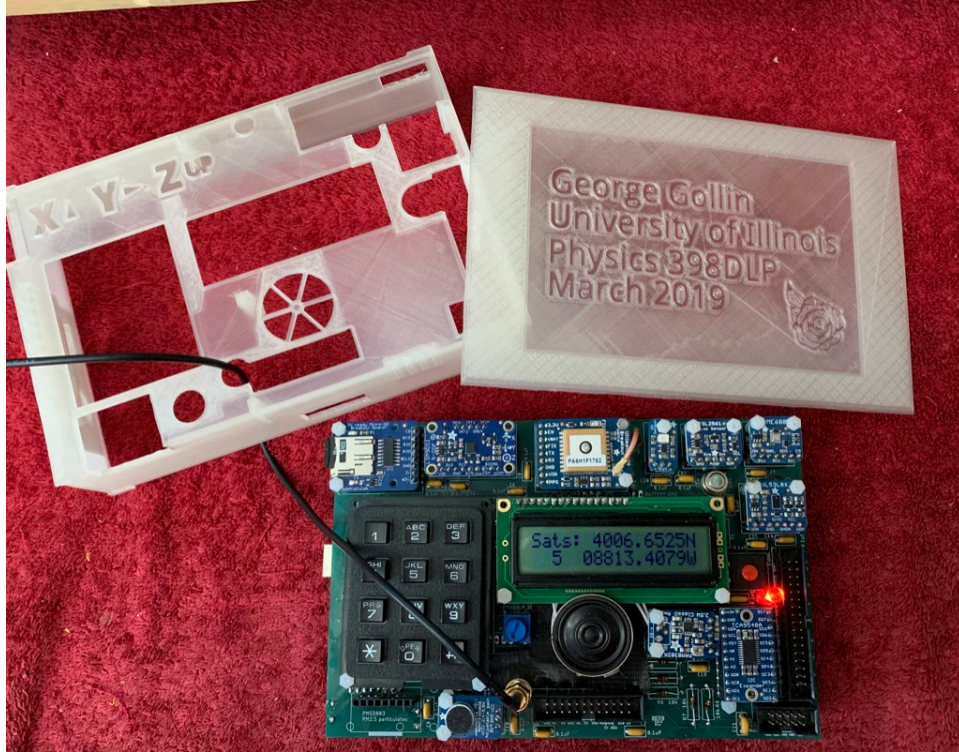- PPT presentations by each group

Reading Day
- Meet at GG's house for a Serious Pizza Party !

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.

Physics 398DLP
Design Like a Physicist
Fall 2019

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…

…are in the course syllabus. See the "Introduction and Syllabus" document I distributed, or else the syllabus page on the course website.

Groups, projects, electronic diaries

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 me for 30 minutes. My preference is sometime on Tuesday (10 am – 1 pm; 2 pm – 4 pm), Wednesday (10 am – 4 pm), or Thursday (10 am – 1 pm; 2 pm – 4 pm).

Open up your diary file, 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.

Distribution of stuff!

I have an alarmingly large amount of stuff available for you. Let's start slowly, though if you have a reasonably clear idea of what your project might involve, feel free to take more. I'd suggest you fetch the following parts from the supply bins, assuming you're just going to be working on your breadboard circuit for the time being.
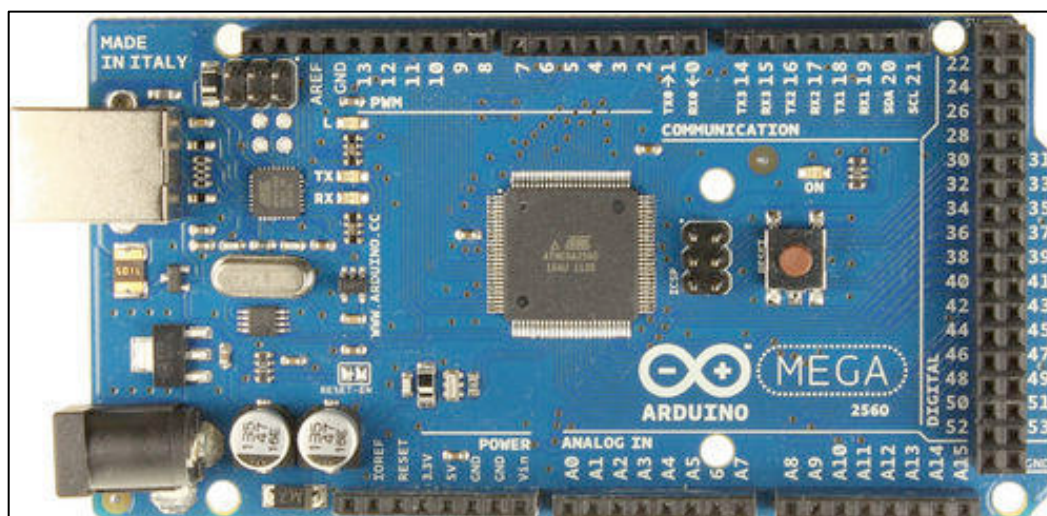
part	description	I2C address	quantity you'll need
Arduino Mega 2560	microcontroller		1
BME680	T/P/RH/VOC	0x76 or 0x77	1
DS3231	real time clock	0x68	1

INA219	voltage/current sensor	0x40	1
MicroSD card breakout	holds microSD card	SPI	1
3 x 4 keypad	tap tap tap		1
16 × 2 LCD	liquid crystal display		1
0.1uF capacitors	for 5V bypass		about 20
10kΩ potentiometer	sets contrast for LCD		1
of/off/on/off pushbutton	battery vs. USB power		1
AA batteries and case	power to go		1 case; 5 batteries
CR1220 batteries	batteries for GPS and RTC		1
MicroSD memory card	8 GB SDHC		1
22 gauge wire	for breadboarding, various colors		several feet of each
USB cable	talk to the Arduino		1
Breadboard	for building your prototype; on loan		1
cardboard box	for your breadboard, parts, and tools		1
digital multimeter	on loan for the semester		1
wire stripper	on loan for the semester		1
tweezers	on loan for the semester		1
eye protection	on loan for the semester		1
needle-nosed pliers	on loan for the semester		1

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. See information in the Physics 398DLP "C++ and Python Primer" for more details about how to do this.

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

Arduino Mega 2560

It's 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. See https://store.arduino.cc/arduino-mega-2560-rev3 for more details.

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.

Arduino Integrated Development Environment editor window

Install the Anaconda Python IDE

See the "C++ and Python Primer" for more details about how to do this.

Soldering

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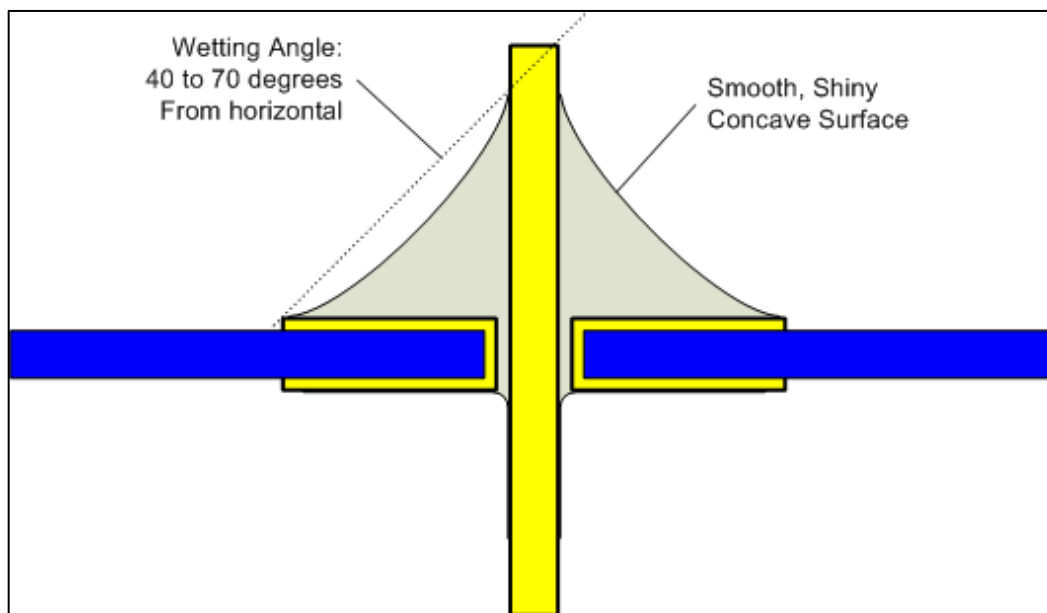
Heat the pin and pad with your soldering iron, not the solder itself. As it melts, the solder will flow onto the conductors you've heated.

Types of Solder Joints
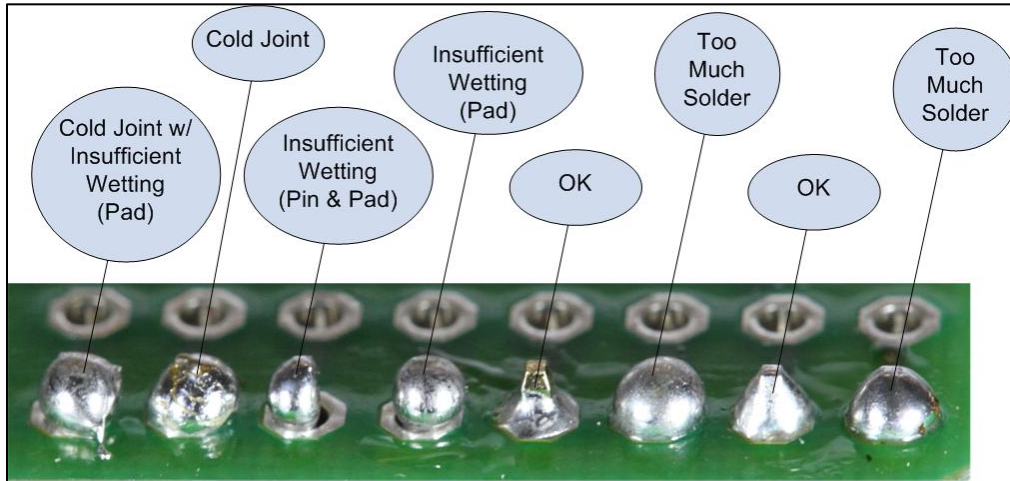
(NOT) The Proper Solder Joint

Overheated joint

Solder bridge

Lifted pad from overheating or desoldering

It's time to build Version Zero of your data logging device.

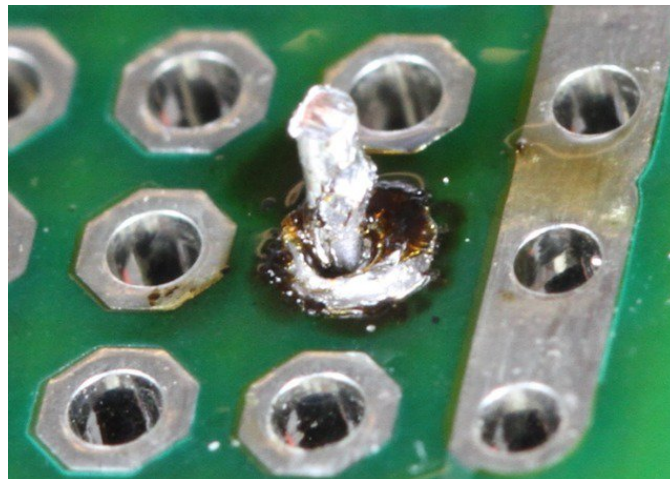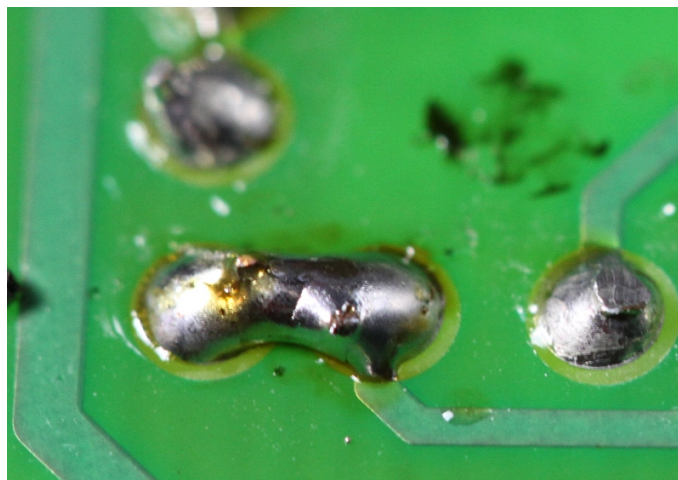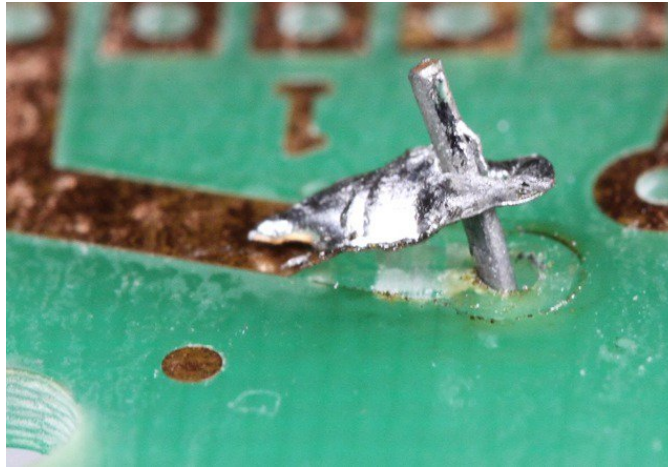Let's prep our breadboards. Please attach the connectors and rubber feet to your breadboard. Please fasten an Arduino, still in its plastic carrier, to your breadboard. I recommend duct tape (the baby sitter's friend!); position the device to that it doesn't cover any of the breadboard's plastic structures that are used to hold components.

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 today. Please have Todd teach you how to solder! You'll attach pin headers to a few of the sensor breakout boards in your collection of goodies. Please start with a BME680 breakout board and an LCD.

Temperature, relative humidity, etc.

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) 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.

Download from Adafruit a sample program that communicates with the BME680. Compile the code, download it to the Arduino, and run it!

Get it to run! 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.

Liquid crystal display

Take a look at the (somewhat confusing) schematic diagram on the course web site for last semester's data logger. Use this as a guide for how to wire up the LCD and necessary 10kΩ potentiometer.

Download the demo code I wrote last fall to drive the LCD; it's available on the course web site's "Code and design resource repository" page. Compile this, load it into the Arduino, and make it work! You'll need to play around with the potentiometer setting to be able to see something on the LCD. You'll probably need the voltage on the LCD's VO pin (which you set with the potentiometer) to sit at very roughly +1.5 volts.

Keypad

The "Post-class assignment" asks you to work with the 4 × 3 keypad. Take a look at the schematic diagram and the sample code on the course website for something you can download, and modify to suit your needs.

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 TAs for assistance in making sense of how to approach all this technology.

Physics 398DLP
Design Like a Physicist
Fall 2019

George Gollin
University of Illinois at Urbana-Champaign

Week 2: Breadboarding, TinkerCad
▷ supplemental material ◁

Week 2 supplemental material   3

Week 2: Breadboarding, TinkerCad

Goals for this week…

…are in the course syllabus. See the "Introduction and Syllabus" document I distributed, or else the syllabus page on the course website.

The microSD breakout board uses SPI, not I2C

The microSD card breakout board will use the SPI (Serial Peripheral) interface communications protocol, rather than the slower I2C—Inter Integrated Circuit—protocol used by the BME680. There's demonstration software at the Adafruit site that you can use to test it. Wire it up as shown in the schematic.

Installing the Real Time Clock

When you assemble the DS3231 real time clock module you'll need to solder a pin header, then the backup battery carrier (careful when doing that!) before sliding in a CR1220 battery.

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

Find some demo software and use it to set the clock's time.

Register an Autodesk user account, then visit the TinkerCad website

In the Post-class assignment for week 2 I ask you to start using TinkerCad. Go to https://knowledge.autodesk.com/customer-service/account-management/education-program/create-education-account/create-account-students-educators to register an account with Autodesk as a student. You'll have much better (free!) access to Autodesk's tools this way than you would if you registered a non-student account, even if it were free.

All I want you to do is to figure out how to engrave your name into the lid of a small plastic box!

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.

Week 2 supplemental material   4

Cute Drawer Box With Lid

GEPL
5 years ago
PUBLIC
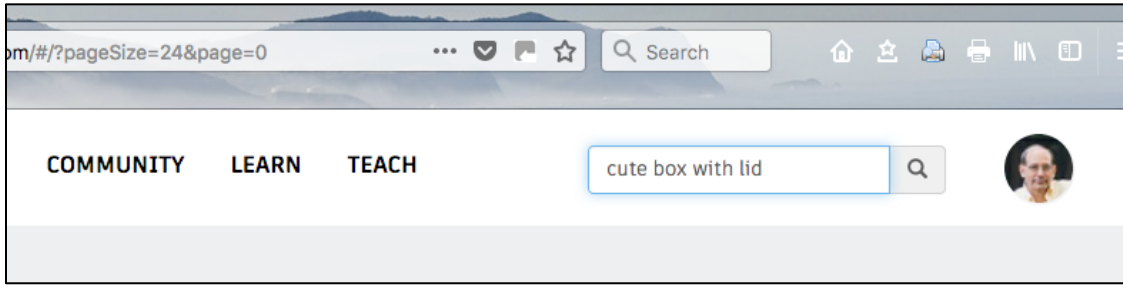
Click on the image, then select "Copy and Tinker." Here's what the working area looks like:
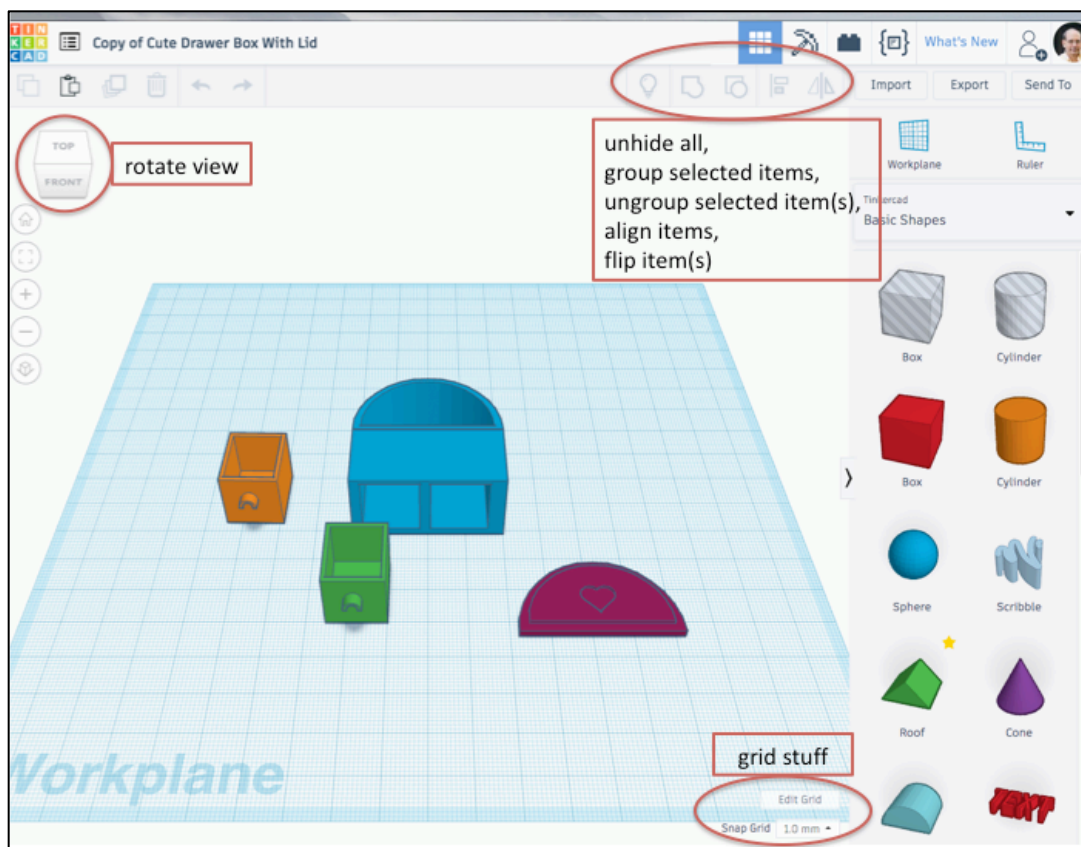
Consider engaging with a basic TinkerCad tutorial to see how to do simple things. But you'll probably find that there are plenty of good YouTube videos that will show you how to do most anything you want. Make very brief notes on any tricks that you find useful, for later recall and reference. I'll give you a hands-on TinkerCad demo next week.

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.

Download and install the most recent version of Cura

The trio of 3D printers in my lab are built by Ultimaker B.V., a Dutch company. Cura is their software product that generates gcode files (the inputs to a 3D printer) from STL files.

Download and install the latest version of Cura 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; please just try to muddle through it.

Week 2 supplemental material   6
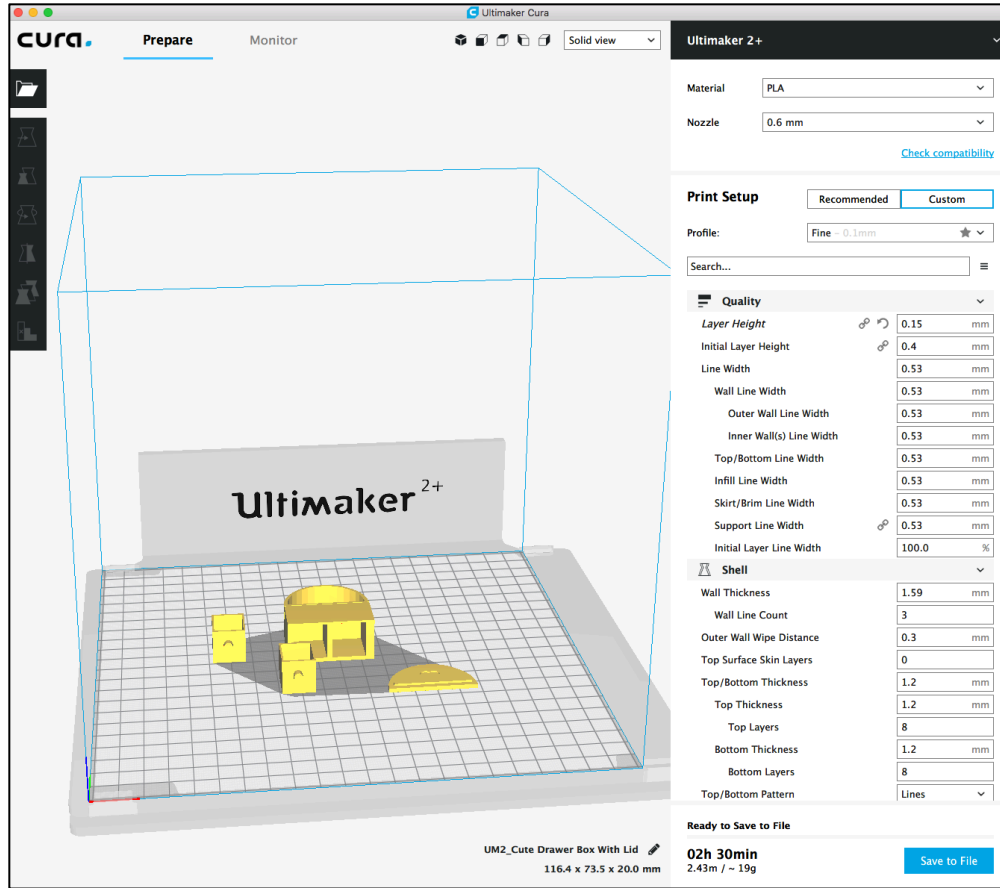
Next week I'll talk you through the Cura "slicing" process and discuss a couple of settings you'll want to impose. A couple of technical issues we'll consider: (1) adhesion to the build surface; (2) support for bridged features. I'll explain what I mean in class.

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

Physics 398DLP
Design Like a Physicist
Fall 2019

George Gollin
University of Illinois at Urbana-Champaign

Week 3: DAQ, TinkerCad, Schematic Capture
▷ supplemental material ◁

Week 3 supplemental material   2

Week 3: DAQ, TinkerCad, Schematic Capture

Goals for this week…

…are in the course syllabus. See the "Introduction and Syllabus" document I distributed, or else the syllabus page on the course website.

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.

Please take a look, if you're interested, at chapter 26 in the Atmel 1560 manual on the course web site for more register information:
https://courses.physics.illinois.edu/phys398dlp/sp2019/code/Atmel_2560_datasheet.pdf.

Week 3 supplemental material   5

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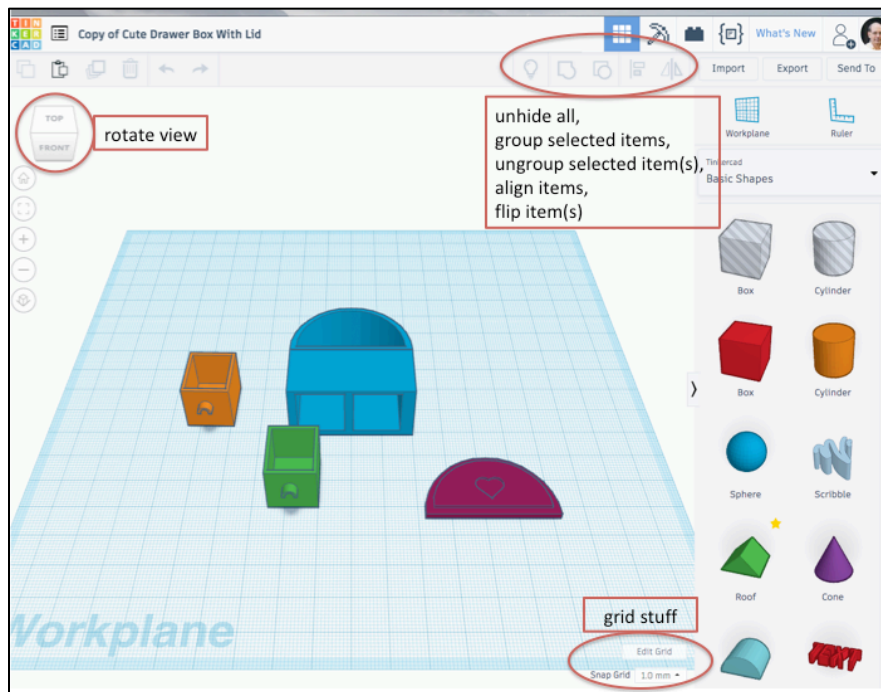
You should find something amusing to work on, then follow along. Take notes. We'll try some of these actions:

- 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.

Cura parameters

The most vexing issue in generating good 3D prints is obtaining good adhesion between the object being built and the (glass) build surface.. Many of Cura's parameters relate to this. I'll talk you through some of them.

Physics 398DLP, University of Illinois   ©George Gollin, 2019
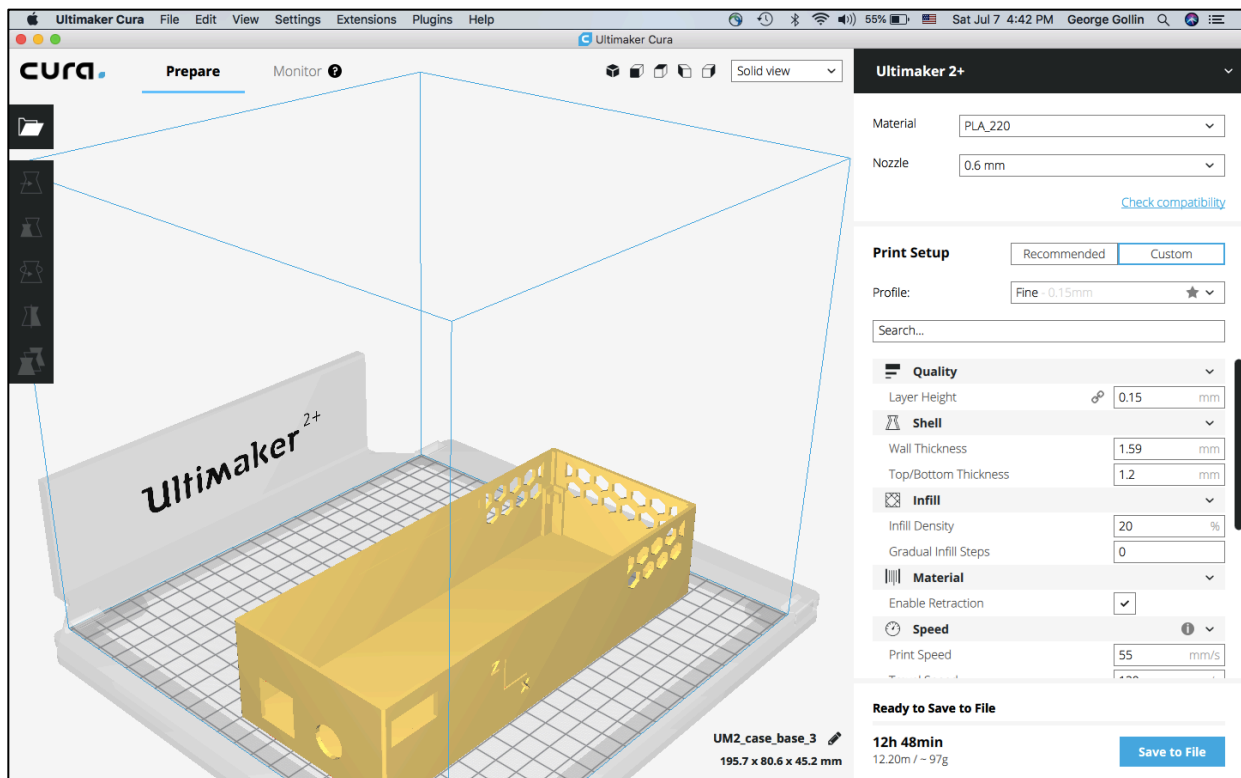
Whole-class discussion

- 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?

DAQ time.

"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.

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.

Group work: generate an outline or flowchart of an initial version of your offline analysis code.

Post-class assignment: log in to Autodesk and download EAGLE

In the Week 3 post-class assignment I ask you to install the EAGLE schematic capture tool. 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. You'll do a little with EAGLE after the end of class this week. But here's a quick look: this is my schematic for a prototype data logger from last semester, viewed in EAGLE.
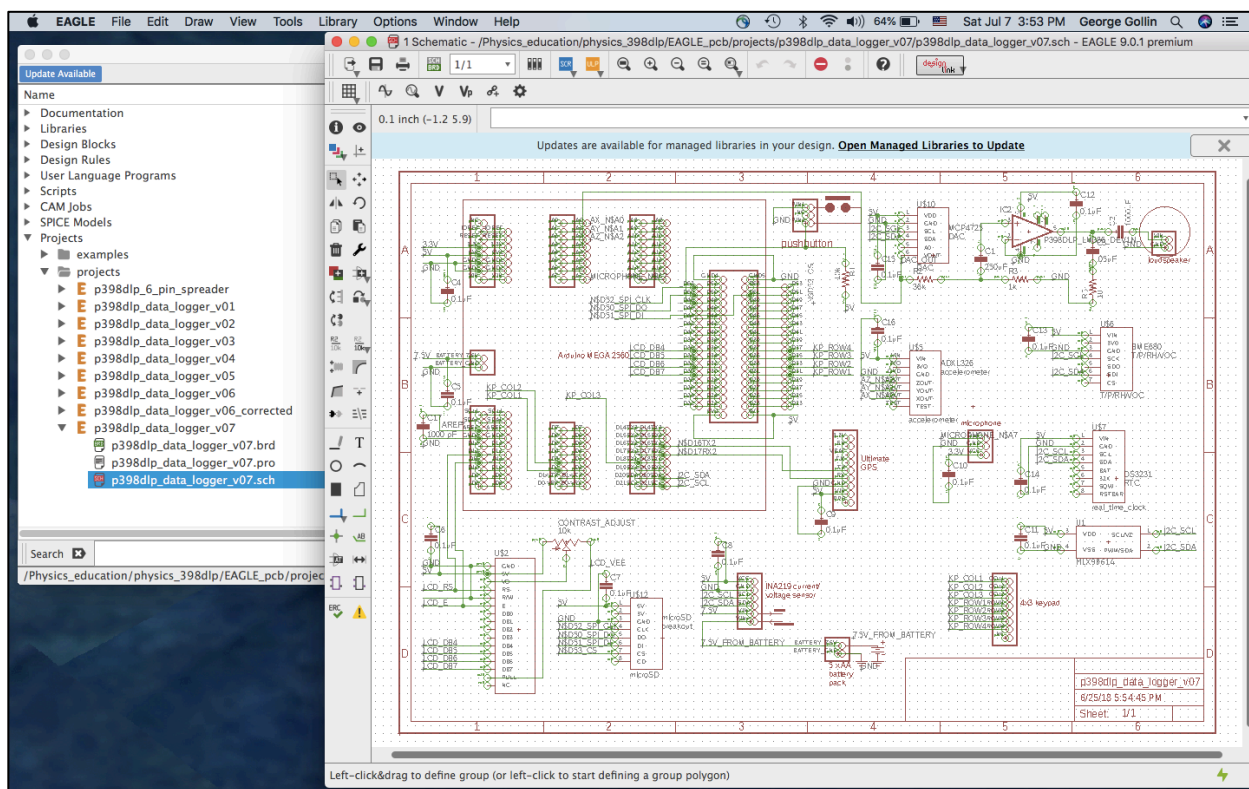
Circuit schematics and PCB layouts

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. Here's an example of (a small) part of a schematic for a mixed-signal board I designed for the CLEO III experiment at Cornell some years ago. (A mixed signal board holds both analog and digital signal lines.)

Week 3 supplemental material   8
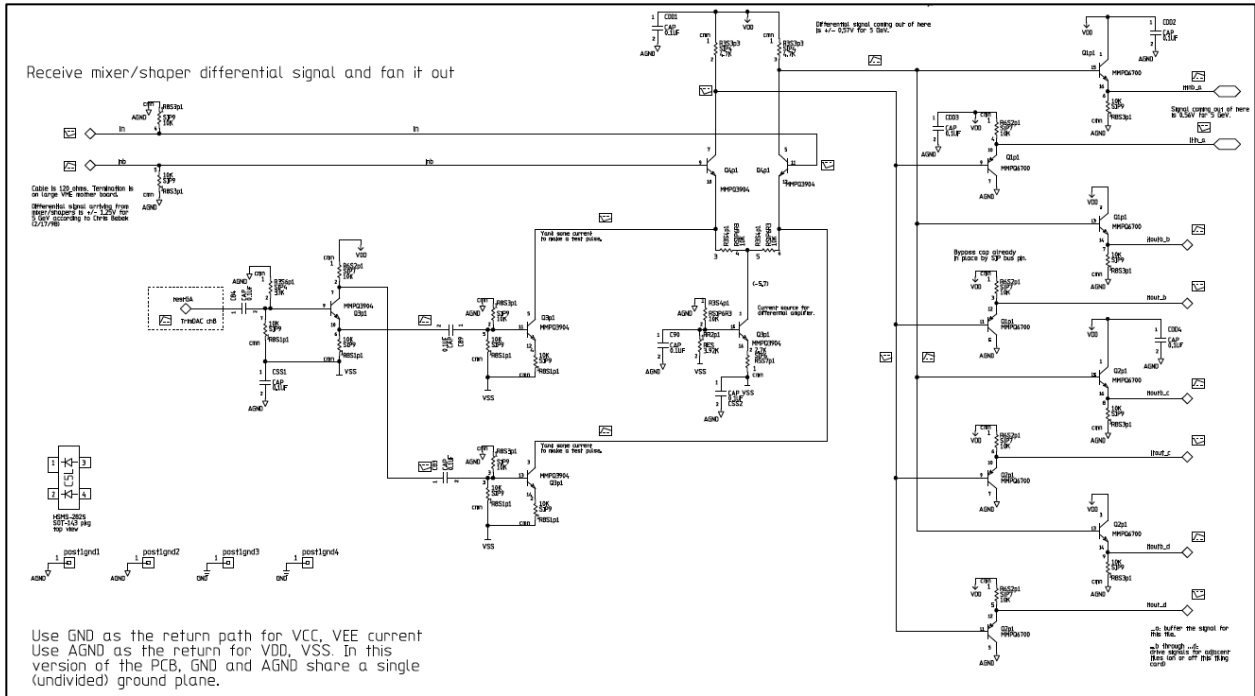
If you go to the Adafruit site to see how to wire up an ADXL326 accelerometer, you will probably see a representation of the circuit that looks like the following. That's not a schematic: though the illustration clearly represents the components and interconnections it is inefficient in its use of space.

Physics 398DLP, University of Illinois   ©George Gollin, 2019
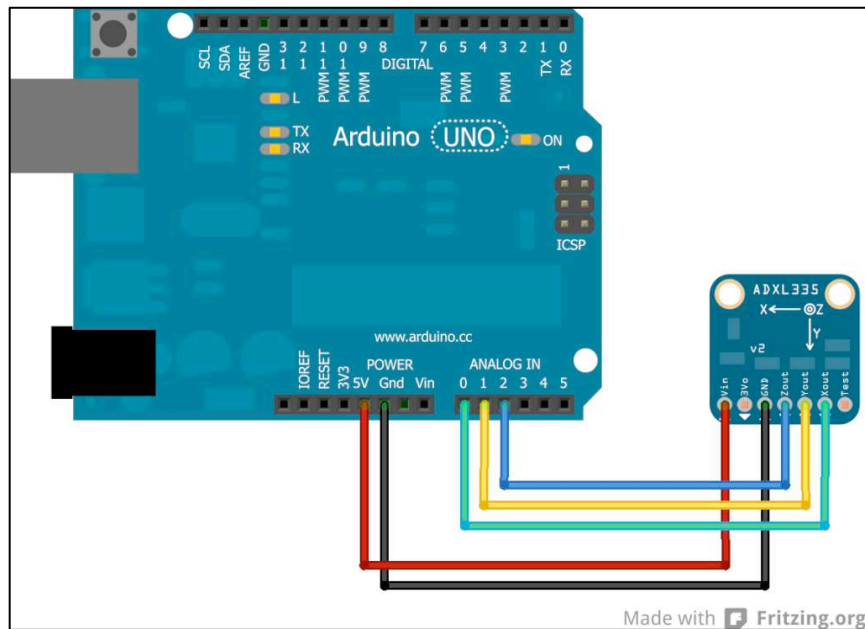
Did any of you play with Colorforms when you were little? It's a creative toy which includes a variety of thin vinyl shapes—circles, rectangles, and the like—that will adhere to the smooth, black surface that a child uses as a drafting environment.

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.

Autodesk EAGLE schematic capture

Let's say you'd like to create the schematic for your breadboard circuit. To do so, please open EAGLE. You should find that it starts by displaying the control panel.

Week 3 supplemental material 10
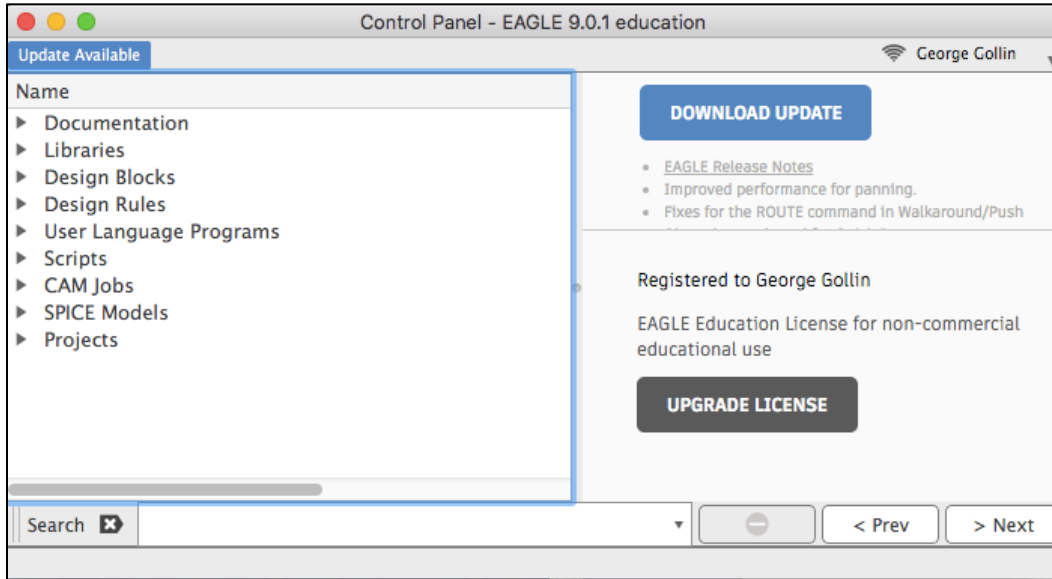
Click on the arrowheads next to "Libraries," "Managed Libraries,", and "74ac-logic.lbr, then click once on "74AC11000." The device displayed is a quad 2-input NAND.
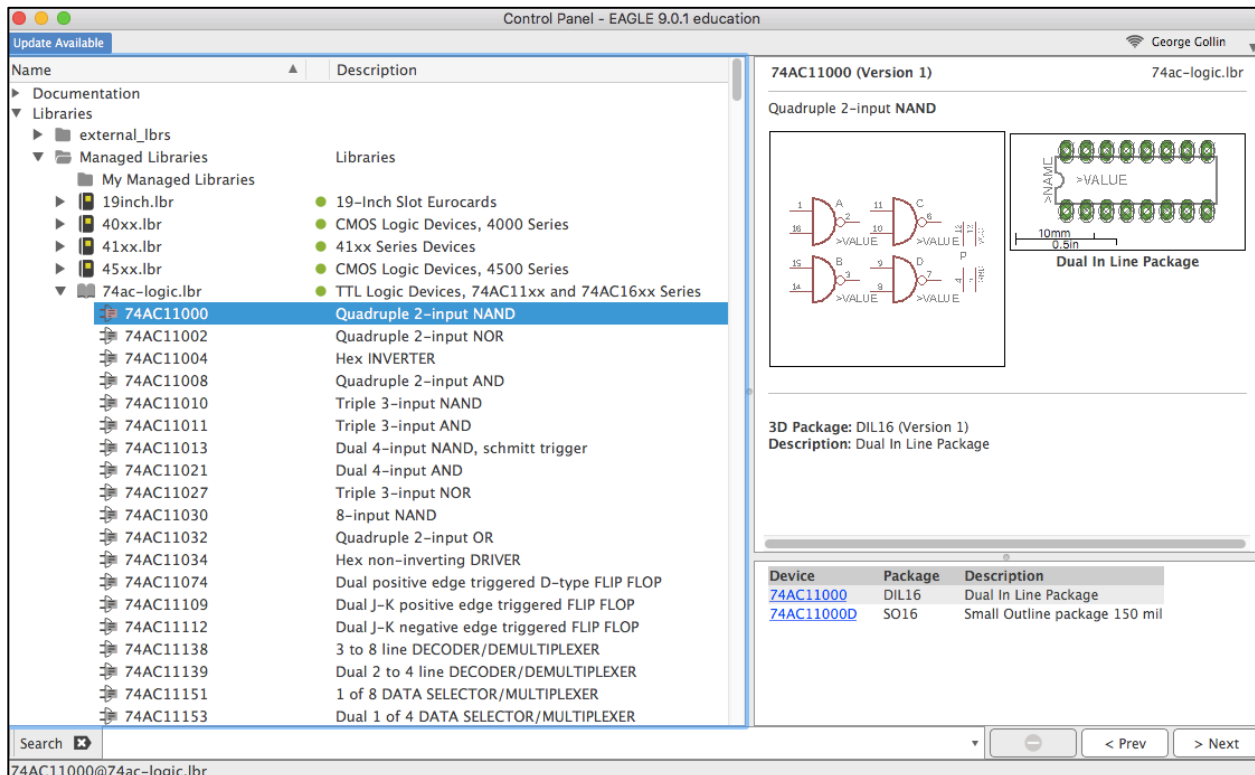
"TTL" stands for "transistor-transistor logic." It's a logic family of integrated circuit chips in which 0 (or false) is represented by a voltage between ground and 0.8 V, while 1 (or true) is represented by a voltage between about 1.8 V and 5 V. The truth table for a NAND gate

is simple: the gate's output is the logical complement of an AND of its inputs. The NAND's output is 1 if either or both of its inputs are 0. If both inputs are 1 the output is 0.

Take note of the two diagrams for the device. The one on the left is what EAGLE will offer to you to place on a schematic. The one on the right represents the actual footprint, including conducting pads to which you'll solder the chip's pins, for installation on a printed circuit board. Since this chip is available in two different packages, there are two choices listed in the Device/Package/Window box at the bottom. DIL16 is a 16-pin "dual inline package" while SO16 is a 16-pin "small outline" package.

We're not actually going to use this part, but I thought it would make a good example.

If you haven't already done this, please create a sensibly-named folder into which you will store all your EAGLE stuff. (Mine is named Eagle_pcb.) Inside this folder create another folder named external_lbrs. Go to the course web site and download https://courses.physics.illinois.edu/phys398dlp/fa2019/Eagle_pcb/external_lbrs/p398dlp_eagle_libraries.zip, then unpack it and move the files it contains into your external_lbrs folder. You may need to go to View → Refresh to update EAGLE's awareness of the libraries it has at its disposal.

In the EAGLE control panel window look inside external_lbrs/p398dlp.lbr. You'll see a number of parts that I have created for your convenience. Here, for example, is how the ADXL326 accelerometer presents.
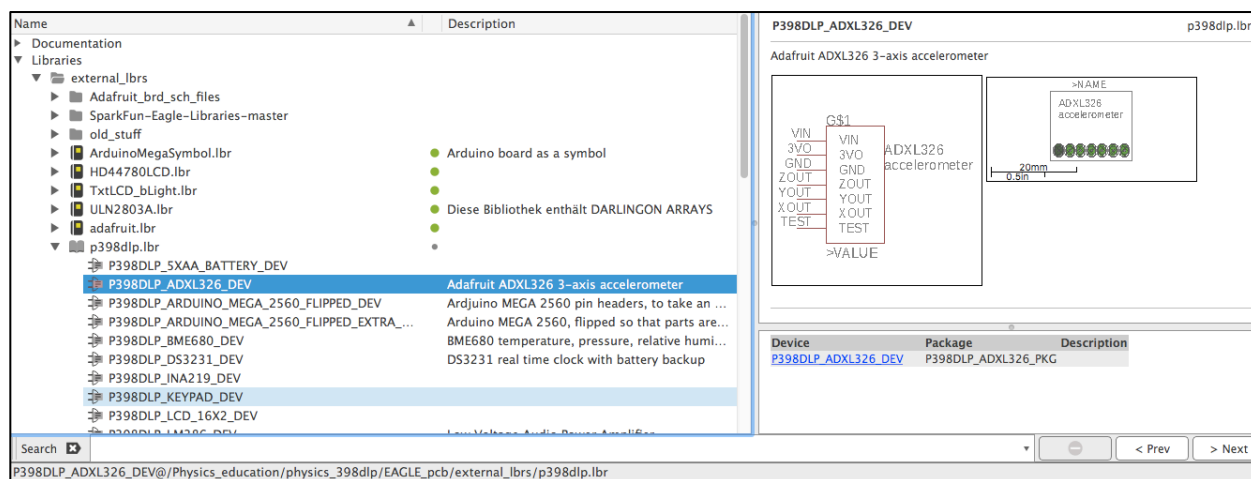
I've also created a part called "P398DLP_ARDUINO_MEGA_2560_FLIPPED_DEV." It's an Arduino Mega 2560, but upside down so that it can be plugged into properly positioned pin headers on a circuit board.

To help you visualize what I mean, here's a view of the EAGLE symbol that is used when designing a printed circuit board, as well as a pair of photographs of an Arduino installed on a prototype data logger. To orient yourselves, take note of the positions of the power jack and USB connector.

Week 3 supplemental material   12
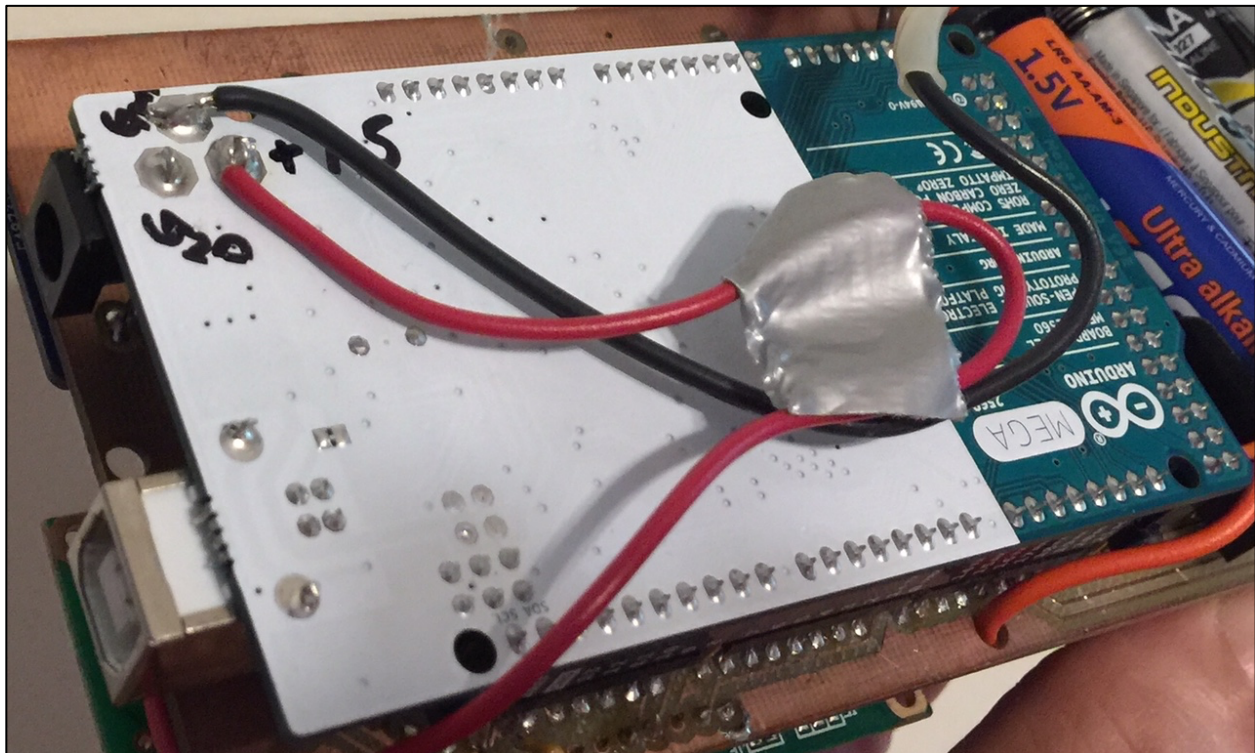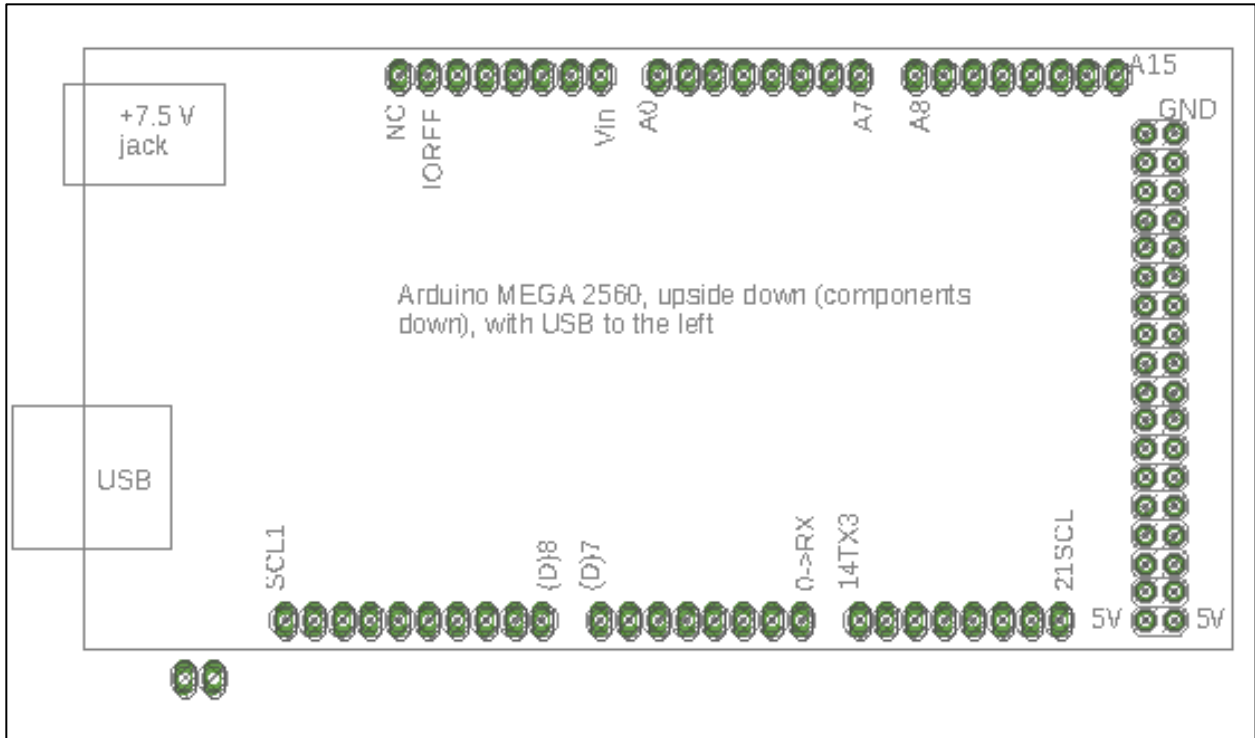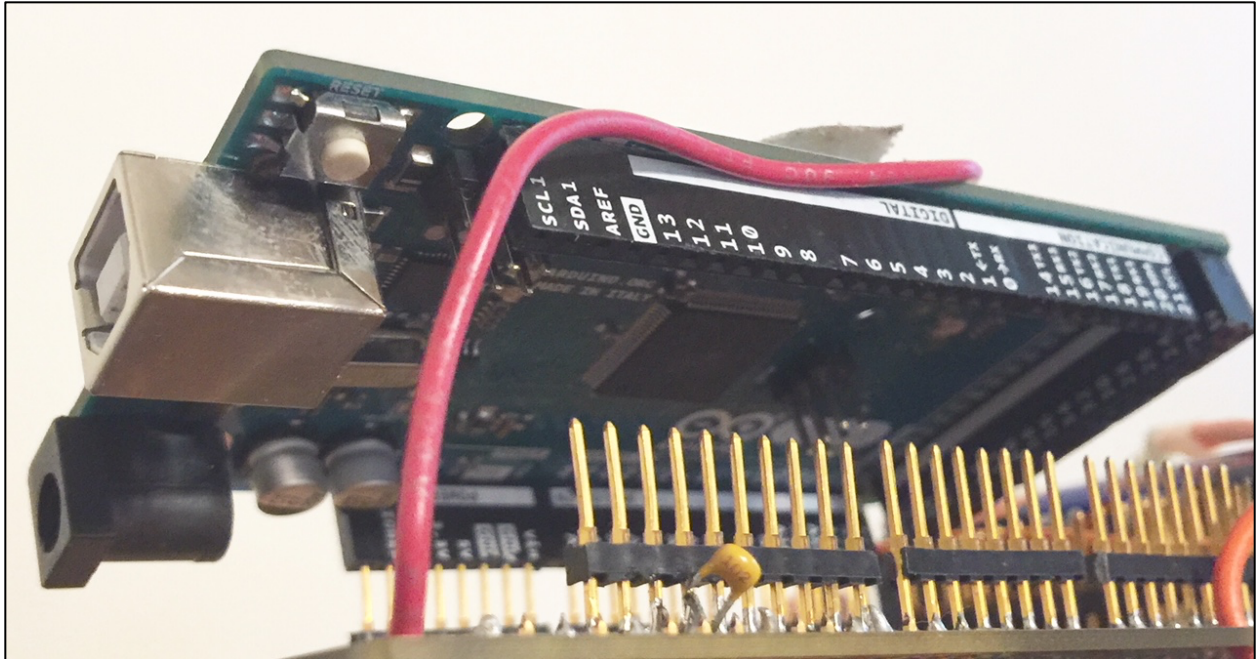
Note that this isn't the same as the representation of the Arduino that you'll use when defining the schematic if your circuit: there's no need for the placement of pins on the schematic symbol to bear any resemblance to the positions of the pins on the actual device. Here's what the symbol looks like after it's placed on a schematic.
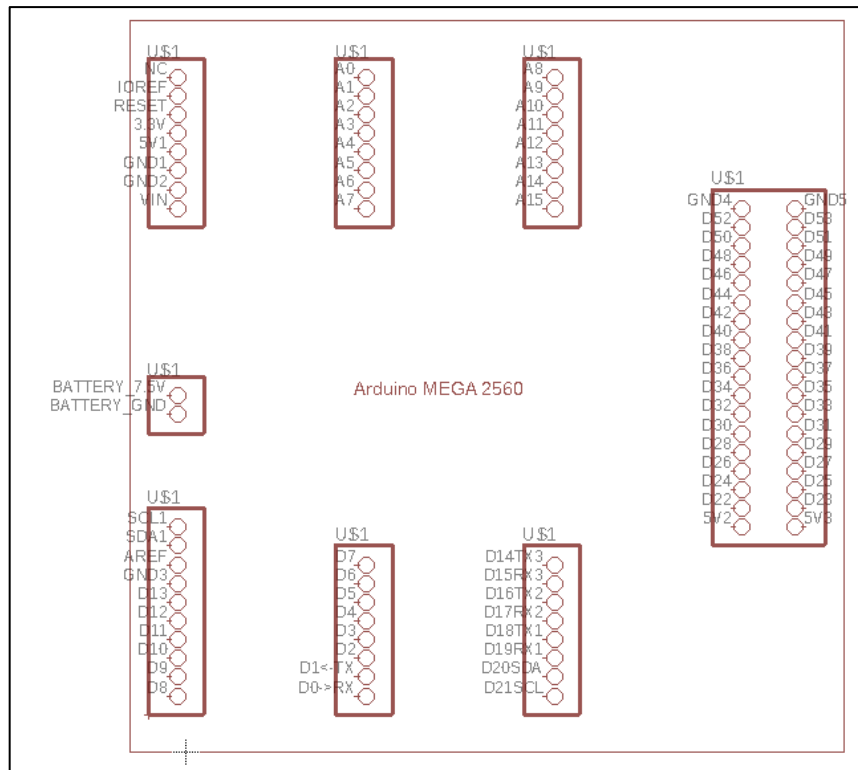
Capturing your circuit's schematic

By now you should have—at minimum—an Arduino, an LCD, and a BME680 on your breadboard. In the same folder that you've used as the home for your external_lbrs folder, create a new folder named "projects." In EAGLE, go to File → New → Project:
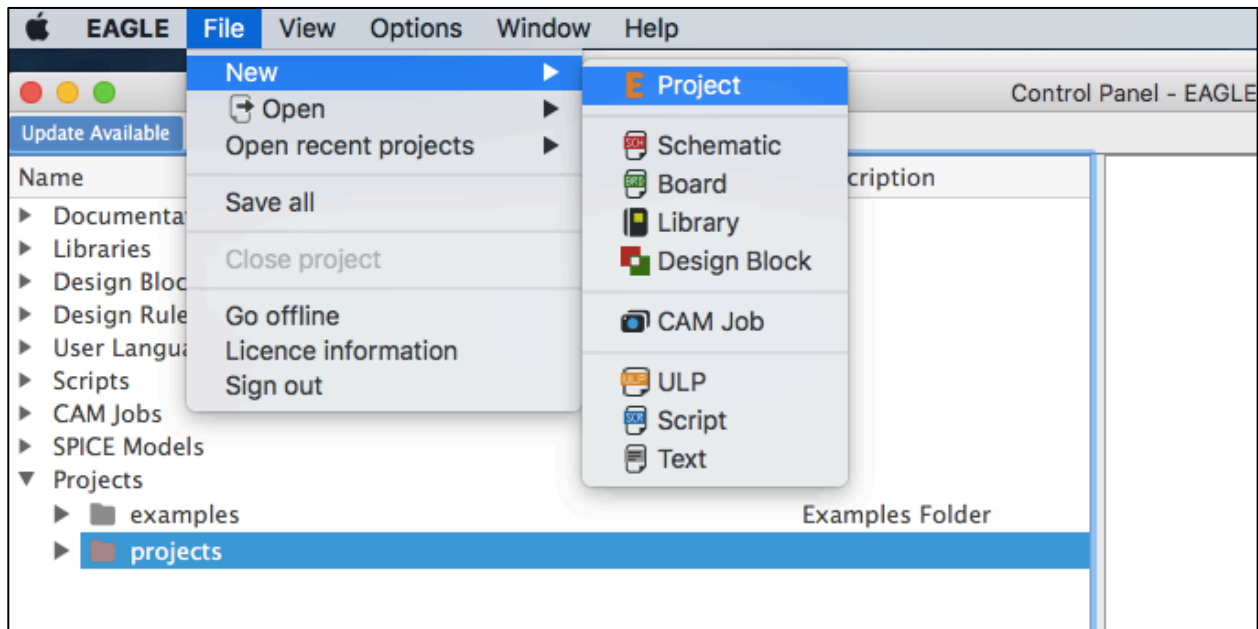
Give your project a sensible name, of course. If the project doesn't land in your projects folder you can drag it over to that folder from wherever EAGLE creates it. I am calling my project "p398dlp_unit2."

Right click on the project and go to New → Schematic.

Week 3 supplemental material   15
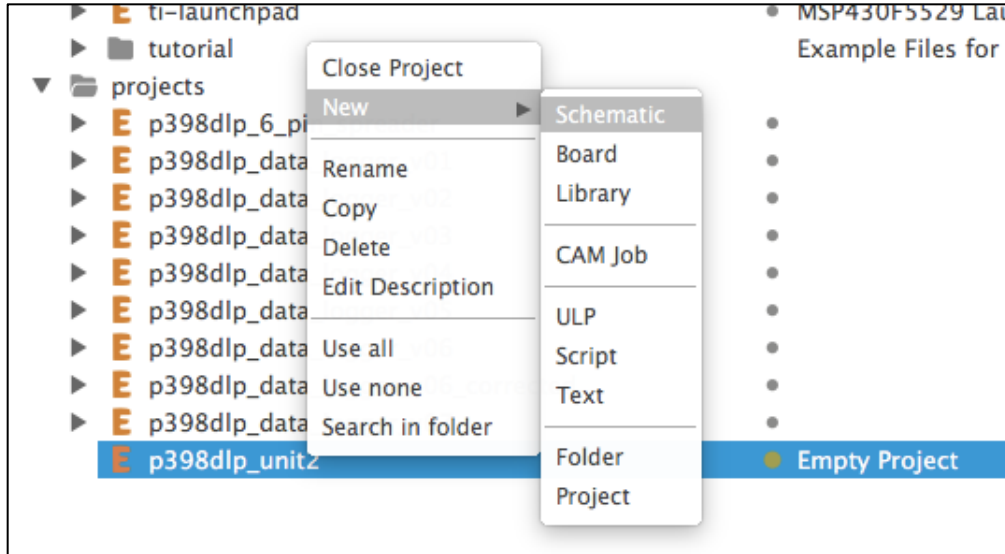
The schematic capture window will open. It should look something like this:
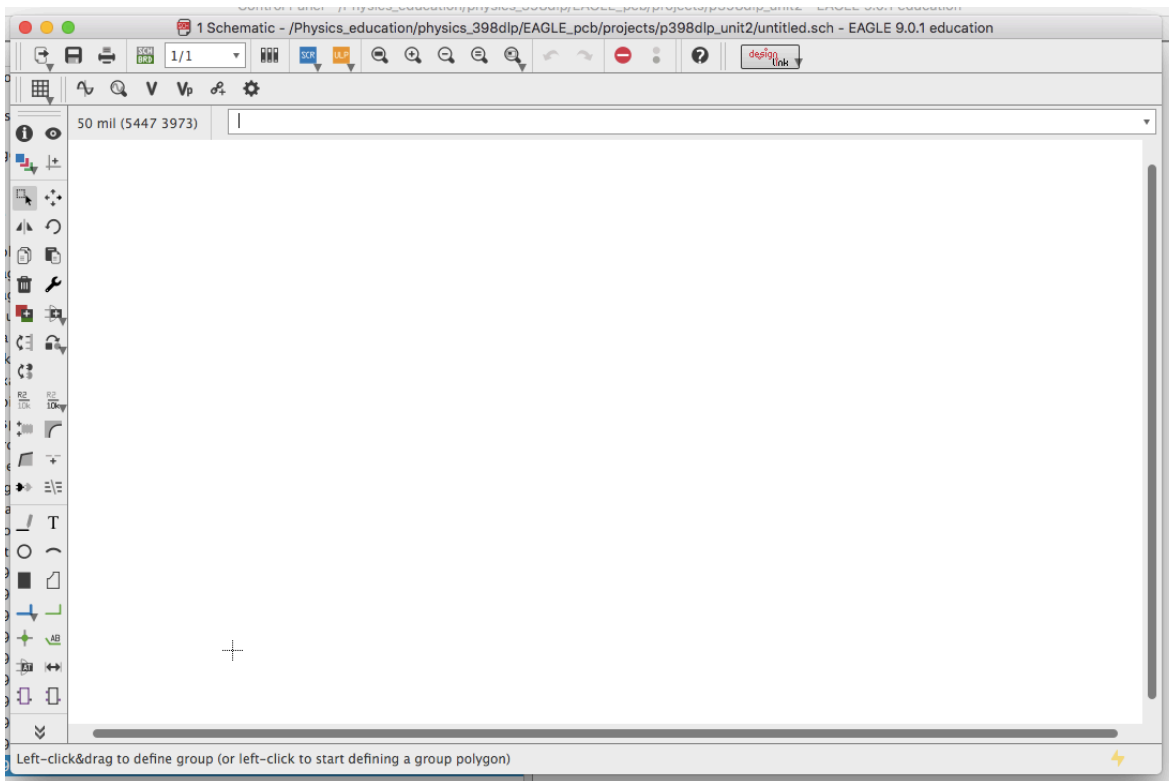
Looks intimidating, right? Don't worry—you'll get the hang of it pretty quickly. Here's the meaning of some of the tiles on the toolbars at the left and top of the frame.

Physics 398DLP, University of Illinois   ©George Gollin, 2019
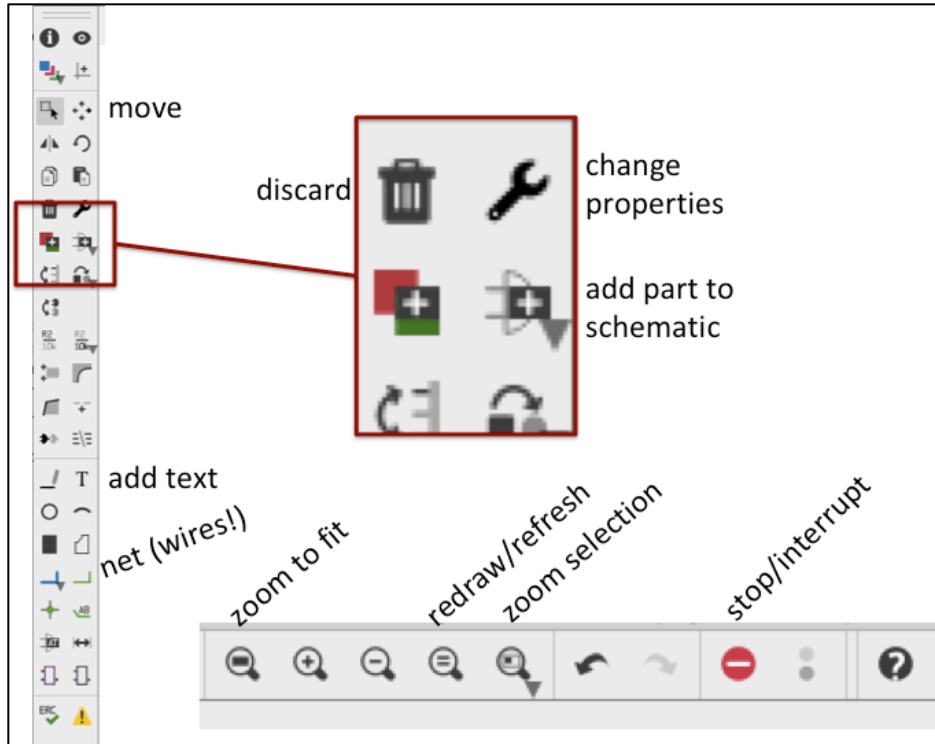
The stop/interrupt button is especially useful: if EAGLE decides to ignore your commands and keeps beeping at you, click on it.

Let's put a frame on the schematic and fill in some of its fields. Click the "add part to schematic" button and search for "A4." Select A4L-LOC and click on the schematic to drop it into place. You'll need to click on the stop/interrupt button to avoid placing a second frame on your schematic.
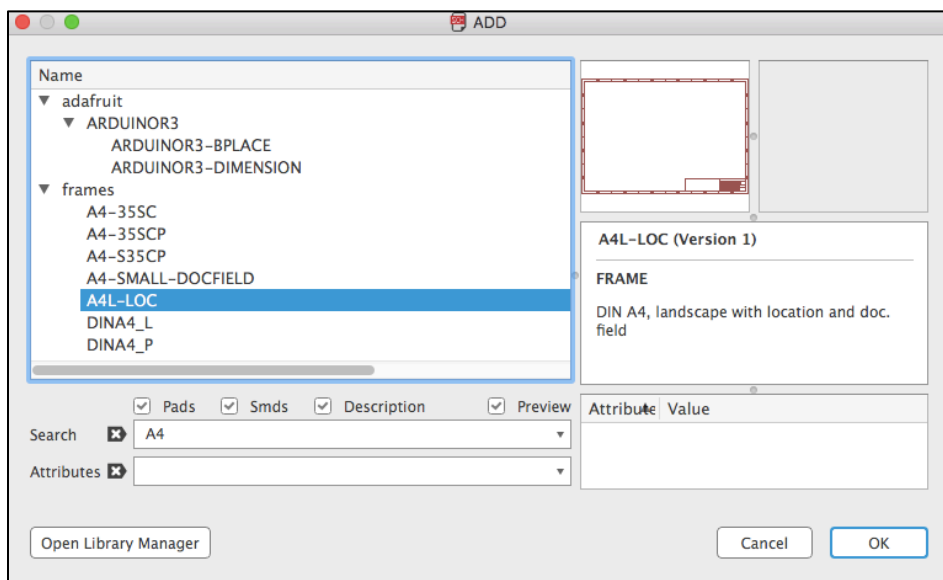
Click on the "zoom selection" button, then drag a rectangle over the information box at the lower right. Now save your schematic, then go to View → Redraw. You'll see that the filename and date appear in the box. If you'd like to add more text (your name, for example), click the "T" tile in the left 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.

George Gollin
University of Illinois
Physics 398DLP

unit02_circuit

7/29/18 10:17:15 AM

Sheet: 1/1

5

6
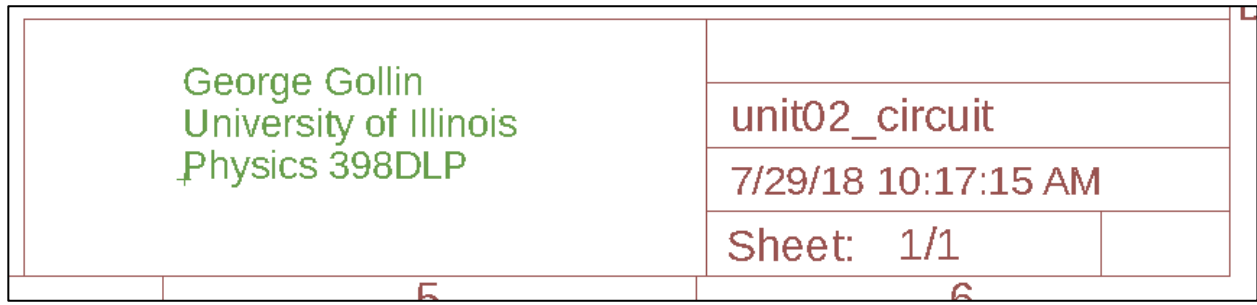
Here's what you would do to make the just-added text larger. Do the following, in sequence:
1. click the change tile (the open-end wrench);
2. select size, then the new font size;
3. left click once on the text's + sign;
4. click the "move" tile (the four-outwards-arrow symbol on the left toolbar)
5. left click once on the text's + sign;
6. move the text to a new position;
7. left click once to drop the text in place;
8. click the stop/interrupt button.
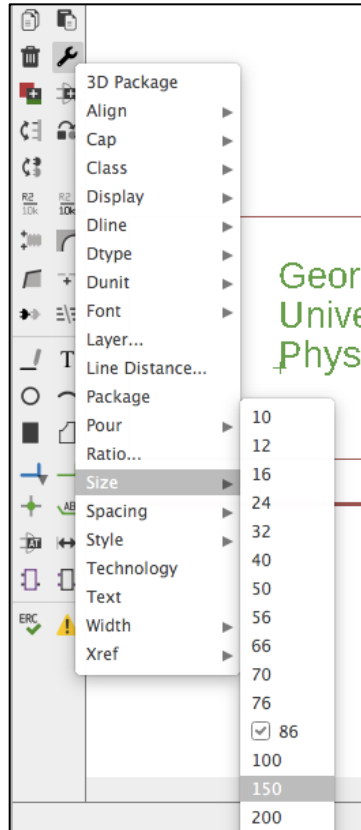
Now put an Arduino Mega 2560 onto your schematic. Please use the P398DLP_ARDUINO_MEGA_2560_FLIPED_DEV version.

Here's what to do if you put two of them on the schematic by mistake. Locate the small (and difficult to see!) + sign on the part you'd like to remove. It's at the bottom left corner of one of the blocks of pins, as shown below.

Week 3 supplemental material   19
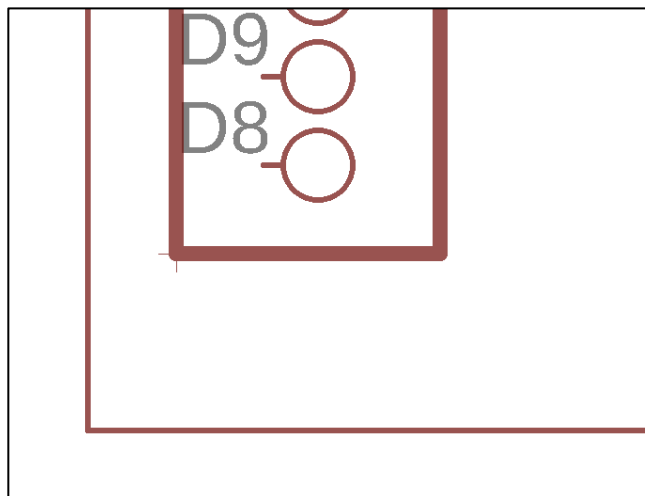
Left click once on the trash can tile on the left toolbar, click once on the + sign, then click once on stop/interrupt.

Now add an ADXL326 and a BME680 to your schematic. Sometimes the search tool for parts is balky, so you may need to help it out: in the Add Parts window, select the p398dlp library. You'll find the devices there.

Your schematic ought to look something like this now:
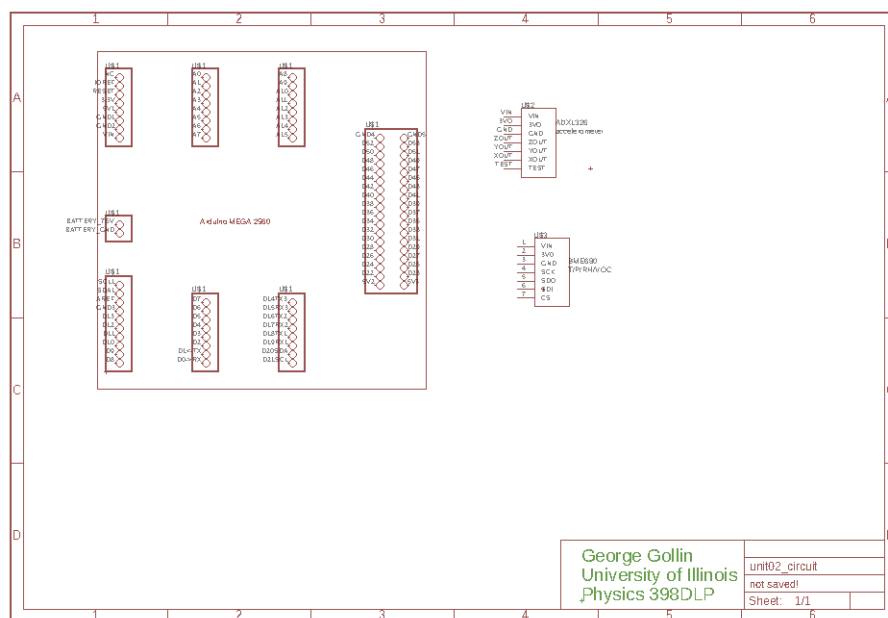
Let's connect the VIN pins on the two sensors to each other, and then to one of the Arduino's 5V lines. Click once on the "net" button on the left toolbar, then click on one of the pins. You can turn corners and connect to other pins by clicking. If you decide you don't like the placement of nets that you've established, click once on the "move" tile, then click once on the part of the net you'd like to move.

Physics 398DLP, University of Illinois   ©George Gollin, 2019
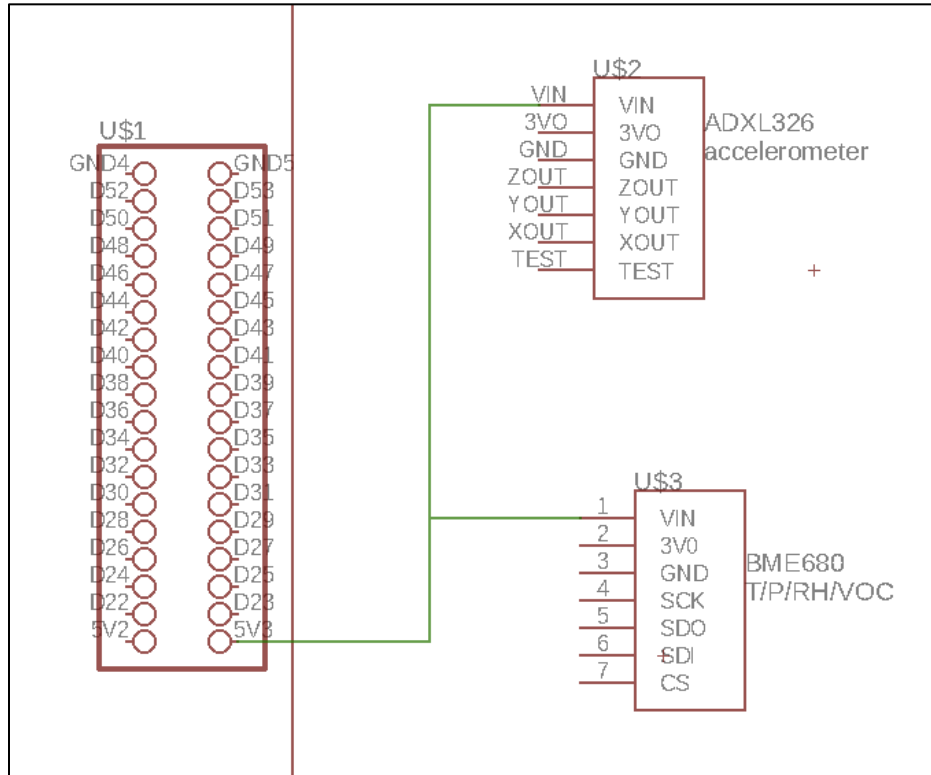
Now right click on the net you've just drawn and select "properties." You'll see something like the following screen shot. Many of the fields can be edited. I'd suggest you change the net name from the uninformative "N$1" to something more descriptive like "5V3." Right click on the net a second time and select Name, then click in the "Place label" box.
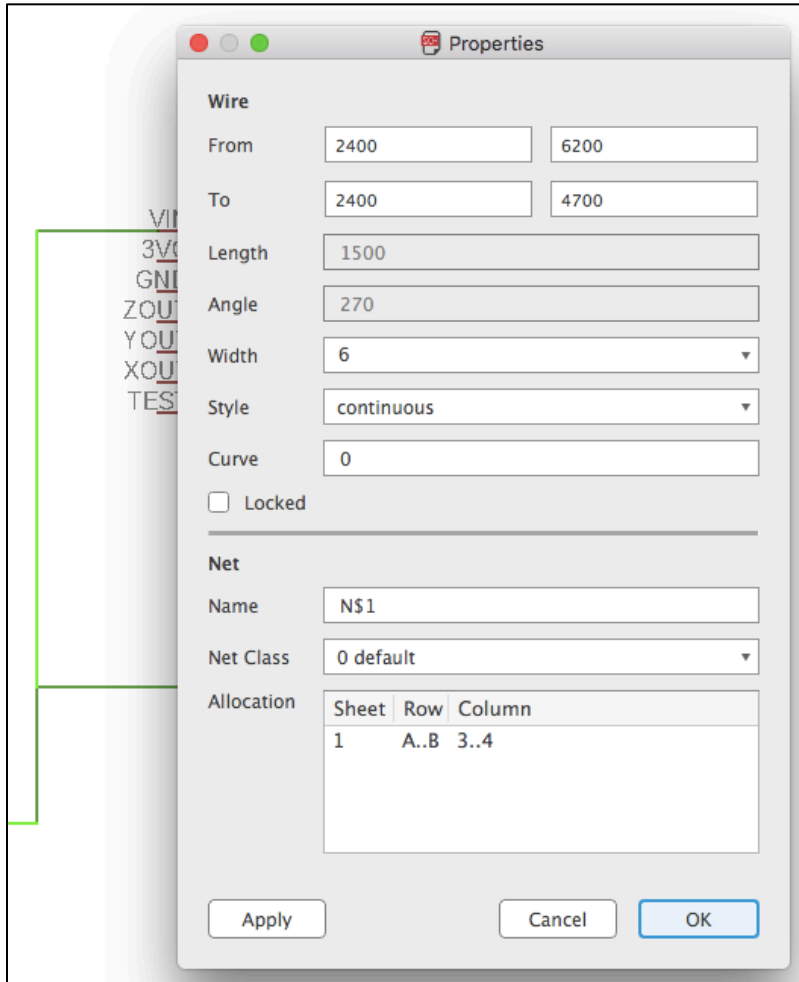
Now connect the GND pins to an Arduino ground. Add a visible label to the net. You may find that it's too crowded to allow a clean positioning of the net name on your schematic. If that happens you can rotate the text this way: click the rotate tile in the left toolbar, then click on the net name's + sign. Click the stop/interrupt button, then click the "move" button to reposition the text.
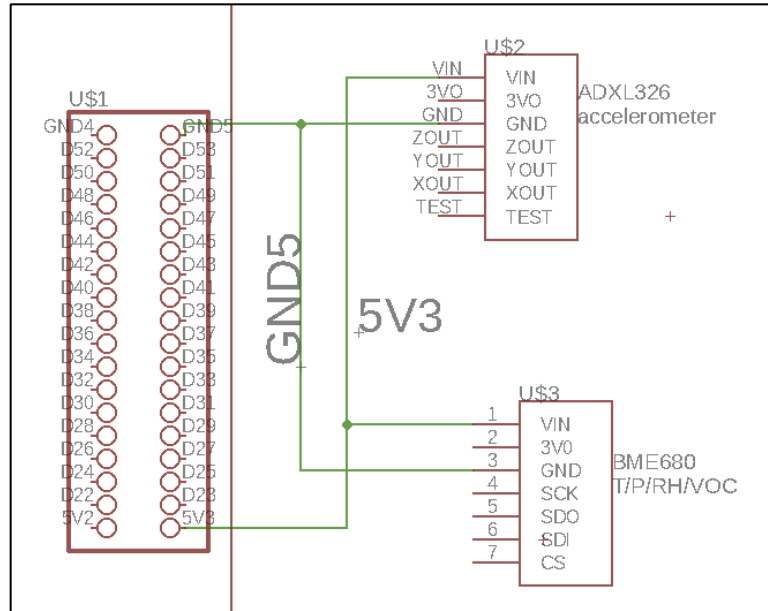
Now connect the rest of the pins as you've done on your breadboard. Naturally, the pins that you've left unconnected on your breadboard circuit will not be attached to any nets on your schematic. Label the nets.

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 a pair of $0.1\mu F$ 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.
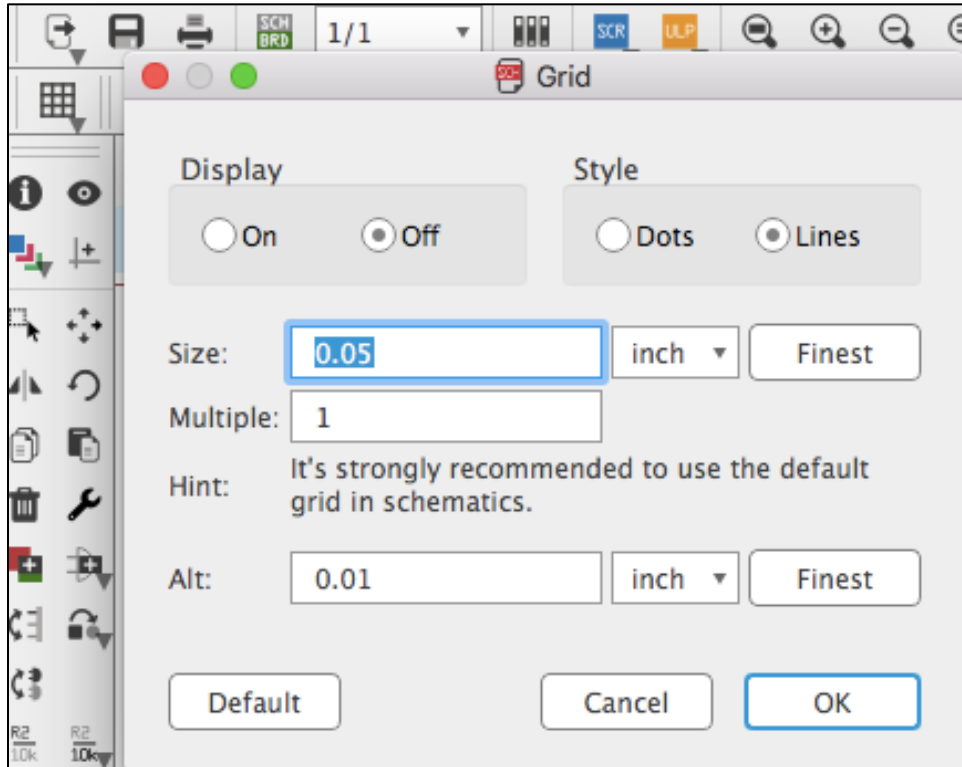
Here's what my schematic looks like:
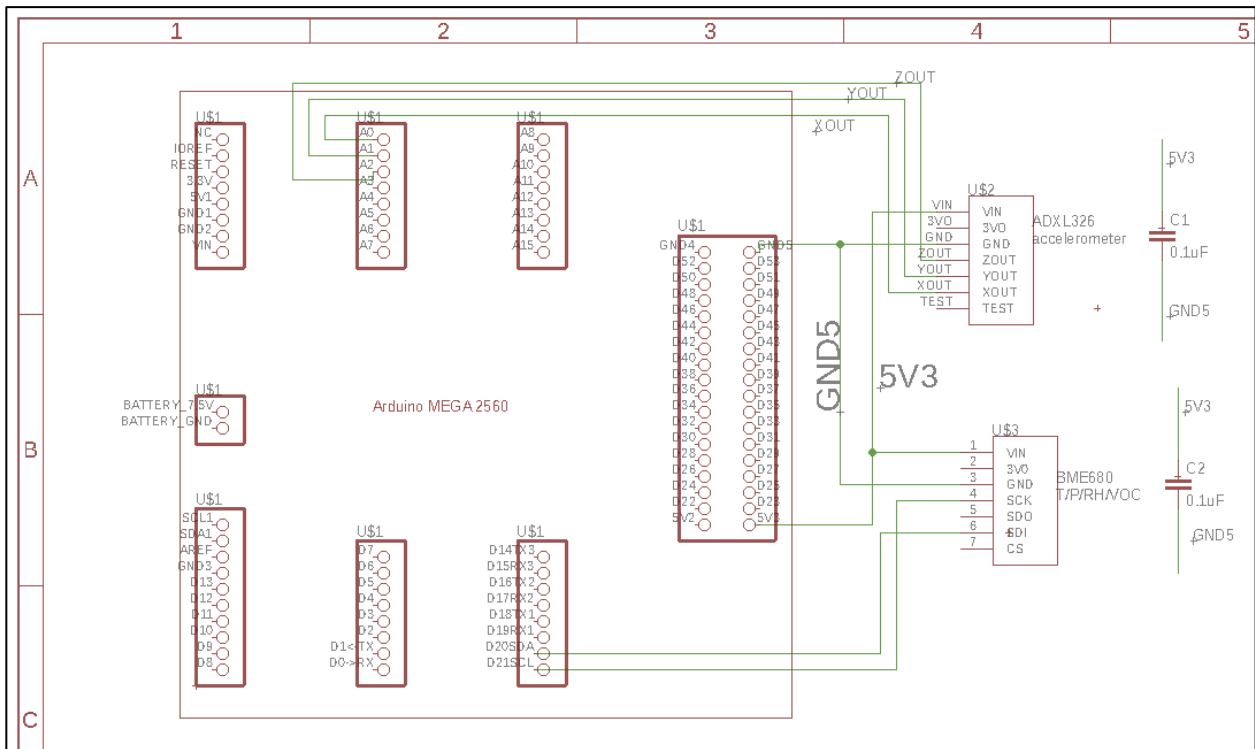
Week 3 supplemental material   24

Physics 398DLP

Design Like a Physicist

Fall 2019

George Gollin

University of Illinois at Urbana-Champaign

Weeks 4 - 14: Away we go!

▷ supplemental material ◁

Goals for weeks 4 – 14…

      …are in the course syllabus. See the "Introduction and Syllabus" document I distributed, or else the syllabus page on the course website.

Physics 398DLP

Design Like a Physicist
Fall 2019

George Gollin
University of Illinois at Urbana-Champaign

C++ and Python Primer/Refresher

C++ and Python Primer/Refresher

C++ and Python Primer/Refresher   4

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 398DLP 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.)

C++ and Python Primer/Refresher   5

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);
}
```
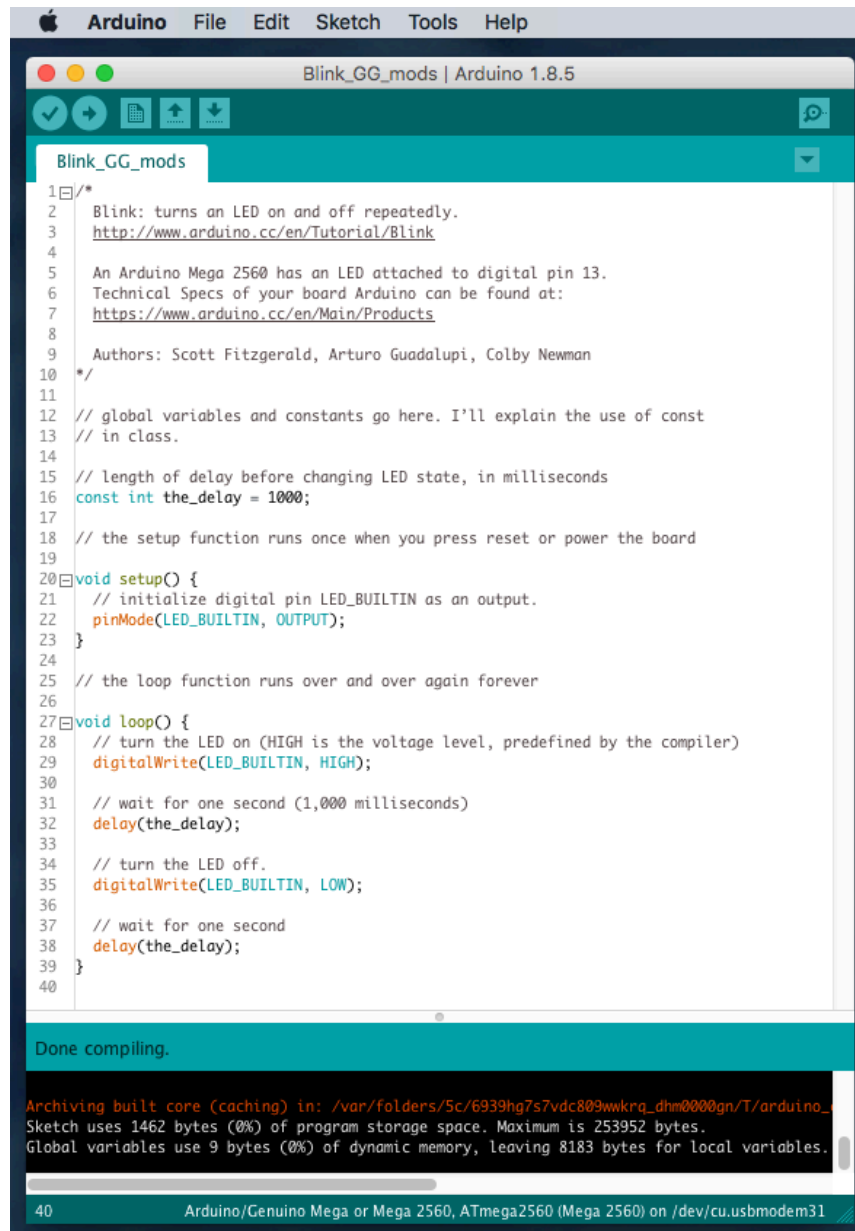
Here's how it looks in the IDE's editor:

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher　　　　6

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

C++ and Python Primer/Refresher   7

might need to access Navigator here: Start → All Programs → Anaconda3 (64-bit) → Anaconda Navigator.
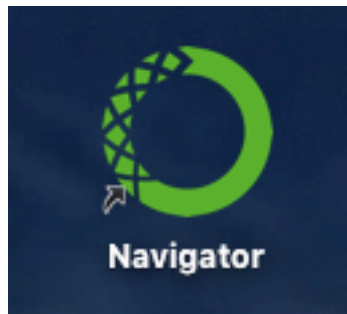
The Anaconda software contains a number of different programs. We will be working with spyder, the "Scientific Python Development Environment." This is an integrated development environment (IDE), which incudes an editor, a control console, a debugger, a table of program variables, and other tools.

Click on the spyder launch button in the navigator window. The development environment workspace will open.
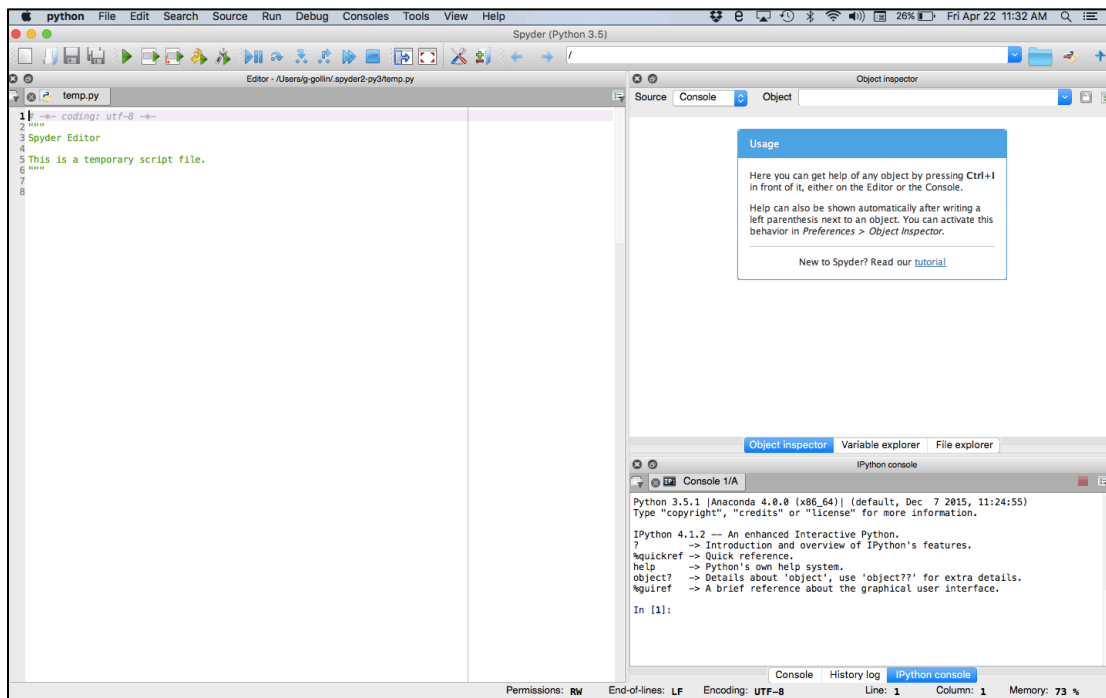
The window on the left is an editor, which you will use to create script files (program files containing executable instructions. The paired triple quotes enclose comments). Here's a screen shot of part of it.

C++ and Python Primer/Refresher   8
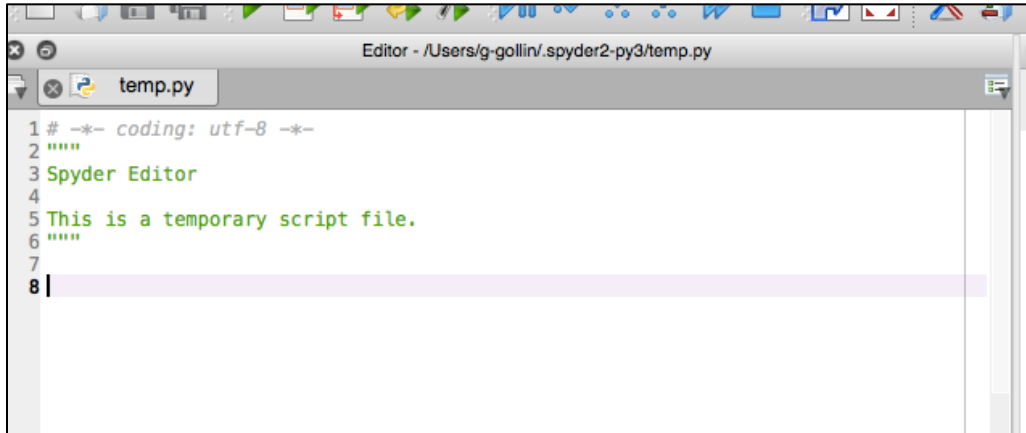
The upper right window allows you to look at the contents of "objects," variables, and file directories. Note the tabs at the bottom of the window for selecting what is shown. You will probably find the file and variable explorer tabs most useful. Sometimes the file modification dates shown by file explorer do not update when I save changes to a file! That is surely a bug.
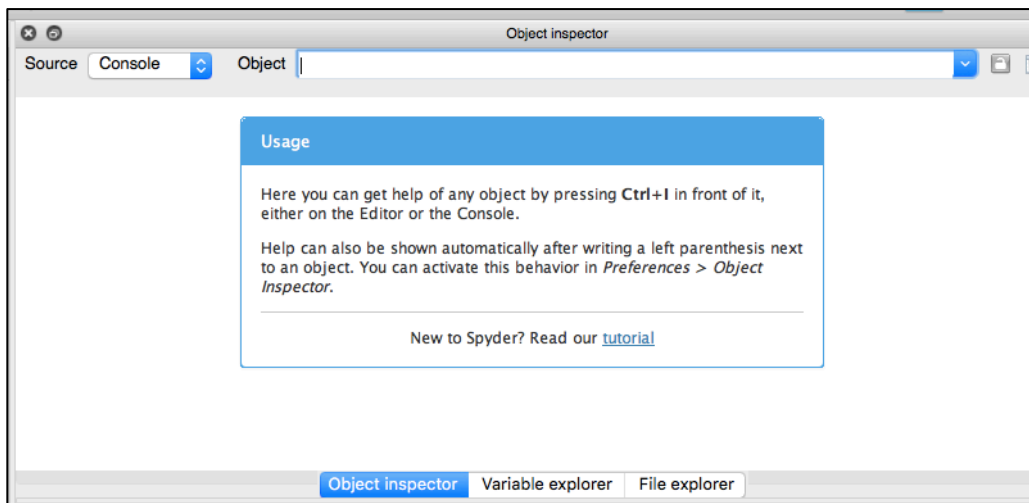
The lower right window shows an iPython console, a sort of operator's station from which you can issue commands to Python. It also displays program output.

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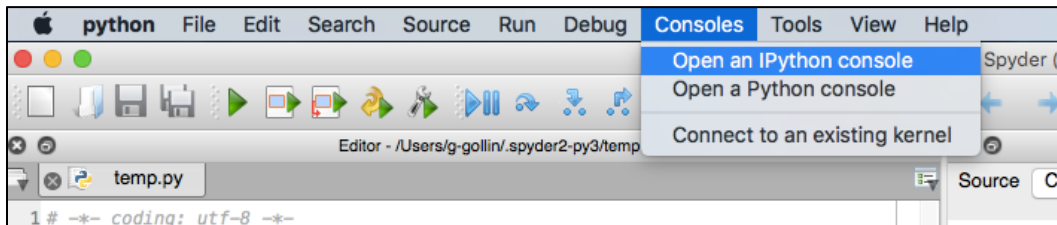
There are a few parameters that you should set. Go to the Python preferences menu and do this:
preferences : run : default working directory
        set to a sensibly-named folder that will hold all your scripts

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

C++ and Python Primer/Refresher   10

uses indentation. The following three code snippets (the first two are C++ for the Arduino, the third is Python) are functionally equivalent.

```cpp
// 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
# 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
# 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:

```cpp
/*
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:

C++ and Python Primer/Refresher   11

```
"""
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.

C++ and Python Primer/Refresher   12

```python
# 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.

C++ and Python Primer/Refresher   13

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…

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   14

```python
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:

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

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

```cpp
#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;
```

C++ and Python Primer/Refresher   16

```
////////////////////// 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);
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

```
      // 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 π;
- 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.

C++ and Python Primer/Refresher   18

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 lines or separate them with semicolons. Use \ to continue to next line.	def QuadraticFormula(a, b, c):
Assignment statements and variable types	root1 = (-b + (b**2 - 4 * a * c) ** 0.5) / (2 * a)
a = 1; b = 1.2; pi1 = 0.031416e2; pi2 = 31415.9265e-4	root2 = (-b - (b**2 - 4 * a * c) ** 0.5) / (2 * a)
SqrtMinusOne = 1j # this one is complex	return [root1, root2]
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.

C++ and Python Primer/Refresher   19

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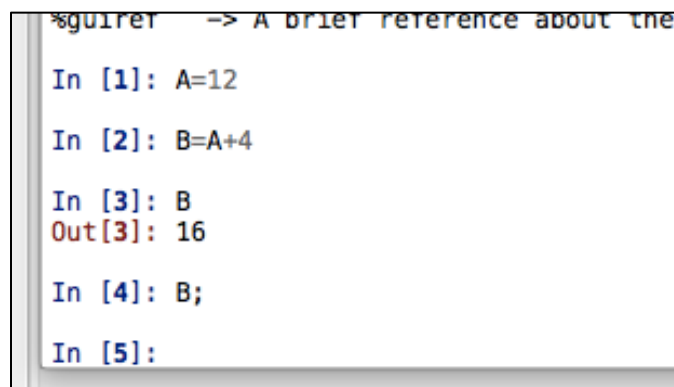
To inspect the value of **B** you would just type its name into the console:

```
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]:
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   20

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 $1 + 2i$. ($i = \sqrt{-1}$.) 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

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   21

```
        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 nth 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
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

```
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)
```

C++ and Python Primer/Refresher   23

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.

.

Physics 398DLP, University of Illinois   ©George Gollin, 2019

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.

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!

C++ and Python Primer/Refresher   25

```
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: $a_0, a_1, a_2, \ldots$ 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.
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

```
        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.

C++ and Python Primer/Refresher   27

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."
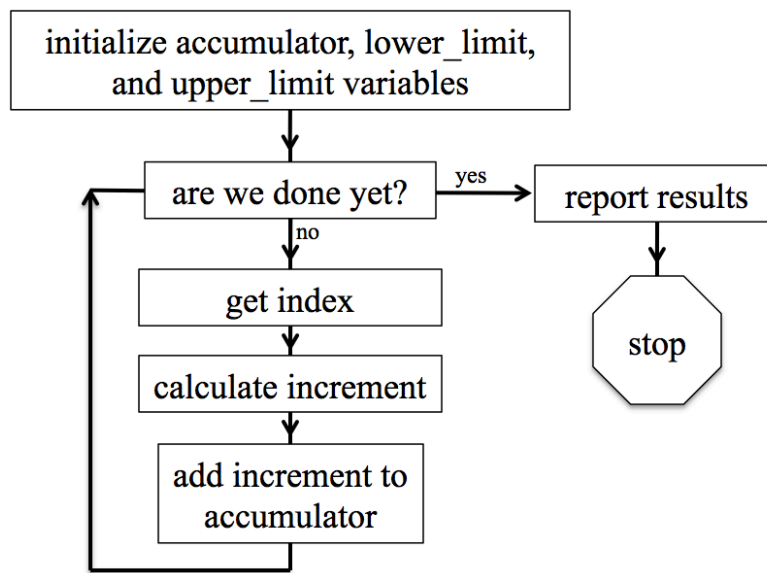
A loop to calculate the sum of a few squares

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,
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   28

```python
    # 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

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

C++ and Python Primer/Refresher   29

```
        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:
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   30

```
        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,

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

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

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \left(1 - \frac{1}{3}\right) + \left(\frac{1}{5} - \frac{1}{7}\right) + \left(\frac{1}{9} - \frac{1}{11}\right) + \cdots$$

$$= \frac{3-1}{3 \cdot 1} + \frac{7-5}{7 \cdot 5} + \frac{11-9}{11 \cdot 9} + \cdots$$

$$= \frac{2}{3} + \frac{2}{35} + \frac{2}{99} + \cdots$$

$$= 2 \cdot \sum_{n=0}^{\infty} \left[\frac{1}{(4n+3)(4n+1)} \right].$$

The value of π is 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 π 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*[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…

C++ and Python Primer/Refresher   32
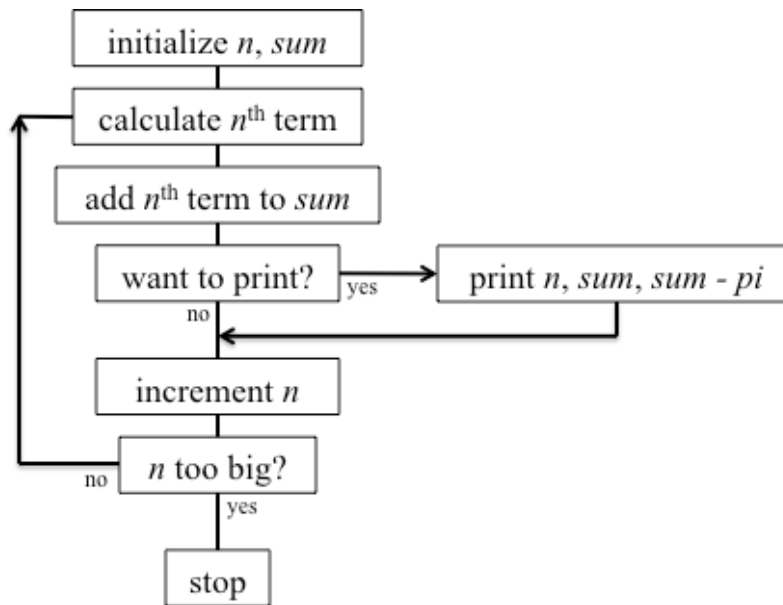
…and here's a listing of a template you could start with.

```
"""
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
################################
```

C++ and Python Primer/Refresher   33

```
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

C++ and Python Primer/Refresher   34

```
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 π:

```
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

C++ and Python Primer/Refresher   35

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.

```python
# 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)
```

Physics 398DLP, University of Illinois   ©George Gollin, 2019

C++ and Python Primer/Refresher   36

```
# 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:
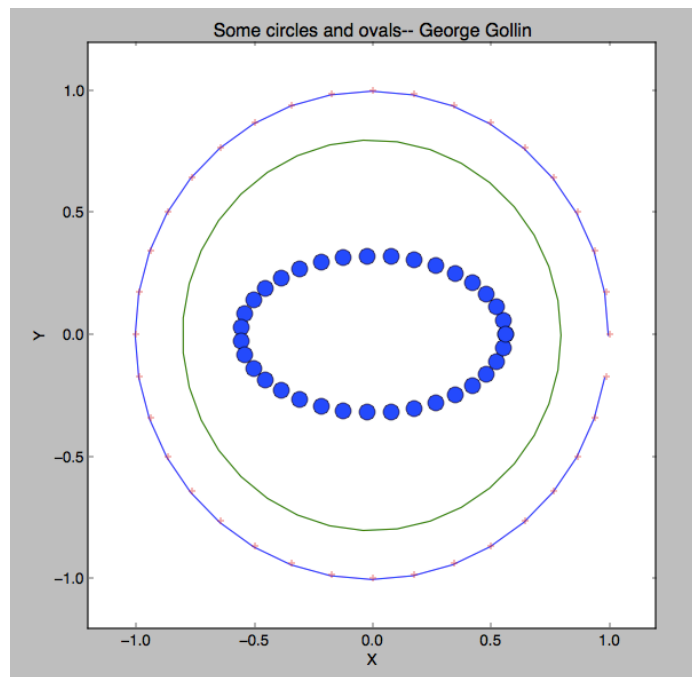
Drawing figures in new windows

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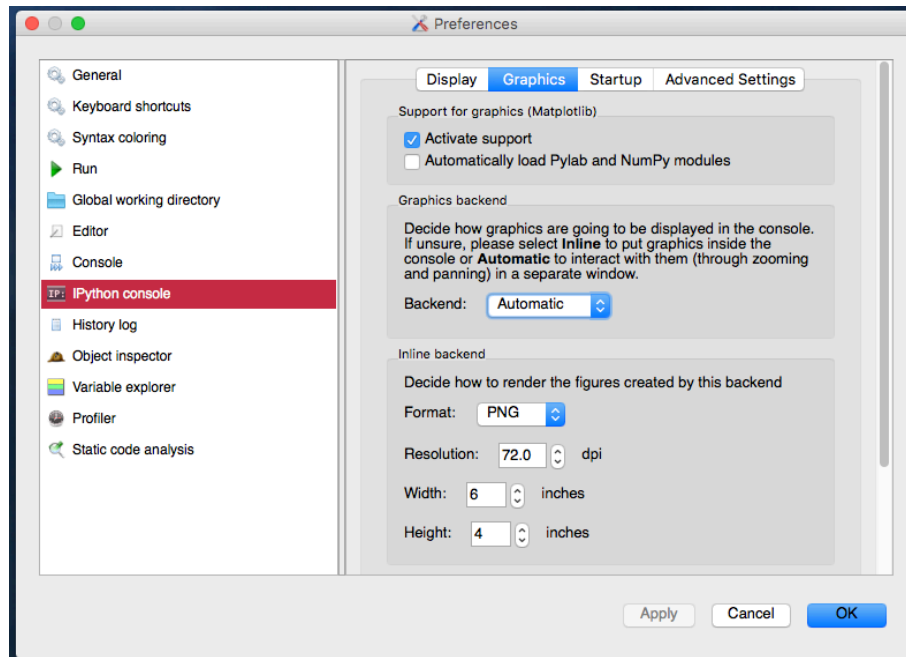
Once you've done this, each figure should open in a new window.

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

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

(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

C++ and Python Primer/Refresher   38

massive object. And the "magnification" is instead a measure of the discrepancy between 2π 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:
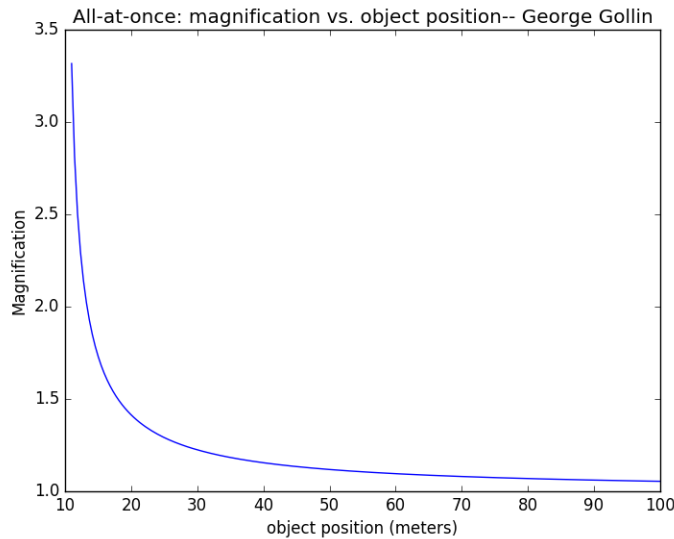
Graphical representations of three dimensions

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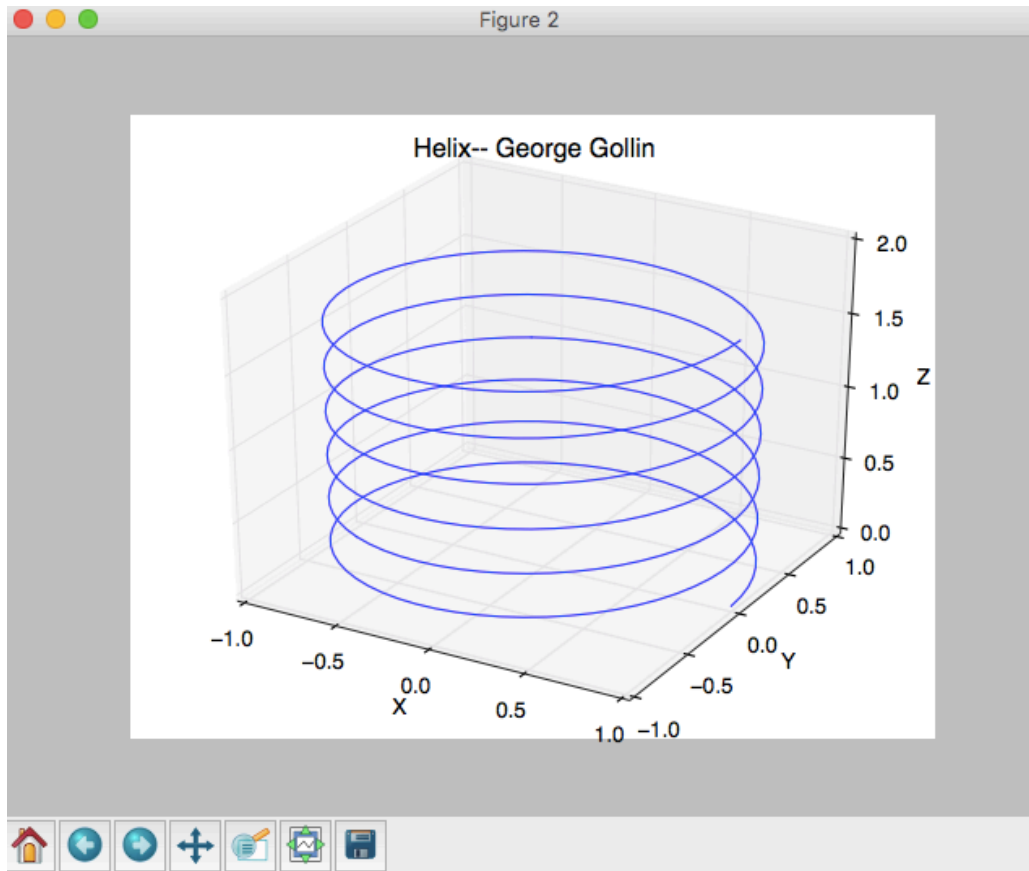
The result follows. Take note of the pan/scroll/rotate button (the cross made of double-headed arrows): it allows you to rotate a 3D figure to view it from different angles. Try this out.

C++ and Python Primer/Refresher   40

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 $k_2 > k_1$.
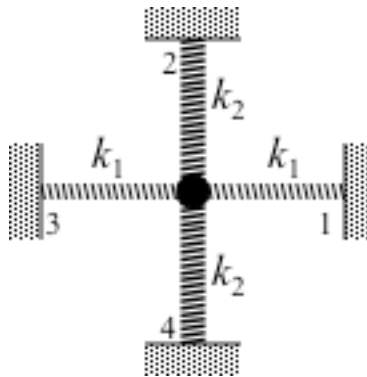
If the mass is displaced from equilibrium to the point (x, y) (with displacement that is very small compared to the springs' lengths L), the system's potential energy will increase by

C++ and Python Primer/Refresher   41

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

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 $61 \times 61 = 3,721$ 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.

```python
# 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.

Potential energy surface-- George Gollin

C++ and Python Primer/Refresher   44
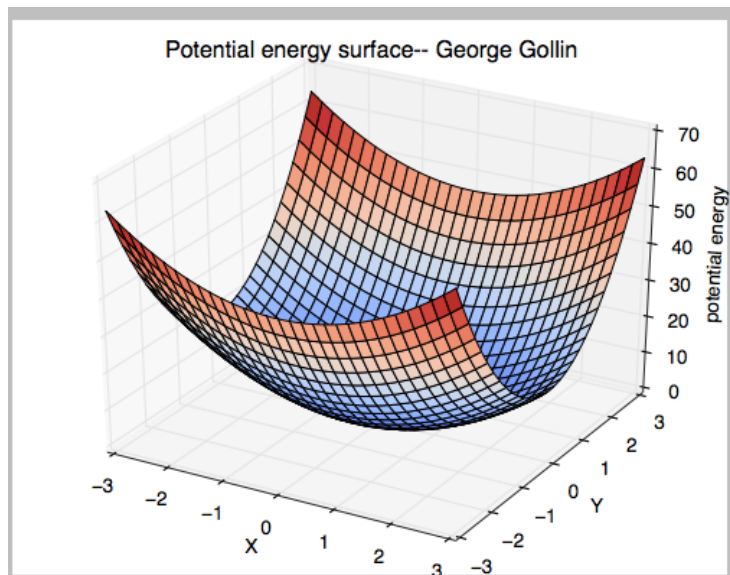
*Drawing a surface using polar coordinates*

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.
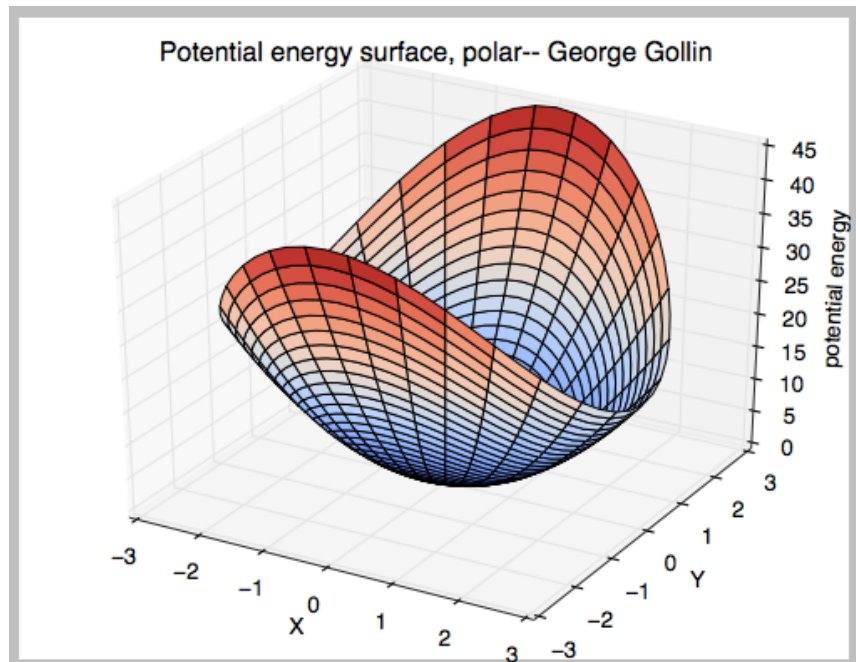
A final exercise: graphing a surface you'll see in General Relativity

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, θ are the familiar polar coordinates. (Note that z is actually independent of θ.)

C++ and Python Primer/Refresher   45

$$z(r,\theta) = 2r_S \left\{ \sqrt{\frac{r}{r_S} - 1} - \sqrt{\frac{r_0}{r_S} - 1} \right\}.$$

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)
```