Session 1 activities worksheet

Due: January 12 *Regrades until:* February 9

Online handouts: Recommended C++ options; "Useful Unix Commands"; GSL intro

Your goals for today (note: the goals are usually over-ambitious!):

- Get used to the GNU/Linux environment
- Download, unzip, compile and run some simple C++ programs, then modify and repeat
- Try out Python
- Understand how underflow, overflow, and machine precision limits can be determined "experimentally"
- Try out a makefile: using g++ compiler warnings
- Try out a program that uses a function from the Gnu Scientific Library (GSL)

Please work in pairs (or slightly bigger groups) with comparable computing and physics experience. The instructors will bounce around the room and answer questions (ask about anything!). *Fill out and submit this document at the latest on its due date (it is fine to submit it before the due date as well).*

A) GNU-Unix Environment

1. Start an Interactive Session at the Ohio Supercomputer Center per the instructions on Carmen.
2. Start a terminal window in your interactive session.
3. Make a directory for the class in your home directory (e.g. `mkdir phy5810`)
4. Change into that directory (using `cd phy5810`)
5. Copy the session file via `cp /fs/ess/PAS2137/PHYSICS5810_OSU/sessions/session01.zip .` (Don't forget the "dot" at the end separated from the "session01.zip" by a space - that dot denotes the current directory and is the destination you want to copy the session01.zip file to!)
6. Unzip the session file via `unzip session01.zip`.
7. Change into the directory containing the session files (using `cd session_01`)

B) A Simple C++ Program (look at the file first in an editor!)

1. *Make a note below of anything you don't understand about the code for the simple program area.cpp.* (If you are new to C++ or just rusty, you might not understand much; see the discussion in the background notes for some details.) Editor: use gedit or nedit or emacs or vi or ... (I personally prefer emacs but suggest using gedit if you don't know any of these) to look at area.cpp. E.g., type gedit area.cpp &. (The "&" runs the editor in the "background" so the terminal is still available.) *Why use "area" and "radius" for variables rather than "A" and "R", which are quicker to type?*

→ We use “area” and “radius” for variables because they’re easier to understand for ourselves and for others who may read our code. In fact, with a fast enough typing speed, it may indeed be faster to type “area” than “A”, as the latter requires holding the shift key.

2. Compile and link area.cpp to create area.x using `g++ -o area.x area.cpp` (the executable would be called a.out if you left out "-o area.x"). Run it by typing "area.x" (or "./area.x" if you get an error message about not finding the file; why does this work?). Add "<< endl" to the end of the output line (starting "cout ...") and recompile and rerun. *What did this do?*

   → This added a new line after printing the result, meaning the terminal shell prompt moves to the next line instead of being printed on the same line as the output from the program.

3. To get used to error messages, modify the program to create some errors and see what the compiler says (try these then invent your own):
   a. remove a semi-colon (leaving out a ; is the most common error you will make!)
   b. use // to comment out using namespace std; [fix this by adding std::]
   c. remove a } (or change it from "}" to ")" or "]")

   Each time, try compiling again. The compiler should give an error message with the line number it thinks is a problem and an explanation (which is often not helpful!). Fix the error and make sure the program still works. Note that "not declared" errors are reported as being at different lines than where the error is made. *Why aren't they noticed immediately?*

   → The not declared errors may be in different locations depending on the cascading effect of previous errors. For example, if you forget the semicolon when inputting the radius at “cin >> radius”, then the error may also be related to the declaration of the area variable. Thus, the declaration error only shows up when trying to print the area later, instead of where the variable is declared.

4. *List two ways to* **verify** *that the program is giving correct answers using known special cases and scaling arguments (e.g., multiply radius by factors of 10):*

   → One way is to try with special cases, like using r=0, r=1, r=2, where the returned values are typically well-known and easily findable. Another way is by using scaling arguments. For example, you can use r=1 and then use r=10 and ensure the value is 100 times the previous one. You can go further and keep doing this to ensure correctness.

   Later (in PS#1) you will be asked to carry out some of the "to do" list in the comments (with assistance, as needed). *Which, if any, of them do you know how to do now?*

   → In C++, I believe I know how to output with higher precision, using a predefined value of pi, splitting into a function, output to a file, add checks to input, and how to use a class.

5. Copy, rename, and modify area.cpp to make a program volume.cpp, which calculates the volume of a sphere using $V = (4/3)\pi r^3$. Test it. *[Warning: Many people get this wrong the first time. Test carefully!]*
6. Let's try Python! Look at area0.py in the editor. Type `module load python/3.6-conda5.2` (you have to do this for this class once in every terminal window you want to run python scripts in) and then run the area0.py script by typing `python3 area0.py`. Compare with the C++ version. *Do they have the same pseudocode? List three differences between the C++ and Python implementations.*

   → There are three differences between the implementations. The first is that the Python file explicitly casts the input into a float, whereas the static-typed C++ doesn't need that casting. The second is that Python uses an operator to square "**" instead of just multiplying by the radius twice. The third difference is that the output is printed twice in the Python file, to show the ability to use single and double quotes. The pseudocode is effectively the same, read the radius, calculate the area of the circle, and then print the area, except that the Python one prints twice.

C) Overflows, Underflows, and Machine Precision

Refer to the background notes on number representation as you do this section.

1. Look at the file for the code "flows.cpp". *Where does the output go?* A single-precision number in C++ is a "float" and a double-precision number in C++ is a "double". You are to empirically determine (in base 10, not powers of 2) *(put your answers in the blank spaces and compare to the notes)*

   → The output goes to the screen and to a file.

   1. where underflow and overflow occur for single-precision floating-point numbers;

      → The underflow is around 1.4013e-45, and the overflow 1.70141e38.

   2. where underflow and overflow occur for double-precision floating-point numbers.

      → The underflow is around 4.94066e-324, and the overflow 8.98847e307.

2. Now for the machine precision, which is calculated crudely in "precision.cpp". *[Note: This code has an intentional bug. Compare to flows.cpp to track it down.]* Determine empirically (in base 10)
   1. the machine precision for single-precision floating-point numbers;

      → The precision appears to be around 6.276e-08.

2. the machine precision for double-precision floating-point numbers. *[Note: you have to change the code for this to work correctly. Hint: the correct answer is smaller than 1e-12.]*

   → The precision appears to be around 1.16e-16.

Explain briefly what the machine precision is and how it is found (note the output file):

→ Machine precision is the value such that when you add that number to the value 1.0, it returns a value greater than 1.0 in the computer's internal arithmetic. Values too small will still be stored as 1.0 if smaller than machine precision. It can be found by gradually reducing a variable epsilon and adding it to a variable that holds a value 1.0, and then outputting it to see if the program views that number as 1.0 or some number >1.0.

D) Using a Makefile

Look at the makefile "make_area" in an editor. Makefiles are useful tools to organize the code for computational physics projects. They contain instructions for how to build an executable (what files have the code, what options to use in compiling, what libraries to link to). Here we have only one source (.cpp) file, but the makefile keeps track of all the g++ options.

1. Look at make_area in an editor. We'll go through the details of a makefile later; for now note the following:
   - Comments start with "#" and continuation lines are indicated by "\" (so "SRCS= \" and "area.cpp" is all on one line). [Warning: There can't be any spaces after the "\".]
   - The list of options defined by "CWARNS" are options to g++ (see C++ options handout for details).
2. Run the makefile using "`make -f make_area`". It will try to compile area.cpp and then link to create area.x. *Did it work?*
   → Yes, it worked.
3. Make a copy of make_area ("`cp make_area make_flows`") and edit it so that it compiles and links flows.cpp. [Follow the instructions in the makefile.] *Success?*

   → Success.

E) Using the GSL Scientific Library

A link to the full GSL documentation can be found on Carmen. Here we look at our first example program, which has been adapted from the documentation.

1. Examine the file "J0_test.cpp" in an editor. This test program calculates the cylindrical Bessel function $J_0(x)$ at x=5 using the GSL library function "gsl_sf_bessel_J0" (you would find out the name of the function by looking at the web page with the GSL reference manual). *Look up*

"Bessel function Wikipedia" using Google. How can you use a graph on the webpage to check you are calculating the right thing?

→ We can use the graph on the webpage to check if we are calculating the right thing. There is a graph of $J_0(x)$, and we can check if it's correct by comparing our returned value at x=5 with x=5 on the graph.

2. Compile J0_test.cpp *according to the directions in the file* and verify that the correct answer is given. *Is it?*
   → The answer does look close to the answer provided in the graph on Wikipedia, so it is likely correct.
3. Now calculate for x=3 (modify the code for this value or add code to read in any x). *Answer:*

   →-0.260051954901933446.

4. Verify your answer using Mathematica or MATLAB or Python. If Mathematica, under the Help menu, start the Help Browser, choose "Master Index", and look under Bessel function). If MATLAB, under the Help menu, select "MATLAB Help", choose "Search Results", and search for "Bessel Functions".
   Did you succeed?

   → Yes, Mathematica verifies the same result (to lesser precision).

5. *Discuss with your partner how a more general program could be structured, and make at least two suggestions here.*

   → You could make a more general program in two ways. The first would be to allow inputs, so you don't need to make changes or recompile the code to test different values of x. Another way would be to allow for another input to allow for different Bessel functions than just J_0.

F) Jupyter notebooks

1. Start a jupyter notebook on OSC per the instructions on Carmen
2. Double click on "materials" and then on "session_01"
3. Double click on J0_test.ipynb
4. Run through the notebook by clicking on every code box and hitting Shift-Enter

---

G) Some More Python (if time permits)

1. Remember `module load python/3.6-conda5.2` before running python in a terminal window (once per terminal window)!
2. Create volume0.py.
3. Look at and run the Python versions of the flows and precision codes (included in session01.zip).

4. Try running Python interactively. At the prompt, just type `python3` and return. You will get a >>> prompt. Try import area1 and then help(area1). Use quit() to exit.
5. Tkinter demo: Look at hello_world_gui.py and run it. Try to change the text and font.

H) Sharing Your Session

You are going to submit your codes at the end of each session through Carmen.

1. Go to the directory above session_01 using "`cd ..`"
2. Create an archive of all you did during the session using "`zip -r session01_work.zip session_01`
3. Upload the session01_work.zip file in the assignment on Carmen after starting a web browser in the Interactive Session

---

I) Reflection

Write down which of the six learning goals on the syllabus this worksheet addressed for you and how:

→ I now know a bit about some of the tools of computational physics through the usage of Tkinter and the GSL scientific library (for using Bessel's functions). I also understand the limitations to computational solutions a bit by understanding factors like machine precision and overflow/underflow. I also saw examples of how to write clear, well-documented code by examining all the C++/Python files in this session.