1. 6810 Session 1

a. Background to 6810 Computational Physics

This Computational Physics course originated over a decade ago from two motivations: first, I found that my graduate students knew how to program (more or less) but knew little about computational physics and I observed that other graduate students were in the same situation, and second, there has been a long-term need for both undergraduate and graduate computational physics and someone had to dive in and start developing the curriculum. Over the years the need for such a course (and, to be honest, additional more advanced courses) has only increased as almost every facet of physics has become more computationally oriented. The course continues to evolve significantly, both in content and structure, which means that feedback from the participants is important.

A computational physics course could be:

- a physics course using the computer as a tool;
- a numerical algorithms course;
- a simulations course;
- a programming course;
- a guide to using environments such as MATLAB or Mathematica.

The plan for 6810 is to have aspects of *all* these. This means we'll sacrifice depth in favor of exposure to the most important aspects using prototypical examples; in doing so you'll get practice in learning more on your own. Indeed, being able to learn new computational methods and applications on your own is probably the most important skill to develop! Another way to run such a course is to have a single but multi-faceted project covering the entire quarter (this has been used with success elsewhere in *advanced* courses). Here we'll do many mini-projects in class and everyone will do a "project" of their own.

Based on past experience, the class backgrounds in both physics and programming will cover a wide range. We will have tutorial handouts as needed, but mostly learn as we go, a bit at a time, never knowing the complete picture until later. Thus, it is patterned after physics *research* rather than textbook physics. There are no definite prerequisites for the course besides some basic physics and mathematics. The course is self-paced for each session, so those who know more coming in (physics or programming) should find sufficiently challenging tasks (if not, let me know and we'll find you some!). There will be something for everyone.

The early versions of the course followed *Computational Physics* by Landau and Paez [1], covering topics from roughly half the chapters. An updated version of this bood using Python (with additional author Cristian Bordeianu) is available as an e-book; see the 6810 course information webpage. It used a hands-on, project-oriented approach, with breadth instead of depth, and special emphasis on error analysis and numerical libraries. There has been no required text in recent years, but we have used Morten Hjorth-Jensen's *Computational Physics* lecture notes [2] as a companion guide; these notes have been developed for a similar although more extensive set of courses taught

at the University of Oslo in Norway, with programs in C++, Fortran 2008, and Python. There is a version from 2015 that we'll refer to regularly. This year we'll continue to build on both of these references (and others) with our own notes. A useful supplementary reference for background on numerical algorithms is *Numerical Recipes* [3], for which an older version is freely available a chapter at a time online (see 6810 info web page).

More generally, Google (or Bing or ...) is a good way to find information on programming (e.g., try "double precision C++" and you'll get pointed to useful links at cplusplus.com) or special functions and algorithms (e.g., try "Bessel function" and you'll get useful links at Wolfram Mathworld or Wikipedia, which are generally trustworthy on such topics). In many cases, the most effective debugging procedure is to cut-and-paste the error message from compiling or running your code directly into a Google search. This observation has led to an online joke that "the discipline of Computer Programming to be officially renamed to 'Googling Stackoverflow'." (See this link for the full joke. The Stack Overlow website `stackoverflow.com` is one of the most reliable sources of information for debugging.) Take this as a reminder that we don't want to *blindly* use what we read online, but use it as a foundation for greater understanding (although in some cases, we just need the bug fix :).

The class has also evolved so that there is only a rather minimal overview at the beginning of each session by the instructor. Feedback from past years indicated that the limited in-class time is best spent with hands-on work, with the "lecture" material provided in advance in the form of background notes. (In modern educational parlance this is known as having a "flipped" classroom.) These notes will generally be made available at least the day before a session. *It is important that you read them before the session.* Grades will be based on the in-class session sheets, several problem sets, which will be follow-up activities to what is done in class, and a project due at the end of the quarter (see the course info page for grading percentages). You be evaluated on effort and accomplishment, but scaled to your background. The accumulated experience is that if you try diligently and do all of the assigned work, you *will* learn computational physics!

Everyone will be required to do a project based on a physics problem of interest to them. It could be part of undergraduate or PhD thesis research or simply a topic chosen from one of the references or class sessions. There are many examples from earlier instances of 6810 (a list will be available on the course web page) and in exercises in the Hjorth-Jensen notes and the Landau/Paez book. The pedagogical goal is to have you, on your own, combine the tools and techniques used in class. The project must involve some visualization (i.e., plotting) and an analysis of correctness (How do you know the numerical calculations are correct? How accurate are they?). The scope and complexity of your project will scale according to your background and can use almost any relevant programming language (that is, C++ is not required; past projects have used such diverse languages as R and LabVIEW). Projects will be formulated in consultation with the instructor, but on the initiative of the student. Don't worry if you have no ideas at this point in time; everyone finds a project eventually and we are rarely disappointed.

Finally, here are some details about the current instructor's (Dick Furnstahl) background. I have many years of experience writing computer programs; my first program was written over 45 years ago! However, long experience does not automatically translate into updated expertise on

numerical algorithms or good programming practices (for the latter, it is often the opposite if the practitioner is stuck in obsolete ways of programming). But I hope that along the way I've picked up some good habits and updated my viewpoint appropriately. If not, you are invited to challenge me ("question authority!").

I am a nuclear theorist who studies low-energy quantum chromodynamics (QCD) and the nuclear many-body problem. My particular focus these days is on so-called "effective field theory" and Renormalization Group methods, and lately I've been applying Bayesian statistical techniques. The theoretical tools I use day-to-day require primarily what I would call "medium-scale" rather than "large-scale" computation. That means that I seldom use supercomputers myself but instead use small-scale parallel processing ($\lesssim 50$ cores) or run my programs for hours or less on a fast PC. However, for the last ten years I've been part of SciDAC collaborations (first UNEDF [`unedf.org`], now NUCLEI [`computingnuclei.org`]) that use petascale computing (and work twoard exascale), and members of my group have been among the large users of the Ohio Supercomputer Center (OSC). My programming is primarily done these days in C/C++, Fortran-95, Python, Perl, MATLAB, and Mathematica on Linux systems after many years of Fortran programming. I've also programmed at times in PHP, Java, Pascal, Modula-2, Lisp, Maple, Macsyma, Basic, Cobol (!), and a number of languages now long dead (e.g., Focal, Telecomp). I'm quite enthusiastic about Python as a language for at least some aspects of computational physics.

b. Computing Environment

A general theme in the course is to use basic and portable tools, which is why we concentrate on the GNU compilers and programming utilities. You can periodically find religious wars on Slashdot about using GUI (graphical user interface) menu-based tools versus a command-line interface (CLI). For us, this is both a pedagogical and practical decision. Using the command line lets us look inside the black box and gives us more flexibility, while also preparing you for using remote machines (e.g., supercomputers at OSC); it is easy to later switch to IDE's (integrated development environments) like Eclipse, if desired. As for operating systems, we note that Linux is the choice of 99.6% (!) of the Top 500 Supercomputers (according to the November 2016 list) and the vast majority of smaller clusters.

The main programming language for the course will be C++ and the main compiler we'll use is `g++`. We'll also look at Python, although mostly as a scripting language. We will use the GNU/Linux environment either on Windows using Cygwin (or by logging onto a public Linux machine via Xwin32 or equivalent) or directly on Linux (or switch between them; it is always your choice in Smith 1094). *Everyone will need an active account on the Physics Department Windows* and *Linux computers. Please see the computing staff in PRB 1199 ASAP if there is a problem with your account or if you have forgotten your password.* (Note: If you are from outside the Physics Department, you will be given access to the computers by the first class meeting.) You are encouraged to set up your own computer to match the Smith 1094 setups; there will be instructions linked on the 6810 homepage for Macs as well as Windows and Linux PCs. The GSL ("Gnu Scientific Library"), which is written in portable ANSI C, will serve as a (free!) numerical

library (as opposed to the routines from *Numerical Recipes*, which cannot be freely distributed). Plotting will be done primarily in Gnuplot (and sometimes Matlab or Mathematica or Python or xmgrace or Veusz or . . .).

We will not assume that you know how to use these software tools (although you might). The general strategy is to give you a program and/or set of instructions to follow, and have you modify, correct, or extend them. This approach is efficient and realistic but it is very much a "toss them in the water to teach them how to swim" approach. That means that, just like in real physics research, YOU have to figure things out (and be confused much of the time!).

At various points during the quarter, we will look at using the object-oriented features of C++ from a computational physics perspective. For example, we'll look at rewrites of some of our simple codes using C++ classes (including "wrappers" for the GSL routines) and we'll look at classes for complex numbers and linear algebra. Unfortunately, we will not be able to explore this topic in great detail, given our time limitations; however, depending on your feedback I can supplement the official class material with more object oriented programming or advanced C++ features.

c. Some First Comments on Programming

This is not primarily a programming course but we'll talk about programming frequently in bits and pieces. Here we'll make some first comments based on the sample program area.cpp used in Session 1 (a listing is available from the 6810 web page before Session 1 as a pdf file area_listing.pdf). *Please read the first two chapters in the Hjorth-Jensen notes for more generalities on programming and a reminder of (or introduction to) some C++ features.* First, what you need to know (now) about filenames and compiling:

- The convention for C source code files is that they end in ".c", e.g., area.c would be the C version. There is not a fixed convention for C++. In different places, you might find C++ source files could ending in ".C", ".cpp", ".c++", or ".cc". Here we'll use ".cpp".
- You can *compile* the *source code* area.cpp and *link* it to create the executable program "area.x" (using ".x" for executables is our choice!) in one step at the command line (in Cygwin under Windows or in Linux) using the gnu C++ compiler (called `g++`): `g++ -o area.x area.cpp`
  This command combines two steps, which also can be done separately:
  compile: `g++ -c area.cpp` creates the *object* file area.o;
  link: `g++ -o area.x area.o` combines area.o with system libraries to create `area.x`.
  If you use Eclipse (or another IDE), these steps will be carried out through menu commands.
  [Note: the convention is that one-character command line options are preceded with a single hyphen, such as `-c`. In this example, `-c` tells `g++` to *only* compile `area.cpp` while `-o area.x` tells `g++` to name the executable `area.x`.]

There are many possible options for `g++` and when we have many different files (in larger programs), it can be awkward to manage. Also, you may use different options for different C++ compilers (it will be useful to use more than one for tests and optimization). The "make" utility can manage it for us, using "makefiles"; we'll see an example in Session 1. The makefiles will use most of the `g++`

options that warn about *possible* (but not certain) problems. Fixing them before proceeding will catch many potential errors as well as help enforce good programming style.

Some brief comments on good computational programming practices and details of the program listing `area.cpp`:

- Always start with a "pseudocode" for the task (which is independent of the programming language). A pseudocode is an *outline* of what you want to do:

  ```
  read radius
  calculate area of circle
     π = 3.14159
     area = π × radius^2
  print area
  ```

  Don't start coding before planning the structure of your program or subprogram! You will want to break the overall task into smaller subtasks that can be implemented and tested independently. [Note: The above pseudocode is for what is known as "procedural programming". This is the standard for implementing computational algorithms but it is not the best choice in all cases. More on object oriented programming later!]
- When converting pseudocode to code, add the comments *first*; if you say "I'll add them later" you will seldom do it (based on long experience!). YOU MUST USE COMMENTS! Your comments should allow the user (which may be you, after some time has passed) to understand, verify, and extend your code. In the program listings for the course we will often include comments on language details for pedagogical purposes; in actual practice comments would be focused mostly on definitions or algorithmic details.
- Note the elements in the documentation comments at the top of `area.cpp`. There are many possible styles, but the content should include the programmer(s) name(s), contact information, a basic description of what the code does with any relevant references, revision history, and notes. I like to include a "to do" list of planned upgrades. (We'll talk about "revision control systems" later, which can better keep track of updates.)
- In C++, you can use `//` to start a comment (which continues for the rest of the line), or use C-style comments, which means to sandwich the comments between `/*` and `*/` (which can run over any number of lines). Most C++ style guides prefer the first type. I recommend using `//` for documentation and `/*` ⟨*stuff*⟩ `*/` to "comment out" sections of the code when debugging or testing.
- For readability, indent your code according to consistent rules. This is facilitated by a good editor or a utility such as "indent" (this is a command-line program). Note that the C++ compiler doesn't care about the indentation; this is for the human readers! This is *not* true in Python, where formatting is part of the language definition; it will be a *bug* to have inconsistent indentation in Python!
- Use appropriate variable names that are easy to understand (e.g., "radius" and "area"), which makes them self-documenting. In most cases you should also add a comment when the variable is declared, possibly referring to corresponding conventions used in your notes or a paper. Note that an "appropriate" name is also not too long, which hinders readability.

- The "include" files play the same role as packages included in Mathematica (to read in definitions or special functions). One of the most common is "iostream", which has the basic input and output functions (note: it takes the place of stdio.h in C). Note that ".h" is omitted and we need to declare the "namespace" (this is like the context in Mathematica). We'll come back to this later, but note that some programmers avoid the `using namespace std` declaration in favor of prepending "std::" to the relevant function names.
- All variables are *declared* to be an int, float, double, etc., but unlike in C, in C++ you can postpone declarations until the variable is first used (e.g., the way "area" is declared as a double in area.cpp), which is usually preferred (by me, at least).
- A "double" is a double precision floating-point number while a "float" is single precision. Precision refers to the number of digits kept in the computer representation of the number (which in turn depends on how many bytes are allocated for storing the number). More on this below, but keep in mind that the precision is less than ∞!
- If floating-point and integer numbers are mixed in a statement, there are implicit conversion rules defined in C++. *Do not rely on them but "cast" the variables explicitly.* E.g., if `i` is an index variable for a loop defined as an `int` and you are summing their inverses, use
  ```
  sum_inverses += 1./double(i);
  ```
  rather than
  ```
  sum_inverses += 1./i;
  ```
  and *never* use (why not?)
  ```
  sum_inverses += 1/i;
  ```
- If a number is a constant, which means it doesn't change during the running of the program (e.g., pi), declare it with "const". This allows the compiler to check if it changes in the program, which would point to a mistake. [Note to C programmers: Using const is preferable to using `#define` for constants.] *As a general principle, we want the compiler to catch as many potential errors as possible; this is why we will invoke most of the compiler warning options.*
- In general, declare a parameter for *every* constant rather than "hard-wiring" the value (which means to just use the numerical value directly). This improves the understandability of the code and sets you up to generalize it without introducing bugs.
- Unformatted input and output is easy in C++ using the cin and cout functions. Note the direction of the `>>`'s in each case. The cout command doesn't include a newline command at the end of the output line. You can add it with `<< endl`. [Note: if we didn't declare `using namespace std`, we would call these functions `std::cin` and `std::cout`.] You should also be familiar with the `printf` syntax from C (but also available in C++ and used by many other programs).
- There is a function for raising a variable to any power, which is called `pow`. Can you guess why it was not used in `area.cpp`? [Hint: `pow` works for any real exponent, not just integer exponents, and doesn't distinguish the two cases. But does this matter in practice (i.e., does the compiler fix this)?]

d. A First Look at Python

Consider the Python scripts area0.py and area1.py on the area_listing.pdf handout. These are translations of area.cpp to Python. Here are some of the differences:

- Python files end in ".py".
- Python is generally *interpreted* rather than compiled (more on that later!). The Python interpreter can be evoked on area0.py at the command line using: `python area0.py`
- Use `#` instead of `//` for one-line comments, and `"""` to start and end multi-line comments.
- Indentation matters in Python! It is used by the interpreter to determine where blocks end (more later!). No semi-colons to end lines and no curly braces.
- Use `import` to read modules that have Python definitions; e.g., see `import math` in area1.py.
- No variable declarations! Instead they are "dynamically typed" (more later!).
- All Python floating point numbers are double precision (but are called "floats").
- Examples of input and unformatted and formatted output can be found in area0.py and area1.py, respectively.
- The operation x^n is coded as `x**n`.

There are excellent online resources for python. You starting point should be the official website `http://www.python.org`, which has comprehensive documentation on Python. (Warning: Python 2.x and Python 3.x have some significant differences and it would be best to use the 3.x version. We were stuck with 2.6 or 2.7 in class last year, but we should be able to use Python3 this year.)

e. Computer Representation of Floating-Point Numbers (naive)

A classic computer geek T-shirt reads:

> "There are 10 types of people in the world.
> Those who understand binary and those who don't."

We need to be among those who do understand, because the use of a finite-precision binary representation of numbers has important implications for computational programming. If we use N bits (a bit is either 0 or 1) to store an integer, we can only represent 2^N different integers. Because the sign takes up the first bit (in general), we have $N-1$ bits for the absolute value, which is then in the range $[0, 2^{N-1}-1]$. A C++ `int` uses 32 bits = 4 bytes, which means the maximum integer is $2^{31} \approx 2 \times 10^9$ (an `unsigned int` goes up to $2^{32} \approx 4 \times 10^9$). This doesn't seem very large when you consider that the ratio of the size of the universe to the size of a proton is about 10^{41} [1]! [Old but interesting anecdote: Comair had computer problems during Christmas 2004 caused by a computer using 16 bit integers, which limited the number of scheduling changes per month to 2^{15}. That was exceeded because of storms and the whole software system crashed!] You need to be

aware that sometimes it is necessary to use a `long int` rather than an `int` for really large integers, but otherwise you will rarely run into problems with integers.

A unique, well-defined, but in general *approximate* representation (as opposed to the representation for integers) is used for "floating point" numbers. The representation is a form of "normalized scientific notation", where in the decimal version 35.216 can be represented as 0.35216×10^2 or, more generally,

$$x = \pm r \times 10^n \quad \text{with} \quad 1/10 \le r < 1 \ , \tag{1.1}$$

with r called the "mantissa" and n the "exponent" (note that the first digit in r must be nonzero except when $x = 0$). Here is the basic form for the binary equivalent (not unique!):

$$(\text{any number}) = (-1)^{\text{sign}} \times (\text{base 2 mantissa}) \times 2^{[(\text{exponent field})-\text{bias}]} \ , \tag{1.2}$$

and the computer stores the sign, base 2 mantissa, and the exponent field. The *bias* serves to keep the stored exponent positive, so that we don't need to store its sign, saving one bit.

Let's look at an analogous base 10 representation with six digits kept in the mantissa, one digit in the exponent, and a bias of 5:

$$-\frac{4}{3} = -1.33\overline{3} \doteq (-1)^1 \times (.133333) \times 10^{[6-5]} \ , \tag{1.3}$$

where we would store the 1 (for the sign), 133333, and 6. What are the largest and smallest possible numbers, and what is the precision? The exponent can be as large as 9 or as small as 0 so the numbers range from 0.1×10^{-5} to $.999999 \times 10^4$. The precision is six decimal digits. This means that while we can represent $x = 3500 = 0.35 \times 10^{[9-5]}$ and $y = 0.0021 = 0.21 \times 10^{[2-5]}$, if we try to add them and store the result, we find that $x + y = x$! This is our first example of a source of "round-off errors" and of the general maxim that

"Computational math ≠ regular math."

In base 2, the mantissa for a single-precision float can take the form

$$\text{mantissa} = m_1 \times 2^{-1} + m_2 \times 2^{-2} + \cdots + m_{23} \times 2^{-23} \ , \tag{1.4}$$

where each m_i is either 0 or 1, so there are 23 bits to store (each of the m_i's is either 0 or 1). For the sign we need 1 bit. If we use 8 bits for the exponent, that is a total of 32 bits or 4 bytes (i.e., 1 byte = 8 bits). To have a unique representation with all numbers having roughly the same precision, we require $m_1 = 1$ (except for 0) since, for example, we could otherwise represent 1/2 as both $(1 \times 2^{-1}) \times 2^0$ and $(1 \times 2^{-2}) \times 2^1$. (Thus, m_1 doesn't have to be stored in practice and we could, in principle, pick up an extra bit of storage.) The largest number stored would be

$$\underbrace{0}_{\text{sign}} \quad \underbrace{1111\ 1111}_{\text{exponent}} \quad \underbrace{1111\ 1111\ 1111\ 1111\ 1111\ 111}_{\text{mantissa}} \tag{1.5}$$

and the smallest number would be

$$\underbrace{0}_{\text{sign}} \quad \underbrace{0000\ 0000}_{\text{exponent}} \quad \underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 000}_{\text{mantissa}} \tag{1.6}$$

To figure out the actual range of numbers that can be stored, we also need to specify the bias, which is $127_{10} = 0111\,1111_2$ for single precision. This means that the number 0.5 is stored as [1]

$$\underbrace{0}_{\text{sign}} \quad \underbrace{0111\ 1111}_{\text{exponent}} \quad \underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 000}_{\text{mantissa}} \tag{1.7}$$

It also implies that (you verify these!)

$$\text{largest number:} \quad 2^{128} \approx 3.4 \times 10^{38} \tag{1.8}$$

$$\text{smallest number:} \quad 2^{-128} \approx 2.9 \times 10^{-39} \tag{1.9}$$

$$\text{precision:} \quad \text{6–7 decimal places (1 part in } 2^{23}) \tag{1.10}$$

If a single-precision number becomes larger than the largest number, we have an *overflow*. If it becomes smaller than the smallest number, we have an *underflow*. An overflow is typically a disaster for our calculation (but see later discussion on `inf`), while an underflow is usually just set to zero automatically without a problem. For a double precision number, eight bytes or 64 bits are used, with 1 for the sign, 52 for the mantissa, 11 for the exponent, and a bias of 1023. *Figure out the expected range of numbers and the precision for doubles.*

This discussion of floating point numbers is based closely on Ref. [1]. *Are the results* (1.8) *and* (1.9) *and the corresponding results for double precision consistent with what you find empirically in Session 1? Can you think of how to explain any discrepancies?* In the Session 2 notes, we'll follow up with some discussion of the IEEE standard for floating-point numbers, which is the actual implementation on the computers we use (with Intel chips).

Most floating-point numbers cannot be represented exactly (those that can are called "machine numbers"; there are only a finite number in total!). For example, the decimal 0.25 is a machine number but 0.2 is not! We can use Mathematica to find the first digits of the base 2 representation of 0.2:

```
BaseForm[0.2,2]
```

yields $0.00110011001100110011001101_2$ and the pattern actually repeats indefinitely (can you do base 2 long division to derive this by hand?). Now suppose we only had enough storage to keep 0.00110011. Use Mathematica again to convert to a decimal (base 10) with 10 significant figures:

```
NumberForm[BaseForm[2^^0.00110011, 10], 10]
```

which yields 0.19921875. So the actual number deviates from the computer representation in the fourth decimal place. The maximum deviation possible is related to the *machine precision*.

Any number z is related to its machine number computer representation z_c by

$$z_c = z(1+\epsilon) \quad \text{for some } \epsilon \text{ with} \quad |\epsilon| \lesssim \epsilon_m \ , \tag{1.11}$$

where ϵ_m is the machine precision, which is defined as the largest number ϵ for which $1 + \epsilon = 1$ in a given representation (e.g., float or double). [In the example above, $z = 0.2$ and $z_c = 0.19921875$. What are ϵ and ϵ_m?] Note that the machine precision ϵ_m is *not* the smallest floating-point number

that can be represented. The machine precision depends on the number of bits in the mantissa while the smallest number depends on the number of bits in the exponent [3].

You will roughly determine empirically the machine precision for C++ floats and doubles in Activity 1. When printing out a decimal number using `cout`, the computer must convert its internal representation to decimal. If the internal number is almost garbage, the output may be unpredictable. This may be relevant for the method of determining the machine precision in Activity 1.

Repeated operations (e.g., multiplications or subtractions) can *accumulate* errors, depending on how numbers are combined. We'll explore the perils and possibilities in detail in Activity 2.

f. First Comments on the 1094 Sessions

Finally, some general comments on the course and the sessions.

- It is not required that you finish all tasks in an activity. We'll let you know if you need to continue working on an activity in the next class period. [Note: Some activities are designed for more than one period and some tasks will be completed as homework.]
- The various groups of two in the class will work at different rates, particularly at the beginning of the quarter. There is *no* problem with this. If you fall way behind, I might suggest to you that you spend some time outside class catching up. I will have special office hours for that purpose.
- *Please read the activity instructions very carefully as you go so you don't miss things!*
- Take advantage of your classmates, who are your colleagues in 1094. Comparing and discussing results is a "win-win" process for any level of expertise or confusion.
- You will be asked to hand in the 1094 activity guides at the end of class (or after completing some parts), with the questions answered. They will be reviewed and returned at the next class session; this will be part of your grade. It's easy to get sloppy and skip some of the tasks and questions, but it will pay off in the long run to do and discuss everything.
- When working through the activity guides, try to *predict* outcomes whenever possible, but always *postdict* at least, particularly if you made a wrong prediction or none at all. This means to *understand* your result after you get it.
- Question Authority! Computational physics is in many ways an experimental science, which means that (correct) data trumps authoritative pronouncements. You'll encounter this in Activity 1, where the expected result from the discussion in these notes for one of the tasks will disagree with your observations. You are encouraged to challenge any statements in the notes or other references or made by the instructors.
- Validation and verification (which are different procedures; look up "verification and validation (software)" on Wikipedia) are critical in computational science. How do you *know* a program is working and calculating the correct result (to the accuracy it should)? Getting an

answer (no error messages) is not enough! In Activity 1 we'll ask these questions for a simple problem as a warm-up; you should adapt your laundry list of "checks" from other courses and research to the results we will encounter. For example:

- compare to an analytic (or otherwise known) solution;
- solve by another method;
- consider special or limiting cases (e.g., to check whether the formula for the period of a pendulum $T = 2\pi\sqrt{L/g}$ is correctly implemented in a code, test if $g \to 0$ or $L \to \infty$ gives the expected trend);
- test with scaling (what happens when you multiply a parameter by some factor?). For the pendulum example, calculate with a value of L and then $4L$; you should find $T \to 2T$.

The pendulum example is very simple, but the concepts carry over to the most sophisticated algorithms.

g. References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 6810 info webpage for details on the updated eTextbook version.]

[2] M. Hjorth-Jensen, *Computational Physics* (2015). These are notes from a course offered at the University of Oslo. See the 6810 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from http://www.nrbook.com/a/. There are also Fortran versions.