

1. 780.20 Session 1

a. Background to 780.20 Computational Physics

This Computational Physics course originated 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-time need for both undergraduate and graduate computational physics and someone had to dive in and start developing the curriculum. So here we are in the eighth edition of the course. Even after this many cycles the course is still evolving significantly, both in content and structure, which means that feedback from the participants is important.

Here are some details about the current instructor's (Dick Furnstahl) background. I have a lot of years of experience writing computer programs; my first program was written over 40 years ago! That doesn't necessarily translate into expertise on numerical algorithms or good programming practices, of course. Hopefully I've picked up some good habits along the way. 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. 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 but instead use small-scale parallel processing or run my programs for hours or less on a fast PC. (However, for the last three years I've been part of a SciDAC collaboration that uses petascale computing.) My programming is primarily done in C/C++, Fortran-90, Perl, MATLAB, and Mathematica on Linux systems these days 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 growing increasingly enthusiastic about Python as a language for computational physics.

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 780.20 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. 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 I'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. It used a hands-on, project-oriented approach, with breadth instead of depth, with special emphasis on error analysis and numerical libraries. Example programs were in C and Fortran77. There has been no required text for the last several years, but we have used Morten Hjorth-Jensen's *Computational Physics* lecture notes [2] as a further guide; these notes have been developed for a similar (but more extensive) course taught in Oslo, Norway with programs in C++ and Fortran90. There is an updated version from fall, 2009 that we'll use extensively. This year we'll continue to build on both of these references (and others) with our own notes. A very useful supplementary reference for numerical algorithms is *Numerical Recipes* [3], available a chapter at a time online in PDF form (see 780 web page).

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. These notes will generally be made available at least the day before a 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.

Everyone will be required to do a project based on a physics problem of interest to them. It could be part of senior 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 780.20 (a list will be available on the course web page). The pedagogical goal is to have you 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. 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 I am rarely disappointed.

b. Computing Environment

A general theme is to use basic and portable tools, which is why we concentrate on the GNU compilers and programming utilities. The main programming language will be C++ and the main compiler we'll use is g++. We'll also look at Python, 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 directly on Linux (your choice in Smith 1011). *Everyone will need an account on the Physics Department Windows or Unix computers.* You are encouraged to set up on your own computer to match the Smith 1011 setups; there will be instructions linked on the 780 homepage. 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 xmgrace).

We will not assume that you know how to use any of these software tools (although you might). The 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.

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. We will not be able to explore this topic in great detail, given our time limitations; however, if there is interest I'll supplement the official class material with more object oriented programming.

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 780 web page before Session 1 as a pdf file `area_listing.pdf`). *Please read the first two chapters in the Hjorth-Jensen notes (under Supplementary Reading on the web page) for more generalities on programming.* 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* `area.cpp` and *link* it to create the program “area” in one step at the command line (in Cygwin under Windows or in Linux) using the gnu C++ compiler (called `g++`): `g++ -o area area.cpp`
This command combines two steps, which can be done separately:
compile: `g++ -c area.cpp` creates the *object* file `area.o`,
link: `g++ -o area area.o` combines `area.o` with system libraries to create the program `area`.
If you use Dev-C++ (or another IDE), these steps will be carried out through menu commands.

There are many possible options for `g++` and when we have many different files (in larger programs), it can be awkward to manage. The “make” utility can manage it for us, using “makefiles”; we'll see an example in Session 1.

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

- 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
     $\pi = 3.14159$ 
     $\text{area} = \pi \times \text{radius}^2$ 
print area
```

Don’t start coding before thinking about 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 never do it (based on long experience!).
- Note the elements in the documentation comments at the top. 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.
- 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.) Also, use appropriate variable names that are easy to understand (e.g., “radius” and “area”), which makes them self-documenting.
- The “include” files play the same role as packages included in Mathematica (to read in special functions or definitions). 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 people 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, 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.
- 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 the number). More on this below, but keep in mind that the precision is less than ∞ !

- 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.*
- 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 carriage return 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.]
- There is a function for raising a variable to any power, which is called pow. Can you guess why it is not used here? (Hint: pow works for any real exponent, not just integer exponents, and doesn't distinguish the two cases.)

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 *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`

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 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. Since 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^{24} [1]! [Note: 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!]

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

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

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:

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

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 a 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:

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

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 $x + y = x$!

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

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

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

$$\underbrace{0}_{\text{sign}} \quad \underbrace{1111 \ 1111}_{\text{exponent}} \quad \underbrace{1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111}_{\text{mantissa}} \quad (1.5)$$

and the smallest number would be

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

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]

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

It also implies that (you verify these!)

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

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

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

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 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 $0.0011001100110011001101_2$ 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. As a decimal, this is 0.19921875. So the actual number deviates from the computer representation. The maximum deviation is related to the *machine precision*.

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

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

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 former depends on the number of bits in the mantissa while the latter depends on the number of bits in the exponent [3].

You will roughly determine empirically the machine precision for C++ floats and doubles in Session 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 Session 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 Session 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 a session. I'll let you know if you need to continue working on a session in the next class period. [Note: Sometimes sessions will be 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 session instructions very carefully as you go so you don't miss things!*
- You will be asked to hand in the 1094 session guides at the end of class (or after completing some parts), with the questions answered (they will be 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.

g. References

- [1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 780.20 info webpage for details on a new version.]
- [2] M. Hjorth-Jensen, *Lecture Notes on Computational Physics* (2009). These are notes from a course offered at the University of Oslo. See the 780.20 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.