Program Grading Rubric

This document lays out common criteria used to grade PHYS 409 Computational Methods programming assignments. Each criterion has a number of different levels of achievement, with a description of how a submission will attain that level and the number of points assigned for reaching it. Please email or ask me if you have any questions about this rubric.

Criteria

Program Specifications / Correctness

This is the most important criterion. A program must meet its specifications (whether from a textbook problem or as written in the assignment) and function correctly. This means that it behaves as desired, producing the correct output, for a variety of inputs. This criterion includes the need to meet specifications by writing a program in a particular way or using a particular language feature or numerical approach, if such a thing is mentioned in the problem.

If a specification is ambiguous or unclear, you have two choices: You can either make a reasonable assumption about what is required, based on what makes the most sense to you, or you can ask the instructor. If you make an assumption about an ambiguous specification, you should mention that somewhere in a comment so that the reader/grader knows what you were thinking. Points may be taken off for poor assumptions, however.

Presentation of Results

Care should be taken when you present your results. Numerical answers should be appropriately formatted and organized in tables if lots of data are involved. Graphs should be constructed in such a way that they communicate what is intended efficiently and elegantly. Communicating information through visual representations can be something of an art form. Decisions that affect how effectively a graph communicates the desired idea include whether to use a linear or log scale, whether to plot multiple results on a single graph or on multiple graphs, whether to include reference lines or curves, whether to include a legend or label curves directly, etc.

Readability

Code needs to be readable to both you and a knowledgeable third party. This involves:

- Using indentation consistently (e.g., every loop, if statement, etc. is indented to the same level)
- Adding whitespace (blank lines, spaces) where appropriate to help separate distinct parts of the code, but not too much white space

```
i = i + 1;          % not i=i+1;
x = 2*x - 1;        % not x=2*x+1;
A(i+2) = 3;         % not A( i + 2 ) = 3;
```

- Don't include unnecessary parentheses. Know the order of operations.

```
z = 2*a*x / y.^2 + 3;   % not z = (2*(a*x))/(y.^2) + 3;
```

- Give variables meaningful names. Don't define `A` as the mass of an electron, use `me` or `mElect` or something similar. Use comments to define variables the first time they are used. Also, don't go crazy with variable names. For example, `massOfElectronInKg` is certainly specific, but can actually make complex equations more rather than less difficult to read.

```
g = 9.8;                 % GOOD for acceleration of gravity
x = 2;                   % GOOD for position x
m1 = 100;                % GOOD for mass of particle 1
mElectron = 1.609e-19;   % OK for mass of electron, but too
                         % verbose if you use it a lot
me = 1.609e-19;          % BETTER if you will use it a lot
```

The code should be well organized. Functions should be defined in one section of the program, code should be organized into functions so that blocks of code that need to be reused are contained within functions to enable that, and functions should have meaningful names. This is a concept that we will be learning about as we write more and more code in CS 127, and so few points, if any, will be taken off for organization issues that we have not yet addressed in class.

Documentation

Every code should start with a header comment. At the very least, this header should contain:

- name of the program
- a short description of what your program does including any input that the program requires and any output that it may produce
- your name
- the names of any lab partners with whom you might have collaborated
- date you turn the program in
- name of the class
- assignment and problem number
- detailed description of the approach and numerical methods used in the code if it is complex including references to resources you may have used to write it

All code should also be well-commented. This requires striking a balance between commenting everything, which adds a great deal of unneeded noise to the code, and commenting nothing, in which case the reader of the code (or you, when you come back to it later) has no assistance in understanding the more complex or less obvious sections of code. In general, aim to put a

comment on any line of code that you might not understand yourself if you came back to it in a month without having thought about it in the interim. Like code organization, appropriate commenting is also something we will be learning about as we write code throughout the semester. Here are some guidelines adapted from a Phython course (CS11) at CalTech:

- Write general comments in full, grammatically correct sentences.

```
% This code calculates the fast Fourier transform of a
% two-dimensional image. It uses the method outlined in..
```

- Always leave a space after the comment sign:

```
% This is easier to read
%This is harder to read
```

- Don't state the obvious

```
x = x + 1;      % increment x   (BAD - redundant)
i = 1;          % i             (BAD - Meaningless)
```

- As you change the code, make sure you update your comments.
- Try to line up your comments with each other as best you can, but don't go overboard

```
x0 = 100;       % initial value of x
y0 = 20;    % initial value of y       (BAD)
```

  This is better:

```
x0 = 100;       % initial value of x
y0 = 20;        % initial value of y    (GOOD)
```

- Longer codes can benefit from labeling sections with comments that stand out like this. Also use blank lines between sections of the code that have different functions.

```
blah
blah

%%%%%%%%%%%%%%%  Curve-Fitting  %%%%%%%%%%%%%%%%%%

more
blah
blah
```

Reusability

In general, scientific code written to solve specific problems will not be as general or as flexible as larger applications written by a software engineering firm. Even still, physics codes can benefit from a few simple rules:

- Define all parameters or constants as variables. Don't substitute numbers directly into code.

  This is BAD:

```
period = 2 * pi * sqrt(1.5/0.8);   % period of pendulum
```

  This is BETTER:

```
g = 9.8;       % acceleration of gravity (in m/s^2)
L = 1.5;       % length of pendulum (in m)

period = 2 * pi * sqrt(L/g);       % period of pendulum
```

- Once we have covered functions, it is best to define functions that are used to calculate often-used quantities, i.e. Fourier transforms, Hankel transforms, etc. Defining a function that you can pass parameters and data to is the best way of maximizing the reusability of your code.

Code Efficiency

There are often many ways to write a program that meets a particular specification, and several of them are often poor choices. They may be poor choices because they take many more lines of code (and thus your effort and time) than needed, or they may take much more of the computer's time to execute than needed. For example, a certain section of code can be executed ten times by copying and pasting it ten times in a row or by putting it in a simple `for` loop. The latter is far superior and greatly preferred, not only because it makes it faster to both write the code and read it later, but because it makes it easier for you to change and maintain. We will discuss tricks throughout the course to help you write efficient, elegant code.

Program Grading Rubric

Criteria	Exceptional (3)	Good (2)	Acceptable (1)	Unacceptable (0)
Specifications* (counts 2x)	The code works and meets all of the specifications including using the proper numerical methods.	Minor details of the program specification are violated, but generally produces correct results.	The code contains some minor bugs that prevent it from performing optimally in all cases.	The code does not function optimally or gives incorrect results and does not meet the specifications.
Presentation of Results	The code presents any graphical or numeric results in a professional and elegant manner.	The results are clearly presented yet not completely "polished"	The results are readable but could benefit from better organization.	The results are hard to read or disorganized.
Readability	The code is exceptionally well organized and very easy to follow.	The code is fairly easy to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is poorly organized and difficult to read.
Documentation	The documentation is well written and clearly explains what the code is accomplishing and how.	The documentation consists of embedded comment and some simple header documentation that is somewhat useful in understanding the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation is simply comments embedded in the code and does not help the reader understand the code.
Reusability	The code is easily adaptable to other related problems.	The code could be adapted to other related problems with some effort.	The code contains portions that make specific assumptions about the problem and are not easily generalized.	The code is not general and very difficult to reuse in other situations.
Efficiency	The code is extremely efficient without sacrificing readability or understanding.	The code is fairly efficient without sacrificing readability or understanding.	The code is slow, uses brute force methods and/or unnecessarily long.	Many things could have been accomplished in an easier, faster or otherwise better fashion.

*In the event that the code does not meet the specifications at all, no credit will be received for the other categories.