UC Santa Cruz
Physics 115
Spring 2018

Exercise Set #1

due online Wednesday, April 11 at 10:10 AM

Here are some instructions and advice on coding and submitting your work.
PLEASE READ THEM CAREFULLY.

1. The answers to the questions must include the computer code and output, in addition to any text explanations that might be needed. If you are not sure what is required, please ask.

2. Please assemble all parts of a question together. We don't want to have the code for all the questions together, followed by all the output, followed by all the explanations. Instead, present all parts of question 1, followed by all parts of question 2, etc. The easiest way to do this is with a notebook-style format (Jupyter, etc.). The second-easiest way is in a LaTeX document, using the `verbatim` environment. Either way, submit the file itself instead of taking a screenshot of the item on your screen. The file should compile so that we can check the output.

3. Present the appropriate number of digits in your output. We recommend using the `%f` and `%e` formatting flags. For example,

```
>>> a=13.946
>>> print(a)
13.946
>>> print("%.2f" % a)
13.95
>>> print("%.2f" % round(a,2))
13.95
```

4. In some problems there will be many lines of output, only a few of which are relevant. Just print the relevant lines. You could indicate the missing lines by a line with something like ......

5. In problems where you have to show that the error scales as a certain power of a small parameter h, choose values for h which decrease in a geometric (rather than arithmetic) sequence. For example, you can divide by 2 or 10, whichever is most appropriate, each time.

6. Make sure that constants are defined with the desired precision. In one of the homework problems you will need to specify the golden mean, $(\sqrt{5}-1)/2$, whose numerical value is about 0.61803399. However, this numerical value is not to double

precision accuracy (16 decimal places). The simplest way to get the golden mean to double precision accuracy is to let the computer calculate it:

```
>>> import numpy as np
>>> phi = (np.sqrt(5) - 1) / 2
>>> phi
0.6180339887498949
```

(Check the decimal places.)

7. Regular Python does not have a single-precision `float` variable type; all Python floating-point numbers are `double` precision. The `numpy` package has `float32` single-precision float and `int32` integer types if you need them.

```
>>> import numpy as np
>>> a32 = np.float32(5.)
>>> a64 = 5.
>>> 1./a32
0.20000000000000001
>>> 1./a64
0.2
```

Other languages (C, C++) have separate single-precision and double-precision representations.

Now we start the questions.

1. (a) Kernighan and Ritchie, who created the C language, advised, "The first program to write in any language is the same for all languages: Print the words 'Hello, World.'. This is the basic hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy."
Clear the hurdle by writing a `Hello World` program. (It's still called `Hello World` even if we use a comma. Don't forget the comma.)

   (b) i. Print the number "19 billion" in scientific notation. The result should look something like `1.9e+10` (You will need to use the `%e` formatting string.)

   ii. Print
`The value of the golden mean is ...`
where you replace "..." by the numerical value of $(\sqrt{5} - 1)/2$, which you should give to 8 decimal places.

2. Even calculating the sum of a simple series can require some care. Consider the series:

$$S^{(\text{up})} = \sum_{n=1}^{N} \frac{1}{n}$$

which is finite as long as N is finite. When summed analytically, the series converges to the same result whether you sum the series upwards from $n = 1$ or downwards from $n = N$, as in

$$S^{(\text{down})} = \sum_{n=N}^{1} \frac{1}{n}$$

However, when the series are summed numerically, we find that $S^{(\text{up})} \neq S^{(\text{down})}$.

   (a) Calculate $S^{(\text{up})}$ and $S^{(\text{down})}$ in double precision.

   (b) Show that in double precision $S^{(\text{up})}$ and $S^{(\text{down})}$ agree to high accuracy.

   (c) Write a program to calculate $S^{(\text{up})}$ and $S^{(\text{down})}$ in single (`float`) precision for $N = 10^p$, with $p = 2, 3, 4, 5, 6$ and 7.

   (d) Taking the double precision result to be correct within the desired accuracy, show that, with single precision, $S^{(\text{up})}$ is less accurate than $S^{(\text{down})}$ and that the error increases with increasing N.

   (e) Explain in words why $S^{(\text{up})}$ is less accurate than $S^{(\text{down})}$.

3. Roundoff errors increase exponentially in certain algorithms. We say that such algorithms are unstable. The first rule of numerical analysis is to avoid unstable algorithms. The "Golden Mean," ϕ is given by $\phi = \left(\sqrt{5} - 1\right)/2 \cong 0.61803399$.

   (a) Determine the nth power of ϕ by using successive multiplications,

   $$\phi^0 = 1, \quad \phi^n = \phi \cdot \phi^{n-1}$$

   for $n = 1, 2, 3, \ldots$

   (b) Now let's try to be clever and develop the algorithm further. Show that the following recursion relation holds:

   $$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1}$$

   Hint: reduce this to a quadratic equation.

   With that relation in hand, we can calculate ϕ^n using only subtraction, rather than multiplication, by constructing an array of numbers $\Phi[0], \Phi[1], \ldots$ from the following relation

   $$\Phi[n+1] = \Phi[n-1] - \Phi[n] \tag{2}$$

   with the "boundary conditions" $\Phi[0] = 1, \Phi[1] = \phi$. The value of $\Phi[n]$ will then be ϕ^n.

(c) Use Eq. 2 in both single and double precision to compute ϕ^n, for $n = 1, 2, ..., 50$. Compare the results with those in part (a). Is the recursion relation unstable?

(d) Show that there is another number, $\tilde{\phi}$, which satisfies the recursion relation in Eq. 2. What is the general solution of Eq. 2? Using the form of the general solution, explain the instability.

4. (a) Show analytically that the midpoint expression for the second derivative

$$f''_{\text{mid}}(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2}$$

has an error of order h^2.

(b) Confirm this error estimate numerically for the function $f(x)$ and x value of your choice. (*Warning*: do not let h get too small, or roundoff errors will spoil your analysis.)

5. Write a program to compute the spherical Bessel functions $j_l(x)$ for the first 25 l values at $x = 0.1, 1$, and 10. Choose a value for x and use the recurrence relation

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x), \tag{3}$$

starting from a large value of l (for example, $l = 50$) and some assumed values (perhaps 1.0) for $j_l(x)$ and $j_{l+1}(x)$. (In fact, the result will not depend on these initial assumptions.)

Calculate the ratio of the resulting value for $j_0(x)$ with the value obtained from the analytic function

$$j_0(x) = \frac{\sin(x)}{x}.$$

Multiply your values for the higher l values by this factor. This method works for values of x smaller than the largest values of l used in the downward iteration.

Note:

- Results for the first three values of n are given in the book but it is not enough to quote the result; you must show the computer output and the code that produced it.
- One can rewrite Eq. 3 so it is an *upward* recursion

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x). \tag{4}$$

It might be tempting to use this to compute the value of $j_l(x)$ for any l since we know the first two functions

$$j_0(x) = \frac{\sin x}{x}, \qquad j_1(x) = \frac{\sin x - x\cos x}{x^2}.$$

Unfortunately, Eq. 4 is unstable for small values of x, and so is not useful in practice. You may want to check this. The reason is that there is a second function which satisfies the recurrence relations, the spherical Neumann function, $n_l(x)$. Using the upward recursion, the amount of the unwanted Neumann function function grows so the algorithm is unstable. This is similar to the problem with calculating the golden mean from the recurrence relation Eq. 2. However, with the downward recursion relation, the amount of the Neumann function diminishes (at least if x is smaller than the order, l, at which the iterations start) and eventually becomes negligible even if is significant to start with.