# ENGR 102 - Lab Assignment #4

TEXAS A&M UNIVERSITY

Fall 2020

General course learning outcomes:

- demonstrate the use of programming techniques in computer programs, including: techniques to collect, store, and manipulate data within a computer program, use control structures such as conditionals and loops; decompose a complicated tasking into more manageable pieces.
- apply programming techniques to solve problems in engineering.
- complete a team programming assignment that ties together concepts learned in the class.

---

## Activity 1: Conditionals with Round-Off Error - to do in lab (team)

☑ *Demonstrate a common error occurring in programming.*

When performing numerical computations, one challenge you can run into is floating-point round-off error. This occurs when the computer represents a number that requires an infinite number of decimal digits, but must round them off. That small round-off error can cause significant issues.

*Part 1: Identifying floating-point problems*

We first want to illustrate some examples where floating-point round off *may* (or *may not!*) cause trouble.

a. As a team, type in and run the following program. Notice the value of **a** is rounded off (1/7 is an irrational number). If there is no round-off error, the value of **b** should be 1. Is it?

```
a = 1/7;  print(a);
b = a*7;  print(b);
```

b. Add the following lines to your program. In this case, the value of **e** should be 1 if there is no round-off error. Is it?

```
c = 2*a;  d = 5*a;  e = c+d;
print(e)
```

c. Finally, try the following. Values of y and z should be 1 with no round-off error. Are they?

```
from math import *
x = sqrt(1/3); print("x =", x);
y = x*x*3; print("x*x*3 =", y);
z = x*3*x; print("x*3*x =", z);
```

*Part 2: Rockets' Red Glare*

Sometimes round-off errors occur and cause issues, and sometimes they don't. We can't always predict when round-off error will be obvious, as you should have seen in the previous example. This small error may not seem important, but let's look at how it can affect a computer program.

Your team is creating a program that determines when rocket stages should disengage, and when the following stage should ignite. You measure in 0.1 kilometer increments, and have determined that the first stage should disengage at 0.3 km, and the second stage should ignite at 0.4 km.

a. Create three constants: `FIRST_STOP = 0.3; SECOND_GO = 0.4; INCREMENT = 0.1;`

b. Create a variable for user input, change this input to an *int* type: `current_increment`

c. Create an if/elif/else statement to print to the user what action should occur at the current increment.

   If the value is 0.3, print "Stop the first stage!"; if the value is 0.4, print "Start the second stage!"; if the value is anything else, print "No action required at this point".

d. Once your program is set up, what happens when the user types in values ranging from 1 to 5?

*(continued, next page)*

*Part 3: Tolerances for Comparisons*

In part 2, you should have seen that, in some instances, comparing the value of a floating-point number can lead to unexpected errors.

A common way for dealing with floating-point error is to use *tolerances*, where you will compare two values that are *close* in value, but not identical, to each other. Rather than checking a==b, we compute a–b and see if that is within some small value away from 0 (above or below). To do this, take the absolute value of (a–b), and compare it to some small value, called the tolerance. If it is less than the tolerance value, the two things are considered equal. Tolerance is commonly abbreviated TOL or EPS (short for epsilon).

Let's see if we can fix the issue we saw above.

a. Keep what you worked on above in your rocket program, but add a second if/elif/else block. You can copy/paste what you worked on above if you would like.
b. Create a new constant: `TOL = 1e-08`
c. Modify the conditional expressions for each if/elif case to check if the current distance traveled is *near* the required values within the tolerance you set, as opposed to *exactly* equivalent.

   That is, rather than checking if `distance_traveled == 0.3`, instead check whether `abs(distance_traveled - 0.3) < TOL`.
d. Decrease the tolerance to 1e-20; does this affect the results. At what tolerance does the floating point error affect the results for this program? (Change by factors of 10.)
e. Submit your *Part 3* program at the last tolerance level that did not affect the results.

*Summary:*

As you write future programs, think about your comparisons and whether you need to use tolerances. If you are checking for exact equality and could have floating-point error, you should probably use tolerances. If you are checking which of two things is larger, a tolerance is likely unnecessary. Tolerances are particularly helpful when checking things like whether a denominator is (nearly) 0, and thus division is likely to create division errors.

Tolerances are used extensively in engineering, and understanding the concept will help you in your future courses. On a more short-term basis, this concept is almost certainly going to appear on your exams.

*Note: Tolerances have problems of their own; for instance, you can find that a==b, and b==c, but a != c. Dealing with issues of round-off error is a long-standing and ongoing area of research.*

*(continued, next page)*

## Activity 2: Writing and Testing a Program - to do in lab (team)

☑ *Create a plan, well-designed test cases, and write a larger program according to incremental coding practices.*

*Background*

In engineering and science, we often want to calculate the effect of some complex behavior. To make this possible, we create a *model* to describe the behavior in a way that is understandable and computable. Some models are based on physical laws and principles, some are based on replicating observations, and many are a combination. Once created, you can use a model to analyze and predict the performance or behavior of some system or phenomenon.

This week, you will generate a program that uses a model developed by NIH to estimate a person's 10-year risk for likelihood of a heart attack. This model was constructed from data analysis of various factors that have been demonstrated to contribute to heart attacks for those aged 20–70.

The process is on the final page of this document: *https://www.nhlbi.nih.gov/sites/default/files/publications/05-3290.pdf*

*Part 1: Thinking About the Program*

BEFORE CODING, put together a document *(to be saved as a PDF for submission)* that your team will use to analyze the problem.

As a team, review the table on the last page of the document referenced above, and ensure you understand how it works. To make this slightly less time-consuming, only create your program to handle Women from 20 to 59 years of age.

a. Consider what values you need to store. Make a list of the variables you believe you will need, and the names you will use for each.

b. Consider the general procedure for your program, and specify the sequence of steps it will follow.
   - Each step should be a short description of the goal of the step. These should not be code, but a description of the purpose of an action (e.g. "Compute points based on age").
   - You may indicate each part of any conditional statements as a separate action.

c. List all test cases that you will use in your program. For this model, a "good" set of tests would involve much greater than 100 tests. This is rather cumbersome for class, so instead:
   - Have at least 40 different test cases *(remember to write for incremental coding. Early tests should verify one or two inputs, later tests will verify more—otherwise it's not incremental!)*
   - Your sample test cases should verify all parts of the program (e.g. don't just come up with 40 tests for the age-based points for the women).

d. For each test case, give a BRIEF statement including:
   - What aspect of the program is being tested (e.g., user input, age calculations, etc.)
   - Is it a "typical" or "edge/corner" case
   - What are the input values, and what is the expected result.

   *Example: "Age-based score (typical case); Sex: Male, Age: 62; Age-based score: 10"*

e. You might find it easiest to divide up the work among your team for creating test cases, and then put them all into a table. Agree on formatting up front.

*Part 2: Constructing Your Program*

As a team, construct your program. As you do this, please be sure to do the following:

- Include comments for your program. An easy way is to copy your list of steps into PyCharm, and convert them all into comments ["Code" menu → "Comment with Line Comment"].
- When appropriate, remove and condense comments as you write the code.
- Develop incrementally. That is, write some code, and test it before writing the next section of code. Don't let your teammates skip this!
- Verify that your program runs and passes all test cases. You should, in the process of developing, test *every* test case you put together in Activity 1.
- Include specific instructions to the user, and write well-formatted descriptive output.