Assignment 2

Kole Frazier

Physics 2300

Appendix C – Debugging Practice

See file Appendix C – Debugging Practice.py header for information and fixed program code.

Exercise 2.1

Files: Exercise 2.1.py

Using the program I wrote, I found the time for a ball to fall 100m and hit the ground was 1.8 seconds.

Exercise 2.2

Files: Exercise 2.2.py

Part A: We know that r^3 = (Gravity * Mass of the Earth * Time^2)/(4*Pi^2), which represents the distance from the very center of the Earth that a satellite must have. Converting this to subtract the radius of the Earth (R), we can see that orbit height is:

Orbit Height = (GMT^2/4Pi^2)^(1/3)-R

Part B: See file: Exercise 2.2.py

Part C: Altitudes for orbit periods:

Once a day: 35,870,187.704 meters (approx. 35870.188 km)

90 minutes: 281,570.195 meters (approx. 281.570 km) – I did find that this is Low Earth Orbit (LEO), which comprises the first 100 to 200 miles (161-322 km) of space and is the approximate orbiting height of the International Space Station.

45 minutes: -2180143.388 meters (approx. -2180.143 km) – My program returned a large and negative number for 45 minutes (or 2700 seconds), despite being accurate for other values. I was not able to track down why this was, so I'm theorizing that this may be too low of an orbit to be calculated with a program setup like this one.

Part D: A geosynchronous satellite needs to stay in as close to the same position always. This is what makes it geosynchronous - it's in sync with a geographical location.

Earth really takes 23.93 hours to rotate once. Despite this, a day is recorded as being 24 hours long, so we must make up for it once every four years – on Leap Day. If a satellite were to fly at the altitude needed for a 24-hour orbit, it would slowly drift away from its geographically synced location, making it not have a geosynchronous orbit.

Altitudes for these time values:

24 hours (86400 seconds): 35870187.704 meters (approx. 35870.188 km) (Google confirms this!)

23.93 hours (86148 seconds): 35788012.0824 meters (approx. 35788.012 km)

The different is 82,175.622 meters (approx.. 82.176 km), which is quite a bit in terms of distance and velocity.

I did find it interesting that the International Space Station is able to orbit at just a few hundred kilometers, but that a geosynchronous orbit requires a satellite to be much further out.

Exercise 2.6

Files: Exercise 2.6.py

I'm very sure that what I'm using to calculate V2/Velocity of the second orbiting object is not correct, as the calculated values are not in the ballpark.

Earth:

v2: 887572624.882	L2: 5019552.85134
T: 89117.9833351	e:-0.999931755519

Haley's Comet:

v2: 49440391.7223	L2: 96869824.5131
T: 168203.423494	e:-0.997796581984

Exercise 2.7

See Exercise 2.7.py for program and output.

Exercise 2.8

Files: Exercise 2.8.py

I initially used two test arrays (a=array[1,2] and b=array[3,4]) to verify array-math behavior before proceeding to parts A, B and C. Adding A+B yields [4,6], so I'm assuming that the array math is simply working on a one-to-one basis, A[i] to B[i].

Part A: (b/a+1)

Hand-worked:

2/1+1 = 3, 4/2+1=3, 6/3+1=3, 8/4+1=3 => [3,3,3,3]

Actual: [3,3,3,3]

Part B: (b/(a+1)]

I had a couple of simple things I had to watch out for after trying this by hand: order of operations and mathematical operations with integers and floating-point numbers in programming. When I first ran my program, I had some results that were way different from my hand-worked results until I recalled this issue.

Hand-worked:

2/(1+2)=1, 4/(2+1) = 4/3, 6/(3+1) = 3/2, 8/(4+1) = 8/5 => [1,4/3,3/2,8/5] = [1.0,1.333,1.333,1.6]

Actual: [1,1,1,1] – My hand-worked part did not initially consider the integer math, which truncates (not rounds) any decimal numbers. It is worth noting that I left my answers as integers, as the exercise in the book specifically marks the arrays as integer arrays, so I didn't cast the numbers to floats.

Part C: (1/a)

Hand-worked: One divided by almost any number will result in a decimal value. Where the numbers are all whole integers, I'm expecting a lot of numerical truncations.

[1/1=1, 1/2=0 => 0, 1/3 = 0.333 => 0, 1/4 = 0.25 => 0] => [1, 0, 0, 0]

Actual: [1,0,0,0] (I was correct!)

Exercise 2.12

Files: Exercise 2.12.py

My program finds 1,229 prime numbers less than 10,000.

Exercise 2.13

Files: 1) Exercise 2.13 – Part A.py 2) Exercise 2.13 – Part B.py

Part A: My program finds C100 to be 5.09014835291e+46

Part B: My program finds the greatest common divisor of 108 and 192 to be 12.