CSCI 551

Austin Dollar

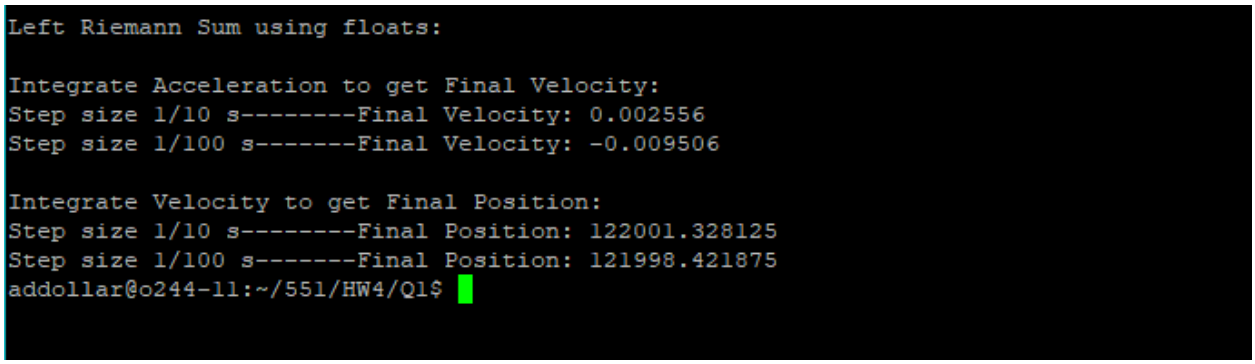
10/19/21

Exercise 4

Below are screenshots of code execution, as well as an analysis of any sources of inaccuracies for each sum type. All code is included in the accompanying .zip folder.

1. Left Riemann Sum, Float

Code Execution:



What are the sources of inaccuracy (errors) in this formulation and are there precision issues?

This specific method of Integration was found to be the least accurate and precise out of all of the methods tested, and this is largely for two reasons. As you can see, acceleration, while decently accurate, the final velocity still starts straying from the accepted value of 0.0000 as soon as after 2 digits, a lack of precision, likely due to the nature of floats. However, our final position is off by almost an entire 2 meters, straying relatively far away from its accepted value of 122000. This is bad, even for floats. This is due to the nature of the Riemann Sum, as there will be overlaps where our rectangles overlap past the curve, and due to the fact that we scaled our curve, these overlaps will not cancel eachother out, and would rather be compounding. Overall, for this specific problem, this would be the method that yields both poor precision and poor accuracy due to the choice of implementation method.

2. Left Riemann Sum, Double

Code Execution:

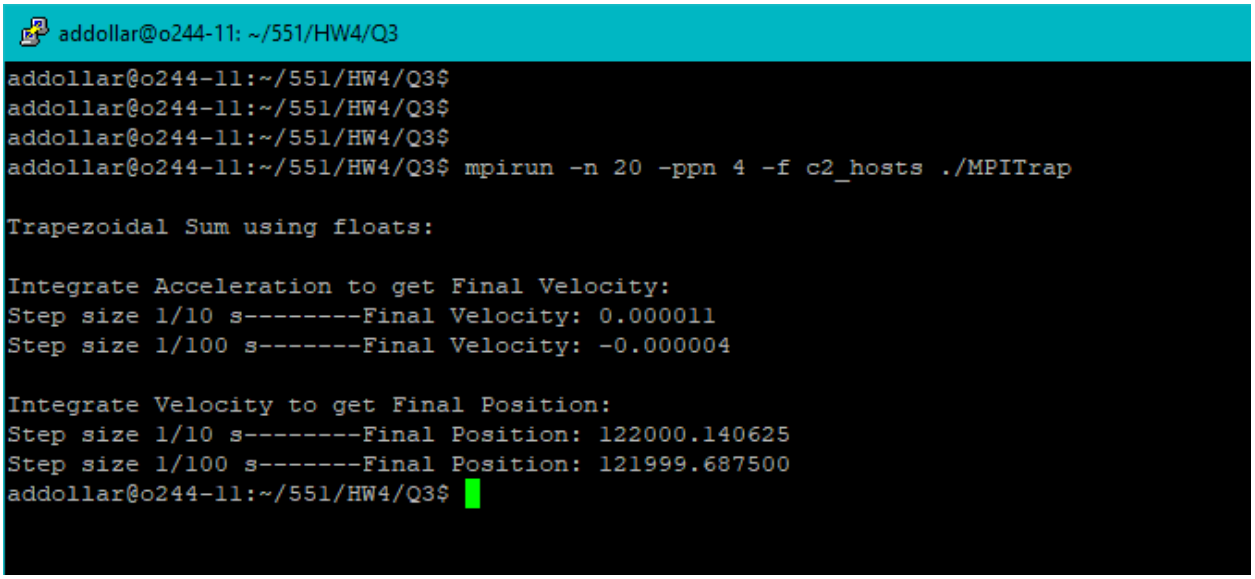
```
Left Riemann Sum using doubles:

Integrate Acceleration to get Final Velocity:
Step size 1/10 s--------Final Velocity: -0.000000
Step size 1/100 s-------Final Velocity: 0.000000

Integrate Velocity to get Final Position:
Step size 1/10 s--------Final Position: 122000.123878
Step size 1/100 s-------Final Position: 122000.123878
addollar@o244-11:~/551/HW4/Q2$
```

What are the sources of inaccuracy (errors) in this formulation and are there precision issues?

When we are using doubles for this integration method, we can see that it is much more precise, as well as accurate for this integration method. We are able to obtain an extremely precise and accurate measurement for the velocity, up to six digits. Additionally, we get much closer to an accurate value in our position calculations. All in all, doubles seem to be a much more accurate and precise data type than floats, which makes sense, as any lack of precision that occurs as a result of the data type will compound with each calculation.

3. Trapezoidal Sum, Float

Code Execution:

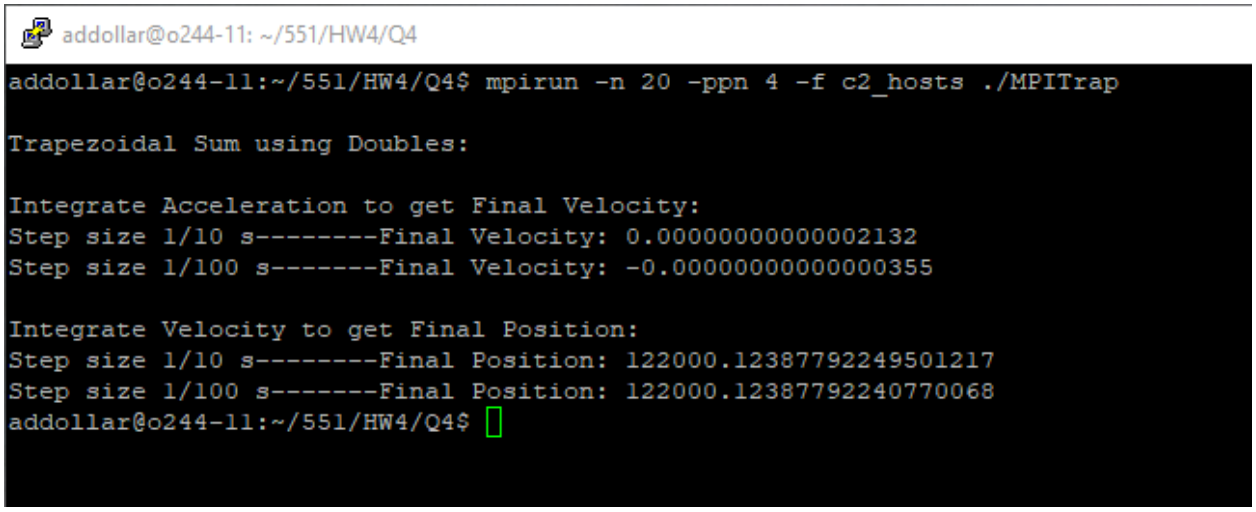


What are the sources of inaccuracy (errors) in this formulation and are there precision issues?

In regards to floats and the Trapezoidal Sum for these integrals, I believe that the trapezoidal sum is very accurate in terms of integration, at least for velocity, as we were able to get up to 6 levels of precision, which is the maximum allowed by floating point values. However, unsurprisingly, we were not as accurate when it came to the position calculation, with one of our position calculations being about .5 off. This is likely because floating points are only accurate for up to 7 digits, and we need at least 6 for this calculation, leaving not very much room to work with, which could lead to errors.

4. Trapezoidal Sum, Double

Code Execution:



What are the sources of inaccuracy (errors) in this formulation and are there precision issues?

This method of integration, as well as the data type used for calculations (double), is the one that I personally would trust the most. This is because, while you will get the inaccuracies from the math library in any version of this implementation, here, everything we can control is at its most precise. Without getting into huge data sizes the likes of long doubles, doubles themselves provide the most sig figs of precision, giving us 15 digits of precision, as opposed to a float's 7. Likewise, a trapezoidal sum is a more precise method of doing a definite integral, as the gaps that are created when taking the area here are much less significant than that of the alternative in this case, a Left Riemann Sum. This is due to the fact that due to the nature of us calling these functions, the overlaps on the Riemann Sum are not guaranteed to cancel out. My trust in these values are backed up by the fact that these values are the closest we get to the accepted value presented in the given spreadsheet. Our final velocity is only off by ~1x10^14, which is the highest level of precision available to a double. Our position is not quite as good, being ~0.1 away from the accepted value, however, it is among the closest solutions obtained.

Conclusions

Overall, it seems that in regards to this specific problem, the type of sum does not matter as much as the data type itself. In both cases, as anticipated, floats are generally much less precise than doubles, which came at no surprise. The only major difference seen between the two sums is the amount of digits we are given upon cout. That being said, I believe that the calculation reached of “122000.12387” is actually our true final position, rather than “122000.00000”. This is because the double solution for both sum types came to the exact same value and it was repeatedly reached as a solution, even when decreasing step size. This could be tested with more and more step sizes, but due to the fact that this calculation itself already uses more steps than the value in the spreadsheet, leads me to believe that my calculated value is more accurate than the one we were given.

All code is contained in the attached .zip folder, with each question having a labeled sub directory.