Case Study READ ME

Andy Zhang

I did most of the Case Study using concepts learned from the lectures, other material provided for the Python course, Python documentation for certain functions, and my knowledge of Python 3 from past coursework projects/homeworks/assignments. In addition, I also consulted Stack Overflow and other websites, which are cited in the appropriate places, whenever I got stuck. Overall, the Case Study was challenging, fun, and a great learning experience!

For Part 1: The Waterfall and Part 2: Waterfall Metrics, there were some minor challenges as well as some greater ones. The minor challenges were figuring out the best way to build the classes. For principal payments, interest payments, principal shortfalls, and interest shortfalls, I chose to use dictionaries to represent them as they seem fast; cf. <https://www.cnts.ua.ac.be/tutorials/python-performance-optimization>. The greater challenges would definitely have to be figuring out things like how principal due is computed as well as other calculations in the provided Excel spreadsheets. After emailing Mark with numerous messages (thank you again Mark), looking over “Elements of Structured Finance” by Ann Rutledge and Sylvain Raynes, and analyzing the provided Excel spreadsheets, things slowly became clearer. As I have not taken an in-depth Structured Finance course before—I did take a Hybrid & Structured Products course but it never covered Asset-Backed Security pricing and courses I took on MBSs and ABSs were never of such depth—I decided to follow the provided Excel spreadsheets in detail (I did deviate slightly as I noticed some potential formula typos from comparing the formulas with the formulas given in “Elements of Structured Finance”.). Of course, doing so may have compromised on code efficiency among other things.

Part 3: Valuing the Structure wasn’t very challenging in the way Part 1 and Part 2 were, but was definitely quite challenging in terms of debugging and fixing code. When I initially built the classes in Part 1, I wasn’t sure of the purpose of the reset() method in the Tranche class. After debugging numerous times, due to outputted values not making much sense or behaving in a non-random way, I eventually realized that I had forgot to reset the values back to their original state in period 0 before/after each Monte Carlo simulation. In addition, because I made StructuredSecurities an iterable as well, I had to make sure to reset its index as well as LoanPool’s index to 0 after iterating them. Note that the code here may also be slightly inefficient—this is due to adding additional code to ensure that certain objects and/or variables got reset when it was necessary to do so, i.e., running additional Monte Carlo simulations.

The code for simulateWaterfall() global function was running quite slowly. As a result, I tried memoization. While it did speed things up very quickly (which I emailed Mark about), upon closer debugging, memoization was not working as expected. This led to implementing multiprocessing as otherwise, the code was just taking a VERY long time. Note that when I first tried implementing multiprocessing, I had similar issues to those listed near the beginning of the previous paragraph, which prompted me to temporarily halt it. After a few hours of debugging, I think everything is (hopefully) working fine now though the program is now quite slow even with multiprocessing (this is referring to Part 3). Note that by “quite slow”, I mean that the simulations still take quite a few hours (the precise number does depend on the computer and I got 406.4 s and 1860.5 s for smaller tests of 20 and 200 simulations, respectively). Depending on the most efficient approach, this may or may not actually be very slow (though I’m more inclined to think that it is very slow). Since the main objective of Part 3 is computing the yields, I have left simulateWaterfall() and doMonte() global functions commented out (expect them to take a VERY long time like a day or so to run based on timings for the smaller number of simulations). One remark that I do want to make is that usually, I would utilize numpy extensively in order to vectorize operations—doing so significantly speed things up; see <https://hackernoon.com/speeding-up-your-code-2-vectorizing-the-loops-with-numpy-e380e939bed3>. In this case, the only mentioned use of numpy is for IRR computations and so I didn’t try vectorization through numpy.

A (Very) Brief Manual

Andy Zhang

All scripts are in the *Case Study – Andy Zhang v2* folder. In Part 1, ABS\_pricer\_main.py script assumes that the data given is of the same format as Loans.csv (make sure that it’s .csv and not other types such as .xlsx). It reads each line after the first line of the file as a list of str, with each str item in the list extracted as a different component of appropriate type to different lists. These lists are then used to create the LoanPool object. Note that it is assumed that the Asset depreciation is 0. Next, the StructuredSecurities object is created using the LoanPool object’s *totalLoanPrincipal*() method and ‘pro rata’ or ‘sequential’ (it currently has ‘pro rata’ but this can be changed). With the StructuredSecurities object instantiated, two tranches A and B are added. *doWaterfall*() function is then called and the results are saved to ‘assets.csv’ and ‘liabilities.csv’ in the *Case Study – Andy Zhang v2* folder. In Part 2, the metrics IRR, DIRR, and AL as well as the letter rating are also computed for each tranche. Note that these previous results didn’t consider loan defaults. It is now incorporated. In Part 3, *simulateWaterfall()* global function runs *doWaterfall()* function 2000 times and then averages out the metrics for each tranche. *runMonte()* global function runs *simulateWaterfall()* global function to get average DIRR and WAL for each tranche and uses those to compute the corresponding yield and thus new tranche rates. This occurs repeatedly until there is convergence, i.e., the tranche’s old and new rates differ by less than the tolerance level. The resulting metrics for each tranche in the situation of convergence are then outputted. Note that this function is commented out because it can take a very long time the larger the number of simulations. *runMonte\_multiprocessing()* improves upon this issue by repeating the same calculations but with multiprocessing.