* No results ... They should appear in your report!!!
* Double toss exercice does not seem to be well understood...
* Not enough explanations regarding the Roulette Method...

Tientso Ning

Metaheuristics for Optimization

2

18 September 2019

TP0 Short Report

[SIMULATION OF BALANCED DICE]: The problem wants you to simulate a six-sided dice through usage of a random number generator (RNG). The theory is that you can generate a random integer [1,7) (essentially 1, 2, 3, 4, 5, and 6) many times, and record the number of times that each number is generated by your RNG. The setup of the first problem is done through Python. An array to store the occurrences is created, and every time the number is generated, the value at the index which corresponds to the dice roll is incremented up by 1 (to record). At the end of the number of rolls, the values in the array are each divided by the total number of rolls in order to evaluate the percent of occurrence. The code is attached and labeled, "tp_balanced_die.py" and running the code will verify the validity of the RNG, as the values are all around 0.16~0.17 or "One-Sixth," which is the proper odds of a balanced die.

I will not run your code to have a look at your results... In a report you must show your results using graphs/tables and comment them!

[SIMULATION OF A BIASED COIN TOSS]: The problem wants you to simulate a biased-coin, with bias *p*, through usage of a random number generator (RNG). The theory is that you can generate a random number between 0 and 1, and add that value to *p* and set the result as X after applying a floor function. Theoretically the RNG should generate a value between 0 and 1 (randomly), and thus adding p to the random number gives you either a value that is less than 1 or greater than 1. Taking the floor of this value (X) will give you either 0 or 1 (heads or tails). This suggests $p < 0.5$ means a bias for heads, and $p > 0.5$ means a bias for tails. In the attached Python code, "tp_biased_coin.py", a 10,000 (ten-thousand) run trial is used to verify that the p value should correspond to the percentage of tail occurrences observed.

You did not explain how λ and p are related...

[SIMULATION OF A DOUBLE BIASED COIN TOSS]: The problem wants you to simulate a fair-coin using a biased-coin, with bias p, through the usage of a random number generator (RNG). The theory is that you should flip the biased coin twice (or flip two biased-coins of the same bias) and observe the outcome. If the two flips are the same (double heads or double tails) then you disregard the flip since the coin is biased. If the two flips are different (heads and tails) then you have achieved the scenario where the probability of that occurring was p*1-p, and that outcome is equally likely despite the bias of the coin. In the attached Python code, "tp_double_biased_coin.py", you can see the number of tails and heads flips (only recorded if the double flips were different) as well as the number of flips "tossed away" (listed as throwaways). The probabilities of occurrence are also recorded, and they should both be around 50%.

[ROULETTE METHOD]: The problem wants you to simulate a roulette-board using a random number generator (RNG). The theory is that you can generate a list of random values, which correlates to the

percent chance that an event (for simplicity, labeled as the index of the array storing the random values) will occur. We can then create an array of cumulative probabilities, with each index storing the value of itself added with the value before it (in the case of the first index, the index stores its own probability value). The roulette board is now set up. We can use the RNG to generate a number between 0 and the total cumulative probabilities, called the 'roll'. We can implement a dichotomous search to search for a value in the cumulative probabilities array that is the smallest value still bigger than the 'roll' value. The attached Python code, "tp_roulette.py" implements this, and the verification is implemented by checking a random index of the recorded occurrences array to see if it is similar to the original probabilities array.