Subsampling leaf variables as an optimisation in Monte-Carlo simulations

Grant Holtes

November 28, 2023

Summary

This note shows how changing the sampling order in Monte Carlo simulations can significantly improve performance without materially impacting on the results. The conditions for this optimisation to be applied are defined and some practical examples are shown to illustrate the optimisation method.

1 Introduction

An issue with Monte Carlo simulation that draw on a large number of random variables is sparsity and the large number of samples required to provide adequate coverage of the high dimensional sample sample space from which values can be drawn, with each additional variable exponentially increasing the size of the "search space".
This issue can be partially addressed by simply making the simulation faster, allowing for more samples to be taken, filling in the search space. This note covers one approach that I have found useful, which manipulates the order in which distributions are sampled to reduce simulation runtime. The random variables in a Monte-Carlo simulation can be divided into 2 groups:

1. Random variables that affect the simulation flow, for example, by determining a logical path or influencing the simulation state, which I will refer to as "node" variables

2. Random variables that affect the outcome or result of the simulation but nothing else, which I will refer to as "leaf" variables

In most simulations, the majority of the compute time is tied to the "node" variables and their interactions with simulation state, while the leaf variables require relatively little compute to have their affects on the final outcome determined. In this setup, we can reduce latency by pausing the sampling of node variables while we sample leaf variables, before moving to the next sample of node variables and repeating the process.

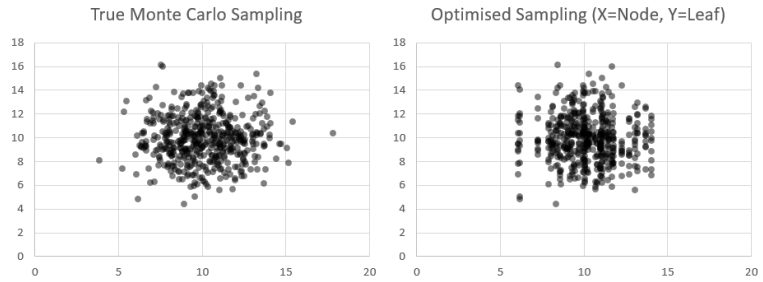

Figure 1: Comparison of samples generated by joint sampling compared to the order-optimised sampling, 2 variable case, 500 samples

2 A Simple Example

Consider a betting process by which the actor flips a coin to determine which of two horses to bet on. A single iteration of the simulation may involve:

1. Drawing a random variable to determine the coin flip outcome
2. Drawing many random variables to simulate the movement of the horses, decisions of the jockeys and the outcome of the race.

In this example, the race simulation is complex and affects both the outcome for the actor and the simulation itself, whereas the coin flip only affects the outcome. Due to this we can first simulate the race, then divide into two simulations, one for each of the equally likely coin flip outcomes which we evaluate. This has the effect of increasing our sample size without greatly increasing the computation required. If the actor rolled a dice or sampled from some other more complex distribution to determine which horse to back, we could accomplish the same result by sampling from this distribution a number of times for each simulated race, recording the outcome in each case.

3 Simulation termination use case

A more complex but useful use case is when a random variable has the effect of terminating the simulation. Consider a lifetime spending model, where each time period the agent has to determine how much to consume and how much to invest for their future, with some probability of death depending on age and other simulation states, with objective being maximisation of total lifetime consumption.
In this simulation, a true Monte Carlo implementation would be to draw from the death probability distribution each period (adjusting parameters based on age and state), and terminate the simulation iteration if the agent dies.
However, the death of the agent can be viewed as a leaf variable in this simulation, with a simple optimisation being:

1. Simulate the agent to an improbably high age, recording at each timestep the probability of death conditional on the current simulation state, and the simulation state itself.

2. Once the computationally intensive life situation process is complete, quickly iterate through the time steps and apply the death probability, computing the outcomes of each of these sub-simulations from the cached simulation state.

As long as an appropriately large number of "node" level samples are still taken, this should provide a close approximation for the true Monte Carlo simulation. If the number of node samples is not large enough, the samples will end up with a "striped" pattern as in Figure 2.

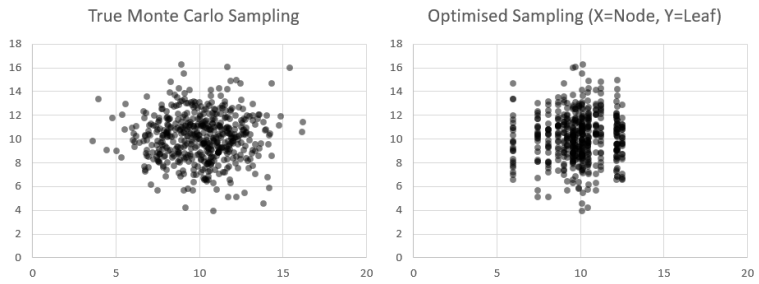

Figure 2: Comparison of samples generated by joint sampling compared to the order-optimised sampling, 2 variable case, 500 samples, 20 Node samples

4 Verification

A simple implementation of this example shows that this is the case:
True Monte Carlo implementation:

```
prob_death = [(1/120)*i for i in range(120)]

def life_simulation_simple(c, n):
    # c = Consumption rate
    outcomes = []
    for i in range(n):
        alive = True
        savings = 100
        spent = 0
        age = 0
        while alive:
            spent += c*savings
            savings = (1-c)*savings
            savings = savings * (1+random.random()/10)
            if random.random() < prob_death[age]:
                # actor dead
                outcomes.append(spent)
                alive = False
            age += 1
    return outcomes
```

Leaf variable optimisation implementation:

```
def life_simulation_opt(c, n, m):
    outcomes = []
    for i in range(n):
        savings = 100
        spent = 0
        age = 0
        death_probs_cache = []
        score_cache = []
        # Sample over node variables
        while age < 120:
            spent += c*savings
            savings = (1-c)*savings
            savings = savings * (1+random.random()/10)
            death_probs_cache.append(prob_death[age])
            score_cache.append(spent)
            age += 1
        # Sample deaths over ages - leaf variable
        for j in range(m):
            for death_prob, score in zip(death_probs_cache, score_cache):
                if random.random() < death_prob:
                    outcomes.append(score)
                    break
    return outcomes
```

For a test of 10,000 samples, which for the optimised version was divided into 100 node samples, each of which has 100 leaf samples, we get the following results with c=0.15:

Metric	True Monte Carlo	Optimised Monte Carlo	Interpretation
Mean result	104.31	104.65	Closer results are better
Variance in score	602.77	602.99	Closer results are better
Execution time (s)	0.0443	0.0180	Lower is better

Table 1: Optimisation verification results