Rapid Monte Carlo Simulation

Robert Chang, FRM and I give practical tips and methods to improve Risk Management through faster calculations.

Monte Carlo (MC) simulation is a widespread method one implements in computer code to create both financial and non-financial models. Despite an air of sophistication, MC may actually be the simplest numerical solution technique. The software developer instructs the computer to generate random numbers which then feed into calculations of financial outcomes such as loan defaults, yield curve movements, commodity prices, *et cetera*.

Virtually all computer languages support this capability. The MC concept fits extremely well with financial applications due to our perception that we can specify approximate *probability distributions* of future events. We write code that generates future market values in line with our prescribed distributions and let software calculations impose the complex interactions among the market values while managing large datasets.

Monte Carlo algorithms always need to be faster

An interesting distinction of MC is that it *always* needs to be faster! Many software calculations are "fast enough" in the sense that users might not even notice a doubling in speed. With MC there is always a trade-off between speed and precision of results. Thus, users tend to set the number of MC trials as a balance between good precision (many trials) and fast execution (fewer trials). When calculation speed increases, users often react by increasing the trials.

Additionally, when developers increase execution speed, users also respond by increasing the complexity of the simulation. Instead of computing risk parameters for bank sub-portfolios separately, a faster algorithm may prompt the user to run all bank portfolios together – which gives more meaningful results. This is progress! It means the quality of risk management improves as MC simulation speeds up.

Joe Pimbley

Almost always possible to speed the calculations

In the "old days" stretching back fifty years of writing computer code it was important to minimize use of memory and maximize calculation efficiency. While it clearly remains good practice to avoid waste of memory and processor time, the impetus is not as it was. The best and brightest of the software world developed brilliant memory conservation techniques that have become technology anachronisms with subsequent explosion of memory capacity.

Just as important, the nature of writing code, even at the spreadsheet level, is that one first simply proves the concept. The first version of any software seeks to discover whether it's workable to solve the user's problem in the chosen platform. If yes, then the next versions are for debugging and making the program properly handle input data errors. Just for the record, all software needs debugging in early development and all input data have errors. Building efficiency and speed into the code is generally a last step. It's not uncommon for users or developers to forego this final stage.

Simple and powerful example with improved code

Avenues to increase MC execution speed include improvements in hardware, software, and exploitation of the specific nature of the financial problem. Let's consider first an improvement in the software. See the snapshot below of a portion of VBA code for MC simulation of loan defaults across an arbitrary number of pools. (See this web page for explanation of the specific loan default investigation. Software in this web page is *Microsoft Visual Studio .Net*, but we produce the same algorithm in VBA to give this discussion.)

```
'   Generate Num_Pools to measure the default fraction of Number_Obligors
'   within each pool.  We provide a single-factor correlation and then
'   determine if the MLE extraction for correlation and Obligor default
'   probability works well.
For kount_Pools = 1 To Num_pools
    Def_number = 0
    '   Set the systemic random variable Pool_Y.
    Call Gauss_RV(G1, G2)
    Pool_Y = G1 * Sqr(rho)

    For kount = 1 To Number_obligors Step 2
        Call Gauss_RV(G1, G2)
        Ob_RV = Pool_Y + Sqr(1# - rho) * G1
        If Application.NormSDist(Ob_RV) < Def_prob Then Def_number = Def_number + 1
        Ob_RV = Pool_Y + Sqr(1# - rho) * G2
        If Application.NormSDist(Ob_RV) < Def_prob Then Def_number = Def_number + 1
    Next kount

    Def_fraction(kount_Pools) = CDbl(Def_number) / CDbl(Number_obligors)
    If Def_number = 0 Then
        x(kount_Pools) = Min_X
    Else
        x(kount_Pools) = Application.NormSInv(Def_fraction(kount_Pools))
    End If

Next kount_Pools
```

The code within the red bracket is most critical since the MC routine runs through this segment 10 million times with typical input values. All code outside this loop receives much less "traffic." Best practice, then, is to scrutinize each line of code to ask if that line must appear within the critical loop and, if so, to find the most efficient way to write that line. This code example has a few shortcomings. We highlight in yellow the worst of these. In order to compute the Normal Distribution Cumulative Distribution Function (CDF) – written mathematically as $\Phi(\ )$ – this code employs the Excel spreadsheet function for the Normal CDF. The Excel function is highly convenient and hence is appropriate for the first version of software. Replacing this Excel function by a publicly available Normal CDF subroutine speeds total program execution by a factor of 4! (See *Numerical Recipes in C* for the Normal CDF code we use.)

Even better, we can increase speed by another factor of 3 by eliminating the necessity of calculating the Normal CDF 20 million times. See the revised VBA code listing below:

```
Sq_rho = Sqr(rho)
Sq_wmrho = Sqr(1# - rho)
Phi_inv_Def_prob = Application.NormSInv(Def_prob)

'   Generate Num_Pools to measure the default fraction of Number_Obligors
'   within each pool.  We provide a single-factor correlation and then
'   determine if the MLE extraction for correlation and Obligor default
'   probability works well.
For kount_Pools = 1 To Num_pools
    Def_number = 0
    '   Set the systemic random variable Pool_Y.
    Call Gauss_RV(G1, G2)
    Pool_Y = G1 * Sq_rho

    For kount = 1 To Number_obligors Step 2
        Call Gauss_RV(G1, G2)
        Ob_RV = Pool_Y + Sq_wmrho * G1
        If Ob_RV < Phi_inv_Def_prob Then Def_number = Def_number + 1
        Ob_RV = Pool_Y + Sq_wmrho * G2
        If Ob_RV < Phi_inv_Def_prob Then Def_number = Def_number + 1
    Next kount

    Def_fraction(kount_Pools) = CDbl(Def_number) / CDbl(Number_obligors)
    If Def_number = 0 Then
        x(kount_Pools) = Min_X
    Else
        x(kount_Pools) = Application.NormSInv(Def_fraction(kount_Pools))
    End If

Next kount_Pools
```

The idea above is that we change the mathematics. Instead of writing an "if" statement to query whether the Normal CDF of a changing random variable is less than a fixed value (a default probability in this case), we re-write the math to ask if the changing random variable is less than the *inverse* Normal CDF of the fixed value. We do this inverse Normal CDF evaluation outside the critical loop where it adds essentially nothing to computation time.

Deeper methods employing Apache Spark and cloud computing

Robert Chang and I recently presented a webinar on this topic (available at this link). This lecture holds a longer discussion of software and also discusses applications of Apache Spark, adapting MC code to parallel processing, and loading simulations to cloud computing services for better hardware. GARP members earn CPD credit with this webinar!

Joe Pimbley, FRM is a financial consultant in his role as Principal of Maxwell Consulting, LLC. His expertise includes enterprise risk management, structured products, derivatives, investment underwriting,

training, and quantitative modeling. Follow this link to get Joe's address to request addition to his E-mail List for further articles and tutorials.

Joe Pimbley