Calibrating Heston & Others

Isaac Drachman

Calibration

I cleaned the data by removing ITM and no-bid options. I considered various loss functions: absolute % error, mean-squared $ error, and % options priced outside bid/ask. The first two are model price vs mid.

Monte Carlo instability makes gradient descent difficult. When perturbing the input, there is also a change in price due to the randomness of the MC. When this is large, the solver assumes a steep gradient and lowers the bump size until it runs out of machine precision. A solution I had is to fix the random variables across the fitting process. Then the loss for a given parameter set is constant between valuations. However, the MC error can still cause the difference in price between two parameter sets to be too large or in the wrong direction and halt the solver early. For the Heston model I used the Feller condition as a constraint and tried different initial conditions to avoid getting stuck in a local minimum.

I also used differential evolution — a genetic algorithm which randomly picks a "generation" of parameter choices, scores them, then merges & mutates them to create the next generation. DE doesn't attempt to compute a gradient and is very useful for problems with many local minima since it tests across the parameter space. The Scipy implementation doesn't support constraints, so for fitting Heston I added a catch which scores a very high loss on parameter choices which violate the Feller condition.

MC is the bottleneck for the calibration, since it runs (num_options x num_evaluations.) I re-wrote part of the MC in Cython to run at roughly 2.5x the speed.

Run Instructions

`python project.py [YYYY-MM-DD] [heston|gbmjd|oujd] [mse|abserror] [gd|de]`

Example: `python project.py 2019-12-06 heston abserror de`

The GD/DE flag is only supported for calibrating Heston. The other two use DE.

The program will save 2 files: 'prices' which writes the option data with each prediction and error, and 'results' which records the fitted parameters. The fits usually take <5m for GD and ~15m for DE.

The Cython can be compiled with: `python setup.py build_ext --inplace`

If you are unable to build the extension, the native python is commented in each generate function.

Results

I originally used 25 options (strikes 7500-9000) from 12/6 and fit with 75k MC paths & 100 time steps. Two optimizations using DE ran for each model, one minimizing absolute % error and the other mean-squared $ error, for comparison. I ran an additional two using GD for Heston. I kept the paths relatively low for runtime purposes. After seeing the initial results, I re-ran Heston on a subset (8000-8700) with 100k paths & 150 steps, which produced much better accuracy.

Heston Model: The comparisons and fitted parameters are documented in:

consolidated_heston_gd.xlsx	gradient descent (25 opts, MC=75k,100)
consolidated_heston_de.xlsx	differential evolution (25 opts, MC=75k,100)
consolidated_heston_new.xlsx	higher precision DE (18 opts, MC=100k,150)

"GBMJD" and "OUJD": These are processes I came up with to test my MC code. I modified them to be more reasonable by reducing the jump parameters to an intensity 'lambda' and a fixed size 'jump'. The results are in consolidated_GBMJD.xlsx and consolidated_OUJD.xlsx

Average absolute % error per model and loss function:

Avg. Abs. % Error	Minimize $ MSE	Minimize Abs. % Error
Heston GD	39.58%	23.69%
Heston DE	36.79%	21.05%
Heston DE new	9.69%	9.53%
OUJD	11.28%	17.44%
GBMJD	33.78%	20.91%

Overall the accuracy on the original settings is poor — the highest proportion of predictions within bid/ask was only 12%. Further-OTM options were the hardest to fit and there is likely a floor on accuracy related to the MC error. Accordingly, the tighter range / higher paths fit was much improved and there is more progress that can be made in that direction. In general, these results are a first pass solution and have room for improvement.

Heston was better fit by DE since GD is very sensitive to initial conditions and may have faced issues with MC noise. I was surprised by the "made-up" processes. The OUJD process likely does so well because it can fit the shape of short-dated skew using a low vol and adding down-jumps.

Potential Improvements

Faster MC — Re-writing more of the MC code in Cython can provide a significant performance uplift as the compiled code can run as fast as pure C. This would allow the use of more paths to stabilize the valuation and run longer optimizations.

Strike selection & Loss function — It might be useful to minimize loss on a quantity derived from the price of several options, for ex: the strip price of a variance swap. Raising the min delta of the selection improves the fit of meatier options. Perhaps the short-dated options price the wings with a factor (ex: jumps) that the Heston model doesn't capture.

Discretization — I found several solutions for discretizing the Heston model. I'm not sure what is used in practice but perhaps some are more convergent. I used full truncation to remedy negative paths of the discrete variance process — there might be a better technique. "GBMJD" is meant to evoke Jump Diffusion but is certainly far from it. I'd be interested to see how one actually discretizes and calibrates a JD model.

Optimization method — There might be a more apt solver or better configurations for the methods available in Scipy or other libraries. Running different strategies in sequence might be useful.