GARCH MODELS, NOISE, AND PARTICLE FILTERS: IN PURSUIT OF PARAMETERS

OMID KHANMOHAMADI

SUMMARY. The problem dealt with in this work is effective (parameter) estimation of a price model. This is of utmost practical importance as a more accurate estimation of parameters is equivalent to a more accurate prediction of the market. The underlying model is of general autoregressive conditional heteroscedasticity (GARCH) type, allowing the variance of the current error term to depend on those of previous time points. As always, to model the reality, "noise" must be taken into consideration, too. Presence of noise and nonlinearity of the model necessitate using a particle filter (of sequential Monte Carlo class), since Kalman filter cannot be used to estimate a nonlinear model unless, of course, it is linearized around some equilibrium state, which would lead to erroneous results as it is known that prices can diverge from their equilibrium values due to microstructure noises (e.g. illiquidity, asymmetric information).

The implementation is a combination of C and MATLAB. The particle filter is implemented in C and, through a `mexFunction` gateway, the compiled file (a `mex` function) is ported to the MATLAB part of the program which is an optimization routine. Several utility scripts, including `jobTasks` discussed in this report, have been developed to facilitate building applications, merging outputs and sundry other grid tasks.

CONTENTS

1. The GARCH Model 1
2. The Particle Filter 3
3. The Optimization Solver 4
4. Grid Implementation: 112 days is a very long time, compared to 4 days 4
4.1. `jobTasks`: Integrating Grid tasks in MATLAB 4
4.2. Internal Structure: `cell arrays` of Symbols 7
4.3. Customized `mexFunction` Gateway 7
Appendix A. A glimpse at parts of `jobTasks` 7

1. The GARCH Model

The title of this report begins with an esoteric term, GARCH. It is definitely fitting to define this term and the associated model. "In econometrics, an autoregressive conditional heteroscedasticity (ARCH) model considers the variance of the current error term to be a function of the variances of the previous time periods' error terms.

Date: October 19, 2009.
Key words and phrases. Particle Filter, GARCH, Noisy Model, Parameter Estimation.
Research under and supported by RMI. This is a joint work with J. C. Duan and A. Fulop.

It is employed commonly in modeling financial time series that exhibit time-varying volatility clustering, i.e. periods of swings followed by periods of relative calm." [1] Here is an example of an ARCH(q) model, depending on q previous time points,

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2.$$

"If an autoregressive moving average model (ARMA model) is assumed for the error variance, the model is a generalized autoregressive conditional heteroskedasticity (GARCH)." Here is an example of a GARCH(p, q) model,

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2,$$

where p is the order of the GARCH terms σ^2 and q is the order of the ARCH terms. "The efficient price model employed in this work follows a fat-tailed GARCH,

$$x_{t+1} = \mu + x_t + \sigma_t \epsilon_{t+1},$$

where the innovation has a generalized error distribution with unit variance and density

$$f(z) = \frac{v \exp\left[-0.5|\frac{z}{\lambda}|\right]}{\lambda 2^{(1+1/v)}\Gamma(1/v)},$$

where $\lambda = \left[2^{(-2/v)}\Gamma(1/v)/\Gamma(3/v)\right]^{1/2}$ and $v > 0$.
The measurement noise itself follows a GARCH process:

$$z_{t+1} = \delta \sigma_t \eta_{t+1}.$$

It is implicitly assumed that the signal-to-noise ratio, δ, is constant and the measurement error, η_{t+1}, is $GED(v_2)$. The observation, y_t, is the sum of the efficient price and the measurement noise:

$$y_t = x_t + z_t.$$

We looked at an extension of the NGARCH specification:

$$\sigma_{t+1}^2 = \alpha_0(1-\alpha) + \alpha_1 \sigma_t^2 + \sigma_t^2 \left[\beta_1 g_{t+1} + \beta_2(\eta_{t+1}^2 - 1) + \beta_3(\zeta_{t+1}^2 - 1)\right],$$

where g_{t+1} is the usual NGARCH innovation from the observed return, defined to have zero mean:

$$g_{t+1} = \left(\frac{y_{t+1} - \mu - y_t}{\sigma_t} - \gamma\right)^2 - (1 + 2\delta^2 + \gamma^2).$$

The parameter β_1 measures the effect of the overall observed returns on the volatility process, while β_2 measures the effect of transient measurement noises. Furthermore, we allow for an independent normal volatility innovation ζ_{t+1} through the parameter β_3. The persistence of the volatility process is α_1 while the unconditional variance is α_0.

[1] Some explanations about introductory concepts have been excerpted from Wikipedia, with the sole intention of making the report easily accessible to readers of a different background.

To ensure positivity and stationarity of the variance process the following parameter restrictions are used:

$$\begin{aligned} \alpha_0, \alpha_1, \beta_1, \beta_2, \beta_3 > 0, \\ \alpha_1 - \beta_1(1 + 2\delta^2 + \gamma^2) - \beta_2 - \beta_3 > 0, \\ \alpha_1 < 1." \end{aligned}$$

2. The Particle Filter

"Particle filters, also known as sequential Monte Carlo methods (SMC), are sophisticated model estimation techniques based on simulation. They are usually used to estimate Bayesian models and are the sequential ("on-line") analogue of Markov chain Monte Carlo (MCMC) batch methods and are often similar to importance sampling methods. Well-designed particle filters can often be much faster than MCMC. They are often an alternative to the Extended Kalman filter (EKF) or Unscented Kalman filter (UKF) with the advantage that, with sufficient samples, they approach the Bayesian optimal estimate, so they can be made more accurate than either the EKF or UKF. The approaches can also be combined by using a version of the Kalman filter as a proposal distribution for the particle filter. Sampling importance resampling (SIR) is a very commonly used particle filtering algorithm, which approximates the filtering distribution by a weighted set of P particles. Resampling is used to avoid the problem of degeneracy of the algorithm, that is, avoiding the situation that all but one of the importance weights are close to zero." Listing 1 is a snippet of the program responsible for resampling[2], one of the most important procedures in particle filtering:

```
void    Resample(vector<int>& resampled_indices,
                 const vector<double>& weights, int R){
  int M = resampled_indices.size();
  double s = 0;
  int j = 0;
  double ordered_uniform = j/(double)M + genrand()/M;
  for(int i = 0; i < R; i++){
    s += weights[i];
    while(ordered_uniform ≤ s && j < M){
        resampled_indices[j] = i;
        j += 1;
        ordered_uniform = j/(double)M + genrand()/M;
    }
  }
}
```

Listing 1. Resampling using stratified multinomial sampling

[2]Code by A. Fulop.

3. The Optimization Solver

First, the constrained, nonlinear optimization is done over all but the feedback parameters. In the process, the following steps are taken:

- Lower and upper bounds of the optimization domain are imposed
- Positivity bounds for noise standard deviation and GARCH parameters are imposed.
- Upper limit on Volitility Persistence is imposed.
- The parameter restriction presented at the end of Section 1 is imposed as a nonliear constraint.

The next step is the solution of the optimization problem using an Active Set algorithm, which is based on Karush-Kuhn-Tucker (KKT) equations. The KKT equations are necessary conditions for optimality for a constrained optimization problem. If the problem is a convex programming problem, then the KKT equations are both necessary and sufficient for a global solution point.
The KKT conditions can be stated using a system of three equations, in addition to the original constraints. The first equation describes a canceling of the gradients between the objective function and the active constraints at the solution point. For the gradients to be canceled, Lagrange multipliers are necessary to balance the deviations in magnitude of the objective function and constraint gradients. Because only active constraints are included in this canceling operation, constraints that are not active must not be included in this operation and so are given Lagrange multipliers equal to zero. This is stated in the last two Kuhn-Tucker equations. [3]

4. Grid Implementation: 112 days is a very long time[4], compared to 4 days

As the section title alludes to and Figure 1 depicts, the wall clock time of a particular run of the meta-NoisyGARCHv16 application with 100 worksunits is just over 4 days whereas the CPU time is over 112 days; cf. Figure 2. It is important to notice that the long CPU time of this application is, unlike those of other applications on the grid, a result of it being computationally intensive per se, as opposed to having been run with a huge number of workunits. Results based on less than 100 workunits would be statistically questionable and even with this minimum number, it is clear, from the 1/3 of a year CPU time, that this estimation problem is of great complexity and would be completely unwieldy without a grid implementation.
Figure 3 shows iterations of one of the estimated parameters, δ, for a number of different simulation path.
Below are a few creative design points implemented in this work.

4.1. `jobTasks`: Integrating Grid tasks in Matlab.

A fairly general M-script has been developed to facilitate several tasks associated with Grid jobs. Using this script, the user can remain in Matlab and, with not more than a few keystrokes and/or clicks of mouse, perform all of the following tasks, and more:

- Build the application module, compress the input files for the job and have the appropriate manifest automatically embedded if necessary;

[3] Matlab `fmincon` Active Set Algorithm.
[4] And so is 10^{435} years; cf. Cleve's Corner article "Random thoughts."

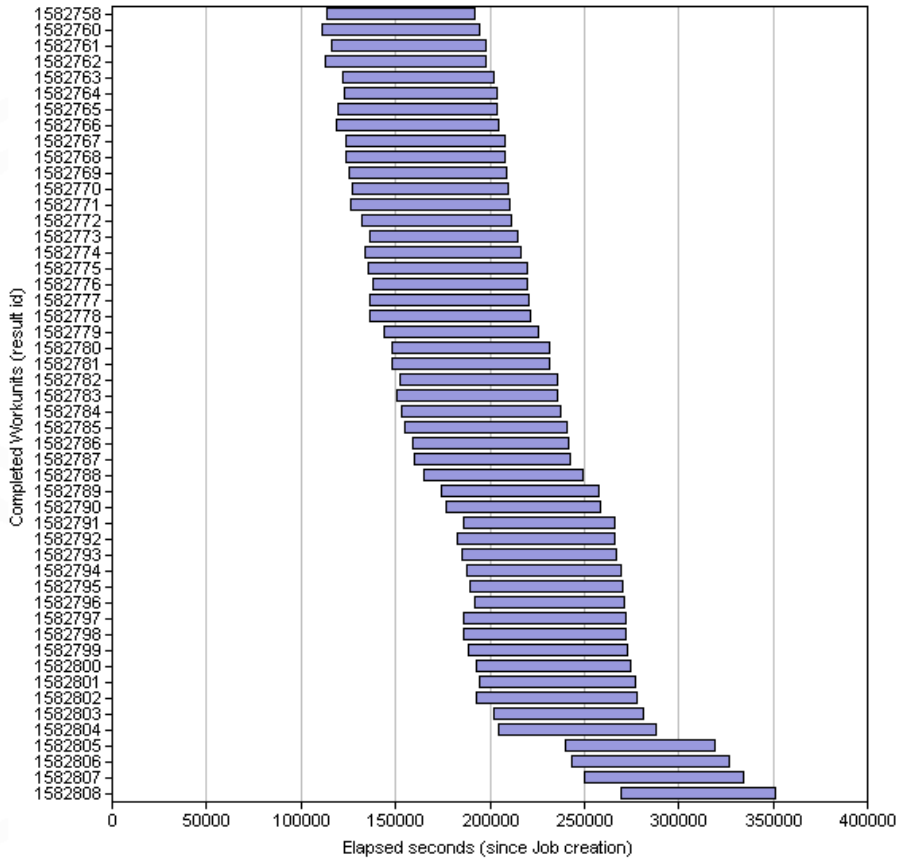


FIGURE 1. Workunit Timeline for a Particular Run of meta-NoisyGARCH (ver 16): Job 10226, Last 50 Workunits

- Submit/Retrieve the job and, if complying with the **`struct`** array data format in the utility, have the outputs merged together automatically.

Here are a couple of "one-liners" that can be used to perform the aforementioned tasks:

```
4 jobTasks('build')
5 jobTasks('zip')
6 jobTasks('submit','meta_NoisyGARCHv16','"NoisyGARCHv16, T=100"',100,{})
7 jobTasks('merge')
8 jobTasks('retrieve',9894)
9 jobTasks('kill',9894)
```

LISTING 2. Basic tasks in `jobTasks`

It is clear from Line 6 of Listing 2 that other options can be passed to `jobTasks`. For instance, if some fields of the output data structure are not required for a

Job

Job ID	**10226**
Description	NoisyGARCHv16, Inputs_T_8000
Application	meta-NoisyGARCHv16
State	**Completed**
Priority	10
Scheduled Execution Start Time	Not Specified
Scheduled Execution End Time	Not Specified
Creator	rmi
Created	2009-09-29 03:58:11 UTC
Last Modifier	rmi
Last Modified	2009-10-03 05:30:28 UTC

Job Statistics

Runnable	No
Last Result Time	2009-10-03 05:30:27 UTC
Last Dispatch Time	2009-10-02 12:07:14 UTC
CPU Time	112 days 18 hours 59 minutes 38 seconds

FIGURE 2. Job Statistics for a Particular Run of meta-NoisyGARCH (ver 16)

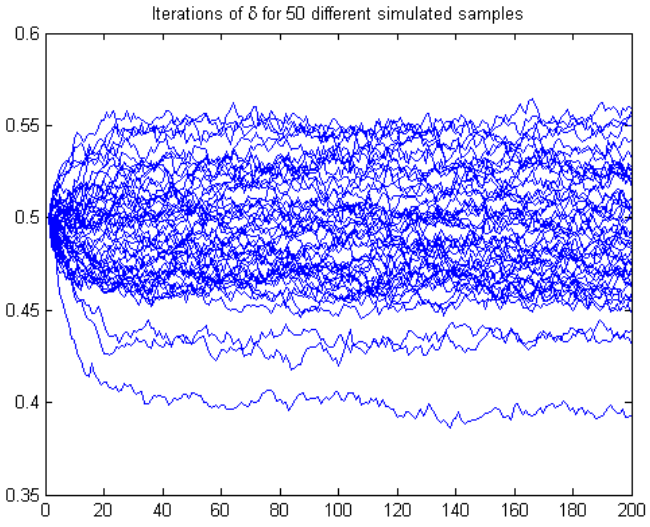


FIGURE 3. Iterations of δ for 50 different simulated samples

particular analysis, they can be very easily dropped, resulting in faster merging and a smaller aggregate output file.
An snippet of the `jobTasks` can be found in Appendix A.

4.2. **Internal Structure: `cell arrays` of Symbols.** Even with the ease that `jobTasks` provides, the task of recompiling the program and re-uploading the application for different sets of parameters to be estimated will soon become cumbersome. To circumvent this obstacle, the program has been designed in such a way that it can accept symbols and employ them in the right segment of its code. This way several basically different implementations of the program are essentially embedded in one executable.

4.3. **Customized `mexFunction` Gateway.** "For medium and especially for large sized arrays, the initialization [done by `mxCalloc`] voids the processor cache and blocks the system memory bus during cache flush. Hence, the data initialization tends to wipe previous results that are potential inputs to the function and delays the read-in of the input arguments. Modern processors have a theoretical peak memory-to-cache transfer rate of about 4GByte/s. This transfer rate is seldom achieved due to alternate reads and writes implying bus delays on each alternation. But a 2GHz processor has a floating-point multiplication throughput of about 1GFlops/s—hence an internal data throughput of about 24GByte/s (two double reads & one write per multiplication). It is clear that memory transfers need to be minimized in order to achieve top performance.

- The total number of data elements in the output array are determined.
- An empty real matrix is created. Functions of the family `mxArray` don't allocate memory for the data elements if the array is empty, i.e., the pointers on the real and imaginary parts are both set to null. This is a convenient way to avoid extra, time-consuming allocations.
- First the dimensions of the output array are set the dimensions of the input array and the first dimension is reduced to singleton. The data elements aren't affected on modifying the array dimensions.
- Finally, memory is allocated for the data elements and the corresponding data pointer(s) are set. In contrast to the array creation functions, this avoids the initialization of the output memory by calling `mxMalloc` instead of `mxCalloc`." [5]

The program has been designed with extensive use of vectorization and the C-MATLAB gateway avoids voiding the cache by allocating enough contiguous heap space without initialization of the corresponding block, according to the steps outlined above.

Appendix A. A glimpse at parts of jobTasks

```
function submitJob

runBatch    = [UD_Path 'setenv.bat'];
pyScript    = [UD_Path 'meta_driver' filesep 'meta_submit.py'];
```

[5] "C-MEX wrapper performance hints" by M. Leutenegger, with some modifications.

```
commandToRun  = [runBatch ' & ' ...
                 'python ' pyScript ...
                 ' -v -c ' description ...
                 ' -r ' outFileStem ' ' ...
                 appName ' ' ...
                 inpDir zipFilename]

if ~canContinue, canCont = false; return, end

[status,result] = dos(commandToRun,'-echo');

idxStartJobID   = strfind(result,pattern2find) + ...
                          length(pattern2find) + 1;

appName
description
jobID           = result(idxStartJobID:idxStartJobID + ...
                         nDigitsInJobID - 1)
submissionTime  = datestr(now, 'dd-mmm-yyyy HH:MM')

end

function mergeOutputs

ind2remove = [];
outNum = 1;
disp('Merging outputs')
for iFile = 1:nFiles
    gridFileName = [outFileStem '.' num2str(iFile,'%06d')];
    try
        load([outDir gridFileName],'-mat')
        fprintf('\n%s', [outDir gridFileName ' ... '])
        if ~isempty(fieldsToRemove)
            results = rmfield(results,fieldsToRemove);
        end

        if iFile == 1
            outputs = repmat(results,nFiles,1);
        else
            outputs(outNum) = results(:);
        end
        outNum = outNum + 1;
        fprintf('%s\n', 'Added.')
    catch ME
        ind2remove = [ind2remove; iFile];
```

```
        fprintf('\n\n%s',ME.message)
    end
end
outputs(ind2remove) = [];

fprintf('\n%s', ['Writing ' outDir outFilename ' ... '])
save([outDir outFilename],'outputs')
fprintf('%s\n', 'Done.')

end

end
```

Listing 3. Parts of `jobTasks` Utility

Risk Management Institute, National University of Singapore
E-mail address: Omid@NUS.edu.sg