Research Methods (MSc Programme), 2018

Introduction to MATLAB 2

Econometrics in MATLAB: ARMAX, pseudo ex-post forecasting, GARCH and EGARCH, implied volatility

Piotr Z. Jelonek

May 23, 2018

1 Modelling conditional mean: ARMAX

Let us go first through a bit of theory. Linear regression is a special case of AutoRegressive–Moving-Average (ARMA) model. In general, ARMA(p,q) model takes the form:

$$u_t = c + \sum_{i=1}^{p} \phi_i u_{t-i} + \varepsilon_t + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}, \qquad \varepsilon_t \sim IID(0, \sigma_\varepsilon^2).$$

Thus u_t depends on: its past values $u_{t-1}, u_{t-2}, ..., u_{t-p}$ weighted with, respectively, $\phi_1, \phi_2, ..., \phi_p$, and current and past residuals $\varepsilon_t, \varepsilon_{t-1}, ..., \varepsilon_{t-q}$, weighted with $\theta_0 = 1, \theta_1, ..., \theta_q$. While the past values of u_t are treated as known, the residuals are not observable and may only be estimated from the data. Two important simple cases of ARMA(p,q) model are:

$$\text{AR(1):} \quad u_t = c + \phi_1 u_{t-1} + \varepsilon_t, \qquad \text{MA(1):} \quad u_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1}.$$

ARMA model has two useful extensions.

First, we might explain the dependent variable with current or past values of additional exogenous variables. This extension is known as AutoRegressive–Moving-Average with eXogenous inputs (ARMAX). In example, if we added to our specification an exogenous variable v_t, a simple ARMAX(1,1) model could take the form of:

$$u_t = c + \phi_1 u_{t-1} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \beta v_t.$$

Next, we could estimate ARMA using differences of the dependent variable of order d. This extension is known as AutoRegressive Integrated Moving-Average and abbreviated to ARIMA(p,d,q). Since the series used in this class are already transformed to log-returns, we do not need any further differencing and thus we may set $d = 0$. In this case ARIMA(p,0,q) and ARMA(p,q) are the same model. ARIMA has also a more complicated variant that allows to capture seasonality. While it is available we will not be using it here (financial data is rarely seasonal).

Since MATLAB default function allows for the estimation of ARMA, from technical point of view what we will be estimating further will be ARIMA(X) models. But since we do not need to take differences of the dependent variable, in fact this will be ARMA(p,q) specifications. Let us look at some syntax.

In order to estimate ARMA(p,q) model in MATLAB you need to type:

```
Mdl = arima(p,0,q); % <- shorthand syntax, model with a constant
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
```

If you intend to estimate a model with no constant term, replace the last line by

$$Mdl = arima('ARLags', 1:p, 'MALags', 1:q, 'Constant', 0);$$

Above input variables p and q are just the desired lags of, respectively, *autoregressive* and *moving average* parts of the ARMA specification. Vectors $1:p$ and $1:q$ enumerate the desired lags (in the latter syntax you may specify separate lags instead of an entire range). Letter r stands for the underlying vector of the data (in my examples it will eb either returns or log-returns of Alphabet Inc. or Apple Inc.). **Mdl** is a *structure* which initializes the model setup.

Structure is just a number of variables of different types, bundled together. Structures have *fields*, which correspond to each separate variable. In order to access a chosen field refer to it, using name of the structure and name of this element, separated by a dot. In example, Mdl structure has field **P**, denoting the lag of the auto-regressive part. To access its value (and save it as small p) refer to it as:

```
p=Mdl.P
```

To view other fields of this structure, click it to inspect it in the *workspace* window.

Among the output variables **EstMdl** is a structure, which contains the estimated model. **EstParamCov** contains the estimate of covariance matrix of model parameters, **logL** referes to logarithm of model likelihood, **info** is yet another structure containing informations regarding convergence of the estimation process.

The following code selects the *best*[1] ARMA(p,q) specification for Alphabet Inc. log returns given $p \leq max\_p$ and $q \leq max\_q$:

```
% selecting best ARIMA(p,0,q)
max_p=5; max_q=5;
crit=zeros(max_p,max_q); c=0;
N=size(r,1);

for i=1:max_p
    for j=1:max_q
         Mdl = arima(i,0,j);    % <- shorthand syntax, model with a constant
        [EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
        [aic,bic]=aicbic(logL,i+j,N);

        if ((i==1) && (j==1)) || (bic<c)
            c=bic;              % <- change this line for AIC
            p=i;
            q=j;
        end

        crit(i,j)=bic;          % <- change this line for AIC
     end
end

% values of BIC criterion
fprintf('\nEstimates of BIC criterion: \n');
crit

fprintf('\n');

% estimating best ARMA
Mdl = arima(p,0,q);             % <- model with a constant
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
[res,v] = infer(EstMdl,r);      % <- ARMA(1,1) residuals
```

Above function **infer()** elicits model residuals and the estimate of their variance(s). You may use function **forcast()** to generate **h**-step ahead forecasts from a model given by **EstMdl** structure, using vector **r** as the past data:

```
[fcast,YMSE] = forecast(EstMdl,h,'Y0',r);
```

[1]It would be more interesting to investigate all possibilities by selecting separate lags.

Output variable **fcast** will contain the forecast, **YMSE** is a vector of the corresponding mean square errors. This example is provided as *'ARMA_selection.m'* script.

The script *'regARMA_selection.m'* selects the best model for the *regularized* ARMA. Hence instead of taking all the AR lags up to and including p and all the ma MA lags up to and including q we indicate which specific lags to include in the model specification.

```
% forecast horizon
h=6;

% selecting best regARIMA(p,0,q) model
max_p=3; max_q=2;

% loading data
data

% reverse order to chronological
g=flipud(g);
a=flipud(a);

% find log-returns
r_a=log(a(1:end-1))-log(a(2:end));
r_g=log(g(1:end-1))-log(g(2:end));

% select the series
r=r_g;

tic

% main body of the code
max_p=2^max_p-1; max_q=2^max_q-1;
crit=zeros(max_p,max_q); c=0;
N=size(r,1);

for i=1:max_p

    v = dec2ind(i); colsv=size(v,2);

    for j=1:max_q

        w = dec2ind(j); colsw=size(w,2);

        % Mdl = regARIMA('ARLags',v,'MALags',w);
        Mdl = regARIMA('ARLags',v,'MALags',w,'Constant',0); % <- model without intercept
        [EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
        [aic,bic]=aicbic(logL,colsv+colsw,N);

        if ((i==1) && (j==1)) || (bic<c)
            c=bic; % <- change this line for AIC
            best_i=i;
            best_j=j;
        end

        crit(i,j)=bic; % <- change this line for AIC
    end
end
v = dec2ind(best_i); % < -selected lags of AR part
w = dec2ind(best_j); % < -selected lags of MA part
clc; toc

% values of BIC criterion
fprintf('\nThe lowest attained value of BIC criterion: %10.2f\n',crit(best_i,best_j));
fprintf('\nSelected lags of AR part: \n'); v
fprintf('\nSelected lags of MA part: \n'); w

% estimating the best model
Mdl = regARIMA('ARLags',v,'MALags',w);
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
[res,var] = infer(EstMdl,r); % <- ARIMA residuals and variance (?)
fprintf('\n');

% forecasting (without the impact of the Exogeneous variable)
[fcast,YMSE] = forecast(EstMdl,h,'Y0',r(1:end,:));
```

2 Generating pseudo ex-post forecasts

In the *Home* tab select *Preferences*, go to *MATLAB* → *Editor/Debugger*→ *Display*, make sure that the *Show line numbers* option is ticked, confirm with *OK*. To see what a pseudo ex-post forecasting script *'Forecasting_ARX.m'* does, run it and inspect the graph.

Denote log-returns across h periods and (standard) log-returns as, respectively:

$$r_{t+h,h} \equiv \ln u_{t+h} - \ln u_t, \qquad r_{t+h} \equiv r_{t+h,1} = \ln u_{t+1} - \ln u_t.$$

There are two easy ways to forecast variable u_t indirectly through its log-returns. We can either specify the basic model as:

$$r_{t+h} = \alpha + \beta r_t + \varepsilon_t, \tag{1}$$

or, alternatively, we may write it down it using log-returns across h periods:

$$r_{t+h,h} = \alpha + \beta r_{t,h} + \varepsilon_t. \tag{2}$$

This script uses the first approach.

```
% DEFINE PARAMETERS
% forecats horizon (in periods)
h=5;

% number of forecasts
```

```
nf=250;

% rolling window length
w=100;

% lag order
l=1;

% scaling/stratching factor (for plotting charts)
scale=1.05;

% LOADING DATA
A = xlsread('it.xlsx');

% reverse the order of the data
g=flipud(A(:,1));
a=flipud(A(:,2));

% find log-returns
r_a=log(a(1:end-1))-log(a(2:end));
r_g=log(g(1:end-1))-log(g(2:end));

% data length
n=size(r_a,1);

% HERE WE WILL SAVE FORECASTS AND BENCHMARKS
fcast=zeros(nf,h);
bmark0=zeros(nf,h);
bmark1=zeros(nf,h);

% FORECASTING
for i=1:nf; % <- for a required number of forecasts

    first=n-(nf-i+1)-w-h+1;
    last=n-(nf-i+1)-h;
    y=r_a(first:last);

    for j=1:h % <- for up to h forecast horizons

        % BENCHMARK - AR(1)
        % select the data
        x=[ones(w,1) r_a(first-(h-j+1):last-(h-j+1),1)];
        % ols estimation
        b=(x'*x)\x'*y; % <- the same as: b=inv(x'*x)*x'*y;
        % take the last row of x's, multiply by betas
        f=x(end,:)*b;
        % save forecasts
        bmark1(i,j)=f;

        % MAIN MODEL - ARX
        % select the data; for l=1 the term (l-1)below can be removed
        x=[ones(w,1) r_a(first-(h-j+1)+(l-1):last-(h-j+1)+(l-1),1)...
        r_g(first-(h-j+1)+(l-1):last-(h-j+1)+(l-1),1)];
        % ols estimation
        b=(x'*x)\x'*y;
        % take the last row of x's, multiply by betas
        f=x(end,:)*b;
        % save forecasts
        fcast(i,j)=f;

    end
end

% HERE WE SAVE ACTUAL VALUES
actual=zeros(nf,h);

for i=1:nf; % <- for a required number of forecasts
    last=n-(nf-i+1)-h;
    for j=1:h % <- for up to h forecast horizons
        actual(i,j)=r_a(last+j);
    end
end

% FORECAST ERRORS AND ACCURACY
% errors
bmark0_errors=bmark0-actual;
bmark1_errors=bmark1-actual;
fcast_errors=fcast-actual;



% standard deviations in different horizons
bmark0_std=zeros(1,h);
bmark1_std=zeros(1,h);
fcast_std=zeros(1,h);

% standard deviations in different horizons
for i=1:h
    bmark0_std(1,i)=std(bmark0_errors(:,i));
    bmark1_std(1,i)=std(bmark1_errors(:,i));
    fcast_std(1,i)=std(fcast_errors(:,i));
end

% forecast accuracy
fprintf(1,'Comparing accuracy of the model with AR(1)\n');
```

```
fcast_acc=fcast_std./bmark0_std
fprintf(1,'Comparing accuracy of the model with naive forecasts\n');
fmark1_acc=bmark1_std./bmark0_std

% FORECASTED ASSET PRICES
P0=a(end-h-nf+1:end-h);

fcast_prices=[P0 exp(fcast)]; % <- this is exponent element by element
for i=1:nf
    for j=2:h+1
        fcast_prices(i,j)=fcast_prices(i,j)*fcast_prices(i,j-1);
    end
end

% PLOTTING THE FORECASTS
plotfcast(a(end-300:end),fcast_prices,scale)
```

The definition of **plotfcast()** function (saved as: *'plotfcast.m'* file):

```
function plotfcast(y,fcast_prices,scale)
% This is my function which plots the forecasted trajectories.

n=size(y,1);
nf=size(fcast_prices,1);
h=size(fcast_prices,2)-1;

x=ones(n,1);
for i=2:n
    x(i)=i;
end

plot(x,y)

hold on;
for i=1:nf
    plot(x(n-h-(nf-i):n-(nf-i)),fcast_prices(i,:),'b');
end
hold off;

xlabel('TIME (IN TRADING DAYS)');         % <- label on the x axis
ylabel('PRICE OF ALPHABET INC. SHARES');  % <- label on the y axis

axis tight;
axis([min(x) max(x) (1/scale)*min(y) scale*max(y)]); % <- defining range of x and y

end
```

To forecast from a different model (e.g. ARMAX) all that you need to modify is lines 77-84 in the *Forecasting_ARX.m* script. The graph of the generated forecast trajectories is presented below.

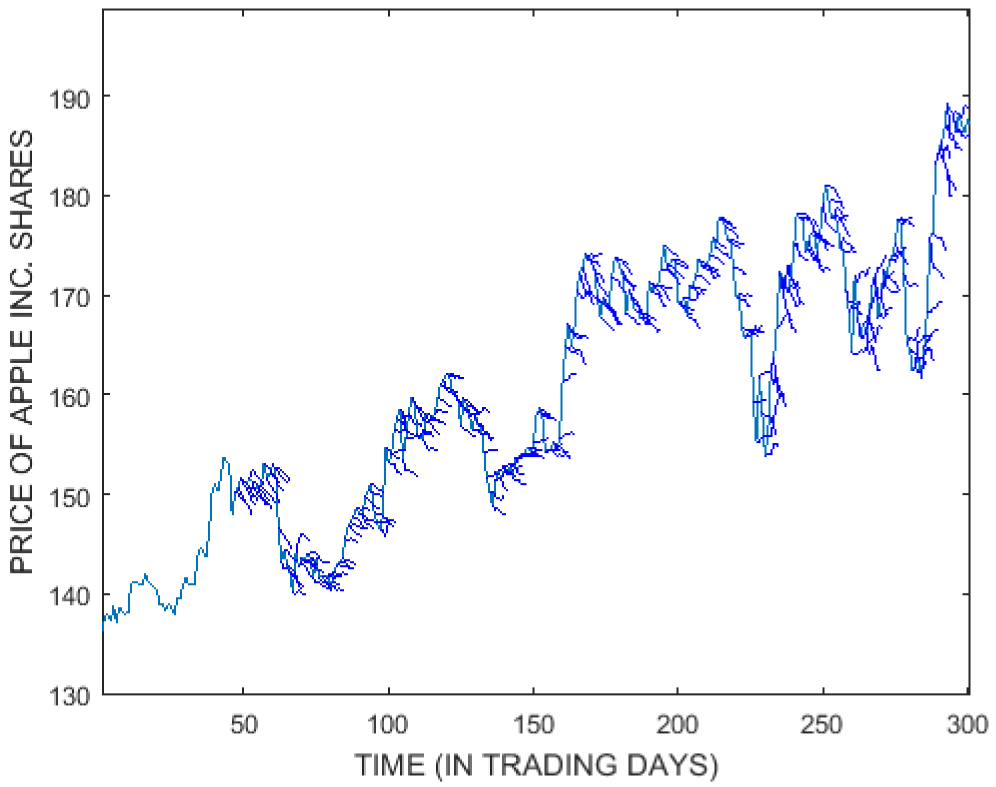

Figure 1: *Pine in the wind*: ex-post Alphabet Inc. share price forecasts (pretty bad, as you see).

In an attempt to generate better forecasts you may try to change the number of forecasts (**nf=250**), the length of the rolling window (**w=100**) or the forecast horizon (**h=5**).

Why is it useful: Contrary to what you might think, most models are bad at forecasting. Good sample statistics might simply signalize **overfitting**. Statistical significance of model coefficients does not yet imply economic significance – since estimates of the coefficients my be tiny and thus do not tell us anything useful about the future. From a point of view of statistics a single forecasted trajectory is equivalent to a sample of size 1 – so it is useless when we are trying to evaluate properties of trajectories, generated by the model. It is hard to evaluate if the model is good or bad without an appropriate point of reference (a **benchmark**). *Pine on the wind* chart allows us to assess forecasts qualitatively (are we getting the trend/turning points right).

In the *'Forecasting_ARX.m'* script I provide a model with a slightly different specification which includes a moving average component. With this one we will generate a smaller number of forecasts – otherwise it takes about 2 hrs to run. You may also run a *'Nasdaq.m'* script which provides another (negative) example of an ARMA forecast. This last bit of code was extracted from the MATLAB help.

3 GARCH and EGARCH models, simulations

Asset returns display an empirical features known as **volatility clustering** and **heavy tails**. Volatility clustering: disturbances u_t of large absolute value tend to be followed by further large absolute values, but not necessarily of the same sign. Heavy tails: extreme observations (very large or very small) happen more often than it would be implied by the fitted normal distribution. Likely explanation is *behavioural.* If investors are anxious and worried, market volatility is large, if they are calm and relaxed, it is small. Panicked or excited investors are more likely to imitate each other. In result market undergoes either more turbulent and volatile or more tranquil periods. Since investors' attitude does not (typically) change overnight, volatility is **persistent**. We may use this observation to model volatility as **unobserved** variable. This variable will be changing with time, depending on the past data. It is useful since volatility is to some extent **forecastable**.

Definition 1 (Volatility) *Standard deviation of market returns, conditional on past data. It is equal to (positive) square root of conditional variance. We will denote it as* $\sigma_t \equiv \sigma_{t|t-1}$.

ARCH/GARCH models represent volatility as unobserved variable, depending on the data from $t-1$. In result conditional standard deviation (thus also conditional variance) will be changing over time. These models also explain heavy tails.

Idea of ARCH: conditional variance depends on the squared past residuals.

Idea of Generalized ARCH: it *also* depends on its own past values.

The basic GARCH(1,1) model is

$$\sigma_t^2 = \mathbb{E}[u_t^2|I_{t-1}] = \alpha_0 + \alpha_1 u_{t-1}^2 + \beta_1 \sigma_{t-1}^2.$$

while the full GARCH(p,q) specification:

$$\sigma_t^2 = \mathbb{E}[u_t^2|I_{t-1}] = \alpha_0 + \sum_{j=1}^{q} \alpha_j u_{t-j}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2.$$

To estimate GARCH(p,q) use the following syntax:

```
Mdl = garch(p,q);
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,r);
```

Above input variables p and q are just the desired lags of, respectively, *garch* and *arch* parts of the garch specification. Letter r stands for the underlying vector of the data. **Mdl** is a *structure* which initializes the model setup.

To find conditional variance (take its square root to obtain a volatility) and to forecast the series use:

```
[V,logL] = infer(EstMdl,r);
fcast= forecast(EstMdl,10,'Y0',V);
```

Among the output variables **EstMdl** is a structure, which contains the estimated model. **EstParamCov** contains the estimate of covariance matrix of model parameters, **logL** referes to logarithm of model likelihood, **info** is yet another structure containing informations regarding convergence of the estimation process.
Simulation of ARMA(1,1)-GARCH(1,1) model:

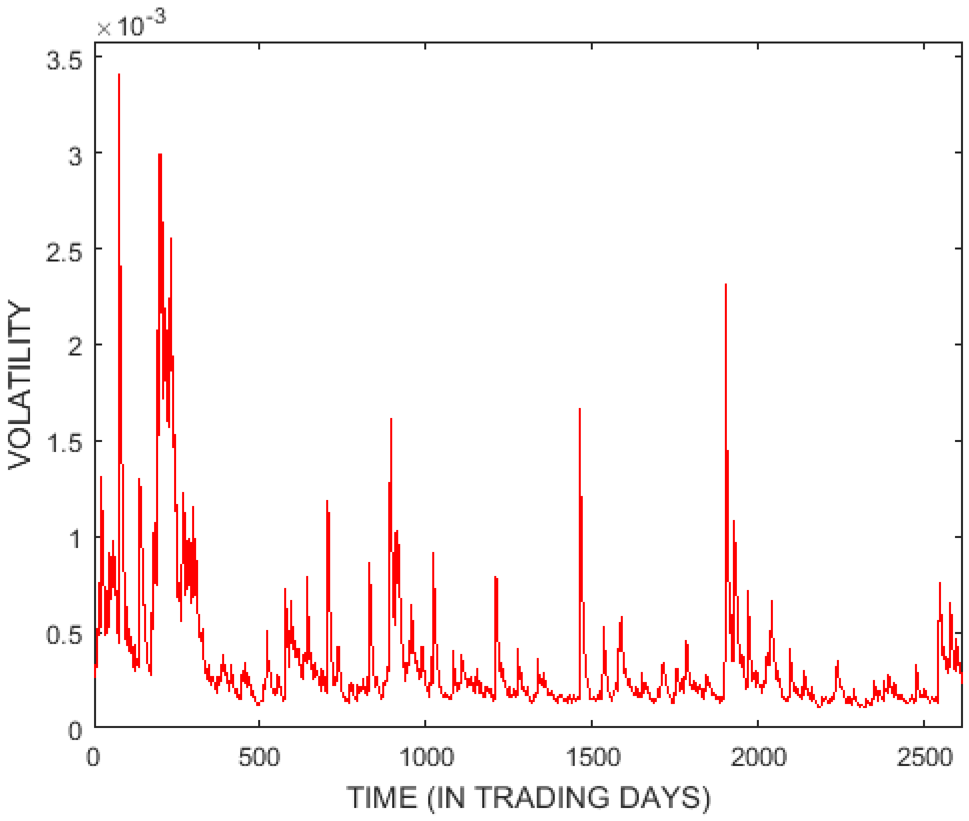

Figure 2: Estimate of volatility, obtained from GARCH(1,1).

```
N=size(g,1);
z=[0; randn(N-1,1)];
lagz=[0; z(1:N-1,1)];
r_sim=zeros(N,1);
s2_sim=[sigma_0^2; zeros(N-1,1)];

for i=2:N
    s2_sim(i,1)=beta_0+s2_sim(i-1,1)*(beta_1+beta_2*z(i-1,1)^2);
    r_sim(i,1)=alpha_0+alpha_1*r_sim(i-1,1)+sqrt(s2_sim(i,1))*z(i,1)+...
    +alpha_2*sqrt(s2_sim(i-1,1))*z(i-1,1);
end
```

ARMA is a model of conditional mean. GARCH – of conditional volatility. To represent the dynamics of the series we typically need both. The stimation and simulation of the ARMA(1,1)-GARCH(1,1) model is provided as *'GARCH.m'* script.

Problem with forecasting volatility – it is unobservable. To evaluate forecast quality we need a proxy. We may either use squared empirical returns/residuals or the high-low proxy, developed by Parkinson (1980)

$$\sigma_t^2 \approx \frac{(\log H_t - \log L_t)^2}{4 \log 2}$$

Alternatively, you might use VIX style index (VIX is calculated for S&P500), although it might be labour consuming (for more details see the next section).

Why is it useful: ARMA model represents conditional mean, GARCH/EGARCH conditional standard deviation. To depict dynamics of given financial time series we need both. It may be difficult to have any meaningful intuitions about parameters of non-linear models (especially is you are estimating model of this kind for the first time). Simulation of the corresponding trajectories (and comparing them with the actual realizations) allows us check if parameters make sense (and if the model is not misspecified).

You may also try *'regARMA_GARCH.m'* ans *Simulate_and_forecast_GARCH* script, the latter uses the predefined MATLAB functions in order to forecast volatility.

Exponential GARCH (EGARCH) model belongs to the class of nonlinear, assymmetric GARCH models. These models account for **asymmetry** of conditional volatility. This asymmetry reflects the fact that positive and negative innovations do not generate the same standard deviation.
The basic EGARCH(1,1) model is:

$$\ln \sigma_t^2 = \omega - \alpha\, \mathbb{E}|\varepsilon_{t-1}| + (\alpha\, |\varepsilon_{t-1}| + \gamma\, \varepsilon_{t-1}) + \beta \ln \sigma_{t-1}^2$$

where $\varepsilon_t \sim \text{NID}(0,1)$ and $\mathbb{E}|\varepsilon_{t-1}| = \sqrt{\frac{2}{\pi}}$. Main features:

- if $\varepsilon_{t-1} = 1$, log of variance increases by $(\alpha + \gamma)$ (in comparison to $\varepsilon_{t-1} = 0$)
- if ε_{t-1} is equal to -1, log of variance increases by $(\alpha - \gamma)$
- ... so the relationship between log of variance and past returns is **asymmetric** and piecewise linear
- since investors react stronger to losses than to gains, we would typically expect $\gamma < 0$
- even if log of variance is negative, exponent of log of variance is positive
- ... so there is **no** need for **non-negativity** conditions

The full EGARCH specification: EGARCH(p,q) developed by Nelson (1991):

$$\ln \sigma_t^2 = \omega + \sum_{j=1}^{q} \Big(\alpha_j (|\varepsilon_{t-j}| - \mathbb{E}|\varepsilon_{t-j}|) + \gamma_j \varepsilon_{t-j} \Big) + \sum_{i=1}^{p} \beta_i \ln \sigma_{t-i}^2, \tag{3}$$

where $\varepsilon_t \sim \text{NID}(0,1)$ denotes innovations and $u_t = \sigma_t\, \varepsilon_t$.

- negative innovations may have a **bigger impact** on future variance than positive innovations of the same magnitude (erroneously: *"leverage effect"*)
- exponent of log is always positive, **no** need for **non- negativity** conditions
- natural logs make construction of **unbiased** volatility **forecasts** more difficult

You may also try *'regARMA_EGARCH.m'* ans *Simulate_and_forecast_EGARCH* script, the latter uses the predefined MATLAB functions in order to forecast volatility.

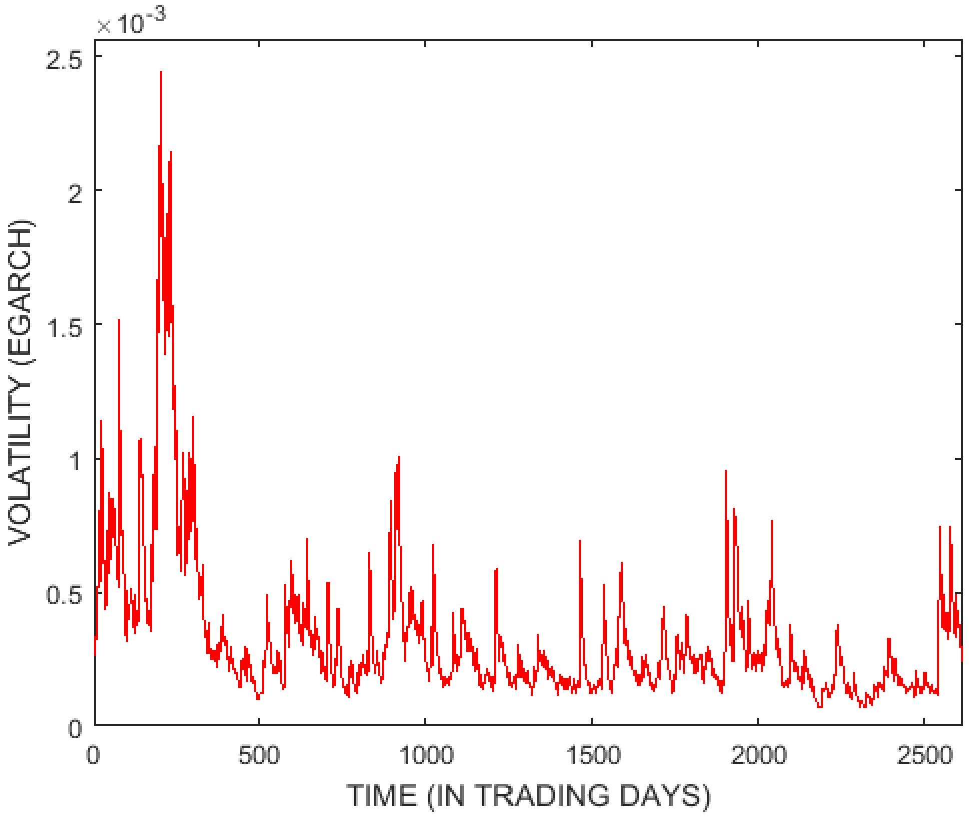

Figure 3: Estimate of volatility, obtained from EGARCH(1,1).

4 Implied volatility

Definition 2 (Implied volatility) *Volatility of the (future) stock prices which yields theoretic option prices (e.g. coming from the Black-Scholes formula) equal to recorded market transaction prices.*

The script *'IV.mat'* calculates *implied volatility* using put option prices (transaction level data) and risk-free interest rates (obtained from futures prices). This code uses a bisection method in order to calculate the level of volatility (σ) which – according to the Black-Scholes formula – would generate the option price recorded in a given transaction. Then the sigmas obtained for all the options traded on a given day are weighted (using transaction volume) to obtain a daily proxy of volatility.

In the code below **pwd** ('print working directory') command produces a **string** (a sequence of characters) which corresponds to current directory. The syntax **[pwd,'\']** appends a single character ('\') to the string, representing the working directory. Hence the command:

```
currentFolder=[pwd,'\'];
```

outputs the current folder. This is later used to create the paths to the files with put option prices. Command **run()** runs the file saved under the indicated path. The command:

```
filename1=names1{i,1};
```

accesses the i-th row, 1-st column of *names1* **cell** and saves it as *filename1*. The cells are MATLAB data structures similar to matrices, but their entries are not required to be numerical. In these case – they are *strings*, listing the names of all the files, containing put option prices. In order to access elements of a cell, we nee to use curly ('{','}') brackets.

```
% fixed parameters
global small_number;
small_number=0.00000001; % <- smallest increment allowed for

% setting folders
currentFolder=[pwd,'\'];
optionsFolder=[currentFolder,'put_options','\'];
futuresFolder=[currentFolder,'futures','\'];

% reading lists labels of excel files
run([currentFolder,'labels_of_put_options_files.m']);
run([currentFolder,'labels_of_futures_files.m']);

% loading data
filename='mibid_and_mibor.xlsx';
[NUM,TXT,RAW]=xlsread([currentFolder,filename]);

% extracting dates and interest rates
interest=NUM(:,[7 11 15]);
dates=datenum(RAW(4:end,1),'dd/mm/yyyy'); % <- conversion of date
first_date=dates(1); last_date=dates(end);

% defining outputs
ivalues=[];
output=[];
count1=0; count2=0; flag=0;

for i=1:size(names1,1) % <- number of labels, in this case equals to 59

    % loading options prices from the files
    filename1=names1{i,1};
    [NUM,TXT,RAW1]=xlsread([optionsFolder,filename1]);

    % processing options
    n=size(NUM,1);
    strike=NUM(:,1);
    call_price=NUM(:,7);
    trading_volume=NUM(:,9);
    open_interest=NUM(:,11);
    option_date=datenum(RAW1(2:end,2),'dd/mm/yyyy'); % <- former t0
    expiry_date=datenum(RAW1(2:end,3),'dd/mm/yyyy');
    dt=(expiry_date-option_date)/365;
```

```
% loading futures prices from the files
filename2=names2{i,1};
[NUM,TXT,RAW2]=xlsread([futuresFolder,filename2]);

% processing futures
future_price=NUM(:,6);
future_date=datenum(RAW2(2:end,2),'dd/mm/yyyy');

% defining outputs
iv=zeros(n,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% MAIN LOOP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
m=1; % <- row with selected call opiton
while m<n
    t=option_date(m);
    T=expiry_date(m);

    if t>=first_date && t<=last_date   % <- if the interest rates are available ...

        % finding future prices
        if any(t==future_date)==1
            f=mean(future_price(future_date==t));
            count1=count1+1;
        else
             % <- since there is only one entry of future price missing,
             % we plug the div'd yield for 25 Sep, 2014 in: f=s*exp((r-q)*t)
             f=7911.85*exp((0.0894-0.0133)*90/365);
             count2=count2+1;
        end

        k=strike(m,1);
        moneyness=k/f;

        % <- OTM need robustness check ie. at 95% significance level;
        % source:option metrics
        if  moneyness<=0.98 && moneyness>=0.9

            if T-t<30
                h=1;
            elseif (T-t>=30 && T-t<60)
                h=2;
            else
                h=3;
            end

            id1=0;
            while (dates(id1+1)<=t) && (id1+1 < size(dates,1))
```

```
                     id1=id1+1;
                end
                id2=size(dates,1);
                while (dates(id2-1)>=t) && (id2-1>1)
                    id2=id2-1;
                end

                % finding interest rate
                if id1==id2
                    r=interest(id1,h);
                else
                    % use linear interpolation
                    w1=(dates(id2)-t)/(dates(id2)-dates(id1));
                    w2=(t-dates(id1))/(dates(id2)-dates(id1));
                    r=w1*interest(id1,h)+w2*interest(id2,h);
                end
                r=0.01*r;

                d=dt(m);
                p=f*exp(-r*d);
                c=call_price(m);

                if c>=(k*exp(-r*d)-p)
                    v_p = cal_iv_p(k,c,d,r,p);
                    iv(m,1)=v_p(1);
                end
                oi=open_interest(m);
                tv=trading_volume(m);
                if ~isnan(v_p(1))
                    if tv>0
                        % data for analysing the relationship, list after commas
                        output=[output;t,T,v_p(1),oi,tv];
                    end
                end
            end
        end
        m=m+1;
    end
    ivalues=[ivalues; iv];
end

% output of the approximation
fprintf(1,'Percentage of cases when future prices are used: %4.3f',...
100*count1/(count1+count2));
z=ivalues(ivalues~=0);
z1=z(isnan(z));
z2=z(~isnan(z));
```

```
size(z2,1);
fprintf(1,'\nTotal number of approximated volatilities: %i',...
size(z2,1));
fprintf(1,'\nPercentage of cases when numerical approx. fails: %4.3f',...
size(z1,1)/size(z2,1));
fprintf(1,'\n');

% saving the output
xlswrite('put_iv.xlsx', output);

% elciting the aggregates
got=0; x=[];
while got<size(output,1)
    y=output(output(:,1)==output(got+1,1),:);
    x=[x; output(got+1,1) y(:,3)'*y(:,5)/sum(y(:,5))];
    got=got+size(y,1);
end
x=sortrows(x,1);
```

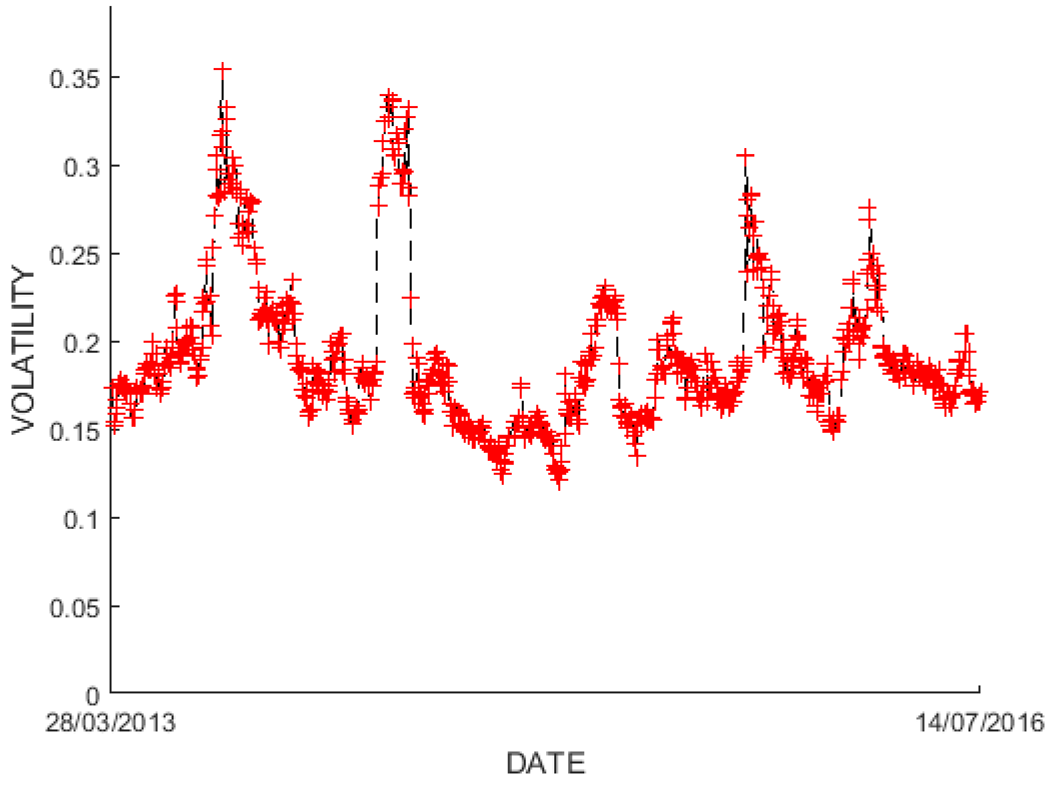

Figure 4: Implied volatilities (volume-weighted) implied by put option prices (NIFTY index)

5 Best coding practices

- use different naming conventions for scripts and function (the names of my scripts always begin with a capital letter, hence you always know which file is executable)
- make sure that running a script does not require any manual operations (like: separate loading the data), all the scripts should work on *plug and play* basis (it makes your work reusable, otherwise after a few months you will not remember how to use the script)
- in general: any operation which you will be doing 3 or more times should be *automated*
- if there are any limitations in using a script, or if you need to do any operations to run the script (which for some reason can not be automated), comment upon it in header of the script (then it is easier to do if you go back to it some time later)
- if you think it might be helpful later on, add comments to the code (again it increases its reusability: the code will be easier to modify later)
- in the header of a script explain what does the script actually do
- any complicated block of operations which has to be repeated again and again is best to be written as a separate function
- in the header of the function explain what does the function do, list the functions inputs (and their type) and its outputs (and their type)

That is all for today. Thank you!