Bayesian Time Series Model: the Application of **PyMc 3** Package

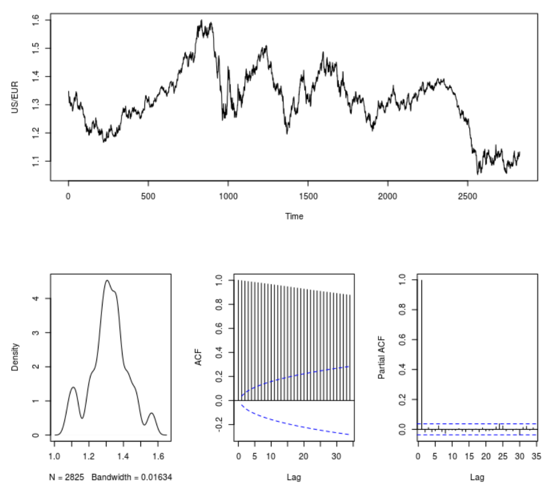
Qi Zhang, Ziwei Zhou, Songzhu Zheng

April 11, 2016

1 DATA[1]

Data is FX rate of USR/EUR from 2005-01-01 to 2016-03-30. With the data, we compute the log return and substracted the mean. The out put given by *stochvol* package is presented as below:

Figure 1: Property plot of Data after treating



The return presents ,to some extent, white noise property, but might be able to see some pattern exist in the volatility part.

2 PyMc 3[2]

PyMc 3 is a python module for Bayesian statistical modeling and model fitting which focuses on advanced Markov chain Monte Carlo fitting algorithms. Its flexibility and extensibility make it applicable to a large suite of problems.While most of **PyMc 3**'s user-facing features are written in pure Python, it leverages Theano to transparently transcode models to C and compile them to machine code, thereby boosting performance. [1]

3 AR(1)

$$
\begin{aligned}
r_t &= \beta + \alpha r_{t-1} + \epsilon_t \\
\epsilon_t &\sim \mathcal{N}(0, \sigma) \\
r_t | r_{t-1} &\sim \mathcal{N}(\phi_0 + \phi_1 r_{t-1}, \sigma) \\
\beta &\sim \mathcal{N}(0, 0, 001) \\
\alpha &\sim Uniform(-1, 1) \\
\sigma &\sim \mathcal{IG}(2, 3)
\end{aligned}
$$

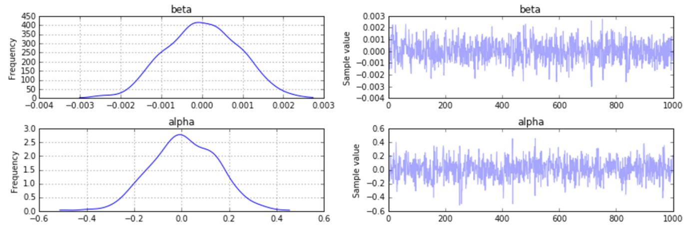
We don't have specific reason to choose the hyperparameters of the prior distribution, so we choose very noninformative prior and since we have quite a long time series, the chosen prior would not leave too much effect on the final inference.

[1] http://pymc-devs.github.io/pymc3/getting_started/

3.1 Posterior of Parameters

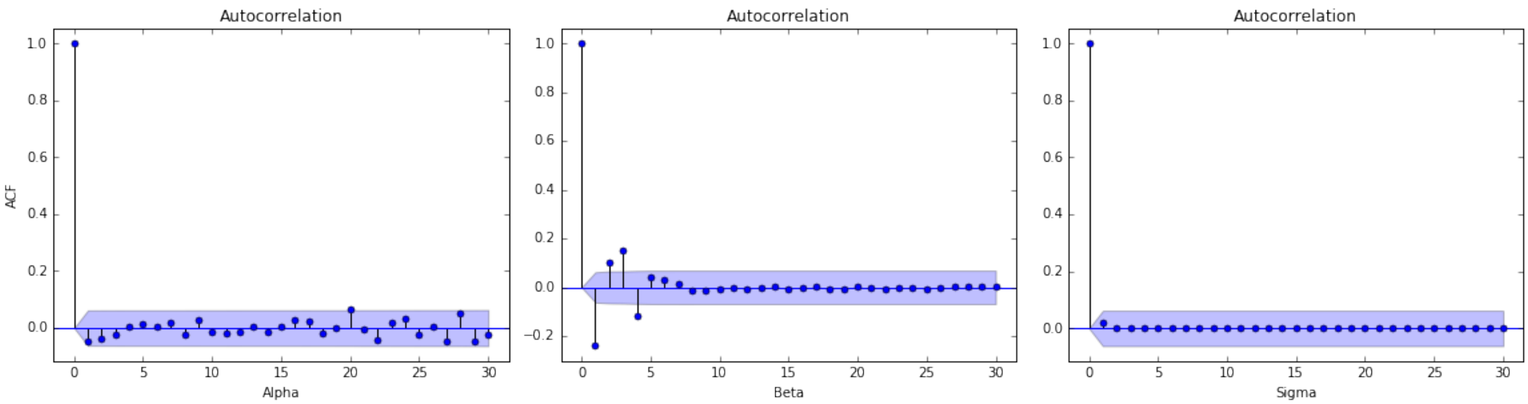
With **PyMc 3** pacakges, we install a MCMC with NUTS sampler, and gain the posterior distribution as given below:

Figure 2: Posterior Distribution AR(1) Model



From the trace plot we could see that the chain mixs quite well, and from the ACF plot of the draw, there is not obvious autocorrelation among samples, so the sample is quite good.
The ACF is presented as below:

Figure 3: ACF of Posterior Draw



We could see the mixing is quite well from the ACF.
And the estimate is presented as below:

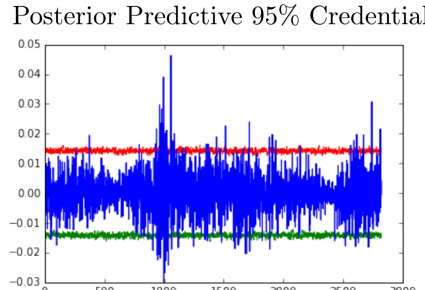
Table 1: Estimated Parameters

Parameters	Median	Mean	ESS(based on 1000 posterior draw)
β	0	0	2034.1
α	0.011	0.009	2354.6
σ_0	0.007	0.007	1297

3.2 Good of Fit

To check the good of fit of the model, we draw samples from our posterior predictive distribution, and construct a 95% credencial interval and the cover rate is 96.21%. The cover rate of our credencial interval is presented as below:

Figure 4: Posterior Predictive 95% Credential Interval



3.2.1 Computing Time

The sampling and calculation time for AR(1) model in **PyMc 3** is 1.6 seconds, while it takes 76 seconds including initialization and warm up with Rstan.
For different sampler, the computing time is different. Theoritically, NUTS or HamiltonianMC should be the most efficient sampler, however, they take relatively longer time to initialization.

Table 2: Computing Time over Different Sampler

	HamiltonianMC	Slice	Metropollis	NUTS
Time(sec)	0.9	1.5	0.2	0.4

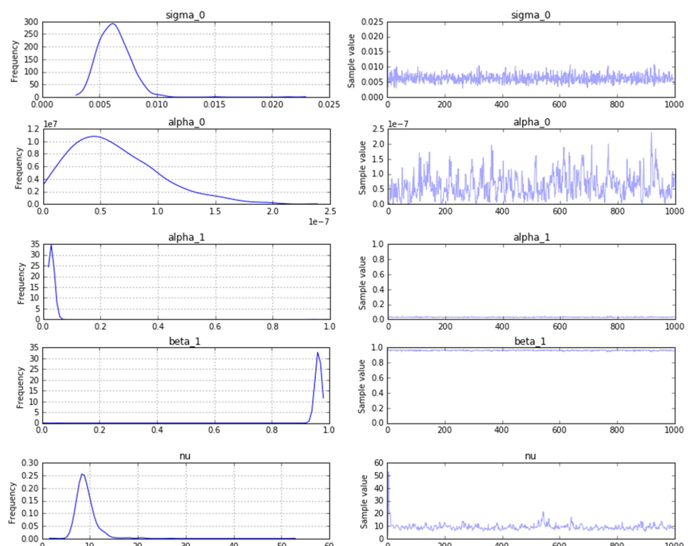
4 GARCH(1,1)

4.1 Model Setting and Estimation

$$
\begin{aligned}
r_t &= \sigma_t \epsilon_t \\
\sigma_t^2 &= \alpha_0 + \alpha_1 r_{t-1} + \alpha_2 \sigma_{t-1}^2 \\
r_t|\sigma_t &\sim \mathcal{N}(0, \sigma_t^2 \tau^2) \\
\alpha_0 &\sim exp(30) \\
\alpha_1 &\sim Uniform(0,1) \\
\beta_2 &\sim Uniform(0, \alpha_1) \\
\sigma_0^2 &\sim IG(2,3)
\end{aligned}
$$

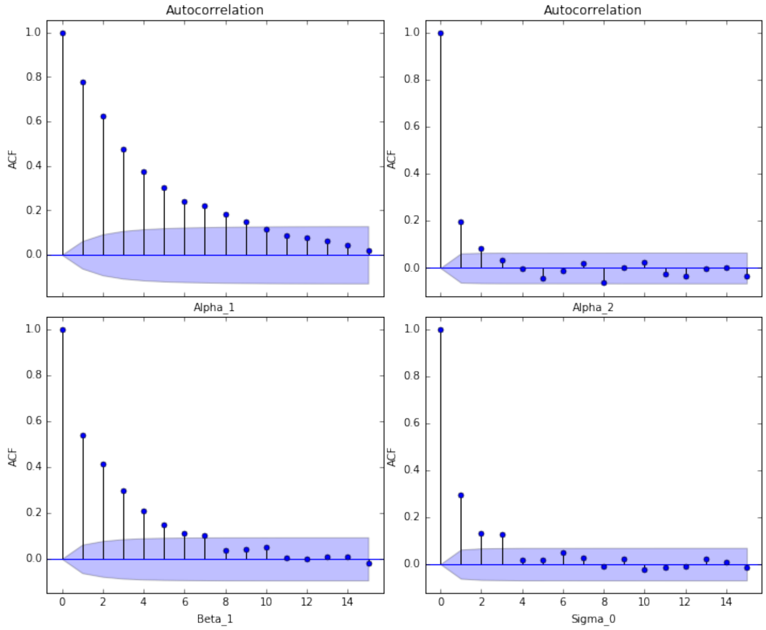
This is the Garch(1,1) with normal shock model, for Student-t shock, we only need to let degree of freedom $\nu \sim exp(0.1)$, and let $r_t|\sigma_t \sim t(\nu, 0, \sigma_t^2)$. With **PyMc 3** we install a NUTS sampler and get the posteior distribution. (Garch with Normal shock see **Appendix I**)

Figure 5: Garch-Student t Shock



Proposed parameters show some extent correlation from the ACF plot, the Metropolis sampler may stick to a local optimal, but with NUTS sampler we could avoid such dilema effectively. (See **Appen II** for very similar result for Garch-Normal Model)

Figure 6: ACF of Posterior Draw



And the model estimation and effective sample size of posterior samples is presented as below: (Result of Garch-Normal Model is in **Appen III**):

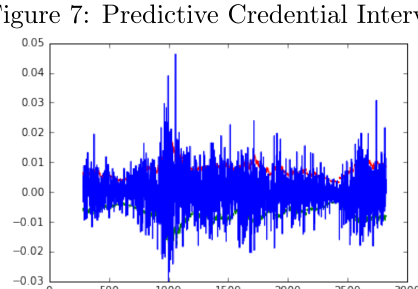
Table 3: Model Estimate and ESS

Parameter	Mean	Median	ESS(based on 1000 posterior samples)
α_0	0	0	153.66
α_1	0.029	0.031	800.00
β_1	0.96	0.96	1013.81
σ_0	0.006	0.96	2575.53

4.2 Prediction

The predictive interval and original series is presented as below, red dash and green dash indicate the 80% credencial interval :

Figure 7: Predictive Credential Interval



The 90% posterior credential interval could cover around 71.14% of original series, so we could see that the prediction quality is quite good .

4.3 Computing Time

We compare the computing time of **PyMc 3** with **Rstan** and the result is presented as below (computation is finished with the same computer):

Table 4: Computing Time over Packages

Model	Garch(1,1)-Normal	Garch(1,1)-Student_t
PyMc 3	6996.9 seconds	177.7 seconds
Rstan	251 seconds	455 seconds

For Garch-t model, since we rewrite its build-in function (actually we just rewrite several parameters and call the function manually), it performs quite well. But for Garch-normal, it takes a lot of time to warm-up.

5 STOCASTIC VOLATILITY

5.1 Student-t Distributed Shock

5.1.1 Prior Distribution

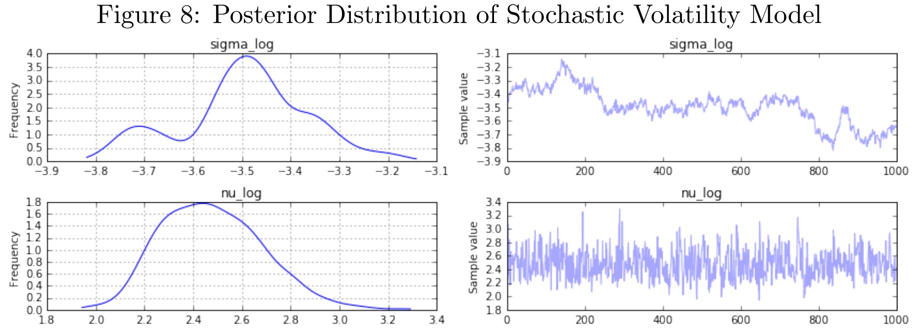
Since we have large dataset, the choice of prior distribution won't have too much effect on our inference. For this reason we choose non-informative prior distribution:

$$\begin{aligned} \sigma &\sim exp(50) \\ s_i &\sim N(s_{i-1}, \sigma^{-2}) \\ \nu &\sim exp(0.1) \\ log(\frac{y_i}{y_{i-1}}) &\sim t(\nu, 0, exp(-2s_i)) \end{aligned}$$

5.1.2 Posterio Distribution

The sampler we use is NUTS. The posterior distribution are shown as follows:

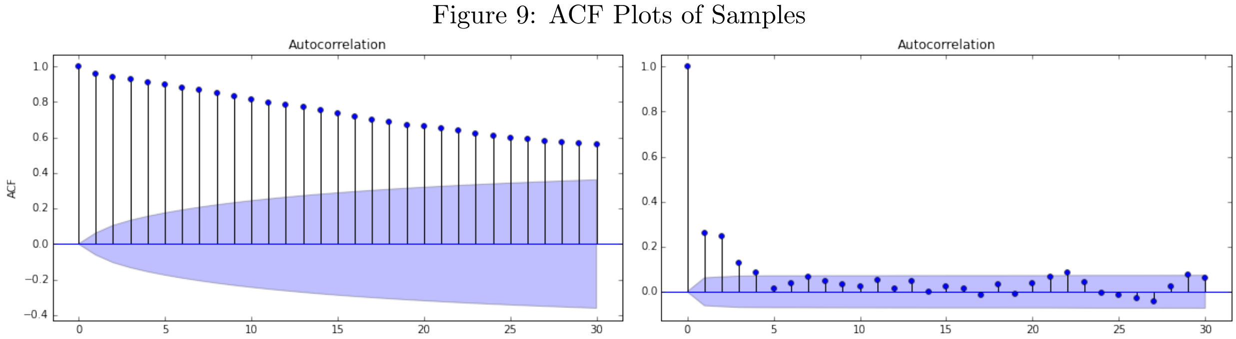
Figure 8: Posterior Distribution of Stochastic Volatility Model



The trace plot of the process shows an highly correlated chain for ν, but mixs for σ is not quite well.

The ACF of plots of of samples confirmed the problem:

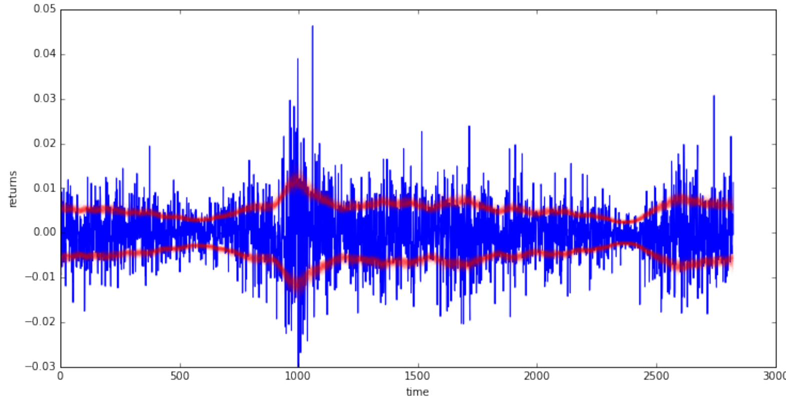
Figure 9: ACF Plots of Samples



5.1.3 Goodness of fit

In order to visually inspect how well our model is, we overlay the estimated standard deviation as follows:

Figure 10: Estimated Standard Deviation on Data



From the figure above, we can see that our model capture most of the volatilities.

5.1.4 Model Estimation

Sampling size is 1000

Table 5: Model Estimation

Parameters	Mean	Median	ESS (sec)
$\log(\sigma_0)$	-3.553	-3.597	7.8
$\log(\nu)$	2.469	2.454	3461.1

5.1.5 Computing Time

Table 6: Computing Time

Model	PyMC3	RStan
SV-Normal	42.3 second	3.8 second
SV-t	56.8 second	-

References

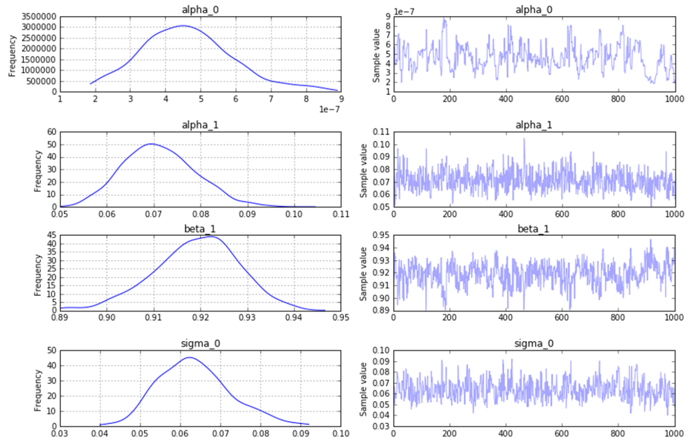
[1] Board of Governors of the Federal Reserve System (US), U.S. / Euro Foreign Exchange Rate [DEXUSEU], retrieved from FRED, Federal Reserve Bank of St. Louis https://research.stlouisfed.org/fred2/series/DEXUSEU, April 11, 2016

[2] Patil, A., D. Huard and C.J. Fonnesbeck. (2010) PyMC: Bayesian Stochastic Modelling in Python. Journal of Statistical Software, 35(4), pp. 1-81

[3] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2016. Stan: A probabilistic programming language. Journal of Statistical Software (in press)

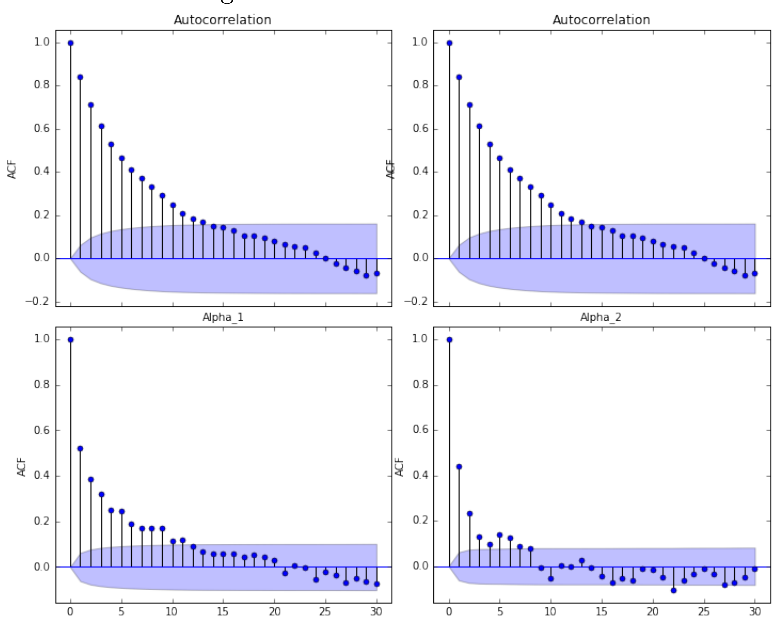
Appendix I Garch-Normal Trace Plot

Figure 1: Trace Plot of Garch-Normal Model



Appendix II Garch-Normal Posterior Draw ACF

Figure 2: ACF of Posterior Draw



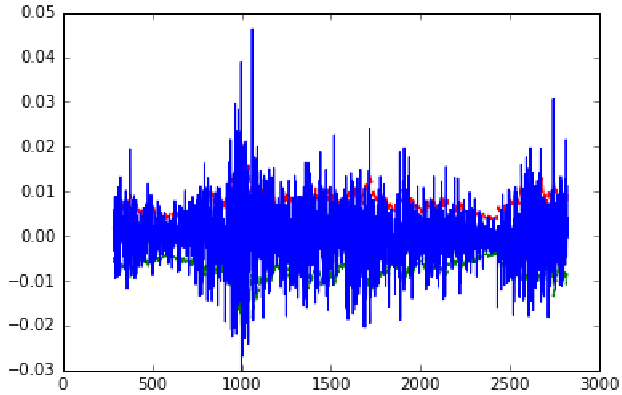
Appendix III Garch-Normal Parameter Estimation

Table 1: Garch-Normal Parameter Estimation

Parameter	Mean	Mode	ESS(based on 1000 Posterior Draw)
α_0	0	0	193.58
α_1	0.071	0.071	582.24
β_1	0.919	0.920	275.84
σ_0	0.064	0.063	2500.00

Appendix IV Garch-Normal Prediction Credential Interval (Cover Rate: 72.56%)

Figure 3: Prediction Credential Interval and Original Series



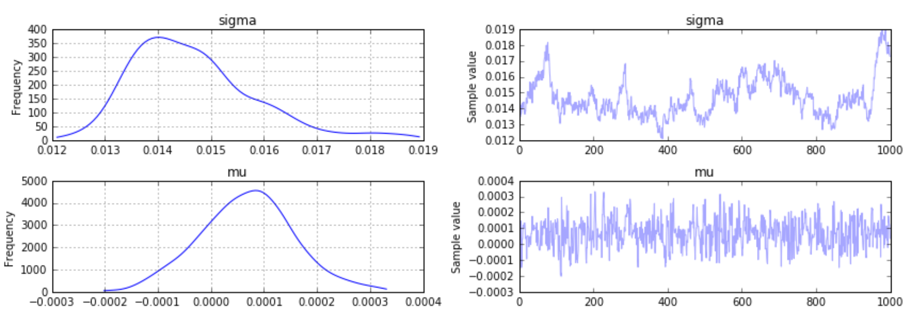
Appendix V Stochastic Volatility Model with T-Distributed Shock

The method is similiar to normal distributed, for the volatility part we only use a random walk framework. Prior and model settings are as follows

$$
\begin{aligned}
\sigma &\sim exp(50) \\
s_i &\sim N(s_{i-1}, \sigma^{-2}) \\
log(\frac{y_i}{y_{i-1}}) &\sim N(0, exp(-2s_i))
\end{aligned}
$$

Posterior Distribution

Figure 4: Stochastic Volatility with Normal Shock



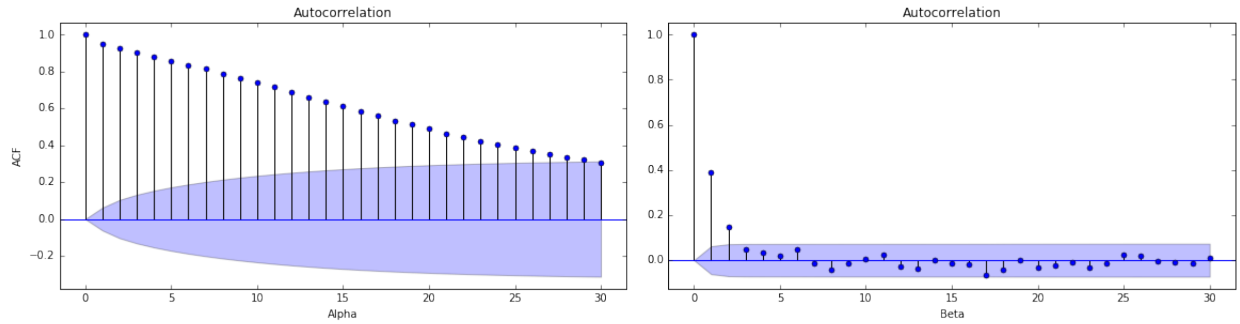
Appendix VI ACF of SV-Normal Model

Appendix VII Estimation

Table 2: Estimation of Parameters and ESS

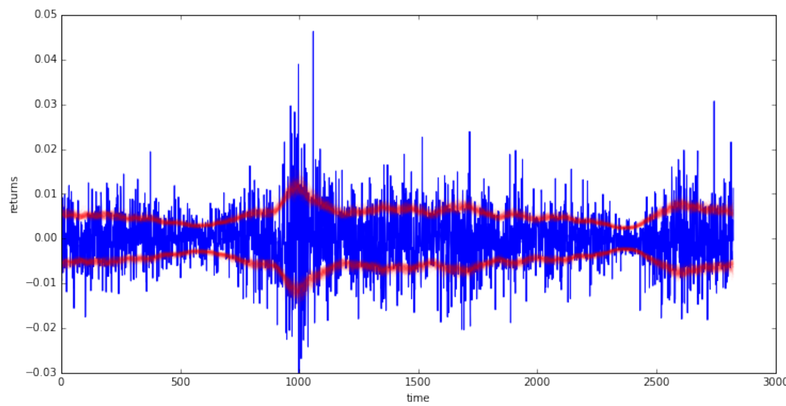
Parameters	Mean	Median	ESS(based on 1000 posterior draw)
ν	10.165	6.060	9713
σ_0	0.028	0.027	14.9

Figure 5: ACF of SV-Normal Model



Appendix VIII Posterior Prediction

Figure 6: Posterior Credential Interval 2 Times Standard Deviation Interval



Code

Listing 1: Python Code

```
#-----------------------------------AR-----------------------------------------
#---------------------------------- Install MCMC ------------------------------
basic_model = Model()
with basic_model:
# Prior Setting
alpha = Uniform('alpha', -1, 1)
beta  = Normal('beta', mu=0, sd=2)
sig   = InverseGamma('sig', alpha=2, beta=3)
# Expectation of Return
mu  = beta + alpha*r_1
# Likelihood
Y_t = Normal('Y_t', mu=mu, sd=np.sqrt(sig), observed=r[1:len(r)])
# Initial Value
start = find_MAP(model=basic_model)
# Slice Sampler
step = Slice(vars=[alpha,beta,sig])
# Slice could be replaced with NUTS/Metropolis
# Sample Process
samp = sample(1000, step=step, start=start)
#---------------------------------------- Estimation ---------------------------------------
estimate_MAP = find_MAP(model=basic_model)
print(estimate_MAP)
traceplot(samp)
summary(trace)
#---------------------------------- Posterior Prediction -----------------------------------
sig = 0.996 y_pre = np.empty([len(samp.alpha),len(Y)])
T = np.linspace(start=0, stop=len(Y), num=len(Y)) y_pre[:,0] = Y_1[0]
for i in xrange(0,len(samp.alpha)-2):
         for j in xrange(1,len(r)-2):
                  y_pre[i,j] = samp.beta[i] - samp.alpha[i]*Y_1[j-1] + np.random.normal(0, sig, 1)
pre_lo = np.percentile(y_pre, 2.5, 0)
pre_up = np.percentile(y_pre, 97.5, 0)
# Cover Rate count=0 for i in xrange(0,len(Y_1)-1):
if Y_1[i] > pre_lo[i] and Y_1[i] < pre_up[i]:
         count = count + 1 cover_rate = float(count)/len(Y_1)
print(cover_rate)

#-----------------------------------Garch-t--------------------------------------------
# ---------------------------------- MCMC Sample ---------------------------------------
garchmodel = pymc3.Model() with garchmodel:
# Model setting
alpha_0 = pymc3.Exponential('alpha_0', 30., testval=.02)
alpha_1 = pymc3.Uniform('alpha_1', lower=0, upper=1, testval=.9)
upper   = pymc3.Deterministic('upper', 1-alpha_1)
beta_1  = pymc3.Uniform('beta_1', lower=0, upper=upper, testval=.05)
sigma_0 = pymc3.Normal('sigma_0', sd=30., testval=.02)
nu      = pymc3.Exponential('nu', 1./10)
# Model with data involved
garch  =  GARCH11('garchmodel', omega=alpha_0, alpha_1=alpha_1,
beta_1=beta_1, initial_vol=sigma_0,
nu=nu, observed=returns)
# Sample Process
start  = find_MAP(model = garchmodel)
step   = Slice(vars=[alpha_0,alpha_1,beta_1,sigma_0])
trace  = sample(50, start=start, step=step)
#---------------------------------------- Estimation ---------------------------------------
%matplotlib inline traceplot(trace) summary(trace)
```

Continued Code

```
#------------------------------PosteriorPrediction------------------------------------
return_pre = np.empty([len(trace.alpha_0),len(returns)])
sigma_pre  = np.empty(len(returns))
sigma_pre[0]   = np.mean(trace.sigma_0)
for i in xrange(0,len(trace.alpha_0)):
        for j in xrange(1,len(returns)):
                sigma_pre[j]  = np.sqrt(trace.alpha_0[i-1]+trace.alpha_1[i-1]*returns[j-1]**2+
                                                trace.beta_1[i-1]*sigma_pre[j-1]**2)
return_pre[i,j]=np.random.standard_t(df=trace.nu[i-1],size=1)[0]*
                                                np.sqrt(trace.alpha_0[i-1]+trace.alpha_1[i-1]*
                                                returns[j-1]**2+trace.beta_1[i-1]*sigma_pre[j-1]**
pre_up = np.percentile(return_pre,90, 0)
pre_lo = np.percentile(return_pre,10, 0)
T = np.linspace(start=0, stop=len(returns), num=len(returns))
burnin = int(0.1*len(T))
# Cover Rate count=0 for i in xrange(burnin,len(returns)-1):
if returns[i] > pre_lo[i] and returns[i] < pre_up[i]:
count = count + 1 cover_rate = float(count)/len(returns)
print(cover_rate)

#-----------------------------------GarchNormal-----------------------------------
Garch - Normal Model is almost the same, we just leave out the prior for the degree
#  freedem and give a normal distribution to the return

#-----------------------------------Stoval-Student T-----------------------------
model = pm.Model() with model:
        sigma = pm.Exponential('sigma', 1./.02, testval=.1)
    nu = pm.Exponential('nu', 1./10)
        s = GaussianRandomWalk('s', sigma**-2, shape=n)
    r = pm.StudentT('r', nu, lam=pm.exp(-2*s), observed=mdata)
with model:
        start = pm.find_MAP(vars=[s], fmin=optimize.fmin_l_bfgs_b)
with model:
        step = pm.NUTS(vars=[s, nu,sigma],scaling=start, gamma=.25)
        start2 = pm.sample(1000, step, start=start)[-1]
# Start next run at the last sampled position.
        step = pm.NUTS(vars=[s, nu,sigma],scaling=start2, gamma=.55)
trace = pm.sample(1000, step, start=start2)
summary(trace)
#------------------------------ Posterior Predictive------------------------------
get_ipython().magic(u'matplotlib inline')
fig, axes = plt.subplots(1, 2, sharex=True, figsize=(16,4))
fig.tight_layout()
stm.graphics.tsaplots.plot_acf(trace.sigma, lags=30, ax=axes[0])
stm.graphics.tsaplots.plot_acf(trace.nu,lags=30,ax=axes[1]) axes[0].set_ylabel('ACF');
axes[0].set_xlabel('Sigma');
axes[1].set_xlabel('nu')
#------------------------------Stoval-Normal---------------------------------------
# Almost similar method, but leave out the prior for degree of freedom and give a normal
# distribution for the return
```