FE542 Midterm

Yuxuan Xia, Email:yxia16@stevens.edu

1.(50 points) Solve problems 2.1 and 2.2 on page 104 of the textbook (Tsay's Financial Time Series 3rd edition).

2.1 Suppose that the simple return of a monthly bond index follow the MA(1) model

$$R_t = a_t + 0.2a_{t-1},\ \sigma_a = 0.025$$

Assume that $a_{100} = 0.01$. Compute the 1-step- and 2-step-ahead forecasts of the return at the forecast origin t=100. What are the standard deviations of the associated forecast errors? Also compute the lag-1 and lag-2 autocorrelations of the return series.

i) Forecasts of return

$$\begin{aligned} E[R_t|\mathcal{F}_{t-1}] &= E[a_t + 0.2a_{t-1}|\mathcal{F}_{t-1}] \\ &= E[a_t|\mathcal{F}_{t-1}] + 0.2a_{t-1} \\ &= 0.2a_{t-1} \\ E[R_t|\mathcal{F}_{t-2}] &= E[a_t + 0.2a_{t-1}|\mathcal{F}_{t-2}] \\ &= E[a_t|\mathcal{F}_{t-2}] + 0.2E[a_{t-1}|\mathcal{F}_{t-2}] \\ &= 0 + 0.2 \cdot 0 \\ &= 0 \end{aligned}$$

Then, by substitution

$$E[R_{101}|\mathcal{F}_{100}] = 0.2a_{100} = 0.002$$
$$E[R_{102}|\mathcal{F}_{100}] = 0$$

ii) Standard deviation of the associated forecast error

Definition:

$$\begin{aligned} Std(p) &:= Std(R_t - E[R_t|\mathcal{F}_{t-p}] \\ Std(1) &= Std(R_t - 0.2a_{t-1}) \\ &= Std(a_t) \\ &= \sigma_a \\ &= 0.025 \end{aligned}$$

Since a_t and a_{t-1} are independent

$$\begin{aligned} Std(2) &= Std(R_t - 0) \\ &= Std(a_t + 0.2a_{t-1}) \\ &= \sigma_a\sqrt{1 + 0.2^2} \\ &= 0.0255 \end{aligned}$$

iii) Autocorrelation of return series

Since this MA(1) process is staionary

$$\begin{aligned} ACF(p) &= \frac{E[(R_t-\mu)(R_{t-p}-\mu)]}{Var[R]} \\ &= \frac{E[(R_tR_{t-p}]}{Var[R]} \end{aligned}$$

autocovariance

$$\begin{aligned} E[R_tR_{t-1}] &= E[(a_t+0.2a_{t-1})(a_{t-1}+0.2a_{t-2})] \\ &= E[0.2a_{t-1}^2]+0 \\ &= 0.2\sigma_a^2 \\ E[R_tR_{t-2}] &= E[(a_t+0.2a_{t-1})(a_{t-2}+0.2a_{t-3})] \\ &= 0 \end{aligned}$$

variance

$$\begin{aligned} Var[R] &= E[(a_t+0.2a_{t-1})^2] \\ &= \sigma_a^2(1+0.2^2) \\ &= 1.04\sigma_a^2 \end{aligned}$$

Autocorrelations

$$ACF(1) = \frac{0.2\sigma_a^2}{1.04\sigma_a^2} = 0.192$$

$$ACF(2) = \frac{0}{1.04\sigma_a^2} = 0$$

2.2 $r_t = 0.01 + 0.2r_{t-2} + a_t, a_t \sim \mathcal{N}(0, 0.02)$

i) What are the mean and variance of the return series r_t?

$$(1-0.2B^2)r_t = 0.01 + a_t$$

Solve the characteristic polinomial $1-0.2B^2=0$, all of the roots are greater than 1, so the process is stationary Then, use the property of stationary process. Add expectation and variance operator to both sides, we get

$$E[r] = 0.01 + 0.2E[r] + 0$$

$$E[r] = 0.01/0.8 = 0.0125 = \mu$$

Taking the variance of the prior equation, we have

$$\begin{aligned} Var[r_t] &= E[(0.01+0.2r_{t-2}+a_t-\mu)^2] \\ &= E[((0.01+a_t-0.8\mu)+0.2(r_{t-2}-\mu)))^2] \\ &= E[(0.01+a_t-0.8\mu)^2] + 0.2^2Var[r_{t-2}] + 0.4E[(r_{t-2}-\mu)(0.01+a_t-0.8\mu)] \\ &= (0.01-0.8\mu)^2 + \sigma_a^2 + 0.2^2Var[r] + 0 \end{aligned}$$

$$Var[r] = \frac{(0.01-0.8\mu)^2+\sigma_a^2}{1-0.2^2} = 0.0208$$

ii)Comput the lag-1 and lag-2 autocorrelations of r_t

lag-1 autocorrelation

$$\begin{aligned} ACF(1) &= \frac{E[(r_t - \mu)(r_{t-1} - \mu)]}{Var[r]} \\ &= 0 \end{aligned}$$

lag-2 autocorrelation

$$\begin{aligned} ACF(2) &= \frac{E[(r_t - \mu)(r_{t-2} - \mu)]}{Var[r]} \\ &= \frac{E[(0.01 + 0.2r_{t-2} + a_t - \mu)(r_{t-2} - \mu)]}{Var[r]} \\ &= \frac{E[(0.01 + a_t - 0.8\mu + 0.2(r_{t-2} - \mu))(r_{t-2} - \mu)]}{Var[r]} \\ &= \frac{0 + 0.2Var[r]}{Var[r]} \\ &= 0.2 \end{aligned}$$

iii) Compute the 1- and 2-step-ahead forecasts of the return series at the forecast origin t=100. Assume $r_{100} = -0.01$, and $r_{99} = 0.02$

$$\begin{aligned} E[r_{t+1}|\mathcal{F}_t] &= 0.01 + 0.2r_{t-1} + E[a_{t+1}|\mathcal{F}_t] \\ &= 0.01 + 0.2r_{t-1} \end{aligned}$$

$$\begin{aligned} E[r_{t+2}|\mathcal{F}_t] &= 0.01 + 0.2r_t + E[a_{t+2}|\mathcal{F}_t] \\ &= 0.01 + 0.2r_t \end{aligned}$$

By substitution,

$$\begin{aligned} E[r_{101}] &= 0.01 + 0.2r_{99} = 0.014 \\ E[r_{102}] &= 0.01 + 0.2r_{100} = 0.008 \end{aligned}$$

iv) What are the associated standard deviations of the forecast errors?

1-step-ahead

$$\begin{aligned} Std[(r_{t+1} - E[r_{t+1}|\mathcal{F}_t])|\mathcal{F}_t] &= Std[a_{t+1}|\mathcal{F}_t] \\ &= \sigma_a \\ &= 0.141 \end{aligned}$$

2-step-ahead

$$\begin{aligned} Std[(r_{t+2} - E[r_{t+2}|\mathcal{F}_t])|\mathcal{F}_t] &= Std[a_{t+2}|\mathcal{F}_t] \\ &= \sigma_a \\ &= 0.141 \end{aligned}$$

2.(100 points) For this problem download data for any equity you like. Please download data a.s.a.p. after you see this. Choose an equity that has a history of at least 20 years. Download daily data for the equity. A great concern for time series estimation is that one needs a large number of observations from the time series for a reliable estimation and forecast. On the other hand the data has to be stationary for a reliable estimation and in general it is believed that data stays stationary for only short periods of time. Here we will study the performance of various models and the effect of the data length on the estimation. To do this create new data vectors. Data 1 contains the last 6 months of the equity return. Data 2 contains the last 12 months. Data 3 the last 24 months. Data 4 the last 30 months. Data 5 the last 60 months. Data 6 the last 120 months and finally Data 7 the last 240 months.

(a) Work with continuously compounded returns. Pay attention tothe proper date order in the data you downloaded.

Preprocessing

In [1]:
```
import numpy as np
import pandas as pd
import scipy as sp
import matplotlib.pyplot as plt
import statsmodels.api as sm
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/compat/pandas.py:56: FutureWarnin
g: The pandas.core.datetools module is deprecated and will be removed in a future version. Please use the pan
das.tseries module instead.
  from pandas.core import datetools
```

In [2]:
```
import datetime
def preprocess_wrds_data(df):
    df['date'] = df['caldt'].apply(lambda x: datetime.datetime.strptime(str(x),"%Y%m%d"))
    df.index = df.date
    df.drop(['date'],axis=1,inplace=True)
    df['simple_return'] = (df.totval-df.totval.shift(1))/df.totval.shift(1)
    df['log_return'] = np.log(df.totval/df.totval.shift(1))
    df.dropna(axis=0,how='any',inplace=True)
```

In [3]:
```
df = pd.read_csv('CRSP_on_SP500_20y.csv')
print(df.head())
preprocess_wrds_data(df)
print(df.tail())
```

```
      caldt    vwretd        totval  spindx
0  19971103  0.025821  7.170800e+09  938.99
1  19971104  0.002083  7.185659e+09  940.76
2  19971105  0.002565  7.202972e+09  942.76
3  19971106 -0.005009  7.166324e+09  938.03
4  19971107 -0.011187  7.084584e+09  927.51
               caldt    vwretd        totval   spindx  simple_return  \
date
2017-11-24  20171124  0.002211  2.310906e+10  2602.42       0.002211
2017-11-27  20171127 -0.000270  2.310033e+10  2601.42      -0.000378
2017-11-28  20171128  0.009686  2.332317e+10  2627.04       0.009647
2017-11-29  20171129 -0.000280  2.331360e+10  2626.07      -0.000410
2017-11-30  20171130  0.008424  2.349011e+10  2647.58       0.007571

            log_return
date
2017-11-24    0.002208
2017-11-27   -0.000378
2017-11-28    0.009600
2017-11-29   -0.000411
2017-11-30    0.007543
```

In [4]:
```
mask_6m = (df.index>=datetime.datetime(2017,5,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [5]:
```
mask_12m = (df.index>=datetime.datetime(2016,12,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [6]:
```
mask_24m = (df.index>=datetime.datetime(2015,12,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [7]:
```
mask_30m = (df.index>=datetime.datetime(2015,5,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [8]:
```
mask_60m = (df.index>=datetime.datetime(2013,12,1)) & (df.index<=datetime.datetime(2017,11,30))
```

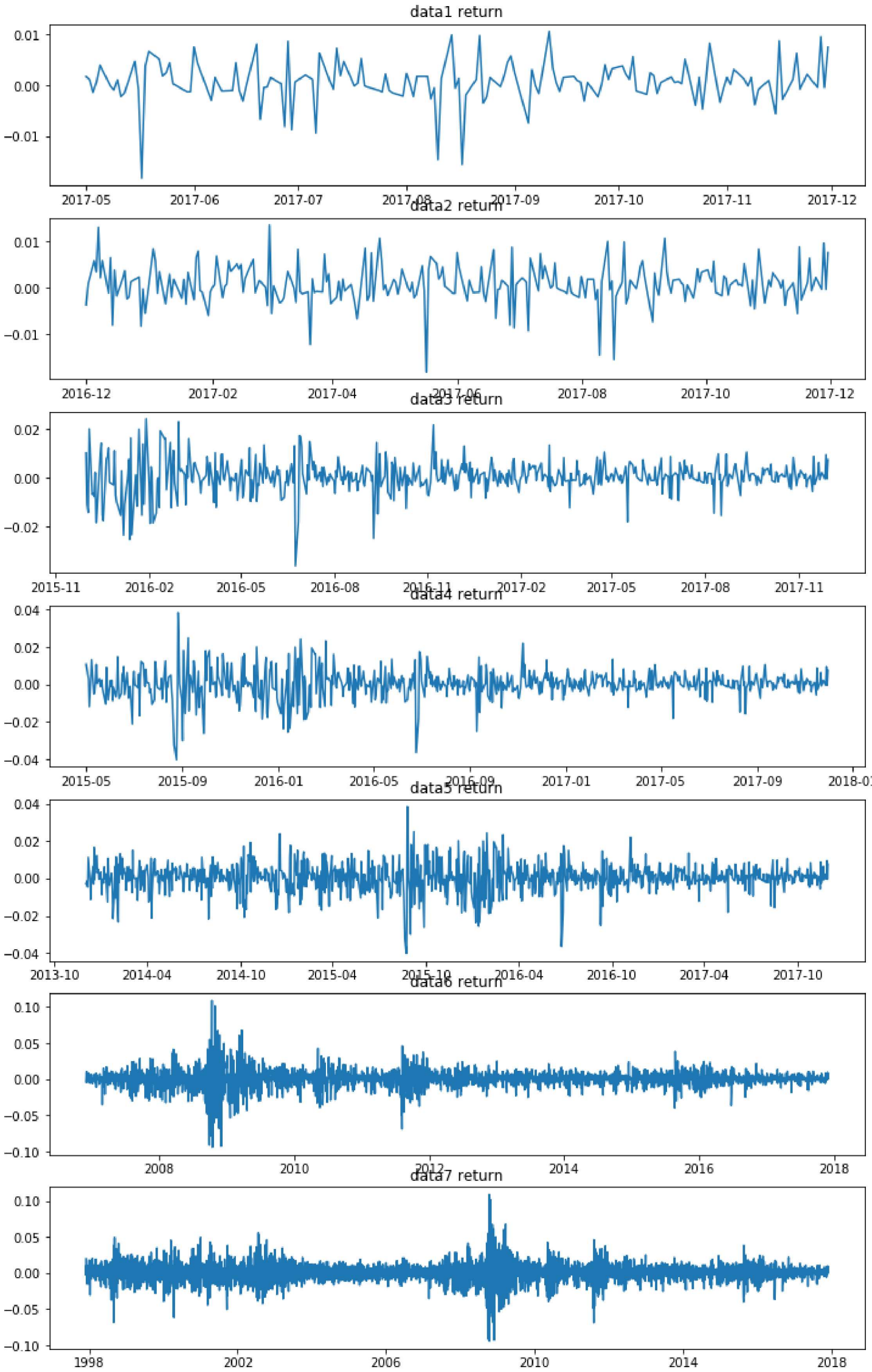
In [9]:
```
mask_120m = (df.index>=datetime.datetime(2006,12,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [10]:
```
mask_240m = (df.index>=datetime.datetime(1997,12,1)) & (df.index<=datetime.datetime(2017,11,30))
```

In [11]:
```
data = []
data.append(df.loc[mask_6m]['log_return'])
data.append(df.loc[mask_12m]['log_return'])
data.append(df.loc[mask_24m]['log_return'])
data.append(df.loc[mask_30m]['log_return'])
data.append(df.loc[mask_60m]['log_return'])
data.append(df.loc[mask_120m]['log_return'])
data.append(df.loc[mask_240m]['log_return'])
```

Ploting Data

In [12]:
```
fig = plt.figure(figsize=(12,20))

plt.subplot(7,1,1)
plt.plot(data[0])
plt.title("data1 return")

plt.subplot(7,1,2)
plt.plot(data[1])
plt.title("data2 return")

plt.subplot(7,1,3)
plt.plot(data[2])
plt.title("data3 return")

plt.subplot(7,1,4)
plt.plot(data[3])
plt.title("data4 return")

plt.subplot(7,1,5)
plt.plot(data[4])
plt.title("data5 return")

plt.subplot(7,1,6)
plt.plot(data[5])
plt.title("data6 return")

plt.subplot(7,1,7)
plt.plot(data[6])
plt.title("data7 return")

plt.show()
```



(b) Study unit root non-stationarity. Compare for all data periods.

Unit-root checking

In [13]:
```
from statsmodels.tsa.stattools import adfuller

def unit_root_cheking(time_series,regression):
    # perform Dickey-Fuller test
    dftest = adfuller(time_series,regression='c',autolag="BIC")
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value',
                                             '#Lags Used', 'Number of Observations Used'])
    for key, value in dftest[4].items():
        dfoutput['Critical Value (%s)' % key] = value
    print(dfoutput)
```

In [14]:
```
for i in range(7):
    print("data",i+1)
    unit_root_cheking(data[i],'c')
```

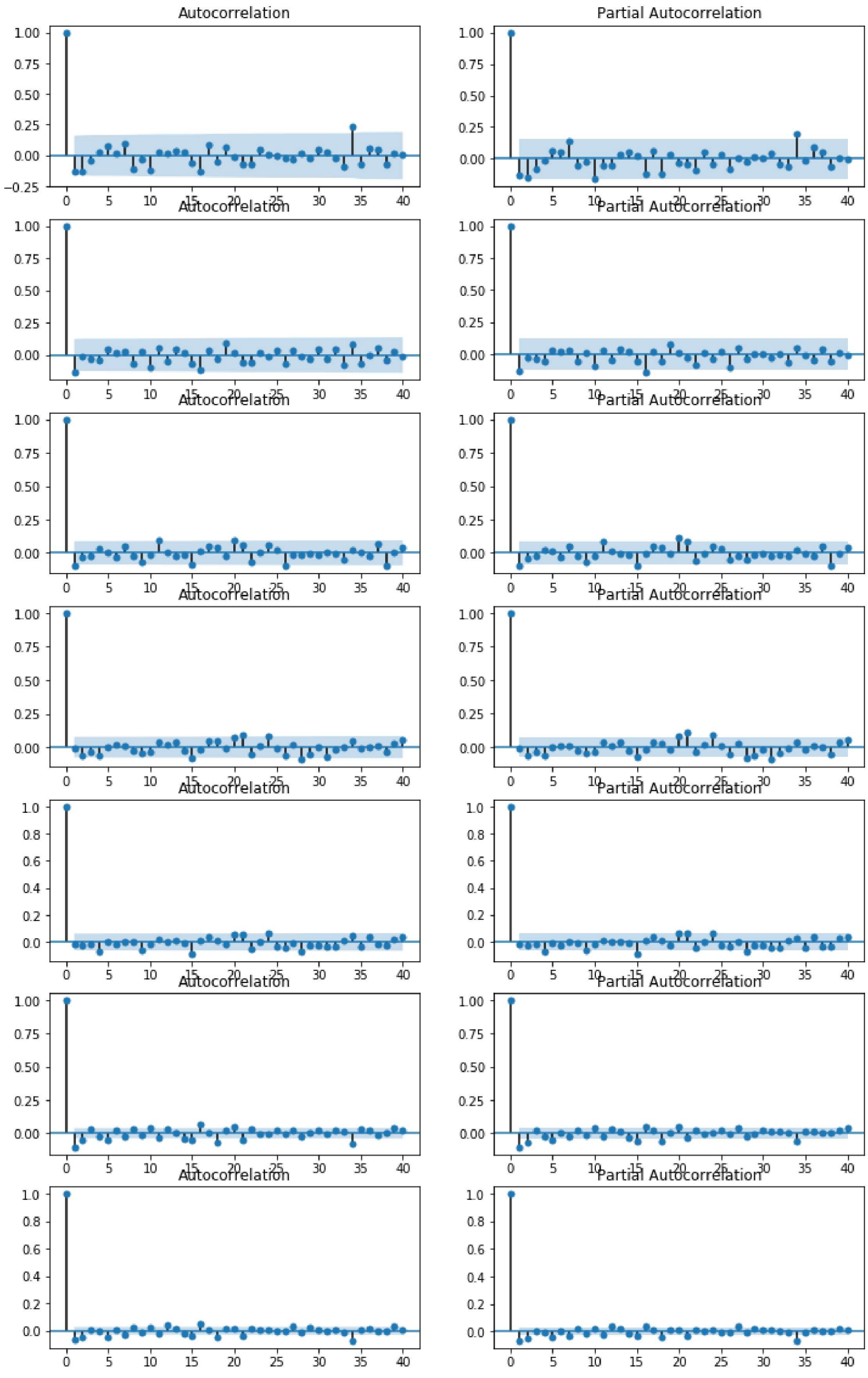
```
data 1
Test Statistic                -1.378442e+01
p-value                        9.168658e-26
#Lags Used                     0.000000e+00
Number of Observations Used    1.490000e+02
Critical Value (1%)           -3.475018e+00
Critical Value (5%)           -2.881141e+00
Critical Value (10%)          -2.577221e+00
dtype: float64
data 2
Test Statistic                -1.799568e+01
p-value                        2.744925e-30
#Lags Used                     0.000000e+00
Number of Observations Used    2.510000e+02
Critical Value (1%)           -3.456674e+00
Critical Value (5%)           -2.873125e+00
Critical Value (10%)          -2.572944e+00
dtype: float64
data 3
Test Statistic                 -24.807529
p-value                          0.000000
#Lags Used                       0.000000
Number of Observations Used    504.000000
Critical Value (1%)             -3.443392
Critical Value (5%)             -2.867292
Critical Value (10%)            -2.569833
dtype: float64
data 4
Test Statistic                -25.673568
p-value                         0.000000
#Lags Used                      0.000000
Number of Observations Used    652.000000
Critical Value (1%)             -3.440419
Critical Value (5%)             -2.865983
Critical Value (10%)            -2.569136
dtype: float64
data 5
Test Statistic                 -32.296463
p-value                          0.000000
#Lags Used                       0.000000
Number of Observations Used   1007.000000
Critical Value (1%)              -3.436860
Critical Value (5%)              -2.864414
Critical Value (10%)             -2.568300
dtype: float64
data 6
Test Statistic                 -41.697627
p-value                          0.000000
#Lags Used                       1.000000
Number of Observations Used   2767.000000
Critical Value (1%)              -3.432716
Critical Value (5%)              -2.862585
Critical Value (10%)             -2.567326
dtype: float64
data 7
Test Statistic                 -54.436227
p-value                          0.000000
#Lags Used                       1.000000
Number of Observations Used   5032.000000
Critical Value (1%)              -3.431650
Critical Value (5%)              -2.862115
Critical Value (10%)             -2.567076
dtype: float64
```

Conclusion: All results reject the unit-roots. Because we are considering the log returns of the price itself it is much more stationary.

(c) Build the best AR, the best MA and the best ARMA model afteryou remove non-stationarity (if needed) for each time period.

ACF and PACF

In [15]:
```
plt.figure(figsize=(12,20))
for i in range(7):
    ax1 = plt.subplot(7,2,2*i+1)
    sm.graphics.tsa.plot_acf(data[i], alpha=0.05, lags=40, ax=ax1)
    ax2 = plt.subplot(7,2,2*i+2)
    sm.graphics.tsa.plot_pacf(data[i], alpha=0.05, lags=40, ax=ax2)

plt.show()
```



Model screening. Zero indicate there is an error in training data. Unfortunately it seems that in Python's statsmodels library, we cannot use ARMA(0,0) properly. It's not a big deal though.

In [16]:
```
bic_matrix = np.zeros((7,3,3))
for i in range(7):
    for j in range(3):
        for k in range(3):
            try:
                model=sm.tsa.ARMA(data[i],(j,k)).fit(trend='nc')
                bic_matrix[i,j,k] = model.bic
            except:
                pass

print(bic_matrix)
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:646: F
utureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In
future, it will be treated as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:650: F
utureWarning: Conversion of the second argument of issubdtype from `complex` to `np.complexfloating` is depre
cated. In future, it will be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:473: HessianInversio
nWarning: Inverting hessian failed, no bse or cov_params available
  'available', HessianInversionWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:628: RuntimeWarnin
g: overflow encountered in exp
  newparams = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:628: RuntimeWarnin
g: invalid value encountered in true_divide
  newparams = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:629: RuntimeWarnin
g: overflow encountered in exp
  tmp = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:629: RuntimeWarnin
g: invalid value encountered in true_divide
  tmp = ((1-np.exp(-params))/(1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:584: RuntimeWarnin
g: overflow encountered in exp
  newparams = ((1-np.exp(-params))/
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:585: RuntimeWarnin
g: overflow encountered in exp
  (1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:585: RuntimeWarnin
g: invalid value encountered in true_divide
  (1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:586: RuntimeWarnin
g: overflow encountered in exp
  tmp = ((1-np.exp(-params))/
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:587: RuntimeWarnin
g: overflow encountered in exp
  (1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/tsatools.py:587: RuntimeWarnin
g: invalid value encountered in true_divide
  (1+np.exp(-params))).copy()
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)

[[     0.          -1199.40701422  -1196.08335807]
 [ -1198.80198193      0.              0.        ]
 [ -1196.14440903  -1191.34922797  -1196.2713733 ]]

[[     0.          -2012.96380426  -2007.48823006]
 [ -2013.01985841  -2007.49998833  -2001.97109296]
 [ -2007.49904683  -2001.80949831      0.        ]]

[[     0.          -3577.57072525  -3571.76547942]
 [ -3577.22822962  -3571.75254392      0.        ]
 [ -3571.69064291  -3565.56424628  -3565.72429433]]

[[     0.          -4461.84486814  -4458.04535762]
 [ -4461.84286862      0.              0.        ]
 [ -4457.66108355  -4453.7947891       0.        ]]

[[     0.          -6941.02813226  -6934.77362705]
 [ -6941.0139287       0.              0.        ]
 [ -6934.65426186  -6931.65852094  -6925.23643312]]

[[     0.         -16385.91090869 -16385.11231454]
 [-16382.14972211 -16383.54498994 -16378.61470967]
 [-16386.10165537 -16379.35236333      0.        ]]

[[     0.         -30135.72403489 -30137.04851641]
 [-30133.40442262 -30137.87065096 -30128.57408617]
 [-30136.94429464 -30128.44378273 -30120.95159489]]]
```

In [17]:
```
def find_min_idx(a):
    p = np.nanargmin(a)
    return (p//a.shape[1],p%a.shape[1])

def model_screening(bic_matrix):
    ar_model = []
    ma_model = []
    arma_model = []
    ndata = bic_matrix.shape[0]
    for i in range(ndata):
        best_ar_order = np.nanargmin(bic_matrix[i][:][0])
        ar_model.append(sm.tsa.ARMA(data[i],(best_ar_order,0)).fit(trend='nc'))
        best_ma_order = np.nanargmin(bic_matrix[i][0][:])
        ma_model.append(sm.tsa.ARMA(data[i],(0,best_ma_order)).fit(trend='nc'))
        best_arma_order = find_min_idx(bic_matrix[i][1:,1:])
        best_arma_order = (best_arma_order[0]+1,best_arma_order[1]+1)
        arma_model.append(sm.tsa.ARMA(data[i],best_arma_order).fit(trend='nc'))
    return (ar_model,ma_model,arma_model)
```

In [18]:
```
(ar_model,ma_model,arma_model) = model_screening(bic_matrix)
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:646: F
utureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In
future, it will be treated as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:650: F
utureWarning: Conversion of the second argument of issubdtype from `complex` to `np.complexfloating` is depre
cated. In future, it will be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
```

Best Models Output

In [19]:
```
for i in range(7):
    print("*****************************************************")
    print("*****************************************************")
    print("The best ARMA model of data", i+1)
    print(arma_model[i].params)
    print("The best AR model of data", i+1)
    print(ar_model[i].params)
    print("The best MA model of data", i+1)
    print(ma_model[i].params)
```

```
**********************************************************
**********************************************************
The best ARMA model of data 1
ar.L1.log_return    1.182371
ar.L2.log_return   -0.893597
ma.L1.log_return   -1.354516
ma.L2.log_return    0.999998
dtype: float64
The best AR model of data 1
ar.L1.log_return   -0.111767
dtype: float64
The best MA model of data 1
ma.L1.log_return   -0.148012
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 2
ar.L1.log_return   -0.164859
ma.L1.log_return    0.060827
dtype: float64
The best AR model of data 2
ar.L1.log_return   -0.104706
dtype: float64
The best MA model of data 2
ma.L1.log_return   -0.1027
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 3
ar.L1.log_return    0.235219
ma.L1.log_return   -0.337019
dtype: float64
The best AR model of data 3
ar.L1.log_return   -0.096965
dtype: float64
The best MA model of data 3
ma.L1.log_return   -0.103982
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 4
ar.L1.log_return    0.688066
ar.L2.log_return   -0.058957
ma.L1.log_return   -0.698442
dtype: float64
The best AR model of data 4
ar.L1.log_return   -0.004763
dtype: float64
The best MA model of data 4
ma.L1.log_return   -0.005409
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 5
ar.L1.log_return    0.884136
ar.L2.log_return   -0.013334
ma.L1.log_return   -0.905570
dtype: float64
The best AR model of data 5
ar.L1.log_return   -0.016853
dtype: float64
The best MA model of data 5
ma.L1.log_return   -0.017695
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 6
ar.L1.log_return    0.363451
ma.L1.log_return   -0.473663
dtype: float64
The best AR model of data 6
ar.L1.log_return   -0.102355
dtype: float64
The best MA model of data 6
ma.L1.log_return   -0.115097
dtype: float64
**********************************************************
**********************************************************
The best ARMA model of data 7
ar.L1.log_return    0.577309
ma.L1.log_return   -0.646258
dtype: float64
The best AR model of data 7
ar.L1.log_return   -0.070463
ar.L2.log_return   -0.048930
dtype: float64
The best MA model of data 7
ma.L1.log_return   -0.070436
ma.L2.log_return   -0.045045
dtype: float64
```

(d) Build a seasonal model for your data using whatever lag you feelapproximates the data best.

i) Experiment (One can skip this section, it's not relevant to the answer)

Experiment: Cosine period data This is an independent problem, one can skip this section Assume we know

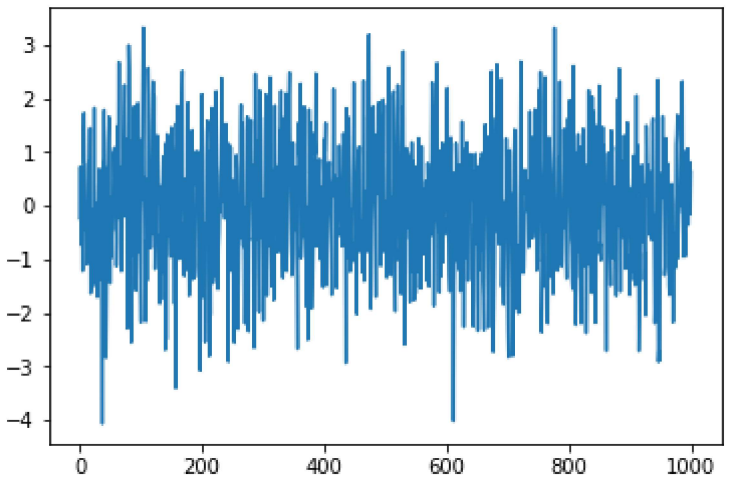
$$y_k = cos(\frac{k\pi}{4}) + e_k$$

where

$$e_k \sim N(0,1)$$

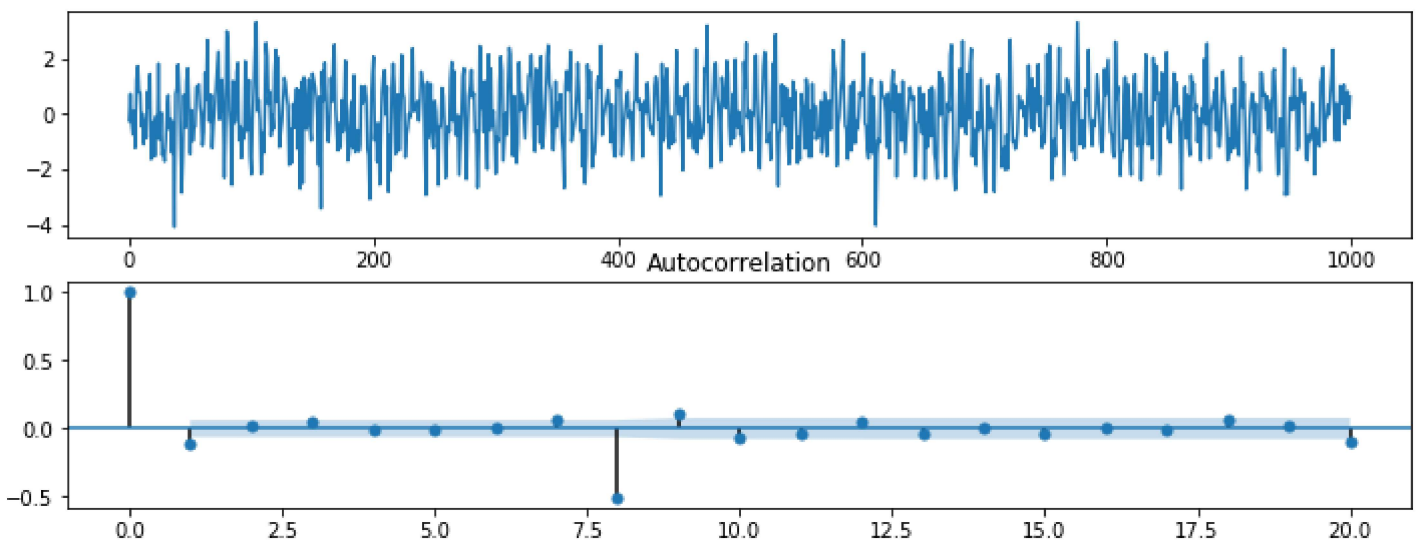
Apparently there is a period 8 under y_k, should we construct a seasonal model?

In [20]:
```
x = np.cos([i*0.25*np.pi for i in range(1000)])
e = sp.stats.norm(0,1).rvs(1000)
y = x+e
y = pd.DataFrame(y)
plt.plot(y)
plt.show()
```



Plot ACF, one can see there is an obvious bump at lag=8

In [21]:
```
plt.figure(figsize=(12,12))
plt.subplot(5,1,1)
plt.plot(y)
ax2 = plt.subplot(5,1,2)
sm.graphics.tsa.plot_acf((y-y.shift(8)).dropna(axis=0,inplace=False),alpha=0.05,lags=20,ax=ax2)
plt.show()
```



ARMA(1,8) model with the specific non-zero 8th MA lag

In [22]:
```
x = np.cos([i*0.25*np.pi for i in range(1000)])
e = sp.stats.norm(0,1).rvs(1000)
y = x+e
ma_lag = np.zeros(8)
ma_lag[7]=1
mod = sm.tsa.SARIMAX(y,order=(1,0,ma_lag))
cos_arma_model = mod.fit()
print(cos_arma_model.summary())
```

```
                           Statespace Model Results
================================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:             SARIMAX(1, 0, (8,))   Log Likelihood               -1588.993
Date:                 Sun, 01 Apr 2018   AIC                           3183.985
Time:                         19:32:24   BIC                           3198.708
Sample:                              0   HQIC                          3189.581
                                - 1000
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.1696      0.034      5.048      0.000       0.104       0.235
ma.L8          0.2055      0.031      6.631      0.000       0.145       0.266
sigma2         1.4046      0.068     20.770      0.000       1.272       1.537
===================================================================================
Ljung-Box (Q):                      980.82   Jarque-Bera (JB):                 2.71
Prob(Q):                              0.00   Prob(JB):                         0.26
Heteroskedasticity (H):               0.86   Skew:                            -0.01
Prob(H) (two-sided):                  0.17   Kurtosis:                         2.75
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

ARMAX(1,0,0) model with hidden process $x_k = cos(\frac{k\pi}{4})$

In [23]:
```
x = np.cos([i*0.25*np.pi for i in range(1000)])
e = sp.stats.norm(0,1).rvs(1000)
y = x+e
mod = sm.tsa.SARIMAX(y,x,order=(1,0,0))
cos_armax_model = mod.fit()
print(cos_armax_model.summary())
```

```
                           Statespace Model Results
================================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:                SARIMAX(1, 0, 0)   Log Likelihood               -1448.086
Date:                 Sun, 01 Apr 2018   AIC                           2902.172
Time:                         19:32:24   BIC                           2916.895
Sample:                              0   HQIC                          2907.768
                                - 1000
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
x1             1.0022      0.045     22.246      0.000       0.914       1.091
ar.L1         -0.0148      0.033     -0.456      0.649      -0.079       0.049
sigma2         1.0600      0.047     22.339      0.000       0.967       1.153
===================================================================================
Ljung-Box (Q):                       40.72   Jarque-Bera (JB):                 0.15
Prob(Q):                              0.44   Prob(JB):                         0.93
Heteroskedasticity (H):               1.02   Skew:                            -0.03
Prob(H) (two-sided):                  0.83   Kurtosis:                         3.00
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

In [24]:
```
x = np.cos([i*0.25*np.pi for i in range(1000)])
e = sp.stats.norm(0,1).rvs(1000)
y = x+e
cos_sarima_model = sm.tsa.SARIMAX(y,order=(1,0,0),seasonal_order=(1,1,1,8)).fit()
print(cos_sarima_model.summary())
```

```
                                 Statespace Model Results
==============================================================================================
Dep. Variable:                                   y   No. Observations:                 1000
Model:             SARIMAX(1, 0, 0)x(1, 1, 1, 8)   Log Likelihood               -1440.587
Date:                           Sun, 01 Apr 2018   AIC                           2889.174
Time:                                   19:32:26   BIC                           2908.805
Sample:                                        0   HQIC                          2896.636
                                            - 1000
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0164      0.033      0.502      0.615      -0.048       0.080
ar.S.L8       -0.0494      0.032     -1.562      0.118      -0.111       0.013
ma.S.L8       -0.9997      0.639     -1.565      0.117      -2.251       0.252
sigma2         1.0273      0.647      1.587      0.112      -0.241       2.296
===================================================================================
Ljung-Box (Q):                       33.00   Jarque-Bera (JB):                 4.04
Prob(Q):                              0.78   Prob(JB):                         0.13
Heteroskedasticity (H):               1.09   Skew:                             0.00
Prob(H) (two-sided):                  0.41   Kurtosis:                         3.31
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

In [25]:
```
print("arma",cos_arma_model.bic)
print("armax",cos_armax_model.bic)
print("sarima",cos_sarima_model.bic)
```

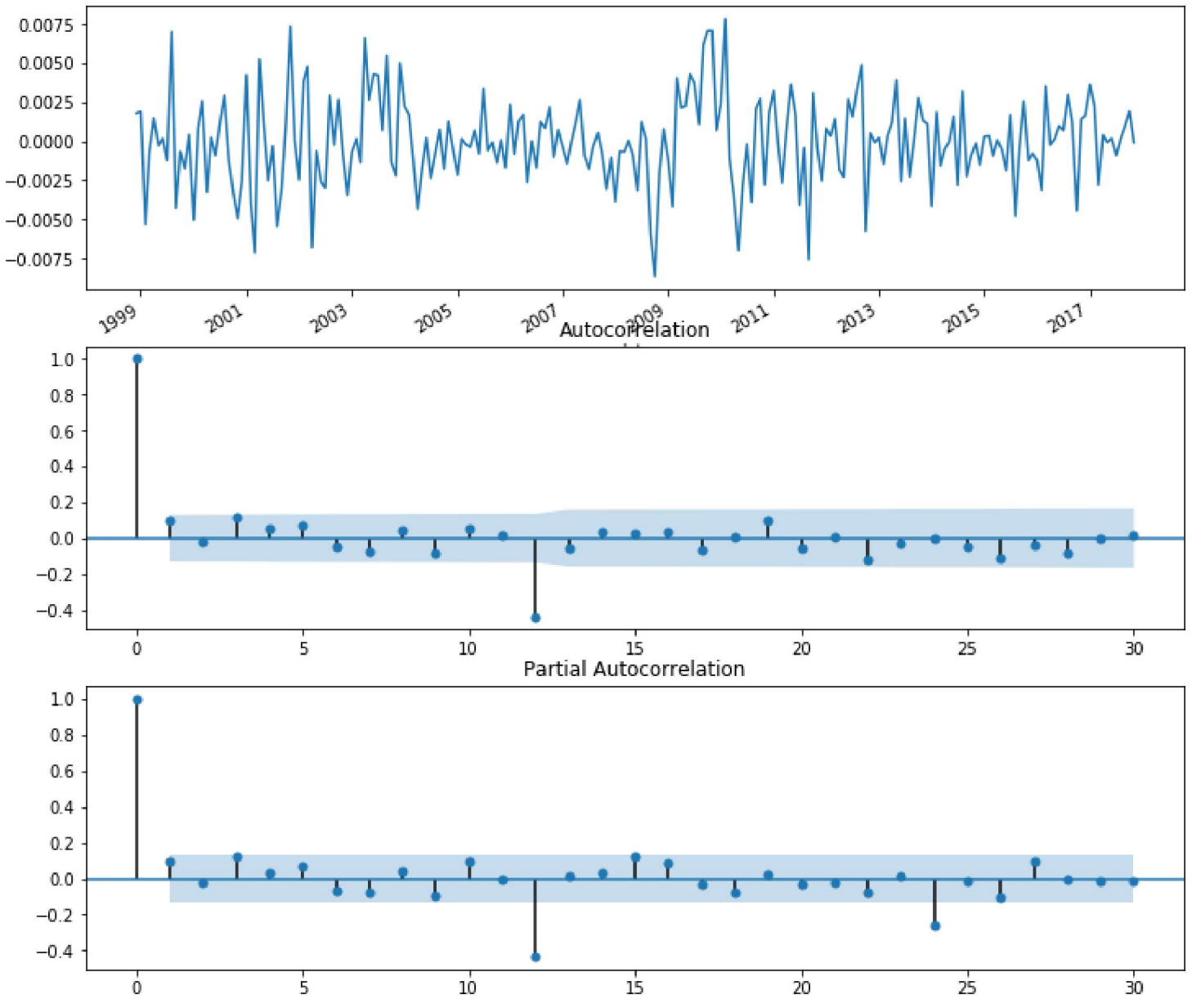
```
arma 3198.708393152093
armax 2916.8952383048204
sarima 2908.8054754311056
```

Conclusion: If you know the hidden process x, armax should be most effective. However, in practice, we are not supposed to know them, seasonal arima model should be an good alternative

ii) Monthly data testing (One can skip this section, it's not relevant to the answer)

Visualize Seasonality of Month S&P500 return Data, there is an obvious bump at lag=12, indicating the period = 12 months

In [26]:
```
plt.figure(figsize=(12,12))
plt.subplot(3,1,1)
(data[6].resample('1BMS').mean()-data[6].resample('1BMS').mean().shift(12)).dropna().plot()
ax2 = plt.subplot(3,1,2)
sm.graphics.tsa.plot_acf((data[6].resample('1BMS').mean()-data[6].resample('1BMS').mean().shift(12)).dropna
(),
                         alpha=0.05,lags=30,ax=ax2)
ax2 = plt.subplot(3,1,3)
sm.graphics.tsa.plot_pacf((data[6].resample('1BMS').mean()-data[6].resample('1BMS').mean().shift(12)).dropna
(),
                         alpha=0.05,lags=30,ax=ax2)
plt.show()
```



In [27]:
```
month_data = data[6].resample('1BMS').mean()
annual_difference_data = data[6].resample('1BMS').mean()-data[6].resample('1BMS').mean().shift(12)
annual_difference_data.dropna(axis=0,inplace=True)
print(month_data.head())
```

```
date
1997-12-01    0.000815
1998-01-01    0.000703
1998-02-02    0.003592
1998-03-02    0.002300
1998-04-01    0.000462
Freq: BMS, Name: log_return, dtype: float64
```

ARMA(1,12) Calibration Here, we only reserved the 12th lag of MA part

In [28]:
```
ar_order = 1
ma_order = np.zeros(12)
ma_order[11] = 1
mod = sm.tsa.SARIMAX(annual_difference_data, order=(ar_order, 0, ma_order))
res = mod.fit()
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [29]:
```
print(res.summary())
```

```
                           Statespace Model Results
==============================================================================
Dep. Variable:               log_return   No. Observations:                  228
Model:             SARIMAX(1, 0, (12,))   Log Likelihood                1061.851
Date:                  Sun, 01 Apr 2018   AIC                          -2117.703
Time:                          19:32:27   BIC                          -2107.415
Sample:                      12-01-1998   HQIC                         -2113.552
                           - 11-01-2017
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.1573      0.053      2.943      0.003       0.053       0.262
ma.L12        -0.6639      0.060    -11.023      0.000      -0.782      -0.546
sigma2      5.111e-06   4.28e-07     11.929      0.000    4.27e-06    5.95e-06
===================================================================================
Ljung-Box (Q):                       41.70   Jarque-Bera (JB):                 5.78
Prob(Q):                              0.40   Prob(JB):                         0.06
Heteroskedasticity (H):               0.47   Skew:                            -0.28
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.54
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Using the equivalent SARIMAX model and get the same result

In [30]:
```
ma_order = np.zeros(12)
ma_order[11] = 1
seasonal_first_difference_month_mod = sm.tsa.SARIMAX(month_data,order=(1,0,ma_order),seasonal_order=(0,1,0,12
),enforce_invertibility=False)
seasonal_first_difference_month_res = mod.fit()
print(seasonal_first_difference_month_res.summary())
```

```
                           Statespace Model Results
==============================================================================
Dep. Variable:               log_return   No. Observations:                  228
Model:             SARIMAX(1, 0, (12,))   Log Likelihood                1061.851
Date:                  Sun, 01 Apr 2018   AIC                          -2117.703
Time:                          19:32:27   BIC                          -2107.415
Sample:                      12-01-1998   HQIC                         -2113.552
                           - 11-01-2017
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.1573      0.053      2.943      0.003       0.053       0.262
ma.L12        -0.6639      0.060    -11.023      0.000      -0.782      -0.546
sigma2      5.111e-06   4.28e-07     11.929      0.000    4.27e-06    5.95e-06
===================================================================================
Ljung-Box (Q):                       41.70   Jarque-Bera (JB):                 5.78
Prob(Q):                              0.40   Prob(JB):                         0.06
Heteroskedasticity (H):               0.47   Skew:                            -0.28
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.54
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

Apparently, SARIMA model is good for monthly average data

iii) Daily data analysis (This is the answer)

However, if we use the daily data, seasonality is not always obvious. Here I use SARIMAX model with hidden process and seasonal orders. The hidden process is set to be the annual moving average process, and the seansonal order=(0,1,0,252)

```
seasonal_period=252
seasonal_mod = sm.tsa.SARIMAX(data[4][seasonal_period-1:],data[4].rolling(seasonal_period).mean()[seasonal_period-1:],
                order=(1,0,1),seasonal_order=(0,1,0,seasonal_period))
seasonal_res = seasonal_mod.fit(disp=True)
print(seasonal_res.summary())
```

However, when I exucute the above code, the PC crashed. So, I'll choose a shorter seasonal_period, say 5 (one week).

In [31]:
```
seasonal_period=5
seasonal_bic_matrix = np.zeros((7,4,4))
for i in range(7):
    for j in range(4):
        for k in range(4):
            try:
                seasonal_model = sm.tsa.SARIMAX(data[i], order=(j,0,k),seasonal_order=(0,1,0,seasonal_period
)).fit()
                seasonal_bic_matrix[i,j,k] = seasonal_model.bic
            except:
                pass
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [32]:
```
seasonal_bic_matrix
```

Out[32]:
```
array([[[     0.        ,      0.        ,  -1075.75451869,
          -1070.21829042],
         [ -1074.23853319,  -1073.06108844,  -1068.24791803,
          -1065.32385132],
         [ -1077.36579397,  -1072.30499534,  -1069.77279755,
          -1070.15956004],
         [ -1072.53122025,  -1067.58092737,      0.        ,
          -1065.027034  ]],

        [[     0.        ,  -1825.55757036,  -1820.35631795,
          -1814.91199446],
         [ -1825.13652385,  -1820.29520496,      0.        ,
          -1809.36679183],
         [ -1820.29881843,  -1814.66755972,  -1813.27974129,
          -1807.31432519],
         [ -1814.8085657 ,  -1809.26570724,      0.        ,
          -1804.01802003]],

        [[     0.        ,  -3205.44932533,  -3199.91340889,
          -3193.15453095],
         [ -3205.07064602,  -3200.21381513,      0.        ,
          -3186.97182818],
         [ -3199.87741374,  -3193.62854031,      0.        ,
          -3181.33813121],
         [ -3193.65064638,  -3187.40773022,      0.        ,
          -3176.20208861]],

        [[     0.        ,  -3980.86946551,  -3975.5586311 ,
          -3970.40005461],
         [ -3980.85043934,  -3974.4238187 ,      0.        ,
          -4003.75103516],
         [ -3975.71376182,  -3969.22506306,      0.        ,
          -3956.9585924 ],
         [ -3969.26114473,  -3962.78231257,      0.        ,
          -4041.49108481]],

        [[     0.        ,  -6209.54523968,  -6202.79880387,
          -6243.88125604],
         [ -6209.51827095,  -6201.75649424,      0.        ,
          -6299.15933972],
         [ -6202.99358878,  -6196.07902654,      0.        ,
          -6182.93886364],
         [ -6196.07919566,  -6189.16330905,      0.        ,
          -6313.02701279]],

        [[     0.        , -14300.07968432, -14301.32111391,
         -14414.96296233],
         [-14296.80296886, -14297.88591103,      0.        ,
          -14725.34513205],
         [-14301.14513095, -14293.92750108,      0.        ,
          -14698.41710261],
         [-14296.00565737, -14288.17209889,      0.        ,
          -14699.32598803]],

        [[     0.        , -26394.48481796, -26391.3057639 ,
          -26608.57493301],
         [-26393.10715923, -26392.50889013,      0.        ,
          -26637.40476438],
         [-26391.25345929, -26376.05954328,      0.        ,
          -27116.36144873],
         [-26383.92847773, -26375.47897473,      0.        ,
          -27069.72846764]]])
```

In [33]:
```
best_sarma_order = find_min_idx(seasonal_bic_matrix[0])
print(best_sarma_order[0],best_sarma_order[1])
print(seasonal_period)
best_sarima_order = (int(best_sarma_order[0]),0,int(best_sarma_order[1]))
best_sarima_order2 = (2,0,0)
print(best_sarima_order==best_sarima_order2)
print(best_sarima_order,best_sarima_order2)
sm.tsa.SARIMAX(data[i],order=best_sarima_order,seasonal_order=(0,1,0,5)).fit()
print((best_sarma_order[0],0,best_sarma_order[1]))
```

```
2 0
5
True
(2, 0, 0) (2, 0, 0)
(2, 0, 0)
```

In [34]:
```
best_sarma_order = find_min_idx(seasonal_bic_matrix[1])
best_sarima_order = (int(best_sarma_order[0]),0,int(best_sarima_order[1]))
print(best_sarima_order)
```

```
(0, 0, 0)
```

In [35]:

```python
sarma_model = []
ndata = bic_matrix.shape[0]
for i in range(0,ndata):
    best_sarma_order = find_min_idx(seasonal_bic_matrix[i])
    best_sarima_order = (int(best_sarma_order[0]),0,int(best_sarma_order[1]))
#     print(best_sarima_order)
    sarma_model.append(sm.tsa.SARIMAX(data[i],order=best_sarima_order,
                                     seasonal_order=(0,1,0,seasonal_period)).fit())
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [36]:
```
for i in range(ndata):
    print("**********************************************************")
    print("**********************************************************")
    print("the best seasonal arma model for data %d" % i)
    print(sarma_model[i].params)
```

```
**********************************************************
**********************************************************
the best seasonal arma model for data 0
ar.L1    -0.208448
ar.L2    -0.237375
sigma2    0.000031
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 1
ma.L1    -0.145124
sigma2    0.000034
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 2
ma.L1    -0.097685
sigma2    0.000094
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 3
ar.L1     0.128432
ar.L2    -0.003670
ar.L3     0.373477
ma.L1    -0.226414
ma.L2    -0.291575
ma.L3    -0.482224
sigma2    0.000106
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 4
ar.L1     0.121076
ar.L2     0.069689
ar.L3     0.288342
ma.L1    -0.219852
ma.L2    -0.341832
ma.L3    -0.436604
sigma2    0.000102
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 5
ar.L1    -0.716112
ma.L1     0.890948
ma.L2    -0.131242
ma.L3    -0.564571
sigma2    0.000280
dtype: float64
**********************************************************
**********************************************************
the best seasonal arma model for data 6
ar.L1     0.005349
ar.L2     0.506354
ma.L1    -0.177639
ma.L2    -0.675927
ma.L3    -0.146233
sigma2    0.000265
dtype: float64
```

Conclusion: Does the S&P500 data has seasonality? Yes, but it is only obvious in monthly(or rougher sampling frequency) data (see (c) ii)monthly data experiment). For daily data, we would examine the forecasting accuracy in the following question

(e) Using each of the models that you constructed forecast the nextfive observations. Pay attention to the holidays (i.e., you can't forecast Saturdays and Sundays return). You should have 4 models for each dataset. Do the best models in each category (AR,MA, etc.) depend to a particular time period length?

i) forecast the nextfive observations and calculate the sum squre errors

In [37]:
```
test = pd.read_csv("CRSP_on_SP500_test.csv")
preprocess_wrds_data(test)
test_data = test['log_return'].loc[test['log_return'].index>datetime.datetime(2017,11,30)]
test_data.head()
```

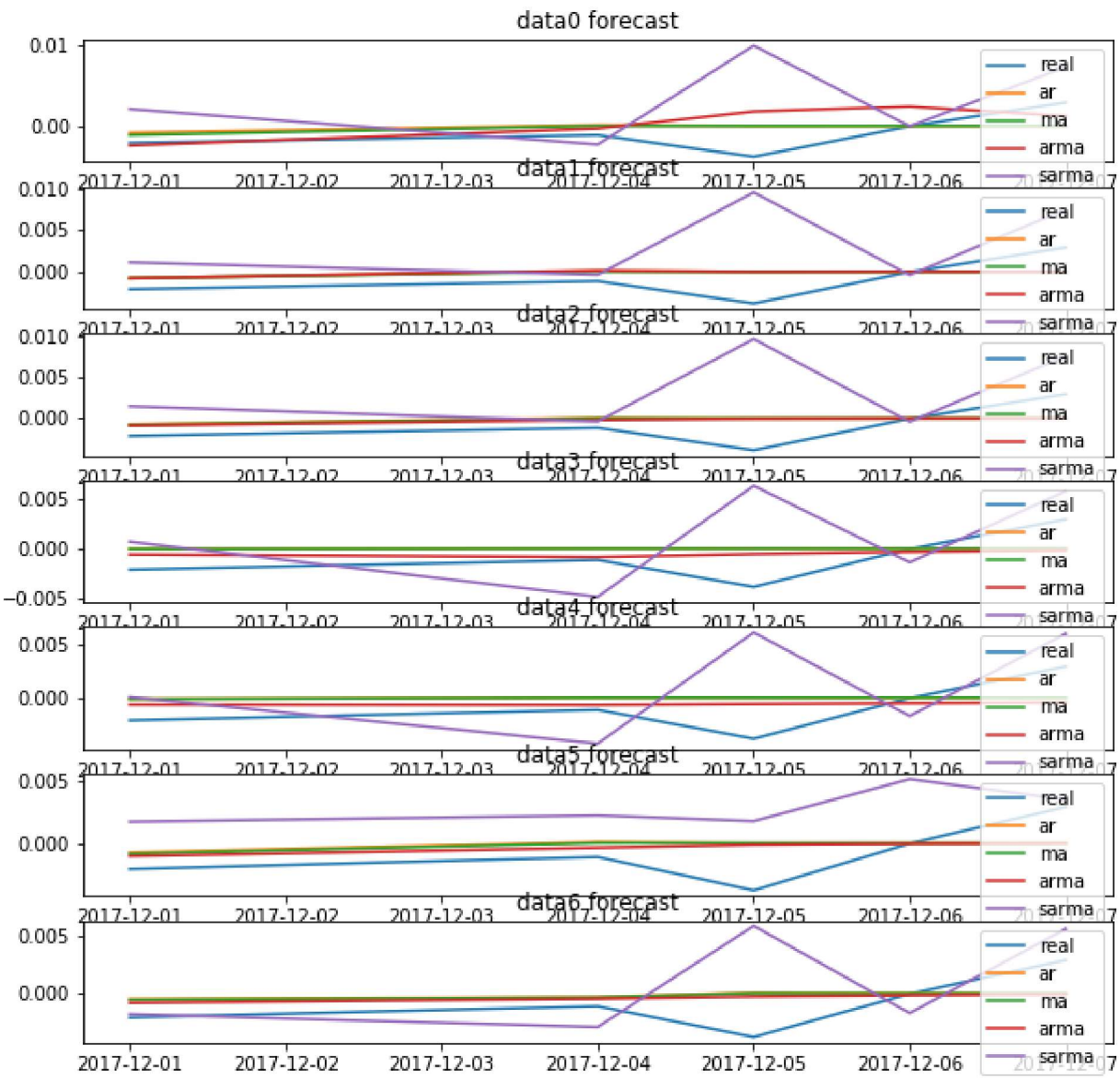
Out[37]:
```
date
2017-12-01   -0.002121
2017-12-04   -0.001137
2017-12-05   -0.003843
2017-12-06   -0.000042
2017-12-07    0.002926
Name: log_return, dtype: float64
```

In [38]:
```
forecast_data=test_data.copy()
mod_col_list = ["real_log_return"]
for i in range(7):
    arma_out_sample_predict = arma_model[i].predict(start=len(data[i])-1,end=len(data[i])+4,dynamic=False)[1
:]
    ar_out_sample_predict = ar_model[i].predict(start=len(data[i])-1,end=len(data[i])+4,dynamic=False)[1:]
    ma_out_sample_predict = ma_model[i].predict(start=len(data[i])-1,end=len(data[i])+4,dynamic=False)[1:]
    sarma_out_sample_predict = sarma_model[i].predict(start=len(data[i])-1,end=len(data[i])+4,dynamic=False)[
1:]

    arma_out_sample_predict.index=test_data.index[:len(arma_out_sample_predict)]
    ar_out_sample_predict.index=test_data.index[:len(ar_out_sample_predict)]
    ma_out_sample_predict.index=test_data.index[:len(ma_out_sample_predict)]
    sarma_out_sample_predict.index=test_data.index[:len(sarma_out_sample_predict)]

    forecast_data = pd.concat([forecast_data,
                       ar_out_sample_predict,
                       ma_out_sample_predict,
                       arma_out_sample_predict,
                       sarma_out_sample_predict],axis=1)
    mod_col_list += ["ar_pred_data%d"%i,"ma_pred_data%d"%i,"arma_pred_data%d"%i,"sarma_pred_data%d"%i]
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:577: F
utureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In
future, it will be treated as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```

In [39]:
```
forecast_data.dropna(inplace=True,axis=0)
forecast_data.columns = mod_col_list
# print(forecast_data)
```

In [40]:
```
plt.figure(figsize=(10,10))
for j in range(7):
    plt.subplot(7,1,j+1)
    forecast = forecast_data.iloc[:,[0]+[i+(j*4+1) for i in range(4)]]
    plt.plot(forecast)
    plt.title("data%d forecast"%j)
    plt.legend(["real","ar","ma","arma","sarma"],loc=1)
plt.show()
```

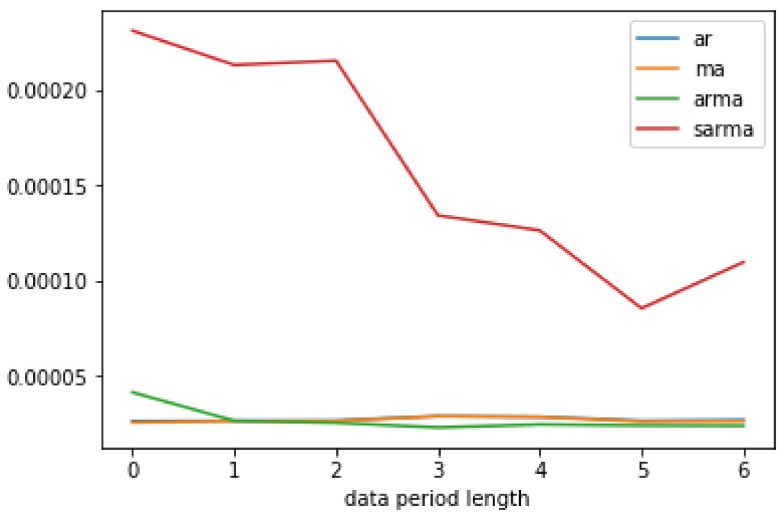


Definition

$$square\ error := (predicted\ value_5 - real\ value_5)^2$$

$$mean\ square\ error := \frac{1}{5}\sum_{i=1}^{5}(predicted\ value_i - real\ value_i)^2$$

In [41]:
```
forecast_errors=pd.DataFrame()
for j in range(7):
    for i in range(4):
        forecast_errors[mod_col_list[j*4+i+1]] = forecast_data.iloc[:,i+(j*4+1)]-forecast_data.iloc[:,0]
# print(forecast_errors)
forecast_sum_squre_errors = []
for j in range(7):
    for i in range(4):
        forecast_sum_squre_errors.append(forecast_errors.iloc[:,j*4+i].apply(np.square).sum())
# print(forecast_sum_squre_errors)
ar_errors = []
ma_errors = []
arma_errors = []
sarma_errors = []
for j in range(7):
    ar_errors.append(forecast_sum_squre_errors[j*4 + 0])
    ma_errors.append(forecast_sum_squre_errors[j*4 + 1])
    arma_errors.append(forecast_sum_squre_errors[j*4 + 2])
    sarma_errors.append(forecast_sum_squre_errors[j*4 + 3])


plt.plot(ar_errors)
plt.plot(ma_errors)
plt.plot(arma_errors)
plt.plot(sarma_errors)
plt.legend(["ar","ma","arma","sarma"])
plt.xlabel("data period length")
plt.show()
```



In [42]:
```
best_model_for_each_period = []
models = ["ar","ma","arma","sarma"]
for i in range(7):
    idx = np.argmin(np.array([ar_errors,ma_errors,arma_errors,sarma_errors])[:,i])
    best_model_for_each_period.append(models[idx])
print(best_model_for_each_period)
```

```
['ma', 'ma', 'arma', 'arma', 'arma', 'arma', 'arma']
```

Conclusion: SARMA model is bad. AR, MA and ARMA perform similar. but we can see as the data period size increase, more complex model performs better than the simpler one

ii) Do the best models in each category (AR,MA, etc.) depend to a particular time period length?

Based on the BIC criterion. The answer is Yes, because the time period length determined the sample size. The large the sample size, the finner the model can be tunned since there is more information to utilize.

From (c), we can see, as time period increase, the best AR model changes from AR(1) to AR(2). the best MA model changes from MA(1) to MA(2), the best ARMA model changes from ARMA(0,1) or ARMA(1,0) to ARMA(1,1)

From (d), the best SARMA model changes from seasnonal AR(2) to seasonal AR(2,3). For the following, I will construct forecast of the next 5 observations and make the comparison.

(f) 5 working days afterthe date you downloaded your equity data, go ahead and download the same equity data again. Test theperformance of each model by comparing the forecasted returns(and prices) with the real observations collected after 5 workingdays. Calculate the squared error of your forecast for each model and the absolute value of your forecast. What was the best model? Alternately you could set aside the last 5 days of your data and only use those for the evaluation of the forecast.

i) Download further more 5 datas and construct the forecasted returns(and prices).

In [43]:
```
test = pd.read_csv("CRSP_on_SP500_test.csv")
preprocess_wrds_data(test)
test_data = test['log_return'].loc[test['log_return'].index>datetime.datetime(2017,11,30)]
test_data[5:10]
```

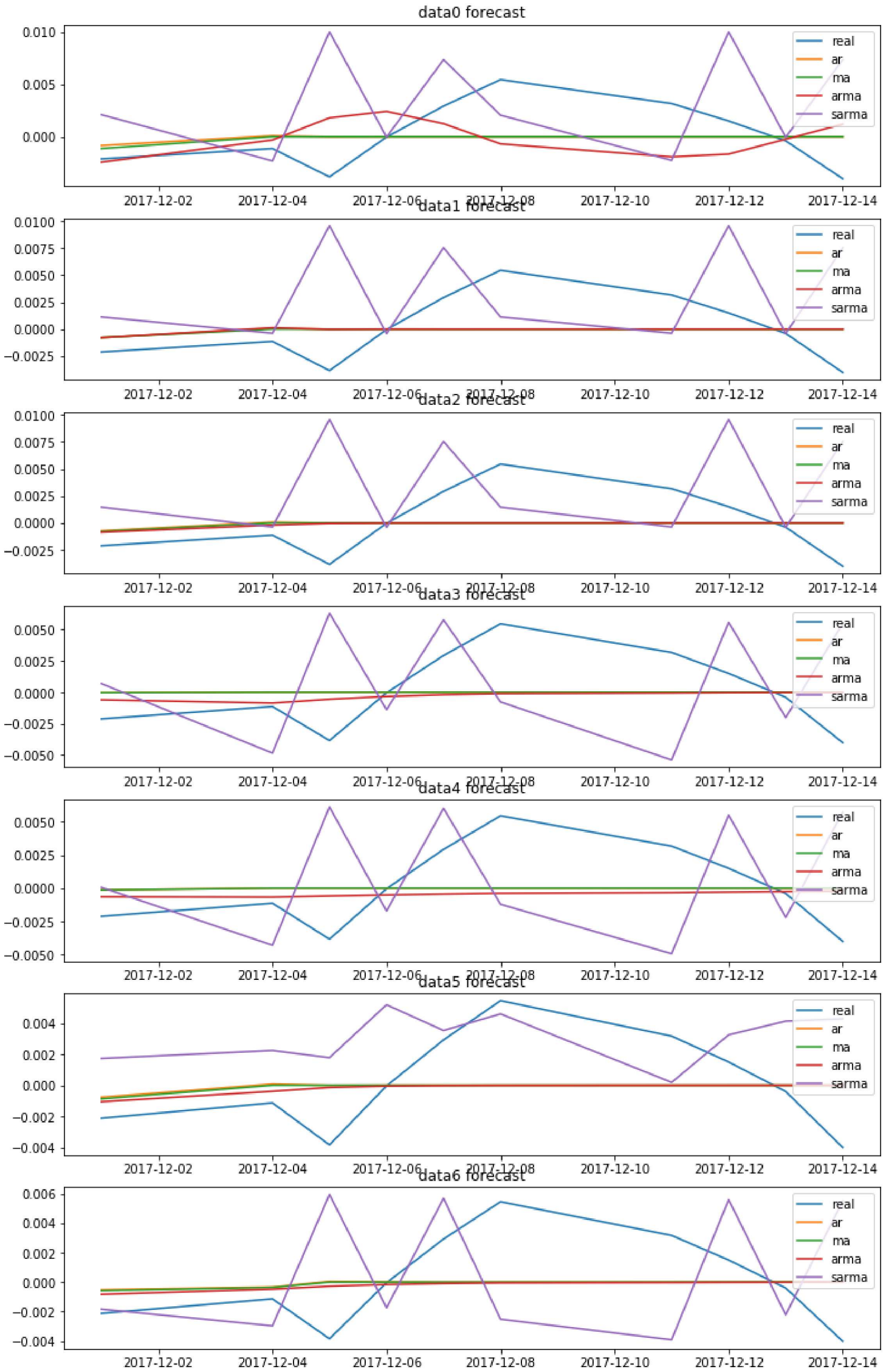
Out[43]:
```
date
2017-12-08    0.005454
2017-12-11    0.003170
2017-12-12    0.001499
2017-12-13   -0.000400
2017-12-14   -0.004012
Name: log_return, dtype: float64
```

In [44]:
```
forecast_data=test_data.copy()
mod_col_list = ["real_log_return"]
for i in range(7):
    arma_out_sample_predict = arma_model[i].predict(start=len(data[i])-1,end=len(data[i])+9,dynamic=False)[1:]
    ar_out_sample_predict = ar_model[i].predict(start=len(data[i])-1,end=len(data[i])+9,dynamic=False)[1:]
    ma_out_sample_predict = ma_model[i].predict(start=len(data[i])-1,end=len(data[i])+9,dynamic=False)[1:]
    sarma_out_sample_predict = sarma_model[i].predict(start=len(data[i])-1,end=len(data[i])+9,dynamic=False)[1:]

    arma_out_sample_predict.index=test_data.index[:len(arma_out_sample_predict)]
    ar_out_sample_predict.index=test_data.index[:len(ar_out_sample_predict)]
    ma_out_sample_predict.index=test_data.index[:len(ma_out_sample_predict)]
    sarma_out_sample_predict.index=test_data.index[:len(sarma_out_sample_predict)]

    forecast_data = pd.concat([forecast_data,
                       ar_out_sample_predict,
                       ma_out_sample_predict,
                       arma_out_sample_predict,
                       sarma_out_sample_predict],axis=1)
    mod_col_list += ["ar_pred_data%d"%i,"ma_pred_data%d"%i,"arma_pred_data%d"%i,"sarma_pred_data%d"%i]
#     if(i>=2):
#         seasonal_period=252
#         seasonal_difference = (data[i] - data[i].shift(seasonal_period)).dropna()
#         sarma_size=len(seasonal_difference)
#         sarma_out_sample_predict = sarma_model[i-2].predict(start=sarma_size-1,end=sarma_size+4,dynamic=False)[1:]
#         sarma_out_sample_predict += data[i][sarma_size-seasonal_period:sarma_size-seasonal_period+5].values
#         sarma_out_sample_predict.index=test_data.index[:len(sarma_out_sample_predict)]
#         forecast_data = pd.concat([forecast_data, sarma_out_sample_predict],axis=1)
#         mod_col_list += ["sarma_pred_data%d"%i]
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/kalmanf/kalmanfilter.py:577: FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```

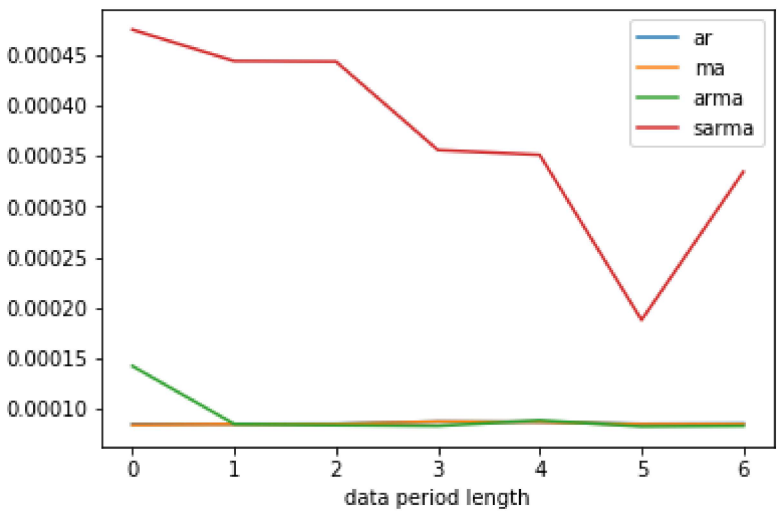
In [45]:
```
forecast_data.dropna(inplace=True,axis=0)
forecast_data.columns = mod_col_list
# print(forecast_data)
```

In [46]:
```
plt.figure(figsize=(12,20))
for j in range(7):
    plt.subplot(7,1,j+1)
    forecast = forecast_data.iloc[:,[0]+[i+(j*4+1) for i in range(4)]]
    plt.plot(forecast)
    plt.title("data%d forecast"%j)
    plt.legend(["real","ar","ma","arma","sarma"],loc=1)
plt.show()
```



ii) Calculate the sum squre errors

In [47]:
```
forecast_errors=pd.DataFrame()
for j in range(7):
    for i in range(4):
        forecast_errors[mod_col_list[j*4+i+1]] = forecast_data.iloc[:,i+(j*4+1)]-forecast_data.iloc[:,0]
# print(forecast_errors)
forecast_sum_squre_errors = []
for j in range(7):
    for i in range(4):
        forecast_sum_squre_errors.append(forecast_errors.iloc[:,j*4+i].apply(np.square).sum())
# print(forecast_sum_squre_errors)
ar_errors = []
ma_errors = []
arma_errors = []
sarma_errors = []
for j in range(7):
    ar_errors.append(forecast_sum_squre_errors[j*4 + 0])
    ma_errors.append(forecast_sum_squre_errors[j*4 + 1])
    arma_errors.append(forecast_sum_squre_errors[j*4 + 2])
    sarma_errors.append(forecast_sum_squre_errors[j*4 + 3])


plt.plot(ar_errors)
plt.plot(ma_errors)
plt.plot(arma_errors)
plt.plot(sarma_errors)
plt.legend(["ar","ma","arma","sarma"])
plt.xlabel("data period length")
plt.show()
```



iii) What is the best model?

In [48]:
```
best_model_for_each_period = []
models = ["ar","ma","arma","sarma"]
for i in range(7):
    idx = np.argmin(np.array([ar_errors,ma_errors,arma_errors,sarma_errors])[:,i])
    best_model_for_each_period.append(models[idx])
print(best_model_for_each_period)
```

```
['ma', 'ma', 'arma', 'arma', 'ma', 'arma', 'arma']
```

In [49]:
```
idx = np.argmin(forecast_sum_squre_errors)
mod_col_list[idx+1]
```

Out[49]: 'arma_pred_data5'

Conclusion: SARMA model is bad. AR, MA and ARMA perform similar. but we can see as the data period size increase, more complex model performs better than the simpler one (The best model change from ma model to arma model)

(g) Was there a difference in the time periods (i.e., is there a type of model that consistently outperformed the others for all the datasets under consideration)?

- There is no evidence that a type of model can outperform others for all periods.
- Complex model (e.g. ARMA) fit better for Long period (more observations) whereas simple one (e.g. MA(1) model) fit better for short period (less observations).
- Seasonal model should only be used when there is a significant seasonal effect. And I think monthly return data should work better for more sophisticated models as they are not that random (See (d) experiment ii) monthly return data)

(h) Are the perceived differences between models due to chance or there is a model that performed significantly better? (Hint: Use the standard error of the forecast as given by the R output.)

From the experiment in (e) and (f), we got the sum squre error of the forecast. There is no evidence that there is a model performs significantly better. Or, there is no model works really well. I suppose it is caused by the large noise in daily return series. Predict daily return data only based on history data is not wise. However, if one don't know which one to use, MA(1) seems just suitable in any case. But the defect is that the prediction always fall to zero as time increases. Predicting return=0 is not valuable in practice.

In conclusion, I think we need more information/factors (e.g. volumn, liquidity, wage) models in stock prediction rather than just history data.

3. (100 points) Please use the high frequency data provided in "INTCdata.xls" (minute data for Intel Corporation). The data is from the period June 19-June 30 2006. The data columns contain: Ticker, time stamp, Open, High, Low, Close, Volume, Nr. of trades, Average price all within the minute. You may use whichever column you wish as data input. These are two weeks (notice that there are only 10 days worth of data due to the weekend).

(a) Divide the data into two parts: one (larger) part would be used to construct the models, and the later part would be used to evaluate the model performance. It is up to you how to divide the data to evaluate the performance of the models but you need to decide on a model which you consider to be the best.

We will use the first 9 days average price to construct the model and test the last day data.

In [50]:
```
df2 = pd.read_csv("INTCdata.csv")
print(df2.size)
df2.head()
```

```
42900
```

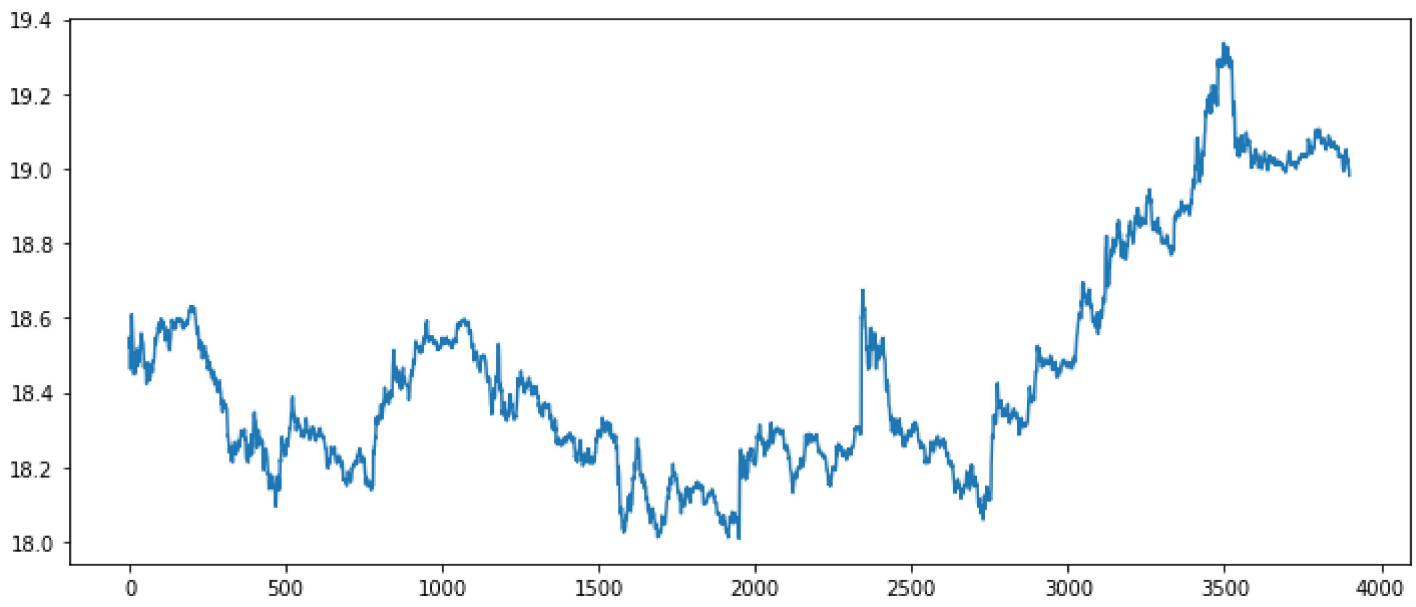
Out[50]:

	ticker	time_stamp	open	high	low	close	volumn	NR	average	Unnamed: 9	Unnamed: 10
0	INTC	930	18.51	18.55	18.51	18.55	1079160	816	18.5226	7	NaN
1	INTC	931	18.55	18.57	18.53	18.55	382424	623	18.5490	2	NaN
2	INTC	932	18.56	18.56	18.45	18.46	460394	698	18.5053	3	NaN
3	INTC	933	18.45	18.50	18.41	18.49	376212	577	18.4632	3	NaN
4	INTC	934	18.50	18.53	18.48	18.52	323147	347	18.5060	3	NaN

In [51]:
```
def train_test_split(array_like,train_size):
    array_size = array_like.size
    train_set = array_like[:int(train_size*array_size)]
    test_set = array_like[int(train_size*array_size):]
    return train_set,test_set
```

In [52]:
```
hf_data = df2.average
hf_data_train, hf_data_test = train_test_split(df2.average,0.9)
```

In [53]:
```
plt.figure(figsize=(12,5))
plt.plot(hf_data)
plt.show()
```



(b) Construct the best model you can with the material taught until now (and including seasonal time series). You should discuss a lot. Pay attention to the fact that the data contains a weekend.

In [54]:
```
unit_root_cheking(hf_data_train,'c')
```

```
Test Statistic                  0.066744
p-value                         0.963716
#Lags Used                      1.000000
Number of Observations Used    3508.000000
Critical Value (1%)              -3.432215
Critical Value (5%)              -2.862364
Critical Value (10%)             -2.567209
dtype: float64
```

The price data has unit root, so we consider the log return as follows

In [55]:
```
hf_log_return_train = np.log(hf_data_train/hf_data_train.shift(1)).dropna(inplace=False)
unit_root_cheking(hf_log_return_train,'c')
```

```
Test Statistic                 -49.840030
p-value                          0.000000
#Lags Used                       0.000000
Number of Observations Used    3508.000000
Critical Value (1%)              -3.432215
Critical Value (5%)              -2.862364
Critical Value (10%)             -2.567209
dtype: float64
```

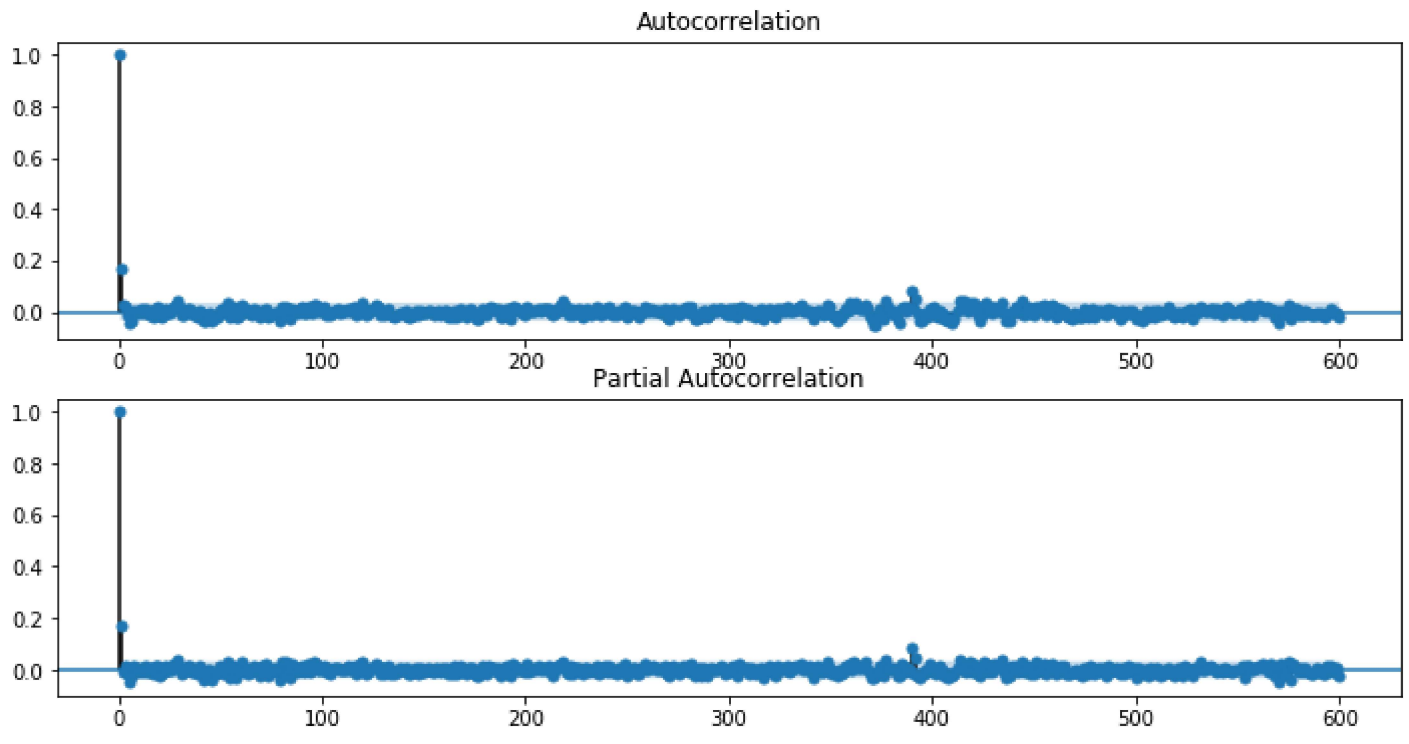
The log return reject the unit root. So it is stationary and we use this series to build our model Correspondingly, we redefine the train test dataset obtained in (a) (using return series instead)

In [56]:
```
hf_returns = (np.log(df2.average)-np.log(df2.average.shift(1))).dropna()
hf_data_train,hf_data_test = train_test_split(hf_returns,0.9)
```

i) Seasonal ARMA model

ACF and PACF to show the marginal AR, MA factors

In [57]:
```
plt.figure(figsize=(12,6))

ax1 = plt.subplot(2,1,1)
sm.graphics.tsa.plot_acf(hf_data_train, alpha=0.05, lags=600, ax=ax1)

ax2 = plt.subplot(2,1,2)
sm.graphics.tsa.plot_pacf(hf_data_train, alpha=0.05, lags=600, ax=ax2)
plt.show()
```



From this ACF PACF plot, we find that the values of acf and pacf are significant at lag=390. Therefore, when considering seasonal arima model, it is natural to use seasonal period=390

In [58]:
```
hf_acf,hf_confit = sm.tsa.stattools.acf(hf_log_return_train,alpha=0.05,nlags=400)
print(np.argsort(np.abs(hf_acf))[-10:])
print(hf_acf[np.argsort(np.abs(hf_acf))[-10:]])
```

```
[384  29 218   5 372 392 371 390   1   0]
[-0.04058009  0.04302714  0.04352219 -0.04629441 -0.04847878  0.05071594
 -0.05166691  0.0850957   0.17117321  1.        ]
```

There is a special bump at lag=390 (ranked 3rd), might indicate the daily seasonality (6.5h)

In [59]:
```
hf_data_train = hf_data_train.values
```

One should use SARIMA model or ARIMA model with seasonalily. However, in Python SARIMAX is a newly added class, and it has a deadly defect that the time consumed explode when seasonal period increase. And seasonality larger than 100 can cause my PC crash

I try to execute

sarima_model = sm.tsa.SARIMAX(hf_data_train,order=(1,1,1),seasonal_order=(0,1,0,390)).fit()

and then, my computer crashed. From now on, use the reduced ARIMA model and then inverse the seasonal difference

PS: it is equivalent to the following code in R:

arima(Y,order=c(1,1,1), seasonal = list(order = c(0, 1, 0), period=390))

For the following, we use the reduced ARIMA model and applied to seasonal difference series to replicate the SARIMA model. They are equivalent. And we define a inverse function to inverse the prediction in SARIMA model

In [60]:

```
sarima_bic_matrix = np.zeros((4,4))
hf_data_train_seasonal_difference = hf_data_train[390:] - hf_data_train[:-390]
for i in range(4):
    for j in range(4):
        try:
            sarima_model = sm.tsa.SARIMAX(hf_data_train_seasonal_difference,order=(i,0,j)).fit()
            sarima_bic_matrix[i,j] = sarima_model.bic
        except:
            pass
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/statespace/tools.py:405: Runti
meWarning: invalid value encountered in sqrt
  x = r / ((1 - r**2)**0.5)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [61]:

```
print(sarima_bic_matrix)
```

```
[[     0.         -34817.61467308 -34809.54585097 -34801.60720986]
 [-34813.85637966 -34809.61925133 -34801.29115046 -34793.34856378]
 [-34809.04320172 -34798.88631101 -34791.72590824             nan]
 [-34802.4500637  -34794.25521669      0.                     nan]]
```

In [62]:

```
find_min_idx(sarima_bic_matrix)
```

Out[62]:

```
(0, 1)
```

Based on the BICs, we conclude that SARIMA (0,0,1)×(0,1,0,390) is the best model for log return series

In [63]:

```
seasonal_first_difference = hf_data_train[390:]-hf_data_train[:-390]
sarima_model = sm.tsa.SARIMAX(seasonal_first_difference,order=(0,0,1)).fit()
```

In [64]:

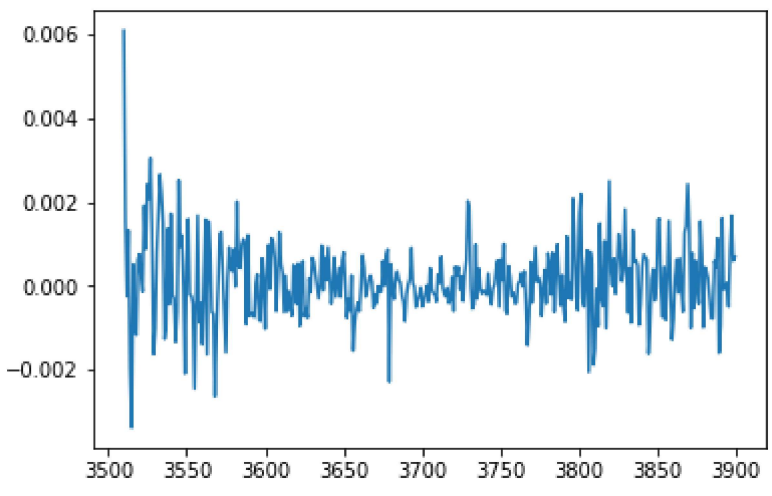
```
def inverse_seasonal_difference(pred,history,seasonal_period):
    history_size = len(history)
    return pred + history[history_size-seasonal_period:history_size-seasonal_period+len(pred)]
```

In [65]:

```
train_size = len(seasonal_first_difference)
test_size = len(hf_data_test)
# print(train_size)
pred = sarima_model.predict(start=train_size,end = train_size + test_size - 1)
# print(pred.size)
return_pred = inverse_seasonal_difference(pred,hf_data_train,390)
```

In [66]:

```
sarima_residual = return_pred - hf_data_test
plt.plot(sarima_residual)
plt.show()
```



The Mean Square Error for the asscociated forecast

In [67]:

```
sarima_mse = sarima_residual.apply(np.square).mean()
print("the mean square error of arima",sarima_mse)
```

```
the mean square error of arima 8.864715399677503e-07
```

ii) AR, MA, ARMA model

In [68]:

```
arima_bic_matrix = np.zeros((4,4))
for i in range(4):
    for j in range(4):
        try:
            arima_model = sm.tsa.SARIMAX(hf_data_train,order=(i,0,j)).fit()
            arima_bic_matrix[i,j] = arima_model.bic
        except:
            pass
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/tsa/statespace/tools.py:405: Runti
meWarning: invalid value encountered in sqrt
  x = r / ((1 - r**2)**0.5)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [69]:

```
print(arima_bic_matrix)
```

```
[[     0.         -41182.13525686 -41174.66574498 -41168.25533936]
 [-41183.55671413 -41175.31417379 -41156.84544766 -41159.19747229]
 [-41175.46337991      0.              0.                     nan]
 [-41168.22126698 -41160.03083628      0.                     nan]]
```

In [70]:

```
def arima_model_screening(bic_matrix):
    best_ar_order = np.nanargmin(bic_matrix[:][0])
#     print(best_ar_order)
    ari_model=sm.tsa.SARIMAX(hf_data_train,order=(best_ar_order,0,0)).fit()

    best_ma_order = np.nanargmin(bic_matrix[0][:])
#     print(best_ma_order)
    ima_model=sm.tsa.SARIMAX(hf_data_train,order=(0,0,best_ma_order)).fit()

    best_arma_order = find_min_idx(bic_matrix[1:,1:])
    best_arma_order = (best_arma_order[0]+1,best_arma_order[1]+1)
#     print(best_arma_order)
    arima_model=sm.tsa.SARIMAX(hf_data_train,order=(best_arma_order[0],0,best_arma_order[1])).fit()
    return (ari_model,ima_model,arima_model)
```

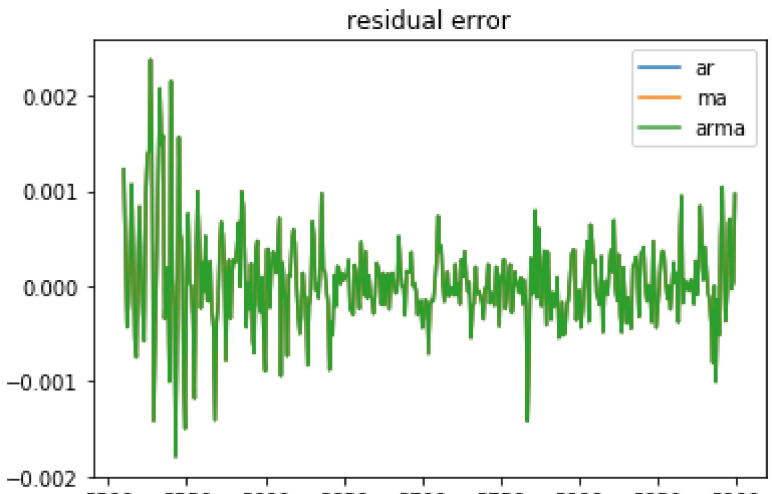
In [71]:

```
bic_matrix = arima_bic_matrix
ar_model,ma_model,arma_model = arima_model_screening(arima_bic_matrix)
```

```
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
/home/jerryxyx/anaconda3/envs/py36/lib/python3.6/site-packages/statsmodels/base/model.py:496: ConvergenceWarn
ing: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

In [72]:
```
train_size = len(hf_data_train)
test_size = len(hf_data_test)
ar_pred = ar_model.predict(start=train_size,end=test_size+train_size-1)
ma_pred = ma_model.predict(start=train_size,end=test_size+train_size-1)
arma_pred = arma_model.predict(start=train_size,end=test_size+train_size-1)
ar_residual = ar_pred - hf_data_test
ma_residual = ma_pred - hf_data_test
arma_residual = arma_pred - hf_data_test
```

In [73]:
```
plt.plot(ar_residual)
plt.plot(ma_residual)
plt.plot(arma_residual)
plt.legend(["ar","ma","arma"])
plt.title("residual error")
plt.show()
```



In [74]:
```
ar_mse = ar_residual.apply(np.square).mean()
ma_mse = ma_residual.apply(np.square).mean()
arma_mse = arma_residual.apply(np.square).mean()

print("the mean square error of ar",ar_mse)
print("the mean square error of ma",ma_mse)
print("the mean square error of arma",arma_mse)
print("the mean square error of sarima",sarima_mse)
```

```
the mean square error of ar 2.370915998107467e-07
the mean square error of ma 2.3705253760672548e-07
the mean square error of arma 2.3707788776799288e-07
the mean square error of sarima 8.864715399677503e-07
```

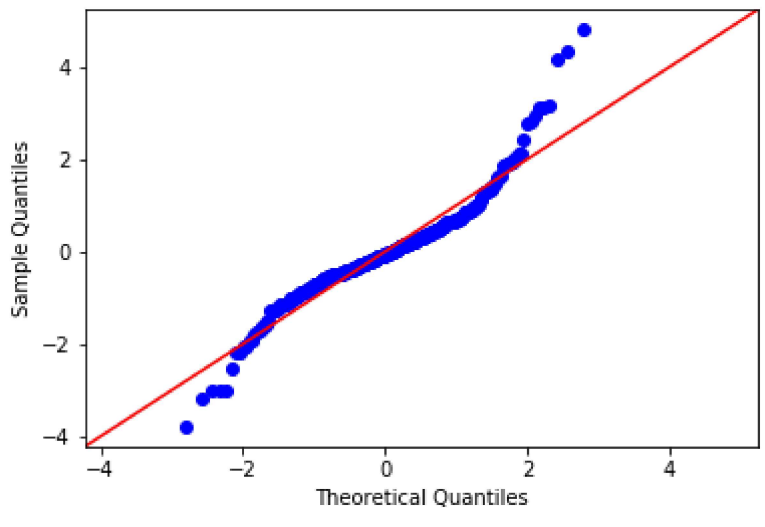
Conclusion, even we know the seasonal period is 390, seasonal ARIMA model is not that good under the criterion of mean square error. The best model in this case is MA(1)

(c) After constructing (what you consider to be) the best model, obtain the residuals and test if they are likely to be white noise. Use (and name) all the testing procedures for time series you learned for this purpose.

I consider MA(1) as the best model based on the minimum mean square error in (b)

i) qq-plot

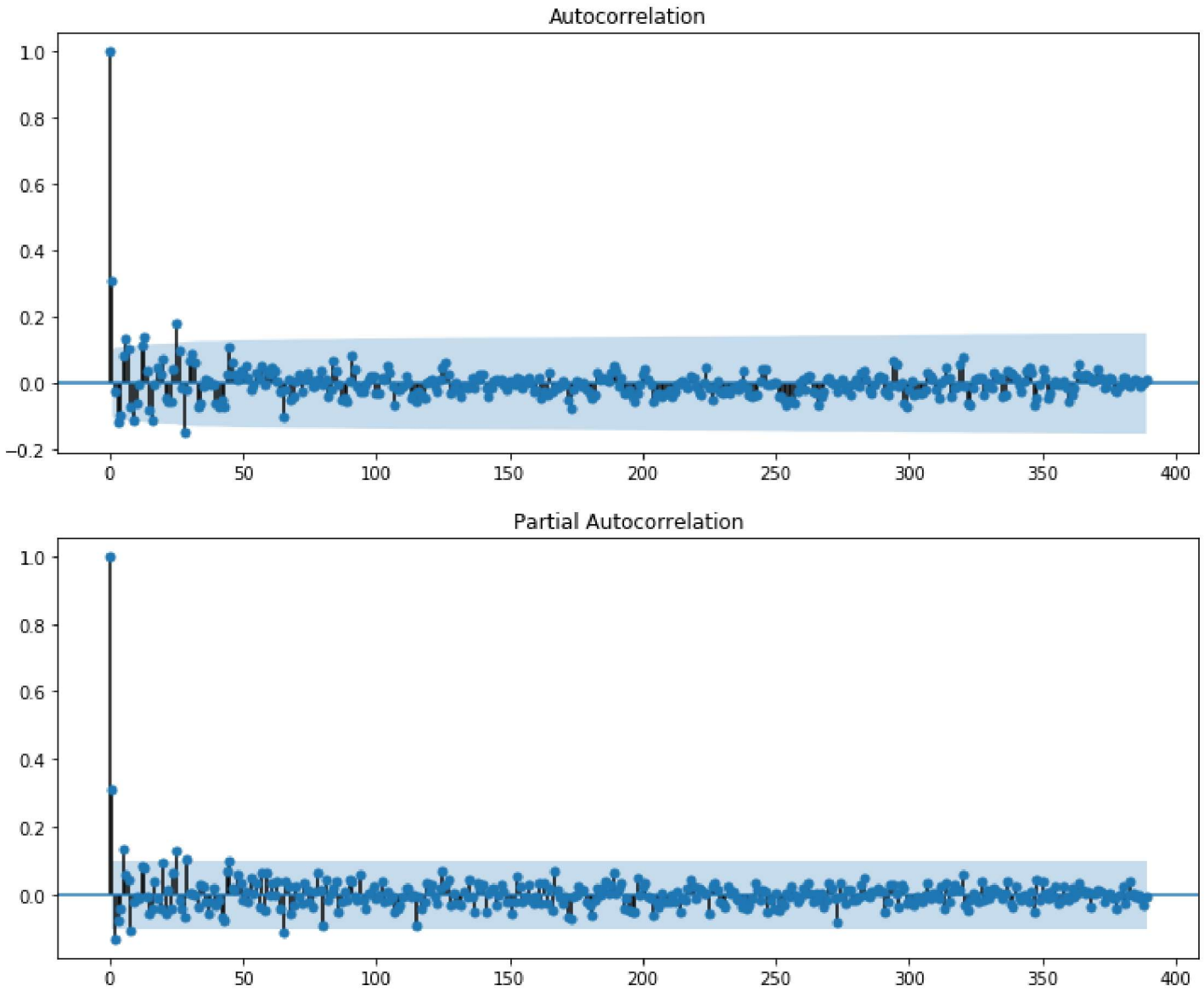
In [75]:
```
sm.qqplot(ma_residual,dist=sp.stats.norm,fit=True,line="45")
plt.show()
```



From the qq-plot, we conclude that the residual series is fat-tailed.

ii) ACF and PACF

In [76]:
```
plt.figure(figsize=(12,10))
ax1 = plt.subplot(2,1,1)
sm.graphics.tsa.plot_acf(ma_residual,ax=ax1,alpha=0.05)
ax1 = plt.subplot(2,1,2)
sm.graphics.tsa.plot_pacf(ma_residual,ax=ax1,alpha=0.05)
plt.show()
```



There are still many significant acf and pacf values. It seems not to be a white noise, because white noise is independently distributed

iii) Ljung-Box test (portmanteau test)

In [77]:
```
statistics,pvalues = sm.stats.acorr_ljungbox(ma_residual,lags=10)
print(pvalues)
```

```
[8.60594521e-10 6.20006352e-09 2.54655517e-09 1.99378124e-09
 1.79645617e-09 1.86016027e-10 7.95660841e-11 1.08877453e-10
 3.26100108e-11 5.06193084e-11]
```

Since all p-values are smaller than 1e-06, we reject that the residual series is independently distributed. So it's not a white noise

iv) Jarque-Bera Normality test

In [78]:
```
from scipy.stats import jarque_bera
jarque_bera(ma_residual)
```

Out[78]: (295.53808372489624, 0.0)

Apparently, the residual is not normal distributed

4. (50 points) For this problem you will study the change in behavior of random walk with parameters. Please decide on a number of observations to be generated which is large enough to actually observe a behavior. The problem looks at two random walks:

- Regular random walk $r_t = r_{t-1} + a_t$, where $r_0 = 0$
- Random walk with drift μ: $r_t = \mu + r_{t-1} + a_t$, where $r_0 = 0$

Please include representative images and comment on the observed behavior. Please note that one model is a subset of the other one. The idea is to study the effect of changing parameters in the model and understand the change in behavior of the process as a result of changing these parameters. You have to come up with representative images and numerical measures that will allow you to illustrated what you perceive.

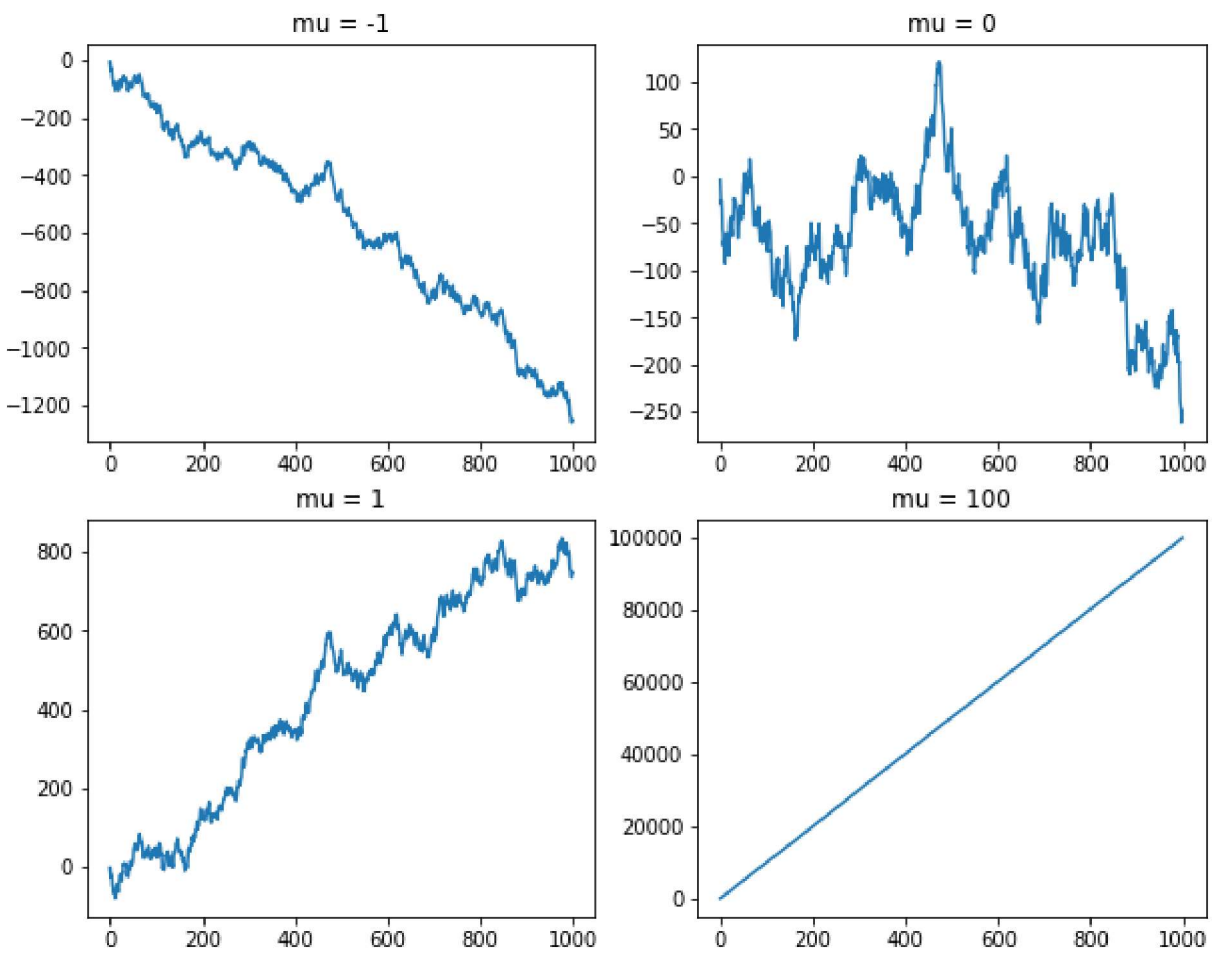
In [79]:
```
def generate_random_walks(at,drift):
    drifted = np.cumsum(at+drift)
    return drifted
```

(a) Study the effect of changing μ. To this end, experiment with $\mu = -1, 0, 1, 100$

In this section, we set scale parameter sigma=10

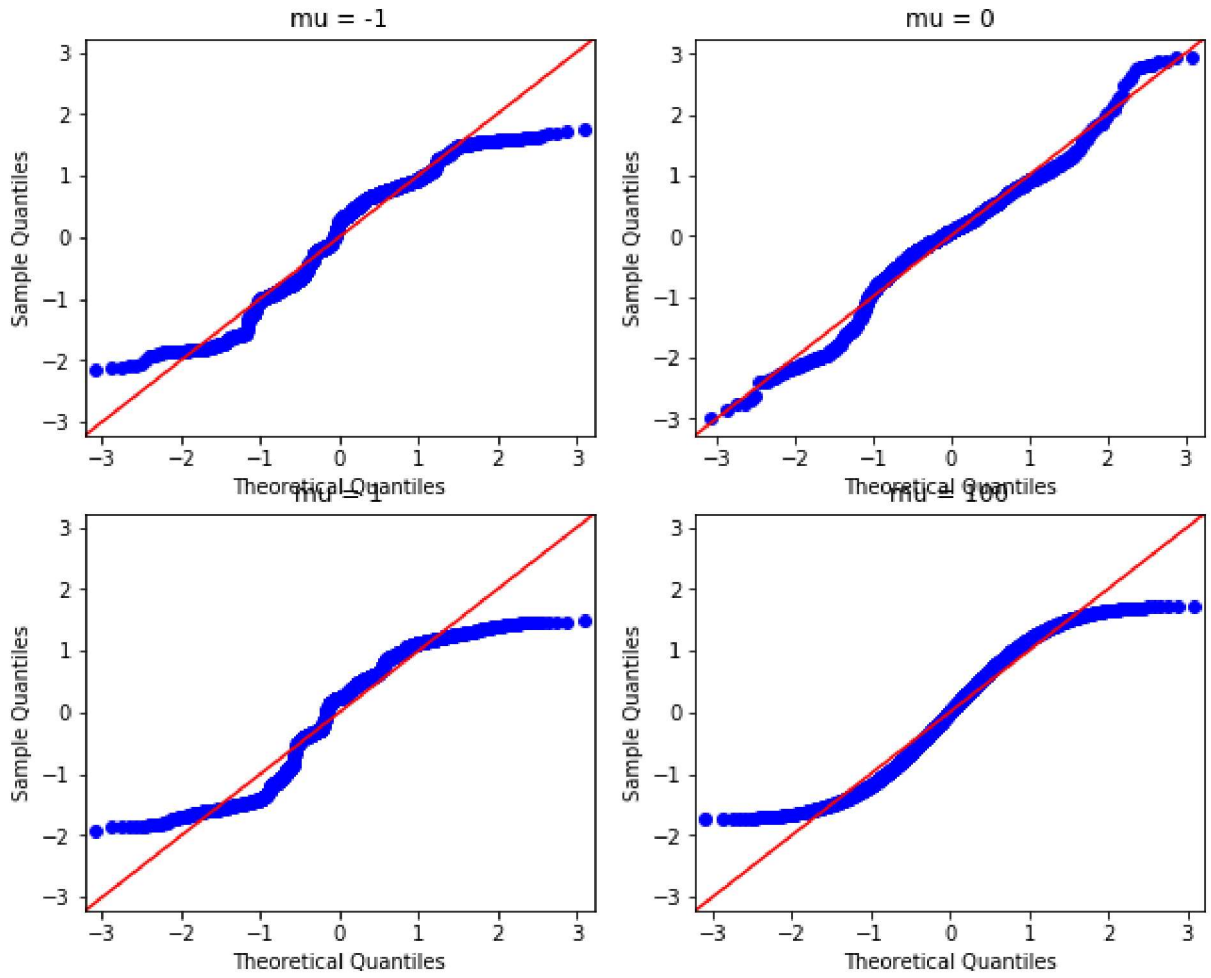
In [80]:
```
sigma = 10
at = sp.stats.norm(0,sigma).rvs(1000)
```

In [81]:
```
plt.figure(figsize=(10,8))
for i,mu in enumerate([-1,0,1,100]):
    plt.subplot(2,2,i+1)
    rw = generate_random_walks(at,mu)
    plt.plot(rw)
    plt.title("mu = %d"%mu)

plt.show()
```



ii) qq-plot of the random walks

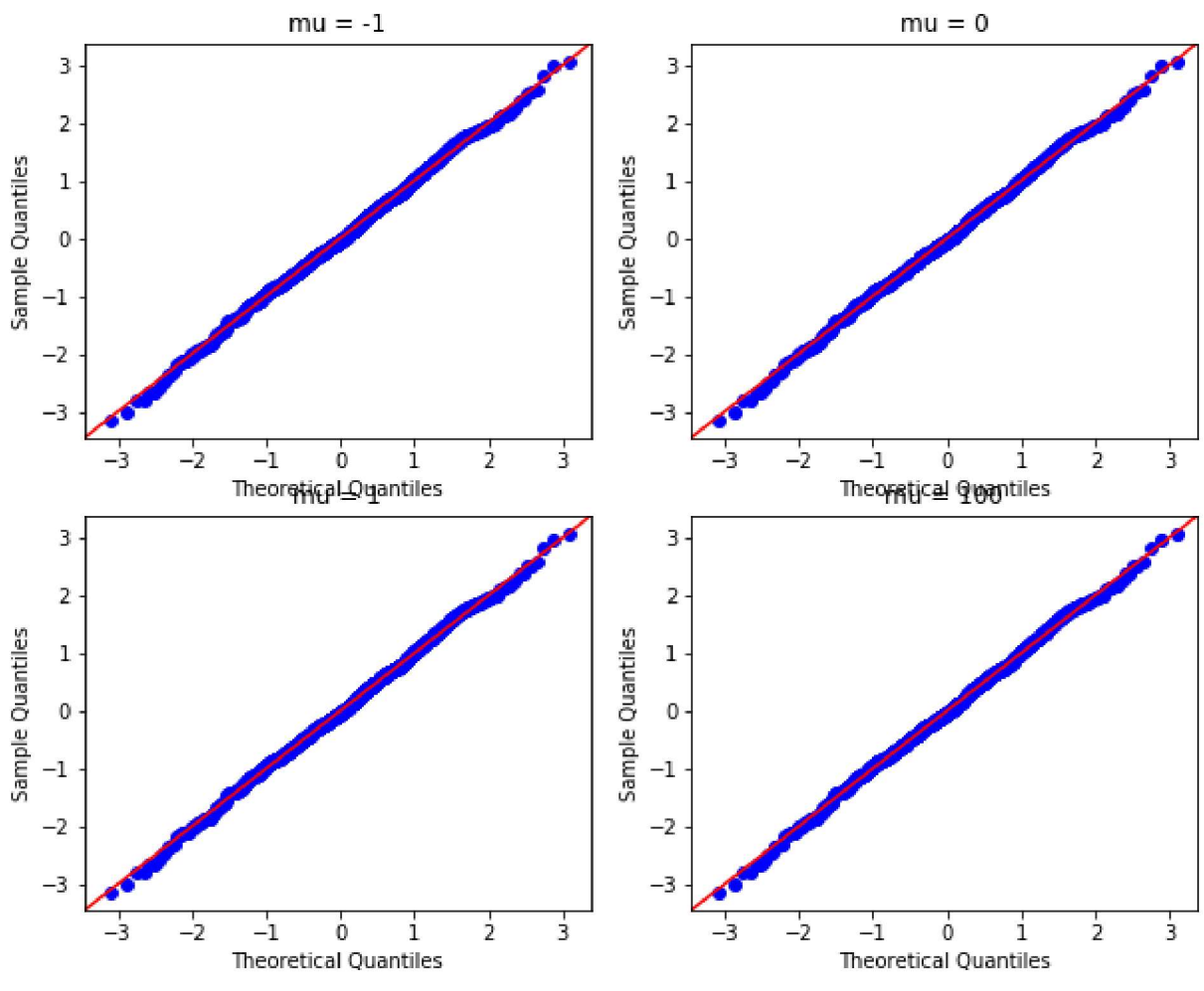
In [82]:
```
plt.figure(figsize=(10,8))
for i,mu in enumerate([-1,0,1,100]):
    ax=plt.subplot(2,2,i+1)
    rw = generate_random_walks(at,mu)
    sm.qqplot(rw,dist=sp.stats.norm,line='45',fit=True,ax=ax)
    plt.title("mu = %d"%mu)

plt.show()
```



Even though qq-plot are fitted by norm distribution, there are still "light-tail" effect in random walks data

iii) fitted qq-plot of the first difference of the random walks

In [83]:
```
plt.figure(figsize=(10,8))
for i,mu in enumerate([-1,0,1,100]):
    ax=plt.subplot(2,2,i+1)
    rw = generate_random_walks(at,mu)
    first_difference = [rw[0]] + rw[1:]-rw[:-1]
    sm.qqplot(first_difference,dist=sp.stats.norm,line='45',fit=True,ax=ax)
    plt.title("mu = %d"%mu)

plt.show()
```



First difference eliminate the random walk effect. It is linear in qq-plot Note that here, we first fitted the series to the normal distribution and then, eliminate the drift effect

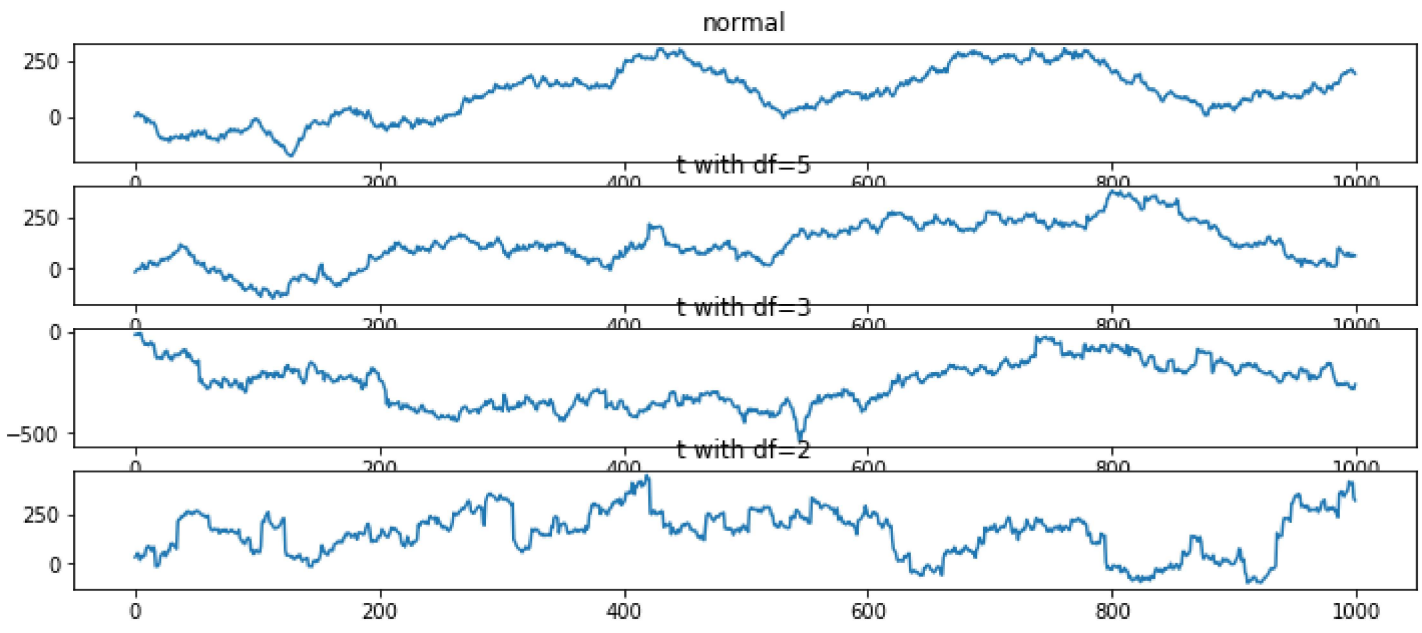
(b) Study the effect of changing the distribution of the white noise. Experiment with an i.i.d. noise distributed as σZ where Z is a standard normal random variable or a t distributed noise with say 5 degrees of freedom.

Since first difference can elimiate the drift effect, in this part, we do not consider drift random walk

i) Raw Data

In [84]:
```
sigma = 10
at1 = sp.stats.norm(0,1).rvs(1000)
at2 = sp.stats.t(df=5).rvs(1000)
at3 = sp.stats.t(df=3).rvs(1000)
at4 = sp.stats.t(df=2).rvs(1000)
```

In [85]:
```
plt.figure(figsize=(12,5))
rw1 = generate_random_walks(sigma*at1,0)
rw2 = generate_random_walks(sigma*at2,0)
rw3 = generate_random_walks(sigma*at3,0)
rw4 = generate_random_walks(sigma*at4,0)

plt.subplot(4,1,1)
plt.plot(rw1)
plt.title("normal")

plt.subplot(4,1,2)
plt.plot(rw2)
plt.title("t with df=5")

plt.subplot(4,1,3)
plt.plot(rw3)
plt.title("t with df=3")

plt.subplot(4,1,4)
plt.plot(rw4)
plt.title("t with df=2")
plt.show()
```

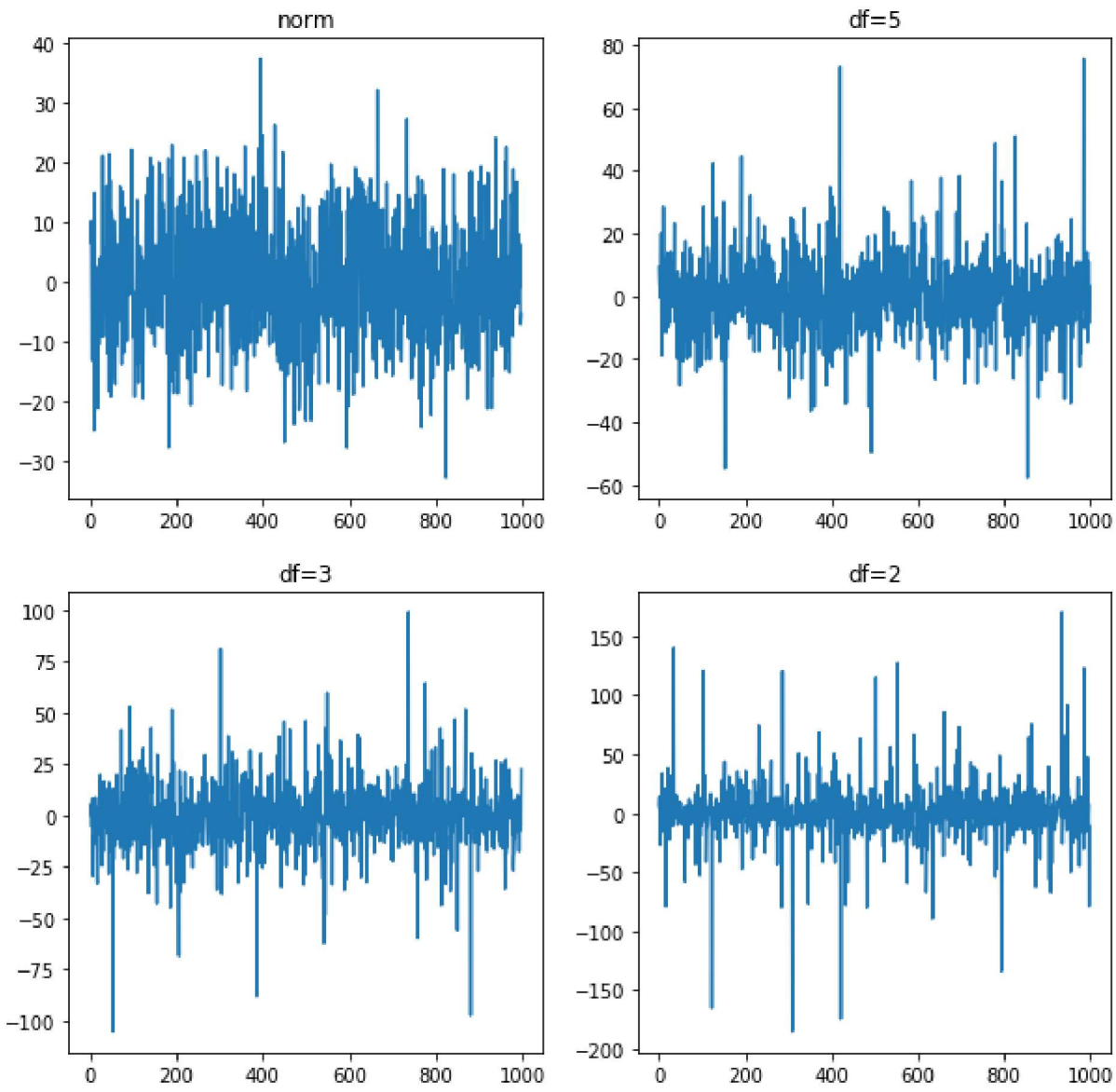


As degree of freedom of t-distribution decrease, the variance increase. When it less or equal than 2, the process behaved like a jump process

ii) First Difference

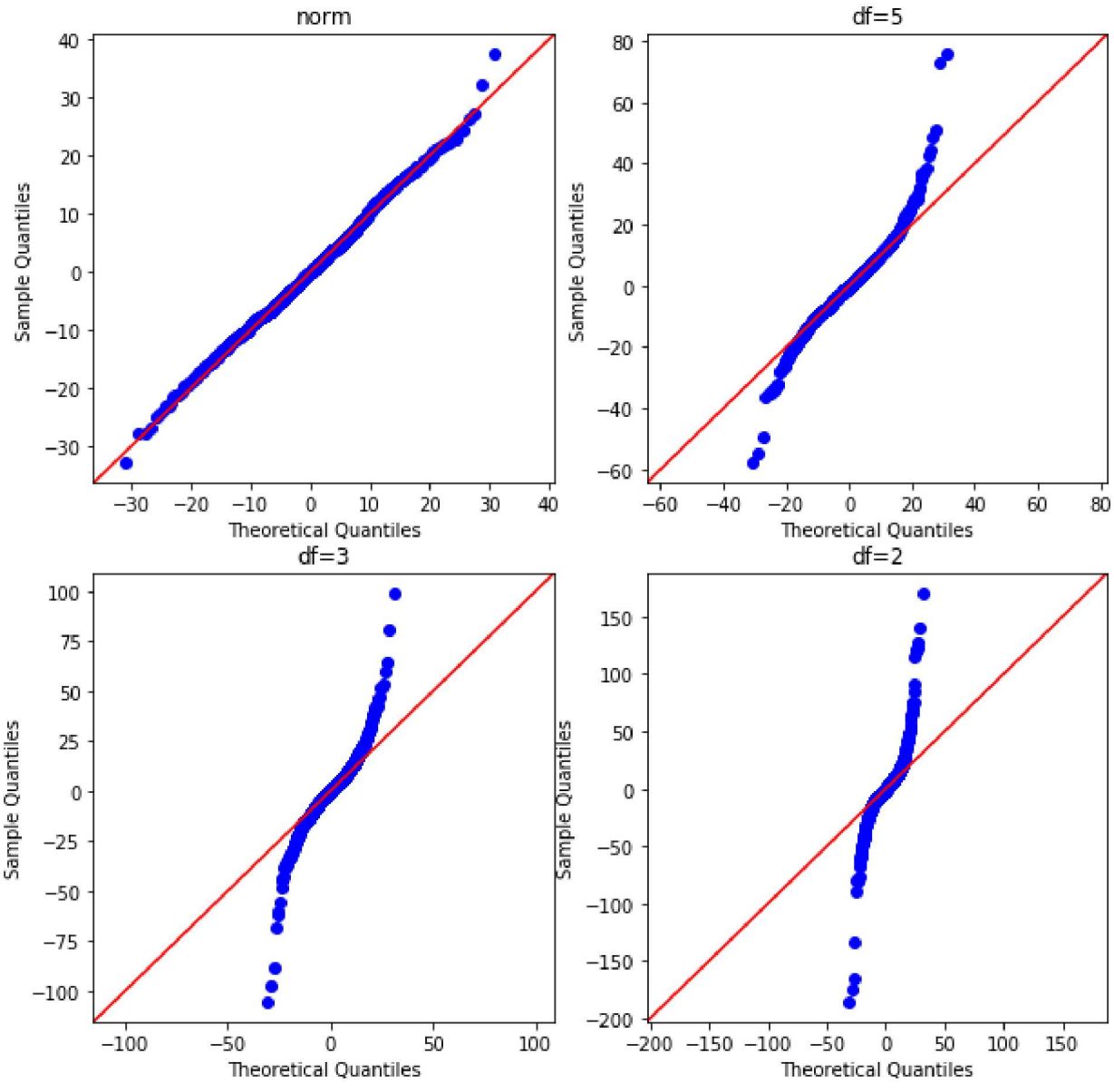
In [86]:
```
def first_difference(series):
    return series[1:]-series[:-1]
```

In [87]:
```
fd = []
fd.append(first_difference(rw1))
fd.append(first_difference(rw2))
fd.append(first_difference(rw3))
fd.append(first_difference(rw4))
plt.figure(figsize=(10,10))
titles = ["norm","df=5","df=3","df=2"]
for i in range(4):
    plt.subplot(2,2,i+1)
    plt.plot(fd[i])
    plt.title(titles[i])
plt.show()
```



iii) qq-plot of first difference data

In [88]:
```
plt.figure(figsize=(10,10))
titles = ["norm","df=5","df=3","df=2"]
for i in range(4):
    ax = plt.subplot(2,2,i+1)
    sm.qqplot(fd[i],dist=sp.stats.norm,fit=False,line='45',ax=ax,loc=0,scale=10)
    plt.title(titles[i])
plt.show()
```



From the result, we noticed that the curve of student-t distribution is non linear although the central part of the curve is close to linear. In other words, when we do QQ-plot, the distribution with fat tail will result in steeper slopes in the tails.

iv) moments of the first difference data

In [89]:
```
for i in range(4):
    print("**************************************************")
    print("describe (%s):"%titles[i],sp.stats.describe(fd[i]))
```

```
**************************************************
describe (norm): DescribeResult(nobs=999, minmax=(-32.75509643063734, 37.364339584199), mean=0.18952232130350
6, variance=95.17071249286707, skewness=0.10653934841103037, kurtosis=0.027577670567207502)
**************************************************
describe (df=5): DescribeResult(nobs=999, minmax=(-57.74340889432966, 75.60053975909497), mean=0.081774028864
43899, variance=148.9269361959991, skewness=0.41895012183822367, kurtosis=4.38833165484317)
**************************************************
describe (df=3): DescribeResult(nobs=999, minmax=(-105.18831902299587, 99.03824811110995), mean=-0.2473818733
1160594, variance=254.24836935961517, skewness=-0.2562364693003789, kurtosis=7.251297501794477)
**************************************************
describe (df=2): DescribeResult(nobs=999, minmax=(-185.4764936040655, 170.46938100224503), mean=0.28974482843
108823, variance=580.4668956792927, skewness=-0.22989832379686034, kurtosis=16.33423775883411)
```

The moments are totally different, especially the variance, skewness and excessed kurtosis

v) Jarque-Berra normality test

In [90]:
```
from scipy.stats import jarque_bera
for i in range(4):
    print("Jarque-Berra (%s)"%titles[i],jarque_bera(fd[i]))
```

```
Jarque-Berra (norm) (1.9215373289311588, 0.3825986829057483)
Jarque-Berra (df=5) (830.8154999883496, 0.0)
Jarque-Berra (df=3) (2199.6291678481916, 0.0)
Jarque-Berra (df=2) (11114.654890964213, 0.0)
```

Only normal distributed noise random walk servived, t-distributed random walks are all rejected to be a white noise

(c) Study the effect of changing σ. Look at σ = 0.1, 0.6, 1, 10.

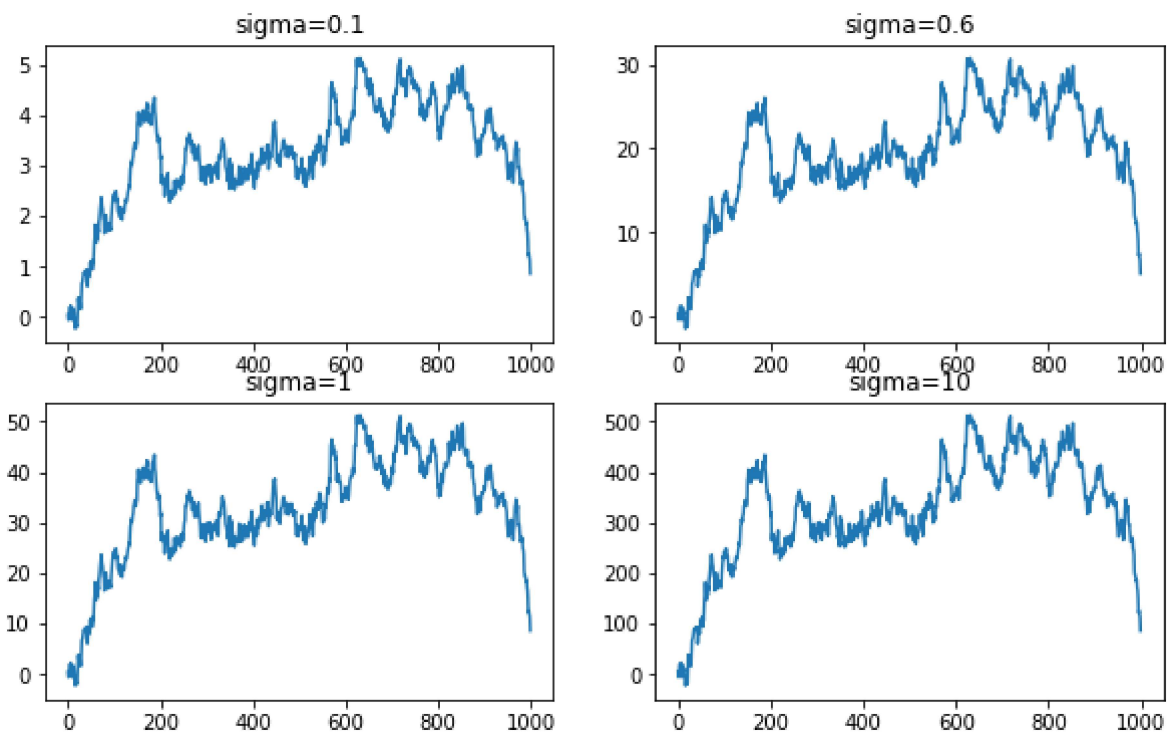
In this section, we only consider the student-t (df=5) random walks without drift (we can always eliminate drift by first difference operation)

In [91]:
```
at = sp.stats.t(df=5).rvs(1000)
rws = []
for i,sigma in enumerate([0.1,0.6,1,10]):
    rws.append(generate_random_walks(at*sigma,0))
```

i) Raw data

Generated by the same series

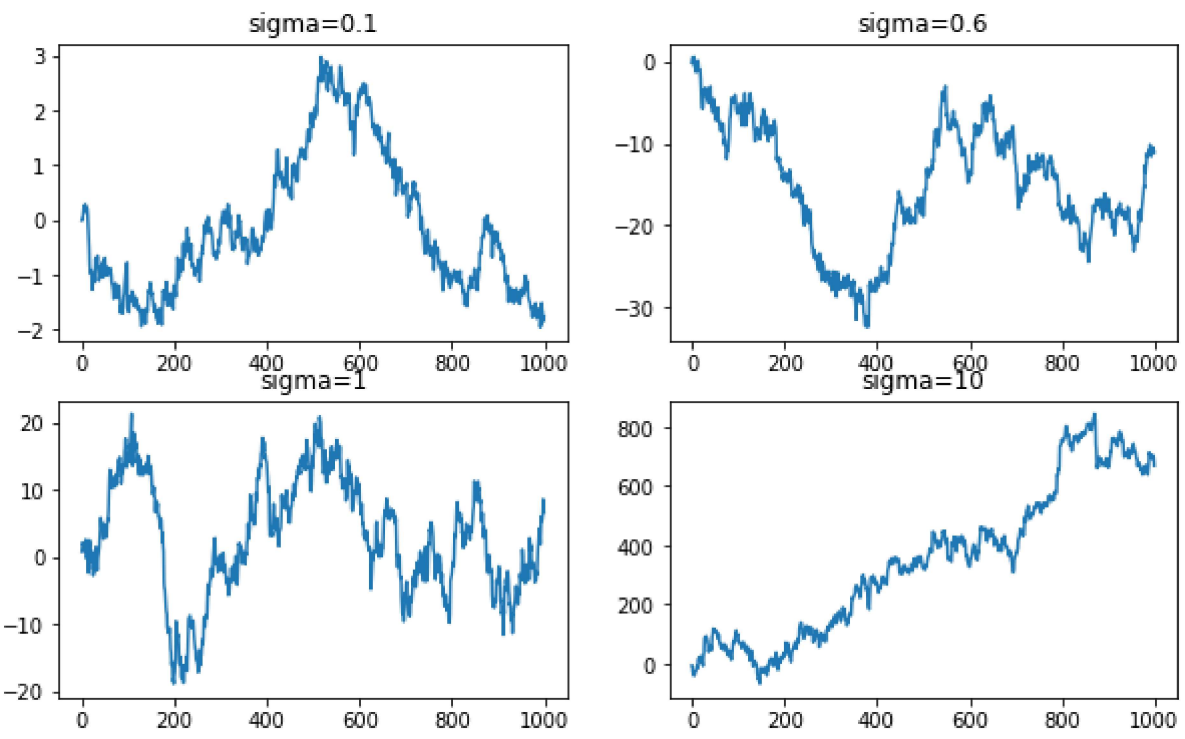
In [92]:
```
titles = ["sigma=0.1","sigma=0.6","sigma=1","sigma=10"]
plt.figure(figsize=(10,6))
for i in range(4):
    plt.subplot(2,2,i+1)
    plt.plot(rws[i])
    plt.title(titles[i])
plt.show()
```



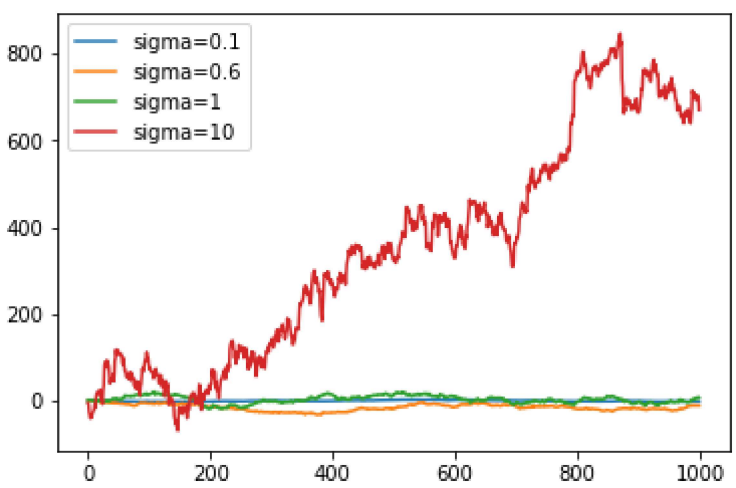
Apparently the only difference is the scale

Generated by different series

In [93]:
```
rws = []
for i,sigma in enumerate([0.1,0.6,1,10]):
    rws.append(generate_random_walks(sp.stats.t(df=5).rvs(1000)*sigma,0))

titles = ["sigma=0.1","sigma=0.6","sigma=1","sigma=10"]
plt.figure(figsize=(10,6))
for i in range(4):
    plt.subplot(2,2,i+1)
    plt.plot(rws[i])
    plt.title(titles[i])
plt.show()
```



In [94]:
```
for i in range(4):
    plt.plot(rws[i])
plt.legend(titles)
plt.show()
```

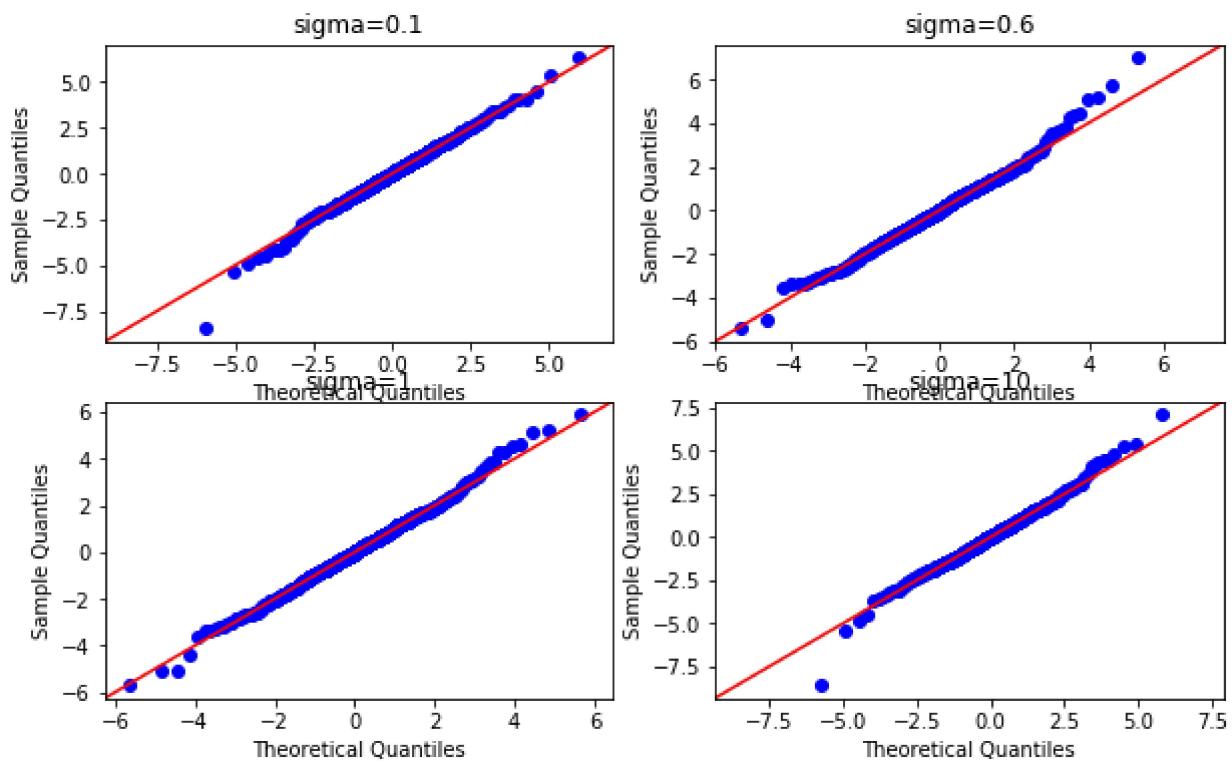


Still the difference arise from the scale, we analysis the behavior of its first difference by the following

In [95]:
```
fds = []
for i in range(4):
    fds.append(first_difference(rws[i]))
```

ii) fitted qq-plot of the first difference (set the theoretical distribution to be the student-t distribution)

In [96]:
```
titles = ["sigma=0.1","sigma=0.6","sigma=1","sigma=10"]
plt.figure(figsize=(10,6))
for i in range(4):
    ax=plt.subplot(2,2,i+1)
    sm.qqplot(fds[i],dist=sp.stats.t,ax=ax,fit=True,line="45")
    plt.title(titles[i])
plt.show()
```



iii) Moments of the first difference

In [97]:
```
for i in range(4):
    print("**************************************************")
    print("describe (%s):"%titles[i],sp.stats.describe(fds[i]))
```

```
**************************************************
describe (sigma=0.1): DescribeResult(nobs=999, minmax=(-0.8274164277122296, 0.6156127081807446), mean=-0.0017
605633593704097, variance=0.015997586680215267, skewness=-0.2226211660358448, kurtosis=3.4496793282547555)
**************************************************
describe (sigma=0.6): DescribeResult(nobs=999, minmax=(-3.3274979656867494, 4.264100800591994), mean=-0.01095
1442321624273, variance=0.5766342295043796, skewness=0.3026599863044285, kurtosis=2.6189291523208524)
**************************************************
describe (sigma=1): DescribeResult(nobs=999, minmax=(-6.031965179094026, 6.248583373919114), mean=0.005837620
59623893, variance=1.7910541369287412, skewness=0.10582485384255663, kurtosis=2.1453101898266995)
**************************************************
describe (sigma=10): DescribeResult(nobs=999, minmax=(-85.4310328687418, 71.61069659061218), mean=0.676327596
3655737, variance=166.3015305677176, skewness=0.041536024911747994, kurtosis=4.0274861581729375)
```

The standard deviation of the first difference series is propotional to the sigma

In [98]:
```
stds = []
for i in range(4):
    print("**************************************************")
    stds.append(sp.std(fds[i]))
    print("std (%s):"%titles[i],sp.std(fds[i]))
```

```
**************************************************
std (sigma=0.1): 0.126418246625773
**************************************************
std (sigma=0.6): 0.7589842014583928
**************************************************
std (sigma=1): 1.3376327186282595
**************************************************
std (sigma=10): 12.889339105212157
```

In [99]:
```
sigmas = [0.1,0.6,1,10]
plt.scatter(stds,sigmas)
plt.show()
```