Time Series Analysis with Python (first draft)

Peter von Tessin

August 30th 2009

1 Introduction

With numpy and scipy python offers quite a wide range of capabilities for serious econometric work including time series analysis. In this short paper I collect some of my recent work in order to reuse and enhance it in the future. Obviously this is just a glimmer of what python can provide in this area and I hope I will have the time to enlarge these small scripts into something more usefull in the future.

2 Creating ARMA time series data

An autoregressive time series process has the following form:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + ... + \alpha_n y_{t-n} + \epsilon_t \tag{1}$$

and we use the following python code to create such a data-series:

```
def ar(values, errors, alpha):
    for i in range(len(values)):
        value = 0
        n = len(alpha) if len(alpha)<=i else i
        for j in range(n):
            value = value + alpha[j] * values[i-j-1]
        values[i] = value +  errors[i]
```

A moving average time series process has the form:

$$y_t = \beta_0 + \beta_1 \epsilon_{t-1} + ... + \beta_n \epsilon_{t-n} + \epsilon_t \tag{2}$$

which can be created with the following python code:

```
def ma(values, errors, beta):
    for i in range(len(values)):
        value = 0
        n = len(beta) if len(beta)<=i else i
```

```
        for j in range(n):
            value = value + beta[j] * errors[i-j-1]
        values[i] = value + errors[i]
```

We can combine those two processes to the famous ARMA-process:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + ... + \alpha_n y_{t-n} + \beta_1 \epsilon_{t-1} + ... + \beta_m \epsilon_{t-m} + \epsilon_t \tag{3}$$

Such an ARMA time series can be created with the following code:

```
import numpy

n = 10
mu = 0
sig = 1
errors = numpy.random.normal(mu, sig, n)

n_ar = 3
alpha = numpy.random.uniform(0,1,n_ar)

n_ma = 2
beta = numpy.random.uniform(0,1,n_ma)

def arma(values, errors, alpha, beta):
    for i in range(len(values)):

        value_ar = alpha[0]
        n_ar = len(alpha)-1 if len(alpha)-1<=i else i
        for j in range(n_ar):
            value_ar = value_ar + alpha[j+1] * values[i-j-1]

        value_ma = 0
        n_ma = len(beta) if len(beta)<=i else i
        for j in range(n_ma):
            value_ma = value_ma + beta[j] * errors[i-j-1]

        values[i] = value_ar + value_ma + errors[i]

values = numpy.zeros(n)
arma(values, errors, alpha, beta)
```

3 Creating GARCH time series data

An autoregressive conditional heteroscedastic (ARCH) process has the following form:

$$y_t = \sigma_t \epsilon_t \tag{4}$$

$$\sigma_t = \alpha_0 + \alpha_1 y_{t-1}^2 + ... + \alpha_n y_{t-n}^2 \tag{5}$$

which can be created using the following python code:

```
def arch(values, errors, alpha):
    for i in range(len(values)):
        sig2 = alpha[0]
        n = len(alpha)-1 if len(alpha)-1<=i else i
        for j in range(n):
            sig2 = sig2 + alpha[j+1] * math.pow(values[i-j-1], 2)
        values[i] = math.sqrt(sig2) *  errors[i]
```

Such an ARCH process can be extended to a general autoregressive conditional heteroscedastic (GARCH) process by incorporating also laged values of σ:

$$y_t = \sigma_t \epsilon_t \tag{6}$$

$$\sigma_t = \alpha_0 + \alpha_1 y_{t-1}^2 + ... + \alpha_n y_{t-n}^2 + \beta_1 \sigma_{t-1} + ... + \beta_m \sigma t - m \tag{7}$$

Such a GARCH process can be created by using:

```
import numpy
import math

n = 10
mu = 0
sig = 1
errors = numpy.random.normal(mu, sig, n)

n_a = 2
alpha = numpy.random.uniform(0,1,n_a)

n_b = 2
beta = numpy.random.uniform(0,1,n_b)
ef garch(values, errors, alpha, beta):
    for i in range(len(values)):
        sig2 = alpha[0]
        n1 = len(alpha)-1 if len(alpha)-1<=i else i
        for j in range(n1):
            sig2 = sig2 + alpha[j+1] * math.pow(values[i-j-1], 2)

        value = 0
        n2 = len(beta) if len(beta)<=i else i
        for j in range(n2):
            value = value + beta[j] * sigma2[i-j-1]

        sigma2[i] = value + sig2
        values[i] = (math.sqrt(sigma2[i])) *  errors[i]

values = numpy.zeros(n)
```

```
sigma2 = numpy.zeros(n)

garch(values, errors, alpha, beta)
```

4 Estimating ARMA parameters

In order to estimate the parameters of an ARMA process, we use maximum likelihood. In order to do this numerically we create the numeric first and second derivatives of such a process and use them in a Newton-Raphson algorithm:

```
import math
import numpy
import numpy.linalg
import arma

def logLikArMa(arPars, maPars, sig, values):
    n = len(values)
    eps = numpy.zeros(n)

    for i in range(n):

        sumAR = arPars[0]
        nAR = len(arPars)-1 if len(arPars)-1<=i else i
        for j in range(nAR):
            sumAR = sumAR + arPars[j+1]*values[i-j-1]

        sumMA = 0
        nMA = len(maPars) if len(maPars)<=i else i
        for j in range(len(maPars)):
            sumMA = sumMA + maPars[j]*eps[i-j-1]

        eps[i] = values[i]- sumAR - sumMA


    lik = -0.5 * n * math.log(2*math.pi)
    lik = lik - 0.5 * n* math.log(sig*sig)
    for i in range(n):
        lik = lik - 0.5 * math.pow(eps[i]/(sig*sig), 2)

    return lik

def split(allParas, ar, ma, s):
    counter = 0
    for i in range(len(ar)):
        ar[i] = allParas[counter]
        counter = counter + 1
```

```
    for i in range(len(ma)):
        ma[i] = allParas[counter]
        counter = counter + 1
    s[0] = allParas[counter]
    return

def firstDeriv(paras):
    fd = numpy.zeros(len(paras))
    oldPara = 0
    e = 0.00001
    ar = numpy.zeros(n_ar)
    ma = numpy.zeros(n_ma)
    s = numpy.zeros(1)
    for i in range(len(paras)):
        oldPara = paras[i]
        paras[i] = oldPara + e
        split(paras, ar, ma, s)
        ll_1 = logLikArMa(ar, ma, s, values)
        paras[i] = oldPara
        split(paras, ar, ma, s)
        ll_2 = logLikArMa(ar, ma, s, values)
        fd[i] = (ll_1-ll_2)/e

    return fd

def secondDeriv(paras):
    n_p = len(paras)
    help = numpy.zeros(n_p*n_p)
    sd = help.reshape([n_p, n_p])
    oldParai = 0
    oldParaj = 0
    e = 0.00001
    ar = numpy.zeros(n_ar)
    ma = numpy.zeros(n_ma)
    s = numpy.zeros(1)
    for i in range(len(paras)):
        oldParai = paras[i]
        for j in range(len(paras)):
            if i==j:
                paras[i] = oldParai + e
                split(paras, ar, ma, s)
                ll_ip = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai - e
                split(paras, ar, ma, s)
                ll_im = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai
```

```
                split(paras, ar, ma, s)
                ll_ij = logLikArMa(ar, ma, s, values)
                sd[i][j] = (ll_ip-2*ll_ij+ll_im)/(e*e)
            else:
                oldParaj = paras[j]
                paras[i] = oldParai + e
                paras[j] = oldParaj + e
                split(paras, ar, ma, s)
                ll_ipjp = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai + e
                paras[j] = oldParaj - e
                split(paras, ar, ma, s)
                ll_ipjm = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai - e
                paras[j] = oldParaj + e
                split(paras, ar, ma, s)
                ll_imjp = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai - e
                paras[j] = oldParaj - e
                split(paras, ar, ma, s)
                ll_imjm = logLikArMa(ar, ma, s, values)
                paras[i] = oldParai
                paras[j] = oldParaj
                sd[i][j] = (ll_ipjp-ll_ipjm-ll_imjp+ll_imjm)/(4*e*e)
    return sd

n = 5000

n_ar = 2
n_ma = 2

parameters = numpy.random.uniform(0,1,n_ar+n_ma+1)

alpha = numpy.zeros(n_ar)
beta = numpy.zeros(n_ma)
sigma = numpy.zeros(1)

split(parameters, alpha, beta, sigma)

print(alpha)
print(beta)
print(sigma)

errors = numpy.random.normal(0, sigma, n)
values = numpy.zeros(n)
arma.arma(values, errors, alpha, beta)
```

```
alpha_2 = numpy.zeros(n_ar)
beta_2 = numpy.zeros(n_ma)
sigma_2 = numpy.zeros(1)

parameters_2 = numpy.random.uniform(0.05,0.15,n_ar+n_ma+1)

for i in range(n_ar+n_ma+1):
    parameters_2[i] = parameters[i]   +  parameters_2[i]

split(parameters_2, alpha_2, beta_2, sigma_2)

for i in range(5):
    split(parameters_2, alpha_2, beta_2, sigma_2)
    print(alpha_2)
    print(beta_2)
    print(sigma_2)
    print("******")
    fd = numpy.mat(firstDeriv(parameters_2))
    sd = numpy.mat(secondDeriv(parameters_2))
    abc = sd.I*fd.T
    for j in range(len(parameters_2)):
        parameters_2[j] = parameters_2[j]-abc[j]
```

Obviously this is far from finished. The way I created the start values is rather naive and the optimisation does not always converge to the proper parameter values. But it is a start, and I hope to be able to work on it some more in the future.