muarch Documentation

Release 0.0.4

Daniel Bok

Feb 24, 2021

CONTENTS:

This is a wrapper on top of Kevin Sheppard's ARCH package. The purpose of which are to:

1. Enable faster Monte Carlo simulation
2. Simulate innovations through copula marginals

In the package, there are 2 classes to aid you - `UArch` and `MUArch`. The `UArch` class can be defined using a similar API to `arch_model()` in the original `arch` package. The `MUArch` is a collection of these `UArch` models.

Thus, if you have a function that generates uniform marginals, like a copula, you can create a dependence structure among the different marginals when simulating the GARCH processes.

If you need a copula package, I have one here. :)

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Python version support

Only Python 3.6 and 3.7.

1.1.2 Installing MUArch

`MUArch` can be installed via pip from PyPI

```
pip install muarch
```

Alternatively, you can install it via `conda` with

```
conda install -c danielbok muarch
```

1.2 Examples

Here are some examples to get you started

1.2.1 A Simple Copula-GARCH Example

In this example, we will load a dataset which contains returns from 3 ETF and attempt to simulate future returns. Instead of fitting a multivariate GARCH model, what we will do instead is to fit a univariate GARCH model to each returns stream and construct a dependency model among these returns streams with a copula.

The copulas package can be installed separately at

```
conda install -c conda-forge copulae  # for anaconda

pip install copulae  # pip
```

Model Overview.

We will assume an AR(1)-GARCH(1, 1)-Normal model for each returns stream. For their dependency structure, we will assume a Student (T) copula.

[1]:
```
from muarch import MUArch, UArch
from muarch.datasets import load_etf
from copulae import TCopula


returns = load_etf()  # load returns data
returns.head()
```

[1]:
```
                 VOO       EEM        VT
Date
2010-10-01  0.043390  0.030154  0.037955
2010-11-01 -0.001108 -0.029055 -0.026458
2010-12-01  0.064337  0.063868  0.056120
2011-01-01  0.026914 -0.030360  0.033688
2011-02-01  0.034664 -0.000436  0.029297
```

[2]:
```
num_assets = returns.shape[1]

# sets up a MUArch model collection where each model defaults to
# mean: AR(1)
# vol: GARCH(1, 1)
# dist: normal
models = MUArch(num_assets, mean='AR', lags=1)
```

We could overwrite each model in the `MUArch` instance. For example, if we believe that a skew-t distribution better describes the innovation of `VOO` (an ETF tracking S&P 500), we could overwrite it as follows.

[3]:
```
# set first model to AR(1)-GARCH(1, 1) with skewt innovations
models[0] = UArch('AR', lags=1, dist='skewt')
```

In fact, we could set all of the models separately. All we have to do is to call `MUArch` like this

```
models = MUArch(5)  # 5 models

for i in range(5):
    models[i] = make_uarch_model(...)
```

To fit the model, we just need to call the `.fit()` method. This applies to both the `UArch` and `MUArch` models

[4]:
```
models.fit(returns)
```

We can see the summary of the models using the `.summary()` method.

[5]:
```
models.summary()
```

[5]:
```
<class 'muarch.summary.SummaryList'>
"""
VOO

                              AR - GARCH Model Results
==============================================================================================
Dep. Variable:                                   y   R-squared:                          0.
↪003
```

(continues on next page)

(continued from previous page)

```
Mean Model:                                AR   Adj. R-squared:                  -0.
↪007
Vol Model:                              GARCH   Log-Likelihood:                   209.
↪083
Distribution:      Standardized Skew Student's t   AIC:                          -404.
↪166
Method:                        Maximum Likelihood   BIC:                          -385.
↪860
                                                    No. Observations:                ⌊
↪101
Date:                           Mon, Mar 18 2019   Df Residuals:                    ⌊
↪ 94
Time:                                  14:12:56   Df Model:                        ⌊
↪  7
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
Const          0.0158  3.120e-03      5.048  4.466e-07 [9.635e-03,2.187e-02]
y[1]          -0.3131  7.794e-02     -4.017  5.884e-05     [ -0.466, -0.160]
                            Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega      2.6889e-04  1.293e-04      2.079  3.759e-02 [1.543e-05,5.224e-04]
alpha[1]       0.2066  7.759e-02      2.663  7.735e-03   [5.458e-02,  0.359]
beta[1]        0.5258      0.131      4.025  5.702e-05     [  0.270,  0.782]
                              Distribution
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
nu            37.8296    132.062      0.286      0.775 [-2.210e+02,2.967e+02]
lambda        -0.4572      0.110     -4.172  3.021e-05      [ -0.672, -0.242]
============================================================================

Covariance estimator: robust

****************************************************************************************************

EEM

                          AR - GARCH Model Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.006
Mean Model:                        AR   Adj. R-squared:                 -0.004
Vol Model:                      GARCH   Log-Likelihood:                157.484
Distribution:                  Normal   AIC:                          -304.968
Method:            Maximum Likelihood   BIC:                          -291.893
                                        No. Observations:                  101
Date:                Mon, Mar 18 2019   Df Residuals:                       96
Time:                        14:12:56   Df Model:                           5
                            Mean Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
Const      5.0854e-03  5.935e-03      0.857      0.392 [-6.546e-03,1.672e-02]
y[1]          -0.0628      0.109     -0.579      0.563      [ -0.275,  0.150]
```

(continues on next page)

(continued from previous page)

```
                             Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega      2.1956e-03  9.297e-04      2.362  1.820e-02 [3.734e-04,4.018e-03]
alpha[1]       0.1735      0.139      1.246      0.213  [-9.935e-02,  0.446]
beta[1]        0.0000      0.330      0.000      1.000     [ -0.647,  0.647]
============================================================================

Covariance estimator: robust

********************************************************************************************************

VT

                          AR - GARCH Model Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.015
Mean Model:                        AR   Adj. R-squared:                  0.005
Vol Model:                      GARCH   Log-Likelihood:                194.275
Distribution:                  Normal   AIC:                           -378.551
Method:            Maximum Likelihood   BIC:                           -365.475
                                        No. Observations:                  101
Date:                Mon, Mar 18 2019   Df Residuals:                       96
Time:                        14:12:56   Df Model:                           5
                            Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
Const          0.0114  4.305e-03      2.650  8.039e-03 [2.972e-03,1.985e-02]
y[1]          -0.1711  8.939e-02     -1.914  5.559e-02   [ -0.346,4.086e-03]
                             Volatility Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
omega      3.2937e-04  4.396e-04      0.749      0.454 [-5.322e-04,1.191e-03]
alpha[1]       0.2420      0.189      1.280      0.201      [ -0.129,  0.613]
beta[1]        0.5257      0.471      1.117      0.264      [ -0.397,  1.448]
============================================================================

Covariance estimator: robust
"""
```

Now that we have a model for each of the returns streams, the question is how can we create a dependency model amongst them? We can do so by fitting the residuals for each `UArch` model into a copula.

This fitting of residuals means for the copula to find a relationship among the different models. Subsequently, we can use the copula to randomly generate the residuals which can be used by the `UArch` model to simulate returns stream for the assets.

[6]:
```
residuals = models.residuals() # defaults to return the standardized residuals

cop = TCopula(dim=num_assets)
cop.fit(residuals)

print(cop.summary())
```

```
Student T Copula with 3 dimensions

Degrees of Freedom: 9.837148817580086

Correlation Matrix (P):
    [[1.         0.59016153 0.90300841]
 [0.59016153 1.         0.80087304]
 [0.90300841 0.80087304 1.        ]]

Log. Lik        : -149.63318356908047
Var. Est.       : Not Implemented Yet
Method          : Maximum pseudo-likelihood
Data Pts.       : 101

Optim Options
        bounds         : [(0.0, inf), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0)]
        options        : {'maxiter': 20000, 'ftol': 1e-06, 'iprint': 1, 'disp': False,
↪ 'eps': 1.5e-08}
        method         : SLSQP

Results
        x              : [9.83714882 0.59016153 0.90300841 0.80087304]
        fun            : -149.63318356908047
        jac            : [-1.70530257e-05 -1.11034145e-03  2.97859515e-03  1.
↪59540529e-03]
        nit            : 13
        nfev           : 84
        njev           : 13
        status         : 0
        message        : Optimization terminated successfully.
        success        : True
```

We could of course overwrite the correlation matrix of the TCopula with the historical correlation. I'll show an example below. For more details, check out the Copulae package documentation.

[7]:
```
cop[:] = returns.corr()

print(cop.summary())
```

```
Student T Copula with 3 dimensions

Degrees of Freedom: 9.837148817580086

Correlation Matrix (P):
    [[1.         0.72108633 0.94803336]
 [0.72108633 1.         0.85096406]
 [0.94803336 0.85096406 1.        ]]

Log. Lik        : -149.63318356908047
Var. Est.       : Not Implemented Yet
Method          : Maximum pseudo-likelihood
Data Pts.       : 101

Optim Options
        bounds         : [(0.0, inf), (-1.0, 1.0), (-1.0, 1.0), (-1.0, 1.0)]
        options        : {'maxiter': 20000, 'ftol': 1e-06, 'iprint': 1, 'disp': False,
↪ 'eps': 1.5e-08}
        method         : SLSQP
```

(continues on next page)

(continued from previous page)

```
Results
        x               : [9.83714882 0.59016153 0.90300841 0.80087304]
        fun             : -149.63318356908047
        jac             : [-1.70530257e-05 -1.11034145e-03  2.97859515e-03  1.
↪59540529e-03]
        nit             : 13
        nfev            : 84
        njev            : 13
        status          : 0
        message         : Optimization terminated successfully.
        success         : True
```

Notice the difference in the correlation matrix above. Note that correlation matrix is a parameter only for elliptical copulas. For Archimedean and others, change the parameters accordingly.

Let us now simulate 100 trials of returns, 10 steps into the future with the copula. All you have to do is to pass in the `.random` method to generate the innovations.

[8]:
```
horizon = 10
trials = 100

models.simulate_mc(horizon, trials, custom_dist=cop.random)
```

[8]:
```
array([[[ 2.36819180e-02,  5.91736570e-02,  4.76506242e-02],
        [ 1.44822161e-02,  6.27440217e-02,  1.37890054e-02],
        [-5.48242994e-02, -7.61327302e-02, -4.33654769e-02],
        ...,
        [ 2.16221301e-02, -3.66574706e-03,  5.80400414e-03],
        [ 1.24950196e-02, -6.65987110e-02, -1.91601512e-03],
        [ 4.35219595e-02,  1.42134542e-01,  1.18264754e-01]],

       [[ 1.14673295e-02, -7.24770972e-03,  1.12111664e-03],
        [ 1.15180971e-02,  3.51954043e-02,  1.45830445e-02],
        [ 5.44165821e-02,  2.19578347e-02,  3.13810703e-02],
        ...,
        [ 2.89393067e-02,  5.70089775e-02,  3.36656830e-02],
        [ 4.99865506e-04,  4.40031966e-02,  8.95130002e-05],
        [ 6.15984202e-02,  8.85007780e-02,  1.28838163e-01]],

       [[ 2.39743360e-02,  3.34112771e-02,  3.14031296e-02],
        [ 2.92497275e-02, -2.05959398e-03,  1.59909456e-02],
        [-8.70354146e-02, -8.66656129e-02, -6.46162277e-02],
        ...,
        [-6.16685073e-03, -2.02064819e-02, -7.07666634e-03],
        [ 2.26282761e-02, -3.40303169e-03,  1.63187322e-02],
        [ 1.32278891e-02,  2.03830411e-02,  1.30598931e-02]],

       ...,

       [[ 1.38241116e-02, -1.77706225e-02,  3.15010579e-03],
        [ 4.07693772e-02,  3.72229328e-02,  3.48073220e-02],
        [-5.15138745e-02, -3.47109312e-02, -3.92909818e-02],
        ...,
        [ 1.99715299e-02,  8.51965041e-02,  4.27344365e-02],
        [-1.78759597e-02,  6.98907216e-03, -1.19619336e-02],
        [ 2.82702866e-02,  2.89281519e-02,  3.18538287e-02]],
```

(continues on next page)

(continued from previous page)

```
       [[ 1.41677467e-02, -1.76165401e-02,  1.20177353e-02],
        [-8.85086028e-02, -1.42317567e-01, -8.50680186e-02],
        [ 1.49899395e-02, -1.54490974e-02, -4.67326285e-03],
        ...,
        [ 2.33224844e-03, -5.60107187e-02, -2.00618802e-02],
        [ 1.67808821e-02,  3.39528074e-02,  2.07668467e-02],
        [ 7.08545343e-03,  3.61847390e-02,  1.77982130e-02]],

       [[ 1.70144077e-02, -2.83423287e-02,  9.56741286e-03],
        [ 2.73957468e-02, -1.38556690e-01, -3.99365353e-02],
        [ 4.69939240e-02,  2.94684632e-02,  4.31496651e-02],
        ...,
        [ 4.15675721e-02,  5.30936408e-02,  3.20754793e-02],
        [ 2.85645038e-02,  9.01270560e-02,  4.26091035e-02],
        [ 3.27343634e-02,  1.04311493e-02,  2.55611539e-02]]])
```

1.3 MUArch Core API

A dictionary of core `muarch` classes and functions. The core classes are `UArch` and `MUArch`.

1.3.1 UArch

`UArch` is short for Univariate ARCH models and `MUArch` stands for multiple (or many) Univariate ARCH models. In essence `MUArch`, is a list of many `UArch` models. This helps when you need to simulate many univariate ARCH models together. Also, it is helpful when you need to specify the marginals as in a Copula-GARCH model.

class muarch.uarch.**UArch**(*mean='Constant'*, *lags=0*, *vol='GARCH'*, *p=1*, *o=0*, *q=1*, *power=2.0*, *dist='Normal'*, *hold_back=None*, *scale=1*)

Univariate ARCH model that wraps on top of Mean, Volatility and Distribution classes defined in the *arch* package. Mainly, this class combines the original model and fitted model in the *arch* package for convenience. It has also some additional methods such as `simulate_mc` for Monte Carlo simulations.

__init__(*mean='Constant'*, *lags=0*, *vol='GARCH'*, *p=1*, *o=0*, *q=1*, *power=2.0*, *dist='Normal'*, *hold_back=None*, *scale=1*)

Creates the wrapping arch model

Parameters

- **mean** (*{ 'zero', 'constant', 'harx', 'har', 'ar', 'arx', 'ls' }, optional*) – Name of the mean model. Currently supported options are:
  - **Constant** (default) - Constant mean model
  - **Zero** - Zero mean model
  - **AR** - Autoregression model
  - **ARX** - Autoregression model with exogenous regressors. Falls back to **AR** if no exogenous regressors
  - **HAR** - Heterogeneous Autoregression model
  - **HARX** - Heterogeneous Autoregressions with exogenous regressors
  - **LS** - Least squares model

For more information on the different models, check out the documentation at https://arch.readthedocs.io/en/latest/univariate/mean.html

- **lags** (*int or list (int), optional*) – Either a scalar integer value indicating lag length or a list of integers specifying lag locations.
- **vol** (*{ 'GARCH', 'ARCH', 'CONSTANT' 'EGARCH', 'FIGARCH' and 'HARCH', 'CONSTANT' }, optional*) – Name of the volatility model. Currently supported options are:
  - **GARCH** (default) - Standard GARCH process which can be used to specify the following models:
    - ARCH(p)
    - GARCH(p,q)
    - GJR-GARCH(p,o,q)
    - AVARCH(p)
    - AVGARCH(p,q)
    - TARCH(p,o,q)
    - Models with arbitrary, pre-specified powers
  - **ARCH** - ARCH process
  - **EGARCH** - EGARCH process
  - **FIGARCH** - Fractionally Integrated (FI) GARCH process
  - **HARCH** - Heterogeneous ARCH process
  - **Constant** (default) - Constant volatility process
- **p** (*int, optional*) – Lag order of the symmetric innovation
- **o** (*int, optional*) – Lag order of the asymmetric innovation
- **q** (*int, optional*) – Lag order of lagged volatility or equivalent
- **power** (*float, optional*) – Power to use with the innovations. Default is 2.0, which produces ARCH and related models. Using 1.0 produces AVARCH and related models. Other powers can be specified, although these should be strictly positive, and usually larger than 0.25.
- **dist** (*{ 'normal', 'gaussian', 'studentst', 't', 'skewstudent', 'skewt', 'ged', 'generalized error' }, optional*) – Name of the distribution for the innovations. Currently supported options are:
  - **normal**, **gaussian** (default) - Standard Normal distribution
  - **t**, **studentst** - Standardized Student's distribution
  - **skewstudent**, **skewt** - Standardized Skewed Student's distribution.
  - **ged**, **"generalized error" - Generalized Error Distribution
- **hold_back** (*int, optional*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.

- **scale** (*float*) – Factor to scale data up or down by. This is useful when your data is too small leading to numerical errors when fitting. It will be used to scale simulation data

fit(*y*, *x=None*, *update_freq=1*, *disp='off'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)
Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **y** (*{ndarray, Series}*) – The dependent variable
- **x** (*{ndarray, DataFrame}, optional*) – Exogenous regressors. Ignored if model does not permit exogenous regressors.
- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output
- **disp** (*'final' or 'off' (default)*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recursions.

Returns Fitted UArch instance

Return type *UArch*

forecast(*params=None*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*ndarray, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes

can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.

- **align** (*{'origin', 'target'}, optional*) – When set to 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h.

  When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplifies computing forecast errors since the realization and h-step forecast are aligned.

- **method** (*{'analytic', 'simulation', 'bootstrap'}, optional*) – Method to use when producing the forecast. The default is 'analytic'. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.

- **simulations** (*int, optional*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.

- **rng** (*{callable, ndarray}, optional*) – If using a custom random number generator to for simulation-based forecasts, function must produce random samples using the syntax *rng(size)* where size is a 2-element tuple (simulations, horizon).

  Else, if a numpy array is passed in, array must have shape (simulation x horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type ARCHModelForecast

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (*model.x is not None*), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

hedgehog_plot(*params=None*, *horizon=10*, *step=10*, *start=None*, *type_='volatility'*, *method='analytic'*, *simulations=1000*)

Plot forecasts from estimated model

Parameters

- **params** (*{Series, ndarray}, optional*) – Alternative parameters to use. If not provided, the parameters computed by fitting the model are used. Must be 1-d and identical in shape to the parameters computed by fitting the model

- **horizon** (*int, optional*) – Number of steps to forecast

- **step** (*int, optional*) – Non-negative number of forecasts to skip between spines

- **start** (*int, datetime or str, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'. If not provided, the start is set to the earliest forecastable date
- **type** (*{'volatility', 'mean'}*) – Quantity to plot, the forecast volatility or the forecast mean
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap

Returns Handle to the figure

Return type figure

property params
Model Parameters

residual_plot(*annualize=None*, *scale=None*)
Plot standardized residuals and conditional volatility

Parameters

- **annualize** (*str, optional*) – String containing frequency of data that indicates plot should contain annualized volatility. Supported values are 'D' (daily), 'W' (weekly) and 'M' (monthly), which scale variance by 252, 52, and 12 respectively
- **scale** (*float, optional*) – Value to use when scaling returns to annualize. If scale is provides, annualize is ignored and the value in scale is used.

Returns Handle to the figure

Return type figure

residuals(*standardize=True*) → numpy.ndarray
Model residuals

Parameters **standardize** (*bool, optional*) – Whether to standardize residuals. Residuals are standardized by dividing it with the conditional volatility

Returns Residuals

Return type ndarray

simulate(*nobs: int*, *burn=500*, *initial_value: Optional[Union[float, numpy.ndarray]] = None*, *x: Optional[Union[numpy.ndarray, pandas.core.frame.DataFrame]] = None*, *initial_value_vol: Optional[Union[float, numpy.ndarray]] = None*, *data_only=False*, *params: Optional[numpy.ndarray] = None*, *custom_dist: Optional[Union[Callable[[Union[int, Collection[int]]], numpy.ndarray], numpy.ndarray]] = None*) → Union[pandas.core.frame.DataFrame, numpy.ndarray]
Simulates data from a ARMA-GARCH model

Parameters

- **nobs** (*int*) – Length of series to simulate

- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used
- **x** (*{ndarray, DataFrame}, optional*) – nobs + burn by k array of exogenous variables to include in the simulation. This should be a 2D matrix
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.
- **data_only** (*bool, default True*) – If True, this returns only the simulated data, omits the volatility and error. In this case, it will return as a numpy array. Otherwise, it returns a data frame with the data, volatility and error
- **params** (*ndarray, optional*) – If not None, model will use the parameters supplied to generate simulations. Otherwise, it will use the fitted parameters.
- **custom_dist** (*{ndarray, Callable}, optional*) – Optional density from which to simulate the innovations (Distribution) in the GARCH models. This is useful when working with the copula-GARCH model where each univariate model innovations has dependence on others. It is assumed that the values supplied are standardized [0, 1] innovations instead of the unstandardized residuals.

  The shape of the array must be at least as long as the simulation size required after accounting for burn and type of innovation process. If unsure, use `simulation_size_required` to check.

  If a random number generator function is passed in, ensure that it only takes only argument and returns a numpy array. The argument can be an integer or a tuple of integers. In this case, the size will be automatically derived to save the user the trouble.

Returns DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation. If data_only, it returns the 'data' column as a numpy array

Return type DataFrame or ndarray

simulate_mc(*nobs: int*, *reps: int*, *burn=500*, *initial_value: Optional[Union[float, numpy.ndarray]] = None*, *x: Optional[Union[numpy.ndarray, pandas.core.frame.DataFrame]] = None*, *initial_value_vol: Optional[Union[float, numpy.ndarray]] = None*, *params: Optional[numpy.ndarray] = None*, *custom_dist: Optional[Union[Callable[[Union[int, Collection[int]]], numpy.ndarray], numpy.ndarray]] = None*) → Union[pandas.core.frame.DataFrame, numpy.ndarray]

Simulates data from a ARMA-GARCH model with multiple repetitions.

This is used for Monte Carlo simulations.

Parameters

- **nobs** (*int*) – Length of series to simulate
- **reps** (*int*) – Number of repetitions in Monte Carlo simulation
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used

- **x** (*{ndarray, DataFrame}, optional*) – nobs + burn by k array of exogenous variables to include in the simulation. This should be a 2D matrix
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.
- **params** (*{Series, ndarray}, optional*) – If not None, model will use the parameters supplied to generate simulations. Otherwise, it will use the fitted parameters.
- **custom_dist** (*{ndarray, Callable}, optional*) – Optional density from which to simulate the innovations (Distribution) in the GARCH models. This is useful when working with the copula-GARCH model where each univariate model innovations has dependence on others. It is assumed that the values supplied are standardized [0, 1] innovations instead of the unstandardized residuals.

  The shape of the array must be at least as long as the simulation size required after accounting for burn and type of innovation process. If unsure, use `simulation_size_required` to check.

  If a random number generator function is passed in, he size will be automatically derived to save the user the trouble. However, the function must:

  – take as it first argument an integer or a tuple of integer

  – have other parameters that are optional

  – return a numpy array

Returns **simulated_data** – Array containing the simulated values

Return type ndarray

See also:

UArch.simulation_horizon_required Calculates the simulation size required

simulation_horizon(*nobs: int*, *burn: int*)

Calculates the number of random generations needed for simulation

Parameters

- **nobs** (*int*) – number of observations
- **burn** (*int*) – number of observations burnt in simulation

Returns number of random generations required

Return type int

summary(*short=False*, *dp=4*) → Union[pandas.core.series.Series, muarch.summary.Summary]

Summary of fitted model

Parameters

- **short** (*bool, optional*) – Whether to show short summary or full summary.
- **dp** (*int, optional*) – Number of decimal places to show in short summary

Returns Model Summary

Return type Summary

1.3.2 MUArch

`UArch` is short for Univariate ARCH models and `MUArch` stands for multiple (or many) Univariate ARCH models. In essence `MUArch`, is a list of many `UArch` models. This helps when you need to simulate many univariate ARCH models together. Also, it is helpful when you need to specify the marginals as in a Copula-GARCH model.

class muarch.muarch.**MUArch**(*n: Union[int, Collection[muarch.uarch.UArch]]*, *mean='Constant'*, *lags=0*, *vol='GARCH'*, *p=1*, *o=0*, *q=1*, *power=2.0*, *dist='Normal'*, *hold_back=None*, *scale=1*)

Multi-univariate ARCH model. Unlike a multivariate ARCH model, this fits each univariate time series individually. Any simulations returns simulations of each univariate series column bound together.

__init__(*n: Union[int, Collection[muarch.uarch.UArch]]*, *mean='Constant'*, *lags=0*, *vol='GARCH'*, *p=1*, *o=0*, *q=1*, *power=2.0*, *dist='Normal'*, *hold_back=None*, *scale=1*)

Initializes the MUArch model.

The MUArch model holds multiple univariate models which are determined during fitting. If the models are not specified, the global default options will be used. Models can be individually specified after initializing the MUArch instance.

Parameters

- **n** (*int or list of UArch models*) – Number of univariate models to fit. Alternatively, a list of UArch (univariate) models can be specified.
- **mean** (*{ 'zero', 'constant', 'harx', 'har', 'ar', 'arx', 'ls' }, optional*) – Name of the global default mean model. Currently supported options are:
  - **Constant** (default) - Constant mean model
  - **Zero** - Zero mean model
  - **AR** - Autoregression model
  - **ARX** - Autoregression model with exogenous regressors. Falls back to **AR** if no exogenous regressors
  - **HAR** - Heterogeneous Autoregression model
  - **HARX** - Heterogeneous Autoregressions with exogenous regressors
  - **LS** - Least squares model

  For more information on the different models, check out the documentation at https://arch.readthedocs.io/en/latest/univariate/mean.html
- **lags** (*int or list (int), optional*) – Global default lag. Either a scalar integer value indicating lag length or a list of integers specifying lag locations.
- **vol** (*{ 'GARCH', 'ARCH', 'EGARCH', 'FIGARCH' and 'HARCH', 'CONSTANT' }, optional*) – Name of the global default volatility model. Currently supported options are:
  - **GARCH** (default) - Standard GARCH process which can be used to specify the following models:
    - ARCH(p)
    - GARCH(p,q)
    - GJR-GARCH(p,o,q)
    - AVARCH(p)

    - AVGARCH(p,q)
    - TARCH(p,o,q)
    - Models with arbitrary, pre-specified powers
  - **ARCH** - ARCH process
  - **EGARCH** - EGARCH process
  - **FIGARCH** - Fractionally Integrated (FI) GARCH process
  - **HARCH** - Heterogeneous ARCH process
  - **Constant** (default) - Constant volatility process
- **p** (*int, optional*) – Global default lag order of the symmetric innovation
- **o** (*int, optional*) – Global default lag order of the asymmetric innovation
- **q** (*int, optional*) – Global default lag order of lagged volatility or equivalent
- **power** (*float, optional*) – Global default power to use with the innovations, Default is 2.0, which produces ARCH and related models. Using 1.0 produces AVARCH and related models. Other powers can be specified, although these should be strictly positive, and usually larger than 0.25.
- **dist** (*{ 'normal', 'gaussian', 'studentst', 't', 'skewstudent', 'skewt', 'ged', 'generalized error' }, optional*) – Name of the global default distribution for the innovations. Currently supported options are:
  - **normal**, **gaussian** (default) - Standard Normal distribution
  - **t**, **studentst** - Standardized Student's distribution
  - **skewstudent**, **skewt** - Standardized Skewed Student's distribution.
  - **ged**, **generalized error" - Generalized Error Distribution
- **hold_back** (*int*) – Global default. Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.
- **scale** (*float*) – Global default factor to scale data up or down by. This is useful when your data is too small leading to numerical errors when fitting. It will be used to scale simulation data

fit(*y: Union[pandas.core.frame.DataFrame, numpy.ndarray], x: Optional[Union[Collection[Optional[numpy.ndarray]], numpy.ndarray]] = None, update_freq=1, disp='off', cov_type='robust', show_warning=True, tol: Optional[float] = None, options=None*)
Fits the MUArch model.

If finer control over the MUArch models is required, set the UArch models separately. Otherwise, method will set the default parameters.

Parameters

- **y** (*{ndarray, Series}*) – The dependent variable. If a vector is passed in, it is assumed that the same vector (endog) is used for all models. Otherwise, the last value of the shape must match the number of models
- **x** (*list of {ndarray, None}, optional*) – Exogenous regressors. Ignored if model does not permit exogenous regressors. If passed in, the first shape must match the number of models.

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output
- **disp** (*'final' or 'off' (default)*) – Either 'final' to print optimization result or 'off' to display nothing
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **tol** (*float, optional*) – Tolerance for termination
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'

Returns Fitted self instance

Return type *MUArch*

residuals(*standardize=True*) → numpy.ndarray

Model residuals

The residuals will be burnt by the maximum lag of the underlying models. For example, given 3 models - AR(1), AR(10), Constant with 400 data points each, the residuals will be 399, 390 and 400 long. The function will cut off the first 10 data points in this instance.

Parameters **standardize** (*bool, optional*) – Whether to standardize residuals. Residuals are standardized by dividing it with the conditional volatility

Returns Residuals

Return type ndarray

simulate(*nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*, *data_only=True*, *custom_dist: Optional[Union[Callable[[Union[int, Collection[int]]], numpy.ndarray], numpy.ndarray]] = None*)

Simulates data from the multiple ARMA-GARCH models

Parameters

- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used. If array, the last column must be of the same size as the number of models
- **x** (*{ndarray, list of ndarray}, optional*) – If supplied as a list, this list should have the same number of elements as the number of models in the MUArch model. Each array inside is the specified exogenous variable for that particular model and this must be a nobs + burn by k matrix of exogenous variables to include in the simulation. Otherwise, leave the value as *None* to indicate no exogenous variables are used for simulation in the model.

  If an array is supplied directly, it means every model has an exogenous variable associated with it. In this case, it should be a 3 dimensional tensor where the first dimension represents the number of models.

- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process. If array, the last column must be of the same size as the number of models
- **data_only** (*bool, default True*) – If True, this returns only the simulated data, omits the volatility and error. In this case, it will return as a numpy array. Otherwise, it returns a data frame with the data, volatility and error
- **custom_dist** (*{ndarray, Callable}, optional*) – Optional density from which to simulate the innovations (Distribution) in the GARCH models. This is useful when working with the copula-GARCH model where each univariate model innovations has dependence on others. It is assumed that the values supplied are standardized [0, 1] innovations instead of the unstandardized residuals.

  The shape of the array must be at least as long as the simulation size required after accounting for burn and type of innovation process. If unsure, use `simulation_size_required` to check. It must also account for the number of dimensions of the MUArch model. For example, if MUArch model is simulating a horizon of 120 time steps, 10000 trials and has 5 UArch models, the shape of the numpy array should be (120, 10000, 5).

  If a random number generator function is passed in, ensure that it only takes only argument and returns a numpy array. The argument can be an integer or a tuple of integers. In this case, the size will be automatically derived to save the user the trouble.

Returns **simulated_data** – List of DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type {List[DataFrame], ndarray}

See also:

UArch.simulation_horizon_required Calculates the simulation size required

simulate_mc(*nobs*, *reps*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*, *custom_dist: Optional[Union[Callable[[Union[int, Collection[int]]], numpy.ndarray], numpy.ndarray]] = None*, *n_jobs: Optional[int] = None*)

Simulates data from the multiple ARCH-GARCH models.

This function is specially crafted for Monte-Carlo simulations.

Parameters

- **nobs** (*int*) – Length of series to simulate
- **reps** (*int*) – Number of repetitions
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used. If array, the last column must be of the same size as the number of models
- **x** (*{ndarray, list of ndarray}, optional*) – If supplied as a list, this list should have the same number of elements as the number of models in the MUArch model. Each array inside is the specified exogenous variable for that particular model and this must be a nobs + burn by k matrix of exogenous variables to include in the

simulation. Otherwise, leave the value as *None* to indicate no exogenous variables are used for simulation in the model.

If an array is supplied directly, it means every model has an exogenous variable associated with it. In this case, it should be a 3 dimensional tensor where the first dimension represents the number of models.

- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process. If array, the last column must be of the same size as the number of models
- **custom_dist** (*{ndarray, Callable}, optional*) – Optional density from which to simulate the innovations (Distribution) in the GARCH models. This is useful when working with the copula-GARCH model where each univariate model innovations has dependence on others. It is assumed that the values supplied are standardized [0, 1] innovations instead of the unstandardized residuals.

  The shape of the array must be at least as long as the simulation size required after accounting for burn and type of innovation process. If unsure, use `simulation_size_required` to check.

  If a random number generator function is passed in, he size will be automatically derived to save the user the trouble. However, the function must:

  – take as it first argument an integer or a tuple of integer

  – have other parameters that are optional

  – return a numpy array

- **n_jobs** (*int or None, optional*) – The number of jobs to run in parallel for simulation. This is particularly useful when simulating large number of repetitions with more than 1 dimension. None defaults to using 1 processor. Any numbers less or equal to 0 means to use all processors. Even if a large number is used, it will be capped at the maximum number of processors available.

Returns **simulated_data** – Array containing simulated data from the Monte Carlo Simulation

Return type numpy array

summary(*short=False*, *dp=4*)
Summary of fitted models

Parameters

- **short** – bool, default False Whether to show short summary or full summary.
- **dp** – int, default 4 Number of decimal places to show in short summary

Returns SummaryList summary of fitted models

1.3.3 Utility Functions

The utility functions help in adjusting the simulated data cube. There are some assumptions about the cube. Namely, assuming that we are running a Monte-Carlo simulation of asset returns, the axis will be 3 dimensional where each axis represents the time, trials and asset class respectively.

Calibrate Data

muarch.calibrate.**calibrate_data**(*data: numpy.ndarray*, *mean: Optional[Collection[float]] = None*, *sd: Optional[Collection[float]] = None*, *time_unit: Union[int, str] = 'month'*, *inplace=False*, *tol=1e-06*) → numpy.ndarray

Calibrates the data given the target mean and standard deviation.

Parameters

- **data** (*ndarray*) – Data tensor to calibrate
- **mean** (*iterable float, optional*) – The target annual mean vector
- **sd** (*iterable float, optional*) – The target annual standard deviation (volatility) vector
- **time_unit** (*int or str*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, you could put in a string name of the time_unit. Accepted values are weekly, monthly, quarterly, semi-annually and yearly
- **inplace** (*bool*) – If True, calibration will modify the original data. Otherwise, a deep copy of the original data will be made before calibration. Deep copy can be time consuming if data is big.
- **tol** (*float*) – Tolerance used to determine if calibrate should be called. For example, if the cube's target annualized mean is similar to the actual tolerance, function will skip the mean adjustment.

Returns An instance of the adjusted numpy tensor

Return type ndarray

Truncate Outliers

muarch.calibrate.**truncate_outliers**(*data: numpy.ndarray*, ***, *bounds: Optional[List[Tuple[float, float]]] = None*, *sd=0*, *replacement='mean'*, *inplace=False*)

Truncates outliers by replacing it with the mean, median or a specified value.

Outlier is determined by the number of standard deviations past the mean within the asset group.

Parameters

- **data** (*ndarray*) – The tensor (data cube) where the axis represents time, trials and number of asset classes respectively
- **bounds** (*List of numbers*) – A list containing the lower and upper bound for each asset class. If specified, this takes precedence over the `sd` parameter. If `sd` is set to 0 and bounds are not specified, no changes will be made
- **sd** (*float*) – The number of standard deviations to consider a point an outlier. If `sd` is set to 0 and bounds are not specified, no changes will be made
- **replacement** (*{float, 'mean', 'median'}*) – The value to replace outliers with. Valid values are 'mean', 'median' or a number.
- **inplace** (*bool*) – If True, calibration will modify the original data. Otherwise, a deep copy of the original data will be made before calibration. Deep copy can be time consuming if data is big.

Returns A data cube with the outliers replaced

Return type ndarray

Basic Statistics

muarch.funcs.moments.**get_annualized_mean**(*data: numpy.ndarray*, *time_unit: Union[int, str] = 'monthly'*) → Union[float, numpy.ndarray]

Gets the annualized mean for each asset class in the data cube

Parameters

- **data** – Data matrix or tensor. The axis must represent time, trials and assets respectively where the assets axis is valid only if the data is a tensor.
- **time_unit** (*int or str*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, you could put in a string name of the time_unit. Accepted values are weekly, monthly, quarterly, semi-annually and yearly

Returns The annualized mean for the asset class or an array containing the annualized mean for each asset class.

Return type float or ndarray

muarch.funcs.moments.**get_annualized_sd**(*data: numpy.ndarray*, *time_unit: Union[int, str] = 'monthly'*) → Union[float, numpy.ndarray]

Gets the annualized standard deviation for each asset class in the data cube

Parameters

- **data** – Data matrix or tensor. The axis must represent time, trials and assets respectively where the assets axis is valid only if the data is a tensor.
- **time_unit** (*int or str*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, you could put in a string name of the time_unit. Accepted values are weekly, monthly, quarterly, semi-annually and yearly

Returns The annualized standard deviation (volatility) for the asset class or an array containing the annualized standard deviation for each asset class.

Return type float ndarray

muarch.funcs.moments.**get_annualized_skew**(*data: numpy.ndarray*, *time_unit: Union[int, str] = 'monthly'*) → Union[float, numpy.ndarray]

Gets the annualized skew for each asset class in the data cube

Parameters

- **data** – Data matrix or tensor. The axis must represent time, trials and assets respectively where the assets axis is valid only if the data is a tensor.
- **time_unit** (*int or str*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, you could put in a string name of the time_unit. Accepted values are weekly, monthly, quarterly, semi-annually and yearly

Returns The annualized skew for the asset class or an array containing the annualized skew for each asset class.

Return type float or ndarray

muarch.funcs.moments.**get_annualized_kurtosis**(*data: numpy.ndarray*, *time_unit: Union[int, str] = 'monthly'*) → Union[float, numpy.ndarray]

Gets the annualized kurtosis for each asset class in the data cube

Parameters

- **data** – Data matrix or tensor. The axis must represent time, trials and assets respectively where the assets axis is valid only if the data is a tensor.
- **time_unit** (*int or str*) – Specifies how many units (first axis) is required to represent a year. For example, if each time period represents a month, set this to 12. If quarterly, set to 4. Defaults to 12 which means 1 period represents a month. Alternatively, you could put in a string name of the time_unit. Accepted values are weekly, monthly, quarterly, semi-annually and yearly

Returns

The annualized kurtosis for the asset class or an array containing the annualized kurtosis for each asset class.

Return type ndarray

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

F

G

H

M

P

R

S

T

U